```
#

/*
 * ADC interface module
 *
 * This module provides the interface from the Reader to Comedi.  It
 * handles interaction with the Comedi device and mapping the Comedi
 * data buffer.
 *
 * The routines (apart from adc_new(), which returns a pointer to the
 * semi−opaque adc structure representing this object) in this module
 * return 0 on success and −1 on failure; they leave error information
 * in the adc structure from which it can be retrieved with the
 * adc_error() method.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
#include <comedi.h>
#include <comedilib.h>

#include <time.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "assert.h"
#include "strbuf.h"
#include "chunk.h"
#include "lut.h"
#include "mman.h"
#include "queue.h"
#include "adc.h"

/*
 * Private information for the ADC module.
 */

#define N_USBDUX_CHANS  16

#define MIN_SAMPLING_FREQUENCY  6e4     /* Minimum sampling frequency per channel [Hz] */
#define MAX_SAMPLING_FREQUENCY  3.75e5  /* Maximum sampling frequency per channel [Hz] */
#define MIN_COMEDI_BUF_SIZE     8       /* Minimum Comedi Buffer size [MiB] */
#define MAX_COMEDI_BUF_SIZE     256     /* Maximum Comedi Buffer size [MiB] */

struct _adc {
  const char *a_path;                   /* The path to the Comedi device (assumed permanent string) */
  comedi_t   *a_device;                 /* Comedi device handle */
  int         a_devflags;               /* Comedi device flags */
  int         a_fd;                     /* Device file descriptor */
  int         a_req_bufsz_mib;          /* Requested buffer size [MiB] */
  int         a_bufsz_bytes;            /* Size of the buffer in bytes */
  int         a_bufsz_samples;          /* Size of the buffer in samples */
  sampl_t    *a_comedi_ring;            /* Ring buffer for the device */
  double      a_totfrequency;           /* Total sampling frequency */
  int         a_intersample_ns;         /* Time between samples [ns] */
  int         a_range;                  /* Current conversion range */
  int         a_raw;                    /* Don’t convert the data, deliver it raw */
  convertfn   a_convert;                /* Current conversion function */
  comedi_cmd  a_command;                /* Comedi command descriptor structure */
  unsigned    a_chans[N_USBDUX_CHANS];  /* Channel descriptors for hardware channels */
  int         a_running;                /* True when an ADC data conversion is running */
  int         a_live;                   /* True when we have seen data, and a_start_time is set */
  uint64_t    a_start_time;             /* Time the current data conversion stream started */
  uint64_t    a_head_time;              /* Timestamp of latest buffer sample */
  uint64_t    a_head;                   /* Latest sample present in the ring buffer */
  uint64_t    a_tail;                   /* Earliest sample present in the ring buffer */
};

#define USBDUXFAST_COMEDI_500mV 1 /* Bit 3 control output is 0 iff the CR_RANGE is one */
#define USBDUXFAST_COMEDI_750mV 0 /* Bit 3 control output is 1 iff the CR_RANGE is zero */
```

```
/*
 * ADC is a singleton class, so we can get away with defining a single private structure.
 */

private struct _adc snapshot_adc;

/*
 * Error string function for strbuf module.
 */

private int comedi_error_set_up = 0;

private const char *comedi_error() {
  return comedi_strerror( comedi_errno() );
}

/*
 * Allocate and set up a new ADC descriptor.
 */

public adc adc_new(strbuf e) {
  adc ret = &snapshot_adc;

  if( !comedi_error_set_up++ ) {        /* Install the routine to interpolate %C strings */
    int ret = register_error_percent_handler(’C’, comedi_error);
    assertv(ret==0, "Failed to register handler for Comedi errors (%%C): %m\n");
  }
  ret−>a_fd = −1;
  return ret;
}

/*
 * Release ADC resources and free an ADC structure.
 */

public void adc_destroy(adc a) {
  adc_stop_data_transfer(a);

  if(a−>a_fd >= 0)
    close(a−>a_fd);

  if(a−>a_device)
    comedi_close(a−>a_device);

  if(a−>a_comedi_ring)
    munmap(a−>a_comedi_ring, a−>a_bufsz_bytes);

  /* Zero the structure −− back to initial state */
  bzero(a, sizeof(struct _adc));
}

/*
 * Set the device path
 */

public int adc_set_device(adc a, const char *device) {
  a−>a_path = device;
  return 0;
}

/*
 * Set the total capture sampling frequency from the per−channel frequency.
 */

public int adc_set_chan_frequency(adc a, strbuf e, double *freq) {
  double f = *freq;

  if(f < MIN_SAMPLING_FREQUENCY || f > MAX_SAMPLING_FREQUENCY) {
    strbuf_appendf(e, "Sampling frequency %g not within compiled−in ADC limits [%g,%g] Hz",
                   f, MIN_SAMPLING_FREQUENCY, MAX_SAMPLING_FREQUENCY);
    return −1;
  }

  int ns   = 1e9 / (f*NCHANNELS); /* Inter−sample period */
  int xtra = ns % 100;
  /* Adjust period for 30[MHz] USBDUXfast clock rate */
  ns = 100 * (ns / 100);
  if( xtra > 17 && xtra < 50 )
    ns += 33;
  if( xtra >= 50 && xtra < 83 )
    ns += 67;
  if( xtra >= 84 )
    ns += 100;
  a->a_intersample_ns = ns; /* Need a plausible value at all times for computing snapshot data */
  a->a_totfrequency = 1e9 / ns;
  *freq = a->a_totfrequency / NCHANNELS;
  return 0;
}

/*
 * Set the desired ring buffer size.
 */

public int adc_set_bufsz(adc a, strbuf e, int bufsz) {
  if(bufsz < MIN_COMEDI_BUF_SIZE || bufsz > MAX_COMEDI_BUF_SIZE) {
    strbuf_appendf(e, "Comedi buffer size %d MiB outwith compiled-in range [%d,%d] MiB",
                   bufsz, MIN_COMEDI_BUF_SIZE, MAX_COMEDI_BUF_SIZE);
    return -1;
  }
  a->a_req_bufsz_mib = bufsz;
  return 0;
}

/*
 * Set the desired ADC range
 */

public int adc_set_range(adc a, strbuf e, int range) {
  /* Set up the conversion function:  500mV or 750mV FSD */
  switch(range) {
  case 500:                     /* Narrow FSD range */
    a->a_convert = a->a_raw? convert_raw_raw : convert_raw_500mV;
    a->a_range = USBDUXFAST_COMEDI_500mV;
    break;

  case 750:                     /* Wide FSD range */
    a->a_convert = a->a_raw? convert_raw_raw : convert_raw_750mV;
    a->a_range = USBDUXFAST_COMEDI_750mV;
    break;

  default:
    strbuf_appendf(e, "Comedi range spec %d unknown", range);
    return -1;
  }
  return 0;
}

/*
 * Set ADC to raw mode, i.e. don’t range-map the incoming data.
 */

public void adc_set_raw_mode(adc a, int on) {
  a->a_raw = (on != 0);
  if(a->a_raw)
    a->a_convert = convert_raw_raw;
  else {
    if(a->a_range == USBDUXFAST_COMEDI_500mV)
      a->a_convert = convert_raw_500mV;
    if(a->a_range == USBDUXFAST_COMEDI_750mV)
      a->a_convert = convert_raw_750mV;
  }
}

/*
 * Initialise the ADC structure for data capture.
 */

public int adc_init(adc a, strbuf e) {
  int ret;
  int i;

  if( !a->a_path ) {
    strbuf_appendf(e, "Comedi device path not set");
    return -1;
  }

  /* Open the Comedi device */
  a->a_device = comedi_open(a->a_path);
  if(a->a_device == NULL) {
    strbuf_appendf(e, "Comedi device %s failure setting up ADC structure: %C", a->a_path);
    return -1;
  }
  a->a_fd = comedi_fileno(a->a_device);
  a->a_devflags = comedi_get_subdevice_flags(a->a_device, 0);

  /* Initialise Comedi streaming buffer */
  int request = a->a_req_bufsz_mib * 1024 * 1024;
  ret = comedi_get_buffer_size(a->a_device, 0);
  if( request > ret ) {
    ret = comedi_get_max_buffer_size(a->a_device, 0);
    if( request > ret ) {
      ret = comedi_set_max_buffer_size(a->a_device, 0, request);
      if( ret < 0 ) {
        strbuf_appendf(e, "Comedi set max buffer to %dMiB failed: %C", a->a_req_bufsz_mib);
        return -1;
      }
    }
    ret = comedi_set_buffer_size(a->a_device, 0, request);
    if( ret < 0 ) {
      strbuf_appendf(e, "Comedi set streaming buffer to %dMiB failed: %C", a->a_req_bufsz_mib);
      return -1;
    }
  }

  a->a_bufsz_bytes = comedi_get_buffer_size(a->a_device, 0);
  a->a_bufsz_samples = a->a_bufsz_bytes / sizeof(sampl_t);
  comedi_set_global_oor_behavior(COMEDI_OOR_NUMBER);

  /* Initialise the command structure */
  ret = comedi_get_cmd_generic_timed(a->a_device, 0, &a->a_command, N_USBDUX_CHANS, 0);
  if(ret < 0) {
    strbuf_appendf(e, "Comedi generic command setup failed: %C");
    return -1;
  }

  /* Set the command parameters from the reader parameter values */
  for(i=0; i<N_USBDUX_CHANS; i++)
    a->a_chans[i] = CR_PACK_FLAGS(i, a->a_range, AREF_GROUND, 0);
  a->a_command.chanlist    = &a->a_chans[0];
  a->a_command.stop_src    = TRIG_NONE;
  a->a_command.stop_arg    = 0;
  a->a_command.convert_arg = a->a_intersample_ns;

  /* Ask the driver to check the command structure and complete any omissions */
  (void) comedi_command_test(a->a_device, &a->a_command);
  ret = comedi_command_test(a->a_device, &a->a_command);
  if( ret < 0 ) {
    strbuf_appendf(e, "Comedi second command test fails: %C");
    return -1;
  }

  /* Check the timing:  a difference here means a problem with the driver */
  if(a->a_command.convert_arg != a->a_intersample_ns) {
    a->a_intersample_ns = a->a_command.convert_arg;
    a->a_totfrequency = 1e9 / a->a_command.convert_arg;
    /* TODO: consider logging a warning here */
  }

  /* Map the Comedi buffer into memory, duplicated */
  void *map = mmap_and_lock(a->a_fd, 0, a->a_bufsz_bytes, PROT_RDONLY|PREFAULT_RDONLY|MAL_LOCKED|MAL_DOUBLED);
  if(map == NULL) {
    strbuf_appendf(e, "Unable to mmap Comedi streaming buffer: %m");
    return -1;
  }
  a->a_comedi_ring = map;

  /* Initialise the sample position indices */
  a->a_head = 0;
  a->a_tail = 0;
  a->a_start_time = 0;
  a->a_head_time  = 0;
  a->a_running = 0;
  return 0;
}

/*
 * Start the ADC data collection.
 */

public int adc_start_data_transfer(adc a, strbuf e) {
  int ret;

  /* Execute the command to initiate data acquisition */
  ret = comedi_command(a->a_device, &a->a_command);
  if(ret < 0) {
    strbuf_appendf(e, "Comedi command failed: %C");
  }
  else {
    a->a_running = 1;
  }
  return ret;
}

/*
 * Stop the ADC data collection.
 */

public void adc_stop_data_transfer(adc a) {
  if(a->a_running) {
    comedi_cancel(a->a_device, 0);
    a->a_running = 0;
    a->a_live = 0;
  }
}

/*
 * Convert a sample index into an ADC ring pointer.  This is used by
 * adc_setup_chunk().  It depends on the fact that the Comedi buffer
 * is double-mapped so the pointer is always the start of a contiguous
 * block of memory that will at some time hold the data for the chunk.
 */

private sampl_t *adc_sample_to_ring_ptr(adc a, uint64_t sample) {
  return &a->a_comedi_ring[sample % a->a_bufsz_samples];
}

/*
 * Set up the ADC-dependent information in a chunk, and determine whether the chunk is recordable.
 * In case of error, set the c_error strbuf and set the c_status code to SNAPSHOT_ERROR.
 */

public void adc_setup_chunk(adc a, chunk_t *c) {
  if(c->c_first < a->a_tail) {  /* Too late */
    strbuf_appendf(c->c_error, "Chunk was %d [us] too late", (int)((a->a_tail - c->c_first)/1000));
    c->c_ring = NULL;
    return;
  }
  c->c_ring = adc_sample_to_ring_ptr(a, c->c_first);
  return;
}

/*
 * Convert times to sample indices and vice versa
 */

public uint64_t adc_time_to_sample(adc a, uint64_t time) {
  uint64_t ret;

  ret = (time - a->a_start_time) / a->a_intersample_ns;
  return ret;
}

public uint64_t adc_sample_to_time(adc a, uint64_t sample) {
  uint64_t ret;

  ret = a->a_start_time + sample*a->a_intersample_ns;
  return ret;
}

/*
 * Read-only access to some ADC parameters
 */

public int adc_ns_per_sample(adc a) {
  return a->a_intersample_ns;
}

public double adc_tot_frequency(adc a) {
  return a->a_totfrequency;
}

public uint64_t adc_capture_start_time(adc a) {
  return a->a_start_time;
}

public uint64_t adc_capture_head_time(adc a) {
  return a->a_head_time;
}

public int adc_is_running(adc a) {
  return a && a->a_running;
}

public int adc_is_live(adc a) {
  return a && a->a_live;
}

public uint64_t adc_ring_head(adc a) {
  return a->a_head;
}

public uint64_t adc_ring_tail(adc a) {
  return a->a_tail;
}

/*
 * The buffer strategy implied below is an explicit one of
 * periodically advancing the tail to avoid buffer overrun.  The data
 * bounded by the tail and head pointers in the ring buffer is valid,
 * under this explicit stragety.
 */

/*
 * Recognise data in the Comedi buffer: ask Comedi how much new data
 * is available, set the local data structure to match, tell Comedi we
 * have accepted the data.  If this is the first data received this
 * time, we compute the start time, i.e. the timestamp for sample
 * index 0, from the current head timestamp and the amount of data
 * obtained.
 */

public int adc_data_collect(adc a) {
  uint64_t now;
  int      nb;
  int      ns;

  /* Retrieve any new data if possible */
  nb  = comedi_get_buffer_contents(a->a_device, 0);
  now = monotonic_ns_clock();
  if(nb) {
    ns  = nb / sizeof(sampl_t);
    a->a_head_time = now;
    a->a_head = a->a_tail + ns; /* Assume that nb accumulates if mark read not called */
    if( !a->a_live ) {          /* Estimate the timestamp of sample index 0 */
```

```
      a->a_start_time = a->a_head_time - ns*a->a_intersample_ns;
      a->a_live++;
    }
  }
  return nb;
}

/*
 * Purge data from the tail of the ring buffer if explicit data
 * lifetime management is used.
 */

public int adc_data_purge(adc a, int ns) {
  int nb = ns*sizeof(sampl_t);
  int ret;

  ret = comedi_mark_buffer_read(a->a_device, 0, nb);
  if(ret != nb)
    return -1;
  if( a->a_head >= a->a_bufsz_samples ) {
    a->a_tail += ns;
  }
  return 0;
}
```

```
#

#ifndef _ADC_H
#define _ADC_H

#include "general.h"

/*
 * Descriptor structure for Reader ADC interface.
 */

#define NCHANNELS       8       /* Public number of channels offered */

typedef struct _adc *adc;

export adc  adc_new(strbuf);
export void adc_destroy(adc);

export int  adc_set_frequency(adc, strbuf, double *);
export int  adc_set_bufsz(adc, strbuf, int);
export int  adc_set_range(adc, strbuf, int);
export int  adc_set_device(adc, const char *);
export void adc_set_raw_mode(adc, int);

export int  adc_init(adc, strbuf);
export int  adc_start_data_transfer(adc, strbuf);
export void adc_stop_data_transfer(adc);
export void adc_setup_chunk(adc, chunk_t *);

export uint64_t adc_time_to_sample(adc, uint64_t);
export uint64_t adc_sample_to_time(adc, uint64_t);

export int adc_ns_per_sample(adc);
export double adc_tot_frequency(adc);
export uint64_t adc_capture_start_time(adc);

export uint64_t adc_ring_head(adc);
export uint64_t adc_ring_tail(adc);

export int adc_data_collect(adc);
export int adc_data_purge(adc,int);

#endif /* _ADC_H */
```

```
#

#ifndef _ARGTAB_HELPER_H
#define _ARGTAB_HELPER_H

#include "general.h"
#include "assert.h"
#include "param.h"

/*
 * Simplify definition of command line parsing tables
 *
 *   The argument syntax definition, in the form of calls to argtable2
 * constructors, goes between the BEGIN_CMD_SYNTAX() and APPLY_CMD_DEFAULTS()
 * macro calls, enclosed in { } as an initialiser list (comma−separated).  The
 * default assignments are placed between { } as a statement list, following
 * APPLY_CMD_DEFAULTS() and finishing with END_CMD:SYNTAX().  The result is that
 * the default assignments are placed inside the constructor built by the whole
 * macro set.
 *
 * Keeping the { } outside the macros, and following the END_CMD_SYNTAX() call
 * with a semicolon (null statement) allows emacs font−lock to keep up :−))
 */

#define BEGIN_CMD_SYNTAX(name) void **arg_make_ ## name () { void **ret, *argtable[] =

#define APPLY_CMD_DEFAULTS(name)        ;                               \
                                                                        \
  if( arg_nullcheck(argtable) ) {                                       \
    arg_freetable(argtable, sizeof(argtable)/sizeof(void *));           \
    return NULL;                                                        \
  }                                                                     \
                                                                        \
  ret = malloc(sizeof(argtable));                                       \
  if( !ret ) {                                                          \
    arg_freetable(argtable, sizeof(argtable)/sizeof(void *));           \
    return NULL;                                                        \
  }                                                                     \
  do

#define INCLUDE_PARAM_DEFAULTS(ps,nps)                                  \
  int rv = arg_defaults_from_params(argtable,                           \
                                    sizeof(argtable)/sizeof(void *),    \
                                    (ps), (nps));                       \
  assertv(rv == 0, "Argtable has no end mark\n");

#define END_CMD_SYNTAX(name)                                            \
                                                                        \
  while(0);                                                             \
                                                                        \
  int n = sizeof(argtable)/sizeof(void *);                              \
  while( n−− > 0 ) {                                                    \
    ret[n] = argtable[n];                                               \
  }                                                                     \
  return ret;                                                           \
}

#endif /* _ARGTAB_HELPER_H */
```

```
#

#include "general.h"

#include <argtable2.h>
#include <stdlib.h>
#include "argtab−int16.h"

/*
 * Callback functions for this argument class −− based closely on the
 * argtable2 examples.
 */

/* Private error codes for this type */
enum {OK=0,EMINCOUNT,EMAXCOUNT,EBADVALUE};

/* Reset the parent argument count */
private void resetfn(struct arg_int16 *parent)
{
  parent−>count=0;
}

/* Read a value from an argument string */
private int scanfn(struct arg_int16 *parent, const char *argval)
{
  long long int val;
  char *left;

  if (parent−>count == parent−>hdr.maxcount)
    {
      /* maximum number of arguments exceeded */
      return EMAXCOUNT;
    }
  if (!argval)
    {
      /* an argument with no argument value was given. */
      /* This happens when an optional argument value was invoked. */
      /* leave parent argument value unaltered but still count the argument. */
      parent−>count++;
      return 0;
    }

  /* Try to convert the argument string */
  val = strtoll(argval, &left, 0);

  if (*left == ’\0’) {
    /* success; value was scanned ok, and it is within our desired range.  */
    parent−>data[parent−>count++] = val;
    return OK;
  }

  /* failure; command line string was not a valid integer */
  return EBADVALUE;
}

/* Check for presence of required arguments */
private int checkfn(struct arg_int16 *parent)
{
  /* return EMINCOUNT if the minimum argment count has not been satisfied */
  if( parent−>count < parent−>hdr.mincount )
    return EMINCOUNT;
  else
    return OK;
}

/* Error handler function */
private void errorfn(struct arg_int16 *parent, FILE *fp, int errorcode, const char *argval, const char *progname)
{
  const char *shortopts = parent−>hdr.shortopts;
  const char *longopts  = parent−>hdr.longopts;
  const char *datatype  = parent−>hdr.datatype;

  /* make argval NULL safe */
  argval = argval ? argval : "";
  fprintf(fp,"%s: ",progname);
  switch(errorcode)
    {
    case EMINCOUNT:
      /* We expected more arg_int16 arguments than we received. */
      fputs("missing option \"",fp);
      arg_print_option(fp,shortopts,longopts,datatype,"\"\n");
      break;

    case EMAXCOUNT:
      /* We received more arg_int16 arguments than we expected. */
      fputs("excess option \"",fp);
      arg_print_option(fp,shortopts,longopts,argval,"\"\n");
      break;

    case EBADVALUE:
      /* An arg_int16 option was given with an invalid value */
      fprintf(fp,"invalid argument \"%s\" to option ",argval);
      arg_print_option(fp,shortopts,longopts,datatype,"\n");
      break;
    }
}

/* Generic constructor for an arg_int16 structure */
struct arg_int16* arg_int16n(const char* shortopts, const char* longopts,
                             const char *datatype,
                             int mincount, int maxcount, const char *glossary) {
  int bytes;
  struct arg_int16 *ret;

  bytes = sizeof(struct arg_int16) + maxcount*sizeof(uint16_t);
  ret = (struct arg_int16 *)calloc(1, bytes);
  if( ret ) {
    ret->hdr.flag      = ARG_HASVALUE;
    ret->hdr.shortopts = shortopts;
    ret->hdr.longopts  = longopts;
    ret->hdr.datatype  = datatype ? datatype : "<[u]int16_t>";
    ret->hdr.glossary  = glossary;
    ret->hdr.mincount  = mincount;
    ret->hdr.maxcount  = maxcount;
    ret->hdr.parent    = ret;
    ret->hdr.resetfn   = (arg_resetfn *)resetfn;
    ret->hdr.scanfn    = (arg_scanfn *)scanfn;
    ret->hdr.checkfn   = (arg_checkfn *)checkfn;
    ret->hdr.errorfn   = (arg_errorfn *)errorfn;
    ret->count = 0;
    ret->data = (int16_t *)&ret[1];
  }
  return ret;
}

/* Special case: 0 or 1 arguments */
struct arg_int16* arg_int160(const char* shortopts, const char* longopts,
                             const char *datatype,  const char *glossary) {

  return arg_int16n(shortopts, longopts, datatype, 0, 1, glossary);
}

/* Special case: exactly 1 argument */
struct arg_int16* arg_int161(const char* shortopts, const char* longopts,
                             const char *datatype,  const char *glossary) {

  return arg_int16n(shortopts, longopts, datatype, 1, 1, glossary);
}
```

```
#

/*
 * Public definitions for a int16_t argument consistent with argtable2.
 */

#ifndef _ARGTAB_INT16_H
#define _ARGTAB_INT16_H

#include <stdint.h>
#include <argtable2.h>

struct arg_16b
{
  struct arg_hdr hdr;      /* The mandatory argtable header struct */
  int count;               /* Number of matching command line arguments found */
  int16_t *data;           /* Array of matching command line argument data  */
};

struct arg_16b* arg_16b0(const char* shortopts, const char* longopts, const char *datatype,
                             const char *glossary);

struct arg_16b* arg_16b1(const char* shortopts, const char* longopts, const char *datatype,
                             const char *glossary);

struct arg_16b* arg_16bn(const char* shortopts, const char* longopts, const char *datatype,
                             int mincount, int maxcount, const char *glossary);

#endif /* _ARGTAB_INT16_H */
```

```
#

#include "general.h"

#include <argtable2.h>
#include <stdlib.h>
#include "argtab−int64.h"

/*
 * Callback functions for this argument class −− based closely on the
 * argtable2 examples.
 */

/* Private error codes for this type */
enum {OK=0,EMINCOUNT,EMAXCOUNT,EBADVALUE};

/* Reset the parent argument count */
private void resetfn(struct arg_int64 *parent)
{
  parent−>count=0;
}

/* Read a value from an argument string */
private int scanfn(struct arg_int64 *parent, const char *argval)
{
  long long int val;
  char *left;

  if (parent−>count == parent−>hdr.maxcount)
    {
      /* maximum number of arguments exceeded */
      return EMAXCOUNT;
    }
  if (!argval)
    {
      /* an argument with no argument value was given. */
      /* This happens when an optional argument value was invoked. */
      /* leave parent argument value unaltered but still count the argument. */
      parent−>count++;
      return 0;
    }

  /* Try to convert the argument string */
  val = strtoll(argval, &left, 0);

  if (*left == ’\0’) {
    /* success; value was scanned ok, and it is within our desired range.  */
    parent−>data[parent−>count++] = val;
    return OK;
  }

  /* failure; command line string was not a valid integer */
  return EBADVALUE;
}

/* Check for presence of required arguments */
private int checkfn(struct arg_int64 *parent)
{
  /* return EMINCOUNT if the minimum argment count has not been satisfied */
  if( parent−>count < parent−>hdr.mincount )
    return EMINCOUNT;
  else
    return OK;
}

/* Error handler function */
private void errorfn(struct arg_int64 *parent, FILE *fp, int errorcode, const char *argval, const char *progname)
{
  const char *shortopts = parent−>hdr.shortopts;
  const char *longopts  = parent−>hdr.longopts;
  const char *datatype  = parent−>hdr.datatype;

  /* make argval NULL safe */
  argval = argval ? argval : "";
  fprintf(fp,"%s: ",progname);
  switch(errorcode)
    {
    case EMINCOUNT:
      /* We expected more arg_int64 arguments than we received. */
      fputs("missing option \"",fp);
      arg_print_option(fp,shortopts,longopts,datatype,"\"\n");
      break;

    case EMAXCOUNT:
      /* We received more arg_int64 arguments than we expected. */
      fputs("excess option \"",fp);
      arg_print_option(fp,shortopts,longopts,argval,"\"\n");
      break;

    case EBADVALUE:
      /* An arg_int64 option was given with an invalid value */
      fprintf(fp,"invalid argument \"%s\" to option ",argval);
      arg_print_option(fp,shortopts,longopts,datatype,"\n");
      break;
    }
}

/* Generic constructor for an arg_int64 structure */
struct arg_int64* arg_int64n(const char* shortopts, const char* longopts,
                             const char *datatype,
                             int mincount, int maxcount, const char *glossary) {
  int bytes;
  struct arg_int64 *ret;

  bytes = sizeof(struct arg_int64) + maxcount*sizeof(uint64_t);
  ret = (struct arg_int64 *)calloc(1, bytes);
  if( ret ) {
    ret->hdr.flag      = ARG_HASVALUE;
    ret->hdr.shortopts = shortopts;
    ret->hdr.longopts  = longopts;
    ret->hdr.datatype  = datatype ? datatype : "<[u]int64_t>";
    ret->hdr.glossary  = glossary;
    ret->hdr.mincount  = mincount;
    ret->hdr.maxcount  = maxcount;
    ret->hdr.parent    = ret;
    ret->hdr.resetfn   = (arg_resetfn *)resetfn;
    ret->hdr.scanfn    = (arg_scanfn *)scanfn;
    ret->hdr.checkfn   = (arg_checkfn *)checkfn;
    ret->hdr.errorfn   = (arg_errorfn *)errorfn;
    ret->count = 0;
    ret->data = (int64_t *)&ret[1];
  }
  return ret;
}

/* Special case: 0 or 1 arguments */
struct arg_int64* arg_int640(const char* shortopts, const char* longopts,
                             const char *datatype,  const char *glossary) {

  return arg_int64n(shortopts, longopts, datatype, 0, 1, glossary);
}

/* Special case: exactly 1 argument */
struct arg_int64* arg_int641(const char* shortopts, const char* longopts,
                             const char *datatype,  const char *glossary) {

  return arg_int64n(shortopts, longopts, datatype, 1, 1, glossary);
}
```

```
#

/*
 * Public definitions for a int64_t argument consistent with argtable2.
 */

#ifndef _ARGTAB_INT64_H
#define _ARGTAB_INT64_H

#include <stdint.h>
#include <argtable2.h>

struct arg_64b
{
  struct arg_hdr hdr;      /* The mandatory argtable header struct */
  int count;               /* Number of matching command line arguments found */
  int64_t *data;           /* Array of matching command line argument data  */
};

struct arg_64b* arg_64b0(const char* shortopts, const char* longopts, const char *datatype,
                             const char *glossary);

struct arg_64b* arg_64b1(const char* shortopts, const char* longopts, const char *datatype,
                             const char *glossary);

struct arg_64b* arg_64bn(const char* shortopts, const char* longopts, const char *datatype,
                             int mincount, int maxcount, const char *glossary);

#endif /* _ARGTAB_INT64_H */
```

```
#

#ifndef _LOCAL_ASSERT_H
#define _LOCAL_ASSERT_H

/*
 * Local version of assert, bit more informative than system version
 */

#ifdef USE_SYSTEM_ASSERT

#include <assert.h>

#define assertv(cond, ...) assert(cond)

#else

#include <stdio.h>
#include <stdlib.h>

#define assertv(cond,fmt, ...) do {                      \
  if(!(cond)) { \
    fprintf(stderr, "FAILED ASSERTION −− %s:%d %s %s\n" fmt, __FILE__, __LINE__, __FUNCTION__, "’" #cond "’" , ## __VA_ARGS__ ); \
    abort(); \
  } } while(0)

#endif

#endif /* _LOCAL_ASSERT_H */
```

```
#

#include "general.h"

#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <comedilib.h>

#include "assert.h"
#include "queue.h"
#include "mman.h"
#include "strbuf.h"
#include "chunk.h"
#include "writer.h"

struct _frame {
  queue f_Q;
  block f_map;
};

/*
 * Set up the mmap frames for data transfer to snapshot files.
 */

private int    nframes;   /* The number of simultaneous mmap frames */
private frame *framelist;  /* The list of mmap frame descriptors */
private int    n_frame_Q = 0;

private QUEUE_HEADER(frameQ);

public int init_frame_system(strbuf e, int nfr, int ram, int chunk) {

  framelist = (frame *)calloc(nfr, sizeof(frame));
  if( framelist ) {
    void *map = mmap_locate(ram*1024*1024, 0);
    int   n;

    if(map == NULL) {
      strbuf_appendf(e, "Cannot mmap %d MiB of locked transfer RAM: %m", ram);
      free((void *) framelist );
      return −1;
    }
    for(n=0; n<nfr; n++) { /* Initialise the frame memory pointers, leave sizes as 0 */
      framelist[n].f_map.b_data = map;
      map += chunk;
      init_queue(&framelist[n].f_Q);
      queue_ins_before(&frameQ, &framelist[n].f_Q);
      n_frame_Q++;
    }
  }
  else {
    strbuf_appendf(e, "Cannot allocate frame list memory for %d frames: %m", nfr);
    return −1;
  }
  nframes = nfr;
  return 0;
}

/*
 * Scan the frame list and pull any free frame descriptors into the free queue.
 *
 * A descriptor is free if its byte count is zero, and it is not in
 * the free queue if its queue structure is a singleton.
 */

private void scan_framelist() {
  int    n;
  frame *f;

  for(n=0,f=framelist; n<nframes; n++, f++) {
    if(f−>f_map.b_bytes)        /* If non−zero, it’s in use */
      continue;
    if( !queue_singleton(&f−>f_Q) )
      continue;
    queue_ins_before(&frameQ, &f−>f_Q);
    n_frame_Q++;
  }
}

/*
 * Allocate a frame descriptor.
 */

private frame *alloc_frame() {
  frame *f;

  if( !n_frame_Q ) {
    assertv(queue_singleton(&frameQ), "Frame queue count is zero for non−empty queue\n");
    scan_framelist();
  }
  if( !n_frame_Q ) {
    errno = EBUSY;
    return NULL;
  }
  f = (frame *)de_queue(queue_next(&frameQ));
  assertv(f != NULL, "Frame queue count %d but queue is empty\n", n_frame_Q);
  n_frame_Q−−;
  f−>f_map.b_bytes = 1;         /* In−use;  real size is filled in by caller */
  return f;
}

/*
 * Release a frame descriptor.  N.B.  this is done in the TIDY thread,
 * so must be atomic: the frame is released by setting the bytes value
 * to zero.  The munmap() here complements the mmap call in the chunk
 * mapper below.
 */

public void release_frame(frame *f) {
  if(f−>f_map.b_data == NULL)
      return;
  munmap(f−>f_map.b_data, f−>f_map.b_bytes);
  f−>f_map.b_bytes = 0;
}

/*
 * Report the index of a frame pointer in the table.
 */

public int frame_nr(frame *f) {
  return f − framelist;
}

/*
 * Functions for dealing with transfer chunk descriptors.
 */

private uint16_t chunk_counter;

#define N_CHUNK_ALLOC   (4096/sizeof(chunk_t))

private QUEUE_HEADER(chunkQ);
private int N_in_chunkQ = 0;

/*
 * Allocate n new chunk descriptors, chained using the WRITER queue descriptor
 */

public chunk_t *alloc_chunk(int nr) {
  queue *ret;

  if( N_in_chunkQ < nr ) {      /* The queue doesn’t have enough */
    int n;

    for(n=0; n<N_CHUNK_ALLOC; n++) {
      queue *q = (queue *)calloc(1, sizeof(chunk_t));

      if( !q ) {                        /* Allocation failed */
        if( N_in_chunkQ >= nr )
```

```
          break;                        /* But we have enough now anyway */
        return NULL;
      }
      init_queue(q);
      queue_ins_after(&chunkQ, q);
      N_in_chunkQ++;
    }
  }

  ret = de_queue(queue_next(&chunkQ));
  chunk_t *c = qp2chunk(ret);
  init_queue(&c->c_rQ);
  c->c_name = ++chunk_counter;

  while(--nr > 0) {             /* Collect enough to satisfy request */
    chunk_t *c = qp2chunk(de_queue(queue_next(&chunkQ)));

    init_queue(&c->c_wQ);       /* Redundant... */
    init_queue(&c->c_rQ);
    c->c_name = ++chunk_counter;
    queue_ins_before(ret, chunk2qp(c));
  }

  return c;
}

/*
 * Finished with chunk descriptors chained using the writer queue descriptor.
 * Assume the reader queue descriptor is detached.
 */

public void release_chunk(chunk_t *c) {
  queue *q = chunk2qp(c);
  queue *p;

  while( (p = de_queue(queue_next(q))) != NULL ) {
    init_queue(p);
    queue_ins_before(&chunkQ, p);
    N_in_chunkQ++;
  }
  init_queue(q);
  queue_ins_before(&chunkQ, q);
  N_in_chunkQ++;
}

/*
 * FInd a frame for a chunk and map the chunk into memory.  This may
 * take arbitrary time since this is where Linux has to find us new
 * pages.  This code runs in the WRITER thread.
 */

public int map_chunk_to_frame(chunk_t *c) {
  frame *fp = alloc_frame();
  void  *map;

  if(fp == NULL) {              /* This is not an error:  there may be no frames available for transient reasons */
    return -1;
  }

  fp->f_map.b_bytes = c->c_samples*sizeof(sampl_t);
  /* Would really like to do WRONLY here, but I *think* that will break */
  map = mmap_and_lock_fixed(c->c_fd, c->c_offset, fp->f_map.b_bytes, PROT_RDWR|PREFAULT_RDWR|MAL_LOCKED, fp->f_map.b_data);
  if(map != fp->f_map.b_data) { /* A (fatal) mapping error occurred... */
    strbuf_appendf(c->c_error, "Unable to map chunk c:%04hx to frame %d: %m", c->c_name, frame_nr(fp));
    c->c_status = SNAPSHOT_ERROR;
    return -1;
  }
  c->c_frame = fp;              /* Succeeded, chunk now has a mapped frame */
  return 0;
}

/*
 * Copy the data for a chunk from the ring buffer into the frame.
 * Apply the appropriate ADC conversion.
 */
```

```
public void copy_chunk_data(chunk_t *c) {
  convertfn fn = c->c_convert;

  (*fn)((sampl_t *)c->c_frame->f_map.b_data, (sampl_t *)c->c_ring, c->c_samples);
  c->c_status = SNAPSHOT_WRITTEN;
}

/*
 * Generate a debugging line for a chunk desdcriptor.  Put it in the buffer buf.
 * Return the actual size, no greater than the space available.
 */

#define qp2cname(p)     (qp2chunk(p)->c_name)
#define rq2cname(p)     (rq2chunk(p)->c_name)

public int debug_chunk(char buf[], int space, chunk_t *c) {
  import const char *snapshot_status(int);
  import uint16_t    snapfile_name(snapfile_t *);
  int used;

  used = snprintf(buf, space,
                  "chunk c:%04hx at %p"
                  "wQ[c:%04hx,c:%04hx] "
                  "rQ[c:%04hx,c:%04hx] "
                  "RG %p FR %p PF f:%04hx status %s "
                  "S:%08lx F:%016llx L:%016llx\n",
                  c->c_name, c,
                  qp2cname(queue_prev(&c->c_wQ)), qp2cname(queue_next(&c->c_wQ)),
                  rq2cname(queue_prev(&c->c_rQ)), rq2cname(queue_next(&c->c_rQ)),
                  c->c_ring, c->c_frame, snapfile_name(c->c_parent), snapshot_status(c->c_status),
                  c->c_samples, c->c_first, c->c_last
                  );
  if(used >= space)
    used = space;
  return used;
}
```

```
#

#ifndef _CHUNK_H
#define _CHUNK_H

#include "general.h"
#include <comedilib.h>
#include "lut.h"

/* Structure for a memory block */

typedef struct {
  void *b_data;
  int   b_bytes;
}
  block;

typedef struct _frame frame;
typedef struct _sfile snapfile_t;

#include "queue.h"

typedef struct {
  queue         c_Q[2];     /* Q header for READER capture queue and WRITER file chunk list*/
#define c_wQ c_Q[0]         /* Chunk Q linkage associated with the file */
#define c_rQ c_Q[1]         /* Chunk Q linkage associated with the data flow */
  frame        *c_frame;    /* Mmap’d file buffer for this chunk */
  strbuf        c_error;    /* Error buffer, for error messages (copy from snapshot_t origin) */
  snapfile_t   *c_parent;   /* Chunk belongs to this file */
  uint64_t      c_first;    /* First sample of this chunk */
  uint64_t      c_last;     /* First sample beyond this chunk */
  int16_t      *c_ring;     /* Ring buffer start for this chunk */
  convertfn     c_convert;  /* Function to copy samles into frame with conversion */
  uint32_t      c_samples;  /* Number of samples to copy */
  uint32_t      c_offset;   /* File offset for this chunk */
  int           c_status;   /* Status of this capture chunk */
  int           c_fd;       /* File descriptor for this chunk */
  uint16_t      c_name;     /* Unique name for this chunk */
}
  chunk_t;

#define qp2chunk(q)     ((chunk_t *)(q))
#define chunk2qp(c)     (&(c)->c_Q[0])

#define chunk2rq(c)     (&(c)->c_rQ)
#define rq2chunk(q)     ((chunk_t *)&((q)[-1]))

export chunk_t *alloc_chunk(int);
export void release_chunk(chunk_t *);
export int map_chunk_to_frame(chunk_t *);

export int debug_chunk(char [], int, chunk_t *);

export void release_frame(frame *);

#endif /* _CHUNK_H */
```

```
#

#ifndef _GENERAL_H

/*
 * Macro definitions to make static and extern more explicit.
 */

#define public
#define import  extern
#define export  extern
#define private static
#define persist static

#endif /* _GENERAL_H */
```

```
#

/*
 * Program to grab data from USBDUXfast via Comedi.
 *
 * Arguments:
 * --verbose|-v         Increase reporting level
 * --freq|-f            Sampling frequency in [Hz], default 2.5 [MHz]
 * --range|-r           ADC range 'hi' (750 mVpk) or 'lo' (500 mVpk)
 * --raw                ADC output as raw data
 * --device|-d          Comedi device to use, default /dev/comedi0
 * --bufsz|-B           Comedi buffer size to request [MiB], default 40 [MiB]
 * --help|-h            Print usage message
 * --version            Print program version
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include "assert.h"
#include <argtable2.h>
#include <regex.h>
#include <comedi.h>
#include <comedilib.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#include "argtab.h"
#include "mman.h"
#include "lut.h"

#define N_CHANS      16
#define BUFSZ      4096
#define BUFSPSZ    (BUFSZ/sizeof(sampl_t))

char read_buf[BUFSZ];

#define COMEDI_DEVICE   "/dev/comedi0"

#define COMEDIBUFFERSIZE (*40)
#define COMEDIBUFFERSPLS (COMEDIBUFFERSIZE/sizeof(sampl_t))

#define PROGRAM_VERSION         "2.0"
#define VERSION_VERBOSE_BANNER  "MCLURS ADC toolset...\n"

/* Standard arguments + flags */
int   verbose = 0;
char *program = NULL;

/* Command line syntax options */

struct arg_lit *h1, *vn1, *v1;
struct arg_end *e1;

BEGIN_CMD_SYNTAX(help) {
  v1  = arg_litn("v",   "verbose", 0, 2,        "Increase verbosity"),
  h1  = arg_lit0("h",   "help",                 "Print usage help message"),
  vn1 = arg_lit0(NULL,  "version",              "Print program version string"),
  e1  = arg_end(20)
} APPLY_CMD_DEFAULTS(help) {
  /* No defaults to apply here */
} END_CMD_SYNTAX(help)

struct arg_lit *v2, *rw2;
struct arg_int *b2;
struct arg_dbl *f2;
struct arg_str *d2;
struct arg_rex *rn2;
struct arg_end *e2;

BEGIN_CMD_SYNTAX(main) {
  v2  = arg_litn("v",   "verbose", 0, 2,                "Increase verbosity"),
  f2  = arg_dbl0("f",   "freq", "<real>",               "Sampling frequency [Hz], default 2.5 [MHz]"),
  b2  = arg_int0("B",   "bufsz",  "<int>",                       "Comedi buffer size [MiB], default 40 [MiB]"),
  d2  = arg_str0("d",   "device",  "<path>",                     "Comedi device to open, default /dev/comedi0"),
  rn2 = arg_rex0("r",   "range",  "hi|lo", NULL, REG_EXTENDED,          "Specify range in {hi, lo}, default hi"),
  rw2 = arg_lit0(NULL,  "raw",                                   "Emit raw ADC sample values"),
  e2  = arg_end(20)
} APPLY_CMD_DEFAULTS(main) {
  *f2->dval  = 2.5e6;            /* Default frequency 2.5 [MHz] */
  *b2->ival  = 40;               /* Default buffer size 40 [MiB] */
  *d2->sval  = COMEDI_DEVICE;    /* Default device for Comedi */
  *rn2->sval = "hi";             /* Default ADC range (hi) */
} END_CMD_SYNTAX(main);

/* Standard help routines: display the version banner */
void print_version(FILE *fp, int verbosity) {
  fprintf(fp, "%s: Vn. %s\n", program, PROGRAM_VERSION);
  if(verbosity > 0) {            /* Verbose requested... */
    fprintf(fp, VERSION_VERBOSE_BANNER);
  }
}

/* Standard help routines: display the usage summary for a syntax */
void print_usage(FILE *fp, void **argtable, int verbosity, char *program) {
  if( !verbosity ) {
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntax(fp, argtable, "\n");
    return;
  }
  if( verbosity ) {
    char *suffix = verbosity>1? "\n\n" : "\n";
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntaxv(fp, argtable, suffix);
    if( verbosity > 1 )
      arg_print_glossary(fp, argtable, "%-25s %s\n");
  }
}

/*
 * The main() entry point.
 */

int main(int argc, char *argv[]) {
  float         sr_total;
  int           bufsz;
  char         *device;
  int           range = 0;       /* Default range is +/- 750mV */

  int           buf_samples;
  unsigned int convert_arg;
  comedi_t     *dev;
  int           errs, ret, i;
  unsigned int chanlist[N_CHANS];
  void         *map;
  sampl_t      *start;
  uint64_t      head, tail;
  int           data_coming;
  void        (*convert)(sampl_t *, sampl_t *, int);

  program = argv[0];

  /* Create and parse the command lines */
  void **cmd_help = arg_make_help();
  void **cmd_main = arg_make_main();

  /* Try first syntax */
  int err_help = arg_parse(argc, argv, cmd_help);
  if( !err_help ) {              /* Assume this was the desired command syntax */
    if(vn1->count)
      print_version(stdout, v1->count);
    if(h1->count || !vn1->count) {
      print_usage(stdout, cmd_help, v1->count>0, program);
      print_usage(stdout, cmd_main, v1->count, program);
    }
    exit(0);
  }
```

```
/* Try second syntax */
int err_main = arg_parse(argc, argv, cmd_main);
if( err_main ) {              /* This is the default desired syntax; give full usage */
  arg_print_errors(stderr, e2, program);
  print_usage(stderr, cmd_help, v2->count>0, program);
  print_usage(stderr, cmd_main, v2->count, program);
  exit(1);
}

/* The second syntax was correctly parsed, so retrieve the important values from the table */
errs = 0;

/* Deal with the sampling frequency */
sr_total    = f2->dval[0];
if(sr_total < 5e4 || sr_total > 3e6) {
  fprintf(stderr, "%s: Error -- total sample rate %g [Hz] out of sensible range (50 [kHz] to 3 [MHz])\n", program, sr_total);
  errs++;
}
convert_arg = (unsigned int) 1e9 / sr_total;

/* Deal with the requested buffer size */
bufsz       = b2->ival[0];
if(bufsz < 8 || bufsz > 256) {
  fprintf(stderr, "%s: Error -- requested buffer size %d [MiB] out of sensible range (8 to 256 [MiB])\n", program, bufsz);
  errs++;
}
bufsz *= 1048576;
buf_samples = bufsz / sizeof(sampl_t);

/* Deal with the Comedi device */
device      = (char *) d2->sval[0];
if( !(dev = comedi_open(device)) ) {
  fprintf(stderr, "%s: Error -- cannot open %s: %s\n", program, device, comedi_strerror(comedi_errno()));
  errs++;
}

/* Deal with specification of range and raw */
range       = !strcmp(rn2->sval[0], "hi") ? 0 : 1;
if(rw2->count) {              /* Requested raw */
  convert   = convert_raw_raw;
} else {                      /* Convert from specified range  */
  convert   = range? convert_raw_500mV : convert_raw_750mV;
}

/* Record desired verbosity */
verbose     = v2->count;

/* All finished with the argument syntax tables */
arg_free(cmd_main);
arg_free(cmd_help);

/* Exit 2 if argument errors */
if(errs) {
  exit(2);
}

fprintf(stderr, "%s $s\n\n", program, PROGRAM_VERSION);
fprintf(stderr, "Total sample rate requested = %g [Hz]\n", sr_total);
fprintf(stderr, "Using ADC range +/-%s [mV] full-scale\n", range? "500" : "750");

comedi_cmd *cmd = (comedi_cmd *) calloc(1, sizeof(comedi_cmd));
if( !cmd ) {
  fprintf(stderr, "%s: Error -- failed to get memory for Comedi command structure: %s\n", program, strerror(errno));
  exit(3);
}

ret = comedi_get_max_buffer_size(dev, 0);

if(ret < bufsz) {
  if(comedi_set_max_buffer_size(dev, 0, bufsz) < 0 ) {
    fprintf(stderr, "%s: Error -- failed to set %s max buffer size to %d bytes: %s\n", program, device, bufsz, comedi_strerror(comedi_errno()));
    exit(3);
  }
}
else {
    if(verbose)
      fprintf(stderr, "%s: Comedi maximum buffer size requested %d, actual %d\n", program, bufsz, ret);
  }

  if(comedi_set_buffer_size(dev, 0, bufsz) < 0) {
    fprintf(stderr, "%s: Error -- failed to set %s buffer size to %d bytes: %s\n", program, device, bufsz, comedi_strerror(comedi_errno()));
    exit(3);
  }

  for(i=0; i<N_CHANS; i++)
    chanlist[i] = CR_PACK(i, range, AREF_GROUND);

  /* Print numbers for clipped inputs */
  comedi_set_global_oor_behavior(COMEDI_OOR_NUMBER);

  /* get the correct command structure to run usbduxfast */
  if((ret = comedi_get_cmd_generic_timed(dev, 0, cmd, N_CHANS, 0)) < 0) {
    fprintf(stderr, "%s: Error -- comedi_get_cmd_generic_timed failed for %s: %s\n", program, device, comedi_strerror(comedi_errno()));
    exit(3);
  }

  populate_conversion_luts();

  /* adjust some cmd parameters */
  cmd->chanlist    = chanlist;
  cmd->stop_src    = TRIG_NONE;
  cmd->stop_arg    = 0;
  convert_arg = (unsigned int) 1e9 / sr_total;
  cmd->convert_arg = convert_arg;

  /* call test twice because different things are tested?
   * if tests are successful run sampling command */
  if( (ret = comedi_command_test(dev, cmd)) != 0 && verbose > 1 ) {
    fprintf(stderr, "First test, err: %s; ", comedi_strerror(comedi_errno()));
    fprintf(stderr, " cmd->convert_arg = %d\n", cmd->convert_arg);
  }
  if( (ret = comedi_command_test(dev, cmd)) != 0 && verbose > 1 ) {
    fprintf(stderr, "Second test, err: %s; ", comedi_strerror(comedi_errno()));
    fprintf(stderr, " cmd->convert_arg = %d\n", cmd->convert_arg);
  }

  map = mmap_and_lock(comedi_fileno(dev), 0, bufsz, PROT_RDONLY|PREFAULT_RDONLY|MAL_DOUBLED|MAL_LOCKED);
  if(map == NULL) {
    fprintf(stderr, "%s: Error -- failed to map Comedi buffer to RAM: %s\n", program, strerror(errno));
    exit(3);
  }

  if(verbose)
    fprintf(stderr, "%s: Comedi buffer (size %u bytes) mapped at 0x%p\n", program, bufsz, start);

  if( (ret = comedi_command(dev, cmd))       < 0 ) {
    fprintf(stderr, "%s: Error -- Comedi command returns %d: %s\n", program, ret, comedi_strerror(comedi_errno()));
    exit(4);
  }

  if(verbose > 1) {
    fprintf(stderr, "comedi_command returns %d\n", ret);
    fprintf(stderr, "stop src = %d\n", cmd->stop_src);
    fprintf(stderr, "stop arg = %d\n", cmd->stop_arg);
    fprintf(stderr, "convert arg = %d\n", cmd->convert_arg);
  }

  if(verbose)
    fprintf(stderr, "%s: Total sample rate allocated = %g Hz\n", program, 1e9 / cmd->convert_arg);

  head=tail=0;
  data_coming = 1000;           /* Is data arriving? After this many pauses with no data, exit... */
  while( 1 ) {
    int nb  = comedi_get_buffer_contents(dev, 0); /* Find out how many new bytes there are */
    sampl_t *back = &start[ tail % buf_samples ];
    ret = 0;

    if(nb <= 0) {
      usleep(10000);
      if( --data_coming == 0 ) break;
      continue;
    }

    data_coming = 100;          /* Some data has come, use a smaller value henceforth */
    head += nb/sizeof(sampl_t); /* This many new samples have arrived */
    nb = head - tail;           /* And this is how many remain to process */

    while( nb >= BUFSPSZ ) {
      /* Convert and dump a buffer-full to stdout, repeat while possible */
      (*convert)((sampl_t *)read_buf, back, BUFSPSZ);
      ret  =  comedi_mark_buffer_read(dev, 0, BUFSZ);
      if(ret < 0) {
        fprintf(stderr, "%s: Error -- comedi_mark_buffer_read during loop: %s\n", program, comedi_strerror(comedi_errno()));
        break;
      }
      fwrite(read_buf, sizeof(sampl_t), BUFSPSZ, stdout);
      back += BUFSPSZ;
      tail += BUFSPSZ;
      nb   -= BUFSPSZ;
    }
  }

  fprintf(stderr, "%s: Error? -- Comedi data flow interrupted for more than 1 second\n", program);

  comedi_cancel(dev, 0);
  exit(0);
}
```

```
#

#include "general.h"

#include "assert.h"
#include <string.h>
#include <comedi.h>

#include "lut.h"

/*
 * Construct a look up table to map the USBDUXfast ADC outputs into 1V
 * pk s16 representation.
 *
 * The raw data goes from 000 to FFF, or-ed with 0x1000 if an overflow occurs.
 * The table is indexed with the raw data value to generate the s16 value.
 *
 * In the case of overflow, the value converted is 1 more than the
 * maximum raw value returnable by the ADC.
 *
 * There is one table for when the USBduxFAST is in 0.5V scale mode
 * and one for 0.75V scale.
 */

#define CONVERT_BY_LUT          /* Use a lookup table to do the complete conversion from raw to normalised */

#define ADC_BITS        12

#define USBDUXFAST_OOR  (1<<ADC_BITS)
#define USBDUXFAST_SIGN (1 << (ADC_BITS-1))
#define USBDUXRAW_MIN   0
#define USBDUXRAW_MAX   (~((~0)<<ADC_BITS))

#define RAW_500mV_TO_OUT_500mV(raw)     (((short)(((raw)<<4) ^ 0x8000)) >> 1)       /* Shift up and correct sign bit, then arithmetic shift back 1 */
#define OUT_500mV_TO_OUT_750mV(raw)     ((raw)+(short)((raw) >> 1))                 /* Add 0.5 times value you first thought of... */

#define USBDUXFAST_OOR_POS_500mV        (RAW_500mV_TO_OUT_500mV(USBDUXRAW_MAX)+1)
#define USBDUXFAST_OOR_NEG_500mV        (RAW_500mV_TO_OUT_500mV(USBDUXRAW_MIN)-1)
#define USBDUXFAST_OOR_POS_750mV        (OUT_500mV_TO_OUT_750mV(RAW_500mV_TO_OUT_500mV(USBDUXRAW_MAX))+1)
#define USBDUXFAST_OOR_NEG_750mV        (OUT_500mV_TO_OUT_750mV(RAW_500mV_TO_OUT_500mV(USBDUXRAW_MIN))-1)

/* Define the look-up tables for the conversion */
/* Using LUT only doubles the table size (but probably saves some time) */
#ifdef  CONVERT_BY_LUT
#define TABLE_SIZE      (2*(1<<ADC_BITS))
#else
#define TABLE_SIZE      (1<<ADC_BITS)
#endif

private sampl_t lut_raw_to_1Vpk_500mV[TABLE_SIZE];
private sampl_t lut_raw_to_1Vpk_750mV[TABLE_SIZE];

private int lut_not_ready = 1;

public void populate_conversion_luts() {
  short raw;

  assertv(sizeof(sampl_t) == 2, "sizeof(sampl_t) is %d not 2\n", sizeof(sampl_t));                /* Check type definitions on this architecture */
  assertv(RAW_500mV_TO_OUT_500mV(USBDUXRAW_MAX) > 0, "ADC mapped max not positive\n");          /* Should work if sampl_t is signed short */
  assertv(RAW_500mV_TO_OUT_500mV(USBDUXRAW_MIN) < 0, "ADC mapped min not negative\n");

  if( !lut_not_ready )          /* i.e. the tables are already ready */
    return;

  for(raw=0; raw<=0xFFF; raw++) {
    short conv = RAW_500mV_TO_OUT_500mV(raw);

    lut_raw_to_1Vpk_500mV[raw] = conv;                     /* Raw value maps to itself x 8 with sign corrected */
    lut_raw_to_1Vpk_750mV[raw] = OUT_500mV_TO_OUT_750mV(conv);  /* Values in 0.75pk range are scaled by 1.5 */
#ifdef CONVERT_BY_LUT
    lut_raw_to_1Vpk_500mV[raw+0x1000] = (raw&0x800)? USBDUXFAST_OOR_POS_500mV : USBDUXFAST_OOR_NEG_500mV;
    lut_raw_to_1Vpk_750mV[raw+0x1000] = (raw&0x800)? USBDUXFAST_OOR_POS_750mV : USBDUXFAST_OOR_NEG_750mV;
#endif
  }
  lut_not_ready = 0;            /* The tables are ready now... */
}

public void convert_raw_500mV(sampl_t *dst, sampl_t *src, int nsamples) {
  if(lut_not_ready)
    populate_conversion_luts();

  while(nsamples-- > 0) {
#ifdef  CONVERT_BY_LUT
    *dst++ = lut_raw_to_1Vpk_500mV[*src++ & (USBDUXFAST_OOR | USBDUXRAW_MAX)];
#else
    sampl_t s = *src++ & (USBDUXFAST_OOR | USBDUXRAW_MAX);
    if(s&USBDUXFAST_OOR) {
      *dst++ = (s&0x800)? USBDUXFAST_OOR_POS_500mV : USBDUXFAST_OOR_NEG_500mV;
      continue;
    }
    else
      *dst++ = lut_raw_to_1Vpk_500mV[*src++];
#endif
  }
}

public void convert_raw_750mV(sampl_t *dst, sampl_t *src, int nsamples) {
  if(lut_not_ready)
    populate_conversion_luts();

  while(nsamples-- > 0) {
#ifdef  CONVERT_BY_LUT
    *dst++ = lut_raw_to_1Vpk_750mV[*src++ & (USBDUXFAST_OOR | USBDUXRAW_MAX)];
#else
    sampl_t s = *src++ & (USBDUXFAST_OOR | USBDUXRAW_MAX);
    if(s&USBDUXFAST_OOR) {
      *dst++ = (s&0x800)? USBDUXFAST_OOR_POS_750mV : USBDUXFAST_OOR_NEG_750mV;
      continue;
    }
    else
      *dst++ = lut_raw_to_1Vpk_500mV[*src++];
#endif
  }
}

public void convert_raw_raw(sampl_t *dst, sampl_t *src, int nsamples) {
  if(dst == src)
    return;
  memcpy(dst, src, nsamples*sizeof(sampl_t));
}
```

```
#

#ifndef _LUT_H
#define _LUT_H

#include "general.h"

export void populate_conversion_luts();

export void convert_raw_500mV(sampl_t *, sampl_t *, int);
export void convert_raw_750mV(sampl_t *, sampl_t *, int);
export void convert_raw_raw(sampl_t *, sampl_t *, int);

typedef void (*convertfn)(sampl_t *, sampl_t *, int);

#endif /* _LUT_H */
```

```
#

#include "general.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/mman.h>
#include "mman.h"

/*
 * Useful utility function to ensure pages are pre-faulted.
 */

public void prefault_pages(void *p, int n, int w) {
  int ret = 0;

  while( n-- > 0 ) {
    if( (w&PREFAULT_RDONLY) )                   /* Read page */
      ret = *(int *)p;
    if( (w&PREFAULT_WRONLY) )                   /* Write page */
      *(int *)p = ret;
    p += sysconf(_SC_PAGESIZE);
  }
}

/*
 * Locate a region of memory where one could map a file of size size.
 */

public void *mmap_locate(size_t length, int flags) {
  void *map;

  if( flags & MAL_DOUBLED ) length *= 2;

  map = mmap(NULL, length, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
  if(map == NULL || map == (void *)-1)
    return NULL;

  return map;
}

/*
 * Map and lock a region of a file into memory at given fixed address.
 */

public void *mmap_and_lock_fixed(int fd, off_t offset, size_t length, int flags, void *fixed) {
  void *map;
  int   mflags = 0;

  if( flags&PROT_RDONLY )
    mflags |= PROT_READ;
  if( flags&PROT_WRONLY )
    mflags |= PROT_WRITE;

  if( !mflags )
    mflags = PROT_NONE;

  map = mmap(fixed, length, mflags, MAP_SHARED, fd, offset);
  if(map == NULL || map == (void *)-1)
    return NULL;

  if( (flags&MAL_LOCKED) && mlock(map, length) < 0 ) {
    munmap(map, length);
    return NULL;
  }

  if( flags & PREFAULT_RDWR )
    prefault_pages(map, length / sysconf(_SC_PAGESIZE), (flags & PREFAULT_RDWR));

  return map;
}

/*
```

```
 * Map and lock a region of a file into memory, don’t care where...
 */

public void *mmap_and_lock(int fd, off_t offset, size_t length, int flags) {
  void *map;
  void *ms;

  map = mmap_locate(length, flags);
  if( !map )
    return NULL;

  if( mmap_and_lock_fixed(fd, offset, length, flags, map) == NULL )
    return NULL;

  if( flags & MAL_DOUBLED ) {
    if( mmap_and_lock_fixed(fd, offset, length, flags, map+length) == NULL ) {
      munmap(map, length);
      return NULL;
    }
    length *= 2;
  }

  if( flags & PREFAULT_RDWR )
    prefault_pages(map, length / sysconf(_SC_PAGESIZE), (flags & PREFAULT_RDWR));

  return map;
}
```

```
#

#ifndef _MMAN_H
#define _MMAN_H

#include "general.h"

/* Memory mapping and locking utilities */

export void  prefault_pages(void *, int, int);
export void *mmap_locate(size_t, int);
export void *mmap_and_lock_fixed(int, off_t, size_t, int, void *);
export void *mmap_and_lock(int, off_t, size_t, int);

#define PROT_RDONLY     1
#define PROT_WRONLY     2
#define PROT_RDWR       (PROT_RDONLY|PROT_WRONLY)

#define PREFAULT_RDONLY 4
#define PREFAULT_WRONLY 8
#define PREFAULT_RDWR   (PREFAULT_RDONLY|PREFAULT_WRONLY)

#define MAL_LOCKED      16
#define MAL_DOUBLED     32

#endif /* _MMAN_H */
```

```
#

#include "general.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "assert.h"
#include <ctype.h>
#include "argtab.h"
#include "param.h"

/*
 * Parameter types, represented by constant strings...
 */

PARAM_TYPE_DECL(bool,   int,       "%d",   "%d");
PARAM_TYPE_DECL(int16,  uint16_t, "%hi",  "%hu");
PARAM_TYPE_DECL(int32,  uint32_t, "%li",  "%u");
PARAM_TYPE_DECL(int64,  uint64_t, "%Li",  "%llu");
PARAM_TYPE_DECL(double, double,   "%lg",   "%g");
PARAM_TYPE_DECL(string, char *,   NULL,   "%s");

/*
 * Reset the str and val pointers in a param_t structure.  Free strings
 * as needed, but assume that a string val that is dynamic is dealt with
 * by the caller.
 */

public void reset_param(param_t *p) {
  if(p->p_type == PARAM_TYPE(string)) { /* Special case of dynamic string in two places */
    if( p->p_val && *(char **)p->p_val == p->p_str) { /* String copy is in both places */
      p->p_dyn = 0;                                  /* Ignore dynamic:  caller is responsible */
    }
  }
  if(p->p_dyn)
    free( (void *)p->p_str );
  p->p_str = NULL;
  p->p_dyn = 0;
  if(p->p_val)
    p->p_val = NULL;
}

/*
 * Set the val pointer for a param.
 */

public void setval_param(param_t *p, void **val) {
  p->p_val = val;
}

/*
 * Push an extra value for the given parameter onto its value stack,
 * if there is room.  If p_dyn is set, the last (top) value is an
 * allocated copy; free it and overwrite the slot.  Otherwise the
 * value is a permanent buffer, so just push.
 */

public int set_param_value(param_t *p, char *v) {
  /*  fprintf(stderr, "Pushing value %s\n", v); */
  if( !p->p_source ) {
    errno = EPERM;
    return -1;
  }
  if( p->p_dyn && p->p_str ) {
    if(p->p_val && *(const char **)p->p_val == p->p_str)
      *(const char **)p->p_val = NULL;
    free((void *)p->p_str);
    p->p_str = NULL;
  }
  p->p_str = v;
  return 0;
}
```

```
/*
 * Locate the parameter descriptor in the ps array for the named
 * parameter, if it exists.
 */

public param_t *find_param_by_name(const char *name, int sz, param_t ps[], int nps) {
  int i;

  /*  fprintf(stderr, "Looking for name "); fwrite(name, sz, 1, stderr); fprintf(stderr, "\n"); */
  errno = 0;
  for(i=0; i<nps; i++)
    if( !strncmp(name, ps[i].p_name, sz) )
      return &ps[i];
  errno = EBADSLT;
  return NULL;
}

/*
 * Scan the parameters in the ps array and check whether environment
 * variables provide values for any of them.  The environment variable
 * name must match the parameter using case insensitive matching.  The
 * environment variable’s value replaces the parameter’s value, if any.
 */

public int set_param_from_env(char *env[], param_t ps[], int nps) {
  int i;

  if( !env )
    return 0;

  for(i=0; i<nps; i++) {
    char **e, *p;
    int   sz;

    if( !(ps[i].p_source & PARAM_SRC_ENV) )             /* Only look for params with environment source */
      continue;
    for(e=env; p=*e; e++) {
      for(sz=0; *p && *p != ’=’; p++, sz++);    /* Find the = */
      if( !strncasecmp(ps[i].p_name, *e, sz) ) {        /* Unless true, it’s not this one */
        if( ps[i].p_name[sz] )                  /* If true, there is more name left over:  not this one */
          continue;
        if( set_param_value(&ps[i], (*p ? p+1 : p)) < 0 )
          return −1;
      }
    }
  }
  return 0;
}

/*
 * Read the presented string and look for Name=Value where Name is the
 * name of a parameter.  If found, push a pointer to the value.  The
 * cmd string is assumed NUL−terminated after the value.
 */

public int set_param_from_cmd(char *cmd, param_t ps[], int nps) {
  char *s;
  param_t *p;

  /*  fprintf(stderr, "Working on cmd %s\n", cmd); */
  if( !cmd )
    return 0;
  for(s=cmd; *s && *s != ’=’; s++);
  p = find_param_by_name(cmd, s−cmd, ps, nps);
  if( !p )
    return −1;
  if( !(p−>p_source & PARAM_SRC_CMD) ) {
    errno = EPERM;
    return −1;
  }
  if( !*s++ ) {           /* If *s non−zero, step over the = */
    errno = EINVAL;       /* Name=Value string has no ’=Value’ part */
    return −1;
  }
  return set_param_value(p, s);
```

```c
}

/*
 * Given a string comprising a set of space/comma/semicolon separated
 * Name=Value pairs, instantiate parameters from them.  Use strtok_r
 * to parse the string, which alters the input string by replacing
 * separators with NUL characters.  Each string returned by strtok_r
 * is a single NUL−terminated Name=Value element.  On error, return
 * the negative of the position in the string of the current token
 * start.
 */

private int do_set_params_from_string(char *str, int opt, param_t ps[], int nps) {
  char *save;
  char *cur;
  int   done;
  int   ret;

  /* Initialise the strtok_r scan: skip to space */
  cur = strtok_r(str, " \t", &save);
  if( cur == NULL ) {
    errno = EBADMSG;
    return opt? 0 : −1;         /* If parameters are optional, may succeed here for empty */
  }

  /* First parameter Name=Value should come next */
  done = 0;
  while( (cur = strtok_r(NULL, " \t,;", &save)) != NULL ) {
    if( !isalpha(*cur) ) {
      errno = EBADMSG;
      return str−cur;
    }
    ret = set_param_from_cmd(cur, ps, nps);
    if( ret < 0 )
      return str−cur;
    done++;
  }
  return (done || opt)? 0 : −1;
}

/* Parameters are compulasory */

public int set_params_from_string(char *str, param_t ps[], int nps) {
  return do_set_params_from_string(str, 0, ps, nps);
}

/* String may be empty of parameters */

public int set_opt_params_from_string(char *str, param_t ps[], int nps) {
  return do_set_params_from_string(str, 1, ps, nps);
}

/*
 * Assign a parameter value, i.e. parse its string value and write the result to
 * the location pointed to by the val pointer, which must be of the correct kind.
 */

public int assign_param(param_t *p) {
  if(p == NULL) {
    errno = EINVAL;
    return −1;
  }
  if( !p−>p_val ) {             /* Nowhere to put value */
    errno = EFAULT;
    return −1;
  }

  param_type *pt = p−>p_type;
  if(pt == PARAM_TYPE(bool)) {  /* Special cases for booleans */
    const char *s = p−>p_str;   /* May be NULL, for a boolean (== false) */

    if( !*s || !strncasecmp(s, "false", 6) || !strncasecmp(s, "no", 3) || !strncasecmp(s, "off", 4) ) {
      *(int *)p−>p_val = 0;
      return 0;
    }
    if( !strncasecmp(s, "true", 5) || !strncasecmp(s, "yes", 4) || !strncasecmp(s, "on", 3) ) {
      *(int *)p->p_val = 1;
      return 0;
    }
  }

  if( !p->p_str )               /* No value to put anywhere */
    return 0;

  if(pt == PARAM_TYPE(string)) { /* Special case for strings -- no conversion needed */
    *(const char **)p->p_val = p->p_str;
    return 0;
  }

  // fprintf(stderr, "Scan param %s with str %s to %p using %s\n",
  //      p->p_name, p->p_str, p->p_val, pt->t_scan);
  return sscanf(p->p_str, pt->t_scan, p->p_val) == 1? 0 : -1;
}

/*
 * Scan the parameter table and copy out the values present, converting strings to
 * appropriate types and installing them in the external addresses where provided.
 */

public int assign_all_params(param_t ps[], int nps) {
  int n;

  for(n=0; n<nps; n++) {
    param_t *p = &ps[n];

    if(assign_param(p) < 0)
      return -1-n;
  }
  return 0;
}

/*
 * Same as above but only for parameters sourced from commands.
 */

int assign_cmd_params(param_t ps[], int nps) {
  int n;

  for(n=0; n<nps; n++) {
    param_t *p = &ps[n];

    if( p->p_source & PARAM_SRC_CMD ) {
      if(assign_param(p) < 0)
        return -n-1;
    }
  }
  return 0;
}

/*
 * Retrieve the string value of the parameter and store it in the
 * buffer pointed to by vp, which must be suitable to receive it.
 */

public int get_param_str(param_t *p, const char **vp) {
  const char *v = NULL;
  if( !p->p_str  ) {
    if( p->p_type == PARAM_TYPE(bool) ) {
      *vp = "false";
      return 0;
    }
    errno = EINVAL;
    return -1;
  }
  v = p->p_str;         /* The string value for the parameter */
  if( !v ){
    errno = EINVAL;
    return -1;
  }
  *vp = v;
  return 0;
 }

/*
 * Find parameters that match an argxxx structure.  An arg_xxx structure matches a
 * parameter if (one of) its long name(s) matches a parameter name for which an ARG
 * source has been set.  The long option names are tried in order; only one may match!
 */

private param_t *arg_param_match(const char *a, param_t ps[], int nps) {
  param_t *p, *ret;
  const char *ap;

  if(a == NULL)                 /* There are no long option names */
    return NULL;
  ret = NULL;
  ap = a;                       /* First option name starts here */
  for(ap=a; *a; a=ap) {

    while(*ap && *ap != ',') ap++; /* Skip to end of (first) option name */

    p = find_param_by_name(a, ap-a, ps, nps);
    if(p == NULL)               /* No match for that name */
      continue;

    if(ret && ret != p) {       /* Multiple matches! */
      errno = EBADSLT;
      return NULL;
    }

    ret = p;                    /* At least one match found */
    if(*ap == ',') ap++;        /* Skip a comma, if more to come */
  }
  return ret;
}


/* ASSUME that the count and 'data' values in every argxxx follow the hdr directly */
/* IF THAT IS TRUE, then we can use the ->count and ->data members of ANY arg_xxx */
#define ARG_COUNT(a)    (((struct arg_int *)(a))->count)
#define ARG_DATA(a)     ((void *)((struct arg_int *)(a))->ival)

/*
 * Install defaults into an argtable from matching parameter structures.  This assumes
 * internal knowledge of the arg_hdr structures to determine the relevant parameter
 * structure etc. to use.  The parameter's string value is converted using the arg_hdr
 * structure's scan function and is pre-installed in the arg_xxx structure, which is then
 * reset.
 */

public int arg_defaults_from_params(void **argtable, int nargs, param_t ps[], int nps) {
  struct arg_hdr **ate = (struct arg_hdr **)&argtable[nargs-1]; /* The arg_end structure slot */
  param_t *endp = &ps[nps];

  if( !((*ate)->flag & ARG_TERMINATOR) ) {
    errno = EINVAL;
    return -1;
  }

  struct arg_hdr **atp;
  for(atp=(struct arg_hdr **)argtable; atp<ate; atp++) {
    struct arg_hdr *a = *atp;

    param_t *p = arg_param_match(a->longopts, ps, nps);

    if(p == NULL)               /* No parameter matched this argument */
      continue;
    /* Must be an ARG parameter, or coding problem */
    assertv( (p->p_source & PARAM_SRC_ARG), "Param %s not ARG sourced\n", p->p_name );
    if( !p->p_str )             /* No string value, no default */
      continue;

    //    fprintf(stderr, "Found parameter %s with addr %p, str %s\n", p->p_name, p->p_val, p->p_str);

    /* Copy the parameter's value to the arg structure -- fake an argument parse */
    (*a->resetfn)(a->parent);   /* Reset the counter;  init the structure */
    int ret = (*a->scanfn)(a->parent, p->p_str);
    /* Else value compatibility error:  abort */
    assertv(ret == 0, "Param %s str %s does not pass arg scanfn\n", p->p_name, p->p_str);
    ret = (*a->checkfn)(a->parent);
    /* Else value compatibility error:  abort */
    assertv(ret == 0, "Param %s str %s fails arg checkfn\n", p->p_name, p->p_str);
    ARG_COUNT(a) = 0;           /* This was a default */
  }

  return 0;
}

/*
 * Copy the results from a parsed argtable back to the locations pointed to by their
 * matching param structures; the matching structures are determined as for the
 * arg_defaults_from_params() routine above.  Unfortunately, there is no way to know what
 * kind of value the argxxx structure describes -- we have to assume that the parameter
 * knows the type (and therefore the size to copy).
 *
 * We also copy the value back into the parameter string form -- this might entail some loss
 * of precision for real number values.
 */

public int arg_results_to_params(void **argtable, param_t ps[], int nps) {
  struct arg_hdr **ate =  (struct arg_hdr **)argtable;
  param_t *endp = &ps[nps];

  while( (*ate) && !((*ate)->flag & ARG_TERMINATOR) ) ate++; /* Find the end */
  if( !(*ate) || !((*ate)->flag & ARG_TERMINATOR) ) {
    errno = EINVAL;
    return -1;
  }

  struct arg_hdr **atp;
  for(atp=(struct arg_hdr **)argtable; atp<ate; atp++) {
    struct arg_hdr *a = *atp;
    void *av;

    if( ARG_COUNT(a) == 0 )     /* There is no command-line argument value */
      continue;

    param_t *p = arg_param_match(a->longopts, ps, nps);

    if(p == NULL)               /* No parameter matched this argument */
      continue;
    if( !p->p_val )             /* Nowhere to put the value */
      continue;

    av = ARG_DATA(a) + (ARG_COUNT(a)-1)*p->p_type->t_size;
    memcpy(p->p_val, av, p->p_type->t_size);

    /*
     * This one copy back is tricky...  If *p->p_val is not already the
     * same as p->p_str, it must have come from a static string from
     * argument or environment, since only the assign code changes the
     * val content and it copies the str.  Therefore we copy back the
     * val content and turn off the free-it flag.
     *
     * On the other hand, if p->p_str is in fact *p->p_val, there is
     * nothing further to do.
     */
    if(p->p_type == PARAM_TYPE(string)) {
      const char *v = *(char **)p->p_val;
      if(v != p->p_str) {
        if(p->p_dyn)
          free((void *)p->p_str);
        p->p_dyn = 0;
        p->p_str = v;
      }
      continue;                 /* We are done, in this case */
    }

    if(p->p_dyn)                /* Free the old str value if necessary */
      free((void *)p->p_str);
    int ret = param_value_to_string(p, &p->p_str);
    p->p_dyn = 1;               /* The new value is a dynamic string */
    assertv(ret >=0, "Update of parameter %s str from val for arg %d failed\n",
            p->p_name, ate-atp+1);
  }
  return 0;
}

#undef ARG_COUNT
#undef ARG_DATA

/*
 * Generate part of a usage message based on the parameter structures
 */

public void param_option_usage(FILE *f, int spc, param_t ps[], int nps) {
  int i;
  char *buf = malloc(spc+1);

  for(i=0; i<spc; i++) buf[i] = ' ';
  buf[spc] = '\0';
  for(i=0; i<nps; i++) {
    param_t *p = &ps[i];
    if( !(p->p_source & PARAM_SRC_ARG) )
      continue;
    fprintf(f, "%s--%s <%s> : %s\n", buf, p->p_name, p->p_type, p->p_gloss);
  }
}

public void param_brief_usage(char *buf, int sz, param_t ps[], int nps) {
  int i,
    used = 0,
    rest = sz-1;

  for(i=0; i<nps && rest > 0; i++) {
    param_t *p = &ps[i];
    char *type = "NULL";
    int n;
    if( !(p->p_source & PARAM_SRC_ARG) )
      continue;
    n = snprintf(&buf[used], rest, "[ --%s <%s> ]", p->p_name, p->p_type);
    used += n;
    rest -= n;
  }
  buf[used] ='\0';
}

/*
 * Convert a parameter value and store in a dynamically allocated string
 */

#define LOCALBUF_SIZE 64

public int param_value_to_string(param_t *p, const char **s) {
  param_type *pt = p->p_type;
  char buf[LOCALBUF_SIZE];
  int  used = 0;

  if( !p->p_val )               /* Nowhere to get the value from */
    return 0;

  /* These cases are systematically treatable */
  if(pt == PARAM_TYPE(string)) {
    used = snprintf(&buf[0], LOCALBUF_SIZE-1, pt->t_show, *(char **)p->p_val);
  }
  if(pt == PARAM_TYPE(bool)) {
    used = snprintf(&buf[0], LOCALBUF_SIZE-1, pt->t_show, *(int *)p->p_val);
  }
  if(pt == PARAM_TYPE(int16)) {
    used = snprintf(&buf[0], LOCALBUF_SIZE-1, pt->t_show, *(uint16_t *)p->p_val);
  }
  if(pt == PARAM_TYPE(int32)) {
    used = snprintf(&buf[0], LOCALBUF_SIZE-1, pt->t_show, *(uint32_t *)p->p_val);
  }
  if(pt == PARAM_TYPE(int64)) {
    used = snprintf(&buf[0], LOCALBUF_SIZE-1, pt->t_show, *(uint64_t *)p->p_val);
  }
  if(pt == PARAM_TYPE(double)) {
    used = snprintf(&buf[0], LOCALBUF_SIZE-1, pt->t_show, *(double *)p->p_val);
  }
  if( !(*s = strndup(&buf[0], used)) )
    return -1;

  return used;
}

public void debug_params(FILE *fp, param_t ps[], int nps) {
  int i;

  for(i=0; i<nps; i++) {
    param_t *p = &ps[i];
    param_type *pt = p->p_type;

    fprintf(fp, "Parameter ’%s’: type %s addr %p str %p=’%s’",
            p->p_name, pt->t_name, p->p_val, p->p_str, p->p_str);
    if(p->p_val) {
      fprintf(fp, " val ’");
      if(pt == PARAM_TYPE(bool)) {
        fprintf(fp, pt->t_show, *(int *)p->p_val);
      }
      if(pt == PARAM_TYPE(int16)) {
        fprintf(fp, pt->t_show, *(uint16_t *)p->p_val);
      }
      if(pt == PARAM_TYPE(int32)) {
        fprintf(fp, pt->t_show, *(uint32_t *)p->p_val);
      }
      if(pt == PARAM_TYPE(int64)) {
        fprintf(fp, pt->t_show, *(uint64_t *)p->p_val);
      }
      if(pt == PARAM_TYPE(string)) {
        fprintf(fp, pt->t_show, *(char **)p->p_val);
      }
      if(pt == PARAM_TYPE(double)) {
        fprintf(fp, pt->t_show, *(double *)p->p_val);
      }
    }
    fprintf(fp, "’\n");
  }
}
```

```
#

#ifndef _PARAM_H
#define _PARAM_H

#include <stdio.h>
#include <stdint.h>

#include "general.h"

typedef const struct {
  const char *t_name;
  int         t_size;
  const char *t_scan;
  const char *t_show;
}
  param_type;

#define PARAM_TYPE(name) param_type_ ## name
#define PARAM_TYPE_DECL(name,size,scan,show) param_type PARAM_TYPE(name)[] = { "<" #name ">" , sizeof(size), scan, show, }
#define PARAM_TYPE_EXPORT(name) export param_type PARAM_TYPE(name)[];

PARAM_TYPE_EXPORT(bool);
PARAM_TYPE_EXPORT(int16);
PARAM_TYPE_EXPORT(int32);
PARAM_TYPE_EXPORT(int64);
PARAM_TYPE_EXPORT(double);
PARAM_TYPE_EXPORT(string);

typedef struct
{ const char    *p_name;                /* Name of this parameter */
  const char    *p_str;                 /* String value for this parameter */
  void          *p_val;                 /* Location where value is to be stored */
  param_type    *p_type;                /* Type of the parameter, for value conversion */
  int            p_source;              /* Possible sources of the values */
  const char    *p_gloss;               /* Explanation of this parameter */
  int            p_dyn;                 /* If true, free and replace str on push */
}
  param_t;

#define PARAM_SRC_ENV   0x1
#define PARAM_SRC_ARG   0x2
#define PARAM_SRC_CMD   0x4

export int set_param_value(param_t *, char *);
export param_t *find_param_by_name(const char *, int, param_t [], int);
export int set_param_from_env(char *[], param_t [], int);
export int set_params_from_string(char *, param_t [], int);
export int set_opt_params_from_string(char *, param_t [], int);
export int get_param_str(param_t *, const char **);
// export void param_brief_usage(char *, int, param_t [], int);
// export void param_option_usage(FILE *, int, param_t [], int);
// export const char *pop_param_value(param_t *);
export void reset_param(param_t *);
export void setval_param(param_t *, void **);
export int assign_param(param_t *);
export int assign_all_params(param_t *, int);
export int assign_cmd_params(param_t *, int);
export int param_value_to_string(param_t *, const char **);
export int arg_defaults_from_params(void **, int, param_t [], int);
export int arg_results_to_params(void **, param_t [], int);
export void debug_params(FILE *, param_t [], int);

#endif /* _PARAM_H */
```

```
#

#include "general.h"

#include <stdlib.h>
#include "assert.h"
#include "queue.h"

/*
 * Implements a doubly-linked queue in ring form.
 *
 * Invariant:  every q structure is doubly-linked;  new structures are singletons.
 */

public queue *init_queue(queue *p) {
  if( p == NULL ) {
    p = (queue *)calloc(1, sizeof(queue));
    assertv(p != NULL, "Queue alocation failure\n");
  }
  p->q_next = p->q_prev = p;
  return p;
}

/*
 * Remove p from its queue and make it a singleton.  You cannot detach
 * a singleton from its queue.
 */

public queue *de_queue(queue *p) {
  if( p->q_next == p )
    return NULL;
  p->q_prev->q_next = p->q_next;
  p->q_next->q_prev = p->q_prev;
  p->q_next = p->q_prev = p;
  return p;
}

/*
 * Splice q and p together so that p immediately follows q and the
 * next and prev chains continue in the correct senses
 */

public queue *splice_queue(queue *q, queue *p) {
  queue *qn, *pp;

  qn = q->q_next;
  q->q_next = p;
  pp = p->q_prev;
  p->q_prev = q;
  qn->q_prev = pp;
  pp->q_next = qn;
  return q;
}

/*
 * Unsplice a queue: cut the ring at start and end and relink.  Also
 * join start and end.
 */

public queue *unsplice_queue(queue *start, queue *end) {
}

/*
 * Apply a function to each queue member in [start,end).  The function
 * is called with arg as its first argument and the queue structure
 * pointer as its second.  The first function, map_queue_nxt,
 * traverses the segment "forward" while the second goes "backward".
 *
 * If start == end or end is not in the list (e.g. end is NULL) the
 * functions traverse the whole list visiting each node exactly once.
 */

public void map_queue_nxt(queue *start, queue *end, void (*fn)(void *, queue *), void *arg) {

  for_nxt_in_Q(queue *p, start, end)
```

```
    (*fn)(arg, p);
  end_for_nxt;
}

public void map_queue_prv(queue *start, queue *end, void (*fn)(void *, queue *), void *arg) {

  for_prv_in_Q(queue *p, start, end)
    (*fn)(arg, p);
  end_for_prv;
}
```

```
#

#ifndef _QUEUE_H
#define _QUEUE_H

#include "general.h"

typedef struct q
{
  struct q   *q_next;
  struct q   *q_prev;
}
  queue;

export queue *de_queue(queue *);
export queue *init_queue(queue *);
export queue *splice_queue(queue *,queue *);
export queue *unsplice_queue(queue *, queue *);
export void   map_queue_nxt(queue *, queue *, void (*)(void *, queue *), void *);
export void   map_queue_prv(queue *, queue *, void (*)(void *, queue *), void *);

#define queue_next(q)   ((q)->q_next)
#define queue_prev(q)   ((q)->q_prev)

#define queue_ins_after(q,i)  splice_queue((q), (i))
#define queue_ins_before(q,i)  splice_queue((i), (q))

#define queue_singleton(q)  ((q)->q_next == (q) && (q)->q_prev == (q))

#define QUEUE_HEADER(name)  queue name = { &name, &name }

/*
 * These macro definitions do essentially the same as the
 * map_queue_nxt and map_queue_prv but they don’t leave the current
 * local scope −− so for instance one can break the loop early in this
 * form whereas one cannot in the (default) map function.
 *
 * The var argument is a variable that will hold the current node
 * pointer as the loop proceeds.  It can be declared locally to the
 * for_nxt by including its declaration in the macro call:
 *
 * for_nxt_in_Q(queue *ptr,start,end) ...
 *
 * or it can be a variable declared outside the scope of the for_nxt
 * in which case just its name is given as argument and it will
 * persist after the map−loop ends.
 *
 * The macros evaluate start and end exactly once and execute the User
 * Code once for each list element in the range [start,end) with var
 * set to that element.  If start==end or end is not actually in the
 * list, the loop traverses the whole list exactly once visiting each
 * node exactly once.
 *
 * Note that it is also possible to remove node __p during the USER
 * CODE because it is neither the node we are about to work on nor the
 * end point node.  It may be the start node, however: the user should
 * deal with that case!
 */

#define for_nxt_in_Q(var,start,end)                      \
do { queue *__s = (start), *__e = (end);                 \
     queue *__p = __s;                                   \
     int    __done = 0;                                  \
     while(!__done) { queue *__n = queue_next(__p);      \
       __done = (__n == __s || __n == __e);              \
       var = __p;  __p = __n;                            \
       /* USER CODE GOES HERE */

#define end_for_nxt  \
     } } while(0)

#define for_prv_in_Q(var,start,end)                      \
do { queue *__s = (start), *__e = (end);                 \
     queue *__p = __s;                                   \
     int    __done = 0;                                  \
     while(!__done) { queue *__n = queue_prev(__p);     \
       __done = (__n == __s || __n == __e);             \
       var = __p;  __p = __n;
       /* USER CODE GOES HERE */

#define end_for_prv  \
     } } while(0)

#endif /* _QUEUE_H */
```

```
#

#include "general.h"

#include <stdio.h>
#include <stdlib.h>
#include "assert.h"
#include <time.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/capability.h>

#include <zmq.h>
#include <pthread.h>

#include <comedi.h>
#include <comedilib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#include "util.h"
#include "param.h"
#include "queue.h"
#include "strbuf.h"
#include "mman.h"
#include "chunk.h"
#include "adc.h"
#include "snapshot.h"
#include "tidy.h"
#include "reader.h"
#include "writer.h"

/*
 * READER global data structures
 */

public rparams reader_parameters;   /* The externally−visible parameters for the reader thread */
public adc reader_adc;              /* The ADC object for the READER */

/*
 * READER state machine definitions.
 *
 * The READER state is kept in the rp_state variable, private to the
 * READER thread.
 *
 * ERROR state: this occurs when a serious error happens, normally due
 *   to bad parameters.  One can leave ERROR state using the Param
 *   command.
 *
 * PARAM state: results from initialisation by the main thread routine
 *   and after the receipt of a Param command, because of the activity
 *   of the verify function.  Failure of parameters to verify sends us
 *   to ERROR state.  Successful verification also results in the
 *   creation and parameterisation of an ADC object.
 *
 * RESTING state: a successful execution of the Init command leaves us
 *   in RESTING state.  In this state, an initialised ADC object is
 *   available.  Errors in parameter verfication or instantiation of
 *   the initialised ADC object put us into ERROR state.
 *
 * ARMED state: executing the Go command initiates a data transfer and
 *   moves the READER to this state.  We stay in ARMED state until the
 *   first data has been seen (i.e. the ADC object has changed from
 *   running to running and live).  Failure of data to arrive within a
 *   reasonable time causes an automatic transition to the ERROR
 *   state, with the same cleanup as done by the Halt command, which
 *   may be issued in this or the RUN state.
 *
 * RUN state: automatic transition from ARMED on receipt of the first
 *   data.  In ARMED and RUN state the Halt command will terminate
 *   data acqusition and return the READER to ERROR state (as a
 *   special case; the parameters are valid, but after Halt there is
 *   no ADC object).
 *
 * The Quit command issued in any state will cause the READER to shut
 * down cleanly.
 *
 * The WRITER will reject Snap commands unless the READER is in ARMED
 * or RUN state (in fact, unless the ADC object exists and reports
 * itself as running).
 */

private int rp_state;

#define READER_ERROR    0       /* An error occurred, base start state */
#define READER_PARAM    1       /* There are parameters that need to be verified */
#define READER_RESTING  2       /* READER is ready, Comedi and mmap setup has been done */
#define READER_ARMED    3       /* The ADC has been started */
#define READER_RUN      4       /* Data from the ADC has been seen in the buffers */

/*
 * READER forward definitions
 */

private void drain_reader_chunk_queue();

/*
 * READER thread comms initialisation.
 * Called after the context is created.
 */

private void *writer;
private void *tidy;
private void *log;
private void *command;

private void create_reader_comms() {
  import void *snapshot_zmq_ctx;
  /* Create necessary sockets */
  command  = zh_bind_new_socket(snapshot_zmq_ctx, ZMQ_REP, READER_CMD_ADDR);    /* Receive commands */
  assertv(command != NULL, "Failed to instantiate reader command socket\n");
  log      = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_PUSH, LOG_SOCKET);  /* Socket for log messages */
  assertv(log != NULL, "Failed to instantiate reader log socket\n");
  writer = zh_bind_new_socket(snapshot_zmq_ctx, ZMQ_PAIR, READER_QUEUE_ADDR);
  assertv(writer != NULL, "Failed to instantiate reader queue socket\n");
  tidy     = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_PAIR, TIDY_SOCKET);  /* Socket to TIDY thread */
  assertv(tidy != NULL, "Failed to instantiate reader−>tidy socket\n");
}

/* Close everything created above. */

private void close_reader_comms() {
  zmq_close(command);
  zmq_close(log);
  zmq_close(writer);
  zmq_close(tidy);
}

/*
 * Copy the necessary capabilities from permitted to effective set (failure is fatal).
 *
 * The READER needs:
 *
 * CAP_IPC_LOCK −− ability to mmap and mlock pages.
 * CAP_SYS_NICE −− ability to set RT scheduling priorities
 *
 * These capabilities should be in the CAP_PERMITTED set, but not in CAP_EFFECTIVE which was cleared
 * when the main thread dropped privileges by changing to the desired non−root uid/gid.
 */

private int set_up_reader_capability() {
  cap_t c = cap_get_proc();
  const cap_value_t vs[] = { CAP_IPC_LOCK, CAP_SYS_NICE, };

  cap_set_flag(c, CAP_EFFECTIVE, sizeof(vs)/sizeof(cap_value_t), &vs[0], CAP_SET);
  return cap_set_proc(c);
}
```

```
/*
 * Get a value from the monotonic krnel clock and express in nanoseconds.
 */

public uint64_t  monotonic_ns_clock() {
  uint64_t ret;
  struct timespec now;

  clock_gettime(CLOCK_MONOTONIC, &now);         /* Timestamp for debugging */
  ret = now.tv_sec;
  ret = ret*1000000000 + now.tv_nsec;
  return ret;
}

/*
 * Process a READER command from MAIN thread.  Generate replies as necessary.
 * Returns true if processing messages should continue..
 */

private int process_reader_command(void *s) {
  rparams *rp = &reader_parameters;
  int      used;
  int      ret;
  strbuf   cmd;
  char    *cmd_buf;
  strbuf   err;

  used = zh_get_msg(s, 0, sizeof(strbuf), &cmd);
  if( !used ) {                 /* It was a quit message */
    if(rp_state == READER_ARMED || rp_state == READER_RUN || rp_state == READER_RESTING)
      adc_stop_data_transfer(reader_adc);
    return false;
  }

  cmd_buf = strbuf_string(cmd);
  err = strbuf_next(cmd);

  if(verbose > 1)
    zh_put_multi(log, 3, "READER cmd: ’", &cmd_buf[0], "’");

  ret = 0;
  switch(cmd_buf[0]) {
  case ’p’:
  case ’P’:
    if( rp_state != READER_PARAM && rp_state != READER_RESTING && rp_state != READER_ERROR ) {
      strbuf_printf(err, "NO: Param issued but not in PARAM, RESTING or ERROR state");
      ret = −1;
      break;
    }
    ret = set_params_from_string(&cmd_buf[0], globals, n_global_params);
    if( ret < 0 ) {
      strbuf_printf(err, "NO: Param −− parse error at position %d", −ret);
      break;
    }
    ret = assign_cmd_params(globals, n_global_params);
    if( ret < 0 ) {
      strbuf_printf(err, "NO: Param −− assign error on param %d: %m", −ret);
      break;
    }

    /* Otherwise, succeeded in updating parameters */
    strbuf_printf(err, "NO: Param −− verify error: ");
    ret = verify_reader_params(&reader_parameters, err);
    if( ret < 0 ) {
      break;
    }
    strbuf_printf(err, "OK Param");
    rp_state = READER_PARAM;
    break;

  case ’i’:
  case ’I’:
    if( rp_state != READER_PARAM ) {
      strbuf_printf(err, "NO: Init issued but not in PARAM state");
      ret = -1;
      break;
    }
    strbuf_printf(err, "NO: Init -- param verify error: ");
    ret = verify_reader_params(&reader_parameters, err);
    if( ret < 0 ) {
      rp_state = READER_ERROR;
      break;
    }
    ret = adc_init(reader_adc, err);
    if( ret < 0 ) {
      rp_state = READER_ERROR;
      break;
    }
    if(verbose > 0) {           /* Borrow the err buffer */
      strbuf_printf(err, "READER Init with dev %s, freq %g [Hz], isp %d [ns] and buf %d [MiB]",
                    rp->r_device, rp->r_frequency, adc_ns_per_sample(reader_adc), rp->r_bufsz);
      zh_put_multi(log, 1, strbuf_string(err));
    }
    strbuf_printf(err, "OK Init -- nchan %d isp %d [ns]", NCHANNELS, adc_ns_per_sample(reader_adc));
    rp_state = READER_RESTING;
    break;

  case 'g':
  case 'G':
    if( rp_state != READER_RESTING ) {
      strbuf_printf(err, "NO: Go issued but not in RESTING state");
      ret = -1;
      break;
    }
    ret = adc_start_data_transfer(reader_adc, err);
    if( ret < 0 ) {
      rp_state = READER_ERROR;
      break;
    }
    strbuf_printf(err, "OK Go");
    rp_state = READER_ARMED;
    break;

  case 'h':
  case 'H':
    if( rp_state != READER_ARMED && rp_state != READER_RUN ) {
      strbuf_printf(err, "NO: Halt issued but not in ARMED or RUN state");
      ret = -1;
      break;
    }
    adc_stop_data_transfer(reader_adc); /* Terminate any transfer in progress */
    drain_reader_chunk_queue();         /* Empty the chunk queue */
    strbuf_printf(err, "OK Halt");
    adc_destroy(reader_adc);
    reader_adc = NULL;
    rp_state = READER_ERROR;
    break;

  default:
    strbuf_printf(err, "NO: READER -- Unexpected reader command");
    ret = -1;
    break;
  }
  if( ret < 0 ) {
    strbuf_revert(cmd);
    zh_put_multi(log, 4, strbuf_string(err), "\n > '", &cmd_buf[0], "'"); /* Error occurred, log it */
  }
  strbuf_clear(cmd);
  zh_put_msg(s, 0, sizeof(strbuf), (void *)&err); /* return message */
  return true;
}

/*
 * Set the READER thread to real-time priority, if RTPRIO is set...
 */

public int set_reader_rt_scheduling() {

  if( reader_parameters.r_schedprio > 0 ) {     /* Then there is RT priority scheduling to set up */
```

```
    if( set_rt_scheduling(reader_parameters.r_schedprio) < 0 )
      return -1;

    /* Successfully applied RT scheduling */
    return 1;
  }

  /* RT scheduling not applicable:  no RTPRIO set */
  return 0;
}


/*
 * Handle a message from the WRITER.  The message will be a chunk
 * which is ready to add to the READER's pending-work queue.  Chunks
 * arrive here with a state of SNAPSHOT_WAITING or SNAPSHOT_ERROR (if
 * they were in transit when an error occurred).  The latter are sent
 * straight back to the WRITER, which is counting down pending chunks
 * to file completion, after their frame has been released.
 */

private QUEUE_HEADER(ReaderChunkQ);
private chunk_t *rq_head = NULL;

private int process_queue_message(void *s) {
  rparams *rp = &reader_parameters;
  chunk_t *c;
  int      ret;

  ret = zh_get_msg(s, 0, sizeof(chunk_t *), (void *)&c);
  assertv(ret==sizeof(chunk_t *), "Received message from WRITER with wrong size %d (not %d)\n", ret, sizeof(chunk_t *));

  if(rp_state != READER_ARMED && rp_state != READER_RUN) {
    strbuf_appendf(c->c_error, "READER thread ADC is not running");
    c->c_status = SNAPSHOT_ERROR;
  }
  else {  /* Check the chunk is still current -- set SNAPSHOT_ERROR state on failure */
    adc_setup_chunk(reader_adc, c);
    if( !c->c_ring )
      c->c_status = SNAPSHOT_ERROR;
  }

  if(c->c_status==SNAPSHOT_ERROR) {             /* we send it straight back */
    ret = zh_put_msg(writer, 0, sizeof(chunk_t *), (void *)&c);
    assertv(ret==sizeof(chunk_t *), "Message returned to WRITER with wrong size %d (not %d)\n", ret, sizeof(chunk_t *));
    ret = zh_put_msg(tidy, 0, sizeof(frame *), &c->c_frame);
    assertv(ret==sizeof(frame *), "Frane message to TIDY with wrong size %d (not %d)\n", ret, sizeof(frame *));
    c->c_frame = NULL;
    return true;
  }

  assertv(c->c_status==SNAPSHOT_WAITING, "Received chunk c:%04hx has unexpected state %s\n", c->c_name, snapshot_status(c->c_status));

  /* Add the chunk to the READER chunk queue in order of increasing *last* sample */
  queue *pos = &ReaderChunkQ;
  if( !queue_singleton(&ReaderChunkQ) ) {
    for_nxt_in_Q(queue *p, queue_next(&ReaderChunkQ), &ReaderChunkQ);
    chunk_t *h = rq2chunk(p);
    if(h->c_last > c->c_last) {
      pos = p;
      break;
    }
    end_for_nxt;
  }
  queue_ins_before(pos, chunk2rq(c));
  if(pos == &ReaderChunkQ) {
    rq_head = c;                /* Points to the chunk at the head of the READER queue, when not NULL */
  }
  return true;
}

/*
 * Abort the chunk which is at the head of the ReaderChunkQ, i.e. it is
 * queue_next(&ReaderChunkQ).  This means we must scan for its
 * siblings in the queue, remove them and set their status to
 * SNAPSHOT_ERROR, and return them to the WRITER.  We assume that the
 * caller has set the c_error strbuf.
 */

private void abort_queue_head_chunk() {
  snapfile_t *parent = rq2chunk(queue_next(&ReaderChunkQ))->c_parent;
  int         ret;

  for_nxt_in_Q(queue *p, queue_next(&ReaderChunkQ), &ReaderChunkQ);
  chunk_t *c = rq2chunk(p);
  if(c->c_parent == parent) {
    de_queue(p);
    c->c_status = SNAPSHOT_ERROR;
    ret = zh_put_msg(tidy, 0, sizeof(frame *), &c->c_frame);
    assertv(ret==sizeof(frame *), "Frane message to TIDY with wrong size %d (not %d)\n", ret, sizeof(frame *));
    c->c_frame = NULL;
    ret = zh_put_msg(writer, 0, sizeof(chunk_t *), (void *)&c);
    assertv(ret==sizeof(chunk_t *), "Abort to WRITER with wrong size %d (not %d)\n", ret, sizeof(chunk_t *));
  }
  end_for_nxt;
  rq_head = queue_singleton(&ReaderChunkQ) ? NULL : rq2chunk(queue_next(&ReaderChunkQ));
}

/*
 * Complete the chunk at the head of the ReaderChunkQ: remove the queue
 * head and compute new head chunk; copy the data for the old head;
 * send the frame to TIDY for release and the chunk pointer back to
 * WRITER for book-keeping.  Before doing this, check we still have
 * the data for the head chunk and if not then abort it.
 */

private void complete_queue_head_chunk() {
  int      ret;
  chunk_t *c = rq_head;

  if(c->c_first < adc_ring_tail(reader_adc)) { /* Oops, we are too late */
    abort_queue_head_chunk();
    return;
  }

  de_queue(chunk2rq(rq_head));
  rq_head = queue_singleton(&ReaderChunkQ) ? NULL : rq2chunk(queue_next(&ReaderChunkQ));

  copy_chunk_data(c);

  ret = zh_put_msg(tidy, 0, sizeof(frame *), &c->c_frame);
  assertv(ret==sizeof(frame *), "Frane message to TIDY with wrong size %d (not %d)\n", ret, sizeof(frame *));
  c->c_frame = NULL;

  ret = zh_put_msg(writer, 0, sizeof(chunk_t *), (void *)&c);
  assertv(ret==sizeof(chunk_t *), "Abort to WRITER with wrong size %d (not %d)\n", ret, sizeof(chunk_t *));
}

/*
 * Drain the READER chunk queue when turning off the data capture.
 * Any snapshots in progress are aborted.
 */

private void drain_reader_chunk_queue() {

  while( !queue_singleton(&ReaderChunkQ) ) {
    chunk_t *c = rq2chunk(queue_next(&ReaderChunkQ));
    strbuf_appendf(c->c_error, "aborted because of READER ADC shutdown");
    abort_queue_head_chunk();
  }
}

/*
 * READER thread message loop
 *
 * The two variables buf_hwm_samples and buf_window_samples are
 * determined by the program parameters window and bufhwm and set the
 * policy for moving the ring buffer tail pointer.  Their values are
 * computed in the parameter verify routine for the READER (see below).
 *
 * Operation is as follows.  The routine waits for incoming messages
 * up to a certain maximum delay; then on each pass through the loop,
 * at least once per delay interval assuming we got some new data, we
 * do two things:
 *
 * - first, try to advance the adc_ring_head position which records
 *   data placed in the ADC ring buffer by Comedi’s ADC driver.  If
 *   the head advances past the last sample index of any chunk we can
 *   write that chunk out, recomputing the next theshold for head.
 *
 * - second, check if the head has passed the ring buffer high-water
 *   mark threshold, which is computed by adding buf_hwm_samples to
 *   the adc_ring_tail value.  If it has, the ring buffer is too full
 *   and we must move the adc_ring_tail using adc_data_purge().  We
 *   advance the tail to (at most) buf_window_samples before the
 *   current head position -- this ensures that we have at least the
 *   specified ’window’ duration in the ring buffer at all times.
 *
 * In the first step, if the first sample index of the chunk is
 * earlier than the current tail, we have been forced to purge data
 * (to avoid buffer overrun in Comedi) before we got the complete
 * chunk.  This can only happen if the chunks are very large compared
 * to the buffer, which should be disallowed by parameter checking.
 *
 * Furthermore, if the main loop is executed for too long without any
 * data being captured, we shut down the ADC and enter error state.
 */

#define ADC_DRY_PERIOD_MAX 1000 /* Initial default is 10 [s] */

private int buf_hwm_samples = 0;
private int buf_window_samples = 0;
private int adc_dry_period_max = ADC_DRY_PERIOD_MAX;
private int reader_poll_delay = 100; /* Poll wait time [ms] */

private void reader_thread_msg_loop() {    /* Read and process messages */
  uint64_t high_water_mark;
  int      adc_dry_period;
  int      ret;
  int      running;

  /* Main loop:  read messages and process messages */
  zmq_pollitem_t  poll_list[] =
    { { writer,  0, ZMQ_POLLIN, 0 },
      { command, 0, ZMQ_POLLIN, 0 },
    };
#define N_POLL_ITEMS    (sizeof(poll_list)/sizeof(zmq_pollitem_t))
  int (*poll_responders[N_POLL_ITEMS])(void *) =
    { process_queue_message,
      process_reader_command,
    };

  zh_put_multi(log, 1, "READER thread is initialised");
  rp_state = READER_PARAM;

  high_water_mark = adc_ring_tail(reader_adc) + buf_hwm_samples;
  adc_dry_period  = adc_dry_period_max;

  reader_parameters.r_running = true;
  running = true;

  while( running && !die_die_die_now ) {
    int ret;
    int nb;
    int delay;
    int n;

    if(reader_adc && adc_is_running(reader_adc)) {
      adc_dry_period--;
      nb = adc_data_collect(reader_adc);
      if( nb ) {                        /* There was some new data, adc_ring_head has advanced */
        adc_dry_period = adc_dry_period_max;
        /* Once the ADC head pointer has advanced past the READER queue head’s end, a chunk is ready */
        while( rq_head && rq_head->c_last <= adc_ring_head(reader_adc) ) {
          complete_queue_head_chunk();
        }

        /* Check buffer fullness;  if necessary, call adc_data_purge to move adc_ring_tail */
        uint64_t head = adc_ring_head(reader_adc);
        if(head > high_water_mark) {
          uint64_t lwm  = head - buf_window_samples;
          uint64_t tail = adc_ring_tail(reader_adc);
          if(lwm > tail) {
            ret = adc_data_purge(reader_adc, (int)(lwm-tail));
            assertv(ret==0, "Comedi mark read failed for %d bytes: %C", (int)(lwm-tail));
            high_water_mark = lwm + buf_hwm_samples;
          }
        }
      }
      if(adc_dry_period <= 0) { /* Data capture interrupted or failed to start... */

      }
    }

    ret = zmq_poll(&poll_list[0], N_POLL_ITEMS, reader_poll_delay);     /* Look for commands here */
    if( ret < 0 && errno == EINTR ) { /* Interrupted */
      zh_put_multi(log, 1, "READER loop interrupted");
      break;
    }
    if(ret < 0)
      break;

    for(n=0; n<N_POLL_ITEMS; n++) {
      if( poll_list[n].revents & ZMQ_POLLIN ) {
        running = running & (*poll_responders[n])(poll_list[n].socket); /* N.B. not && */
      }
    }
  }
}

/*
 * READER thread main routine
 *
 * This loop either waits for a command on the command socket, or
 * loops reading from Comedi.  It aborts if it cannot get the sockets
 * it needs.
 */

public void *reader_main(void *arg) {
  int ret;
  char *thread_msg = "normal exit";

  create_reader_comms();

  if( set_up_reader_capability() < 0 ) {
    zh_put_multi(log, 1, "READER thread capabilities are deficient");
  }

  ret = set_reader_rt_scheduling();
  switch(ret) {
  case 1:
    zh_put_multi(log, 1, "READER RT scheduling succeeded");
    break;
  case 0:
    zh_put_multi(log, 1, "READER using normal scheduling: RTPRIO unset");
    break;
  default:
    zh_put_multi(log, 2, "READER RT scheduling setup failed: ", strerror(errno));
    break;
  }

  struct timespec test_stamp;
  ret = clock_gettime(CLOCK_MONOTONIC, &test_stamp);
  assertv(ret == 0, "Test failed to get monotonic clock time\n");

  reader_thread_msg_loop();
  if(rp_state == READER_ARMED || rp_state == READER_RUN || rp_state == READER_RESTING) {
    adc_stop_data_transfer(reader_adc);
    adc_destroy(reader_adc);
  }
  zh_put_msg(tidy, 0, 0, NULL); /* Tell TIDY thread to finish */

  zh_put_multi(log, 1, "READER thread terminates by return");

  /* Clean up our ZeroMQ sockets */
  close_reader_comms();
  reader_parameters.r_running = false;
  return (void *) thread_msg;
}

/*
 * Verify reader parameters and generate reader state description.
 */

public int verify_reader_params(rparams *rp, strbuf e) {
  import int writer_chunksize_samples();

  if( rp->r_schedprio != 0 ) { /* Check for illegal value */
    int max, min;

    min = sched_get_priority_min(SCHED_FIFO);
    max = sched_get_priority_max(SCHED_FIFO);
    if(rp->r_schedprio < min || rp->r_schedprio > max) {
      strbuf_appendf(e, "RT scheduling priority %d not in kernel's acceptable range [%d,%d]",
                    rp->r_schedprio, min, max);
      return -1;
    }
  }

  if(reader_adc) {
    adc_destroy(reader_adc);
    reader_adc = NULL;
  }
  reader_adc = adc_new(e);

  if( adc_set_chan_frequency(reader_adc, e, &rp->r_frequency) < 0 )
    return -1;

  if(rp->r_window < 1 || rp->r_window > 30) {
    strbuf_appendf(e, "Min. capture window %d seconds outwith compiled-in range [%d,%d] seconds",
                   rp->r_window, 1, 30);
    return -1;
  }

  int pagesize = sysconf(_SC_PAGESIZE)/sizeof(sampl_t);

  /* Compute the size of the desired capture window in samples, rounded up to a full page */
  int rbw_samples = rp->r_window * rp->r_frequency * NCHANNELS;
  rbw_samples = (rbw_samples*sizeof(sampl_t) + pagesize - 1) / pagesize;
  rbw_samples *= pagesize / sizeof(sampl_t);

  if(rp->r_buf_hwm_fraction < 0.5 || rp->r_buf_hwm_fraction > 0.95) {
    strbuf_appendf(e, "Ring buffer high-water mark fraction %g outwith compiled-in range [%g,%g] seconds",
                   rp->r_buf_hwm_fraction, 0.5, 0.95);
    return -1;
  }

  /* Compute ring buffer high-water mark in samples, rounded up to a full page */
  int bhwm_samples = rp->r_buf_hwm_fraction * rp->r_bufsz * 1024 * 1024;
  bhwm_samples = (bhwm_samples + pagesize - 1) / pagesize;
  bhwm_samples = pagesize * bhwm_samples / sizeof(sampl_t);

  if(rbw_samples > bhwm_samples) {
    strbuf_appendf(e, "Capture window of %d [kiB] is bigger than ring buffer high-water mark at %d [kiB]",
                   rbw_samples*sizeof(sampl_t)/1024, bhwm_samples*sizeof(sampl_t)/1024);
    return -1;
  }

  /* Check the window and high-water mark against the chunk size */
  int chunksize = writer_chunksize_samples();
  if(chunksize) {
    if(rbw_samples < chunksize) {
      strbuf_appendf(e, "Capture window of %d [kiB] is smaller than chunk size %d[kiB]",
                     rbw_samples*sizeof(sampl_t)/1024, chunksize*sizeof(sampl_t)/1024);
      return -1;
    }
    if(bhwm_samples+2*chunksize > rp->r_bufsz*1024*1024/sizeof(sampl_t)) {
      strbuf_appendf(e, "Ring overflow region %d [kiB] is smaller than twice the chunk size %d[kiB]",
                     (rp->r_bufsz*1024*1024-bhwm_samples*sizeof(sampl_t))/1024, chunksize*sizeof(sampl_t)/1024);
      return -1;
    }
  }

  if( adc_set_bufsz(reader_adc, e, rp->r_bufsz) < 0 )
    return -1;

  if( adc_set_range(reader_adc, e, rp->r_range) < 0 )
    return -1;

  adc_set_device(reader_adc, rp->r_device); /* Record the path, don't open the device */

  /* Determine the READER main loop poll delay from the chunk duration */
  double d = 1e-6 * chunksize * adc_ns_per_sample(reader_adc); /* Length of a chunk in [ms] */
  reader_poll_delay = ((d+2.5)/5 > 100)? 100.0 : (d+2.5)/5; /* One fifth of a chunk or 100[ms] */

  /* Set the tail policy variables */
  buf_hwm_samples = bhwm_samples;
  buf_window_samples = rbw_samples;

  rp_state = READER_PARAM;
  return 0;
}
```

```
#

#include "general.h"

/*
 * The ZMQ addresses for the reader thread
 */

#define READER_CMD_ADDR "inproc://Reader−CMD"
#define READER_QUEUE_ADDR "inproc://Reader−Q"

/*
 * Reader parameter structure.
 */

typedef struct {
  double      r_frequency;         /* Per−channel sampling frequency [Hz] */
  int         r_schedprio;         /* Reader real−time priority */
  int         r_bufsz;             /* Reader buffer size [MiB] */
  int         r_range;             /* ADC full−scale range [mV] */
  double      r_window;            /* Snapshot window [s] (must fit in buffer) */
  double      r_buf_hwm_fraction;  /* Ring buffer high−water mark as fraction of size */
  const char *r_device;            /* Comedi device to use */
  int         r_running;           /* Thread is running and ready */
}
  rparams;

export rparams   reader_parameters;

export int       verify_reader_params(rparams *, strbuf);
export void     *reader_main(void *);
export uint64_t  monotonic_ns_clock();
```

```
#

#include "general.h"

#define __GNU_SOURCE

#include <syscall.h>
#include <sys/capability.h>
#include <assert.h>
#include <pthread.h>
#include <errno.h>

#include "rtprio.h"

#ifdef __GNU_SOURCE

#define  gettid()       (syscall(SYS_gettid)) /* No glibc interface, Linux−only call */

/*
 * Routine(s) for establishing threads in RT FIFO scheduling mode using Linux tricks
 */

public int set_rt_scheduling(int p) {
  pid_t  me = gettid();
  struct sched_param pri;
  int    mode;

  /* Attempt the operation */
  pri.sched_priority = p;
  if( sched_setscheduler(me, SCHED_FIFO, &pri) < 0 ) {
    return −1;          /* Failed for some reason */
  }

  /* Verify the operation */
  mode = sched_getscheduler(me);
  if( mode != SCHED_FIFO ) {    /* Didn’t work, despite no errors... */
    errno = ENOSYS;
    return −1;
  }

  pri.sched_priority = −1;      /* Check correct priority was set... */
  if( sched_getparam(me, &pri) < 0
      || pri.sched_priority != p ) {
    errno = ENOSYS;
    return −1;
  }

  /* Successfully applied RT scheduling */
  return 0;
}

#else

/*
 * Routine(s) for establishing threads in RT FIFO scheduling mode using POSIX calls
 */

public int set_rt_scheduling(int p) {
  pthread_t me = pthread_self();
  struct sched_param pri;
  int mode;

  /* Attempt the operation */
  pri.sched_priority = p;
  if( pthread_setschedparam(me, SCHED_FIFO, &pri) < 0 ) {
    return −1;          /* Failed for some reason */
  }

  /* Verify the operation */
  pri.sched_priority = −1;
  if( pthread_getschedparam(me, &mode, &pri) < 0
      || mode != SCHED_FIFO
      || pri.sched_priority != p ) {    /* Didn’t work, despite no errors... */
    errno = ENOSYS;
    return −1;
  }

  /* Successfully applied RT scheduling */
  return 0;
}

#endif

/*
 * Routine to check we have the permitted capabilities needed for program operations
 *
 * The various threads need the following capabilities:
 *
 * CAP_IPC_LOCK  (READER and WRITER) -- ability to mmap and mlock pages.
 * CAP_SYS_NICE  (READER and WRITER) -- ability to set RT scheduling priorities
 * CAP_SYS_ADMIN (READER) -- ability to set (increase) the Comedi buffer maximum size
 * CAP_SYS_ADMIN (WRITER) -- ability to set RT IO scheduling priorities (unused at present)
 * CAP_SYS_ADMIN (TIDY)   -- ability to set RT IO scheduling priorities (unused at present)
 *
 * Otherwise the MAIN thread and the TIDY thread need no special powers.  The ZMQ IO thread
 * is also unprivileged, and is currently spawned during context creation from TIDY.
 */

public int check_permitted_capabilities_ok() {
  cap_t c = cap_get_proc();
  cap_flag_value_t v = CAP_CLEAR;

  if( !c )                      /* No memory? */
    return -1;

  if( cap_get_flag(c, CAP_IPC_LOCK,  CAP_PERMITTED, &v) < 0 || v == CAP_CLEAR ||
      cap_get_flag(c, CAP_SYS_NICE,  CAP_PERMITTED, &v) < 0 || v == CAP_CLEAR ||
      cap_get_flag(c, CAP_SYS_ADMIN, CAP_PERMITTED, &v) < 0 || v == CAP_CLEAR
      ) {
    cap_free(c);
    errno = EPERM;
    return -1;
  }

  return 0;
}
```

```
#

#ifndef _RTPRIO_H
#define _RTPRIO_H

#include "general.h"

#include <sys/capability.h>

/*
 * Routine(s) for establishing thread real−time scheduling
 */

export int set_rt_scheduling(int);
export int check_permitted_capabilities_ok();

#endif /* _RTPRIO_H */
```

```
#

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include "assert.h"

#include <argtable2.h>
#include <zmq.h>

#include <getopt.h>

#include "util.h"
#include "param.h"
#include "argtab.h"

/*
 * Snapshot version
 */

#define PROGRAM_VERSION "1.0"
#define VERSION_VERBOSE_BANNER  "MCLURS ADC toolset...\n"

/*
 * Global parameters for the snapshot program
 */

extern char *snapshot_addr;

param_t globals[] ={
  { "snapshot",    "ipc://snapshot-CMD", &snapshot_addr, param_type_string, PARAM_SRC_ENV|PARAM_SRC_ARG,
    "address of snapshot command socket"
  },
};

const int n_global_params =     (sizeof(globals)/sizeof(param_t));

/*
 * Debugging print out control
 */

int   verbose   = 0;
char *program   = NULL;

/* Command line syntax options */

struct arg_lit *h1, *vn1, *v1, *q1;
struct arg_end *e1;

BEGIN_CMD_SYNTAX(help) {
  v1  = arg_litn("v",    "verbose", 0, 2,           "Increase verbosity"),
  q1  = arg_lit0("q",   "quiet",                    "Decrease verbosity"),
  h1  = arg_lit0("h",    "help",                    "Print usage help message"),
  vn1 = arg_lit0(NULL,  "version",                  "Print program version string"),
  e1  = arg_end(20)
} APPLY_CMD_DEFAULTS(help) {
  /* No defaults to apply here */
} END_CMD_SYNTAX(help)

struct arg_lit *v2, *q2;
struct arg_end *e2;
struct arg_str *u2;
struct arg_str *n2;
struct arg_str *m2;

BEGIN_CMD_SYNTAX(main) {
  v2  = arg_litn("v",    "verbose", 0, 3,           "Increase verbosity"),
  q2  = arg_lit0("q",   "quiet",                    "Decrease verbosity"),
  u2  = arg_str0("s",   "snapshot", "<url>",       "URL of snapshotter command socket"),
  m2  = arg_str0("m",    "multi", "<prefix>",       "Send multiple messages if replies begin with <prefix>"),
  n2  = arg_strn(NULL, NULL, "<args>", 1, 30,      "Message content"),
  e2  = arg_end(20)
} APPLY_CMD_DEFAULTS(main) {
  m2−>hdr.flag |= ARG_HASOPTVALUE;
  m2−>sval[0] = "";
  INCLUDE_PARAM_DEFAULTS(globals, n_global_params); /* Use defaults from parameter table */
} END_CMD_SYNTAX(main);

/* Standard help routines: display the version banner */
void print_version(FILE *fp, int verbosity) {
  fprintf(fp, "%s: Vn. %s\n", program, PROGRAM_VERSION);
  if(verbosity > 0) {           /* Verbose requested... */
    fprintf(fp, VERSION_VERBOSE_BANNER);
  }
}

/* Standard help routines: display the usage summary for a syntax */
void print_usage(FILE *fp, void **argtable, int verbosity, char *program) {
  if( !verbosity ) {
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntax(fp, argtable, "\n");
    return;
  }
  if( verbosity ) {
    char *suffix = verbosity>1? "\n\n" : "\n";
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntaxv(fp, argtable, suffix);
    if( verbosity > 1 )
      arg_print_glossary(fp, argtable, "%−25s %s\n");
  }
}

/*
 * Snapchat globals...
 */

void      *zmq_main_ctx;        /* ZMQ context for messaging */
char      *snapshot_addr;       /* The URL of the snapshotter */

/*
 * Print a reply message to stdout
 */

int print_message(char *msg, int size) {
  if( msg[size−1] != ’\n’) {
    msg[size] = ’\n’;
    fwrite(msg, size+1, 1, stdout);
  }
  else {
    fwrite(msg, size, 1, stdout);
  }
  fflush(stdout);
}

/*
 * Return true if the string p is an initial prefix of str
 */

int checked_prefix(const char *p, const char *str) {
  while(*p && *str && *p == *str) {
    if( *p != *str )            /* Mismatch with prefix */
      return 0;
    p++, str++;
  }
  return *p? 0 : 1;             /* True iff prefix has run out */
}

/*
 * Main entry point
 */

#define LOGBUF_SIZE     1024

int main(int argc, char *argv[], char *envp[]) {
  const char *prefix = NULL;
  char        buf[LOGBUF_SIZE];
  void       *snapshot;
  param_t    *p;
  int         ret, n;

  program = argv[0];

  /* Set up the standard parameters */
  /* 1. Process parameters:  internal default, then environment. */
  set_param_from_env(envp, globals, n_global_params);

  /* 2. Process parameters:  push values out to program globals */
  ret = assign_all_params(globals, n_global_params);
  assertv(ret == 0, "Push parameters failed on param %d out of %d\n", -ret, n_global_params);

  /* 3. Create and parse the command lines -- installs defaults from parameter table */
  void **cmd_help = arg_make_help();
  void **cmd_main = arg_make_main();

  /* Try first syntax */
  int err_help = arg_parse(argc, argv, cmd_help);
  if( !err_help ) {             /* Assume this was the desired command syntax */
    if(vn1->count)
      print_version(stdout, v1->count);
    if(h1->count || !vn1->count) {
      int verbose = v1->count - q1->count;
      print_usage(stdout, cmd_help, verbose>0, program);
      print_usage(stdout, cmd_main, verbose, program);
    }
    exit(0);
  }

  /* Try second syntax */
  int err_main = arg_parse(argc, argv, cmd_main);
  verbose = v2->count - q2->count;
  if( err_main ) {              /* This is the default desired syntax; give full usage */
    arg_print_errors(stderr, e2, program);
    print_usage(stderr, cmd_help, verbose>0, program);
    print_usage(stderr, cmd_main, verbose, program);
    exit(1);
  }

  /* 4. Process parameters:  copy argument values back through the parameter table */
  ret = arg_results_to_params(cmd_main, globals, n_global_params);

  /* 5. Process parameters:  deal with non-parameter table arguments where necessary */
  if(m2->count) {               /* Repeat-mode with prefix */
    prefix = m2->sval[0];
  }

  if(verbose > 2)               /* Dump global parameters for debugging purposes */
    debug_params(stderr, globals, n_global_params);

  /* Create the ZMQ contexts */
  zmq_main_ctx  = zmq_ctx_new();
  if( !zmq_main_ctx ) {
    fprintf(stderr, "%s: Error -- ZeroMQ context creation failed: %s\n", program, strerror(errno));
    exit(2);
  }

  /* Create the socket to talk to the snapshot program */
  snapshot = zh_connect_new_socket(zmq_main_ctx, ZMQ_REQ, snapshot_addr);
  if( snapshot == NULL ) {
    fprintf(stderr, "%s: Error -- unable to create socket to snapshot at %s: %s\n",
            program, snapshot_addr, strerror(errno));
    zmq_ctx_term(zmq_main_ctx);
    exit(2);
  }

  const char **msg = n2->sval;
  int          parts = n2->count;

  if(prefix && verbose > 0)
    fprintf(stderr, "Sending %d parts in multi-message mode with reply prefix '%s'\n", parts, prefix);

  do {
    int used, left;
    /* Send the message, wait for the reply;  data is in arg_str *n2 */
    if(verbose > 0)
      fprintf(stderr, "Sending message to %s...\n", snapshot_addr);

    if(verbose > 1)
      fprintf(stderr, "Build:");

    if( !prefix ) {
      used = 0;
      left = LOGBUF_SIZE−1;
      for(n=0; n<parts; n++) {
        int  len;

        len = snprintf(&buf[used], left, "%s ", msg[n]);
        if(len >= left) {
          len=left;
          fprintf(stderr, "%s: ran out of space composing message ’%s’\n", program, &buf[0]);
          exit(2);
        }
        if(verbose > 1)
          fprintf(stderr, " [%s]", buf);
        used += len;
        left −= len;
      }
      if(used)
        used−−;
    }
    else {
      used = snprintf(&buf[0], LOGBUF_SIZE−1, "%s", *msg++);
      parts−−;
      if(verbose > 1)
        fprintf(stderr, " [%s]", buf);
    }

    if(verbose > 1)
      fprintf(stderr, "\n");

    /* Send the message, omit the final null */
    ret = zh_put_msg(snapshot, 0, used, buf);
    if( ret < 0 ) {
      fprintf(stderr, "\n%s: Error −− sending message failed: %s\n", program, strerror(errno));
      zmq_close(snapshot);
      zmq_ctx_term(zmq_main_ctx);
      exit(3);
    }

    /* Wait for reply */
    if(verbose > 0)
      fprintf(stderr, "Awaiting reply from %s...\n", snapshot_addr);
    used = zh_collect_multi(snapshot, &buf[0], LOGBUF_SIZE−1, "");
    buf[LOGBUF_SIZE−1] = ’\0’;
    if(verbose >= 0)
      print_message(&buf[0], used);

  } while( prefix && parts > 0 && checked_prefix(prefix, &buf[0]) );

  /* Clean up ZeroMQ sockets and context */
  zmq_close(snapshot);
  zmq_ctx_term(zmq_main_ctx);
  exit(0);
}
```

```
#

#include "general.h"

#define _GNU_SOURCE     /* Linux−specific code below (O_PATH) */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/capability.h>
#include <sys/prctl.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>
#include <signal.h>
#include <argtable2.h>
#include "argtab.h"

#include <zmq.h>
#include <pthread.h>

#include <unistd.h>
#include <errno.h>
#include <string.h>
#include "assert.h"
#include <sched.h>

#include <comedi.h>
#include <comedilib.h>

#include "util.h"
#include "param.h"
#include "queue.h"
#include "strbuf.h"
#include "chunk.h"
#include "rtprio.h"
#include "snapshot.h"
#include "reader.h"
#include "writer.h"
#include "tidy.h"

/*
 * Snapshot version
 */

#define PROGRAM_VERSION "1.1"
#define VERSION_VERBOSE_BANNER  "MCLURS ADC toolset...\n"

/*
 * Global parameters for the snapshot program
 */

public int die_die_die_now = 0;

import  rparams     reader_parameters;
import  wparams    writer_parameters;
import  const char *tmpdir_path;
private const char *snapshot_addr;
private const char *snapshot_user;
private const char *snapshot_group;
private int        schedprio;

public param_t globals[] ={
  { "tmpdir",    "/tmp",
    &tmpdir_path,
    PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "directory for creation of temporary files"
  },
  { "freq",      "312.5e3",
    &reader_parameters.r_frequency,
    PARAM_TYPE(double), PARAM_SRC_ENV|PARAM_SRC_ARG|PARAM_SRC_CMD,
    "sampling frequency (divided by 8) of the ADC [Hz]"
  },
  { "snapshot", "ipc://snapshot−CMD",
    &snapshot_addr,
    PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "address of snapshot command socket"
  },
  { "snapdir",   "snap",
    &writer_parameters.w_snapdir,
    PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "directory where samples are written"
  },
  { "dev",        "/dev/comedi0",
    &reader_parameters.r_device,
    PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "the Comedi device to open"
  },
  { "range",      "750",
    &reader_parameters.r_range,
    PARAM_TYPE(int32), PARAM_SRC_ENV|PARAM_SRC_ARG|PARAM_SRC_CMD,
    "the ADC converter full−scale range [mV]"
  },
  { "bufsz",      "32",
    &reader_parameters.r_bufsz,
    PARAM_TYPE(int32),  PARAM_SRC_ENV|PARAM_SRC_ARG|PARAM_SRC_CMD,
    "size of the Comedi buffer [MiB]"
  },
  { "window",     "10",
    &reader_parameters.r_window,
    PARAM_TYPE(double),  PARAM_SRC_ENV|PARAM_SRC_ARG|PARAM_SRC_CMD,
    "guaranteed window in the ring buffer [s]"
  },
  { "bufhwm",     "0.9",
    &reader_parameters.r_buf_hwm_fraction,
    PARAM_TYPE(double), PARAM_SRC_ENV|PARAM_SRC_ARG|PARAM_SRC_CMD,
    "ring buffer high−water mark fraction"
  },
  { "rtprio",     NULL,
    &schedprio,
    PARAM_TYPE(int32),  PARAM_SRC_ENV|PARAM_SRC_ARG,
    "priority of real−time threads [0−99]"
  },
  { "rdprio",     NULL,
    &reader_parameters.r_schedprio,
    PARAM_TYPE(int32),  PARAM_SRC_ENV|PARAM_SRC_ARG,
    "priority of real−time reader thread [0−99]"
  },
  { "wrprio",     NULL,
    &writer_parameters.w_schedprio,
    PARAM_TYPE(int32),  PARAM_SRC_ENV|PARAM_SRC_ARG,
    "priority of real−time writer thread [0−99]"
  },
  { "user",       NULL,
    &snapshot_user,
    PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "user/UID for file system access and creation"
  },
  { "group",      NULL,
    &snapshot_group,
    PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "group/GID for file system access and creation"
  },
  { "ram",        "64",
    &writer_parameters.w_lockedram,
    PARAM_TYPE(int32), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "amount of data RAM to lock [MiB]"
  },
  { "wof",        "0.5",
    &writer_parameters.w_writeahead,
    PARAM_TYPE(double), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "write overbooking fraction"
  },
  { "chunk",      "1024",
    &writer_parameters.w_chunksize,
    PARAM_TYPE(int32), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "size of a transfer chunk [KiB]"
  },
};
```

```c
public const int n_global_params =      (sizeof(globals)/sizeof(param_t));

/*
 * Debugging print out control
 */

public int   verbose;
public char *program   = NULL;

/* Command line syntax options -- there are no mandatory arguments on the main command line! */

private struct arg_lit *h1, *vn1, *v1, *q1;
private struct arg_end *e1;

BEGIN_CMD_SYNTAX(help) {
  v1  = arg_litn("v",   "verbose", 0, 3,    "Increase verbosity"),
  q1  = arg_lit0("q",   "quiet",            "Decrease verbosity"),
  h1  = arg_lit0("h",   "help",             "Print usage help message"),
  vn1 = arg_lit0(NULL, "version",           "Print program version string"),
  e1  = arg_end(20)
} APPLY_CMD_DEFAULTS(help) {
  /* No defaults to apply here */
} END_CMD_SYNTAX(help)

private struct arg_lit *v2, *q2;
private struct arg_end *e2;

BEGIN_CMD_SYNTAX(main) {
  v2  = arg_litn("v",   "verbose", 0, 3,            "Increase verbosity"),
  q2  = arg_lit0("q",   "quiet",                    "Decrease verbosity"),
        arg_str0("s",   "snapshot",  "<url>",     "URL of snapshotter command socket"),
        arg_str0(NULL, "tmpdir",  "<path>",       "Path to temporary directory"),
        arg_str0("S",   "snapdir",  "<path>",     "Path to samples directory"),
        arg_dbl0("f",   "freq",  "<real>",        "Per-channel sampling frequency [Hz]"),
        arg_dbl0("w",   "window",  "<real>",      "Min. capture window length [s]"),
        arg_dbl0("B",   "bufhwm",  "<real>",      "Ring buffer High-water mark fraction"),
        arg_str0("d",   "dev",  "<path>",         "Comedi device to use"),
        arg_int0("P",   "rtprio",  "<1-99>",      "Common thread RT priority"),
        arg_int0("R",   "rdprio",  "<1-99>",      "Reader thread RT priority"),
        arg_int0("W",   "wrprio",  "<1-99>",      "Writer thread RT priority"),
        arg_str0("u",   "user",  "<uid/name>",    "User to run as"),
        arg_str0("g",   "group",  "<gid/name>",  "Group to run as"),
        arg_int0("b",   "bufsz",  "<int>",        "Comedi ring buffer Size [MiB]"),
        arg_int0("m",   "ram",  "<int>",          "Data Transfer RAM size [MiB]"),
        arg_int0("r",   "range",  "<int>",        "ADC full-scale range [mV]"),
        arg_int0("c",   "chunk",  "<int>",        "File transfer chunk size [kiB]"),
        arg_dbl0("W",   "wof",  "<real>",         "Write Overbooking Fraction"),
  e2  = arg_end(20)
} APPLY_CMD_DEFAULTS(main) {
  INCLUDE_PARAM_DEFAULTS(globals, n_global_params);
} END_CMD_SYNTAX(main);

/* Standard help routines: display the version banner */
private void print_version(FILE *fp, int verbosity) {
  fprintf(fp, "%s: Vn. %s\n", program, PROGRAM_VERSION);
  if(verbosity > 0) {           /* Verbose requested... */
    fprintf(fp, VERSION_VERBOSE_BANNER);
  }
}

/* Standard help routines: display the usage summary for a syntax */
private void print_usage(FILE *fp, void **argtable, int verbosity, char *program) {
  if( !verbosity ) {
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntax(fp, argtable, "\n");
    return;
  }
  if( verbosity ) {
    char *suffix = verbosity>1? "\n\n" : "\n";
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntaxv(fp, argtable, suffix);
    if( verbosity > 1 )
      arg_print_glossary(fp, argtable, "%-25s %s\n");
  }
}

/*
 * Snapshot globals for this file.
 */

private const char *snapshot_addr = NULL;  /* The address of the main command socket */
private const char *snapshot_user = NULL;  /* The user we should run as, after startup */
private const char *snapshot_group = NULL; /* The group to run as, after startup */
private int        schedprio;             /* Real−time priority for reader and writer */

/*
 * Snapshot globals shared between threads
 */

public void       *snapshot_zmq_ctx;    /* ZMQ context for messaging −− created by the TIDY thread */

public int          tmpdir_dirfd;       /* The file descriptor obtained for the TMPDIR directory */
public const char *tmpdir_path;         /* The path for the file descriptor above */

/*
 * Thread handles for reader and writer
 */

private pthread_t reader_thread,
                  writer_thread,
                  tidy_thread;

private pthread_attr_t reader_thread_attr,
                       writer_thread_attr,
                       tidy_thread_attr;

/*
 * Establish main comms:  this routine runs last, so it mostly does connect() calls.
 * It must run when the other three threads are already active.
 */

private void *log_socket;       /* N.B.  This socket is opened by the TIDY thread, but not used there */
private void *reader;
private void *writer;
private void *command;

private int create_main_comms() {
  int ret;

  /* Create and initialise the sockets: reader and writer command sockets */
  reader = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_REQ, READER_CMD_ADDR);
  if( reader == NULL ) {
    fprintf(stderr, "%s: Error −− unable to connect internal socket to reader: %s\n", program, strerror(errno));
    return −1;
  }
  writer = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_REQ, WRITER_CMD_ADDR);
  if( writer == NULL ) {
    fprintf(stderr, "%s: Error −− unable to connect internal socket to writer: %s\n", program, strerror(errno));
    return −1;
  }

  /* Create and initialise the external command socket */
  command = zh_bind_new_socket(snapshot_zmq_ctx, ZMQ_REP, snapshot_addr);
  if( command == NULL ) {
    fprintf(stderr, "%s: Error −− unable to bind external command socket %s: %s\n",
            program, snapshot_addr, strerror(errno));
    return −1;
  }

  return 0;
}

/* Close everything created above */

private void close_main_comms() {
  zmq_close(reader);
  zmq_close(writer);
  zmq_close(command);
}
```

```
/*
 * Sort out the capabilities required by the process.  (If not running
 * as root, check that we have the capabilities we require.)  Release
 * any capabilities not needed and lock against dropping privilege.
 *
 * The threads need the following capabilities:
 *
 * CAP_IPC_LOCK  (Reader and Writer) -- ability to mmap and mlock pages.
 * CAP_SYS_NICE  (Reader and Writer) -- ability to set RT scheduling priorities
 * CAP_SYS_ADMIN (Reader) -- ability to set (increase) the Comedi buffer maximum size
 * CAP_SYS_ADMIN (Writer) -- ability to set RT IO scheduling priorities (unused at present)
 * CAP_SYS_ADMIN (Tidy)   -- ability to set RT IO scheduling priorities (unused at present)
 *
 * CAP_SETUID (Main)
 * CAP_SETGID (Main)      -- ability to change user ID
 *
 * Otherwise the main thread and the tidy thread need no special powers.  The ZMQ IO thread
 * is also unprivileged, and is currently spawned during context creation from tidy.
 */

private int snap_adjust_capabilities() {
  cap_t c = cap_get_proc();
  uid_t u = geteuid();
  int ret = 0;

  if( !c )                      /* No memory? */
    return -1;

  if( check_permitted_capabilities_ok() < 0 ) {
    fprintf(stderr, "%s: Error -- I do not have the necessary capabilities to operate\n", program);
    return -1;
  }

  if( !u ) {
    const cap_value_t vs[] = { CAP_IPC_LOCK, CAP_SYS_NICE, CAP_SYS_ADMIN, CAP_SETUID, CAP_SETGID, };

    /* So we are root and have the capabilities we need.  Prepare to drop the others... */
    /* Keep the EFFECTIVE capabilities as long as we stay root */
    cap_clear(c);
    cap_set_flag(c, CAP_PERMITTED, sizeof(vs)/sizeof(cap_value_t), &vs[0], CAP_SET);
    cap_set_flag(c, CAP_EFFECTIVE, sizeof(vs)/sizeof(cap_value_t), &vs[0], CAP_SET);
    if( prctl(PR_SET_KEEPCAPS, 1L) <0 ) {
      cap_free(c);
      fprintf(stderr, "%s: Error -- unable to keep required capabilities on user change\n", program);
      return -1;
    }

    ret = cap_set_proc(c);
  }

  cap_free(c);
  return ret;
}

/*
 * Drop privileges and capabilities when appropriate.
 */

private int main_adjust_capabilities(uid_t uid, gid_t gid) {
  cap_t c = cap_get_proc();
  const cap_value_t vs[] = { CAP_SETUID, CAP_SETGID, };

  /* Drop all capabilities except CAP_SETUID/GID from effective set */

  if(c) {
    cap_clear_flag(c, CAP_EFFECTIVE);
    cap_set_flag(c, CAP_EFFECTIVE, sizeof(vs)/sizeof(cap_value_t), &vs[0], CAP_SET);
    if( cap_set_proc(c) < 0 ) {
      cap_free(c);
      fprintf(stderr, "%s: Error -- MAIN thread fails to clear capabilities: %s\n", program, strerror(errno));
      return -1;
    }
    cap_free(c);
  }
  /* Drop all user and group privileges:  set all uids to uid and all gids to gid */
  /* Complain if that fails −− we were not root and uid/gid were not in our set */
  if( setresgid(gid, gid, gid) < 0 ) {
    fprintf(stderr, "%s: Error −− MAIN thread unable to change to gid %d: %s\n", program, gid, strerror(errno));
    return −1;
  }
  if( setresuid(uid, uid, uid) < 0 ) {
    fprintf(stderr, "%s: Error −− MAIN thread unable to change to uid %d: %s\n", program, uid, strerror(errno));
    return −1;
  }

  c = cap_get_proc();
  if(c) {
    cap_set_flag(c, CAP_PERMITTED, sizeof(vs)/sizeof(cap_value_t), &vs[0], CAP_CLEAR);
    if( cap_set_proc(c) < 0 ) {
      cap_free(c);
      fprintf(stderr, "%s: Error −− MAIN thread keeps setuid/gid capabilities: %s\n", program, strerror(errno));
      return −1;
    }
    cap_free(c);
  }

  /* Now check we still have the required permitted capabilities */
  if( check_permitted_capabilities_ok() < 0 ) {
    fprintf(stderr, "%s: Error −− MAIN thread lost capabilities on changing user!\n", program);
    return −1;
  }

  return 0;
}

/*
 * Deal nicely with the interrupt signal.
 * Basically, the signal sets the die_die_die_now flag which the various threads notice.
 * CURRENTLY NOT WORKING PROPERLY SO DISABLED
 */

private void intr_handler(int i) {
  die_die_die_now++;
}

private int set_intr_sig_handler() {
  struct sigaction a;

  bzero(&a, sizeof(a));
  a.sa_handler = intr_handler;
  if( sigaction(SIGINT, &a, NULL) < 0 ) {
    fprintf(stderr, "%s: Error −− unable to install INT signal handler: %s\n", program, strerror(errno));
    return −1;
  }
  return 0;
}

/*
 * Process a (possibly multipart) log message.
 * Collect the various pieces and write to stderr
 */

#define LOGBUF_SIZE     MSGBUFSIZE
private char pfx[] = "Log: ";

private int process_log_message(void *s) {
  char log_buffer[MSGBUFSIZE];
  int used;

  memcpy(&log_buffer[0], &pfx[0], sizeof(pfx));
  used = sizeof(pfx)−1;
  used += zh_collect_multi(s, &log_buffer[used], LOGBUF_SIZE−1, "");
  if( log_buffer[used−1] != ’\n’) {
    log_buffer[used] = ’\n’;
    fwrite(log_buffer, used+1, 1, stderr);
  }
  else {
    fwrite(log_buffer, used, 1, stderr);
  }
  fflush(stderr);
  return 0;
}

/*
 * Handle replies from READER and WRITER threads.  The reply message
 * is a pointer to a set of error strbufs.  We collect and join all
 * the strings in the reply buffer.  The collector maintains as
 * invariant that "used==0 || reply_buffer[used−1] is not NUL" and that
 * "b == &reply_buffer[used]".
 */

#define REPLY_BUFSIZE   MSGBUFSIZE
private char reply_buffer[REPLY_BUFSIZE];

private int process_reply(void *s) {
  int     size;
  strbuf  err;
  char   *b = &reply_buffer[0];
  int     used;

  size = zh_get_msg(s, 0, sizeof(strbuf), (void *)&err);
  assertv(size==sizeof(err), "Reply message of wrong size %d\n", size);

  /* Establish invariants */
  *b = ’\0’;  used = 0;

  /* Traverse the strbuf chain once collecting data, then release */
  for_nxt_in_Q(queue *q, strbuf2qp(err), (queue *)NULL)
    strbuf  s = qp2strbuf(q);
    int     n = strbuf_used(s);
    if(n) {                             /* Empty strbuf, nothing to do */
      strbuf_revert(s);                 /* Remove any internal NUL characters */
      if(n > REPLY_BUFSIZE−used) {      /* There is too much data */
        n = REPLY_BUFSIZE−used−1;       /* We can manage this much of it */
      }
      //      fprintf(stderr, "strbuf %p, used %d, ptr %p, string ’%s’\n",
      //              s, n, b, strbuf_string(s));
      memcpy(b, strbuf_string(s), n);   /* Copy the data */
      b += n;  used += n;               /* Now we have used this much space */
      //      while( b[−1] == ’\0’ ) b−−,used−−;        /* Skip back over any NULs */
      //      fprintf(stderr, "strbuf %p, ptr now %p, total used now %d\n", s, b, used);
    }
  end_for_nxt;

  release_strbuf(err);  /* Free the entire link of strbufs */

  if( b[−1] != ’\n’ )    /* Ensure final newline */
    *b = ’\n’;

  /* Send the complete reply */
  used = b − &reply_buffer[0];
  zh_put_msg(command, 0, used, &reply_buffer[0]);
  return 0;
}

/*
 * Handle commands sent to the snapshotter.  These are forwarded
 * either to the reader thread or the writer thread, and their replies
 * are returned to the originator.  Using the REP socket ensures only
 * one outstanding message is in process, so simplifies the reply routing.
 */

private int process_snapshot_command() {
  strbuf c,e;                   /* Command and Error buffers */
  char  *buf;
  int   size, ret;
  int   fwd;

  c = alloc_strbuf(2);
  e = strbuf_next(c);

  buf = strbuf_string(c);
  size = zh_get_msg(command, 0, strbuf_space(c), buf);
  if( !size ) {
    ret = zh_put_msg(command, 0, 0, NULL); /* If empty message received, send empty reply at once */
    release_strbuf(c);
    assertv(ret == 0, "Reply to command failed, %d\n", ret);
    return 0;
  }
  strbuf_setpos(c, size);
  buf[size] = '\0';
  // fprintf(stderr, "Msg '%c' (%d)\n", buf[0], buf[0]);
  fwd = 0;
  switch(buf[0]) {
  case 'q':
  case 'Q':                     /* Deal specially with Quit command, to close down nicely... */
    ret = zh_put_msg(reader, 0, 0, NULL); /* Forward zero length message to the READER thread */
    assertv(ret == 0, "Quit to READER failed, %d\n", ret);
    ret = zh_put_msg(writer, 0, 0, NULL); /* Forward zero length message to the WRITER thread */
    assertv(ret == 0, "Quit to WRITER failed, %d\n", ret);
    ret = zh_put_msg(command, 0, 7, "OK Quit"); /* Reply to Quit here */
    assertv(ret == 7, "Quit reply failed, %d\n", ret);
    break;

  case 'g':
  case 'G':
  case 'h':
  case 'H':
  case 'i':
  case 'I':
  case 'p':
  case 'P':
    /* Forward these commands to the READER thread */
    ret = zh_put_msg(reader, 0, sizeof(strbuf), (void *)&c);
    assertv(ret == sizeof(c), "Forward to READER failed, %d\n", ret);
    fwd++;
    break;

  case 'd':
  case 'D':
  case 's':
  case 'S':
  case 'z':
  case 'Z':
    /* Forward snapshot and dir commands to WRITER */
    ret = zh_put_msg(writer, 0, sizeof(strbuf), (void *)&c);
    assertv(ret == sizeof(c), "Forward to WRITER failed, %d\n", ret);
    fwd++;
    break;

  case '?':
    buf[0] = '!';
    ret = zh_put_msg(command, 0, size, buf); /* Reply to 'ping' message */
    assertv(ret > 0, "Reply to ping failed, %d\n", ret);
    break;

  default:
    strbuf_printf(e, "NO: Unknown command: '%s'\n", buf);
    fprintf(stderr, "%s: %s", program, strbuf_string(e));
    ret = zh_put_msg(command, 0, strbuf_used(e) , strbuf_string(e));
    assertv(ret == strbuf_used(e), "Reject unknown reply failed, %d\n", ret);
    break;
  }
  if( !fwd )                    /* Didn't use the strbufs */
    release_strbuf(c);
  return 0;
}

/*
 * MAIN thread message loop
 */

#define MAIN_LOOP_POLL_INTERVAL 20

private void main_thread_msg_loop() {    /* Read and process messages */
  int poll_delay;
  int running;
  zmq_pollitem_t  poll_list[] =
    { { log_socket, 0, ZMQ_POLLIN, 0 },
      { command, 0, ZMQ_POLLIN, 0 },
      { reader, 0, ZMQ_POLLIN, 0 },
      { writer, 0, ZMQ_POLLIN, 0 },
    };
#define  N_POLL_ITEMS  (sizeof(poll_list)/sizeof(zmq_pollitem_t))
  int (*poll_responders[N_POLL_ITEMS])(void *) =
    { process_log_message,
      process_snapshot_command,
      process_reply,
      process_reply,
    };

  fprintf(stderr, "Log: starting MAIN thread polling loop with %d items\n", N_POLL_ITEMS);
  running = true;
  poll_delay = MAIN_LOOP_POLL_INTERVAL;
  while(running && !die_die_die_now) {
    int n;
    int ret = zmq_poll(&poll_list[0], N_POLL_ITEMS, poll_delay);

    if( ret < 0 && errno == EINTR ) { /* Interrupted */
      fprintf(stderr, "%s: MAIN thread loop interrupted\n", program);
      break;
    }
    if(ret < 0)
      break;
    running = reader_parameters.r_running || writer_parameters.w_running;
    if( !running )              /* Flush out last messages */
      poll_delay = 1000;
    for(n=0; n<N_POLL_ITEMS; n++) {
      if( poll_list[n].revents & ZMQ_POLLIN ) {
        ret = (*poll_responders[n])(poll_list[n].socket);
        assertv(ret >= 0, "Error in message processing in MAIN poll loop, ret %d\n", ret);
        running = true;
      }
    }
  }
}

/*
 * Snapshot main routine.
 */

public int main(int argc, char *argv[], char *envp[]) {
  char *thread_return = NULL;
  int ret, running, poll_delay;
  char *cmd_addr;
  param_t *p;

  program = argv[0];

  /* Set up the standard parameters */
  /* 1. Process parameters:  internal default, environment, then command−line argument. */
  set_param_from_env(envp, globals, n_global_params);

  /* 2. Process parameters:  push values out to program globals */
  ret = assign_all_params(globals, n_global_params);
  assertv(ret == 0, "Push parameters failed on param %d out of %d\n", −ret, n_global_params);

  if(verbose > 2) {
    fprintf(stderr, "Params before cmdline...\n");
    debug_params(stderr, globals, n_global_params);
  }

  /* 3. Create and parse the command lines -- installs defaults from parameter table */
  void **cmd_help = arg_make_help();
  void **cmd_main = arg_make_main();

  /* Try first syntax -- reject empty command lines */
  int err_help = arg_parse(argc, argv, cmd_help);
  if( !err_help && (vn1−>count || h1−>count) ) {        /* Assume this was the desired command syntax */
    if(vn1−>count)
      print_version(stdout, v1−>count);
    if(h1−>count || !vn1−>count) {
      print_usage(stdout, cmd_help, v1−>count>0, program);
    print_usage(stdout, cmd_main, v1->count, program);
  }
  exit(0);
}

/* Try second syntax -- may be empty, means use default or environment variable parameters */
int err_main = arg_parse(argc, argv, cmd_main);
if( err_main ) {              /* This is the default desired syntax; give full usage */
  arg_print_errors(stderr, e2, program);
  print_usage(stderr, cmd_help, v2->count>0, program);
  print_usage(stderr, cmd_main, v2->count, program);
  exit(1);
}

verbose = v2->count - q2->count;
if(verbose > 2) {
  fprintf(stderr, "Params before reverse pass...\n");
  debug_params(stderr, globals, n_global_params);
}

/* 4. Process parameters:  copy argument values back through the parameter table */
ret = arg_results_to_params(cmd_main, globals, n_global_params);

/* 5. Process parameters:  deal with non-parameter table arguments where necessary */

if(verbose > 1) {
  fprintf(stderr, "Params before checking...\n");
  debug_params(stderr, globals, n_global_params);
}

/* 5a. Verify parameters required by the main program/thread */
tmpdir_dirfd = open(tmpdir_path, O_PATH|O_DIRECTORY); /* Verify the TMPDIR path */
if( tmpdir_dirfd < 0 ) {
  fprintf(stderr, "%s: Error -- cannot access given TMPDIR ’%s’: %s\n", program, tmpdir_path, strerror(errno));
  exit(2);
}

/* Compute the UID and GID for unprivileged operation.
 *
 * If the GID parameter is set, use that for the group; if not, but
 * the UID parameter is set, get the group from that user and set
 * the uid from there too.  If neither is set, use the real uid/gid
 * of the thread.
 */

gid_t gid = -1;
if(snapshot_group) {
  struct group *grp = getgrnam(snapshot_group);

  if(grp == NULL) {           /* The group name was invalid  */
    fprintf(stderr, "%s: Error -- given group %s is not recognised\n", program, snapshot_group);
    exit(2);
  }
  gid = grp->gr_gid;
}

uid_t uid = -1;
if(snapshot_user) { /* Got a UID value */
  struct passwd *pwd = getpwnam(snapshot_user);

  if(pwd == NULL) {           /* The user name was invalid */
    fprintf(stderr, "%s: Error -- given user %s is not recognised\n", program, snapshot_user);
    exit(2);
  }

  uid = pwd->pw_uid;  /* Use this user’s UID */
  if(gid == -1)
    gid = pwd->pw_gid;        /* Use this user’s principal GID */
}
else {
  uid = getuid();             /* Use the real UID of this thread */
  gid = getgid();             /* Use the real GID of this thread */
}

/* 5b. Check capabilities and drop privileges */
```

```
  if( snap_adjust_capabilities() < 0 ) {
    exit(2);
  }
  if( main_adjust_capabilities(uid, gid) < 0 ) {
    exit(2);
  }

  /* Check the supplied parameters;  WRITER must come first as READER needs chunk size */
  strbuf e = alloc_strbuf(1);   /* Catch parameter error diagnostics */

  /* 5c. Verify and initialise parameters for the WRITER thread */
  if( !writer_parameters.w_schedprio)
    writer_parameters.w_schedprio = schedprio;
  strbuf_printf(e, "%s: Error -- WRITER Params: ", program);
  ret = verify_writer_params(&writer_parameters, e);
  if( ret < 0 ) {
    fprintf(stderr, "%s\n", strbuf_string(e));
    exit(3);
  }

   /* 5d. Verify and initialise parameters for the READER thread */
  if( !reader_parameters.r_schedprio )
    reader_parameters.r_schedprio = schedprio;
  strbuf_printf(e, "%s: Error -- READER Params: ", program);
  ret = verify_reader_params(&reader_parameters, e);
  if( ret < 0 ) {
    fprintf(stderr, "%s\n", strbuf_string(e));
    exit(3);
  }

  release_strbuf(e);

#if 0
  /* Exit nicely on SIGINT:  this is done by setting the die_die_die_now flag. */
  if( set_intr_sig_handler() < 0 ) {
    exit(3);
  }
#endif

  /* Create the TIDY thread */
  pthread_attr_init(&tidy_thread_attr);
  if( pthread_create(&tidy_thread, &tidy_thread_attr, tidy_main, &log_socket) < 0 ) {
    fprintf(stderr, "%s: Error -- TIDY   thread creation failed: %s\n", program, strerror(errno));
    exit(4);
  }

  /* Wait here for log_socket */
  while( !die_die_die_now && !log_socket ) {
    usleep(10000);
  }

  if( !die_die_die_now ) {
    pthread_attr_init(&reader_thread_attr);    /* Create the READER thread */
    if( pthread_create(&reader_thread, &reader_thread_attr, reader_main, NULL) < 0 ) {
      fprintf(stderr, "%s: Error -- READER thread creation failed: %s\n", program, strerror(errno));
      exit(4);
    }

    pthread_attr_init(&writer_thread_attr);    /* Create the WRITER thread */
    if( pthread_create(&writer_thread, &writer_thread_attr, writer_main, NULL) < 0 ) {
      fprintf(stderr, "%s: Error -- WRITER thread creation failed: %s\n", program, strerror(errno));
      exit(4);
    }
  }

  /* Wait for the threads to establish comms etc. DON'T WAIT TOO LONG */
  while( !die_die_die_now ) {
    usleep(10000);              /* Wait for 10ms */
    if(reader_parameters.r_running && writer_parameters.w_running)
      break;                    /* Now ready to start main loop */
  }

  /* Run the MAIN thread sevice loop here */
  if( create_main_comms() < 0 ) {
    die_die_die_now++;
  }
  main_thread_msg_loop();

  /* Clean up the various threads */
  if(reader_thread) {
    if( pthread_join(reader_thread, (void *)&thread_return) < 0 ) {
      fprintf(stderr, "%s: Error -- READER thread join error: %s\n", program, strerror(errno));
      thread_return = NULL;
    }
    else {
      if( thread_return ) {
        fprintf(stderr, "Log: READER thread rejoined -- %s\n", thread_return);
        thread_return = NULL;
      }
    }
  }
  if(writer_thread) {
    if( pthread_join(writer_thread, (void *)&thread_return) < 0 ) {
      fprintf(stderr, "%s: Error -- WRITER thread join error: %s\n", program, strerror(errno));
      thread_return = NULL;
    }
    else {
      if( thread_return ) {
        fprintf(stderr, "Log: WRITER thread rejoined -- %s\n", thread_return);
        thread_return = NULL;
      }
    }
  }

  if( pthread_join(tidy_thread, (void *)&thread_return) < 0 ) {
    fprintf(stderr, "%s: Error -- TIDY   thread join error: %s\n", program, strerror(errno));
    thread_return = NULL;
  }
  else {
    if( thread_return ) {
      fprintf(stderr, "Log: TIDY   thread rejoined -- %s\n", thread_return);
      thread_return = NULL;
    }
  }

  /* Clean up our ZeroMQ sockets */
  close_main_comms();

  /* These were created by the TIDY thread */
  zmq_close(log_socket);
  zmq_ctx_term(snapshot_zmq_ctx);
  exit(0);
}
```

```
#

#include "general.h"

/* Shared globals */

export void       *zmq_main_ctx;

export param_t     globals[];
export const int   n_global_params;

export int         verbose;

export int         die_die_die_now;

export int         tmpdir_dirfd;
export const char *tmpdir_path;

/* Common definitions */

#define LOG_SOCKET      "inproc://Main−LOG"

#define MSGBUFSIZE      8192
```

```
#

#include "general.h"

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "strbuf.h"
#include "queue.h"

#define N_STRBUF_ALLOC  8                       /* Allocate this many buffers at one go */
#define MAX_STRBUF_SIZE (512-sizeof(queue))     /* Strbuf will hold 496 characters maximum */

struct _strbuf {
  queue   s_Q;                          /* Queue header to avoid malloc() calls */
  int     s_used;                       /* Pointer to next free space in buffer */
  char    s_buffer[MAX_STRBUF_SIZE];    /* Buffer space when in use */
};

/*
 * Return the usable string space in a strbuf.
 */

public int strbuf_space(strbuf s) {
  return MAX_STRBUF_SIZE;
}

/*
 * Allocate and free strbufs, using a queue to avoid excessive malloc()
 */

private QUEUE_HEADER(sbufQ);
private int N_in_Q = 0;

public strbuf alloc_strbuf(int nr) {
  queue *ret;

  if( N_in_Q < nr ) {   /* The queue doesn’t have enough */
    int n;

    for(n=0; n<N_STRBUF_ALLOC; n++) {
      queue *q = (queue *)calloc(1, sizeof(struct _strbuf));

      if( !q ) {                        /* Allocation failed */
        if( N_in_Q >= nr )
          break;                        /* But we have enough now anyway */
        return NULL;
      }
      init_queue(q);
      queue_ins_after(&sbufQ, q);
      N_in_Q++;
    }
  }
  ret = de_queue(queue_next(&sbufQ));
  while(--nr > 0) {             /* Collect enough to satisfy request */
    queue *p = de_queue(queue_next(&sbufQ));

    init_queue(p);
    ((strbuf)p)->s_used = 0;
    ((strbuf)p)->s_buffer[0] = ’\0’;
    queue_ins_before(ret, p);
  }
  return (strbuf)ret;
}

private void free_strbuf(strbuf s) {
  free( (void *)s );
}

public void release_strbuf(strbuf s) {
  queue *p;

  while( (p = de_queue(queue_next(&s->s_Q))) != NULL ) {
    init_queue(p);
    queue_ins_before(&sbufQ, p);
    N_in_Q++;
  }
  queue_ins_before(&sbufQ, &s->s_Q);
  N_in_Q++;
}

/*
 * Get the string pointer from an strbuf (since the latter is opaque, we need a function for this).
 */

public char *strbuf_string(strbuf s) {
  return &s->s_buffer[0];
}

/*
 * Return the number of characters printed into a strbuf so far
 */

public int strbuf_used(strbuf s) {
  return s->s_used;
}

/*
 * Mark the current used position.
 */

public int strbuf_setpos(strbuf s, int pos) {
  if( !s  ) {
    errno = EINVAL;
    return -1;
  }
  if(pos < 0 || pos > MAX_STRBUF_SIZE) {
    errno = ERANGE;
    return -1;
  }
  int used = s->s_used;
  s->s_used = pos;
  return used;
}

/*
 * Do a formatted print into an strbuf, starting at pos.
 */

private int strbuf_vprintf(strbuf s, int pos, const char *fmt, va_list ap) {
  int   rest;
  int   used;
  char *buf;

  if(pos < 0)                   /* Position one character back from end (i.e. skip NULL) or at start */
    pos = s->s_used ? s->s_used : 0;
  buf  = &s->s_buffer[pos];
  rest = MAX_STRBUF_SIZE - pos; /* There should be this much space remaining */
  if(rest < 0) {
    errno = EINVAL;
    return -1;
  }
  used = vsnprintf(buf, rest, fmt, ap);
  s->s_used = used>=rest? MAX_STRBUF_SIZE : s->s_used + used;
  return used;
}

/*
 * Do fixup of the format: deal with all standard printf % options,
 * plus any extras.  When we see a pc_flag in one of the structures in
 * the list below, then we call the associated pc_func and interpolate
 * the string it returns.
 */

typedef struct _percent *percent;
struct _percent {
  percent       pc_link;
  char          pc_flag;        /* If we see this flag ... */
  const char *(*pc_func)();     /* ... then this function gives us the string */
};
```

```
private percent percent_list = NULL;

private percent find_in_list(char c) {
  percent p = percent_list;

  for( ; p; p=p->pc_link) {
    if(p->pc_flag == c)
      return p;
  }
  return NULL;
}

private void do_extra_percents(char *buf, int size, const char *fmt) {

  /* Copy the format into the buffer, checking each % modifier against the list */
  for( ; size > 1 && *fmt; size-- ) {
    if( (*buf++ = *fmt++) != '%' )
      continue;

    size--;                         /* Count the previous % that was copied */
    percent p = find_in_list(*buf++ = *fmt++);  /* Look up the next character */

    if(p == NULL)                   /* Not an extension, keep copying */
      continue;

    int used = snprintf(buf-2, size, "%s", (*p->pc_func)()); /* Interpolate at most 'size' chars */
    if(used >= size) {              /* Output was truncated */
      buf += size-1;
      break;
    }
    buf  += used-2;                 /* We copied 'used' characters, overwriting 2 */
    size -= used-3;                 /* The for loop will decerement one more...  */
  }
  *buf = '\0';
}

/* Start printing into the buffer at position pos */

public int strbuf_printf_pos(strbuf s, int pos, const char *fmt, ...) {
  va_list ap;
  int     used;
  char    fmt_buf[MAX_STRBUF_SIZE];

  if( strbuf_setpos(s, pos) < 0 )
    return -1;
  do_extra_percents(&fmt_buf[0], MAX_STRBUF_SIZE, fmt);
  va_start(ap, fmt);
  used = strbuf_vprintf(s, pos, fmt_buf, ap);
  va_end(ap);
  return used;
}

/* Start printing into the buffer at position 0 */

public int strbuf_printf(strbuf s, const char *fmt, ...) {
  va_list ap;
  int     used;
  char    fmt_buf[MAX_STRBUF_SIZE];

  if( strbuf_setpos(s, 0) < 0 )
    return -1;
  do_extra_percents(&fmt_buf[0], MAX_STRBUF_SIZE, fmt);
  va_start(ap, fmt);
  used = strbuf_vprintf(s, 0, fmt_buf, ap);
  va_end(ap);
  return used;
}

/* Start printing into the buffer at the current position */

public int strbuf_appendf(strbuf s, const char *fmt, ...) {
  va_list ap;
  int     used;
  char    fmt_buf[MAX_STRBUF_SIZE];
  do_extra_percents(&fmt_buf[0], MAX_STRBUF_SIZE, fmt);
  va_start(ap, fmt);
  used = strbuf_vprintf(s, -1, fmt_buf, ap);
  va_end(ap);
  return used;
}

/*
 * Register new percent interpreters.
 */

public int register_error_percent_handler(char c, const char *(*fn)()) {
  percent p = calloc(1, sizeof(struct _percent));

  if(p == NULL) {
    return -1;
  }

  p->pc_link = percent_list;
  p->pc_flag = c;
  p->pc_func = fn;
  percent_list = p;
  return 0;
}

/*
 * Revert a strbuf -- remove extra NUL characters inserted by tokenising
 */

public void strbuf_revert(strbuf s) {
  char *p = &s->s_buffer[0];
  int   n;

  for(n=0; n<s->s_used; n++,p++)
    if( !*p ) *p = ' ';
  n = (n == MAX_STRBUF_SIZE)? n-1 : n;
  s->s_buffer[n] = '\0';
}

/*
 * Debug a strbuf
 */

public void debug_strbuf(FILE *fp, strbuf s) {
  char *str = strbuf_string(s);
  char  buf[MAX_STRBUF_SIZE+64];
  char *b;
  int   used;
  int   n;

  used = snprintf(&buf[0], 64, "s=%p, n=%p, q=%p: data[0..%d]='",
                  s, s->s_Q.q_next, s->s_Q.q_prev, s->s_used);
  b = &buf[used];
  n = MAX_STRBUF_SIZE+64-used-1;
  if( n > s->s_used )
    n = s->s_used;
  while(n-- > 0) if( (*b++ = *str++) == '\0' ) b[-1] = ' ';
  *b++ ='\'';
  *b = '\0';
  fwrite(&buf[0], 1, b-&buf[0], fp);
}
```

```c
#

#ifndef _STRBUF_H
#define _STRBUF_H

#include "general.h"

/*
 * Error buffer structure.
 */

typedef struct _strbuf *strbuf; /* Opaque object */

export strbuf alloc_strbuf();
export void   release_strbuf(strbuf);
export char  *strbuf_string(strbuf);
export int    strbuf_space(strbuf);
export int    strbuf_used(strbuf);
export int    strbuf_setpos(strbuf,int);

#include <stdio.h>
#include <stdarg.h>

export int  strbuf_printf(strbuf, const char *, ...);
export int  strbuf_appendf(strbuf, const char *, ...);
export int  strbuf_printf_pos(strbuf, int, const char *, ...);
export int  register_error_percent_handler(char, const char *(*)());
export void strbuf_revert(strbuf);
export void debug_strbuf(FILE *, strbuf);

#define strbuf_clear(s) ((void) strbuf_setpos(s, 0))

#define strbuf_next(s)  ((strbuf)queue_next((queue *)(s)))
#define strbuf_prev(s)  ((strbuf)queue_prev((queue *)(s)))

#define strbuf2qp(s)    ((queue *)(s))
#define qp2strbuf(q)    ((strbuf)(q))

#endif /* _STRBUF_H */
```

```
#

#include "general.h"

/*
 * Low-priority thread that unlocks pages after they’ve been filled.
 */

#include <stdio.h>
#include <stdlib.h>
#include "assert.h"
#include <errno.h>
#include <sys/mman.h>
#include <pthread.h>
#include <zmq.h>

#include "util.h"
#include "strbuf.h"
#include "chunk.h"
#include "param.h"
#include "tidy.h"
#include "snapshot.h"

import void *snapshot_zmq_ctx;

private void *tidy;
private void *log;

/*
 * Establish tidy comms:  this routine gets called first of all threads, so it
 * creates the context.
 */

private char *create_tidy_comms(void **s) {
  if( !snapshot_zmq_ctx )
    snapshot_zmq_ctx = zmq_ctx_new();
  if( !snapshot_zmq_ctx ) {
    return "failed to create ZMQ context";
  }

  /* Create and initialise the sockets: */

  /* MAIN thread’s log socket */
  *s = zh_bind_new_socket(snapshot_zmq_ctx, ZMQ_PULL, LOG_SOCKET);
  if( *s == NULL ) {
    return "unable to create MAIN thread log socket";
  }

  /* TIDY’s socket for work messages */
  tidy = zh_bind_new_socket(snapshot_zmq_ctx, ZMQ_PAIR, TIDY_SOCKET);
  if(tidy == NULL)
    return "unable to create TIDY thread listener";

  /* TIDY’s socket for log messages */
  log = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_PUSH, LOG_SOCKET);
  if(log == NULL)
    return "unable to create TIDY thread log socket";

  return NULL;
}

/* Close the TIDY thread’s comms channels */

private void close_tidy_comms() {
  zmq_close(tidy);
  zmq_close(log);
}

/*
 * Unmap data blocks after writing.  Runs as a thread which continues
 * until a zero-length message is received signalling the end of the
 * unmap requests.  The argument passed is the address for the MAIN thread’s
 * log receiver socket, which is created here along with the context.
 */
```

```c
public void *tidy_main(void *arg) {
  char  *err;
  int    ret;
  frame *f;

  err = create_tidy_comms((void **)arg);
  if(err) {
    die_die_die_now++;
    return (void *) err;
  }

  zh_put_multi(log, 1, "TIDY  thread initialised");

  while( ret = zh_get_msg(tidy, 0, sizeof(frame *), &f) && !die_die_die_now ) {
    assertv(ret==sizeof(frame *), "TIDY read message error, ret=%d\n", ret);
    release_frame(f);
  }

  zh_put_multi(log, 1, "TIDY  thread terminates by return");

  /* Clean up our ZeroMQ sockets */
  close_tidy_comms();
  return (void *) "normal exit";
}
```

```
#

#ifndef _TIDY_H
#define _TIDY_H

#include "general.h"

export void *tidy_main(void *);

#define TIDY_SOCKET "inproc://snapshot-TIDY"

#endif /* _TIDY_H */
```

```
#

/*
 * Program to generate triggered snapshots manually.  This communicates with the
 * snapshotter via ZMQ.
 *
 * Arguments:
 * --verbose|-v         Increase reporting level
 * --quiet|-q           Decrease reporting level
 * --snapshot|-s        The snapshotter socket address
 * --pre                Pre-trigger interval
 * --post               Post-trigger interval
 * --trigger            Timepoint of trigger
 * --wait-for-it|-w     Wait for a key-press to generate trigger
 * --repeat|-r          Generate multiple triggers instead of just one
 * --auto|-a            Generate the snapshot name automatically
 * --help|-h            Print usage message
 * --version            Print program version
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include "assert.h"

#include <zmq.h>
#include <argtable2.h>
#include <regex.h>
#include <errno.h>
#include <string.h>
#include <getopt.h>
#include <time.h>

#include "argtab.h"
#include "util.h"
#include "param.h"
//#include "snapshot.h"

/*
 *  Program source version
 */

#define PROGRAM_VERSION "1.0"
#define VERSION_VERBOSE_BANNER  "MCLURS ADC toolset...\n"

/*
 * Auto-name format options
 */

#define AUTO_NAME_FORMAT_DEFAULT "iso"
#define AUTO_NAME_FORMAT_REX     "hex|iso|utc|tai|seq"

/*
 * Code for automatic snapshot name generation -- interprets format options
 */

typedef enum {
  SNAPNAME = 0,
  HEXADECIMAL,
  TAI64N,
  ISOUTC,
  ISODATE,
  SEQUENTIAL,
  SPECIAL,
} name_mode;

static name_mode determine_auto_mode(const char *auto_name) {
  if( !auto_name )
    return SNAPNAME;
  if( !strcmp("hex", auto_name) )
    return HEXADECIMAL;
  if( !strcmp("tai", auto_name) )
    return TAI64N;
  if( !strcmp("utc", auto_name) )
```

```
    return ISODATE;
  if( !strcmp("iso", auto_name) )
    return ISODATE;
  if( !strcmp("seq", auto_name) )
    return SEQUENTIAL;
  return SPECIAL;
}

/*
 * Global parameters for the snapshot program
 */

extern const char *snapshot_addr;
extern const char *auto_name;
extern uint32_t    window_pre;
extern uint32_t    window_pst;

param_t globals[] ={
  { "snapshot", "ipc://snapshot-CMD", &snapshot_addr, PARAM_TYPE(string), PARAM_SRC_ENV|PARAM_SRC_ARG,
    "address of snapshot command socket"
  },
  { "pre", "1000", &window_pre, PARAM_TYPE(int32), PARAM_SRC_ARG,
    "pre-trigger duration [ms]"
  },
  { "pst", "500", &window_pst, PARAM_TYPE(int32), PARAM_SRC_ARG,
    "post-trigger duration [ms]"
  },
  { "auto", AUTO_NAME_FORMAT_DEFAULT, &auto_name, PARAM_TYPE(string), PARAM_SRC_ARG,
    "format of auto-generated name"
  },
};

const int n_global_params =     (sizeof(globals)/sizeof(param_t));

/*
 * Debugging print out control
 */

int   verbose = 0;
char *program   = NULL;

/* Command line syntax options */

struct arg_lit *h1, *vn1, *v1, *q1;
struct arg_end *e1;

BEGIN_CMD_SYNTAX(help) {
  v1  = arg_litn("v",    "verbose", 0, 3,           "Increase verbosity"),
  q1  = arg_lit0("q",   "quiet",                    "Decrease verbosity"),
  h1  = arg_lit0("h",    "help",                    "Print usage help message"),
  vn1 = arg_lit0(NULL,  "version",                  "Print program version string"),
  e1  = arg_end(20)
} APPLY_CMD_DEFAULTS(help) {
  /* No defaults to apply here */
} END_CMD_SYNTAX(help)

struct arg_lit *v2, *q2, *w2;
struct arg_end *e2;
struct arg_str *u2;
struct arg_int *pb2, *pe2;
struct arg_str *n2;

BEGIN_CMD_SYNTAX(single) {
  v2  = arg_litn("v",    "verbose", 0, 3,           "Increase verbosity"),
  q2  = arg_lit0("q",   "quiet",                    "Decrease verbosity"),
  u2  = arg_str0("s",   "snapshot",  "<url>",      "URL of snapshotter command socket"),
  pb2 = arg_int0(NULL, "pre", "<int>",              "Pre-trigger interval [ms]"),
  pe2 = arg_int0(NULL, "pst,post", "<int>",         "Post-trigger interval [ms]"),
  w2  = arg_lit0("w", "wait-for-it",                "Wait for keypress to trigger"),
  n2  = arg_str1(NULL, NULL, "<snapshot name>",     "Name of the snapshot file"),
  e2  = arg_end(20)
} APPLY_CMD_DEFAULTS(single) {
  INCLUDE_PARAM_DEFAULTS(globals, n_global_params);
} END_CMD_SYNTAX(single);
```

```
struct arg_lit *v3, *q3, *w3;
struct arg_end *e3;
struct arg_str *u3;
struct arg_rex *a3;
struct arg_int *pb3, *pe3;

BEGIN_CMD_SYNTAX(autoname) {
  v3  = arg_litn("v",    "verbose", 0, 3,            "Increase verbosity"),
  q3  = arg_lit0("q",   "quiet",                     "Decrease verbosity"),
  u3  = arg_str0("s",   "snapshot",  "<url>",       "URL of snapshotter command socket"),
  pb3 = arg_int0(NULL,  "pre",  "<int>",             "Pre−trigger interval [ms]"),
  pe3 = arg_int0(NULL,  "pst,post",  "<int>",        "Post−trigger interval [ms]"),
  w3  = arg_lit0("w",  "wait−for−it",                "Wait for keypress to trigger"),
  a3  = arg_rex1("a", "auto", AUTO_NAME_FORMAT_REX, "<format>", REG_EXTENDED,        "Automatic snapshot name"),
  e3  = arg_end(20)
} APPLY_CMD_DEFAULTS(autoname) {
  a3−>hdr.flag |= ARG_HASOPTVALUE;
  INCLUDE_PARAM_DEFAULTS(globals, n_global_params);
} END_CMD_SYNTAX(autoname);

struct arg_lit *v4, *q4, *r4, *w4;
struct arg_end *e4;
struct arg_str *u4;
struct arg_rex *a4;
struct arg_int *pb4, *pe4;

BEGIN_CMD_SYNTAX(repeat) {
  v4  = arg_litn("v",    "verbose", 0, 3,            "Increase verbosity"),
  q4  = arg_lit0("q",   "quiet",                     "Decrease verbosity"),
  u4  = arg_str0("s",   "snapshot",  "<url>",       "URL of snapshotter command socket"),
  pb4 = arg_int0(NULL,  "pre",  "<int>",             "Pre−trigger interval [ms]"),
  pe4 = arg_int0(NULL,  "pst,post",  "<int>",        "Post−trigger interval [ms]"),
  a4  = arg_rex0("a", "auto", AUTO_NAME_FORMAT_REX, "<format>", REG_EXTENDED,        "Automatic snapshot name"),
        arg_rem(NULL,  "<format> is " AUTO_NAME_FORMAT_REX),
  w4  = arg_lit0("w",  "wait−for−it",                "Wait for keypress to trigger"),
  r4  = arg_lit1("r",  "repeat",                     "Loop, generating multiple triggers (implies −wa)"),
  e4  = arg_end(20)
} APPLY_CMD_DEFAULTS(repeat) {
  a4−>hdr.flag |= ARG_HASOPTVALUE;
  INCLUDE_PARAM_DEFAULTS(globals, n_global_params);
} END_CMD_SYNTAX(repeat);

/* Standard help routines: display the version banner */
void print_version(FILE *fp, int verbosity) {
  fprintf(fp, "%s: Vn. %s\n", program, PROGRAM_VERSION);
  if(verbosity > 0) {           /* Verbose requested... */
    fprintf(fp, VERSION_VERBOSE_BANNER);
  }
}

/* Standard help routines: display the usage summary for a syntax */
void print_usage(FILE *fp, void **argtable, int verbosity, char *program) {
  if( !verbosity ) {
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntax(fp, argtable, "\n");
    return;
  }
  if( verbosity ) {
    char *suffix = verbosity>1? "\n\n" : "\n";
    fprintf(fp, "Usage: %s ", program);
    arg_print_syntaxv(fp, argtable, suffix);
    if( verbosity > 1 )
      arg_print_glossary(fp, argtable, "%−25s %s\n");
  }
}

/*
 * Globals
 */

void   *zmq_main_ctx;           /* ZMQ context for messaging */

const char *auto_name;          /* Auto−generate snapshot path value */
name_mode   auto_mode;          /* The basis for snapshot name generation */
const char *snap_name;          /* The base name if not auto */
const char *snapshot_addr;      /* URL of the snapshotter program */
int         wait_for_it;        /* Wait for keypress before making message */
int         repeat;             /* Don’t just do one, do many triggers */
uint32_t    window_pre;         /* Window pre−trigger interval [ms] */
uint32_t    window_pst;         /* Window post−trigger interval [ms] */

/*
 * Process a (possibly multipart) log message.
 * Collect the various pieces and write to stderr
 * Use a 1024 byte logging buffer
 */

#define LOGBUF_SIZE     1024

void print_message(void *socket) {
  char log_buffer[LOGBUF_SIZE];
  int used;

  used = zh_collect_multi(socket, &log_buffer[0], LOGBUF_SIZE−1, "");
  if( log_buffer[used−1] != ’\n’) {
    log_buffer[used] = ’\n’;
    fwrite(log_buffer, used+1, 1, stdout);
  }
  else {
    fwrite(log_buffer, used, 1, stdout);
  }
  fflush(stdout);
}

/*
 * Wait for a keypress to generate a trigger time.
 *
 */

uint64_t wait_for_keypress(uint64_t *now_as_ns) {
  struct timespec now;

  if(now_as_ns == NULL)
    return −1;

  fputc(’>’, stdout);
  switch( fgetc(stdin) ) {
  case EOF:
  case ’q’:
    return −1;

  case ’s’:
    repeat = 0;
    break;

  default:
    break;
  }

  /* Discover the current time, as trigger point */
  clock_gettime(CLOCK_MONOTONIC, &now);
  *now_as_ns = now.tv_sec;
  *now_as_ns = *now_as_ns * 1000000000 + now.tv_nsec;
  return 0;
}

/*
 * Construct the name of a snapshot file.  Use the supplied name unless in auto mode.
 */

char *make_path_value(char buf[], int size, const char *snapname, uint64_t trigger, name_mode mode) {
  static int counter = 0;
  time_t     trig;
  uint64_t   secs;
  int        ns;
  int        used;
  struct tm *t;

  switch(mode) {
```

```
  case SNAPNAME:                /* Use the supplied snapshot name */
    snprintf(&buf[0], size, "%s", snapname);
    break;

  case TAI64N:                  /* Use a TAI64N format timestamp */
    assertv(size >= 25, "Buffer too small (%d) for TAI path\n", size);
    secs = trigger / 1000000000;
    ns = trigger - secs * 1000000000;
    snprintf(&buf[0], size, "@%016llx%08lx", secs|0x4000000000000000, ns);
    break;

  case ISODATE:
  case ISOUTC:                  /* Use an ISO standard date with fractional seconds */
    assertv(size >= 26, "Buffer too small (%d) for ISO path\n", size);
    trig = trigger / 1000000000;
    ns = trigger - trig * 1000000000;
    t = (mode==ISOUTC? gmtime(&trig) : localtime(&trig));
    used = strftime(&buf[0], size, "%FT%T", t); /* 2015-07-14T16:55:32.nnnnnn */
    snprintf(&buf[used], size-used, ".%06d", ns/1000);
    break;

  case HEXADECIMAL:             /* Use a hexadecimal print of the trigger time */
    assertv(size >= 16, "Buffer too small (%d) for HEX path\n", size);
    snprintf(&buf[0], size, "%016llx", trigger);
    break;

  case SEQUENTIAL:              /* Generate a sequentially incrementing snapshot name */
    assertv(size >= 10, "Buffer too small (%d) for SEQ path\n", size);
    snprintf(&buf[0], size, "snap%06d", counter++);
    break;

  case SPECIAL:                 /* User-supplied format, not yet implemented */
  default:
    snprintf(&buf[0], size, "%s%d", "unimplemented", counter++);
    break;
  }
  return &buf[0];
}

/*
 * Main entry point
 */

#define PATHBUF_SIZE 128

int main(int argc, char *argv[], char *envp[]) {
  char     buf[LOGBUF_SIZE];
  void    *snapshot;
  param_t *p;
  char    *v;
  int      ret, n;
  int      used, left;
  uint64_t trigger;
  struct timespec now;
  uint64_t now_as_ns;

  /* Discover the current time, as trigger point */
  clock_gettime(CLOCK_MONOTONIC, &now);
  now_as_ns = now.tv_sec;
  now_as_ns = now_as_ns * 1000000000 + now.tv_nsec;

  program = argv[0];

  /* Set up the standard parameters */
  /* 1. Process parameters:  internal default, then environment. */
  set_param_from_env(envp, globals, n_global_params);

  /* 2. Process parameters:  push values out to program globals */
  ret = assign_all_params(globals, n_global_params);
  assertv(ret == 0, "Push parameters failed on param %d out of %d\n", -ret, n_global_params);

  //  fprintf(stderr, "Before command line processing, after environment\n");
  //  debug_params(stderr, globals, n_global_params);

  /* 3. Create and parse the command lines -- installs defaults from parameter table */
```

```c
  void **cmd_help     = arg_make_help();
  void **cmd_single   = arg_make_single();
  void **cmd_autoname = arg_make_autoname();
  void **cmd_repeat   = arg_make_repeat();

  /* Try first syntax */
  int err_help = arg_parse(argc, argv, cmd_help);
  if( !err_help ) {             /* Assume this was the desired command syntax */
    int verbose = v1->count - q1->count;
    if(vn1->count)
      print_version(stdout, verbose);
    if(h1->count || !vn1->count) {
      print_usage(stdout, cmd_help, verbose>0, program);
      print_usage(stdout, cmd_single, verbose>0, program);
      print_usage(stdout, cmd_autoname, verbose>0, program);
      print_usage(stdout, cmd_repeat, verbose, program);
    }
    exit(0);
  }

  struct arg_end  *found = NULL;
  void           **table = NULL;
  int errs = 0, min_errs  = 100;

  /* Try remaining syntaxes */
  errs = arg_parse(argc, argv, cmd_single);
  if( !errs || errs < min_errs ) {      /* Choose single trigger manual-named mode */
    found = e2;
    table = cmd_single;
    verbose = v2->count - q2->count;
    min_errs = errs;
    if( !errs ) {
      auto_name = NULL;
      repeat = 0;
      wait_for_it = w2->count;
      snap_name = n2->sval[0];
    }
  }

  if( errs ) {
    errs = arg_parse(argc, argv, cmd_autoname);
    if( !errs || errs < min_errs) {      /* Choose single trigger auto-named mode */
      found = e3;
      table = cmd_autoname;
      verbose = v3->count - q3->count;
      min_errs = errs;
      if( !errs ) {
        repeat = 0;
        wait_for_it = w2->count;
        snap_name = NULL;
      }
    }
  }

  if( errs ) {
    errs = arg_parse(argc, argv, cmd_repeat);
    if( !errs || errs < min_errs ) {    /* Choose multi-trigger mode */
      found = e4;
      table = cmd_repeat;
      verbose = v4->count - q4->count;
      min_errs = errs;
      if( !errs ) {
        repeat = 1;
        wait_for_it = 1;
        if( !a4->count || !w4->count ) {
          if(verbose >= 0)
            fprintf(stderr, "%s: Warning -- repeat (-r) implies -a and -w, using --auto=%s\n", program, auto_name);
        }
        snap_name = NULL;
      }
    }
  }

  /* Now found indicates the command line with minimum errors in parse */
```

```
if( min_errs ) {              /* No command line matched precisely */
  arg_print_errors(stderr, found, program);
  print_usage(stderr, cmd_help, verbose>0, program);
  print_usage(stderr, cmd_single, verbose>0, program);
  print_usage(stderr, cmd_autoname, verbose>0, program);
  print_usage(stderr, cmd_repeat, verbose, program);
  exit(1);
}

//  fprintf(stderr, "After commandline choice, before reverse push\n");
//  debug_params(stderr, globals, n_global_params);

/* 4. Process parameters:  copy argument values back through the parameter table */
ret = arg_results_to_params(table, globals, n_global_params);

//  fprintf(stderr, "After reverse push\n");
//  debug_params(stderr, globals, n_global_params);

/* Check the auto argument and compute the path generation mode */
auto_mode = determine_auto_mode(auto_name);

/* 5. All syntax tables are finished with now: clean up the mess :-)) */
arg_free(cmd_help);
arg_free(cmd_single);
arg_free(cmd_autoname);
arg_free(cmd_repeat);

if(verbose > 2)               /* Dump global parameters for debugging purposes */
  debug_params(stderr, globals, n_global_params);

/* Create the ZMQ contexts */
zmq_main_ctx  = zmq_ctx_new();
if( !zmq_main_ctx ) {
  fprintf(stderr, "%s: Error -- ZeroMQ context creation failed: %s\n", program, strerror(errno));
  exit(2);
}

/* Create the socket to talk to the snapshot program */
snapshot = zh_connect_new_socket(zmq_main_ctx, ZMQ_REQ, snapshot_addr);
if( snapshot == NULL ) {
  fprintf(stderr, "%s: Error -- unable to create socket to snapshot at %s: %s\n",
          program, snapshot_addr, strerror(errno));
  zmq_ctx_term(zmq_main_ctx);
  exit(2);
}

/* Look at the parameters to construct the snap command */
if(window_pre + window_pst > 8000) {
  fprintf(stderr, "%s: Error -- maximum allowed capture window is 8000 [ms]\n");
  exit(3);
}

trigger = now_as_ns;
do {
  uint64_t time_start, time_stop;
  char  path_buf[PATHBUF_SIZE];
  const char *path;
  int   ret = 0;

  if( wait_for_it )
    ret = wait_for_keypress(&trigger);

  if( ret < 0 ) {
    break;
  }

  path = make_path_value(&path_buf[0], PATHBUF_SIZE-1, snap_name, trigger, auto_mode);

  // fprintf(stderr, "Window parameters pre %d post %d\n", window_pre, window_pst);

  time_start = trigger - 1000000 * (uint64_t) window_pre;
  time_stop  = trigger + 1000000 * (uint64_t) window_pst;

  // fprintf(stderr, "Window parameters start %lld stop %lld\n", time_start, time_stop);
```

```
    /* Send the message, wait for the reply */
    left = LOGBUF_SIZE−1;
    used = snprintf(&buf[0], left, "snap begin=%lld, end=%lld, path=%s", time_start, time_stop, path);
    buf[used] = '\0';
    if(verbose > 1)
      fprintf(stderr, "Sending: %s\n", &buf[0]);
    ret = zh_put_msg(snapshot, 0, used, buf);
    if( ret < 0 ) {
      fprintf(stderr, "%s: Error −− sending message to %s failed\n", program, snapshot_addr);
      break;
    }

    /* Wait for reply */
    if(verbose > 1)
      fprintf(stderr, "Awaiting reply...\n");
    if(verbose > 0)
      print_message(snapshot);

  } while(repeat);

  /* Clean up ZeroMQ sockets and context */
  zmq_close(snapshot);
  zmq_ctx_term(zmq_main_ctx);
  exit(0);
}
```

```
#

#include "general.h"

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include "assert.h"

#include <zmq.h>

#include "util.h"

/*
 * Create, open and bind a ZMQ socket.
 */

public void *zh_bind_new_socket(void *ctx, int type, const char *url) {
  void *skt;

  skt = zmq_socket(ctx, type);
  if(skt != NULL) {
    int ret = zmq_bind(skt, url);
    if(ret < 0) {
      int safe_errno = errno;
      (void) zmq_close(skt);
      errno = safe_errno;
      skt = NULL;
    }
  }
  return skt;
}

/*
 * Create, open and connect a ZMQ socket.
 */

public void *zh_connect_new_socket(void *ctx, int type, const char *url) {
  void *skt;

  skt = zmq_socket(ctx, type);
  if(skt != NULL) {
    int ret = zmq_connect(skt, url);
    if(ret < 0) {
      int safe_errno = errno;
      (void) zmq_close(skt);
      errno = safe_errno;
      skt = NULL;
    }
  }
  return skt;
}

/*
 * Retrieve a ZMG message from a socket.  Put it in the buffer buf and
 * transfer at most size bytes.  If size is zero, we care only about
 * the arrival of the message, not its content.
 */

public int zh_get_msg(void *socket, int flags, size_t size, void *buf) {
  zmq_msg_t  msg;
  int ret;
  size_t msg_size;

  ret = zmq_msg_init(&msg);
  assertv(ret == 0, "Message init failed\n");
  ret = zmq_msg_recv(&msg, socket, flags);
  if( ret < 0 )
    return ret;
  if( !size )
    return 0;
  msg_size = zmq_msg_size(&msg);
  if( !msg_size )
```

```
    return 0;
  if( msg_size < size )
    size = msg_size;
  assertv(buf != NULL, "Called with null buf argument\n");
  bcopy(zmq_msg_data(&msg), buf, size);
  ret = zmq_msg_close(&msg);
  assertv(ret == 0, "Message close failed\n");
  return size;
}

/*
 * Returns true if there is more of this message, otherwise false
 */

public int zh_any_more(void *socket) {
  int ret, more;
  size_t sz;

  sz = sizeof(more);
  ret = zmq_getsockopt(socket, ZMQ_RCVMORE, &more, &sz);
  assertv(ret == 0, "Attempt to get ’more’ flag failed\n");
  return more != 0;
}

/*
 * Get a multipart message in a single buffer.  Concatenate the
 * pieces, with ‘spc’ in between.  End with \0.  Return the size.
 */

public int zh_collect_multi(void *socket, char *buf, int bufsz, char *spc) {
 int used = 0,
     left = bufsz−1,
     nspc = strlen(spc);

  do {
    int ret, sz;

    sz = zh_get_msg(socket, 0, left−nspc, &buf[used]);
    assertv(sz >= 0, "Get message error\n");
    used += sz;
    left −= sz;
    if( !zh_any_more(socket) )
      break;
    bcopy(spc, &buf[used], nspc);
    used += nspc;
    left −= nspc;
  } while( left >= 0 );
  buf[used] =’\0’;
  return used;
}

/*
 * Send a ZMG message via a socket.  If size is zero, send an empty
 * frame, and buf can be NULL.  If ZMQ_SNDMORE is given as flag, this
 * is part of a multipart message.
 */

public int zh_put_msg(void *socket, int flags, size_t size, void *buf) {
  zmq_msg_t  msg;
  int ret;

  assertv(size >= 0, "Put message with −ve size %d\n", size);
  ret = zmq_msg_init_size(&msg, size);
  assertv(ret == 0, "Message init failed\n");
  if( size ) {
    assertv(buf != NULL, "Non−zero size and NULL buf\n");
    bcopy(buf, zmq_msg_data(&msg), size);
  }
  return zmq_msg_send(&msg, socket, flags);
}

/*
 * Send an n−frame message via a socket given an argument list of strings.
 */
```

```
public int zh_put_multi(void *socket, int n, ...) {
  va_list ap;
  int ret;

  va_start(ap,n);
  while( n-- > 0 ) {
    char *next = va_arg(ap, char *);
    int sz = strlen(next);
    ret = zh_put_msg(socket, (n==0? 0 : ZMQ_SNDMORE), sz, next);
    if( ret < 0 )
      return ret;
  }
  va_end(ap);
  return 0;
}
```

```
#

#ifndef _UTIL_H
#define _UTIL_H

#include "general.h"

#include <stdarg.h>
#include <unistd.h>
#include "assert.h"

#define true    1
#define false   0

#define WAIT_FOR_CONDITION(cond,limit)                                   \
  do { double l = (limit); int n = 0, max = 100*l;                       \
    while( n<max && !(cond) ) usleep(10000), n++;                        \
    assertv((cond), "Waited too long (%g [s]) for condition\n", l);      \
  } while(0)

/* Messaging utilities */

#include <zmq.h>

export int zh_get_msg(void *, int, size_t, void *);
export int zh_any_more(void *);

export int zh_put_msg(void *, int, size_t, void *);
export int zh_put_multi(void *, int, ...);

export void *zh_bind_new_socket(void *, int, const char *);
export void *zh_connect_new_socket(void *, int, const char *);

#endif /* _UTIL_H */
```

```
#

#include "general.h"

#define  _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/capability.h>
#include <time.h>
#include <pwd.h>
#include <grp.h>

#include "assert.h"

#include <zmq.h>
#include <pthread.h>

#include <comedi.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#include "util.h"
#include "param.h"
#include "queue.h"
#include "mman.h"
#include "strbuf.h"
#include "chunk.h"
#include "adc.h"
#include "snapshot.h"
#include "reader.h"
#include "writer.h"

/* We import the READER’s ADC object for its time conversion and activity check routines */
import adc reader_adc;

/*
 * --------------------------------------------------------------------------------
 *
 * TYPES INTERNAL TO THE WRITER THREAD
 *
 * -- snapshot descriptor
 * -- snapshot file descriptor
 * -- forward function declarations
 * -- local queue headers
 */

/* Snapshot Descriptor Structure */

typedef struct {                /* Private snapshot descriptor structure used by writer */
  queue       s_xQ[2];          /* Queue headers -- must be first member */
#define s_Q          s_xQ[0]    /* Active snapshot queue header */
#define s_fileQhdr   s_xQ[1]    /* Header for the queue of file descriptor structures */
  uint16_t    s_name;           /* ‘Name’ for snapshot */
  int         s_dirfd;          /* Dirfd of the samples directory */
  uint64_t    s_first;          /* First sample to collect for the next repetition */
  uint64_t    s_last;           /* Collect up to but not including this sample in the next repetition */
  uint32_t    s_samples;        /* Number of samples to save */
  int         s_bytes;          /* Total size of one sample file */
  uint32_t    s_count;          /* Repetition count for this snapshot */
  int         s_pending;        /* Count of pending repetitions */
  int         s_done;           /* Count of completed repetitions */
  int         s_status;         /* Status of this snapshot */
  const char *s_path;           /* Directory path for this snapshot */
  strbuf      s_error;          /* Error strbuf for asynchronous operation */
}
  snap_t;

/* Forward declarations of snapshot descriptor routines */
```

```
private uint16_t snapshot_name(snap_t *);

#define qp2snap(qp)  ((snap_t *)&(qp)[0])
#define snap2qp(s)   (&((s)->s_xQ[0]))

#define fq2snap(fq)  ((snap_t *)&(fg)[-1])
#define snap2fq(s)   (&((s)->s_xQ[1]))

#define qp2sname(p)  snapshot_name(qp2snap(p))

/* Snapshot File Descriptor Structure */

typedef struct _sfile {
  queue       f_Q;                      /* Queue header for file descriptor structures */
  snap_t     *f_parent;                 /* The snap_t structure that generated this file capture */
  int         f_fd;                     /* System file descriptor -- only needed while pages left to map */
  int         f_indexnr;                /* Index number of this file in the full set for the snapshot */
  int         f_nchunks;                /* Number of chunks allocated for this file transfer */
  int         f_written;                /* Number of chunks actually written so far */
  int         f_pending;                /* Number of chunks controlled by the READER thread */
  chunk_t    *f_chunkQ;                 /* Pointer to this file's writer chunk queue */
  int         f_status;                 /* Status flags for this file */
  strbuf      f_error;                  /* The strbuf to write error text into */
  uint16_t    f_name;                   /* Unique number for debugging */
  char        f_file[FILE_NAME_SIZE];   /* Name of this file:  the hexadecimal first sample number .s16 */
}
  snapfile_t;

/* Forward declarations of snapshot file descriptor routines needed by snapshot */

private snapfile_t *alloc_snapfile();
private int setup_snapfile(snapfile_t *, snap_t *);
private void abort_snapfile(snapfile_t *);
private void debug_snapfile(snapfile_t *);
import  uint16_t snapfile_name(snapfile_t *);

#define qp2file(p)      ((snapfile_t *)(p))
#define file2qp(f)      (&(f)->f_Q)

#define qp2fname(p)     snapfile_name(qp2file(p))

/* Local queue headers etc. used by the WRITER thread */

private QUEUE_HEADER(snapQ);            /* The list of active snapshots */
private QUEUE_HEADER(WriterChunkQ);     /* The list of chunks awaiting mapping, in order of first sample */

/*
 * --------------------------------------------------------------------------------
 *
 * INITIALISATION ROUTINES FOR WRITER THREAD:
 *
 * - Establish the communication endpoints needed
 * - Set up the required effective capabilities
 * - Set up RT priority scheduling (if requested)
 *
 * --------------------------------------------------------------------------------
 */

/*
 * Writer parameter structure.
 */

public wparams writer_parameters;

/* The values below are computed from writer_parameters by the verify() function */

private int wp_nframes;         /* Number of transfer frames prepared */
private int wp_chunksamples;    /* Number of samples in a chunk */
private int wp_snap_dirfd;      /* Snapdir path fd */
private int wp_snap_curfd;      /* Path fd of the 'working' directory */
private int wp_totxfrsamples;   /* Total scheduled transfer samples remaining */
private int wp_nfiles;          /* Number of files in progress */

/* Read-only access to chunk size, needed by READER */
```

```
public int writer_chunksize_samples() {
  return wp_chunksamples;
}

/*
 * Reader thread comms initialisation (failure is fatal).
 *
 * Called after the process-wide ZMQ context is created (elsewhere).
 */

private void *log;
private void *reader;
private void *command;

private void create_writer_comms() {
  import void *snapshot_zmq_ctx;
  /* Create necessary sockets */
  command  = zh_bind_new_socket(snapshot_zmq_ctx, ZMQ_REP, WRITER_CMD_ADDR);    /* Receive commands */
  assertv(command != NULL, "Failed to instantiate reader command socket\n");
  log      = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_PUSH, LOG_SOCKET);  /* Socket for log messages */
  assertv(log != NULL,      "Failed to instantiate reader log socket\n");
  reader   = zh_connect_new_socket(snapshot_zmq_ctx, ZMQ_PAIR, READER_QUEUE_ADDR);
  assertv(reader != NULL,  "Failed to instantiate reader queue socket\n");
}

/* CLose everything created above */

private void close_writer_comms() {
  zmq_close(log);
  zmq_close(reader);
  zmq_close(command);
}

/*
 * Copy the necessary capabilities from permitted to effective set (failure is fatal).
 *
 * The writer needs:
 *
 * CAP_IPC_LOCK -- ability to mmap and mlock pages.
 * CAP_SYS_NICE -- ability to set RT scheduling priorities
 * CAP_SYS_ADMIN (Writer) -- ability to set RT IO scheduling priorities (unused at present)
 *
 * These capabilities should be in the CAP_PERMITTED set, but not in CAP_EFFECTIVE which was cleared
 * when the main thread dropped privileges by changing to the desired non-root uid/gid.
 */

private int set_up_writer_capability() {
  cap_t c = cap_get_proc();
  const cap_value_t vs[] = { CAP_IPC_LOCK, CAP_SYS_NICE, CAP_SYS_ADMIN, };

  cap_set_flag(c, CAP_EFFECTIVE, sizeof(vs)/sizeof(cap_value_t), &vs[0], CAP_SET);
  return cap_set_proc(c);
}

/*
 * Set the WRITER thread to real-time priority, if RTPRIO is set...
 */

private int set_writer_rt_scheduling() {

  if( writer_parameters.w_schedprio > 0 ) {     /* Then there is RT priority scheduling to set up */
    if( set_rt_scheduling(writer_parameters.w_schedprio) < 0 )
      return -1;

    /* Successfully applied RT scheduling */
    return 1;
  }

  /* RT scheduling not applicable:  no RTPRIO set */
  return 0;
}

/*
 * Debug writer parameters
 */
```

```
private void debug_writer_params() {
  char buf[MSGBUFSIZE];
  wparams *wp = &writer_parameters;

  if(verbose<1)
    return;

  snprintf(buf, MSGBUFSIZE, "WRITER: TMPDIR=%s, SNAPDIR=%s, RTprio=%d;  WOF=%g;  FrameRAM = %d[MiB], ChunkSize = %d[kiB], nFrames = %d xfrSampleQ = %d[ki]\n",
           tmpdir_path, wp->w_snapdir, wp->w_schedprio, wp->w_writeahead,
           wp->w_lockedram, wp->w_chunksize, wp_nframes, wp_totxfrsamples/1024);
  zh_put_multi(log, 1, buf);
}

/*
 * --------------------------------------------------------------------------------
 *
 * UTILITY FUNCTIONS USED ONLY BY THE WRITER THREAD
 *
 * --------------------------------------------------------------------------------
 */

/*
 * Test for the presence of a directory by getting a path fd for it.
 */

private int test_directory(int dirfd, const char *name) {
  int ret;

  ret = openat(dirfd, name, O_PATH|O_DIRECTORY); /* Try to open the directory */
  return ret;
}


/*
 * Get a path handle to a directory, creating it if necessary.
 */

private int new_directory(int dirfd, const char *name) {
  int ret;

  ret = test_directory(dirfd, name);    /* Try to open the directory */
  if(ret < 0 ) {
    if( errno != ENOENT )               /* OK if it doesn’t exist, otherwise fail */
      return -1;
    ret = mkdirat(dirfd, name, 0750);   /* Didn’t exist, try to create it */
    if( ret < 0 )
      return -1;
    ret = openat(dirfd, name, O_PATH|O_DIRECTORY); /* Try again */
    if(ret < 0)                                  /* Give up on failure */
      return -1;
  }
  return ret;
}

/* ================================ Handle the Dir Command ================================ */

/*
 * Snapshot working directory parameter(s), used by the D command line.
 */

private param_t snapwd_params[] ={
#define SNAP_SETWD  0
  { "path",    NULL, NULL,
    PARAM_TYPE(string), PARAM_SRC_CMD,
    "working (sub-)directory for snapshots"
  },
};

private const int n_snapwd_params =  (sizeof(snapwd_params)/sizeof(param_t));

/*
 * Manage the writer’s ’working directory’:  clear the old, resetting to snapdir;
 * find/create and set a new one, clearing an old if necessary.
 */
```

```
private void clear_writer_wd() {
  int fd = wp_snap_curfd;

  if( fd != wp_snap_dirfd ) {
    wp_snap_curfd = wp_snap_dirfd;
    close(fd);
  }
}


private int set_writer_new_wd(const char *dir) {
  int fd;

  fd = new_directory(wp_snap_dirfd, dir);
  if(fd < 0)
    return −1;
  wp_snap_curfd = fd;
  return 0;
}

/*
 * Process a D command to change the working directory.  The command
 * comprises an introductory Dir verb followed by a  path=... parameter.
 */

private int process_dir_command(strbuf c) {
  strbuf   e   = strbuf_next(c);
  param_t *ps  = &snapwd_params[0];
  int      nps = n_snapwd_params;
  char    *path = NULL;
  int      err;

  /* Initialise the parameter value pointer */
  setval_param(&ps[SNAP_SETWD], (void **)&path);
  err = set_opt_params_from_string(strbuf_string(c), ps, nps);
  if(err < 0) {
    strbuf_appendf(e, "parameter parsing error at position %d", −err);
    reset_param(&ps[SNAP_SETWD]);
    return −1;
  }
  err = assign_param(&ps[SNAP_SETWD]);
  /* If this string copy fails, it’s a programming error! */
  assertv(err==0, "Dir PATH parameter assignment failed: %m");
  reset_param(&ps[SNAP_SETWD]);

  if( !path ) {                 /* No path supplied, reset to snapdir */
    clear_writer_wd();
    return 0;
  }

  /* Path is now instantiated to the given parameter string */
  if(set_writer_new_wd(path) < 0) {
    strbuf_appendf(e, "cannot create path=%s: %m", path);
    return −1;
  }
  return 0;
}

/* ================================ Handle the Snap command ================================ */

/*
 * Snapshot parameters, used by the S command line.
 * Local to this thread.
 *
 * Note the #defines, which are used to extract the parameter values
 * when building snapshot descriptors −− there is no need to search
 * for the parameter when we know exactly where it is.
 */

private param_t snapshot_params[] ={
#define SNAP_BEGIN  0
  { "begin",  NULL, NULL,
    PARAM_TYPE(int64), PARAM_SRC_CMD,
    "start time of snapshot [ns from epoch]"
  },
#define SNAP_END  1
  { "end",     NULL, NULL,
    PARAM_TYPE(int64), PARAM_SRC_CMD,
    "finish time of snapshot [ns from epoch]"
  },
#define SNAP_START  2
  { "start",  NULL, NULL,
    PARAM_TYPE(int64), PARAM_SRC_CMD,
    "start sample of snapshot"
  },
#define SNAP_FINISH 3
  { "finish", NULL, NULL,
    PARAM_TYPE(int64), PARAM_SRC_CMD,
    "end sample of snapshot"
  },
#define SNAP_LENGTH 4
  { "length", NULL, NULL,
    PARAM_TYPE(int32), PARAM_SRC_CMD,
    "length of snapshot [samples]"
  },
#define SNAP_COUNT  5
  { "count",    NULL, NULL,
    PARAM_TYPE(int32), PARAM_SRC_CMD,
    "repeat count of snapshot"
  },
#define SNAP_PATH  6
  { "path",     NULL, NULL,
    PARAM_TYPE(string), PARAM_SRC_CMD,
    "storage path of snapshot data"
  },
};

private const int n_snapshot_params = (sizeof(snapshot_params)/sizeof(param_t));

/*
 * -------------------------------------------------------------------------------
 * FUNCTIONS ETC. TO MANAGE SNAPSHOT DESCRIPTORS
 *
 * The writer maintains a list of "active" snapshot descriptors.  A descriptor
 * is created in response to an S command and is "active" until it has been both
 * (a) completely processed and also (b) reported back in response to a Z
 * command.  These data structures are entirely private to the writer.
 *
 * -------------------------------------------------------------------------------
 */

/*
 * Allocate and free snap_t structures
 */

private uint16_t snap_counter = 0;

private snap_t *alloc_snapshot() {
  snap_t *ret = calloc(1, sizeof(snap_t));

  if( !snap_counter ) snap_counter++; /* Avoid snapshots called 0000 */

  if(ret) {
    init_queue( snap2qp(ret) );
    ret->s_dirfd = -1;
    ret->s_name = snap_counter++;
    init_queue( snap2fq(ret) );
  }
  return ret;
}

private void free_snapshot(snap_t *s) {
  if( !queue_singleton(snap2qp(s)) )
    de_queue(snap2qp(s));
  assertv(queue_singleton(snap2fq(s)),
          "Freeing snapshot %p with non-empty file queue %p", s, queue_next(snap2fq(s)));
  if(s->s_dirfd >= 0)
    close(s->s_dirfd);
  if(s->s_path)
```

```
    free((void *)s->s_path);
  free( (void *)s );
}

/* Debugging routine to return unique name */
uint16_t snapshot_name(snap_t *s) {
  return s->s_name;
}

/*
 * Display snapshot status codes
 */

const char *snapshot_status(int st) {
  private const char *stab[] = {
    "INI", "ERR", "PRP", "RDY", "…", ">>>", "+++", "DON", "FIN",
  };
  if(st>=0 && st<sizeof(stab)/sizeof(char *))
    return stab[st];
  return "???";
}

/*
 * Manage the writer snapshot queue:
 *
 * - Check the parameters in an S command
 */

private int check_snapshot_params(param_t ps[], strbuf e) {
  int ret;

  /* path= is MANDATORY */
  if( !ps[SNAP_PATH].p_str ) {
    strbuf_appendf(e, "missing PATH parameter");
    return -1;
  }
  ret = assign_param(&ps[SNAP_PATH]);
  /* If this string copy fails, it's a programming error! */
  assertv(ret==0, "Snapshot PATH parameter assignment failed: %m");

  /* EITHER begin= OR start= is MANDATORY */
  if( !ps[SNAP_BEGIN].p_str && !ps[SNAP_START].p_str ) {
    strbuf_appendf(e, "neither BEGIN nor START present");
    return -1;
  }

  /* IF begin= THEN end= XOR length= AND NOT finish= is REQUIRED */
  if( ps[SNAP_BEGIN].p_str ) {
    if( ps[SNAP_FINISH].p_str ) {
      strbuf_appendf(e, "BEGIN with FINISH present");
      return -1;
    }
    if( !ps[SNAP_END].p_str && !ps[SNAP_LENGTH].p_str ) {
      strbuf_appendf(e, "BEGIN but neither END nor LENGTH present");
      return -1;
    }
    if( ps[SNAP_END].p_str && ps[SNAP_LENGTH].p_str ) {
      strbuf_appendf(e, "BEGIN with both END and LENGTH present");
      return -1;
    }
    ret = assign_param(&ps[SNAP_BEGIN]); /* Error implies bad number */
    if(ret < 0) {
      strbuf_appendf(e, "cannot assign BEGIN value %s: %m", ps[SNAP_BEGIN].p_str);
      return -1;
    }
    if(ps[SNAP_END].p_str) {
      ret = assign_param(&ps[SNAP_END]); /* Error implies bad number */
      if(ret < 0) {
        strbuf_appendf(e, "cannot assign END value %s: %m", ps[SNAP_END].p_str);
        return -1;
      }
    }
    if(ps[SNAP_LENGTH].p_str) {
      ret = assign_param(&ps[SNAP_LENGTH]); /* Error implies bad number */
      if(ret < 0) {
        strbuf_appendf(e, "cannot assign LENGTH value %s: %m", ps[SNAP_LENGTH].p_str);
        return -1;
      }
    }
  }

  /* IF start= THEN finish= XOR length= AND NOT end= is REQUIRED */
  if( ps[SNAP_START].p_str ) {
    if( ps[SNAP_END].p_str ) {
      strbuf_appendf(e, "START with END present");
      return -1;
    }
    if( !ps[SNAP_FINISH].p_str && !ps[SNAP_LENGTH].p_str ) {
      strbuf_appendf(e, "START but neither FINISH nor LENGTH present");
      return -1;
    }
    if( ps[SNAP_FINISH].p_str && ps[SNAP_LENGTH].p_str ) {
      strbuf_appendf(e, "START with both FINISH and LENGTH present");
      return -1;
    }
    ret = assign_param(&ps[SNAP_START]); /* Error implies bad number */
    if(ret < 0) {
      strbuf_appendf(e, "cannot assign START value %s: %m", ps[SNAP_START].p_str);
      return -1;
    }
    if(ps[SNAP_FINISH].p_str) {
      ret = assign_param(&ps[SNAP_FINISH]); /* Error implies bad number */
      if(ret < 0) {
        strbuf_appendf(e, "cannot assign FINISH value %s: %m", ps[SNAP_FINISH].p_str);
        return -1;
      }
    }
    if(ps[SNAP_LENGTH].p_str) {
      ret = assign_param(&ps[SNAP_LENGTH]); /* Error implies bad number */
      if(ret < 0) {
        strbuf_appendf(e, "cannot assign LENGTH value %s: %m", ps[SNAP_LENGTH].p_str);
        return -1;
      }
    }
  }

  /* count= is OPTIONAL */
  if(ps[SNAP_COUNT].p_str) {
      ret = assign_param(&ps[SNAP_COUNT]); /* Error implies bad number */
      if(ret < 0) {
        strbuf_appendf(e, "cannot assign COUNT value %s: %m", ps[SNAP_COUNT].p_str);
        return -1;
      }
    }

  /* All required parameters present in legal combination and values parse */
  return 0;
}

/*
 * Complete the snap_t structure sample-range contents -- we know the parameter
 * subset is correct We can also assume that the various members of the snap_t
 * structure have been instantiated by parameter assignment handled by the
 * caller.  We need the param[] array to determine which case we are handling.
 * No errors can occur here because they are dealt with by the caller(s) of this
 * routine.
 */

private void setup_snapshot_samples(snap_t *s, param_t p[]) {

  /* Start with length= -- if present, no finish= or end= spec. needed */
  if( p[SNAP_LENGTH].p_str ) {  /* Length was stored in s_samples, round up to integral number of pages */
    s->s_bytes = s->s_samples * sizeof(sampl_t);
    s->s_bytes += (sysconf(_SC_PAGE_SIZE) - (s->s_bytes % sysconf(_SC_PAGE_SIZE))) % sysconf(_SC_PAGE_SIZE);
    s->s_samples = s->s_bytes / sizeof(sampl_t);
  }

  /* Mandatory EITHER begin= OR start= -- it was begin= */
  if( p[SNAP_BEGIN].p_str ) {   /* Begin time was stored in s_first */
    s->s_first = adc_time_to_sample(reader_adc, s->s_first);
    s->s_first = s->s_first - (s->s_first % NCHANNELS);   /* Fix to NCHANNELS boundary */
    if( !s->s_samples ) {                                 /* No length given, need end from s_last */
      s->s_last = adc_time_to_sample(reader_adc, s->s_last);
      s->s_last = s->s_last + ((NCHANNELS - (s->s_last % NCHANNELS)) % NCHANNELS); /* Round up to integral number of channel sweeps */
      s->s_samples = s->s_last - s->s_first;
      s->s_bytes = s->s_samples * sizeof(sampl_t);
      s->s_bytes += (sysconf(_SC_PAGE_SIZE) - (s->s_bytes % sysconf(_SC_PAGE_SIZE))) % sysconf(_SC_PAGE_SIZE);
      s->s_samples = s->s_bytes / sizeof(sampl_t);          /* Round up to integral number of system pages */
    }
    s->s_last = s->s_first + s->s_samples;                  /* Calculate end point using rounded-up sample count */
  }

  /* Mandatory EITHER begin= OR start= -- it was start= */
  if( p[SNAP_START].p_str ) {   /* Start sample was stored in s_first */
    if( !s->s_samples ) {       /* No length given, need end from s_last */
      s->s_last += ((NCHANNELS - (s->s_last % NCHANNELS)) % NCHANNELS); /* Round up to integral number of channel sweeps */
      s->s_samples = s->s_last - s->s_first;                /* Compute requested length */
      s->s_bytes = s->s_samples * sizeof(sampl_t);
      s->s_bytes += (sysconf(_SC_PAGE_SIZE) - (s->s_bytes % sysconf(_SC_PAGE_SIZE))) % sysconf(_SC_PAGE_SIZE);
      s->s_samples = s->s_bytes / sizeof(sampl_t);          /* Round up to integral number of system pages */
    }
    s->s_last = s->s_first + s->s_samples;                  /* Calculate end point using rounded-up sample count */
  }

  /* Optional count=, default is 1 */
  if( !p[SNAP_COUNT].p_str ) { /* The count parameter was written to s_count */
    s->s_count = 1;
  }

  s->s_pending = 0;
  s->s_status  = 0;
}

/*
 * Build snapshot from S command line: the main thread passes a ring
 * of strbufs comprising the command buffer and the error buffer.
 *
 * The sequence of operations is:
 * - allocate a snap_t structure and bind the parameter val pointers to it
 * - populate the parameter structures from the string in the command buffer
 * - check the parameter set for correctness (check_snapshot_params)
 * - check the snapshot path and create the dirfd
 * - populate the sample value elements (setup_snapshot_samples)
 * - return the complete structure
 *
 * Errors arising during the above process cause an error status mark and are
 * reported in the error buffer.
 */

private snap_t *build_snapshot_descriptor(strbuf c) {
  strbuf      e   = strbuf_next(c);
  param_t    *ps  = &snapshot_params[0];
  int         nps = n_snapshot_params;
  const char *path = NULL;
  snap_t     *ret;
  int         err;
  int         i;

  if( !(ret = alloc_snapshot()) ) { /* Allocation failed */
    strbuf_appendf(e, "unable to allocate snapshot descriptor: %m");
    return ret;
  }

  /* Initialise the targets for the parameters */
  setval_param(&ps[SNAP_BEGIN],  (void **) &ret->s_first);
  setval_param(&ps[SNAP_END],    (void **) &ret->s_last);
  setval_param(&ps[SNAP_START],  (void **) &ret->s_first);
  setval_param(&ps[SNAP_FINISH], (void **) &ret->s_first);
  setval_param(&ps[SNAP_LENGTH], (void **) &ret->s_samples);
  setval_param(&ps[SNAP_COUNT],  (void **) &ret->s_count);
  setval_param(&ps[SNAP_PATH],   (void **) &path);

  /* Process the S command parameters */
  err = set_params_from_string(strbuf_string(c), ps, nps);
  if(err < 0) {                 /* Error parsing command string */
```

```
    strbuf_appendf(e, "parameter parsing error at position %d", -err);
    goto FAIL;
  }
  /* Check the populated parameters and assign to values */
  err = check_snapshot_params(ps, e);
  if(err < 0) {                  /* Problems put into strbuf by check function */
    goto FAIL;
  }

  if(ret->s_last <= ret->s_first) { /* Parameter error:  end before start */
    strbuf_appendf(e, "end %016llx before start %016llx", ret->s_last, ret->s_first);
    goto FAIL;
  }

  /* Path may not already exist */
  ret->s_dirfd = test_directory(wp_snap_curfd, path);
  if(ret->s_dirfd >= 0) {       /* Then directory already exists */
    strbuf_appendf(e, "requested dir path=%s already exists", path);
    goto FAIL;
  }

  if( !adc_is_running(reader_adc) ) {
    strbuf_appendf(e, "data acquisition is currently stopped", path);
    goto FAIL;
  }

  /* Now try to create required directory */
  ret->s_dirfd = new_directory(wp_snap_curfd, path);
  if(ret->s_dirfd < 0) {
    strbuf_appendf(e, "unable to create dir path=%s: %m", path);
    goto FAIL;
  }
  ret->s_path = strdup(path);

  /* Set up the sample-dependent values -- cannot fail */
  setup_snapshot_samples(ret, ps);

  /* Finished with the parameters, their values etc. now */
  for(i=0; i<nps; i++) reset_param(&ps[i]);

  /* All done, no errors */
  ret->s_status = SNAPSHOT_PREPARE; /* Structure complete but no files/chunks yet... */
  return ret;

 FAIL:
  for(i=0; i<nps; i++) reset_param(&ps[i]);
  free_snapshot(ret);
  return NULL;
}

/*
 * Set up snapshot -- create the necessary file descriptor structures etc.
 */

private void setup_snapshot(snap_t *s) {
  snapfile_t *f = alloc_snapfile();

  if(f == NULL) {
    strbuf_appendf(s->s_error, "Failed to allocate file %d/%d", s->s_pending+s->s_done+1, s->s_count);
    s->s_status = SNAPSHOT_ERROR;
    return;
  }
  if( setup_snapfile(f, s) < 0 ) {
    s->s_status = SNAPSHOT_ERROR;
    return;
  }
  s->s_first += s->s_samples; /* Move current sample indices to next file */
  s->s_last  += s->s_samples;
  debug_snapfile(f);
}

/*
 * Called when a snapshot file has just been written.
 */
```

```
private void refresh_snapshot(snap_t *s) {

  if(s->s_status == SNAPSHOT_ERROR) {   /* Tidy up after an error */
    while(s->s_pending) {                       /* There are files that have not got the message */
      assertv(!queue_singleton(snap2fq(s)),
              "Pending file count %d and file header Q mismatch in snapshot %p\n", s->s_pending, s);
      abort_snapfile(qp2file(queue_next(snap2fq(s))));
    }
    return;
  }
  else if(s->s_done == s->s_count) {    /* No files left to request */
    s->s_status = SNAPSHOT_COMPLETE;
    return;
  }
  else if(s->s_done + s->s_pending == s->s_count) {  /* All required files are in progress */
    return;
  }
  else {        /* See if this snapshot should have another file */
    if(wp_nfiles < 2 || s->s_pending == 0) {
      setup_snapshot(s);
    }
  }
}

/*
 * Debugging function for snapshot descriptors...
 */

private void debug_snapshot_descriptor(snap_t *s) {
  char buf[MSGBUFSIZE];

  snprintf(buf, MSGBUFSIZE,
           "Snap %04hx at %p: path '%s' fd %d status %s "
           "sQ[s:%04hx,s:%04hx] "
           "fQ[f:%04hx,f:%04hx] "
           "files %d/%d/%d "
           "S:%08lx B:%08lx F:%016llx L:%016llx\n",
           s->s_name, s, s->s_path, s->s_dirfd, snapshot_status(s->s_status),
           qp2sname(queue_prev(&s->s_Q)), qp2sname(queue_next(&s->s_Q)),
           qp2fname(queue_prev(&s->s_fileQhdr)), qp2fname(queue_next(&s->s_fileQhdr)),
           s->s_done, s->s_pending, s->s_count,
           s->s_samples, s->s_bytes, s->s_first, s->s_last);
  zh_put_multi(log, 1, &buf[0]);
}

/* ================================ Handle a Z(Status) Command ================================ */

/*
 * Snapshot status request parameter(s), used by the Z command line.
 */

private param_t status_params[] ={
#define SNAP_NAME  0
  { "name",    NULL, NULL,
    PARAM_TYPE(int16), PARAM_SRC_CMD,
    "snapshot name"
  },
};

private const int n_status_params =  (sizeof(status_params)/sizeof(param_t));

/*
 * The snapshot s should report its status as follows.  If it is a
 * pending snapshot, it should append a status line to the given
 * strbuf x.  If it is completed (with or without error) it should
 * transfer its own error strbuf to the chain by inserting it
 * immediately following x.  The idea is that on success the caller
 * will ignore the c strbuf and the chain following will give status
 * reports for completed snapshots..
 */

private void snapshot_report_status(strbuf x, snap_t *s) {

  if(s->s_status == SNAPSHOT_DONE) {     /* If completed, attach its error strbuf */
    queue_ins_after(strbuf2qp(x), strbuf2qp(s->s_error));
    s->s_error = (strbuf)NULL;
    return;
  }
  /* Snapshot is in progress:  append a status line to x */
  strbuf_appendf(x, "Snap %04hx: %s %d/%d/%d\n",
                 s->s_name, snapshot_status(s->s_status),
                 s->s_done, s->s_pending, s->s_count);
}

/*
 * Process a Z command to collect and return snapshot status.  The command
 * comprises an introductory Z verb followed by an optional name=... parameter.
 *
 * The caller has written an initial NO: prefix into the e strbuf, for
 * the error case.  For success, it will rewrite an OK line.  The c
 * strbuf is not cleared here or in the caller, since it is used by
 * snapshot_report_status for snapshots in progress.
 */

private int process_status_command(strbuf c) {
  strbuf   e     = strbuf_next(c);
  param_t *ps    = &status_params[0];
  int      nps   = n_status_params;
  uint16_t name  = 0;
  int      err;
  snap_t  *s     = NULL;

  /* Initialise the parameter value pointer */
  setval_param(&ps[SNAP_NAME], (void **)&name);
  err = set_opt_params_from_string(strbuf_string(c), ps, nps);
  if(err < 0) {
    strbuf_appendf(e, "parameter parsing error at position %d", -err);
    reset_param(&ps[SNAP_NAME]);
    return -1;
  }
  err = assign_param(&ps[SNAP_NAME]);
  /* If this string copy fails, it's a programming error! */
  assertv(err==0, "Status NAME parameter assignment failed: %m");
  reset_param(&ps[SNAP_NAME]);

  if(queue_singleton(&snapQ)) { /* There are no snapshots in the queue */
    if(name) {
      strbuf_appendf(e, "Snapshot %hd not found: queue empty", name);
      return -1;
    }
    else {
      strbuf_printf(c, " Files: %d, Xfr samples %d [Mi]\n",
                    wp_nfiles, wp_totxfrsamples/(1024*1024));
      return 0;
    }
  }

  if(name) {                    /* A spcific snapshot is requested */
    for_nxt_in_Q(queue *p, queue_next(&snapQ), &snapQ)
      if(name == qp2snap(p)->s_name) {
        s = qp2snap(p);
        break;
      }
    end_for_nxt;
    if(s == NULL) {
      strbuf_appendf(e, "Snapshot %hd not found", name);
      return -1;
    }
    /* ... we got one */
    strbuf_printf(c, "\n");
    snapshot_report_status(c, s);
    if(s->s_status == SNAPSHOT_DONE)     /* If completed, free it */
      free_snapshot(s);
  }
  else {        /* Otherwise, look at all the snapshots in the queue */
    /*
     * Note that, the loop below, we alter the queue being traversed
     * since free_snapshot unlinks the current snapshot.  This is OK,
     * since the loop macros have already determined whether the node
     * being worked is the last one or not.
     */
    strbuf_printf(c, " Files: %d, Xfr space %d [MiB]\n", wp_nfiles, wp_totxfrsamples*sizeof(sampl_t)/(1024*1024));
    for_nxt_in_Q(queue *p, queue_next(&snapQ), &snapQ)
      s = qp2snap(p);
      snapshot_report_status(c, s);      /* Report the status of each one */
      if(s->s_status == SNAPSHOT_DONE)   /* If completed, free it */
        free_snapshot(s);
    end_for_nxt;
  }
  return 0;
}

/* ============================== Deal with the Snapshot File Queue ============================== */

/*
 * --------------------------------------------------------------------------------
 * FUNCTIONS ETC. FOR SNAPSHOT FILE DESCRIPTOR STRUCTURES:  ONE OF THESE PER FILE TO CAPTURE.
 *
 * --------------------------------------------------------------------------------
 */


/*
 * Allocate and free snapfile_t structures
 */

private uint16_t snapfile_counter;

private snapfile_t *alloc_snapfile() {
  snapfile_t *ret = calloc(1, sizeof(snapfile_t));

  if(ret) {
    init_queue(&ret->f_Q);
    ret->f_fd = -1;
    ret->f_name = ++snapfile_counter;
  }
  return ret;
}

private void free_snapfile(snapfile_t *f) {
  if(f->f_fd >= 0)
    close(f->f_fd);
  assertv(f->f_chunkQ == NULL, "Freeing snapfile %p with remaining chunks %p\n", f, f->f_chunkQ);
  free((void *)f);
}

/* Debugging routine to return unique name */
public uint16_t snapfile_name(snapfile_t *f) {
  return f->f_name;
}

/*
 * Initialise a snapfile_t structure from a snap_t structure.
 */

private int setup_snapfile(snapfile_t *f, snap_t *s) {
  wparams *wp = &writer_parameters;
  int fd;
  int ret;

  f->f_indexnr = s->s_done+s->s_pending;

  snprintf(&f->f_file[0], FILE_NAME_SIZE, "%016llx.s16", s->s_first);
  fd = openat(s->s_dirfd, &f->f_file[0], O_RDWR|O_CREAT|O_EXCL, 0600);
  if(fd < 0) {
    strbuf_appendf(s->s_error, "Unable to open sample file %s in path %s: %m\n", &f->f_file[0], s->s_path);
    return -1;
  }

  ret = ftruncate(fd, s->s_bytes); /* Pre-size the file */
  if(ret < 0) {                    /* Try to tidy up... */
    strbuf_appendf(s->s_error, "Unable to truncate sample file %s to size %d [B]: %m\n", &f->f_file[0], s->s_bytes);
    unlinkat(s->s_dirfd, &f->f_file[0], 0);
    close(fd);
    return -1;
```

```
}

/* Allocate and initialise the chunks */
int nc = s->s_bytes / wp->w_chunksize; /* Number of milli-chunks to use (because chunksize is in [kiB] */
f->f_nchunks = (nc+1023) / 1024;
f->f_chunkQ  = alloc_chunk(f->f_nchunks);
if( f->f_chunkQ == NULL ) {
  strbuf_appendf(s->s_error, "Cannot allocate %d chunks for file %s: %m\n", f->f_nchunks, &f->f_file[0]);
  unlinkat(s->s_dirfd, &f->f_file[0], 0);
  close(fd);
  return -1;
}

/* Basic book-keeping entries from here:  no options for failure */
f->f_fd      = fd;
f->f_parent  = s;
f->f_error   = s->s_error;
f->f_written = 0;

/*
 * This next variable accounts for the number of samples we have
 * committed to write.  It is initialised by verify() from the
 * locked RAM and overbooking parameters.
 *
 * It is decremented here when we set up a file for capture.  It is
 * later incremented in one of two places: for a successfully
 * written chunk it is incremented by the queue message handler; for
 * a failed chunk, it is incremented by abort_file when it processes
 * the chunks in the file's chunk list.
 */

wp_totxfrsamples -= s->s_samples;

/* Go through the chunk queue writing in data */
uint64_t first  = s->s_first;
uint64_t rest   = s->s_samples;
uint32_t chunk  = wp_chunksamples;
uint32_t offset = 0;

for_nxt_in_Q(queue *p, chunk2qp(f->f_chunkQ), chunk2qp(f->f_chunkQ))
  chunk_t *c = qp2chunk(p);
  /* Determine chunk parameters */
  c->c_status  = SNAPSHOT_READY;
  c->c_parent  = f;
  c->c_error   = f->f_error;
  c->c_fd      = f->f_fd;
  c->c_ring    = NULL;        /* The ADC object computes this pointer */
  c->c_frame   = NULL;        /* The transfer frames are allocated elsewhere */
  c->c_first   = first;
  if(rest > chunk && rest < 2*chunk) /* Deal with final partial chunk(s) */
    chunk = rest / 2;
  c->c_samples = chunk;
  c->c_last    = first + chunk;
  c->c_offset  = offset;
  offset += chunk*sizeof(sampl_t);
  first  += chunk;

  /* Add the chunk to the WRITER chunk queue */
  queue *pos = &WriterChunkQ;
  if( !queue_singleton(&WriterChunkQ) ) {
    for_nxt_in_Q(queue *p, queue_next(&WriterChunkQ), &WriterChunkQ);
      chunk_t *h = rq2chunk(p);
      if(h->c_first > c->c_first) {
        pos = p;
        break;
      }
    end_for_nxt;
  }
  queue_ins_before(pos, chunk2rq(c));

end_for_nxt;

f->f_status = SNAPSHOT_READY;
s->s_pending++;
wp_nfiles++;             /* One more file in progress */
```

```
  return 0;
}

/*
 * Completed file descriptor -- called when file acquisition ends,
 * both normally and exeptionally.
 *
 *  We assume that the READER has cleared up any assigned frames when
 * deleting the file chunks in the READER queue.  Therefore, at this
 * point, only the file on disk remains -- remove it if there was an
 * error.  Adjust the book-keeping in the snap_t structure to show
 * this file as done.  Release the chunk descriptors.
 *
 * The file is finally written/gone when the TIDY thread has unmapped
 * the frames released by the READER.
 */

private void completed_snapfile(snapfile_t *f) {
  snap_t  *s = f->f_parent;

  if(f->f_fd >= 0)
    close(f->f_fd);

  s->s_pending--;
  wp_nfiles--;          /* One less file in progress */

  release_chunk(f->f_chunkQ);   /* Finished with these now */
  f->f_chunkQ = NULL;

  if(f->f_status == SNAPSHOT_ERROR) {
    s->s_status = SNAPSHOT_ERROR;
    unlinkat(s->s_dirfd, &f->f_file[0], 0);  /* If the file failed, remove it */
  }
  else {
    s->s_done++;                /* This file is done, it was pending before */
    if(s->s_done == s->s_count) {
      s->s_status = SNAPSHOT_COMPLETE;
      strbuf_printf(s->s_error, "OK Snap %04hx: FIN %d/%d files", snapshot_name(s), s->s_done, s->s_count);
    }
  }
  de_queue(file2qp(f));         /* Remove this one from the snapshot */
  free_snapfile(f);             /* And free the structure */
}

/*
 * Abort a file from the WRITER thread's viewpoint: remove all chunks
 * from the WRITER's chunk queue and mark the file in ERROR state.
 *
 * N.B. The READER AND WRITER both use the rq chunk linkage, and keep
 * track of who has it by means of exchanged messages.
 *
 * Adjust the w_totxfrsamples parameter to match new situation.
 */

private void abort_snapfile(snapfile_t *f) {
  snap_t  *s = f->f_parent;

  f->f_status = SNAPSHOT_ERROR;

  assertv(f->f_chunkQ != NULL, "Aborted file f:%04hx at %p has an empty chunk queue\n", snapfile_name(f), f);

  for_nxt_in_Q(queue *p, chunk2qp(f->f_chunkQ), chunk2qp(f->f_chunkQ));
    chunk_t *c = qp2chunk(p);
    if(queue_singleton(chunk2rq(c))) {      /* These were chunks in the READER queue */
      if(c->c_status == SNAPSHOT_WAITING) { /* These were pending chunks in transit from WRITER to READER */
        c->c_status = SNAPSHOT_ERROR;
        wp_totxfrsamples += c->c_samples;
      }
      continue;
    }
    de_queue(chunk2rq(c));                  /* Remove from WRITER chunk queue */
    if(c->c_status == SNAPSHOT_ERROR)       /* Release the write commitment for this chunk */
      wp_totxfrsamples += c->c_samples;
  end_for_nxt;
}
```

```
/*
 * Emit debugging data for a given file descriptor.
 */

private void debug_snapfile(snapfile_t *f) {
  snap_t *s = f->f_parent;
  int     left = MSGBUFSIZE-1,
          used = 0;
  int     i;
  char    buf[MSGBUFSIZE];

  used = snprintf(&buf[used], left,
                  "File %s (f:%04hx) of snapshot %04hx, at %p: "
                  "Q [f:%04hx,f:%04hx] "
                  "fd %d ix %d nc %d/%d st %s\n",
                  &f->f_file[0], f->f_name, s->s_name, f,
                  qp2fname(queue_prev(&f->f_Q)), qp2fname(queue_next(&f->f_Q)),
                  f->f_fd, f->f_indexnr, f->f_written, f->f_nchunks, snapshot_status(f->f_status)
                  );
  if(used >= left) used = left;
  left -= used;
  i = 0;
  for_nxt_in_Q(queue *p, chunk2qp(f->f_chunkQ), chunk2qp(f->f_chunkQ))
    int u = snprintf(&buf[used], left,
                     " > %03d: ", i++);
    if(u >= left) u = left;
    used += u;
    left -= u;
    u = debug_chunk(&buf[used], left, qp2chunk(p));
    used += u;
    left -= u;
  end_for_nxt;
  zh_put_multi(log, 1, &buf[0]);
}

/*
 * --------------------------------------------------------------------------------
 *
 * MAIN LOOP TASKS: deal with command and queue messages as they arrive and transfer
 *                  chunks to the READER when possible.
 *
 *  --------------------------------------------------------------------------------
 */

/*
 * Service the WRITER queue, i.e try to find frames to attach to
 * chunks, and pass such chunks to the READER.  Steps are:
 *
 * - check to see what is in the WRITER queue
 * - allocate at least one frame and pass chunk to READER
 * - loop while time remains...
 *
 * The READER receives messages for chunks that are now in Waiting
 * state, and it returns chunks in either Written state or Error
 * state.
 *
 * Note that when the READER returns a chunk in error state there may
 * be other chunks in transit as messages between the WRITER and
 * READER...  The WRITER needs to keep track of these and make sure
 * they are released in an orderly fashion.
 */

private uint64_t writer_service_queue(uint64_t start) {
  uint64_t now  = start;
  uint64_t stop = start + WRITER_MAX_CHUNK_DELAY;
  int      max;

  for(max=WRITER_MAX_CHUNKS_TRANSFER; max > 0 && !queue_singleton(&WriterChunkQ) && now < stop; --max) { /* Only ever do max chunks at the most */
    chunk_t *c = rq2chunk(queue_next(&WriterChunkQ));

    if( map_chunk_to_frame(c) < 0 ) {
      if(c->c_status == SNAPSHOT_ERROR) { /* Something nasty went wrong! */
        abort_snapfile(c->c_parent);
      }
      max = 0;                  /* Couldn’t get a frame, so we are done */
    }
    else {                      /* We succeeded */
      de_queue(chunk2rq(c));    /* Hand the chunk over to the READER thread */
      c->c_status = SNAPSHOT_WAITING;
      c->c_parent->f_pending++;
      int ret = zh_put_msg(reader, 0, sizeof(chunk_t *), (void *)&c);
      assertv(ret==sizeof(chunk_t *), "Message to READER has wrong size %d not %d\n", ret, sizeof(chunk_t *));
    }
    now = monotonic_ns_clock();
  }
  return now;                   /* Current end-of-loop time */
}

/*
 * Deal with a queue message from the READER thread.  These messages
 * are chunk pointers and fall into two disjoint classes.  In either
 * case any chunk received here has been detached from the READER’s
 * chunk queue and its frame has been released.
 *
 * - a chunk in SNAPSHOT_WRITTEN state:
 *   release the write commitment.
 *
 * If this was the last chunk of a snapfile, then run completed_snapfile.
 *
 * - a chunk in SNAPSHOT_ERROR state:
 *   abort the snapfile.
 *
 * In this case the READER will have released all chunks in its queue
 * and marked them in SNAPSHOT_ERROR state so that the abort_snapfile
 * routine can clean them up.  Chunks in transit between WRITER and
 * READER will still be in SNAPSHOT_WAITING state and are tidied by
 * abort_snapfile which runs for the first erroneous chunk.  The
 * snapfile structure is tidied by completed_snapfile which runs when
 * the last pending chunk is returned.
 */

private int process_reader_message(void *s) {
  chunk_t    *c;
  int         ret;
  snapfile_t *f;

  /* We are expecting a chunk pointer message */
  ret = zh_get_msg(s, 0, sizeof(chunk_t *), (void *)&c);
  assertv(ret == sizeof(chunk_t *), "Queue message size wrong %d vs %d\n", ret, sizeof(chunk_t *));
  assertv(c != NULL, "Queue message from READER was NULL pointer\n");

  f= c->c_parent;
  f->f_pending--;

  if(c->c_status == SNAPSHOT_WRITTEN) {
    f->f_written++;
    wp_totxfrsamples += c->c_samples;
    if(f->f_written == f->f_nchunks) {
      f->f_status = SNAPSHOT_WRITTEN;
      completed_snapfile(f);    /* This file is finished -- all chunks were written */
    }
    return true;
  }
  if(c->c_status == SNAPSHOT_ERROR) {
    if(f->f_status != SNAPSHOT_ERROR)
      abort_snapfile(f);        /* Tidy the chunk list, marking all into SNAPSHOT_ERROR state */
    if(f->f_pending == 0)
      completed_snapfile(f);    /* This file is finished -- no pending chunks in transit */
    return true;
  }

  assertv(false, "Chunk c:%04hx received in unexpected state %s\n", c->c_name, snapshot_status(c->c_status));
}

/* ================================ Process Command Messages ================================ */

private int process_writer_command(void *s) {
  int     used;
  int     ret;
  char   *p;
  strbuf  cmd;
  char   *cmd_buf;
  strbuf  err;

  used = zh_get_msg(s, 0, sizeof(strbuf), &cmd);
  if( !used ) {                 /* Quit */
    return false;
  }

  cmd_buf = strbuf_string(cmd);
  err = strbuf_next(cmd);

  switch(cmd_buf[0]) {
  case ’d’:                     /* Dir command */
  case ’D’:
    /* Call the command handler for Dir */
    strbuf_printf(err, "NO: Dir −− ");
    ret = process_dir_command(cmd);
    if(ret == 0) {
      strbuf_printf(err, "OK Dir");
      strbuf_clear(cmd);
    }
    break;

  case ’z’:
  case ’Z’:
    strbuf_printf(err, "NO: Ztatus −− ");
    ret = process_status_command(cmd);
    if(ret == 0) {
      strbuf_printf(err, "OK Ztatus:");
    }
    break;

  case ’s’:                     /* Snap command */
  case ’S’:
    /* Try to build a snapshot descriptor */
    strbuf_printf(err, "NO: Snap −− ");
    snap_t *s = build_snapshot_descriptor(cmd);
    if(s != NULL) {                 /* Snapshot building succeeded */
      queue_ins_after(&snapQ, snap2qp(s));
      strbuf_printf(err, "OK Snap %04hx ", s−>s_name);
      s−>s_error = (strbuf)de_queue((queue *)cmd);
      strbuf_clear(cmd);
      if(verbose > 0)
        debug_snapshot_descriptor(s);
      refresh_snapshot(s);
    }
    else {
      ret = −1;
    }
    break;

  default:
    strbuf_printf(err, "NO: WRITER −− unexpected writer command");
    ret = −1;
    break;
  }

  if(ret < 0) {
    strbuf_revert(cmd);
    zh_put_multi(log, 4, strbuf_string(err), "\n > ’", &cmd_buf[0], "’"); /* Error occurred, log the problem */
    strbuf_clear(cmd);
  }
  zh_put_msg(s, 0, sizeof(strbuf), (void *)&err);
  return true;
}

/*
 * WRITER thread message loop
 */

private void writer_thread_msg_loop() {    /* Read and process messages */
  int borrowedtime;
  int ret;
  int running;
  int n;

  zmq_pollitem_t  poll_list[] =
    {  { reader, 0, ZMQ_POLLIN, 0 },
       { command, 0, ZMQ_POLLIN, 0 },
    };
#define N_POLL_ITEMS    (sizeof(poll_list)/sizeof(zmq_pollitem_t))
  int (*poll_responders[N_POLL_ITEMS])(void *) =
    { process_reader_message,
      process_writer_command,
    };

  /* WRITER initialisation is complete */
  writer_parameters.w_running = !die_die_die_now;
  zh_put_multi(log, 1, "WRITER thread is initialised");

  running = writer_parameters.w_running;
  borrowedtime = 0;             /* Keeps track of the number of [ms] we owe */

  while( running && !die_die_die_now ) {
    int delay = borrowedtime + WRITER_POLL_DELAY; /* This is how long we wait normally in [ms] */

    int ret = zmq_poll(&poll_list[0], N_POLL_ITEMS, (delay<=0? −1 : delay));

    if( ret < 0 && errno == EINTR ) { /* Interrupted */
      zh_put_multi(log, 1, "WRITER loop interrupted");
      break;
    }
    if(ret < 0)
      break;

    if(delay >= 0)              /* We did some waiting, we owe no time */
      borrowedtime = 0;

    uint64_t tick = monotonic_ns_clock();

    for(n=0; n<N_POLL_ITEMS; n++) {
      if( poll_list[n].revents & ZMQ_POLLIN ) {
        if( (*poll_responders[n])(poll_list[n].socket) )
          running = true;
      }
    }

    uint64_t tock = writer_service_queue(tick);
    borrowedtime −= (tock−tick+500000)/1000000; /* Rounded elapsed time in [ms] */
  }
}

/* ================================ Thread Startup ================================ */

/*
 * WRITER thread main routine
 */

public void *writer_main(void *arg) {
  int ret;

  create_writer_comms();

  if( set_up_writer_capability < 0 ) {
    zh_put_multi(log, 1, "WRITER thread capabilities are deficient");
  }

  ret = set_writer_rt_scheduling();
  switch(ret) {
  case 1:
    zh_put_multi(log, 1, "WRITER RT scheduling succeeded");
    break;
  case 0:
    zh_put_multi(log, 1, "WRITER using normal scheduling: RTPRIO unset");
    break;
  default:
    zh_put_multi(log, 2, "WRITER RT scheduling setup failed: ", strerror(errno));
    break;
  }

  debug_writer_params();
  writer_thread_msg_loop();
  zh_put_multi(log, 1, "WRITER thread terminates by return");

  /* Clean up our ZeroMQ sockets */
  close_writer_comms();
  writer_parameters.w_running = false;
  return (void *)"normal exit";
}

/*
 * Verify the parameters for the WRITER and construct the WRITER state.
 *
 * Called by the MAIN thread during start up initialisation.
 */

public int verify_writer_params(wparams *wp, strbuf e) {
  import int tmpdir_dirfd;      /* Imported from snapshot.c */
  int ret;

  if( wp->w_schedprio != 0 ) {  /* Check for illegal value */
    int max, min;

    min = sched_get_priority_min(SCHED_FIFO);
    max = sched_get_priority_max(SCHED_FIFO);
    if(wp->w_schedprio < min || wp->w_schedprio > max) {
      strbuf_appendf(e, "RT scheduling priority %d not in kernel’s acceptable range [%d,%d]",
                     wp->w_schedprio, min, max);
      return −1;
    }
  }

  /*
   * Check that the requested mmap’d transfer RAM size and the
   * transfer chunk size are reasonable.
   */
  if(wp->w_lockedram < MIN_RAM_MB || wp->w_lockedram > MAX_RAM_MB) {
    strbuf_appendf(e, "Transfer Locked RAM parameter %d MiB outwith compiled−in range [%d, %d] MiB",
                   wp->w_lockedram, MIN_RAM_MB, MAX_RAM_MB);
    return −1;
  }
  if(wp->w_chunksize < MIN_CHUNK_SZ || wp->w_chunksize > MAX_CHUNK_SZ) {
    strbuf_appendf(e, "Transfer chunk size %d KiB outwith compiled−in range [%d, %d] KiB",
                   wp->w_chunksize, MIN_CHUNK_SZ, MAX_CHUNK_SZ);
    return −1;
  }

  /* Compute the number of frames available */
  const int pagesize = sysconf(_SC_PAGESIZE);
  int sz = wp->w_chunksize*1024;
  int nfr;
  sz = pagesize * ((sz + pagesize − 1) / pagesize); /* Round up to multiple of PAGE SIZE */
  wp->w_chunksize = sz / 1024;
  nfr = (wp->w_lockedram * 1024*1024) / sz;         /* Number of frames that fit in locked RAM */
  if(nfr < MIN_NFRAMES) {
    strbuf_appendf(e, "Adjusted chunk size %d KiB and given RAM %d MiB yield too few (%d < %d) frames",
                    wp->w_chunksize, wp->w_lockedram, nfr, MIN_NFRAMES);
    return −1;
  }
  wp_nframes = nfr;
  wp_chunksamples = wp->w_chunksize * 1024 / sizeof(sampl_t);

  /*
   * Check the writeahead fraction −− this is the proportion by which
   * the locked transfer RAM may be "overbooked".  Should be positive
   * and not too big :−)).
   */

  if(wp->w_writeahead < 0 || wp->w_writeahead > 1) {
    strbuf_appendf(e, "Transfer writeahead fraction %g out of compiled−in range [0,1]", wp->w_writeahead);
    return −1;
  }
  wp_totxfrsamples = nfr*wp->w_chunksize*1024*(1 + wp->w_writeahead) + pagesize−1;
  wp_totxfrsamples = pagesize * (wp_totxfrsamples / pagesize);
  wp_totxfrsamples = wp_totxfrsamples / sizeof(sampl_t);

  wp_nfiles = 0;                /* Currently no files in progress */

  /*
   * Check the snapdir directory exists and get a path fd for it.
   * Assumes we are already running as the non-privileged user.
   */
  wp_snap_dirfd = new_directory(tmpdir_dirfd, wp->w_snapdir);
  if( wp_snap_dirfd < 0 ) {     /* Give up on failure */
    strbuf_appendf(e, "Snapdir %s inaccessible: %m", wp->w_snapdir);
    return -1;
  }
  wp_snap_curfd = wp_snap_dirfd; /* Initial default */

  /*
   * Now try to get the memory for the transfer RAM...  This maps in a set of
   * anonymous pages so requires CAP_IPC_LOCK capability.
   */
  ret = init_frame_system(e, nfr, wp->w_lockedram, wp->w_chunksize);
  return ret;
}
```

```
#

#include "general.h"

/*
 * The ZMQ address for the writer thread
 */

#define WRITER_CMD_ADDR "inproc://Writer-CMD"

#define MIN_RAM_MB      16
#define MAX_RAM_MB      256
#define MIN_CHUNK_SZ    128
#define MAX_CHUNK_SZ    4096
#define MIN_NFRAMES     4

#define WRITER_MAX_CHUNK_DELAY          100 /* [ms] */
#define WRITER_MAX_CHUNKS_TRANSFER      8
#define WRITER_POLL_DELAY               50  /* [ms] */

typedef struct {

  /* These values come from environment and/or argument parameters */
  const char  *w_snapdir;
  int          w_schedprio;
  int          w_lockedram;
  int          w_chunksize;
  double       w_writeahead;
  /* Thread is running and ready -- set by main routine */
  int          w_running;
}
  wparams;

export int   verify_writer_params(wparams *, strbuf);
export void *writer_main(void *);

#define FILE_NAME_SIZE          32

#define SNAPSHOT_INIT           0 /* Structure just created */
#define SNAPSHOT_ERROR          1 /* Error found during checking or execution */
#define SNAPSHOT_PREPARE        2 /* Structure filled in, but files/chunks not done yet */
#define SNAPSHOT_READY          3 /* Snapshot etc. is ready, but waiting for READER queue space */
#define SNAPSHOT_WAITING        4 /* Snapshot etc. is ready, but waiting for data */
#define SNAPSHOT_WRITING        5 /* Snapshot file’s chunks are being written */
#define SNAPSHOT_WRITTEN        6 /* Snapshot’s chunk has been successfully written */
#define SNAPSHOT_COMPLETE       7 /* Snapshot written correctly (off queue) */
#define SNAPSHOT_DONE           8 /* Structure is finished with */

export const char *snapshot_status(int);
```