HartRAO Serial Digital Input Output System

Jonathan F.H. Quick

September 2, 1997

Contents

Chapter 1

Introduction

1.1 Historical Perspective

Currently the Hartebeesthoek Radio Astronomy Observatory (HartRAO) runs its 26m antenna via a monolithic control system based on an HP1000 computer running the real time operating system RTE-6. Configuration control is achieved through a single custom built HP1000-specific Digital Input Output (DIO) unit. The DIO consists of a set of unique wire wrapped input and output boards, each customised to match the interface it originally controlled but with a common programming interface. Physically it is constructed as a single contiguous unit with multiple layers of boards connected by short lengths of ribbon cable. All interfacing to equipment is done through a bank of custom connectors, hardwired into the backplane of the DIO itself. Folklore has it that the DIO operates correctly only when extracted about 1 inch from the rack which houses it - a sure sign of instability and poor electrical connections. Upgrades and extensions are exceedingly difficult because of the customised nature of each interface and the instabilty mentioned above.

In 1993, the original Serial Digital Input Output (*SDIO*) instrumentation bus was developed by Justin Jonas of Rhodes University as a low cost, easily extended distributed digital control interface for a portable radio telescope project. The SDIO concept of remote microprocessor based digital input/output units controlled through a serial protocol proved to be highly successful. The same year NASA ported their Mark IV Field System (FS), used to control the data acquisition equipment for Very Long Baseline Interferometry (VLBI) experiments, to a personal computer architecture. The original SDIO design was then borrowed by HartRAO to add a rudimentary control capability to this new PCFS control system, seen as a prototype replacement for the HP1000.

1.2 New Computer Control System

In the move to a replacement New Computer Control System (NCCS), the SDIO concept underwent several changes, prompted by design criteria based on the perceived failings of the current DIO. The requirements were :

- Distributed Nature - all control functions need to be co-located with the equipment they control ie. not in one central location.
- Extendability - it should be possible to add capability at any point as required.
- Reliability - all units should be interchangeable and upgradeable in-situ without wiring changes.
- Fault Tolerance - there should be multiple independent circuits, each tolerant to the failure of individual units.

- Portability - control should be through a standard protocol likely to be supported by future computer architectures, and capable of being carried by many different transmission media, in particular optic fibre.

The Mark II Serial Digital Input Output system detailed in this document meets all the above criteria and should scale easily to meet all future requirements. Note however the present SDIO communication protocol does not include any error correction. Although operation with the PCFS has shown this is not a problem, this may become a further requirement as the communication links increase in length.

1.3 SDIO Hardware

The SDIO concept revolves around the use of microprocessors to provide remote intelligence. Each SDIO unit consists of a single Intel 80C51 microprocessor, acting through three 8-bit bi-directional buffers to produce 24 bits of digital input/output. Each SDIO unit has a unique 4-bit address (set by DIP switches), allowing up to 16 units to be connected to any one circuit. Communication is through a full-duplex 19.2 kBaud RS422 connection, ie. all 16 SDIO units listen simultaneously on a common command bus and respond via a party-line response bus to those requests specifically addressed to them. A simple RS232 $< -- >$ RS422 converter connects this bus to a single standard RS232-based serial port. The bus may also be extended over long distances using a dedicated pair of optical fibres. Each SDIO unit is then connected to external signal conditioning electronics to drive relays contacts, apply power to receiver electronics, sense physical conditions etc.

1.4 SDIO Software

Ths SDIO software is organised on a circuit by circuit basis. All SDIO units on any one circuit are controlled by a single device server process which owns the corresponding serial port. Each device is handled separately in software with all inter-relations being hidden in the SDIO transport layer sub-objects. The SDIO device server knows how to create and manage all type of SDIO devices, being suitably configured as required through the standard resource database mechanism. See the configuration section of each device in turn for more details.

1.5 Layout of this Document

This document attempts to present the details of the design of the proposed SDIO system, as adapted from the original by Jonathan Quick and Keith Jones of HartRAO. It is divided into two chapters dealing with the hardware and software specifics. Both chapters will need to be updated to reflect additional capability as it is added. Everything is dealt with relatively verbosely in the interests of completeness. Please refer to the individual userguides of the various SDIO-based devices for more concise usage notes.

Chapter 2

SDIO Hardware

The schematic of a typical SDIO hardware setup is presented in figure 2.1. This chapter details the hardware design working from the controlling computer to the hardware being controlled. Each component is dealt with individually in the relevant section below.

2.1 SDIO RS232 $< - >$ RS422 Protocol Converter

The SDIO communication bus, is connected to the controlling computer through a custom isolating RS232 $< - >$ RS422 protocol converter. The control computer TD line is converted from a RS232 12V level to the output of opto-isolated RS422 balanced-pair transmitter permanently in control of the command bus. Similarly the output of an opto-isolated RS422 balanced-pair receiver listening on the response bus is converted to a RS232 12V level on the control computer RD line. All handshaking is ignored (the transmitter and the receiver both operate continuously) - all RS232 signal lines other than TD and RD are open circuit. Note that the converter circuitry is only rated for speeds up to 19200 baud.

Physically each converter unit consists of six independent protocol converters on a common board housed in a single cast aluminium housing, using an external regulated 5V power supply. Internally all six converters also utilise a common isolated 5V supply. RS232 connections are through inverted protocol 10-pin RJ-45 connectors, suitable for direct pin-to-pin connection with the BocaBoard 16-way serial breakout box RJ-45 connectors. All twisted pair connections are through standard plug-in screw-terminal blocks.

2.2 SDIO Full-duplex RS422 Communication Bus

The SDIO communication bus actually consists of two independent directional communication buses :

- **Control bus** - controlled by the transmit line of the control computer serial port through the protocol converter. All SDIO remote interface units only receive on this bus. All commands are transmitted from the control computer to all SDIO units simultaneously, but only enacted by the specific unit to which the command is addressed. Note: this bus actually operates on an inverted RS422 protocol such that the RX led on the SDIO remote interface unit lights up whenever the bus connection from the protocol converter is incomplete ie. one or the other is disconnected from the bus or the converter is not powered on.

- **Response** - controlled by the specific SDIO remote interface unit addressed by the command currently being transmitted on the control bus, All responses are relayed onto the receive line of the control computer serial port by the protocol converter.

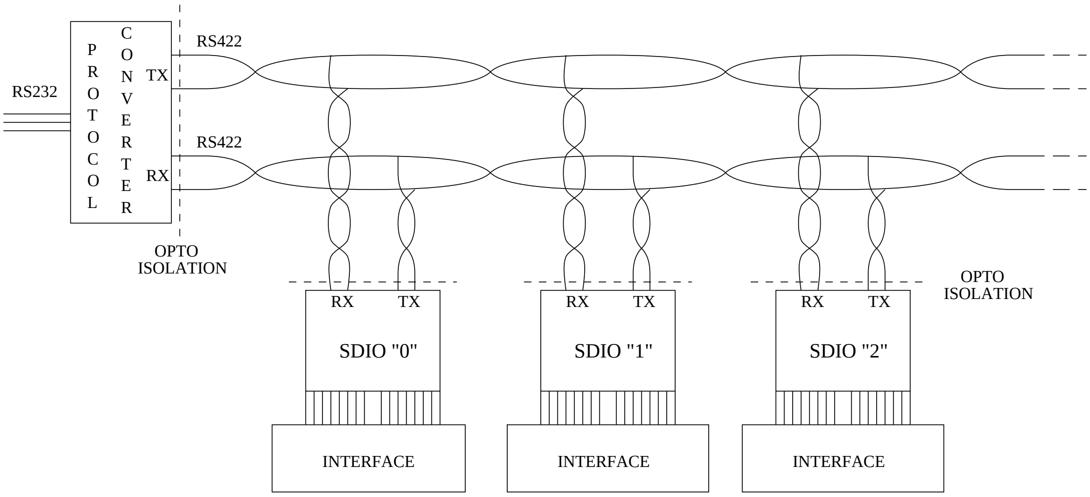

Figure 2.1: Schematic of Serial Digital Input Output (SDIO) Hardware

2.2.1 Optic Fibre $< - >$ RS422 Converters

As part of the development of an optic fibre link capability for use from the main HartRAO control room to the declination room on the antenna, an optic fibre $< - >$ RS422 converter was developed. This allows an RS422 signal to be remotely repeated from one end of an optic fibre to the other. By using these converters to relay the command bus up and the response bus back down, it is thus possible to operate a remote SDIO communication bus on the antenna. Note however that such a remote SDIO bus cannot be linked into a local bus as the RS422 transmitter in the response optic fibre receive unit is always on, in conflict with any locally generated SDIO traffic.

2.3 SDIO Remote Interface Units

Each SDIO remote interface unit basically consists of an Intel 87C51 microprocessor running at 11.032 MHz, all other components handling either buffering or configuration. Functionally the SDIO comprises 24 independent digital input/output lines through three 8-bit bi-directional buffers, together with buffered input-hold and output-valid strobes, shown schematically in Figure 2.2. The direction of each 8-bit buffer can be independently configured by means of hardware jumpers J1, J2 and J3 as follows :

Direction	J1	J2	J3
Port 0 In	.	.	2-3
Port 0 Out	.	.	1-2
Port 1 In	1-2	.	.
Port 1 Out	2-3	.	.
Port 2 In	.	2-3	.
Port 2 Out	.	1-2	.

The remaining jumper J4 can be used alternatively to disable the buffers for use with a daughter board on external feature connector PL5 :

Buffers	J4
Enabled	2-3
Disabled	1-2

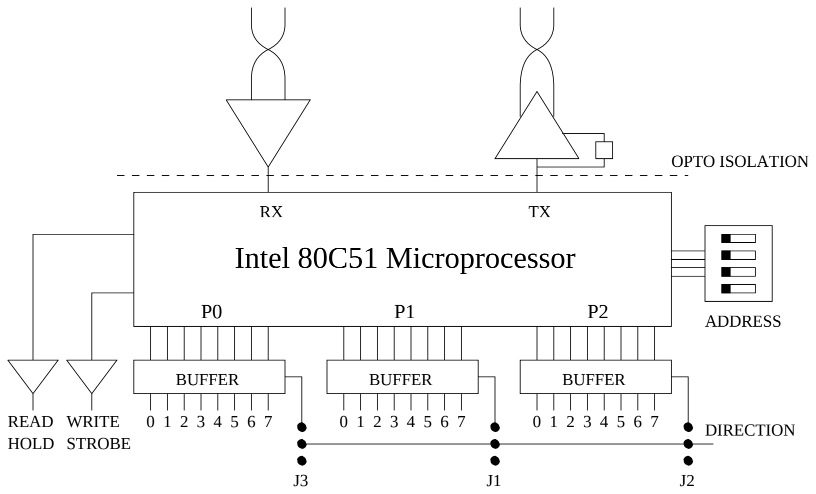

Figure 2.2: Schematic of SDIO Remote Interface Unit Hardware

The 87C51 microprocessor receives serial commands through an opto-isolated RS422 receiver and responds through an opto-isolated RS422 transmitter which is enabled automatically by a transmission activity detector. The communication address of the SDIO remote interface unit (which must be unique to that bus) is assigned using the 4-bit DIP switch package as follows :

Address	1	2	3	4
0	On	On	On	On
1	Off	On	On	On
2	On	Off	On	On
3	Off	Off	On	On
4	On	On	Off	On
5	Off	On	Off	On
6	On	Off	Off	On
7	Off	Off	Off	On
8	On	On	On	Off
9	Off	On	On	Off
10	On	Off	On	Off
11	Off	Off	On	Off
12	On	On	Off	Off
13	Off	On	Off	Off
14	On	Off	Off	Off
15	Off	Off	Off	Off

The current state of each of the 24 digital input/output is reflected in the green leds mounted above the relevant connector pin. Two yellow leds are used to indicate activity on the command (RX) and response (TX) buses. Each SDIO remote interface unit may be powered independently from any available DC power supply in the range 8-28V.

2.4 SDIO Custom Hardware Interfaces

The SDIO remote interface units are standardised to produce/sense only 5V digital levels. In order to control actual equipment hardware, usually some form of custom interface is required. These interfaces will be discussed below in order of increasing complexity.

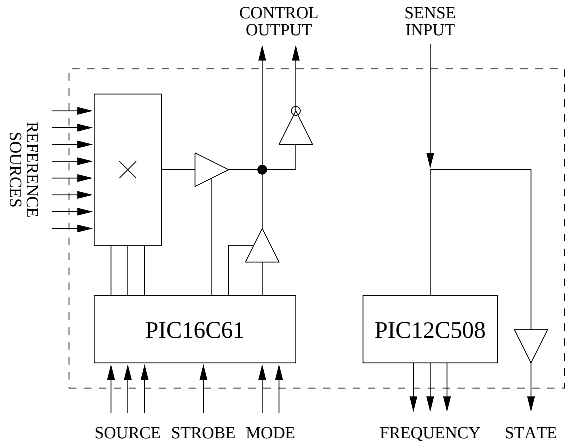

Figure 2.3: Schematic of Noise Diode Controller Hardware

2.4.1 Simple Switches

The simplest interface is the standard transistorised switch board. This interface comes in two distinct flavours, capable either of sourcing or sinking up to 3A at any voltage in the range 5-30V. Each interface board contains eight identical switches which may be driven directly by eight TTL inputs or alternatively through three multiplexed TTL inputs. Each switch also returns a TTL output reflecting its current state as long as the external device is connected - all disconnected switches return ON as a failure mode.

2.4.2 Noise Diode / Dicke Switch Controllers

The Noise Diode Controller (NDC) is one of the most complex interfaces in the SDIO system. It consists of up to eight independent control interfaces, shown schematically in Figure 2.3, multiplexed together and connected to a single SDIO remote unit. The reference switching signals are independently piped right to this remote controller allowing precisely timed modulation of the associated microwave device under asynchronous SDIO control. The NDC produces a TTL control signal (and its complement for Dicke switch control) suitable for use with the transistorised switch above and expects a TTL feedback signal which should reflect the current state of the controlled device as far as possible. The digital interface to the NDC consists of 8 control bits (3 address multiplexing, 2 mode select and 3 modulation source select bits), strobed automatically through the standard SDIO write strobe and 4 feedback bits (1 state and 3 frequency sense bits). The frequency sense bit definitions (updated 4 times a second) are as follows :

Frequency	bit2	bit1	bit0
0-7 Hz	0	0	0
8-15 Hz	0	0	1
16-31 Hz	0	1	0
32-63 Hz	0	1	1
64-127 Hz	1	0	0
128-255 Hz	1	0	1
256-511 Hz	1	1	0
>=512 Hz	1	1	1

Chapter 3

SDIO Software

Due to the party-line nature of the SDIO bus, all traffic on a specific SDIO tree must occur within the same single-threaded device server process. To this end all flavours of SDIO device server are defined to each own and manage one unique SDIO tree, designated by the serial port to which it is attached. The serial port for each individual device server should be designated in the appropriate Class resource file by the resource

```
CLASS/<Server name>/<Personal name>/serial_line
```

defaulting to the resource

```
CLASS/<Server name>/DEFAULT/serial_line
```

or to `/dev/ttyS1` where neither resource has been specified.

The software implementation of the SDIO at the device server level revolves around the concept of implicit internally and explicit externally defined devices. An external SDIO-based device is a device explicitly defined in, created and exported by any one of the device servers detailed below (ie. directly defined in their device lists) and may be accessed directly by external TACO clients. In contrast, an internal SDIO-based device is only implicitly defined and used only internally by client devices within the same device server. Internal devices are not created by the device server itself but must be defined, created and managed by the class level code of the devices that use them. Both types are configured identically via the resource definitions appropriate to the particular device. Note that following the above rules a device cannot be both externally exported and internally used, which significantly simplifies the implementation of single-user access based security.

3.1 SDIO Bus Level Protocol

The 87C51 microprocessors in the remote SDIO units are programmed to communicate at 19200 baud using a standard 8 databits, no parity, 1 stop bit protocol. All bus transactions take the form of simple ASCII character strings transmitted from the master controller to the remote slave units, and echoed by the addressed unit, after suitable modification, back to the controller again. The simple ASCII protocol allows the use of any standard terminal emulator eg. Kermit for debugging purposes. Currently no error detection is employed, all the SDIO remote units merely act on each transmitted character in turn according to the following decision tree:

- If not currently addressed and character is not my address, discard character and continue. Otherwise
- If this character is my address, return lower case version of the character, start new addressed mode (ending any previous addressed mode as with a terminating character below), latch I/O pins into the input buffer (with read hold strobe) and continue. Otherwise

- If this character is a digit in the range `0` to `7`, encode it onto the corresponding I/O pins in the output buffer (see below for more details), return copy of same I/O pins from input buffer as an octal digit and continue. Otherwise
- If this character is the digit `8`, return copy of corresponding I/O pins from input buffer as an octal digit and continue. Otherwise
- Terminate addressed mode by enacting output buffer onto I/O pins and pulsing the write strobe. Discard the character and continue.

The first data digit after the address is mapped as a 3-bit binary value onto bits 0-2 of port 0, with the LSB mapped to bit 0. The following data digit is mapped to bits 3-5 of port 0, the next to bits 6 and 7 plus bit 0 of port 1 etc. Up to 8 data digits are allowed, corresponding to the 24 possible I/O pins. Any further digit is simply treated as a terminator.

Typically the SDIO device server always deals with the entire SDIO remote unit at once such that each bus communication takes the form of a full ten character data frame as follows:

- A single capital letter address in the range `A` through `P` corresponding to remote unit addresses 0 through 15,
- Eight data digits either in the range `0` to `7` for control, or alternatively `8` as a place holder,
- A single terminating `#` (discarded by the remote unit).

The read and write operations are handled completely independently within the device server. Thus the read cycle consists of a `A88888888#` read-only frame, leaving the I/O pin states unchanged. In contrast a write cycle consists of a single write-only frame (eg. `A01234567#`), commanding all 24 I/O pins each time and making use of an internal write-through state buffer to maintain continuity of individual pins.

3.2 SDIO Device Basics - the `Sdio` class

All SDIO-based devices can be abstracted to the level of N-bit digital input/output ports located as some specific point on an SDIO tree. Thus the fundamental SDIO device is the N-bit handle defined by the resources

```
<Domain>/<Family>/<Member>/class   Sdio
<Domain>/<Family>/<Member>/unit    <num>
<Domain>/<Family>/<Member>/offset  <num>
<Domain>/<Family>/<Member>/size    <num>
<Domain>/<Family>/<Member>/dir     <num>
```

which designate a port of `size` pins on unit `unit` starting at pin `offset` with associated direction `dir` (1 for input / 0 for output).

The `Sdio` class layer takes care of all security issues, ensuring that the ports are unique and do not overlap such that each pin is associated with one and only-one `Sdio` handle. A map of the input pin mask is also maintained to ensure that the required configuration does not include conflicts vis-a-vis the required setting of the 8-bit SDIO I/O buffer direction jumpers. The actual jumper setting cannot be directly detected by the SDIO circuitry so the onus is on the user to ensure that the jumpers have been set correctly.

The device server for the `Sdio` class is also the main SDIO device server, supporting all the classes described below and is invoked as

```
Sdio <personal_name> [-m] [-s]
```

and the sample menu program as

```
Sdio_menu <device/under/text>
```

Supported commands are

DevRead	Read the current value on any port,
DevWrite	Set a new value on an output port,
DevReset	Reset any SDIO FAULT condition.

See appropriate userguide and the examples section below for more details.

3.3 SDIO-based Devices

All SDIO-based devices function by defining a set of implicit internal input and output `Sdio` handles by means of which the higher level device is implemented. The class layer of each higher-level device in turn handles the creation, initialisation and control of these `Sdio` handles in the equivalent parts of the device class implementation. Note that none of the classes currently check that the supplied handle definitions actually meet the requirements of the higher-level device. Each device has associated with it a test SDIO device server which is capable of handling only that specific device class and an example menu program for test purposes as detailed below.

3.3.1 Basic Switches - the `Toggle` class

The simplest SDIO-based device is the single-bit toggle switch, which consists of a single bit output port which controls the state of the physical switch and a single bit input port which should reflect the switches current state. The required resource definitions are

```
<Domain>/<Family>/<Member>/class              Toggle
<Domain>/<Family>/<Member>/input_port_name    <some/sdio/handle>
<Domain>/<Family>/<Member>/output_port_name   <some/sdio/handle>
```

This class is designed to use the SDIO with the standard transistorised switch board in straight-through mode to toggle the DC power to a device. The simple test device server for the `Toggle` class is invoked as

```
Toggle <personal_name> [-m] [-s]
```

and the sample menu program as

```
Toggle_menu <device/under/text>
```

Supported commands are

DevOn	Toggle switch to ON position,
DevOff	Toggle switch to OFF position,
DevReset	Reset any switch FAULT condition.

See appropriate userguide and the examples section below for more details.

3.3.2 Multiplexed Selectors - the `Selector` class

A slightly more complex device is the multiplexed selector, which consists of an N-bit output port which sets a multi-position selector to one of up to 2sup N positions, with feedback through a 2sup N bit input port. The required resource definitions are

```
<Domain>/<Family>/<Member>/class              Selector
<Domain>/<Family>/<Member>/start_position     <num>
<Domain>/<Family>/<Member>/no_positions       <num>
<Domain>/<Family>/<Member>/input_port_name    <some/sdio/handle>
<Domain>/<Family>/<Member>/output_port_name   <some/sdio/handle>
```

such that the selector always defaults to the `start_position` on device server startup. Note that `no_positions` need not be an exact power of 2, the multiplexor will simply ignore the state of the remaining unused input port pins.

This class is designed to use the SDIO with the standard transistorised switch board in 3-to-8 mode to drive the six position co-axial IF transfer switches. The simple test device server for the `Selector` class is invoked as

```
Selector <personal_name> [-m] [-s]
```

and the sample menu program as

```
Selector_menu <device/under/text>
```

Supported commands are

DevRead	Read current position of selector,
DevWrite	Set selector to a new position (0 .. `no_positions`-1),
DevReset	Reset any selector FAULT condition.

See appropriate userguide and the examples section below for more details.

3.3.3 Noise Diode / Dicke Switch Controllers - the `Ndmux` class

The Noise Diode Controller (also used for Dicke switches) consists of a 4-bit input port which should reflect the state of the currently selected noise diode (or Dicke switch) and an 8-bit output port which both selects which noise diode is being addressed and controls its state. The required resource definitions are

```
<Domain>/<Family>/<Member>/class              Ndmux
<Domain>/<Family>/<Member>/input_port_name    <some/sdio/handle>
<Domain>/<Family>/<Member>/output_port_name   <some/sdio/handle>
```

This class is designed to control a standard 8-way Noise Diode Controller unit via SDIO. It is not intended to be used directly by users, but rather by the `Ndiode` class as described below. The simple test device server for the `Ndmux` class is invoked as

```
Ndmux <personal_name> [-m] [-s]
```

and the sample menu program as

```
Ndmux_menu <device/under/text>
```

Supported commands are

DevRead	Read current state of any noise diode (or Dicke switch),
DevWrite	Set a new state on any noise diode (or Dicke switch),
DevReset	Reset any Noise Diode Controller FAULT condition.

See appropriate userguide and the examples section below for more details.

3.3.4 Individual Noise Diodes / Dicke Switches - the `Ndiode` class

The `Ndiode` class differs from the other SDIO-based devices in that it does not use SDIO ports directly, but rather through an implicitly defined `Ndmux` Noise Diode Controller. Similar to directly SDIO-based device classes, the `Ndiode` class layer handles the creation, initialisation and control of the underlying `Ndmux` device (which in turn handles those of its `Sdio` handles) in the equivalent parts of the device class implementation, also ensuring that each `Ndiode` device is unique. The required resource definitions are

```
<Domain>/<Family>/<Member>/class     Ndiode
<Domain>/<Family>/<Member>/muxname   <some/ndmux/device>
<Domain>/<Family>/<Member>/offset    <num>
```

for the `offset`'th (0 .. 7) noise diode (or Dicke switch) on the `muxname` Noise Diode Controller.

This class is designed to control a particular noise diode (or Dicke switch) via an 8-way Noise Diode Controller unit and SDIO. The simple test device server for the `Ndiode` class is invoked as

```
Ndiode <personal_name> [-m] [-s]
```

and the sample menu program as

```
Ndiode_menu <device/under/text>
```

Supported commands are

DevOn	Turn noise diode (or Dicke switch) ON,
DevOff	Turn noise diode (or Dicke switch) OFF,
DevNdiodeModulate	Set noise diode (or Dicke switch) to MOD mode,
DevReset	Reset any noise diode (or Dicke switch) FAULT condition.

See appropriate userguide and the examples section below for more details.

3.4 Software Layout

All of the SDIO software is coded in ANSI standard C. The entire source code, complete with documentation is contained in one structured directory sub-tree. A directory tree showing the location of all the software relative to the `dservers` sub-directory of the development area is given below.

DIRECTORY TREE STRUCTURE				
Root	1st level	2nd level	3rd level	Legend
sdio	doc	∗.tex		LaTeX source for this manual
	toggle	bin	linux	`Toggle` class, test and menu programs
		include	∗.h	header files
		res	∗.res	default resource files
		src	∗.c	source files
		ug	∗.tex	LaTeX userguide
	selector	bin	linux	`Selector` class, test and menu programs
		include	∗.h	header files
		res	∗.res	default resource files
		src	∗.c	source files
		ug	∗.tex	LaTeX userguide
	ndmux	bin	linux	`Ndmux` class, test and menu programs
		include	∗.h	header files
		res	∗.res	default resource files
		src	∗.c	source files
		ug	∗.tex	LaTeX userguide
	ndiode	bin	linux	`Ndiode` class, test and menu programs
		include	∗.h	header files
		res	∗.res	default resource files
		src	∗.c	source files
		ug	∗.tex	LaTeX userguide
	sdio	bin	linux	`Sdio` class, menu program and device server
		include	∗.h	header files
		res	∗.res	default resource files
		src	∗.c	source files
		ug	∗.tex	LaTeX userguide

More details on the exact breakdown into individual source files can be found in the README files in the appropriate source directories. All source code, header and most other files are under revision control using the RCS system.

3.5 SDIO Configuration - Examples

In the resource file directories of the SDIO-based device classes described above are a set of default resource files that will be presented here as configuration examples. For each device class there are two files

`<Class_name>Class.res`	Default Class resources
`<Class_name>.res`	Specific Device Server resources

The former merely contains global startup defaults and the assignments of serial ports to specific servers, ie. for the `test` server in this case, invoked as

```
<Class_name> test [-m] [-s]
```

The device specific resource files will be discussed below in terms of the configurations they specify.

3.5.1 Toggle.res

This default resource file describes a very simple configuration, consisting of eight toggle switches commanded by pins 0 through 7 of unit 0 and sensed by pins 8 through 15 of the same unit.

```
#-----------------------------------------------------------------------------
#
# Resource file for toggle server
#
#-----------------------------------------------------------------------------

#-----------------------------------------------------------------------------
#                                Devices list
#-----------------------------------------------------------------------------
Toggle/test/device:     hartrao/toggle/test0 \
                        hartrao/toggle/test1 \
                        hartrao/toggle/test2 \
                        hartrao/toggle/test3 \
                        hartrao/toggle/test4 \
                        hartrao/toggle/test5 \
                        hartrao/toggle/test6 \
                        hartrao/toggle/test7

#-----------------------------------------------------------------------------
#                                Objects Resources
#-----------------------------------------------------------------------------
hartrao/toggle/test0/class:               Toggle
hartrao/toggle/test0/input_port_name:     hartrao/sdio/test0_in
hartrao/toggle/test0/output_port_name:    hartrao/sdio/test0_out
#
hartrao/sdio/test0_in/class:                                Sdio
hartrao/sdio/test0_in/unit:                                 0
hartrao/sdio/test0_in/offset:                               8
hartrao/sdio/test0_in/size:                                 1
hartrao/sdio/test0_in/dir:                                  1
```

```
#
hartrao/sdio/test0_out/class:                           Sdio
hartrao/sdio/test0_out/unit:                            0
hartrao/sdio/test0_out/offset:                          0
hartrao/sdio/test0_out/size:                            1
hartrao/sdio/test0_out/dir:                             0
#
.
.
. <edited to save space>
.
.
#
hartrao/toggle/test7/class:              Toggle
hartrao/toggle/test7/input_port_name:    hartrao/sdio/test7_in
hartrao/toggle/test7/output_port_name:   hartrao/sdio/test7_out
#
hartrao/sdio/test7_in/class:                     Sdio
hartrao/sdio/test7_in/unit:                      0
hartrao/sdio/test7_in/offset:                    15
hartrao/sdio/test7_in/size:                      1
hartrao/sdio/test7_in/dir:                       1
#
hartrao/sdio/test7_out/class:                    Sdio
hartrao/sdio/test7_out/unit:                     0
hartrao/sdio/test7_out/offset:                   7
hartrao/sdio/test7_out/size:                     1
hartrao/sdio/test7_out/dir:                      0
```

Note how only the eight `Toggle` devices are explicitly listed in the `Toggle/test/device` list. All the `Sdio` handles are only defined implicitly by those `Toggle` devices.

3.5.2 `Selector.res`

This default resource file describes a single six-position selector commanded by pins `0` through `2` of unit `0` and sensed by pins `8` through `15` of the same unit.

```
#-------------------------------------------------------------------------
#
# Resource file for selector server
#
#-------------------------------------------------------------------------

#-------------------------------------------------------------------------
#                                Devices list
#-------------------------------------------------------------------------
Selector/test/device:   hartrao/selector/test

#-------------------------------------------------------------------------
#                                Objects Resources
#-------------------------------------------------------------------------
hartrao/selector/test/class:             Selector
hartrao/selector/test/start_position:    0
hartrao/selector/test/no_positions:      6
hartrao/selector/test/input_port_name:   hartrao/sdio/test_in
```

```
hartrao/selector/test/output_port_name:   hartrao/sdio/test_out
#
hartrao/sdio/test_in/class:                       Sdio
hartrao/sdio/test_in/unit:                        0
hartrao/sdio/test_in/offset:                      8
hartrao/sdio/test_in/size:                        8
hartrao/sdio/test_in/dir:                         1
#
hartrao/sdio/test_out/class:                      Sdio
hartrao/sdio/test_out/unit:                       0
hartrao/sdio/test_out/offset:                     0
hartrao/sdio/test_out/size:                       3
hartrao/sdio/test_out/dir:                        0
```

Note that the eight input pins are matched to the three output pins, ie. the underlying selector is 3-to-8 multiplexed, though only the first six positions are in use. (A six pin input port would actually suffice.) Note again the distinction between the explicitly exported `Selector` device and the implicitly defined `Sdio` handles on which it is based.

3.5.3 `Ndmux.res`

This default resource file describes a single Noise Diode Controller unit commanded by pins `0` through `7` of unit `0` and sensed by pins `8` through `11` of the same unit.

```
#-----------------------------------------------------------------------
#
# Resource file for ndmux server
#
#-----------------------------------------------------------------------

#-----------------------------------------------------------------------
#                                 Devices list
#-----------------------------------------------------------------------
Ndmux/test/device:      hartrao/ndmux/testA

#-----------------------------------------------------------------------
#                                 Objects Resources
#-----------------------------------------------------------------------
hartrao/ndmux/testA/class:               Ndmux
hartrao/ndmux/testA/input_port_name:     hartrao/sdio/testA_in
hartrao/ndmux/testA/output_port_name:    hartrao/sdio/testA_out
#
hartrao/sdio/testA_in/class:                      Sdio
hartrao/sdio/testA_in/unit:                       0
hartrao/sdio/testA_in/offset:                     8
hartrao/sdio/testA_in/size:                       4
hartrao/sdio/testA_in/dir:                        1
#
hartrao/sdio/testA_out/class:                     Sdio
hartrao/sdio/testA_out/unit:                      0
hartrao/sdio/testA_out/offset:                    0
hartrao/sdio/testA_out/size:                      8
hartrao/sdio/testA_out/dir:                       0
```

In general operation the `Ndmux` device is usually not exported explicitly as is done here - see the next example.

3.5.4 Ndiode.res

This default resource file describes a typical bank of eight `Ndiode` devices controlled by a single `Ndmux` Noise Diode Controller unit which in turn is controlled by pins `0` through `7` of unit `0` and sensed by pins `7` through `11` of the same unit.

```
#------------------------------------------------------------------------------
#
# Resource file for ndiode server
#
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
#                                 Devices list
#------------------------------------------------------------------------------
Ndiode/test/device:     hartrao/ndiode/testA0 \
                        hartrao/ndiode/testA1 \
                        hartrao/ndiode/testA2 \
                        hartrao/ndiode/testA3 \
                        hartrao/ndiode/testA4 \
                        hartrao/ndiode/testA5 \
                        hartrao/ndiode/testA6 \
                        hartrao/ndiode/testA7

#------------------------------------------------------------------------------
#                                 Objects Resources
#------------------------------------------------------------------------------
hartrao/ndiode/testA0/class:            Ndiode
hartrao/ndiode/testA0/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA0/offset:           0
#
hartrao/ndmux/testA/class:                      Ndmux
hartrao/ndmux/testA/input_port_name:            hartrao/sdio/testA_in
hartrao/ndmux/testA/output_port_name:           hartrao/sdio/testA_out
#
hartrao/sdio/testA_in/class:                            Sdio
hartrao/sdio/testA_in/unit:                             0
hartrao/sdio/testA_in/offset:                           8
hartrao/sdio/testA_in/size:                             4
hartrao/sdio/testA_in/dir:                              1
#
hartrao/sdio/testA_out/class:                           Sdio
hartrao/sdio/testA_out/unit:                            0
hartrao/sdio/testA_out/offset:                          0
hartrao/sdio/testA_out/size:                            8
hartrao/sdio/testA_out/dir:                             0
#
hartrao/ndiode/testA1/class:            Ndiode
hartrao/ndiode/testA1/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA1/offset:           1
#
hartrao/ndiode/testA2/class:            Ndiode
hartrao/ndiode/testA2/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA2/offset:           2
#
```

```
hartrao/ndiode/testA3/class:            Ndiode
hartrao/ndiode/testA3/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA3/offset:           3
#
hartrao/ndiode/testA4/class:            Ndiode
hartrao/ndiode/testA4/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA4/offset:           4
#
hartrao/ndiode/testA5/class:            Ndiode
hartrao/ndiode/testA5/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA5/offset:           5
#
hartrao/ndiode/testA6/class:            Ndiode
hartrao/ndiode/testA6/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA6/offset:           6
#
hartrao/ndiode/testA7/class:            Ndiode
hartrao/ndiode/testA7/muxname:          hartrao/ndmux/testA
hartrao/ndiode/testA7/offset:           7
```

Note here how the same implicit `Ndmux` device is reused with different offsets to produce the eight independent `Ndiode` devices.

3.5.5 Sdio.res

This default resource file describes only the simplest use of this main SDIO device server, namely one output port on pins `0` through `7` of unit `0` and one input port on pins `8` through `15` of the same unit.

```
#-----------------------------------------------------------------------
#
# Resource file for sdio server
#
#-----------------------------------------------------------------------

#-----------------------------------------------------------------------
#                              Devices list
#-----------------------------------------------------------------------
Sdio/test/device:       hartrao/sdio/test0 \
                        hartrao/sdio/test1

#-----------------------------------------------------------------------
#                              Objects Resources
#-----------------------------------------------------------------------
hartrao/sdio/test0/class:               Sdio
hartrao/sdio/test0/unit:                0
hartrao/sdio/test0/offset:              0
hartrao/sdio/test0/size:                8
hartrao/sdio/test0/dir:                 0

hartrao/sdio/test1/class:               Sdio
hartrao/sdio/test1/unit:                0
hartrao/sdio/test1/offset:              8
hartrao/sdio/test1/size:                8
hartrao/sdio/test1/dir:                 1
```

This same server can also be configured to support all SDIO-based devices along the lines of the examples presented above. It must be re-iterated though that no device can be both explicitly exported and implicitly utilised as the two definition processes do not know about each other. Both of the definitions will independently try to create and initialise such a device. The second will always fail as the same `Sdio` handles will already have been allocated to the first.