Miniball/IDS Autofill System Internals

Nigel Warr

20^{th} January 2014.

Contents

Chapter 1

Introduction

1.1 The hardware

The Miniball filling system hardware was designed by Robert Hide of the University of York. It uses a various custom-made units, which are interfaced to a PC by two ISA cards:

- A CIO-DIO96 to control the manifold valves and read the LN2 sensors.
- A CIO-DAS801 for the PT100 readout via an interface unit.

As ISA has been long obsolete, the possibility of replacing this system with a PCI-based system was considered. There was, at that time, a possible replacement for the CIO-DIO96 (the PCI-DIO96) but no suitable replacement for the other card. Later, USB was considered as another alternative and modules were bought to test.

In 2008, new software was written to use these USB modules instead of the ISA cards. This software uses:

- A USB-DIO96H/50 to control the manifold valves and read the LN2 sensors.
- A USB-TEMP for the PT100 readout without an interface unit.

This document describes both the original software for the ISA system and the modifications for the USB system. Most of the code is common to both.

The ISA interface card for the manifold valves (CIO-DIO96) and the LN2 sensors is normally at address 0x0300 and has four ports for each manifold each of which has eight bits. The first is for the sensors, the second for the bias shutdown, which we do not use since the Miniball detectors do not have a bias shutdown output. The third port is for the valve relays and the fourth for control. There are four manifolds.

The USB system needs a configuration file to associate the serial number of the USB-DIO96H/50 module (which is read out over USB) to the appropriate channels. Otherwise, this device is compatible with the ISA card.

For the ISOLDE Decay Station, it was decided to clone this system.

1.1.1 The LN2 sensors

The first port for the LN2 sensors is read only. Each bit corresponds to one of the LN2 sensors on the outlet side of the manifold. There is a potentiometer for each sensor on the control box to set the threshold, so the only information is a single bit per sensor to say whether the temperature of the sensor is above or below the threshold (i.e. LN2 present or not). The first four bits are used to correspond to the four valves. The next two bits are not used for Miniball, but they are for the fifth and sixth valve. The next bit is for the purge sensor, which detects when LN2 is at the purge outlet (i.e. the purge cycle is complete). Finally the last bit is set when the key on the control box is in the manual position. This is used by the software to veto any software control when the key is set to manual, but note this is implemented by the software, not the hardware.

1.1.2 The bias shutdown

Miniball does not use the bias shutdown at all. This was a feature that was put in for exogam (for which the York system was originally designed).

1.1.3 The valve relays

For each valve there is a relay to open the valve and this is controlled by the third port. The first four bits correspond to the four valves used for the detectors. The next two bits are not used for Miniball but are for the fifth and sixth valve. The next bit is for the purge outlet valve and the final bit is for the inlet valve.

The port is read-write for the ISA card. Setting a bit causes the hardware to open the corresponding valve and resetting it closes the valve. The port can be read to find out which valves are open. It does not seem to be possible to read with the USB module. In any case, the values read are most probably only an indication of what the user has requested and not what is actually happening at the level of the valve control unit.

To initiate a purge cycle, the bits corresponding to the purge and inlet valves are opened, so that LN2 can go from the LN2 vessel into the manifold and out through the purge outlet. When LN2 is detected on the LN2 sensor for the purge outlet, the purge cycle is complete.

Once the purge has completed, the inlet is left open, the purge is closed and the valves for the detectors are opened. Once LN2 is detected at the outlet for

each detector, they are closed one by one until none are left and then the inlet is also closed.

Note that for the USB module, although the ports for the sensors and bias shutdown are 8 bits wide, the relays use two 4-bit ports.

1.1.4 The control port (ISA card only)

The control port sets certain characteristics of the way the card works. Bit 0 is for the first four relays and bit 2 for the next four and these bits need to be set to make those ports input ports, but we want an output port, so those bits must not be set. Bits 6 and 7 set the direction. We write 0x90 to that port before we do anything, which is correct for Miniball operation.

Note that this information is gleaned from playing with the system and does not appear to be documented.

1.1.5 USB configuration (USB only)

The USB module is configured by sending special packets over USB. The MCC USB library has functions to perform the settings. As for the ISA system, we set the ports for the relays to write and the others to read mode.

1.1.6 The PT100 readout (ISA only)

The second card is more complicated than the first. It has 4 ports, the meaning of which seems to be context dependent. The interface has several ADCs and several multiplexers. It can read a total of 16 channels (i.e. 16 PT100s), but only one at a time. You have to select the ADC and multiplexer and then acquire a sample. We acquire a few samples and take an average. Note that this card is fairly slow, so we have to give it plenty of time to do its job.

1.1.7 The PT100 readout (USB only)

The ISA system uses a digital I/O channel to control a multiplexer and then an ADC to sample the output of that multiplexer. This is because the CIO-DAS801 is an ADC card with a few of digital outputs. The USB version is much simpler than the ISA one and uses a single module which measures the temperatures directly. It is accessed via the MCC USB library.

1.2 The software

The two halves of the system (control of filling and sensors on the one hand and reading the PT100s on the other) are naturally separate as they run on different cards and perform different tasks. Consequently, the software has been designed

in two separate pieces:

- A filling program which performs a single fill cycle on one or more manifold, including the purge cycle and the fill cycle, operating the manifolds and monitoring the LN2 sensors. It calls a script whenever it completes, whether successful or not and a second script if it fails. These scripts can be used to send out e-mails, turn off high voltage etc.
- A PT100 readout program which reads all the detector temperatures on all channels and writes them to files. It calls one script if the temperature goes above a certain threshold and another if the rate of increase in the temperature exceeds a certain amount. These scripts can be used to trigger emergency fills, send e-mails or turn off the high voltage.

There are two versions of the fill program *fill_isa* and *fill_usb* and two versions of the read PT100 program *read_pt100_isa* and *read_pt100_usb*. Two symbolic links *fill* and *read_pt100* are used to select which of the two versions is actually called.

The readout program can be called every five or ten minutes from a cron job and the filling program can be called either manually from the command line, or automatically from a cron job or from some special script.

The software has been designed so that both programs perform a task and then exit, so that no code is left running continuously. The original software written by the University of Liverpool for this hardware did things differently. It had a program which ran continuously, but it had a memory leak and after running for some time would run out of memory and then lock up the computer. Consequently, the software described in this document uses a different design, so that the programs exit between fills and PT100 readouts, and it is the operating system which is responsible for reclaiming memory etc., something which linux does rather well.

1.3 Naming conventions

The manifolds are given letters A to D. The detector outlets are given numbers from 1 to 4. So “A2” corresponds to outlet 2 on manifold A.

Chapter 2

The fill program

The filling program is called "fill" and is in *fill.c*. It makes use of the code in *logmessage.c* to log messages to standard output and to the system logger, the code in *lock.c* to lock devices so that two fill programs running at the same time cannot access the same device and the code in *disabledlist.c* to determine whether or not the fill has been disabled for a channel. The actual interface with the hardware is done with either *hardware_isa.c* or *hardware_usb.c*. These two files provide the same functions for ISA and USB, respectively. We build two executables, one for ISA and one for USB.

Note that it is perfectly allowable for two fill programs to access different devices as there are separate ports for each device. Indeed, the filling program, as soon as it has processed the command line arguments, forks a separate subprocess for each device.

2.1 The file fill.c

This file contains the main code for the filling program. It parses the command line arguments and switches and forks a subprocess for each device with outlets to be filled. For each device, it performs purge, fill and venting cycles and then waits for LN2 to stop flowing at the sensors before exiting. Signals are trapped and logged. On exit, errors are logged and scripts are called, which can take actions according to the success or failure of the fill.

2.1.1 The main routine

int main(int argc, char **argv);

The *main* routine (in *fill.c* first processes the command line arguments. Switches may be used to set the fill type (see section 2.1.12) or the minimum and maximum purge times, fill times and the minimum LN2 detection time. The remaining arguments indicate which outlets to fill. See section 1.3 for the naming

convention for the outlets.

For example the command:

```
fill A1 C2
```

will fill the first outlet on the first device and the second outlet on the third device.

It calls the function *read_disabled_list* in *disabledlist.c* (which in turn calls *analyse_disabled_list* - see section 2.2.1 and section 2.2.2) to get a list of disabled outlets from a file. These outlets will **never** be filled and if the user requests to fill one of these an error message will be written and that outlet will not be filled (though any valid outlets will be).

Then for each device which is to be filled *fork* (see the fork(2) man page) is called and in the child process the function *treat_device* (see section 2.1.2) is called. The parent process calls *wait4* (see the wait4(2) man page) to wait for its children to exit.

After that, the program drops its privileges using *seteuid* (see the seteuid(2) and getuid(2) man pages) and executes the completion script (see section 2.6).

2.1.2 The treat_device function

static int treat_device(int device);

The function *treat_device* (in *fill.c* is called once for each device, in a separate process running in parallel. We do not want to fill the devices one after another as they may have common LN2 lines, which need to be cooled down and the fill is, consequently, most efficient when everything is filled at once.

The function is called with a device parameter, which tells the function which device it has to fill.

First of all, we establish an exit handler using *atexit* (see the atexit(2) man page) which will be called when the process exits for whatever reason. This exit handler is responsible for logging and triggering certain scripts (see section 2.6).

After that we establish a signal handler (see section 2.1.7) for all signals which can be intercepted using *signal* (see the signal(2) man page), except for SIGCHLD, which we ignore. This ensures that if the program crashes with a segmentation violation, or is killed by the user or by a control C, the signal handler is called. The signal handler makes sure that all the relays are closed again and then exits (causing the exit handler to be called, which logs the event). It is safe to kill the filling program as the signal will be correctly handled. However, it is not safe to use "kill -9" because this will leave the valves open.

Next we lock the appropriate device using the function *lock* (see section 2.4.1), so that no other instance of the filling program can access the same de-

vice. If you issue two fill commands on the same device together, even if they are for different detector outlets the second one will wait for the first to complete before starting. This is unavoidable because they share common hardware. If you want to fill two detector outlets at the same time you should specify both when calling the fill command. On the other hand, there is no problem about filling two outlets on different devices with two fill commands, which will execute simultaneously, as this does pretty much the same thing the fill program does internally.

Then we call *logmessage* (see section 2.5.1) to write a message to the system logger and standard output.

After that we call *init* in either *hardware_isa.c* or *hardware_usb.c*, which initialises the device.

For ISA this means calling *ioperm* (see the ioperm(2) man page) to grant the program access permissions to the ports we need to use. Note that this means that the program must be run with root privileges. The program is written to be installed suid root and it will drop its privileges before calling scripts. This is the only way to access ISA ports without writing a kernel driver. Then it sets the bits of the control register, which determine which port is for input and which for output.

For USB, we first read a configuration file to find out which serial number is associated with the given device. This is the serial number printed on the front of the USB-DIO96H/50 module, which can also be read via USB. Then we scan the USB bus for USB-DIO96H/50 modules and when we find one, we read its serial number and compare with the one associated with this device. When we find it, we configure the direction of the ports.

Now we are ready to start, which we do by calling the function *purge* (see section 2.1.3). This function performs the purge cycle which involves opening the inlet valve and the purge outlet valve and waiting until LN2 is detected at the purge sensor.

Once the purge is complete, we call the *fill* (see section 2.1.4) function to perform the fill cycle. This consists of leaving the inlet valve open, closing the purge outlet valve and opening the individual detector outlet valves. Each detector outlet is closed when LN2 is detected at the corresponding sensor until all detector outlets have closed and then the inlet valve is also closed.

After the purge, we call *vent* (see section 2.1.5) to vent the manifold. We do this by closing the inlet and all the outlets and opening the purge valve. In this way, any gas in the manifold is released.

Finally, we call *wait_for_dry* (see section 2.1.6) which waits until the sensors on all the outputs, which we filled show gas. Then we return, which causes the exit handler to be called.

2.1.3 The purge function

static int purge(int device);

The *purge* function first records the time that purging started and then sets the program state to PROGRAM_STATE_PURGING (this state is given in logging information). Then it calls *get_sensor* (see section 2.3.2) to get the sensors. It doesn't use the value of the sensors, but relies on the side effect that *get_sensor* will exit with an error message if the key is turned to manual.

Next it uses *logmessage* to write a logging message and *set_relay* (see section 2.3.3) to open the inlet and purge outlet valves.

After that, it enters a loop which calls *usleep* (see the usleep(3) man page) to yield the CPU to avoid consuming too much CPU time. On each iteration, it uses *get_sensor* to get the value of the LN2 sensors checks for a timeout condition, and checks for LN2 detected at the purge outlet. If LN2 is detected, it breaks out of the loop and logs this with *logmessage* and returns. If a timeout occurs, it closes all the relays with *set_relay* and sets the program state to PROGRAM_STATE_PURGE_TIMEOUT and returns an error code, which causes *treat_device* to abort. Note that LN2 detected before the minimum purge time is simply ignored.

2.1.4 The fill function

static int fill(int device);

The *fill* function starts off by setting the program state to *PROGRAM_STATE_FILLING* and the mask to *INLET_VALVE*. Then it logs the start of the fill and records the time.

After that it enters a state machine which consists of an iteration loop, within which there is a loop over each channel and a switch over the state for each channel. After each loop on the channel, the valves are set as indicated by the *mask* variable. We call *usleep* on each iteration to avoid consuming too much CPU and *get_sensor* to see if LN2 has been detected anywhere.

The *mask* variable is used in a call to *set_relay* (see section 2.3.3), which opens and closes valves according to this mask.

The values of the state machine are:

```
OUTLET_STATE_PURGE
OUTLET_STATE_WAIT_LN2
OUTLET_STATE_GOT_LN2
OUTLET_STATE_IGNORE
OUTLET_STATE_DONE
OUTLET_STATE_FILLED
OUTLET_STATE_WAIT_ABORT
```

The *purge* function leaves all the outlets which are to be filled in the *OUTLET_STATE_PURGE* state and the rest in the *OUTLET_STATE_IGNORE* state.

OUTLET_STATE_PURGE

This is the first state we get to for outlets, which are to be filled.

In this state, we simply set the bit corresponding to that channel in the mask, which will be used to determine which valves to open, log that we are starting to fill that channel and change its state to *OUTLET_STATE_WAIT_LN2*.

OUTLET_STATE_WAIT_LN2

An outlet is in this state if filling has begun, but LN2 has not yet been detected.

In this state, we wait until LN2 is detected or the time limit is reached. In either case, we log what has happened and if LN2 was detected we change to *OUTLET_STATE_GOT_LN2* and if we timed out, we change to *OUTLET_STATE_WAIT_ABORT*.

OUTLET_STATE_GOT_LN2

An outlet is in this state if filling has begun and LN2 has been detected.

We wait to see if LN2 continues to flow for long enough. If not, we change back to *OUTLET_STATE_WAIT_LN2*. If LN2 flows for the required time, we change to *OUTLET_STATE_FILLED*.

OUTLET_STATE_FILLED

An outlet in this state has been successfully filled, but the valves are still open.

We remove the bit corresponding to that outlet from the mask, so that the valve will be closed afterwards. If this was the last bit corresponding to an outlet, we also remove the inlet bit. Then we change state to *OUTLET_STATE_DONE*.

OUTLET_STATE_IGNORE and OUTLET_STATE_DONE

These two states have the same behaviour. The former is the first and only state for an outlet that should not be filled. The latter is the last state for a channel that has already been successfully filled and the valve has been closed, or for a channel that has encountered an error.

We return from the *fill* function here, breaking right out of the state machine.

OUTLET_STATE_WAIT_ABORT

An outlet in this state has encountered some error condition.

We reset the bit corresponding to that outlet in the mask variable and if it was the last outlet, we also reset the inlet bit, so that these valves will be closed later. Then we change to *OUTLET_STATE_DONE*.

2.1.5 The vent function

static int vent(int device);

This function closes the inlet and outlet valves but opens the purge. This vents the device. It is called after filling is complete to release the pressure in the manifold. Note that it is even called if the purge failed, because we might still have filled the manifold with LN2 even if the purge sensor didn't detect any.

2.1.6 The wait_for_dry function

static int wait_for_dry(int device);

This function waits until the sensor corresponding to each outlet, which was filled indicates gas rather than liquid. The fill is not deemed to be complete until liquid stops flowing out of the detectors.

2.1.7 The signal_handler function

static void signal_handler(int signum);

This function is installed as a signal handler for all interceptable signals except SIGCHILD (which is ignored). It logs the event with *logmessage* and then exits, causing the exit handler to be invoked. It is the exit handler, which will then close all the valves and invoke the failure script if one of the outlets has failed.

2.1.8 The exit_handler function

static void exit_handler();

This function is installed as the exit handler which gets called whenever the program exits for whatever reason (good or bad).

First of all it calls *terminate* (see section 2.3.4) to close all the valves.

Then it adds a line to the log file for each channel that we tried to fill by calling *add_to_log* (see section 2.1.9).

Next it checks each outlet and counts the number of failures. If there were none, that is all and the function returns. However, if there were failures, it builds up the command to execute for the failure script.

Before this script is called, it uses *seteuid* (see the seteuid(2) and getuid(2) man pages) to drop any privileges it may have gained by being installed suid root. This is for security reasons, so that any scripts executed after (see section 2.6) are executed by the calling user not root (unless root is the calling user, of course).

Then it calls the script without privileges, passing the fill type as the first parameter (either "MANUAL", "AUTOMATIC" or "EMERGENCY") and then the list of outlets which failed. The script can then use this parameter to decide how critical the failure was and take action accordingly. Generally, it is assumed that when a manual fill fails, the person alone, who issued it is responsible for taking the appropriate action. For an automatic fill, we would normally send out an SMS to alert someone and if an emergency fill fails, we should probably also shut down the high voltage. However, the precise action is a policy decision to be made in the script based on the parameters passed to it.

2.1.9 The add_to_log function

static int add_to_log(int outlet);

This function generates a line of logging text for the outlet, indicating the program state (see section 2.1.11), the fill state (see section 2.1.10) as well as fill and purge times.

It then prepends this line to the appropriate log file (based on the outlet name) using *prepend_to_file* (see section 2.1.9).

The prepend_to_file function

static int prepend_to_file(char *filename, char *line);

This function prepends the line of text to the file specified by the filename, truncating to a maximum number of lines.

2.1.10 The filling states

The following outlet fill states are defined in *fill.h*

- OUTLET_STATE_PURGE - this outlet is being purged.
- OUTLET_STATE_WAIT_LN2 - we are filling and waiting for LN2 to appear at the outlet sensor.

- OUTLET_STATE_GOT_LN2 - we have LN2 already at the outlet sensor having already filled but we need to make sure it isn't just a short spurt, but that it flows for the prescribed amount of time. If it doesn't we switch back to the previous state.
- OUTLET_STATE_FILLED - we have had LN2 for the prescribed amount of time, so we are sure that the detector is full.
- OUTLET_STATE_WAIT_ABORT - this state means that we've been told to give up, for example because of a timeout.
- OUTLET_STATE_VENT - this outlet is being vented.
- OUTLET_STATE_DRY - we have already filled, and closed the valve but LN2 is still flowing, so we have to wait for it to dry up.
- OUTLET_STATE_DONE - we have successfully completed everything on that outlet, but we may need to wait for other outlets.
- OUTLET_STATE_IGNORE - this outlet doesn't need filling at all. This is the default state for outlets not specified on the command line. As far as the program is concerned it is the same as OUTLET_STATE_DONE (i.e. one which we don't have to fill at all is the same as one which has already been filled).

2.1.11 The program states

The following program states are defined in *fill.h*:

- PROGRAM_STATE_INITIALIZING - the program is starting up.
- PROGRAM_STATE_KILLED - the program has been killed and the signal handler invoked.
- PROGRAM_STATE_PURGING - the program is purging the device.
- PROGRAM_STATE_FILLING - the program is filling the detectors.
- PROGRAM_STATE_SUCCESS - the program has completed its job successfully.
- PROGRAM_STATE_HARDWARE_ERR - this occurs if the hardware doesn't respond correctly.
- PROGRAM_STATE_KEY_ERR - this occurs if the key is set to manual
- PROGRAM_STATE_PURGE_TIMEOUT - a timeout occurred during the purge cycle.
- PROGRAM_STATE_OUTLET_ERR - one or more of the outlets failed to fill correctly. The outlet state should give more information.
- PROGRAM_STATE_VENTING - the program is venting the device after filling.
- PROGRAM_STATE_DRYING - the program is waiting for all the outlets which were filled to stop showing liquid at the sensor.

2.1.12 The fill types

There are three types of fill which are set by specifying switches on the command line:

- Automatic fill. This is performed by using "fill –auto". This should only be used from a cron job filling automatically at a scheduled time.
- Manual fill. This is performed using "fill" without any switches and should be used when a person types the command from the command line.
- Emergency fill. This is performed by using "fill –emergency". This should be used from scripts which have detected some problem and need to initiate a fill at an unscheduled time.

The actual fill is performed in the same manner for all three types of fill, but the type of fill is logged in the log files and should the fill fail, it is possible to take different actions according to the type of fill. For example, if the fill was a manual one, it is not necessary to start sending out panic messages, since it means somebody must have been there to type the command in the first place. If an auto fill fails, attention is needed, but it is not a disaster. However, if an emergency fill fails, it is almost certainly necessary to shutdown the high voltage etc.

2.2 The file disabledlist.c

This file has the code to handle the disabled list. The user can enter here the channels, which should not be filled under any circumstances.

2.2.1 The read_disabled_list function

int read_disabled_list(char *filename);

This function (in *disabledlist.c* parses the file specified by the filename parameter, looking for a line like:

```
disabledlist="A3 B2"
```

It extract the string after the equals sign and passes it to *analyse_disabled_list* (see section 2.2.2).

2.2.2 The analyse_disabled_list function

static int analyse_disabled_list(char *list);

This function parses the list of devices from *all_fill_disabled.sh* and marks the ones it finds in the array fill_disabled, which is used in the main routine to eliminate disabled outlets from the list of outlets to fill. Outlets in this list may not be filled even manually. This is different from the *emergency_fill_disabled.sh* which is only used by the scripts.

2.3 The files hardware_isa.c and hardware_usb.c

These two files implement identical functions, which perform the same tasks, but using different hardware. They contain all the hardware-specific code for ISA and USB, respectively.

2.3.1 The init function

int init(int device);

The *init* function merely writes 0x90 to the control port for the appropriate device (as determined by the global variable *device*). This sets the port directions correctly.

2.3.2 The get_sensor function

int get_sensor(int device);

This function merely uses *inb* (see the inb(2) man page) to read the sensor state from the port. It also checks the key sensor and if that indicates the key is set to manual, it sets the program state to PROGRAM_KEY_ERROR, closes the valves and exits. Otherwise it returns the bit mask read from the port.

2.3.3 The set_relay function

int set_relay(int device, int mask);

This function uses *outb* (see the outb(2) man page) to write to the port controlling the relays for the given device. It then uses *inb* to read back the result to make sure it worked. If it didn't it sets the program state to PROGRAM_STATE_HARDWARE_ERR and returns an error code. The mask parameter contains the bits to set.

2.3.4 The terminate function

int terminate(int device);

This function closes all the valves for the device. It is used to shut down the device.

2.3.5 The set_status function

static int set_status(int N, unsigned char val);

This function writes the value *val* to the N^{th} byte of the file defined as *STATUS_FILE* in *af_config.h*. It is called for each LN2 sensor and each relay (different values of N) and the bits in *val* correspond to the different sensors and relays. This file is read by the *show_status* program (see section 5).

2.3.6 The get_sn_for_device function (USB only)

static int get_sn_for_device(char *filename, int device);

This function is only present in the USB version of the code. It opens the configuration file, and finds the serial number of the USB-TEMP module, which the user has configured to handle a particular device. This value is returned.

2.4 The file lock.c

This file contains the code for locking devices. It uses semaphores to provide *lock* and *unlock* functions. If two processes try to lock with the same semaphore, the first one succeeds and the second one hangs until the first process relinquishes the lock (either by explicitly unlocking or by exiting).

2.4.1 The lock function

int lock(int sem_num);

Locking is performed using Sys V semaphores (see the semget(2), semctl(2) and semop(2) man pages for details). The code is in the file *lock.c*.

A value of zero for sem_num is used to lock the PT100s and values of 1 to 4 correspond to devices A to D.

This function attempts to use a set of semaphores, creating them if necessary using *semget* and if it created them, initializes them to the "unlocked" state using *semctl*. Then it uses *semop* to lock the one specified. The *SEM_UNDO* flag is set, which means that should the process exit, the operating system will automatically unlock the semaphore. Alternatively, we can call the *unlock* function (see section 2.4.2) to relinquish the lock.

If another process already holds the lock, the call to *semop* will hang until that process releases the lock.

2.4.2 The unlock function

int unlock(int sem_num);

This function relinquishes a lock taken out by *lock*, but it isn't actually called.

Instead we rely on the kernel to automatically release the lock, when the program exits.

2.5 The file logmessage.c

This file implements the message logging. Log messages are written both to standard output and to the system logger.

2.5.1 The logmessage function

int logmessage(char *fmt, ...);

This function is in *logmessage.c* and is called in the same way as *printf* (see the printf(3) man page).

It writes the text into the system log file using the *openlog*, *syslog* and *closelog* functions (see the syslog(3) man page). This will end up in some file like */var/log/messages* where it will be rotated by the normal system log file rotation programs.

2.6 The scripts

The filling program has two scripts that it can execute:

- fill_complete_script.sh - called after every fill, whether successful or not. This can be used to send log files to people etc. For example, the default script executes the commands *show_last_fill* and *show_pt100* and mails the output to the *af_info* mailing list. The script is called with the list of outlets which were filled as arguments.
- fill_fail_script.sh - called when a fill fails. The script is called with the fill type (either "MANUAL", "AUTOMATIC" or "EMERGENCY") as the first parameter and the list of outlets for which the fill failed as the remaining arguments. In this way, the script can react differently to a failed manual fill (where we suppose the user is present), an automatic fill (where we should warn the user, and hope that (s)he reacts before it becomes critical) and an emergency fill (where as we were already reacting to an emergency, it is essential to shutdown the high voltage).

Note that if *fill* is called for outlets on more than one device, the completion script is only called once for all the outlets on all the devices, but the failure script is called once for each device where the fill failed with just the outlets on that device which failed. This is a consequence of the structure of the program. The only inconvenience is that if all the devices fail (e.g. you have two devices on one LN2 vessel which has run dry) you get a message for each device, rather

than one for the whole system.

The default script sends an e-mail to the *af_info* list only, if it was a manual fill. If it was an automatic fill, it sends an e-mail to the *af_sms* list (which can be forwarded through an e-mail to SMS gateway), and another to the *af_emerg* list. If an emergency fails it does the same as an automatic fill failure and also turns off the high voltage.

Chapter 3

The read_pt100 program

The second most important program of the filling system is the *read_pt100* program in *read_pt100.c*. It also uses code in *lock.c* and *logmessage.c* (see sections 2.4.1 and 2.5.1).

It reads all the ADCs for the PT100 readout and converts the results into temperatures in Kelvin.

The program uses the files *pt100.c*, *temperature_log.c*, *logmessage.c* and *lock.c* for both USB and ISA versions. The only difference between the two versions is the main routine, which is in *read_pt100.c* for the ISA version and *read_usb_temp.c* for the USB version.

Note that the ISA version must be installed with privileges in order to access the ports, but the USB version does not need this, as long as the udev rule is set up to grant users access to this kind of USB device (e.g. in /etc/udev/rules.d/77-mccusb.rules):

```
KERNEL=="hiddev*", NAME="hiddev%n", MODE="0666"
```

3.1 The file read_pt100.c (ISA only)

3.1.1 The main routine

int main(int argc, char **argv);

First of all it calls *pt100_read_calibration* to read in the calibrations for each PT100 (see section 3.3.1).

Then it calls *lock* (see section 2.4.1) to lock the semaphore corresponding to the PT100 readout (semaphore zero).

After that it calls *ioperm* (see the ioperm(2) man page) to grant the program access permissions to the ports we need to use. Note that this means that the program must be run with root privileges. The program is written to be

installed suid root and it will drop its privileges before calling scripts. This is the only way to access ISA ports without writing a kernel driver.

After that, the program drops its privileges using *seteuid* (see the seteuid(2) and getuid(2) man pages) and executes the completion script (see section 3.4.2).

Then it performs input and output operations on the card using *inb* and *outb* (see the inb(2) and outb(2) man pages) to read each of the 16 ADCs in turn (corresponding to the four outlets on the four devices). The value is converted to Kelvin and the command *temperature_log* (see section 3.4.1) is called passing the number of the ADC and the temperature of the PT100 in Kelvin to log the value.

The *temperature_log* function also checks if the temperature has exceeded a threshold or if the rate of increase in the temperature exceeds another threshold and triggers the execution of scripts in those cases (see section 3.4.2).

3.2 The file read_usb_temp.c (USB only)

3.2.1 The main routine

int main(int argc, char **argv);

First of all it calls *pt100_read_calibration* to read in the calibrations for each PT100 (see section 3.3.1).

Then it calls *lock* (see section 2.4.1) to lock the semaphore corresponding to the PT100 readout (semaphore zero).

Next, it searches the USB bus for USB-TEMP devices using the *PMD_Find* function of the MCC USB library. This opens a file descriptor to each such device and returns the number of devices.

For each device, we configure the port and each of the sensors. We use the RTD two-sensor, two-wire mode, which gives us 8 PT100s per module.

Then we perform a temperature scan for each device and covert each resistance, which was read from Ohms to Kelvin and the command *temperature_log* (see section 3.4.1) is called passing the number of the channel and the temperature of the PT100 in Kelvin to log the value.

The *temperature_log* function also checks if the temperature has exceeded a threshold or if the rate of increase in the temperature exceeds another threshold and triggers the execution of scripts in those cases (see section 3.4.2).

3.3 The file pt100.c

This file contains the code to calibrate the PT100s from Ohm to Kelvin. There are two functions, one to read the calibration from a file or take the default values and the other to convert for a given channel.

3.3.1 The pt100_read_calibration function

int pt100_read_calibration();

The *pt100_read_calibration* function for the *read_pt100* program reads the file defined as PT100_CAL_FILE in *af_config.h* which has the format "channel offset gain" where channel is something like A1, B2 etc. and the offset is typically something like 25.8944 and the gain 2.45849. This allows the user to adjust the calibration of the PT100s individually, so that a single set of global thresholds can be used by the *warm_script.sh* script.

This conversion is from a resistance in Ohm to a temperature in Kelvin. It does not matter too much if it is accurate over a large range, but it should be accurate in the range around liquid nitrogen temperatures. If we ignore additional resistance due to the cables, we expect the default values, which are obtained from a linear fit in the useful regime. However, sometimes wiring issues lead to different coefficients. The best thing is to measure the resistance with a 50 Ω terminator and with two such terminators in parallel (i.e. 25 Ω) and use that to calibrate.

3.3.2 The pt100_ohm_to_kelvin function

int pt100_ohm_to_kelvin(int channel, float resistance);

The *pt100_ohm_to_kelvin* function takes a resistance in Ohm for a given channel and returns the temperature in Kelvin using the calibration read in by *pt100_read_calibration* (see section 3.3.1).

3.4 The file temperature_log.c

This file contains the code for logging the temperatures. A limit is imposed on the maximum number of lines the log file can have, with the oldest entries being removed to make way for new ones.

Furthermore, this code checks the temperature to look for rises in the temperature or warm detectors and calls scripts appropriately.

3.4.1 The temperature_log function

int temperature_log(int num, float temp);

The *temperature_log* function for the *read_pt100* program is similar to the *add_to_log* function for the *fill* program (see section 2.1.9) in that it prepends a line of text to a log file. However, it performs two important additional service in that it checks both the absolute temperature and the relative rise in temperature since the last log entry to see if the detector is either warm or warming up.

If the temperature is rising by more than RISE_THRESH degrees per minute (defined in *af_config.h*), the rise script is called (see section 3.4.2).

If the temperature goes above WARM_THRESH (defined in *af_config.h*), the warm script (see section 3.4.2) is called.

3.4.2 The scripts

There are two scripts which may be called by the *read_pt100* program:

- warm_script.sh - called if the temperature goes above the WARM_THRESH defined in *af_config.h*. It is called with the outlet as the first parameter and the temperature as the second. The script can decide if it wants to try and issue an emergency fill, turn off the high voltage, send an e-mail or an SMS or whatever.
- rise_script.sh - called if the temperature rises faster than RISE_THRESH defined in *af_config.h* per minute. The script is called with the outlet as the first parameter, the old temperature as the second, the current temperature as the third and the time between the two temperature measurements as the fourth parameter. It probably only makes sense to react if the temperature has risen sharply, which probably means the detector is dry, but has not yet warmed up to the thresholds used by the warm script.

Chapter 4

The show_pt100 program

This program reads the log files generated by *read_pt100* (see chapter 3) and writes out the current temperature for each detector.

Note that this program doesn't interface directly to the hardware, but only reads the log files, so it is hardware independent.

4.1 The main routine

int main(int argc, char **argv);

This program accepts an option "-h" which causes the output to be in HTML for inclusion in a web page.

After checking the switches, it writes a heading and then for each device and each outlet, it generates a filename and calls *treat_file* (see section 4.2) to handle it. Finally it writes a trailer if it is in HTML mode.

4.2 The treat_file function

int treat_file(char *filename, char device, int outlet);

This function opens the file specified by filename, reads the temperature from the first line (the data are prepended by *read_pt100* (see chapter 3) so we only need the first line). Then it writes the output.

Chapter 5

The show_status program

While the fill program is filling (see section 2.3.5), it writes to a binary status file the status of all the valves and LN2 sensors. The file name is defined as *STATUS_FILE* in *af_config.h*. The *show_status* program reads this file and writes the information in human readable form.

Note that this program doesn't interface directly to the hardware, but only reads the log files, so it is hardware independent.

Chapter 6

The show_last_fill program

This program reads the log files generated by *fill* (see chapter 2) and writes out the last fill times and status of each outlet.

Note that this program doesn't interface directly to the hardware, but only reads the log files, so it is hardware independent.

6.1 The main routine

int main(int argc, char **argv);

This program accepts an option "-h" which causes the output to be in HTML for inclusion in a web page.

After checking the switches, it writes a heading and then for each device and each outlet, it generates a filename and calls *treat_file* (see section 6.2) to handle it. Finally it writes a trailer if it is in HTML mode.

6.2 The treat_file function

int treat_file(char *filename, char device, int outlet);

This function opens the file specified by filename, reads the first line (the data are prepended by *fill* (see chapter 2) so we only need the first line). Then it writes the output.

Chapter 7

The show_recent_fills program

This program reads the log files generated by *fill* (see chapter 2) and writes out the times of the fills within the last 24 hours.

Note that this program doesn't interface directly to the hardware, but only reads the log files, so it is hardware independent.

7.1 The main routine

int main(int argc, char **argv);

This program accepts an option "-h" which causes the output to be in HTML for inclusion in a web page.

After checking the switches, it writes a heading and then for each device and each outlet, it generates the channel name and calls *treat_channel* (see section 7.2) to handle it. Finally it writes a trailer if it is in HTML mode.

7.2 The treat_channel function

int treat_channel(char *name);

This function opens the file corresponding to the channel name, reads the lines parsing the times and remembering them. It stops doing this when it reaches a time older than 48 hours plus 5 minutes.

Then it sorts the times chronologically (they are in inverse chronological order in the file) by calling the system function *qsort* and using the *qsort_helper* function to sort by time.

7.3 The qsort_helper function

int qsort_helper(const void *a, const void *b);

This is a helper function for *qsort* to sort an array of *time_t* values chronologically.

Chapter 8

The generate_html script

This script generates an html page containing the output from *show_pt100*, *show_last_fill* and *show_recent_fills* which it calls and the temperature and duration plots produced by *generate_temp_plot* and *generate_duration_plot*. See chapters 4, 6, 7, 9 and 10.

Chapter 9

The generate_temp_plot program

This program generates a plot of the temperatures as a function of time, using *gnuplot* (see the gnuplot(1) man page). It uses the information in the log files generated by *read_pt100* (see chapter 3).

9.1 The main routine

int main(int argc, char **argv);

For each device and for each outlet, this routine calls *check_file* (see section 9.2) which returns 0 if there is no useful data (i.e. within the last NDAYS days) or 1 if there is data to plot. This data is read from the files produced by *read_pt100* (see chapter 3).

Then it uses *popen* (see the popen(3) man page) to start a process running *gnuplot* (see the gnuplot(1) man page) if it is the first time, and calls *write_gnuplot_header* (see section 9.3) to write the header for *gnuplot*. Then it writes a command for *gnuplot*. All of this causes *gnuplot* to generate the plot.

9.2 The check_file function

int check_file(char *filename);

This function reads the file specified by the filename and checks if it has any data in the last NDAYS days. If it does, it returns 1, otherwise it returns 0.

9.3 The write_gnuplot_header function

int write_gnuplot_header(FILE *fp);

This function writes the a sequence of *gnuplot* commands to setup the plot to the output stream, which is normally a pipe to *gnuplot*.

Chapter 10

The generate_duration_plot program

This program generates a plot of the fill durations as a function of time, using *gnuplot* (see the gnuplot(1) man page). It uses the information in the log files generated by *fill* (see chapter 2).

10.1 The main routine

int main(int argc, char **argv);

For each device and for each outlet, this routine calls *check_file* (see section 10.2) which returns 0 if there is no useful data (i.e. within the last NDAYS days) or 1 if there is data to plot. This data is read from the files produced by *fill* (see chapter 2).

Then it uses *popen* (see the popen(3) man page) to start a process running *gnuplot* (see the gnuplot(1) man page) if it is the first time, and calls *write_gnuplot_header* (see section 10.3) to write the header for *gnuplot*. Then it writes a command for *gnuplot*. All of this causes *gnuplot* to generate the plot.

10.2 The check_file function

int check_file(char *filename);

This function reads the file specified by the filename and checks if it has any data in the last NDAYS days. If it does, it returns 1, otherwise it returns 0.

10.3 The write_gnuplot_header function

int write_gnuplot_header(FILE *fp);

This function writes the a sequence of *gnuplot* commands to setup the plot to the output stream, which is normally a pipe to *gnuplot*.

Chapter 11

The release_pressure program

Sometimes it is necessary to release the pressure on the device. This can either be done by switching the key to manual and opening the purge valve for a second and then switching back, or by using the *release_purge* program, which does the same thing.

This program takes the device letter as a parameter and uses functions similar to the ones in the fill program (see chapter 2).

Chapter 12

The scripts

12.1 The fill_complete_script.sh script

This script is called whenever a fill has completed regardless of whether it succeeded or not. It simply calls *show_pt100* (see chapter 4) and *show_last_fill* (see chapter 6 and pipes the results into *mail* (see the mail(1) man page) to send it to the *af_info* list (see chapter 13).

12.2 The fill_fail_script.sh script

This script gets called when a fill actually fails. It is called so that the first parameter is either "MANUAL", "AUTOMATIC" or "EMERGENCY" indicating the type of fill. It is assumed that a failure on an emergency fill is the most serious as it implies a detector is already warming up and the fill failed. A failure on a manual fill is assumed to be the least serious, since it is supposed that the person who initiated the manual fill is there to do something about it. For this reason, a failure on an emergency fill triggers a ramping down of the high voltage and a message to the *af_emerg* list (see chapter 13). A failure on an automatic fill also triggers a message to *af_emerg*, but no HV shutdown. A failure of a manual fill, however, only sends a message to *af_info*.

12.3 The warm_script.sh script

This script is called if a detector warms up above the threshold specified in *af_config.h*. It implements some new thresholds which determine what it should do. Above 250 K it just ramps down the high voltage without bothering to try and fill again or sending out e-mails. Between 200 K and 250 K it ramps down the high voltage and sends out e-mails to the *af_emerg* list (see chapter 13), but doesn't bother trying to fill. Between 130 K and 200 K it sends e-mails to *af_emerg*, ramps down the voltage and tries to force fill the detector. Between 105 K and 130 K it sends e-mails to the *af_warn* list (not the *af_emerg* list this time, as it is not an emergency if the temperature goes back down) and tries to

force fill, but leaves the voltage on. Below 105 K it does nothing.

This script sources the *emergency_fill_disabled.sh* script which has a definition of those outlets which should not be filled by the scripts. This list does not effect the automatic or manual filling, just the scripts. It has the format:

```
#!/bin/bash
DISABLEDLIST="A1 B1 B2 B3 B4 D1 D2 D3 D4"
```

which would be appropriate if A2, A3, A4, C1, C2, C3 and C4 all had detectors attached.

12.4 The rise_script.sh script

This script is called if a rise in the temperature is detected. If the temperature exceeds 130 K, it does nothing as the *warm_script.sh* script (see section 12.3) will take care of it. Otherwise it sends e-mails to the *af_warn* list (see chapter 13) and tries to force a fill.

This script sources the *emergency_fill_disabled.sh* script which has a definition of those outlets which should not be filled by the scripts. This list does not effect the automatic or manual filling, just the scripts. It has the format:

```
#!/bin/bash
DISABLEDLIST="A1 B1 B2 B3 B4 D1 D2 D3 D4"
```

which would be appropriate if A2, A3, A4, C1, C2, C3 and C4 all had detectors attached.

Chapter 13

The mailing lists

The file *~/.mailrc* contains mail aliases which are used by the filling system:

- af_info - receives general informational messages about the system which are sent out every time it fills. This is quite a high traffic list and most of it is not important.
- af_warn - receives warning messages when something happens which is not serious, but could become a problem later. This should go to people who might be able to prevent it from becoming a problem preemptively. For example, if a detector warmed up so that it needed an emergency fill, but not enough to need to turn of the high voltage, the system may be able to bring back down the temperature by emergency filling without human intervention, so a warning is sufficient. If somebody is around they should take a look.
- af_emerg - receives emergency messages when something happens that needs immediate intervention. This should go to as many people as possible in the hope that one of them can do something.
- af_sms - receives an emergency message if an automatic or emergency fill fails. This is likely to be a serious problem. This can be rerouted through an e-mail to SMS gateway, so that it gets sent to somebody's mobile telephone. Note that whenever the code sends to af_sms it also sends to af_emerg.

Chapter 14

The automation of the system

The automatic filling is achieved by calling the *fill* program from a *cron* job (see the cron(8) man page) at the desired fill times. The list of detectors to fill and the times to fill is kept in the crontab, which can be viewed using *"crontab -l"* (see the crontab(1) man page) from the autofill account and edited using *"crontab -e"*.

The format for crontab is shown in the crontab(5) man page. A typical line is:

```
00 04,10,16,22 * * * /usr/bin/fill --auto a1 a3 a4 c1 c2 c3 > /dev/null
```

which fills at 4 AM, 10 AM, 4 PM and 10 PM (the 04,10,16,22) on outlets a1, a3, a4, c1, c2, c4. The output is redirected to */dev/null* as it is only generally useful for debugging purposes and the same messages are also sent to the system logger which puts them in */var/log/messages*.

The reading of the PT100s is done every five minutes using the crontab entry:

```
0-59/5 * * * * /usr/bin/read_pt100
```

There are three further jobs which are used to generate a web page summarizing the current state of the filling system every ten minutes.

```
1-59/10 * * * * generate_html
2-59/10 * * * * generate_temp_plot
3-59/10 * * * * generate_duration_plot
```

These files are copied to another computer using another cron job running under a different username.

Chapter 15

The graphical user interface

The graphical user interface (*af_config*) provides an interface to these programs. It is written in Tcl/Tk. It also uses the programs *af_set_times* and *af_status*. The former is used to edit the crontab and the latter displays the status of the valves and LN2 sensors during the fill.

Chapter 16

Troubleshooting

If a fill command hangs and nothing happens, check if there is another fill command running. If so, do a "killall fill" and try again. If one fill program is still running for some reason, even if it is not doing anything, it keeps the semaphore locked, so no other fill can start.

If you get a HARDWARE_ERROR status, check all the connections to the manifolds and the manifold controller. This error occurs when the computer tells the manifold controller to set the valves to a particular state and when it asks them what state they are in, they are not in the desired state.

If LN2 flows out of a purge or detector outlet, but the program doesn't react to that, check that the LN2 detect light on the manifold controller box is reacting to the LN2. If it isn't you probably need to adjust the threshold using the screw (near the light on the controller box). They seem to drift with time, so if you see that fill times are getting longer and longer, check that it is detecting LN2 as soon as it is really flowing.

If you hear the click of the valve for a detector outlet open (it vibrates a little when it is open too) but no LN2 is flowing into the detector, the line is probably blocked. This can either be a kink of the line or ice in the line. In the former case, straighten the line and in the later, stop filling, take the line out of the detector, thaw it out, let the water run out, put it back and try again.

If the valves don't open and the LEDs on the manifold control box don't change when the program runs, check the connection between the computer and the manifold control box.

If the LEDs change from red to green, but the valves don't open, check the connection from the manifold control box to the manifolds.

If the system fills, but no e-mail is sent, check the syntax of the *.mailrc* file in the autofill home directory. Comments are not allowed and the mail program is a bit picky about the syntax. If that is OK, try sending mail from the system using the *mail* command by hand and see what happens.

If the temperature on a PT100 suddenly starts oscillating, I have no idea what that is, but it is a nuisance. The only thing I can suggest is to add that outlet to the disabled list for emergency fills and keep a close eye on it, because it won't have emergency filling to fall back on.

If a PT100 value suddenly appears in the list for a channel which has no PT100 connected, I don't know what that is either. Just make sure that outlet is in the disabled list.

If computer controlled filling doesn't work, try turning the key to manual and operating the switches.

If you get a PURGE_TIMEOUT condition and no LN2 comes out of the purge outlet, check there is LN2 in the vessel and the pressure is between 1.5 and 2 bar. If there is LN2 and no pressure, check that you don't have a leak.

If you get an error message "ERROR: fill disabled on A1" (or another outlet) this means that this outlet is disabled in the *all_fill_disabled.sh* file.

If you the LEDs on the manifold control boxes for the manifold are off (they should normally either be red (closed) or green (open) but never off) check that the manifolds have power and that the cables are connected. If they are OK, it might be one of the two 0.5 Amp fuses inside the manifold control box. Note that there are three fuses for this box and only one is accessible from the back. The other two are inside. If any LEDs are lit at all, the one at the back is OK. We have had ones inside the control box blow on a couple of occasions.