

FREESCADA
TECHNICAL DOCUMENTATION

Patrik Sundberg
<ps@raditex.se>

Robert Johansson
<rob@raditex.se>

Göran Hasse
<gh@raditex.se>

September 6, 2004

Preface

FREESCADA stands for “Free Supervisory, Control and Data Acquisition” and is a system based on opensource-components. The system is designed for controlling and/or collecting data about various processes. The technology used in achieving this is normal modern PC-hardware in combination with various sensors and actuators connected to the PC.

From what we know no such system previously existed in the opensource-community and this approach to implementing SCADA systems is a bit different from most traditional systems, which aren’t based on general purpose computers and networking. FREESCADA is created by computer practitioners and therefore use technologies common in the computer industry, such as SQL-databases, TCP/IP, HTTP and general purpose programming languages. This gives the developer a much more advanced and flexible environment compared to systems not based on general purpose computers and operating systems. We think it gives the developer the chance to create much more elaborate systems which perform better than their competition.

What this document will try to do is to present FREESCADA from a technical standpoint. It will cover:

- an overview of the different parts making up the whole
- how the pieces fit together
- how to install FREESCADA
- how to configure FREESCADA
- some maintenance tasks, like doing backups and similar things

Since FREESCADA is a moving target every little detail of the system is hard to cover in this document, but this document should at least summarize the most important aspects of dealing with FREESCADA¹.

/The FREESCADA-team

¹The most recent version of this document is available from <http://www.freescada.org>

Contents

1	Overview	1
1.1	FREEBSD	1
1.2	The database	2
1.3	fshwd	2
1.4	libfsbc	2
1.5	libfshw	2
1.6	fshw-modules	2
	fshwmod-ds1820	3
	fshwmod-ipio	3
	fshwmod-fecfc34	3
	fshwmod-snmptemp	3
	fshwmod-pci7250	3
	fshwmod-pci6208	3
	fshwmod-dummy	3
1.7	libfsrp	3
1.8	ruby-fsrp	3
1.9	fscvud	3
1.10	fslogd	3
1.11	fslogicd	4
1.12	fsalarmd	4
1.13	fsmed	4
1.14	fsguard	4
1.15	Various small tools	4
	1.15.1 fsdb_create.rb	4
	1.15.2 fs_useradd.rb, fs_userdel.rb and fs_userlist.rb	4
	1.15.3 fs_hwscan.rb	4
2	Installation	5
2.1	Hardware installation	5
	2.1.1 Supported Hardware	5
	2.1.2 DS1820	5
	Kernel	5
	Sensor ID's	5
	2.1.3 IP/IO module	6

2.2	FREEBSD Software	6
2.2.1	PORT Applications	6
2.3	FREESCADA Software	7
2.3.1	FreeSCADA Compoments	7
2.3.2	FreeSCADA PORTS	7
2.3.3	FreeSCADA From Source	8
2.4	FreeSCADA Configuration	8
2.4.1	Configuration files	8
	/etc/services	8
	/usr/local/etc/fscvud.rc	9
	/usr/local/etc/fslogd.rc	9
	/usr/local/etc/fslogicd.rc	9
	/usr/local/etc/fsalarmd.rc	9
	/usr/local/etc/fsmcd.rc	10
	/usr/local/etc/fsreportd.rc	10
	/usr/local/etc/fsguard.conf	10
2.5	Startup the FreeSCADA system	12
2.5.1	FreeSCADA Applications	12
	Debug mode	13
2.5.2	Startup at system start	13
	08ntpdate.sh	13
	09fsguard.sh	13
2.6	LOGIC scripts	13
2.6.1	Installation of demo system	14
2.7	What to do next	16
2.7.1	Checklist	16
3	The FreeSCADA Hardware Architecture	17
3.1	Introduction	17
3.2	Devices and Properties	17
3.3	Device Id's	17
3.4	FreeSCADA Hardware modules	18
	3.4.1 Device structures	18
	3.4.2 Property structures	18
	3.4.3 Read and Write Implementations	19
3.5	An Example - The Dummy Driver	19
3.6	The fsbc library	21
	3.6.1 Error Handling	22
	3.6.2 The FSBC Type System	22
	3.6.3 Data Structures	23

4	Control Scripts in FreeSCADA	25
4.1	Introduction	25
4.1.1	Processes in FreeSCADA	25
4.2	Control Scripts	25
4.2.1	Ruby Control Scripts	26
4.3	The RUCS library	26
4.3.1	DbVar	26
	Control Variables	27
4.3.2	DbRel	28
	Control Relations	28
4.3.3	DbDev	29
	Devices	30
4.3.4	HwDev	30
4.3.5	Alarms	31
4.3.6	ControlScript	31
4.4	The fsrp library	32
4.4.1	Device	32
4.4.2	Property	33
4.5	Additional ruby libraries	34
4.5.1	PIDRegulator	35
4.5.2	SunInfo	35
4.6	Programming Control Script	35
4.6.1	Overview	35
4.6.2	Typical Structure of a Control Script	35
4.7	Orderly shutdown and restart of script	37
4.8	Running control scripts	37
4.8.1	Running control scripts manually	37
4.9	Example control scripts	37
4.9.1	produce warm water	38
5	Additional resources	41
5.1	Ruby	41
5.2	Electrical installation	41
5.2.1	Lightning	41

List of Figures

1.1 *FreeSCADAS components*. 1

List of Tables

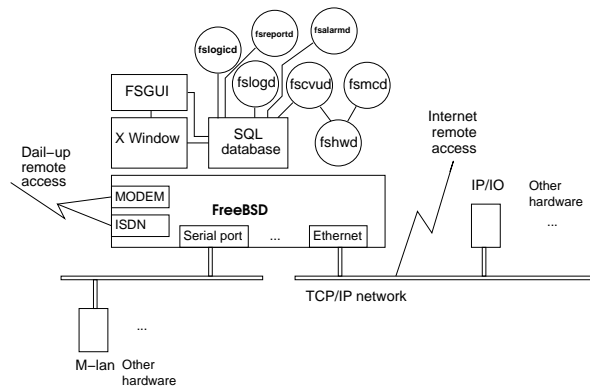
Chapter 1

Overview

One thing to note from the start is that FREESCADA is based on a UNIX operating system and therefore many of the design-decisions are strongly influenced by the UNIX-philosophy. Key principles are:

- Small specialized parts
- Combine small tools to accomplish greater tasks
- Keep things simple
- Use TCP/IP
- Use a SQL-database

A consequence of keeping parts small is that the number of parts gets rather large. In this chapter every part of the FREESCADA-system is presented and a concluding graph displaying how they are tied together is given.



\$Id: fsstructure.fig,v 1.2 2003/08/01 15:33:57 rob Exp \$

Fig. 1.1: *FreeSCADAS components.*

1.1 FREEBSD

The FreeSCADA system is primarily designed to run on the FreeBSD operating system. In order to ease the installation and configuration of FreeSCADA, a custom distribution of FreeBSD is used. This FreeSCADA version of FreeBSD contains all the required packages including the FreeSCADA system, and during installation the FreeSCADA system is automatically configured.

FreeSCADA is also compatible with other UNIX systems, such as Linux for example, but with a slightly more complex installation procedure.

1.2 The database

A very central part of FREESCADA is the database. FREESCADA uses a SQL-database for almost every part of it's operation, it is used both for keeping configuration and storing historical data about the controlled/observed process. In version 1.0 of FREESCADA the DBMS MySQL is used, future versions are planned to be based on PostgreSQL since it is more standards compliant and more likely to not get entangled in bothersome copyright problems.

1.3 fshwd

fshwd stands for "FREESCADA HardWare Daemon" and is the component which gives and receives physical signals via sensors and actuators connected to the computer running fshwd. In the long run we hope that all sensors and actuators will be operable via TCP/IP, but until then some PCI-cards is also used. fshwd uses libfshw to access all hardware, libfshw is described in section (1.5). fshw is a really simple daemon that just listens for client-requests, tries to serve the request and returns the result to the client. In order to talk to fshwd clients use a protocol called "fsrp" which is described with libfsrp in section (1.7). fshwd keeps no configuration, it's just a dumb slave.

1.4 libfsbc

libfsbc (FreeSCADA Basic Components) is a library that contains basics functionality common to most FreeSCADA applications. It includes type definitions, data types such as link list, hash tables etc. It also contains error handling machanism.

1.5 libfshw

libfshw is a library designed for uniform access to different hardware. It abstracts different hardware into "devices" and "properties".

A device in the FREESCADA framework means a single entity of controllable and/or observable equipment. This can for example be a (single) relay on a PCI relaycard or a temperaturesensor on a Dallas-bus

A property is a characteristic of a given device. A device can have many properties, but most simple devices just has one, e.g. a temperaturesensor device only has the property "temperature". It is the properties values that are read and/or written in order to get/set the current state of the process.

libfshw is plug-in/module based in order to reduce coupling between device-drivers, libraries used to access hardware, etc. What this means is that with libfshw alone you can't manipulate a single type of hardware. In order to do so you need the fshw-module corresponding to your hardware. fshw-modules are described in section (1.6).

1.6 fshw-modules

A fshw-module is a bit of code abstracting a specific hardware to the devices and properties interface used by libfshw. A libfshw might operate a PCI card, communicate on a Dallas-bus or communicate with hardware via TCP/IP - all this is hidden for the user who only needs to know the devices and properties interface presented via libfshw. Writing a fshw-module is a quite simple task (depends somewhat on the hardware of course), and modules are added as new hardware is used with FREESCADA. In the rest of this section the current fshw-modules are presented.

fshwmod-ds1820

This module uses libmlan to receive temperatures from a Dallas-sensor called “ds1820”.

fshwmod-ipio

This module uses libipio to read and write states on a Raditex IP/IO module.

fshwmod-fecfc34

This module uses libfesto to read and write states on a Festo FEC FC34 module.

fshwmod-snmptemp

This module uses the net-snmp library to communicate with a SNMP compatible temperature sensors.

fshwmod-pci7250

This module operates the PCI7250 PCI-card. It depends on a devicedriver for this card being present (either in the kernel or as a lkm). In order to build this module the include-files for the device driver need to be installed.

fshwmod-pci6208

This module operates the PCI6208V PCI-card. It depends on a devicedriver for this card being present (either in the kernel or as a lkm). In order to build this module the include-files for the device driver need to be installed.

fshwmod-dummy

This module is a dummy driver to show how the driver interface should be constructed.

1.7 libfsrp

The libfsrp library (FreeSCADA Remote Protocols) contains the protocol definitions and a complete set of protocol functions. It includes request sending, receiving and acking mechanism, packet packing and unpacking functions etc.

1.8 ruby-fsrp

A glue library between Ruby and the fsrp-protocol.

1.9 fscvud

FreeSCADA Current Value Update Daemon. This application keeps the current value fields in the database updated by sending requests of the hardware's state to the FreeSCADA Hardware Daemon.

1.10 fslogd

FreeSCADA LOG Daemon. fslogd watches the current value fields in the database (which are updated by fscvud) and inserts entries in the history value table when the current value has been changed.

1.11 fslogiced

FreeSCADA LOGIC Daemon.

1.12 fsalarmd

FreeSCADA ALARM Daemon. When there is errors in the database this application sends mail and sms messages to the contact persons for the site.

1.13 fsmcd

FreeSCADA Manual Control Daemon. This daemon manages the devices that are set in “manual” mode (as opposed to “automatic” mode) and updates the hardware (via fshwd) according to the values specified in the database.

1.14 fsguard

FreeSCADA Guard. This daemon supervises all other FreeSCADA components and makes sure that they are staying alive and doing their job. This is a safe-guard mechanism that ensures the stability and robustness of the FreeSCADA system.

1.15 Various small tools

In addition to the applications described above, the FreeSCADA system contains a set of minor tools in order to ease the administration and configuration of the system. Details of how the utilities are used is available by passing the “-h” flag when starting them from the command prompt.

1.15.1 fsdb_create.rb

This utility is used when new sites are created. It loads the database definition into the MySQL database. It can either be use directly from the prompt, or indirectly from the FreeSCADA Graphical User Interface.

1.15.2 fs_useradd.rb, fs_userdel.rb and fs_userlist.rb

These tools are use when administrating the FreeSCADA database accounts. In order to log on to the system using the FreeSCADA Graphical User Interface, a FreeSCADA database account is needed.

1.15.3 fs_hwscan.rb

This application scans the system for dallas temperature sensors and IP/IO modules and stores the result in the FreeSCADA database. It's meant to simplify the configuration of the FreeSCADA site.

Chapter 2

Installation

2.1 Hardware installation

2.1.1 Supported Hardware

FreeSCADA mainly uses the temperature sensor DS1820 from Dallas Semiconductors and the IP/IO module from Raditex. But there's also support for a set of ISA/PCI cards and as well as other devices for the dallas one-wire bus.

2.1.2 DS1820

Kernel

In order for the DS1820 device driver to work, the FreeBSD kernel must be compiled with the FIFO disabled on the serial port device driver. This kernel option is default on the FreeSCADA distribution but if a standard FreeBSD kernel is used, this must be considered! The FIFO is disabled by adding a flag to the sio driver configuration. Your kernel configuration file should contain these lines:

```
device      sio0      at isa? port IO_COM1 flags 0x2 irq 4
device      sio1      at isa? port IO_COM2 flags 0x2 irq 3
```

Sensor ID's

After the physical installation of the temperature sensors one uses one of the bus scan programs that comes with the installation of the M-lan library. Both `search_mlan` and `mlan_tool` could be used and they are very similar. The purpose is to get the sensors id's which will be needed later in the installation process. Futher more, you need to sort out which sensor id belong to which physical device.

```
fs-demo# ./mlan_tool /dev/cuaa0
Device 3000000024C84410 ds1820 temperature sensor
Device CE00000025B24C10 ds1820 temperature sensor
Device 0900000025A7FE10 ds1820 temperature sensor
Device 1200000025A83B10 ds1820 temperature sensor
Device AD000000250AC710 ds1820 temperature sensor
Device C2000001888DE209 unknown family
fs-demo# ./search_mlan /dev/cuaa0
```

```
/-----
Loop to find all iButton on MicroLAN.
Press any key to stop.
```

```

----- Start of search
(1) 3000000024C84410
(2) CE00000025B24C10
(3) 0900000025A7FE10
(4) 1200000025A83B10
(5) AD000000250AC710
(6) C2000001888DE209
----- End of search

(ENTER to search again, Q or q to Quit) q
fs-demo#

```

2.1.3 IP/IO module

The IP/IO module is a device which is connected to an Ethernet network. Every IP/IO module thus need an ip address. The addresses are assigned to the modules using the ARP protocol and the **arp** unix program.

```
fs-demo# arp -S 192.168.0.106 00:00:00:10:30:50 pub
```

To test the communication you can ping the IP/IO module, but not with more than 24 bytes of data. Use the **-s** option to the **ping** program.

```

fs-demo# ping -s 12 192.168.0.106
PING 192.168.0.106 (192.168.0.106): 12 data bytes
20 bytes from 192.168.0.106: icmp_seq=0 ttl=64 time=1.001 ms
20 bytes from 192.168.0.106: icmp_seq=1 ttl=64 time=0.990 ms
^C
--- 192.168.0.106 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.990/0.995/1.001/0.005 ms
fs-demo#

```

If you have severall IP/IO modules on you LAN, you can ping the broadcast address and all IP/IO modules will respond.

When the “libipio” is installed there is a bin catalogue in the build directory and you can run the programs there to test the module.

To read the relay settings you could run

```
./iprelcmd -h 192.168.56.20 -p 8 -r
```

And to set all the relays you could run

```
./iprelcmd -h 192.168.56.20 -p 8 -w 0x0f
```

2.2 FREEBSD Software

2.2.1 PORT Applications

The PORTS are the software distribution system that FreeBSD uses. A PORT is a precompiled software package ready for installation on a FreeBSD system. The PORTS packages sources are located in the PORTS tree, `/usr/ports` on a FreeBSD system. In order to install or remove a PORTS package, the **pkg_add**, **pkg_delete** etc. are used, or they may be build from the source in the ports tree.

FreeSCADA depends on the following ports to be installed

- mysql 3.23.42 client and server
- ruby 1.6.7, swig-1.3.12
- ruby-mysql
- gtk 1.2.10
- net-snmp
- gnuplot, ruby-gnuplot, t_eTeX, ghostscript, dvipdfm.

2.3 FREESCADA Software

2.3.1 FreeSCADA Components

The FreeSCADA system is divided into several components and a complete installation requires that the following components are installed.

libfsbc A library named “basic components”

libfshw A library for the hardware daemon.

libfsrp A library for FreeSCADA remote protocol.

lipipio A library for the IP/IO module.

libmlan A library for the m-lan bus from Dallas.

ruby-fsrp A glue between ruby and FreeSCADA remote protocol.

fshwd hardware daemon

fshwmod-ds1820 loadable module for ds1820

fshwmod-ipio loadable module for ipio

fshwmod-dummy example loadable module

fscvud current value updater

fslogd logging daemon

fsmed manual control daemon

fslogicd logic daemon

fsreportd report daemon

2.3.2 FreeSCADA PORTS

The software component of FreeSCADA which were listed in the preceding section may either be build and installed separately or installed from the freescada ports packages **fs-server** and **fs-client** which is available from the ports tree `/usr/ports` on a FreeSCADA system or from the FreeSCADA cd.

2.3.3 FreeSCADA From Source

All FreeSCADA packages are build either with the autotools system or with a ruby setup.rb script, for c programs and ruby programs respectively.

With the autotools system the following procedure are used:

- ./bootstrap
- ./configure
- gmake
- gmake install

There are substancial dependencies amongst the FreeSCADA components as well as to external software packages. The FreeSCADA release tools aims to resolve these issues.

The ruby programs uses another configuration and installation system. To install one of the ruby programs, your run

- ruby setup.rb config
- ruby setup.rb install

i.e.

```
fs-demo# ruby setup.rb config
entering config phase...
config done.
fs-demo# ruby setup.rb install
entering install phase...
mkdir -p /usr/local/bin
install fslogd.rb /usr/local/bin
mkdir -p /usr/local/share/
mkdir -p /usr/local/share/./examples/ruby/fslogd
install ./examples/ruby/fslogd/fslogdrc /usr/local/share/./examples/ruby/fslogd
install done.
fs-demo#
```

2.4 FreeSCADA Configuration

2.4.1 Configuration files

Create or modify the following files and substitute the values of the variables according to your system.

`/etc/services`

Add the listening port for the FreeSCADA hardware server to the services file, i.e. add the line "fsrp 31337/udp" to the `/etc/services` file.

```
echo "fsrp 31337/udp" >> /etc/services
```

/usr/local/etc/fscvud.rc

```
# configuration file for fscvud

# which host, user etc to use when connecting to the DB
host = localhost
user = fs
password = scada

# which site to log for
sitename = democ

# how often to update the current values in the DB
update_intervall = 60
paus_time = 5
```

/usr/local/etc/fslogd.rc

```
# configuration file for fslogd

# which host, user etc to use when connecting to the DB
host = localhost
user = fs
password = scada
# which site to log for
sitename = democ

# how often to log the values to DB
update_intervall = 20
# archive every week
archive_intervall = 604800
# delete entries older than one month
archive_time = 2678400
```

/usr/local/etc/fslogicd.rc

```
# configuration file for fslogicd

# which host, user etc to use when connecting to
# the DB
host = localhost
user = fs
password = scada
# which site to run logic for
sitename = salk

# how often to reread site configuration from DB
update_intervall = 20
```

/usr/local/etc/fsalarmd.rc

```
# configuration file for fsalarmd

# which host, user etc to use when connecting to
# the DB
host = localhost
user = fs
```

```
password = scada
# which site to alarm for
sitename = salk
# how often to check for alarms
check_intervall = 10
# archive every day
archive_intervall = 84600
# delete entries older than one week
archive_time = 604800
```

/usr/local/etc/fsmcd.rc

```
# configuration file for fsmcd

# which host, user etc to use when connecting to
# the DB
host = localhost
user = fs
password = scada
# which site to run logic for
sitename = salk

# how often to reread site configuration from DB
update_intervall = 20
```

/usr/local/etc/fsreportd.rc

```
# configuration file for freportd

# which host, user etc to use when connecting to
# the DB
host = localhost
user = fs
password = scada
# which site to run logic for
sitename = salk

# how often to reread site configuration from DB
update_intervall = 20
```

/usr/local/etc/fsguard.conf

```
# fsguard sample configuration file
#
# How should fsguard handle internal failures?
#
# soft: don't exit fsguard on failures such as resource shortage.
# hard: exit on all failures that disrupt the operation of fsguard.

handle_failures = soft

# Here goes the configuration for the separate daemons.
#
# Note that you must run them in the foreground because fsguard
# will lose control of them otherwise!

# FreeSCADA Hardware Daemon
```

```
fs daemon {
name = fshwd
path = /usr/local/bin/fshwd
args = -f
restart_wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fshwd.out
stderr = /var/log/fs/fshwd.err
}

# FreeSCADA cv(?) Update Daemon
fs_daemon {
name = fscvud
path = /usr/local/bin/ruby
args = /usr/local/bin/fscvud.rb -f
restart_wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fscvud.out
stderr = /var/log/fs/fscvud.err
}

# FreeSCADA Alarm Daemon
fs_daemon {
name = fsalarmd
path = /usr/local/bin/ruby
args = /usr/local/bin/fsalarmd.rb -f
restart_wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fsalarmd.out
stderr = /var/log/fs/fsalarmd.err
}

# FreeSCADA Log Daemon
fs_daemon {
name = fslogd
path = /usr/local/bin/ruby
args = /usr/local/bin/fslogd.rb -f
restart wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fslogd.out
stderr = /var/log/fs/fslogd.err
}

# FreeSCADA Logic Daemon
fs_daemon {
name = fslogicd
path = /usr/local/bin/ruby
args = /usr/local/bin/fslogicd.rb -f
restart_wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fslogicd.out
stderr = /var/log/fs/fslogicd.err
}
```

```
# FreeSCADA Manual Control Daemon
fs_daemon {
name = fscvud
path = /usr/local/bin/ruby
args = /usr/local/bin/fsmcd.rb -f
restart_wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fsmcd.out
stderr = /var/log/fs/fsmcd.err
}

# FreeSCADA Report Daemon
fs_daemon {
name = fscvud
path = /usr/local/bin/ruby
args = /usr/local/bin/fsreportd.rb -f
restart wait = 5
retries = 10
stdin = /dev/null
stdout = /var/log/fs/fsreportd.out
stderr = /var/log/fs/fsreportd.err
}
```

2.5 Startup the FreeSCADA system

2.5.1 FreeSCADA Applications

The FreeSCADA system is divided into serveral servers which provides different services. All these daemons must be started for full functionality of the FreeSCADA system. By starting the fsguard daemon, all other FreeSCADA server are started and kept alive.

fshwd

FreeSCADA Hardware Daemon. (Note when installing this program, the directory /usr/local/lib/fshwd must be created by hand and **LD_LIBRARY_PATH** must be configured accordingly).

fslogicd

FreeSCADA LOGIC Daemon.

fslogd

FreeSCADA LOG Daemon.

fsalarmd

FreeSCADA ALARM Daemon.

fscvud

FreeSCADA Current Value Updater Daemon.

fsreportd

FreeSCADA REPORT Daemon.

fsmcdd

FreeSCADA REPORT Daemon.

fsgruad

FreeSCADA GUARD.

Debug mode

Debug information is printed to the console if the hardware server (fshwd) is started with the **-d** option and the other daemons with the **-foreground**.

2.5.2 Startup at system start

To make the FreeSCADA system start up at boot time one can use the following set of resource files which should be put in the `/usr/local/etc/rc.d` directory.

08ntpdate.sh

```
#!/bin/sh

case "$1" in
  start)
    /usr/sbin/ntpdate fnv.narvarme.nu
    ;;
  stop)
    ;;
  *)
    echo ""
    echo "Usage: `basename $0` { start | stop }"
    echo ""
    exit 64
    ;;
esac
```

09fsguard.sh

```
#!/bin/sh

case "$1" in
  start)
    /usr/local/bin/fsguard
    ;;
  stop)
    killall fsguard
    ;;
  *)
    echo ""
    echo "Usage: `basename $0` { start | stop }"
    echo ""
    exit 64
    ;;
esac
```

2.6 LOGIC scripts

In the FreeSCADA system are all automation and control procedures defined in logic scripts which in turn are interpreted by the logic daemon. In principle any scripting languages could be used, but for now only scripts in Ruby programming language are supported.

All control scripts that the system uses are preferably placed in the `/usr/local/fslogicd/` directory and subdirectories in it. This is a convention only and the scripts could be placed anywhere in the file system.

When the control scripts are installed, the script is configured to the appropriate devices, variables, events etc. using the **fsgui** application.

2.6.1 Installation of demo system

Normally, the FreeSCADA system is configured using the FreeSCADA Graphical User Interface, but in order to verify that the system is installed correctly it may be convenient to manually set up a test site. This section describes how to do that.

First thing to do is to create a FreeSCADA Database. This can be done with the `fsdb_create.rb` and `fs_useradd.rb` utilities.

```
> fsdb_create.rb
FreeSCADA database creation tool
FreeSCADA site: demo
Database hostname [localhost]:
MySQL admin user [root]:
MySQL admin password:

FreeSCADA site = demo
Database hostname = localhost
Database username = root
Database password =

Is this configuration correct? (Y/N) y

Loading database file /usr/local/share/freescada/fsdb_create.sql into the database...
The FreeSCADA database for demo has now been installed!
>
```

```
> fs_useradd.rb
FreeSCADA User Addition Tool

FreeSCADA site: demo
Database hostname [localhost]:
MySQL admin user [root]:
MySQL admin password:

Is this correct? (Y/N) y

Username: fs
Password: *****
Access level: ADMIN

Is this correct? (Y/N) y
Adding the user to the database...
>
```

Next, the control scripts must be registered in the FreeSCADA database.

```
INSERT INTO script VALUES (1,'KeepAliveCounter',
'Detta är en enkel räknare','EXISTING_FILE','RUBY',
'/usr/local/fslogicd/ctrl_counter-0.1.rb');
```

To each of the control scripts, a control event must be connected if `fslogicd` is to start them up.

```
INSERT INTO control_event VALUES (1,1,'start','Y','ALWAYS','*|*|*|*|*|*|*',NULL);
```

The control scripts usually uses control variables, and if so these must be defined in the FreeSCADA database.

```
INSERT INTO control_variable VALUES (1,'keep_alive_counter',
'GLOBAL','INT',1799484,NULL,NULL,'Detta är en test','*|*|*|*|*|*|*',
'*|*|*|*|*|*|*');
```

Below, an example control script is listed which uses the control variable defined above and is started by **fslogi** via the script and control event also defined above.

```
#!/usr/local/bin/ruby
#
# $ Id: ctrl_counter-0.1.rb,v 1.1 2003/01/15 01:14:31 gh Exp $
#
# This scripts need a control_variable named
# keep_alive_counter
#
# It will update this counter every second
#
#

require 'fslogi/rucs'

initialize_RUCS
class MyControl < ControlScript

  def initialize
    @keep_alive_counter = DbVar.new('keep_alive_counter')
    super(method('my_control'), 1)
  end

  def my_control

    counter_temp = @keep_alive_counter.get
    print "The value was ", counter_temp
    print "\n"

    counter_temp = counter_temp + 1

    @keep_alive_counter.set(counter_temp)

    trap("USR1") {

      counter_temp = @keep_alive_counter.get

      print "The counter was : ", counter_temp, "\n"
      counter_temp = 550;
      @keep_alive_counter.set(counter_temp)

      print "Aj jag dog\n"

      exit(0)

    }
  end
end

begin
  cs = MyControl.new
  cs.loop
end
```

2.7 What to do next

When all the steps in this installation guide is performed and all FreeSCADA daemons are running on your system, the next thing to do is to generate the FreeSCADA database and configure the system. This is done with the FreeSCADA Graphical User Interface **fsgui** and a detailed description of how to do that is given in the FreeSCADA User Manual.

But before you go on, take a quite look at the checklist below to make certain that you completed all steps.

2.7.1 Checklist

This checklist list a number of conditions that must be fulfilled in order for the FreeSCADA system to function.

- Mysql must be running and a site database must exist (use fsgui or script in freescada/fsdatabase to create the site).
- The following daemons must be installed and start without error messages: **fshwd**, **fscvud**, **fsalarmd**, **fslogd**, **fslogicd**, **fsmcd**, **fsreportd**, **fsguard**.
- The fshw modules must be installed and accessible for fshwd. Make certain that the directory which holds the modules are in the library path (i.e. `setenv LD_LIBRARY_PATH /usr/local/lib/fshw`). Which modules you need depends on which hardware you use, but you should allways install the dummy module.

Chapter 3

The FreeSCADA Hardware Architecture

3.1 Introduction

The FreeSCADA system provides an abstraction of the hardware used by the system which makes it possible to access all devices by a common network based interface. All direct communication with the hardware is performed by the FreeSCADA Hardware Daemon (fshwd), which uses dynamically loadable modules as drivers for different hardware. Such a module is called a FreeSCADA Hardware Module (fshwmod).

The network protocol in which fshwd communicates with the clients is called FreeSCADA Remote Protocol (libfsrp), and the abstract hardware interface is provided by the FreeSCADA HardWare library (libfshw). A fshw module represent the layer between the abstract hardware framework and the actual hardware.

3.2 Devices and Properties

All hardware entities in the freescada system is represented by a device object, which in turn has one or more properties objects. The device object has a unique device id and besides a list of properties it also has hardware specific information.

A property object contain information about a property of the device. All properties has a name, a type and some attributes. For example, a temperature measurement device could have a property called “temperature” of the type double and the “READ-ONLY” attribute, but the details is up to the author of the hardware module which specifies the available properties for the device.

3.3 Device Id’s

The unique device id (devid) is used for addressing devices in the FreeSCADA system. The device id is made up of four separate fields: host, driver, interface and device. The fields are concatenated into a string with the “/” sign as separator, i.e. “host/driver/interface/device”. Examples of a real devid’s are “fs-demo.raditex.se/pci7250/0/relay”

The host field specifies the address of the host (IP-address or a FQDN) on which the FreeSCADA Hardware Daemon runs. The driver specifies which hardware module is to perform the requests. The interface field specifies which hardware entity or bus to use, as a driver probably is responsible for more than one physical device. The pci7250 driver for example, interprets the interface fields as a specification on which PCI card to use, if the system is equipped with more than one PCI7250 card. The ds1820 driver on the other hand uses the interface field as a specification on which one-wire bus to use (i.e. which serial port to use). Finally, the device field specifies which part of a piece of hardware to use, since it’s very common that hardware provides more than one functionality. The PCI7250 card for example provides 8 relays and 8 digital inputs, and in this case these are accessed through the “relay0”, “relay1”, ... , “relay7” and “di0”, “di1”, ... , “di7” device names.

As with the properties, the exact uses of the different fields of the device id is up to the author of the hardware module.

3.4 FreeSCADA Hardware modules

As mentioned in the previous chapter, the purpose of a FreeSCADA Hardware Module is to constitute a layer between the abstract hardware interface and the physical devices. Either as a complete device driver or as an indirect driver which in turn uses for example driver libraries (ds1820, ipio) or a native kernel driver (pci7250).

Every hardware module must implement a **open_device** function, which is called by the fshwd when the first request to the driver is recieved.

```
struct bc_device *open_device(char *devid);
```

The task of this function is to initialize the hardware and build a **struct bc_device** structure, which fshwd uses to keep track of the device. The **struct bc_string devid;** field in **struct bc_device** (see below) shall be copied from the **char *devid** parameter to **open_device** using the libfsbc function:

```
BC_STRING_SET(bc_d->devid, devid);
```

All types in the libfsbc framework is accessed and manipulated using a set of macros such as this one.

3.4.1 Device structures

```
struct bc_device {
void *data;
struct bc_string devid;
struct bc_list *properties;
};

struct hw_device {
void *data;
void (*close) (struct bc_device * d);
};
```

The **void *data** field in **struct bc_device** shall be pointing to a **struct hw_device**, and the **void *data** is only used by the driver module so it's usage is determined by the module author, e.g. it could point to a stucture with private data holding the devices state. The **void (*close) (struct bc_device * d)** field should contain a pointer to a function that frees all data associated with the device. It's called by fshwd when the device is requested to be closed.

3.4.2 Property structures

The major task in the **open_device** implementation is to build a list of the properties that the device is to provide.

```
struct bc_property {
void *data;

struct bc_string name;
struct bc_number type;
struct bc_number flags;
};

struct hw_property {
int (*read) (struct bc_device * d,
struct bc_property * p,
struct bc_number * value);
int (*write) (struct bc_device * d,
struct bc_property * p,
struct bc_number * value);
};
```

In analogy with the connection between **bc_device** and **hw_device**, the **void *data** field in **bc_property** structure should point to a **hw_property** structure. The other fields in the **bc_property** structures specifies the properties name, type and it's attributes. The type could one of int, double or string and the flags stores the attributes which is readable and writeable.

The **hw_property** structure contains pointers to functions which implements the read och write operation that is supported by the property. A "temperature" property, for example, would likely only implement the read operation, and the **write** field would thus be set to **NULL**.

3.4.3 Read and Write Implementations

A read or a write request on a property of a device will eventually result in a call to the **read** or **write** implementation that the driver module supplies and it's there all communication with the hardware is performed.

```
int read(struct bc_device * d, struct bc_property * p, struct bc_number * value);
int write(struct bc_device * d, struct bc_property * p, struct bc_number * value);
```

3.5 An Example - The Dummy Driver

This section presents an example driver which gives an overall picture of how a hardware module is written. The modules consist of the functions **dummy_property_free**, **open_device**, **dummy_close_device**, **dummy_write_int**, **dummy_read_int**. The first function is passed to the linked list implementation, and has nothing to do with the structure of the hardware module.

The module provides a driver "dummy" which has the device "int" which in turn has the property of an readable and writeable integer.

```
#include <fsbc/fsbc.h>
#include <fshw/fshw.h>
#include "dummy.h"

static void
dummy_property_free(void *data)
{
    struct bc_property *p = (struct bc_property *) data;
    struct hw_property *hw_p = (struct hw_property *) p->data;

    free(hw_p);
    free(p);
}

struct bc_device *
open_device(char *devid)
{
    struct bc_device *d;
    struct hw_device *hw_d;
    struct bc_number *dummy_data;
    struct bc_property *p;
    struct hw_property *hw_p;

    if ((d = malloc(sizeof(struct bc_device))) == NULL) {
        BC_PANIC("libfshw", BC_ERR_SYS, "malloc failed");
        /* NOT REACHED */
        return NULL;
    }
    if ((hw_d = malloc(sizeof(struct hw_device))) == NULL) {
        free(d);
        BC_PANIC("libfshw", BC_ERR_SYS, "malloc failed");
    }
}
```

```

/* NOT REACHED */
return NULL;
}
if ((dummy_data = malloc(sizeof(struct bc_number))) == NULL) {
free(hw_d);
free(d);
BC_PANIC("libfshw", BC_ERR_SYS, "malloc failed");
/* NOT REACHED */
return NULL;
}

BC_STRING_SET(d->devid, devid);
d->data = hw_d;
d->properties = bc_list_new(dummy_property_free);

hw_d->close = &dummy_close_device;
BC_NUMBER_SET_INT(*dummy_data, 0);
hw_d->data = dummy_data;

/* Start building property "int": */
if ((p = malloc(sizeof(struct bc_property))) == NULL) {
bc_list_free(d->properties);
free(dummy_data);
free(hw_d);
free(d);
BC_PANIC("libfshw", BC_ERR_SYS, "malloc failed");
/* NOT REACHED */
return NULL;
}
if ((hw_p = malloc(sizeof(struct hw_property))) == NULL) {
free(p);
bc_list_free(d->properties);
free(dummy_data);
free(hw_d);
free(d);
BC_PANIC("libfshw", BC_ERR_SYS, "malloc failed");
/* NOT REACHED */
return NULL;
}
BC_STRING_SET(p->name, "int");
BC_NUMBER_SET_INT(p->flags, BC_PROPERTY_FLAG_READ | BC_PROPERTY_FLAG_WRITE);
BC_NUMBER_SET_INT(p->type, BC_NUMBER_TYPE_INT);
p->data = hw_p;

hw_p->read = &dummy_read_int;
hw_p->write = &dummy_write_int;

/* Insert property "int": */
if (bc_list_entry_insert(d->properties, p) == 0) {
free(p);
bc_list_free(d->properties);
free(dummy_data);
free(hw_d);
free(d);
BC_ERROR("libfshw", BC_ERR_INTERNAL, "couldn't create device correctly");
return NULL;
}

return d;

```

```

}

void
dummy_close_device(struct bc_device * d)
{
struct hw_device *hw_d = (struct hw_device *) d->data;
struct dummy_device_data *dummy_data = (struct dummy_device_data *) hw_d->data;

bc_list_free(d->properties);

free(dummy_data);
free(hw_d);
free(d);
}

int
dummy_read_int(struct bc_device * d,
               struct bc_property * p,
               struct bc_number * value)
{
struct hw_device *hw_d = d->data;
struct bc_number *dummy_data = hw_d->data;

if (BC_NUMBER_GET_TYPE(*dummy_data) == BC_NUMBER_TYPE_INT) {
*value = *dummy_data;
fprintf(stderr,
"dummy_read_int(%s, ...): returning value %d\n"
BC_STRING_GET(d->devid),
BC_NUMBER_GET_INT(*value));
return 1;
} else {
BC_ERROR("libfshw", BC_ERR_INPUT, "value last written was not an integer");
return 0;
}
/* NOT REACHED */
}

int
dummy_write_int(struct bc_device * d,
               struct bc_property * p,
               struct bc_number * value)
{
struct hw_device *hw_d = d->data;
struct bc_number *dummy_data = hw_d->data;

*dummy_data = *value;
fprintf(stderr, "dummy_write_int(%s, ...): writing %d\n",
BC_STRING_GET(d->devid),
BC_NUMBER_GET_INT(*value));

return 1;
}

```

3.6 The fsbc library

In the example driver in the previous section there's a lot of functions and macros from the FSBC library that is used. This sections will present the most important parts from a hardware module authors point of

view.

3.6.1 Error Handling

The FSBC library provides macros for error handling, which takes care of logging and error reporting. Neither of the two routines that are available every returns, so as far as the hardware module is concerned the execution stops whenever these error reporting macros are used. Two different levels of error are supported

```
BC_PANIC(domain, type, str)
BC_ERROR(domain, type, str)
```

The `str` argument is a free text description of what happened. The `type` is one of the following

- `BC_ERR_REMOTE`
- `BC_ERR_NOERROR`
- `BC_ERR_SYS`
- `BC_ERR_INPUT`
- `BC_ERR_NET`
- `BC_ERR_INTERNAL`
- `BC_ERR_IO`
- `BC_ERR_UNKNOWN`

and for a hardware module, the domain is "libfshw". Examples of usage of these macros is given in it's full context in the dummy driver in the previous section, and here in shorter form.

```
BC_ERROR("libfshw", BC_ERR_INTERNAL, "couldn't create device correctly");
...
BC_PANIC("libfshw", BC_ERR_SYS, "malloc failed");
```

3.6.2 The FSBC Type System

A part of the abstract driver framework, is the FSBC type system. A number or a string in FreeSCADA is represented by the structures `bc_number` (integer or double) and `bc_string`. The usage of these types are pretty straight forward and all access and manipulation of them are done with a set of macros which are defined in `fsbc/fs_number.h` and `fsbc/fs_string.h`. A summary of operation and types is given in the lists below

- `BC_NUMBER_TYPE_INT`
- `BC_NUMBER_TYPE_BOOLEAN`
- `BC_NUMBER_TYPE_DOUBLE`

In the list below `t` denotes a type (one of those listed above). `n`, `n1` and `n2` denotes integers. `s` denotes a char string. `d` denotes a double.

- `BC_NUMBER_CONST_INT(n)`
Make a constant `bc_number` of type `int`.
- `BC_NUMBER_CONST_DOUBLE(d)`
Make a const `bc_number`. Type `double`.
- `BC_NUMBER_GET_TYPE(v)`
Get type of `bc_number` value.

- **BC_NUMBER_SET_TYPE(n, t)**
Set type of bc_number value.
- **BC_NUMBER_GET_INT(n)**
Get bc_number value. Type int.
- **BC_NUMBER_SET_INT(n1, n2)**
Set bc_number value. Type int.
- **BC_NUMBER_GET_DOUBLE(d)**
Get bc_number value. Type double.
- **BC_NUMBER_SET_DOUBLE(d1, d2)**
Set bc_number value. Type double.
- **BC_NUMBER_EQUAL(n1, n2)**
Compare bc_numbers for equalness.
- **BC_NUMBER_LT(n1, n2)**
Compare bc_numbers, n1 lesser than n2?.
- **BC_NUMBER_GT(n1, n2)**
Compare bc_numbers, n1 greater than n2?.
- **BC_STRING_CONST(s)**
Make a const bc_string.
- **BC_STRING_GET(s)**
Get bc_string value.
- **BC_STRING_SET(s2, s2)**
Set bc_string value.

3.6.3 Data Structures

The most important data structure that libfsbc provides is the linked list. It is defined in the `fsbc/fs_list.h` and the most useful functions in hardware modules are shown below.

```
typedef void (*bc_list_free_func) (void *entry);
extern struct bc_list * bc_list_new(bc_list_free_func free_func);
extern int bc_list_entry_insert(struct bc_list * list, void *entry);
extern void bc_list_free(struct bc_list * list);
```

Once again, example usage is available in the dummy driver listed in section 3.5, and in short form below.

```
/* Insert property "int": */
if (bc_list_entry_insert(d->properties, p) == 0) {
...
bc_list_free(d->properties);
...
}
```


Chapter 4

Control Scripts in FreeSCADA

4.1 Introduction

In a UNIX-system it is natural to have many smaller processes doing different tasks, instead of a few more complicated processes. This is an important aspect of the UNIX philosophy. Processes in a UNIX-like environment are protected from one another by the operating system. If one process malfunction no other process must be interfered with. This situation also leads to a problem. Processes have difficulty to communicate with each other.

In FreeSCADA all process communications are handled via the database. All processes that want to hold states should use variables that are stored in the database. Status information like temperatures, on/off conditions and logical states are examples of such variables.

Thus, programs and control scripts should never keep states or status internally, but always store and fetch them in the database. This should be regarded as a general rule. If there are completely isolated variables or status, that nobody else is interested in, these could be maintained in a local program.

Depending on the values of the database variables programs and scripts take different actions, which for example maybe operations on hardware etc. So the database is the center of the FreeSCADA system and it provides a communication area as well as logging and storing functionality for the FreeSCADA processes.

4.1.1 Processes in FreeSCADA

As mentioned before, the FreeSCADA design is very much affected by the UNIX-philosophy. One of the main outcomes of this is that, to a large degree, every process performs one single task. This results in a large number of small processes.

In FreeSCADA two different kinds of processes are distinguished. Tasks that are fundamental to the systems functionality and doesn't depend on the particular FreeSCADA installation are written as stand alone daemons. These programs may be written in any programming language, but FreeSCADA currently use the C and Ruby languages.

On the other hand, tasks that are specific to the target that the FreeSCADA system operates are written as control scripts. Processes like these may for example be responsible for a single part of a automation installation, such as controlling a water heater etc.

4.2 Control Scripts

FreeSCADA is designed to support control scripts written in any programming language, but the FreeSCADA development team has selected Ruby as the programming language for implementation of control scripts.

Currently Ruby is the only supported scripting language, but support for other languages could easily be added. Ruby was selected because it's a modern scripting language with good support for object oriented construction.

All control scripts are controlled by the FreeSCADA Logic Daemon. **fslogicd** runs the control scripts according to definitions in the database. Control scripts may be defined to run at specific times, such as every hour, or continuously. **fslogicd** keeps track of the control scripts and restarts them if needed.

4.2.1 Ruby Control Scripts

The idea is that a control script should utilize the FreeSCADA system in order to perform a specific task. There are several resources in the FreeSCADA system that greatly simplifies the procedure of writing a control scripts.

- **fsrp** The FreeSCADA Remote Protocol and the libfsrp library with corresponding ruby bindings and wrapper classes provides a simple and efficient interface to the hardware connected to the FreeSCADA system.
- **rucs** The rucs library provides ruby classes for accessing the FreeSCADA database and the FreeSCADA alarm functionality.
- **FreeSCADA Database**
 - **devices** Represents hardware entities in the FreeSCADA system. **rucs** and **fsrp** provides wrapper classes for the database and hardware repectively.
 - **control variables** are connected to scripts and should hold any important state that the script uses. Accessed via the **rucs** library.
 - **control relations** are much like control variables, but provides a way of defining an arbitrary relation between two variables.

4.3 The RUCS library

One of the main part of the control script utilities is the RUCS (RUby Control Scripts) library. It provides ruby classes for accessing the FreeSCADA database tables and more. The RUCS library should be the only means of access a control script has and needs to the FreeSCADA system.

In order to use the RUCS library in a ruby script one need to include it using the `require` method, see example below. Further more, in order to startup the RUCS library, the `initialize_RUCS` method must be called. So before using any RUCS class, these lines must be stated.

```
require 'fslogicd/rucs'
initialize_RUCS
```

4.3.1 DbVar

The DbVar class is a wapper class for the `control_variable` table in the database. The DbVar class provides the methods `DbVar::get` and `DbVar::set(newval)` for accessing a control variable.

```
class DbVar
  def initialize(name)
    # name = name of the control variable
    ...
  end

  def get
    # return currently valid value of control variable
    ...
  end
end
```

```

    def set(value)
        # value = new value of the control variable
        ...
    end

    private
    ...
end

```

When creating an instance of the **DbVar** class, the constructor expects the name of the control variable as parameter.

```
db_var = DbVar.new('name_of_db_variable')
```

Example usage of the **DbVar** class is listed below.

```

require 'fslogid/rucs'

initialize_RUCS

#
# Create the object
#
produce_warmwater = DbVar.new('produce_warmwater')

#
# Get the current value of the control variable
#
current_value = produce_warmwater.get

#
# Set the value of the control variable
#
## ON
produce_warmwater.set(1)

## OFF
produce_warmwater.set(0)

```

Control Variables

The **control_variable** table in database stores the names, values, accessibility and validity of the control variables. The exact definition is listed below. **control_variable** control variables may be defined as LOCAL or GLOBAL, as well as having different values in different time intervals.

```

#
# This table is used to bind scripts, control events and script
# configuration together.
#
# each event can have several variables.

CREATE TABLE control_variable (
    script_id      INTEGER UNSIGNED NOT NULL,
    name           CHAR(64) NOT NULL,
    scope          ENUM('LOCAL', 'GLOBAL'),
    vtype          ENUM('INT', 'DOUBLE', 'STRING') NOT NULL,
    int_value      INTEGER,
    double_value   FLOAT,
    string_value   CHAR(32),

```

```

description    CHAR(255),
start          CHAR(128),  end          CHAR(128)

# should be in next version
# last_changed_by CHAR(32),
# last_changed    TIMESTAMP
);

```

4.3.2 DbRel

The DbRel class is a wrapper class for the **control_relation** table in the database. The DbRel class provides the methods **DbRel::get(xval)** which computes the value of the relation (graphically defined function) at the point **xval** and returns it.

```

class DbRel
  def initialize(name)
    # name is the name of the control relation in the database
    ...
  end

def get(val)
  # val is the value of the point where the control
  # relation is to be evaluated
  ...
end
end

```

When creating an instance of the **DbRel** class, the constructor expects the name of the control relation as parameter.

```
db_rel = DbRel.new('name_of_db_relation')
```

Example usage of the **DbRel** class is listed below.

```

require 'fslogicd/rucs'

initialize RUCS

#
# Create the object
#
input = ...
rel = DbRel.new('input_output_relation')

#
# Compute a output signal from the input signal
#
output = rel.get(input)

```

Control Relations

A control relation is very much similar to a control variable, except that they are a graphically defined function of two variables instead of a simple variable. Given a input value, a output value may be computed using a control relation, which are defined in the FreeSCADA Graphical User Interface.

```

# Table: control_relation
#
# This table is used to describe relations between different entities.

```

```
CREATE TABLE control_relation (
  name CHAR(64) NOT NULL,
  script_id INTEGER UNSIGNED NOT NULL,
  scope ENUM('LOCAL', 'GLOBAL'),
  rtype ENUM('LINEAR_APPROXIMATION') NOT NULL,
  start CHAR(128),
  end CHAR(128),
  x TEXT,
  y TEXT,
  xmax FLOAT DEFAULT 0.0 NOT NULL,
  xmin FLOAT DEFAULT 0.0 NOT NULL,
  ymax FLOAT DEFAULT 0.0 NOT NULL,
  ymin FLOAT DEFAULT 0.0 NOT NULL,
  description CHAR(255)
);
```

4.3.3 DbDev

The DbDev class is a wrapper class for the **device** table in the database. The DbDev class provides the methods **DbDev::get(property)** and **DbDev::set(property,value)** for retrieving and setting values of the property that the first parameter names. The constructor takes one argument, which contains the name of the device.

```
class DbDev
  ...
  def initialize(name)
    #
    ...
  end

  def get(property)
    #
    ...
  end

  def set(property, value)
    #
    ...
  end
end
```

Example usage

```
require 'fslogicd/rucs'

initialize_RUCS

dev_temp = DbDev.new('temp1')
temp1 = dev_temp.get('temperature')
print "The temperature is: #{temp1}\n"

dev_shunt = DbDev.new('dac1')
dac1 = dev_shunt.get('dac')
print "The shunt is: #{dac1}\n"

if temp1 <= 19.15
  dev_shunt.set('dac', 0.0)
else
```

```

dev_shunt.set('dac', 9.9 )
end

```

Devices

A device is an object that represent a peice of hardware in the FreeSCADA system. Each device has one or more properties which represents parts or properties of the device. A device may for example be a ds1820 temperature sensor which has been given a name, for example "GT1-VP". Such a device would have the property "temperature", and the procedure of retrieving is illustrated in the previous section. Other types of devices has other properties associated with it. For a complete list of available devices and their associated properties, see the FreeSCADA User Manual.

4.3.4 HwDev

The HwDev class is similar in it's interface and function to the DbDev class, except that the HwDev class operates via FreeSCADA Remote Protocol directly on the hardware instead of the device table in the database. The HwDev class provides the methods **DbDev::get(property)** and **DbDev::set(property,value)** for writing and reading the hardware associated with the class, and the property that the first parameter names.

The constructor takes one or three arguments. The first argument is required and is the name of the device. The seconds specifies the number of times a read or write operataion is to be retried before giving up, if the operation fails (default 4 retries). The third argument specifies the number of seconds between retries upon failure (default 1 second).

```

class DbDev
  ...
  def initialize(name, retries = 4, delay = 1)
    # name = name of the device
    # retries = number of times to retry if operation fails
    # delay = number of seconds to wait between retries
    ...
  end

  def get(property)
    # property = name of property to read
    ...
  end

  def set(property, value)
    # property = name of property to read
    # value = the new value to assign to the property
    ...
  end
end
end

```

Example usage

```

require 'fslogicd/rucs'

initialize_RUCS

temp1 = HwDev.new('temp1')
shunt = HwDev.new('dac1')

if temp1.get('temperature') <= 20.0
  shunt.set('dac', 0.00)
else
  shunt.set('dac', 10.0)
end
end

```

4.3.5 Alarms

The RUCS library also provides classes for raising FreeSCADA alarms. There is different alarm for different situations. The available alarm levels is presented in the following list.

- ScreenAlarm
- DebugAlarm
- InfoAlarm
- WarnAlarm
- ErrorAlarm
- FatalAlarm

All alarm levels is represented by a ruby class, and an alarm is raised by creating an instance of a alarm class, like this

```
# raise FreeSCADA alarm of level Xxx
XxxAlarm.new("Source", "Error message")
```

A realistic example of FreeSCADA Alarm usage is given below.

```
begin
  ...
rescue Exception => e
  DebugAlarm.new(__FILE__+'_'+__LINE__.to_s, "Caught exception: "+e)
end
```

4.3.6 ControlScript

The **ControlScript** class is the super class which all custom control scripts will extend. It provides a common interface to all control scripts which will simplify understanding and handling of different control scripts.

```
class ControlScript
  def initialize(meth, intervall)
    # meth = method to run
    # intervall = intervall at which meth shall be called
    ...
  end

  def do itteration
    ...
  end

  def loop
    ...
  end
end
```

A typical control script will have this structure. The general procedure is to extend the ControlScript class, creating an instance of it and running the **loop** method.

```
#!/usr/local/bin/ruby

require 'fslogicd/rucs'
initialize_RUCS
```

```

class MyControl < ControlScript

  def initialize
    ...
    super(method('my_control'), 1)
  end

  def my_control
    ...
  end
end

begin
  cs = MyControl.new
  cs.loop
end

```

4.4 The fsrp library

Another important part of the FreeSCADA ruby api libraries is the ruby wrapper classes for libfsrp, which is provided by the fsrp module. However, this library provides low a level interface to the FreeSCADA system, and as far as a control script is concerned, this library should only be used indirectly through the RUCS library describe earlier.

The most important part of fsrp is the **Device** and **Property** classes. These classes are used in the implementation of the HwDev class in the RUCS library. Thus, the fsrp library could also be used directly when a property is to be read from the hardware, but the HwDev class provides a much nicer interface including error handling and automatic read/write retries upon failure etc. However, the libfsrp provides a wider range of functionality and flexibility than the RUCS library.

The database value is kept up to date by the FreeSCADA Current Value Updater Daemon, **fscvud**, which uses the ruby bindings to the fsrp library directly and utilizes the more advanced functions in fsrp such as subscription etc.

4.4.1 Device

The **Device** class is a wrapper class for a FreeSCADA device. It's most important methods for are the **Device::open**, **Device::close**, **Device::getProperty(propname)**. The complete list of available methods is given in the source listing below. The constructor of the device object expects a parameter which contains the device id (see previous chapters). A device or a property which belong to the device can't be used unless the device is opened.

```

class Device
  ....
  def initialize(name)
    # name = devid id of the device to open
    ...
  end

  public
  ...
  def open
    # open the device
    ...
  end

  def close
    # close the device

```

```

    ...
end

def devid=(name)
  # assign a new devid
  ...
end

def devid
  # get the devid (if opened)
  ...
end

def properties
  # get a list of properties
  ...
end

def getProperty(propname)
  # get the property named propname
  ...
end

end

```

A typical usage of the Device object is shown below. A Device object is primarily used for retrieving properties.

```

dev = Device.new('localhost/ds1820/0/3000000024C84410')
dev.open
prop = dev.getProperty('temperature')
# use the property object
...
dev.close

```

4.4.2 Property

The **Property** class represents a property of a **Device**. When creating an instance of a **Property** class, one must supply a reference to a **Device** object. Usually, instances of **Properties** is created by the **Device::getProperty(propname)** method, so one don't need to bother about the constructor. The two most useful methods in the **Property** class is **Property::read** and **Property::write(value)**. As the names implies, these methods are used for reading and writing the value of the property. All calls to these methods results in a request being sent to the hardware daemon, and then in some effect on the physical hardware. Due to this, calls to **Property::read** and **Property::write(value)** may be slow or fast depending on the particular hardware associated with the **Property** and **Device** object.

```

class Property
  ...
  def initialize(device, prop)
    # device = reference to device object
    # prop   = property name
    ...
  end

  ...

  def read
    # get value of property
  end
end

```

```

    ...
end

def write(value)
  # set value of property
  ...
end

def name
  # get name of property
  ...
end

def flags
  # get flags of property
  ...
end

def type
  # get type of property
  ...
end

def discard
  # reset property
  ...
end

end

```

Below, a typical usage of the **Device** and **Property** classes are shown.

```

tempdev = Device.new('localhost/ds1820/0/3000000024C84410')
dacdev  = Device.new('localhost/ipio/ipio1:8/dac1')

tempdev.open
dacdev.open

tempprop = tempdev.getProperty('temperature')
dacprop  = dacdev.getProperty('dac')
temp     = tempprop.read
dac      = dacprop.read

if (temp < 20.0)
  dacprop.write(9.9)
else
  dacprop.write(0.0)
end

tempdev.close
dacdev.close

print "Current temperature is #{temp} and old dac value #{dac}\n"

```

4.5 Additional ruby libraries

In addition to the major ruby api libraries RUCS and fsrp, FreeSCADA supplies some minor ruby libraries of different character, which may be of more or less usefull. These libraries are installed in the site global ruby directory `/usr/local/lib/ruby/site_ruby/1.6` and in order to use them one need to include them using the **require** method.

For further information of how to use these classes, see the example programs that is installed together with them in the control script directory.

4.5.1 PIDRegulator

This ruby module provides a PID (Proportional/Integral/Derivative) regulator class. It can be used to calculate inputs signal to a process given the process's output signal and a referens value.

4.5.2 SunInfo

This module provides a class which calculates the time of sunrise and sunset given a date, latitude and longitude.

4.6 Programming Control Script

Programming a FreeSCADA is actually quite simple. Most control scripts will start by reading data from the database using the **DbVar** and the **DbDev** classes. Then, using the values of these datas it will decide weather to take action or not. A typical action would be to alter the state of some hardware in the system, such as for example turn of the water heater or something like that. This would be done using the **HwDev** class. If something goes wrong, a FreeSCADA alarm is raise using one of the Alarm classes.

4.6.1 Overview

- **RUCS** provides
 - **DbDev** read/write value of the device in the database.
 - **HwDev** read/write value of the device to the hardware.
 - **DbVar** read/write value of control variables in the database.
 - **DbRel** computation of a output given an input, using a graphically defined function.
 - **XxxAlarm** Raises FreeSCADA alarms.
 - **ControlScript** Super class which custom control scripts extends.
- **fsrp** provides
 - **Device** represents a FreeSCADA device, i.e. a peice of hardware.
 - **Properties** represents a property of a device. May be used to actually read and write states from the FreeSCADA hardware.

4.6.2 Typical Structure of a Control Script

This section shows how a typical control script would look like. The code listing below code be used as a skeleton when writing new control scripts.

```
#!/usr/local/bin/ruby
#
#      Example of typical FreeSCADA Control Script structure
#
require 'fsrp'
require 'fslogicd/rucs'

initialize_RUCS

class CustomControlScript < ControlScript
```

```

def initialize
  # create control variable objects
  @dbvar_var1 = DbVar.new('var1')
  @dbvar_var2 = DbVar.new('var2')
  # ...

  # create control relation objects
  @dbrel_rel1 = DbRel.new('rel1')
  @dbrel_rel2 = DbRel.new('rel2')
  # ...

  # create database device objects
  @dbdev_dev1 = DbDev.new('dev1')
  @dbdev_dev2 = DbDev.new('dev2')
  # ...

  # create hardware device objects
  @hwdev_dev1 = HwDev.new('dev1')
  @hwdev_dev2 = HwDev.new('dev2')
  # ...

  super(method('custom_procedure'), 1)
end

def custom_procedure
  # get values from variable
  var1 = @dbvar_var1.get
  var2 = @dbvar_var2.get

  # get values from relation
  output1 = @dbrel_rel1.get(input1)
  output2 = @dbrel_rel2.get(input2)

  # get values from db devices
  dbdev1 = @dbdev_dev1.get('property1')
  dbdev2 = @dbdev_dev2.get('property2')

  # get values from hw devices
  hwdev1 = @hwdev_dev1.get('property1')
  hwdev2 = @hwdev_dev2.get('property2')

  # get values from db devices
  @dbdev_dev1.set('property1', new_dbdev1)
  @dbdev_dev2.set('property2', new_dbdev2)

  # get values from hw devices
  @hwdev_dev1.set('property1', new_hwdev1)
  @hwdev_dev2.set('property2', new_hwdev2)

  # ...
end

end

# Execution starts here!
begin
  ccs = CustomControlScript.new
  ccs.loop

```

```
rescue Exception => e
  DebugAlarm.new(__FILE__+'_'+__LINE__.to_s, "Uncaught exception:\n"+e)
end
```

4.7 Orderly shutdown and restart of script

When **fslogiced.rb** is started it reads the database and starts those script that are ready for running.

If the user is sending the signal **USR1** to **fslogiced.rb** all its children are sent this signal. By establish a signal handler the children can do an orderly exit. The script programmer is responsible for this.

If the user is sending a child a **USR1** signal this signal is trapped and the child will do an orderly exit. When the child is terminated the parent **fslogiced.rb** will notice this and restart the child.

The signal USR1 is reserved for this mechanism.

A short outline is presented here.

```
...
def initialize
  ...
  trap("USR1") {
    #
    # Reseting variables and turning
    # devices off etc
    #
    exit(0)
  }
  ...
end
...
```

4.8 Running control scripts

Normally, all control scripts are started by **fslogiced**. In the FreeSCADA database, control events are connected to the scripts. These control events triggers **fslogiced** to start the control scripts. In order to configure the database one uses the FreeSCADA graphical user interface, **fsgui**.

4.8.1 Running control scripts manually

However, during the development of a control script one might want to start a control script manually. When doing this, command line arguments must be supplied according to the definition below.

```
argv[1] hostname
argv[2] user
argv[3] passwd
argv[4] site
argv[5] script_id
argv[6] event_id
```

```
./your_ruby_control_script.rb localhost fs scada site_db 1 1
```

4.9 Example control scripts

Here follows some real example of control scripts. Further examples are available in the FreeSCADA control script directory `/usr/local/fslogiced` on a FreeSCADA installation.

4.9.1 produce warm water

```

#!/usr/local/bin/ruby
#####
#
#   cv-produce_warmwater-0.1.rb
#
#   This scrip checks for warmwater production criteria and
#   sets requirement variable in the database.
#
#####

#
#   suck in classes needed
#
require 'fslogicd/rucs'

#
#   Initialize base classes
#
initialize_RUCS

#
#   Make an instance of 'ControlScript'
#
class MyControlScript < ControlScript

#
#   This method creates objects we are interested in
#
  def initialize
    @produce_warmwater = DbVar.new('produce_warmwater')
    @production_start_temp = DbVar.new('production_start_temp')
    @production_stop_temp_gt4vv = DbVar.new('production_stop_temp_gt4vv')
    @production_stop_temp_gt2ack3 = DbVar.new('production_stop_temp_gt2ack3')

    @gt2ack2 = DbDev.new('GT2-ACK2')
    @gt2ack3 = DbDev.new('GT2-ACK3')
    @gt4vv = DbDev.new('GT4-VV')

#
#   Here we decide how often this production_check will run
#
    super(method('production_check'), 10)
  end

#
#   This method is the main loop
#
  def production_check

#
#   We collect information from the database
#
    gt4vv_temp = @gt4vv.get('temperature')
    gt2ack2_temp = @gt2ack2.get('temperature')
    gt2ack3_temp = @gt2ack3.get('temperature')

```

```
#
# Depending on the temperature we set production criteria
# in the database
#
      if gt2ack2_temp < @production_start_temp.get or
        gt2ack3_temp < @production_start_temp.get
          @produce_warmwater.set(1)
        elsif gt4vv_temp > @production_stop_temp_gt4vv.get and
          gt2ack3_temp > @production_stop_temp_gt2ack3
          @produce_warmwater.set(0)
        end
      end
    end
  end

#
# This is the actual starting of the program
#
begin
  cs = MyControlScript.new
  cs.loop
end

#
# Exeption handling
#
rescue RuntimeException => e
  DebugAlarm.new(__FILE__+'_'+__LINE__.to_s, "Uncaught exception:\n"+e)
rescue Exception => e
  DebugAlarm.new(__FILE__+'_'+__LINE__.to_s, "Uncaught exception:\n"+e)
end

#####
# END
#####
```


Chapter 5

Additional resources

5.1 Ruby

There is many good books about Ruby. We especially recommends

Programming Ruby, The Pragmatic Programmer's Guide
<http://www.rubycentral.com/book/index.html>

5.2 Electrical installation

5.2.1 Lightning

Lightning is a very mean electrical discharge that eats computers. To protect your computer read on.

<http://www.hvi.uu.se/IFH/askskydd.html>
<http://www.hvi.uu.se/IFH/blixtskydd/blixtskydd.html>
http://www.hvi.uu.se/IFH/grundl_askskydd.html

Index

fsalramd, 4
fscvud, 3
fsguard, 4
fshw-modules, 2
fshwd, 2
fshwmod-ds1820, 3
fshwmod-dummy, 3
fshwmod-fecfc34, 3
fshwmod-ipio, 3
fshwmod-pci6208, 3
fshwmod-pci7250, 3
fshwmod-snmptemp, 3
fslogd, 3
fslogicd, 4
fsmcd, 4

libfshw, 2
libfsrp, 3

MySQL, 2

ruby-fsrp, 3

smalltools, 4

USR1, 37