orkhestra: Control Program API Reference
Version 3

CML00084-03

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

October 31, 2022

Contents

1 Introduction

This document describes the interface between *orkhestra* and a control program. *orkhestra* starts a control program as a separate process and communicates to it using *pipes*. Before starting a control program *orkhestra* open two pairs of *pipe* file descriptors, one set for messages to the control program and the other for messages from the control program. The relevant file descriptors are passed to the control program via its command line parameters. This in addition to the user parameters passed.

A *orkhestra* control program is responsible for:

- At start-up open a connection to *orkhestra*, using the *pipe* file descriptors supplied to it via the command line. Inform *orkhestra* of a successful start-up or if failed with a abort message describing the failure.
- Interpreting and act upon `outputs` received from a *orkhestra State Machine*.
- Send relevant `inputs` to the *State Machine*.
- Monitor the API's file descriptors on his behalf (section 1.1.9 on page 12):
  - Ask the API to set his descriptors in the supplied (by the control program) sets of file descriptors
  - Do the `select()` call.
  - Call the API to check his descriptors for attention required..

The API library for interfacing to *orkhestra* is `liborkcpapi.a` with header file `orkcpapi.h`; See appendix A on page 14 for a listing of the header file.

See appendix C on page 24 for sample program that uses this API. It is a very basic control program that uses all the functionality of the API. This program is written to interface with a *State Machine* shown in appendix B on page 21

1.1 API Functions

Synopsis

```
typedef struct orkcpapi orkcpapi_t;
typedef struct
   {
   int  cp_num;
   char *name;
   char *description;
   }  orkcpapi_in_out_t;

orkcpapi_t *orkcpapi_open(int from_pipe,int to_pipe,
      orkcpapi_in_out_t *in_out_array,orkcpapi_sm_notification_t sm_notification,
      orkcpapi_msg_from_sm_t msg_from_sm,
```

```
      orkcpapi_output_from_sm_t output_from_sm);
typedef void (*orkcpapi_sm_notification_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,orkcpapi_notify_type_t type,char *msg);
typedef void (*orkcpapi_msg_from_sm_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,int sm_instance,int echo,
      orkcpapi_message_type_t type,char *msg);
typedef void (*orkcpapi_output_from_sm_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,int sm_instance,int echo,int output_number,
      char *output_name);
int orkcpapi_send_message(orkcpapi_t *orkcpapi,orkcpapi_message_type_t type,
      char *format, ...);
int orkcpapi_send_input(orkcpapi_t *orkcpapi,int sm_instance,int my_echo,
      int cp_input_num,struct timeval *time_stamp);
void orkcpapi_set_fds(orkcpapi_t *orkcpapi,int *fdmax,fd_set *readfds,
      fd_set *writefds);
void orkcpapi_check_fds(orkcpapi_t *orkcpapi,int *fdcnt,fd_set *readfds,
      fd_set *writefds);
char *orkcpapi_error(orkcpapi_t *orkcpapi);
```

`orkcpapi_t` describes the API instance data. This is created by `orkcpapi_open()` (section 1.1.2 on page 3) and serves as an anchor for API instance data. It must be passed on all subsequent calls to the API. Note the definition of this structure is hidden.

1.1.1 API error

Synopsis

```
char *orkcpapi_error(orkcpapi_t *orkcpapi);
```

Function `orkcpapi_error()` returns a message relating to the last error returned by one of the other functions of the API library. If the API instance is supplied as NULL, then the error message returned is for `orkcpapi_open()`.

1.1.2 API open

The `orkcpapi_open()` function initialise the API instance.

Synopsis

```
orkcpapi_t *orkcpapi_open(int from_pipe,int to_pipe,
      orkcpapi_in_out_t *in_out_array,orkcpapi_sm_notification_t sm_notification,
      orkcpapi_msg_from_sm_t msg_from_sm,
      orkcpapi_output_from_sm_t output_from_sm);
```

Parameters:

- `orkcpapi` is the API instance
- `from_pipe` is the *pipe* file descriptor, for receiving data from *orkhestra*; it is supplied to the control program on its command line interface: `--from-pipe=INT`.

- `to_pipe` is the *pipe* file descriptor, for sending data to *orkhestra*; it is supplied to the control program on its command line interface: `--to-pipe=INT`.
- `in_out_array` is an array describing all the inputs and outputs that the control program are dealing with:

```
typedef struct     {
   int  cp_num;
   char *name;
   char *description;
   }  orkcpapi_in_out_t;
```

  `cp_num` is the number that the control program assigned to the input/output named by `name` with `description` describing the entry. The `name` of the last entry must be `NULL`, to signal the end of the array.
- `sm_notification` is the user call back function for any for any status change with the connection to *orkhestra* or when a *State Machine* has started or has been stopped; see section 1.1.3 on page 5.
  This function must be supplied by the control program.
- `msg_from_sm` is the user call back function for textual messages from the *orkhestra State Machine*; see section 1.1.4 on page 6.
  This function must be supplied by the control program.
- `output_from_sm` is the user call back function for `outputs` from the *orkhestra State Machine*; see section 1.1.5 on page 7.
  This function must be supplied by the control program.

The `orkcpapi_open()` function initialise the API instance and connect to *orkhestra* using the supplied *pipe* file descriptors. The array `in_out_array` is used by the API for translating the control programs `input` numbers to what it is known by the *orkhestra State Machine*, and the reverse for the `output` numbers from the *State Machine*.

The API will use relevant user call back functions, supplied on the open, for status changes, messages and `outputs` from the *orkhestra State Machine*.

Return code
On success, the API instance (`orkcpapi_t *`) is returned. If an error occurs then `NULL` is returned and function `orkcpapi_error()` with a `NULL` parameter returns a formatted string detailing the error.

Example:

```
  /* Define all the input and outputs, between me and the state machine.
   */
typedef enum
   {
   CONNECT,
   DISCONNECT,
```

```
   GENERIC_REQUEST,
   GENERIC_RESPONSE,
   } in_output_numbers_t;

static orkcpapi_in_out_t my_in_outputs[] =
   {
   {CONNECT,"connect","Circuit connect"},
   {DISCONNECT,"disconnect","Circuit disconnect"},
   {GENERIC_REQUEST,"GENERIC_REQUEST","Send a canned message"},
   {GENERIC_RESPONSE,"GENERIC_RESPONSE","Received a response"},
   {0,NULL,NULL}
   };

  /* open the orkhestra API.
   */
   orkcpapi = orkcpapi_open(from_pipe,to_pipe,my_in_outputs,sm_notification,
         msg_from_sm,output_from_sm);
   if (!orkcpapi)
      {
     /* orkhestra API failed to initialise, Function orkcpapi_error() with a
      * NULL parameter returns an ASCII string describing the reason.
      */
      fprintf(stderr,"orksample: %s\n",orkcpapi_error(NULL));
      exit(12);
      }
```

1.1.3 Status Notification from *orkhestra*

`orkcpapi_sm_notification_t` is the proto type for notifying the control program of status changes of either the *orkhestra* connection or a *State Machine* status change. The user must supply this function and pass it as a parameter to `orkcpapi_open()`;see section 1.1.2 on page 3

Synopsis

```
typedef void (*orkcpapi_sm_notification_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,orkcpapi_notify_type_t type,char *msg);
```

Parameters:

- `orkcpapi` is the API instance
- `time_stamp` is the time that the event occurred.
- `type` is the notification type:
  - `ORKCPAPI_DISCONNECT` - The connection to *orkhestra* has been disconnected and `msg` gives the reason.
  - `ORKCPAPI_MCH_STARTED` The *State Machine* using this control program has started and `msg` is informational

  - ORKCPAPI_MCH_STOPPED The *State Machine* using this control program was stopped and msg is informational

- msg is a textual description of the event.

Example:

```
static void sm_notification(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      orkcpapi_notify_type_t type,char *msg)
   {
   if (verbose)
      orkcpapi_send_message(orkcpapi,ORKCPAPI_INFORMATION,"Received SM "
            "notification %d: %s\n",type,msg);
   switch (type)
      {
      case ORKCPAPI_DISCONNECT:
         printf("orkhestra killed me, by closing the pipe connection\n");
         shutdown_orksample();
         break;
      case ORKCPAPI_MCH_STARTED:
         sm_is_running = 1;
         break;
      case ORKCPAPI_MCH_STOPPED:
         sm_is_running = 0;
         break;
      }
   }  /* sm_notification */
```

1.1.4 Messages from *orkhestra*

orkcpapi_msg_from_sm_t is the proto type for textual messages to the control program from *orkhestra*. These messages are for information only and do require any action from the control program. They make a use full addition to reporting; if for instance the control program create session reports for later analysis. The user must supply this function and pass it as a parameter to orkcpapi_open();see section 1.1.2 on page 3.

Synopsis

```
typedef void (*orkcpapi_msg_from_sm_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,int sm_instance,int echo,
      orkcpapi_message_type_t type,char *msg);
```

Parameters:

- orkcpapi is the API instance
- time_stamp is the time that the event occurred.
- type categorise the message:
  - ORKCPAPI_INFORMATION - msg contains general information, i.e. state transitions.

  - ORKCPAPI_PROTO_ERR- msg contains details of a protocol violation that has happened for this instance.

- sm_instance is the *State Machine* instance, that the message is about.
- echo is set to what it was set to in the last input for this *State Machine* instance;see section 1.1.8 on page 11. If there was no previous input for this instance, it is set to zero.
- msg is the textual message.

Example:

```
static void msg_from_sm(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      int sm_instance,int echo,orkcpapi_message_type_t type,char *msg)
   {
   if (verbose)
      orkcpapi_send_message(orkcpapi,ORKCPAPI_INFORMATION,"Received SM "
            "message for instance %d, Msg %d - %s\n",sm_instance,type,msg);
   switch (type)
      {
      case ORKCPAPI_INFORMATION:
         break;
      case ORKCPAPI_PROTO_ERR:
        /* I don't care - orkhestra took care of it.
         */
         break;
      }
   } /* msg_from_sm */
```

1.1.5 Outputs from a *State Machine*

orkcpapi_output_from_sm_t is the proto type for outputs from the *State Machine* to the control program. The control program must act on it. The user must supply this function and pass it as a parameter to orkcpapi_open(); see section 1.1.2 on page 3.

Synopsis

```
typedef void (*orkcpapi_output_from_sm_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,int sm_instance,int echo,int output_number,
      char *output_name);
```

Parameters:

- orkcpapi is the API instance
- time_stamp is the time that the event occurred.
- sm_instance is the *State Machine* instance, that the message is about.

- `echo` is set to what it was set to in the last `input` for this *State Machine* instance;see section 1.1.8 on page 11. If there was no previous input for this instance, it is set to zero.
- `output_number` is the control programs output number for the named output `name`. The API converted the *State Machine* `output` number to the control programs `output` number, using the input/output array `my_in_outputs` that was supplied when the instance was opened; see section 1.1.2 on page 3. If there is no matching control program `output`, `output_number` will bet set to -1.
- `output_name` is the name of the output.

Example:

```
static void output_from_sm(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      int sm_instance,int echo,int output_number,char *output_name)
   {
   mch_inst_t *minst;
   unsigned int options;
   char buf[1000];

   if (verbose)
      orkcpapi_send_message(orkcpapi,ORKCPAPI_INFORMATION,"Received SM output"
            "for instance %d, Output %d - %s\n",sm_instance,output_number,
            output_name);

   if (output_number < 0)
      {
     /* Invalid output - I do not know anything about it.
      * Notify orkhestra of this, telling him I have failed; on receiving
      * this, orkhestra will disconnect me, so it is goodnight nurse.
      */
      orkcpapi_send_message(orkcpapi,ORKCPAPI_CP_FAILED,"Received an "
      "unknown output for instance %d: %s",sm_instance,output_name);
      return;
      }

  /* Get the SM instance data structure.
   * If there is not one yet, it will be created.
   */
   minst = get_sm_instance(sm_instance);

   if (output_number == CONNECT)
      {
      .
      .
      .
      return;
      }

   if (!minst->is_connected)
      {
```

```
     /* The instance is not connected - inform the state machine.
      */
      orkcpapi_send_input(orkcpapi,sm_instance,0,DISCONNECT,NULL);
      return;
      }

   switch(output_number)
      {
      case DISCONNECT:
         nc_close_circuit(minst->cir);
         break;
      case GENERIC_REQUEST:
         strcpy(buf,"Bla Bla");
         nc_send(minst->cir,strlen(buf),buf);
         break;

     /* This should not happen - it is a bug in this program:
      * I have an output in my_in_outputs array, and I did not
      * catered for it?
      */
      default:
         orkcpapi_send_message(orkcpapi,ORKCPAPI_CP_FAILED,"Received an "
               "Invalid output (not implemented) for instance %d: %s",
               sm_instance,output_name);
         break;
      }

   } /* output_from_sm */
```

1.1.6 Messages to *orkhestra*

Function `orkcpapi_send_message()` send a message to *orkhestra*.

Synopsis

```
int orkcpapi_send_message(orkcpapi_t *orkcpapi,orkcpapi_message_type_t type,
      char *format, ...);
```

Parameters:

- `orkcpapi` is the API instance
- `type` identifies the message type to be send:
  - `ORKCPAPI_CP_INITIALISED` - The control program has successfully initialised.
  - `ORKCPAPI_CP_FAILED` - The control program encountered a severe error and cannot continue; *orkhestra* on receiving this message will close the pipes to the control program.
    The control program must continue with his main service loop, monitoring

the file descriptors for the API, so that the message can propagate to *orkhestra*.

- `ORKCPAPI_INFORMATION` - Send a verbose message.

- `format, ...` is a format string and a variable number of arguments in the style of printf(). This is the text for this message.

Return code
On success, zero is returned. If an error occurs then -1 is returned and function `orkcpapi_error()` returns a formatted string detailing the error.

Example:

1.1.7 Cleanup of *State Machine* Instances

Function `orkcpapi_register_cleanup_inst()` registers a callback function to be called when *orkhestra* has deleted/destroyed/removed an instance associated with the registering control program.

For example when reducing the number of instances, *orkhestra* will remove instances and as it does it will notify the control program associated with each instance if that control program registered this call back.

The user data pointer supplied on registration will be passed back to the control program receiving the call back in the `void *user_data` parameter.

The call back routine will be called with the corresponding *State Machine* instance number, `int sm_instance`, so that any resources held on behalf of that instance can be finalised. Generally this is to close files or network connections or free allocated memory areas associated with the instance and no longer required.

Synopsis

```
typedef void (*orkcpapi_cleanup_inst_t)(void *user_data,int sm_instance);
void orkcpapi_register_cleanup_inst(orkcpapi_t *orkcpapi, void *user_data,
      orkcpapi_cleanup_inst_t cleanup);
```

Parameters:

- `orkcpapi` is the API instance
- `user_data` is the user data
- `sm_instance` is the *State Machine* instance that has been deleted by *orkhestra*.
- cleanup is the user registered cleanup callback.

1.1.8 Inputs to a *State Machine*

Function `orkcpapi_send_input()` send an `input` to the *orkhestra State Machine*.

Synopsis

```
int orkcpapi_send_input(orkcpapi_t *orkcpapi,int sm_instance,int my_echo,
      int cp_input_num,struct timeval *time_stamp);
```

Parameters:

- `orkcpapi` is the API instance
- `sm_instance` is the *State Machine* that this input is for.
- `my_echo`: All subsequent outputs and messages from the state machine. for this instance number, will echo this value.
- `cp_input_num` is my input number, the API will convert this to the input number that the *State Machine* uses.
- `time_stamp` is the time that this `input` occurred. If the supplied `time_stamp` is `NULL`, the API will set it to the current time, before sending it to the *State Machine*.

Return code
On success, zero is returned. If an error occurs then -1 is returned and function `orkcpapi_error()` returns a formatted string detailing the error.

Errors could be:

- The Control program did not notify *orkhestra*, that it has successfully initialised
- A State machine, using this control program has not been started.
- There is no corresponding `input` in the state machine for the requested `input` number.

Example:

```
if (orkcpapi_send_input(orkcpapi,sm_instance,0,DISCONNECT,NULL) < 0)
   {
  /* Nothing I can do - send an exception message to orkhestra explaining
   * the reason for the exception. On receiving the message, orkhestra
   * will close the connection to me.
   */
   orkcpapi_send_message(orkcpapi,ORKCPAPI_CP_FAILED,"Abort: %s",
         orkcpapi_error((orkcpapi));
   }
```

1.1.9 API file descriptors

The API do not directly expose his file descriptors to the control program. The control program must initialise and populate his requirements in the `readfds` and `writefds`, and set `fdmax` to the correct value. API function `orkcpapi_set_fds()` will add the API's requirements to the file descriptors sets and update `fdmax`. The user, after calling `select()` must call `orkcpapi_check_fds()` with the modified file descriptors. If any of API's file descriptors needs attention, it will be acted upon. Any action for the control program will be through the call back functions, that was supplied on the open of the API's instance.

Synopsis

```
void orkcpapi_set_fds(orkcpapi_t *orkcpapi,int *fdmax,fd_set *readfds,
      fd_set *writefds);
void orkcpapi_check_fds(orkcpapi_t *orkcpapi,int *fdcnt,fd_set *readfds,
      fd_set *writefds);
```

Parameters:

- `orkcpapi` is the API instance
- `fdmax` is the highest-numbered file descriptor in any of the two sets, plus 1.
- `readfds` is the read file descriptor set.
- `writefds` is the write file descriptor set.
- `fdcnt` is the number of file descriptors contained in the descriptor sets that requires attention.

Example:

```
/* Ask orkhestra API to set his file descriptors
 * in readfds and writefds that needs watching.
 */
 orkcpapi_set_fds(orkcpapi,&fdmax,&readfds,&writefds);

 fdcnt = select(fdmax+1,&readfds,&writefds,NULL,&tv);

/* Get orkhestra API to check if any of his file descriptors
 * needs attention and act on it.
 */
 if (fdcnt > 0)
    orkcpapi_check_fds(orkcpapi,&fdcnt,&readfds,&writefds);
```

1.1.10 *orkhestra* Central logging

The `orkcpapi_log_msg()` function logs a message to *orkhestra* central.

Synopsis

```
void orkcpapi_log_msg(char *topic,char *format, ...);
```

Parameters:

- `topic` specifies the message topic.
- `format` specifies the message format string as per the `C printf` function.
- `...` specifies the parameters to be substituted into the format string as per the `C printf` function.

Return code
No return code - void function.

The topic is centrally given a credit. If there is credit available, the message is logged to *orkhestra* central, otherwise it is ignored.

The default credit for a topic is 100. This can be changed by calling the *orkhestra* `set` command. This command can either change the credit for an individual topic or for all known/future topics. For the syntax, enter the command `help set`. To get a list of known topics enter the command `display` to get the syntax.

The API does not keep track of the actual credit left as *orkhestra* will broadcast to all the control programs when the credit for a topic has run out or can be resumed again.

The topic is limited to 255 and the message to 2048 bytes. This includes the terminating null byte.

Examples:

```
orkcpapi_log_msg("Response_1100","%s",rmsg);

orkcpapi_log_msg("ErrorDisconnect","%Instance %d s",inst,msg);
```

A Control Program API header file: orkcpapi.h

```
#ifndef ORKCPAPI_H
#define ORKCPAPI_H
  /* File orkcpapi.h
   *
   * This header file describes the orkgestra API.
   * It describes the data structures and functions exposed by the API.
   *
   * Author: Jan Vlok.
   *
   * Copyright (c) 2008 Code Magus Limited. All rights reserved.
   *
   */

  /* $Author: janvlok $
   * $Date: 2022/10/24 08:47:10 $
   * $Id: orkcpapi.h,v 1.4 2022/10/24 08:47:10 janvlok Exp $
   * $Name:  $
   * $Revision: 1.4 $
   * $State: Exp $
   *
   * $Log: orkcpapi.h,v $
   * Revision 1.4  2022/10/24 08:47:10  janvlok
   * Added attribute packed to clog structs
   *
   * Revision 1.3  2022/09/25 10:07:35  janvlok
   * Implement Central Logging
   *
   * Revision 1.2  2022/08/19 14:32:27  janvlok
   * notify CP's on deleted innstances, if registered
   *
   * Revision 1.1.1.1  2011/06/10 08:26:56  janvlok
   * Take on
   *
   */

static char *cvs_orkcpapi_h =
   "$Id: orkcpapi.h,v 1.4 2022/10/24 08:47:10 janvlok Exp $";

#include <time.h>
#include <sys/time.h>
#include <sys/select.h>
#include <stdint.h>

  /*
   * Data structures and types:
   */
typedef enum
   {
   ORKCPAPI_DISCONNECT,       /* connection to orkgestra has been disconnected */
   ORKCPAPI_MCH_STARTED,      /* state machine started */
```

```
   ORKCPAPI_MCH_STOPPED       /* state machine stopped */
   } orkcpapi_notify_type_t;

typedef enum
   {
   ORKCPAPI_INFORMATION,      /* informational */
   ORKCPAPI_PROTO_ERR,        /* protocol violation by the control program */
   ORKCPAPI_CP_FAILED,        /* control program have a fatal error */
   ORKCPAPI_CP_INITIALISED    /* control program have successfully initialised*/
   } orkcpapi_message_type_t;


  /* orkcpapi instance data structure.
   * This data structure is created by orkcpapi open() and serves as an anchor
   * for orkcpapi instance data. It must be passed on subsequent calls to the
   * orkcpapi functions.
   * Note the definition of this structure is hidden.
   */
typedef struct orkcpapi orkcpapi_t;

typedef struct orkcpapi_in_out orkcpapi_in_out_t;

  /*
   * Structure orkcpapi_in_out_t describes the inputs/outputs that the control
   * program will be processing. It is passed as an array to orkcpapi_open().
   */
struct orkcpapi_in_out
   {
   int  cp_num;      /* in/output number internal to the control program */
   char *name;       /* name of the in/output to/from the state machine */
   char *description;/* description of the in/output to/from the state machine */
   };

  /* Call back functions:
   */

  /* orkcpapi_sm_notification_t is the prototype for the user call back function
   * for any status change:
   *   . Connection to orkgestra
   *   . State machine starting and stopping.
   * Parameter *orkcpapi is the orkcpapi instance. Parameter type is the
   * notification type:
   *    == ORKCPAPI_DISCONNECT - The connection to orkgestra has been disconnected
   *                             and *msg gives the reason.
   *    == ORKCPAPI_MCH_STARTED - The state machine using this control program
   *                              has started and *msg is informational.
   *    == ORKCPAPI_MCH_STOPPED - The state machine using this control program
   *                              has stopped and *msg is informational.
   * *time_stamp is the time that this event happened.
   */
typedef void (*orkcpapi_sm_notification_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,orkcpapi_notify_type_t type,char *msg);
```

```
  /* orkcpapi_msg_from_sm_t is the prototype for the user call back function
   * for textural message received from orkgestra. This enables the
   * control program to use this message for something like a session log.
   * Most control programs ignore this message. sm_instance is the state
   * machine's instance that this message is for. The value of the echo
   * parameter is set to the echo in the last input received from the
   * control program for this state machine's instance, if there was no
   * previous input for this instance, it is set to zero.
   * Parameter *orkcpapi is the orkcpapi instance.
   * Parameter type is the message type and can have the following values:
   *    == ORKCPAPI_INFORMATION - *msg contains information, i.e. state
   *                              transitions.
   *    == ORKCPAPI_PROTO_ERR - *msg contains details of a protocol violation:
   *                            the control program send an output to the state
   *                            machine that it did not recognise.
   * *time_stamp is the time that this message was sent by the state machine.
   */
typedef void (*orkcpapi_msg_from_sm_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,int sm_instance,int echo,
      orkcpapi_message_type_t type,char *msg);

  /* orkcpapi_output_from_sm_t is the prototype for the user call back function
   * for outputs received from orkgestra.
   * Parameter *orkcpapi is the orkcpapi instance. sm_instance is the state
   * machine's instance that this output is for. The value of the echo
   * parameter is set to the echo in the last input received from the
   * control program for this state machine's instance, if there was no
   * previous input for this instance, it is set to zero.
   * output_number is the control program's output number for the named output
   * in *output_name. The API converted the state machine's output number to
   * what the control program expects, using the in/output array that was
   * supplied with the orkcpapi_open() function. If there is no matching output,
   * output_number will bet set to -1.
   * *time_stamp is the time that this output was sent by the state machine.
   */
typedef void (*orkcpapi_output_from_sm_t)(orkcpapi_t *orkcpapi,
      struct timeval *time_stamp,int sm_instance,int echo,int output_number,
      char *output_name);

  /* orkcpapi_open() create and initialise a orkcpapi instance.
   *
   * from_pipe and to_pipe are supplied to the control program at start-up via
   * its command line interface:
   *    --to-pipe=INT        Output pipe descriptor fd
   *    --from-pipe=INT      Input pipe descriptor fd
   * *in_out_array is the data structure, describing the the inputs and outputs
   * to/from the state machine it is able to process. The last entry's name
   * must be NULL to signify the end of the array.
   * Call back functions:
   *    sm_notification() - connection to state machine status change.
   *    msg_from_sm() - informational (textural)  messages to orkgestra.
```

```
   *    output_from_sm() - outputs from the state machine.
   *
   * On success, the API instance (orkcpapi\_t *) is returned.If an error occurs
   * then NULL is returned and function orkcpapi_error() with a NULL parameter
   * returns a formatted string detailing the error.
   * The API instanceIt must be passed on all subsequent calls to the API.
   *
   */
orkcpapi_t *orkcpapi_open(int from_pipe,int to_pipe,
      orkcpapi_in_out_t *in_out_array,orkcpapi_sm_notification_t sm_notification,
      orkcpapi_msg_from_sm_t msg_from_sm,
      orkcpapi_output_from_sm_t output_from_sm);

  /* Function orkcpapi_error() returns a message relating to the last error
   * returned by one of the other functions of the API library. If the API
   * instance is supplied as NULL, then the error message returned is for
   * orkcpapi_open()}.
   */
char *orkcpapi_error(orkcpapi_t *orkcpapi);

  /* orkcpapi_send_message() Send a textural message to orkgestra.
   * Parameter *orkcpapi is the orkcpapi instance.
   * Permissible values for type:
   *   ORKCPAPI_CP_INITIALISED - control program has successfully initialised.
   *   ORKCPAPI_CP_FAILED -      the control program have a severe error and
   *                            cannot continue - orkgestra, on receiving this
   *                            will close the pipes to the control program.
   *   ORKCPAPI_INFORMATION -    information.
   * *format is the associated message using printf style parameters.
   * Returns zero on success, else -1 on failure.
   * Call orkcpapi_error() for a description of the error.
   */
int orkcpapi_send_message(orkcpapi_t *orkcpapi,orkcpapi_message_type_t type,
      char *format,...)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf,3,4)))
#endif
;

  /* orkcpapi_send_input() Send an input to the state machine.
   * Parameter *orkcpapi is the orkcpapi instance.
   * Parameter *orkcpapi is the orkcpapi instance. sm_instance is the state
   * machine's instance that this input is for. All subsequent outputs
   * and messages from the state machine for this instance will echo
   * the value supplied in echo.
   * If *time_stamp is NULL, a time stamp,using the current time, will
   * be use for sending this input to the state machine.
   * Returns zero on success, else -1 on failure.
   * Call orkcpapi_error() for a description of the error. Possible errors:
   *    . Control program did not notify that it successfully initialised.
   *    . State machine not running
   *    . There is no corresponding input in the state machine for the
```

```
   *      requested input number.
   */
int orkcpapi_send_input(orkcpapi_t *orkcpapi,int sm_instance,int my_echo,
      int cp_input_num,struct timeval *time_stamp);

  /* Watching the file descriptors that need attention is the responsibility
   * of the control program, i.e. It needs to do the select(). The sequence of
   * events:
   * 1) Initialise the read and write fd sets.
   * 2) Call orkcpapi_set_fds()
   * 3) Do the select()
   * 4) Call orkcpapi_check_fds()
   *    orkcpapi will call the relevant call back function, if required.
   *
   * orkcpapi_set_fds() set read and write fds for pipes that needs attention.
   * orkcpapi_check_fds() check read and write fds after the select().
   */
void orkcpapi_set_fds(orkcpapi_t *orkcpapi,int *fdmax,fd_set *readfds,
      fd_set *writefds);
void orkcpapi_check_fds(orkcpapi_t *orkcpapi,int *fdcnt,fd_set *readfds,
      fd_set *writefds);


  /* orkcpapi_register_cleanup_inst() registers a callback function for when an
   * instance needs to be cleaned up -- used when the instance no longer exists
   * from orkhestra's viewpoint - for example when reducing the number of
   * instances.
   *
   * Orkhestra will inform the connected CPs - the correct one owning an
   * instance - that the instance has been deleted/destroyed/terminated and that
   * if a callback has been registered by the CP for the cleanup callback,
   * then the cleanup callback must be called.
   *
   * The register will include a void pointer to user data, but this is
   * global user data, which will be passed back to the registered cleanup module.
   *
   * The cleanup module will be called with the corresponding sm_instance
   * number so that any resources held on behalf of that instance can be cleaned
   * up/freed/etc.
   */

typedef void (*orkcpapi_cleanup_inst_t)(void *user_data,int sm_instance);
void orkcpapi_register_cleanup_inst(orkcpapi_t *orkcpapi, void *user_data,
      orkcpapi_cleanup_inst_t cleanup);

  /* Central logging interface to orkhestra.
   * All the messages are preceeded by clog_hdr_t, a integer,
   * defining the message type
   */

typedef struct clog_hdr clog_hdr_t;
typedef struct clog_credit clog_credit_t;
```

```
typedef struct clog_msg clog_msg_t;
typedef struct clog_origin clog_origin_t;
struct clog_hdr
   {
   enum
      {
      CLOG_CREDIT,                /* From central to update a topic's credit */
      CLOG_MSG,                   /* From a CP to log a message */
      CLOG_ORIGINATOR,            /* CP info host and CP names */
      } type;                     /* message type */
   } __attribute__ ((packed));

  /* The topic follows the clog_credit_t data structure.
   */

struct clog_credit
   {
   clog_hdr_t hdr;                /* CLOG_ORIGINATOR */
   int no_credit;                 /* No credit left */
   int topic_length;              /* topic */
   } __attribute__ ((packed));

  /* The topic and message follows the clog_msg_t data structure.
   */

struct clog_msg
   {
   clog_hdr_t hdr;                /* CLOG_ORIGINATOR */
   int topic_length;              /* topic */
   int text_length;               /* message */
   } __attribute__ ((packed));

  /* The circuit host name and CP name follows the clog_origin_t data structure.
   */

struct clog_origin
   {
   clog_hdr_t hdr;                /* CLOG_ORIGINATOR */
   uint32_t pid;                  /* PID */
   int host_length;               /* originator host */
   int name_length;               /* originator name */
   } __attribute__ ((packed));

  /* orkcpapi_log_msg() Logs a message to orkhestral central.
   * The topic is centrally given a credit. If there is some credit, then the
   * message is logged to orkhestra central, else ignored.
   *
   * The default credit for a topic is 100. This can be changed by the orkhestra
   * 'set' command. This command can either change the credit for an individual
   * topic or for all known/future topics. For the syntax, enter command
   * 'help set'. To get a list of known topics enter the command 'display'
   * to get the syntax.
```

```
   *
   * The API will not forward the message to orkhestra central if there is no
   * credit left. The API do not keep track of the actual credit left as
   * orkhestra will broadcast to all the Control programs when the credit for
   * a topic has run out or can be resumed again.
   *
   * Please note:
   *   The topic is limitted to 255 bytes and the message to 2048. This
   *   includs the terminating null byte ('\0').
   *
   * Parameters:
   *    *topic is the message topic.
   *    *format is the message using printf style parameters.
   */

void orkcpapi_log_msg(char *topic,char *format, ...)
#if defined(__linux__) || defined(__CYGWIN__)
__attribute__ ((format (printf,2,3)))
#endif
;

#endif /* ORKCPAPI_H */
```

B Sample State machine: orksample.mch

```
{
This is a very simple machine:

At start-up the first transition for a state machine instance is to
the idle state, and after it transacts, it is back to the idle state.
Before entering the idle state an idle timer will be set, for the
duration a instance (device) spends in the idle state. The value
definition 'think_time' is used for the idle time.

Once the timer expires:
Output 'connect'  for a connection request to the control program and
wait for input as to the outcome. If the input 'disconnect', set the
idle timer and back to the idle state.

Input 'connect' tells us that the instance is connected, and just to
demonstrate the use of the choose() function, for some instances a canned
message (GENERIC_REQUEST) are requested to be send, an for others a immediate
'disconnect' are outputted, which eventually leads back to the idle state.
The relevant weights for this are defined by the weight distribution
'what_to_do'.

When outputting the 'GENERIC_REQUEST', a timeout timer is set, using the value
definition 'timeout_value'.
Waiting for the response, we need to cater for:
   . The time out timer that was set expired - output 'disconnect' and via
     waiting for the 'disconnect' input, back to the idle state.
   . Input 'disconnect' - set the idle time and back to idle state.
   . Input 'GENERIC_RESPONSE', output 'disconnect' and via waiting for the
     'disconnect' input, back to the idle state.
}
machine orksample();

   created by ("Jan Vlok");
   description("orkhestra sample control program.");
   date("2008-01-16T10:51:18");
   target("Demonstration");
   control_program (orksample);
   modified by("Jan Vlok");


value timeout_value
   (
   title("Response time out")
   description("Time in milliseconds to wait for a response message")
   constant(50000)
   );

value think_time
   (
   title("Device Idle time")
```

```
   description("Time in milliseconds an instance will be in the idle state")
   distribution(class(exponential) min(10000) max(3000000) mu(100000))
   );

weight distribution what_to_do
   (
   title("Disconnect or continue")
   description("Some instances we want to disconnect, before sending a message")
   weights(disconnect(1) continue(2))
   );


begin

-- This wild card state [*] is the default for a disconnect in any state,
-- it will only, if a state do not have a transition for disconnect.
-- If we get a disconnect in any state, there is not much we can or want to
-- do about it.
[*]
   disconnect:
   start_timer(device_ready, think_time);
[device_idle]
;
-- The first state defined will be by default initial state, so at
-- start up of an instance, this is the initial state, and the default.
-- input is startup.
[startup]
   startup:
   start_timer(device_ready,think_time);
[device_idle]
;

-- Instance is ready for the next message, once the timer has expired.
-- Get it connected.
--
[device_idle]
   timer_expire(device_ready):
   connect;
[wait_connection]
;

-- Waiting for for the termination of a connection, a disconnect
-- was requested.
[wait_disconnect]
   disconnect:
   start_timer(device_ready,think_time);
[device_idle]
;

-- Just to demonstrate the choose() action:
-- Some connection we are going to terminate, without sending
-- a message, using the what_to_do weight distribution,
```

```
-- that is to say, once we have a connection established.
[wait_connection]
   connect:
   choose(what_to_do);
[choice_what_to_do]
;

-- OK disconnect
[choice_what_to_do]
   choice(disconnect):
   disconnect;
[wait_disconnect]
;

-- Continue to send a message
[choice_what_to_do]
   choice(continue):
   GENERIC_REQUEST;
   start_timer(msg_timed_out,timeout_value);
[wait_response]
;

-- Bummer - Bad parameters give to the control program?
--
[wait_connection]
   connect_error:
[final]
;

-- Wait for the response
[wait_response]
   GENERIC_RESPONSE:
   cancel_timer(msg_timed_out);
   disconnect;
[wait_disconnect]
;

[wait_response]
   timer_expire(msg_timed_out):
   disconnect;
[wait_disconnect]
;

[wait_response]
   disconnect:
   cancel_timer(msg_timed_out);
   start_timer(device_ready,think_time);
[device_idle]
;

end
```

C Sample Control program: orksample.c

```
/* File: orksample.c
 * Sample Orkhestra control program.
 *
 * This program process outputs for a sample state machine run by orkhestra,
 * and respond with the relevant inputs.
 * The source for this state machine is orksample.mch.
 * The state machine and scripts to run the sample is in the sub directory
 * ./scripts.
 *
 * Outputs from the state machine:
 *  . connect
 *      Connect the SM instance using a TCP/IP Internet address from a
 *      address pool. These addresses are read in from a file, which is
 *      passed via the command line option "--host-addr".
 *   . disconnect
 *      Disconnect circuit for the SM instance.
 *   . GENERIC_REQUEST
 *      Send a canned message on the established circuit for the SM instance.
 *
 * Inputs to the state machine:
 *    connect
 *       . A circuit connected for a previous 'connect' request from the SM.
 *    disconnect
 *       . A circuit failed to connect for a previous 'connect' request from
 *         the SM.
 *       . A circuit is not connected when a 'GENERIC_REQUEST' is received
 *         from the SM.
 *       . A circuit has disconnected for a previous 'disconnect' request
 *         from the SM.
 *   . GENERIC_RESPONSE
 *       . Received a message for a SM instance.
 *
 * Note: This program operates in a non-blocking mode and are able to
 *       handle an infinite number of state machine instances, barring
 *       memory constraints. It needs to keep track of the status of the
 *       SM instances it has seen.
 *
 * Author: Jan Vlok.
 *
 * Copyright (c) 2008 Code Magus Limited. All rights reserved.
 *
 */

/*
 * $Author: hayward $
 * $Date: 2012/07/24 09:52:05 $
 * $Id: orksample.c,v 1.3 2012/07/24 09:52:05 hayward Exp $
 * $Name:  $
 * $Revision: 1.3 $
 * $State: Exp $
```

```
   *
   * $Log: orksample.c,v $
   * Revision 1.3  2012/07/24 09:52:05  hayward
   * Add Large File Support (LFS)
   *
   * Revision 1.2  2011/06/13 15:09:23  janvlok
   * Tested
   *
   * Revision 1.1.1.1  2011/06/10 08:29:16  janvlok
   * Take on
   *
   */

static char *cvs_orksample_c =
            "$Id: orksample.c,v 1.3 2012/07/24 09:52:05 hayward Exp $";

  /*
   * Large File Support Required from various environments:
   */

#define _LARGEFILE_SOURCE
#define _LARGEFILE64_SOURCE
#define _LARGE_FILES
#define _FILE_OFFSET_BITS 64

#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <popt.h>
#include <signal.h>
#include <assert.h>

#include <orkcpapi.h>
#include <nc.h>

#include "version.h"
#define MAIN

  /*
   * Data structures and types:
   */

  /* Define all the input and outputs, between me and the state machine.
   * The array 'my_in-outputs' is passed on opening the orkhestra my_in_outputs.
   */
typedef enum
   {
   CONNECT,
   DISCONNECT,
```

```
   GENERIC_REQUEST,
   GENERIC_RESPONSE,
   } in_output_numbers_t;

static orkcpapi_in_out_t my_in_outputs[] =
   {
   {CONNECT,"connect","Circuit connect"},
   {DISCONNECT,"disconnect","Circuit disconnect"},
   {GENERIC_REQUEST,"GENERIC_REQUEST","Send a canned message"},
   {GENERIC_RESPONSE,"GENERIC_RESPONSE","Received a response"},
   {0,NULL,NULL}
   };

  /* During initialisation the error message did not seem to return correctly to
   * orkhestra (or was not printed properly. So this first prints it to stderr
   * before sending back to orkhestra. This requires variadic macros.
   */

#define ORKCPAPI_SEND_MESSAGE(orkcpapi,type,format, ...)\
   do \
   { \
      fprintf(stderr,format,__VA_ARGS__);\
      orkcpapi_send_message(orkcpapi,type,format,__VA_ARGS__);\
   } while(0)

  /*
   * We need to keep track of the status of a state machine instance,
   * for example:
   * The SM outputs a 'connect' for SM instance 123. I immediately start the
   * connection process, however this might take a while and I will be notified
   * via a event of the outcome. On receiving this event, I will send an input
   * to the SM for instance 123, which could be either a 'connect' or
   * 'disconnect', depending whether the circuit was establish or not. Some
   * time in future, the SM might request sending of the a message
   * (GENERIC_REQUEST) for this instance and I will need the NC circuit
   * for this, and so on.
   *
   * Typedef mch_inst_t defines a state machine instance data structure
   * and **mch_inst is our instance array.
   *
   * Function get_sm_instance() returns the data structure for an instance.
   * If it is the first time a particular instance is seen, a new structure
   * will be created for this instance.
   */
typedef struct mch_inst mch_inst_t;
struct mch_inst
   {
   int  sm_instance;       /* instance number, also the index into the array */
   int is_connected;       /* Instance circuit is connected */
   nc_circuit_t *cir;      /* NC connection circuit */
   };
```

```
static int  mch_inst_asz = 0;           /* SM instance array size */
static mch_inst_t **mch_inst = NULL;    /* SM instance array */
static mch_inst_t *get_sm_instance(int sm_instance);


  /*
   * Statics.
   */
static int sm_is_running=0;           /* Te state machine up and running */
static int aborting;                  /* Have notify control process, that I am
                                         aborting - just wait for closure of the
                                         pipes */
static orkcpapi_t *orkcpapi;            /* orkhestra state machine API instance */
static nc_instance_t *nc;             /* NC instance */
static int host_addr_cnt;             /* number of addresses loaded */
static int next_host_addr;            /* index to the next address to use */
static struct sockaddr_in *host_saddr;/* host address array */
static unsigned int num_messages=0;   /* number of messages handled */

  /* NC framing verification values.
   */
static struct
   {
   char *name;
   int value;
   } f_format [] =
   {
   "F1234", NC_FORMAT_1234,         /* long network byte order */
   "F4321", NC_FORMAT_4321,         /* long Intel byte order */
   "F12",   NC_FORMAT_12,           /* short network byte order */
   "F21",   NC_FORMAT_21,           /* short Intel byte order */
   "F1",    NC_FORMAT_1,            /* byte */
   "F1200", NC_FORMAT_1200,         /* byte */
   "FLF",   NC_FORMAT_LF,           /* messages delimited by '\n' */
   "FCRLF", NC_FORMAT_CRLF,         /* messages delimited by '\r', optionally
                                       followed by '\n' */
   "FNONE", NC_FORMAT_NONE,         /* No framing */
   NULL,    0,
   };

  /*
   * Statics for command line parsing.
   */
static int debug = 0;             /* Debug - fgetc(stdin) in init(),so that one
                                     can get in with 'gdb' */
static int verbose=0;             /* Verbose */
static int to_pipe=0;             /* Output pipe to control process */
static int from_pipe=0;           /* Input pipe from control process */
static int copy_num=0;            /* control program copy */
static char *host_addr_file=NULL; /* Host addresses input file */
static char *tcp_trace_fname=NULL;/* TCP/IP trace file name */
static int nc_bias=0;             /* NC framing bias */
```

```
static int nc_frame=0;            /* NC framing format */
static int max_messages=0;        /* Maximum messages to handle */

  /*
   * Command line parsing popt data structure.
   */
static char *frame_format = NULL; /* circuit framing format */
struct poptOption optionsTable[] =
   {
     { "verbose",'v', POPT_ARG_NONE,&verbose,0,
       "verbose",NULL},
     { "to-pipe", 0,POPT_ARG_INT,&to_pipe,0,
       "Output pipe descriptor fd","?" },
     { "from-pipe", 0,POPT_ARG_INT,&from_pipe,0,
       "Input pipe descriptor fd","?" },
     { "host-addr-file", 'h',POPT_ARG_STRING,&host_addr_file,0,
       "Output pipe descriptor fd","?" },
     { "copy-num", 0,POPT_ARG_INT,&copy_num,0,
       "Control program copy number","<integer>" },
     { "host-addr-file", 'h',POPT_ARG_STRING,&host_addr_file,0,
       "Host addresses input file",NULL },
     { "tcp-trace-name",0,POPT_ARG_STRING,&tcp_trace_fname,0,
       "TCP/IP trace file name: %D = ccyymm %T = hhmmss",NULL},
     { "nc-format",0,POPT_ARG_STRING,  &frame_format,0,
       "NC framing format: F1234|F4321|F12|F21|F1|F1200|FLF|FCRLF","FORMAT"},
     { "nc-bias",0,POPT_ARG_INT,  &nc_bias,0,
       "NC framing bias","unsigned int"},
     { "max-messages",0,POPT_ARG_INT,  &max_messages,0,
       "Shutdown after this many messages","{0|unsigned int}"},
     { "debug",0,POPT_ARG_NONE,  &debug,0,
       "For gdb",NULL},
       POPT_AUTOHELP
     { NULL,0,0,NULL,0 }
   };

  /*
   * Function Proto types
   */
static int init(int argc,char *argv[],char *envp[]);
static void signal_handler(int signum);
static void shutdown_orksample(void);
static int nc_data_received(nc_circuit_t *circuit,int len,char *buf,
      struct timeval *time_stamp);
static void nc_status_change(nc_circuit_t *circuit,int status,
      char *msg);
static void sm_notification(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      orkcpapi_notify_type_t type,char *msg);
static void msg_from_sm(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      int sm_instance,int echo,orkcpapi_message_type_t type,char *msg);
static void output_from_sm(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      int sm_instance,int echo,int output_number,char *output_name);
```

```
  /* Function main()
   * Main processing loop.
   */
int main(int argc,char *argv[],char *envp[])
   {
   fd_set readfds,writefds;  /* Read & write descriptor set */
   int fdmax,fdcnt;
   struct timeval tv;

   if (init(argc,argv,envp) < 0)
      {
     /* My initialisation failed, and the reason for it has been
      * dispatched in a abort message to orkhestra. Once he has received
      * it and done the necessary logging, he will close the connection to me,
      * signalling me to terminate.
      * I must enter the main service loop so that the abort message can
      * propagate to orkhestra.
      */
      aborting = 1;
      printf("orksample: ABORTING - waiting for orkhestra to close the "
            "connection\n");
      }
   else
     /* Inform orkhestra that we have successfully started and is up and running.
      */
      ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_INITIALISED,"Version: %s",
            cvs_orksample_c);

  /* For ever loop.
   */
   while (1)
      {
      FD_ZERO(&readfds);
      FD_ZERO(&writefds);
      fdmax = 0;

     /* Ask orkhestra API to set his file descriptors
      * in readfds and writefds that needs watching.
      */
      orkcpapi_set_fds(orkcpapi,&fdmax,&readfds,&writefds);

      if (nc)
         nc_set_fds(nc,&fdmax,&readfds,&writefds);

      tv.tv_sec = 1;
      tv.tv_usec = 0;
      fdcnt = select(fdmax+1,&readfds,&writefds,NULL,&tv);

      if (fdcnt > 0 && nc)
         nc_check_fds(nc,&fdcnt,&readfds,&writefds);

     /* Get orkhestra API to check if any of his file descriptors
```

```
      * needs attention and act on it.
      */
      if (fdcnt > 0)
         orkcpapi_check_fds(orkcpapi,&fdcnt,&readfds,&writefds);
      }

   return 0;
   } /* Main */

  /* Function sm_notification()
   * Callback function for when there is a status change with the state
   * machine or when orkhestra closed our connection with him.
   * *orkcpapi is the orkhestra API instance, *time_stamp is the time
   * that this event occurred, type is the event type and *msg describes the
   * event.
   *
   * If we have been disconnected, the assumption is that orkhestra has closed
   * the connection: he wants us to terminate.
   */
static void sm_notification(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      orkcpapi_notify_type_t type,char *msg)
   {
   if (verbose)
      ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_INFORMATION,"Received SM "
            "notification %d: %s\n",type,msg);
   switch (type)
      {
      case ORKCPAPI_DISCONNECT:
         printf("orkhestra killed me, by closing the pipe connection\n");
         shutdown_orksample();
         break;
      case ORKCPAPI_MCH_STARTED:
         sm_is_running = 1;
         break;
      case ORKCPAPI_MCH_STOPPED:
         sm_is_running = 0;
         break;
      }
   }  /* sm_notification */

  /* Function msg_from_sm()
   * This is the call back function for when a textural message was received
   * from orkhestra.
   * In this program I do not really care about it, I do not create sessions
   * reports, were it would be a use full thing to write to.
   * *orkcpapi is the API instance, type categorise the message and
   * *msg is the message; it is an NULL terminated ASCII string.
   * *time_stamp is the time that this message was sent by orkhestra.
   * sm_instance identifies the SM instances that this message is applicable
   * to and echo is what was send on the previous input to the state machine,
   * using the echo parameter - I do not use it, so it is zero.
   */
```

```
static void msg_from_sm(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      int sm_instance,int echo,orkcpapi_message_type_t type,char *msg)
   {
   if (verbose)
      ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_INFORMATION,"Received SM "
            "message for instance %d, Msg %d - %s\n",sm_instance,type,msg);
   switch (type)
      {
      case ORKCPAPI_INFORMATION:
         break;
      case ORKCPAPI_PROTO_ERR:
        /* I don't care - orkhestra took care of it.
         */
         break;
      }
   } /* msg_from_sm */

  /* Function orkcpapi_output_from_sm()
   * This is the call back function for outputs received from the orkhestra
   * state machine.
   * *orkcpapi is the API instance. sm_instance is the state
   * machine's instance that this output is for. The value of the echo
   * parameter is set to the echo in the last input received from the
   * me, for this state machine's instance, - I do not use it, so it is zero.
   * output_number is the my output number for the named output as per
   * *output_name. The API converted the state machine's output number to
   * what I expect it to be, using the in/output array my_in_outputs that was
   * supplied with the orkcpapi_open() function. If there is no matching
   * output, output_number will bet set to -1.
   * *time_stamp is the time that this output was dispatched by the state
   * machine.
   */
static void output_from_sm(orkcpapi_t *orkcpapi,struct timeval *time_stamp,
      int sm_instance,int echo,int output_number,char *output_name)
   {
   mch_inst_t *minst;
   unsigned int options;
   char buf[1000];

   if (verbose)
      ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_INFORMATION,"Received SM output"
            "for instance %d, Output %d - %s\n",sm_instance,output_number,
            output_name);

   if (output_number < 0)
      {
     /* Invalid output - I do not know anything about it.
      * Notify orkhestra of this, telling him I have failed; on receiving
      * this, orkhestra will disconnect me, so it is goodnight nurse.
      */
      ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"Received an "
      "unknown output for instance %d: %s",sm_instance,output_name);
```

```
      return;
      }

  /* Get the SM instance data structure.
   * If there is not one yet, it will be created.
   */
   minst = get_sm_instance(sm_instance);

   if (output_number == CONNECT)
      {
     /* Round robin through our available host connections and request
      * NC to do the connection.
      */
      options = NO_EXP_OR_CON;
      minst->cir = nc_open_circuit((void *)minst,nc,"orksample",
            &host_saddr[next_host_addr],nc_frame,nc_bias,0,1,options);

     /* If the circuit returned is NULL, we have a serious problem, give
      * orkhestra the details, telling him I have failed.
      */
      if (!minst->cir)
         ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s",nc_error(nc));

      minst->is_connected = 0;
      next_host_addr++;
      if (next_host_addr >= host_addr_cnt)
         next_host_addr = 0;
      return;
      }

   if (!minst->is_connected)
      {
     /* The instance is not connected - inform the state machine.
      */
      orkcpapi_send_input(orkcpapi,sm_instance,0,DISCONNECT,NULL);
      return;
      }

   switch(output_number)
      {
      case DISCONNECT:
         nc_close_circuit(minst->cir);
         break;
      case GENERIC_REQUEST:
         if (max_messages)
            {
            if (++num_messages > max_messages)
               {
               shutdown_orksample();
               break;
               }
            }
```

```
         snprintf(buf,sizeof(buf),"Bla Bla inst(%d), message(%u)",sm_instance,
               num_messages);
         nc_send(minst->cir,strlen(buf),buf);
         break;

      /* This should not happen - it is a bug in this program:
       * I have an output in my_in_outputs array, and I did not
       * catered for it?
       */
       default:
         ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"Received an "
               "Invalid output (not implemented) for instance %d: %s",
               sm_instance,output_name);
         break;
       }

   } /* output_from_sm */

  /* Function Init()
   * Initialise orksample:
   * Verify parameters and initialises all the relevant instances.
   * There is two actions on errors encountered:
   * 1. If the open of orkcpapi failed: verbose to stderr and exit().
   * 2. Any other error, send a "ORKCPAPI_CP_FAILED" message to the orkhestra
   *    and return -1.
   *
   * Return zero on successful, -1 on error.
   */
static int init(int argc,char *argv[],char *envp[])
   {
   poptContext optcon;        /* Context for parsing command line options */
   int rc_pop, i, recno=0, port;
   char t[256];
   FILE *fd;
   char buf[80], *cp;

   optcon = poptGetContext(argv[0],argc,(const char **)argv,optionsTable,0);
   rc_pop = poptGetNextOpt(optcon);

   if (debug) getchar();

  /* open the orkhestra API.
   */
   orkcpapi = orkcpapi_open(from_pipe,to_pipe,my_in_outputs,sm_notification,
         msg_from_sm,output_from_sm);
   if (!orkcpapi)
      {
     /* orkhestra API failed to initialise, Function orkcpapi_error() with a
      * NULL parameter returns an ASCII string describing the reason.
      */
      fprintf(stderr,"orksample: ****** ABORTED ************\n");
      fprintf(stderr,"orksample: %s\n",orkcpapi_error(NULL));
```

```
    fprintf(stderr,"orksample: ****** ABORTED ************\n");
    exit(12);
    }

/* Any errors from here onwards, we will notify orkhestra that I have
 * failed with an appropriate description of the failure.
 */
 if (rc_pop < -1)
    {
    ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s - %s",
       poptBadOption(optcon,0),poptStrerror(rc_pop));
    return -1;
    }

/* NC bias and framing
 */
 if (!frame_format)
    {
    ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s\n","Required --nc-format
          " specified!\n");
    return -1;
    }
 for (i = 0; f_format[i].name; i++)
    if (!strcmp(f_format[i].name,frame_format))
       break;
 if (!f_format[i].name)
    {
    ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s\n","Invalid --nc-format "
          "specified!\n");
    return -1;
    }
 nc_frame = f_format[i].value;

/* Load our TCP/IP addresses.
 */
 if (!host_addr_file)
    {
    ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s\n","Required "
           "--host-addr-file missing!");
    return -1;
    }
 fd = fopen(host_addr_file, "r");
 if (!fd)
    {
    ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"Error opening "
          "host address file %s: %s", host_addr_file, strerror(errno));
    return -1;
    }
 while (fgets(buf, sizeof(buf), fd))
    {
    recno++;
    if (strlen(buf) < 3 || buf[0] == '#')
```

```
      continue;
   if ((cp = strchr(buf, '\n'))) *cp = '\0';
   if ((cp = strchr(buf, '\r'))) *cp = '\0';
   if (!(cp = strchr(buf, ':')))
      {
      ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"Invalid addr at "
            "record [%s] at recno %d\n", buf, recno);
      return -1;
      }
   *cp = '\0';
   sscanf(cp+1, "%d", &port);
   host_saddr = (struct sockaddr_in *)realloc(host_saddr,
         (host_addr_cnt+1)*sizeof(*host_saddr));
   if (nc_resolve_ipaddr(nc,&host_saddr[host_addr_cnt],buf,port) < 0)
      {
       ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"Unable to resolve IP "
             "address: %s",nc_error(nc));
      return -1;
      }
   host_addr_cnt++;
   }
 if (host_addr_cnt < 1)
   {
   ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s\n","require at  "
         "least one host name and port\n");
   return -1;
   }
 fclose(fd);

/* Open our NC instance.
 */
 nc = nc_open(NULL,nc_status_change,nc_data_received,
       NULL,NULL);
 if (!nc)
   {
   ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"%s\n","can not open NC insta
         );
   return -1;
   }

/* If TCP trace required, ask NC to do the honours.
 */
 if (tcp_trace_fname)
   if (nc_open_trace(nc,tcp_trace_fname) < 0)
      {
       ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_CP_FAILED,"Unable to open "
             "trace file: %s",nc_error(nc));
      return -1;
      }

/* Set signal_handler() to intercept the 'kill' signals.
 */
```

```
   sigset ( SIGHUP , signal_handler );   /* Sig  1. */
   sigset ( SIGINT ,signal_handler );   /* Sig  2. Ctrl C */
   sigset ( SIGQUIT,signal_handler );   /* Sig  3. Ctrl \ */
   sigset ( SIGTERM,signal_handler );   /* Sig 15         */

   return 0;
   } /* init */

  /* Function shutdown_orksample()
   * This function do a logical shutdown, clean up a few things and exit the
   * program.
   * Note there is no return out of this function.
   */
static void shutdown_orksample(void)
   {
   printf("orksample: shutting down\n");
   if (nc)
      {
      if (tcp_trace_fname) nc_close_trace(nc);
      nc_close(nc);
      }
   exit(0);
   } /* shutdown_orksample */

  /* Function signal_handler()
   * Intercepts the 'kill' signals. I will ignore signal SIGHUP, for the rest,
   * I will call shutdown_orksample() to terminate the program.
   */
static void signal_handler(int signum)
   {
   if (signum == SIGHUP)
      {
      printf("orksample: SIGHUP received\n");
      return;
      }
   shutdown_orksample();
   } /* signal_handler */

  /* Function nc_status_change()
   * Callback function for NC circuit status changes.
   * *circuit is the  circuit this call is applicable to.
   * status is the new status of the circuit and *msg is a description
   * of the status.
   */
static void nc_status_change(nc_circuit_t *circuit,int status,char *msg)
   {
   mch_inst_t *mch_inst;

   mch_inst = (mch_inst_t *)circuit->user_data;
   if (verbose)
         ORKCPAPI_SEND_MESSAGE(orkcpapi,ORKCPAPI_INFORMATION,"Instance %d: "
               "%s",mch_inst->sm_instance,msg);
```

```
   switch (status)
      {
      case NC_CONNECTED:
         mch_inst->is_connected = 1;
         orkcpapi_send_input(orkcpapi,mch_inst->sm_instance,0,CONNECT,NULL);
         break;
      case NC_DISCONNECTED:
         mch_inst->is_connected = 0;
         mch_inst->cir = NULL;
         orkcpapi_send_input(orkcpapi,mch_inst->sm_instance,0,DISCONNECT,NULL);
         break;
      }
   }  /* nc_status_change */

  /* Function nc_data_received()
   * Callback function for data received on the circuit.
   *
   * *circuit is the circuit that the data is received on, len the length of
   * data received and *buf the data.
   * *time_stamp is the time stamp when the data was received.
   *
   * Return zero on success. If an error occurs return -1, which will
   * result in the circuit being closed.
   */
static int nc_data_received(nc_circuit_t *circuit,int len,char *buf,
      struct timeval *time_stamp)
   {
   mch_inst_t *mch_inst;

   mch_inst = (mch_inst_t *)circuit->user_data;

  /* Process the data:
   * It is a response to the canned message I sent previously,
   * send input GENERIC_RESPONSE to the state machine.
   */
   orkcpapi_send_input(orkcpapi,mch_inst->sm_instance,0,GENERIC_RESPONSE,
         time_stamp);

   return 0;
   }  /* nc_data_received */

  /* Function get_sm_instance() returns the data structure for an instance.
   * If it is the first time a particular instance is seen,
   * create one for the instance.
   */
mch_inst_t *get_sm_instance(int sm_instance)
   {
   mch_inst_t *misnt;
   int n;

   if (sm_instance >= mch_inst_asz || !mch_inst[sm_instance])
      {
```

```
    if (sm_instance >= mch_inst_asz)
      /* Need to expand the instance array
       */
       {
       n = sm_instance+10;
       if (!(mch_inst = (mch_inst_t **)realloc(mch_inst,
             n*sizeof(*mch_inst))))
          {
          perror("get_sm_instance(): realloc failure");
          abort();
          }
       memset(mch_inst+mch_inst_asz, 0,
              (n-mch_inst_asz)*sizeof(*mch_inst));
       mch_inst_asz = n;
       }

    /* Create a new instance and initialises it.
     */
     misnt = (mch_inst_t *)malloc(sizeof(*misnt));
     if (!misnt)
        {
        perror("get_sm_instance(): malloc failure");
        abort();
        }
     memset(misnt,0,sizeof(*misnt));
     misnt->sm_instance = sm_instance;
     mch_inst[sm_instance] = misnt;
     return misnt;
     }
  else
     {
     assert(mch_inst[sm_instance]->sm_instance == sm_instance);
     return mch_inst[sm_instance];
     }
  } /* get_sm_instance */
```