

# Release 11

## Software for Controlling Industrial Input Output Devices— the IIO Library Revised for Release 11

Open Technical Report CMST-P-97-04

Robin Kirkham  
September 2000



Commonwealth Scientific and Industrial Research Organisation  
Manufacturing Science and Technology  
Preston Site

Corner Albert and Raglan Streets, Preston, Victoria 3072  
Postal Address: Locked Bag No. 9, Preston 3072  
Telephone: (03) 9662 – 7700 Facsimile: (03) 9662 – 7850  
World Wide Web: <http://www.cmst.csiro.au/>



# Summary

This report describes IIO, the Industrial Input Output library.

The term ‘industrial IO’ refers to computer peripherals such as analogue-to-digital and digital-to-analogue converters, lamp, relay and solenoid drivers, or timers, counters and interrupters. The term ‘industrial’ discriminates such devices from the more traditional forms of computer IO, such as graphical displays, disc drives, or serial ports.

The IIO library simplifies the writing of software which uses this industrial IO hardware in practical computer control applications. It comprises a generic ‘core’, presenting a standardised interface to the user software, and an array of ‘module drivers’ which deal with the intricacies of individual hardware modules. It makes user software independent of the particular brand or model of hardware. If the hardware is already supported by the library, IIO frees the programmer from the need to write a module driver, or, if not, provides a framework within which a new module driver can be written.

The library also simplifies the management of such computer-controlled installations. All the configuration information and IO channel assignments for all the hardware in the system is entered into a configuration file. This file is intended to relate closely to the installation’s wiring diagram, assisting programmers and system integrators to agree about the type and naming of all IO devices.

## Structure of the Report

The first part of this report is intended for C programmers wishing to make use of the IIO library in application programs. After an introductory section, the specific concepts and terminology used by IIO are introduced. Then follows a description of IIO’s very simple C language interface. The next section explains how to write IIO configuration files. A general understanding of C programming and industrial IO hardware is assumed for this first part of the report.

The second part assumes a deeper knowledge of real-time programming, device drivers, computer architecture and IO hardware. It is meant for people who want to extend, improve or repair the IIO library. Section 5 is a step-by-step description of how to write and install module drivers for new hardware. The following section deals with a number of specific issues related to module drivers. Finally, the internals (and hopefully, some subtleties) of the IIO core library are exposed.

The report ends with a short concluding assessment, followed by a number of appendices. Most important is Appendix A, which contains the module driver descriptions. The IIO library currently supports over twenty-five such modules. This is followed by a description of the IIO interactive shell, which is useful for testing configuration files and installations. The library installation and maintenance procedures precede listings of important header files and an index.

## Changes to this Report

This report has been changed to conform to Release 11 of the IIO sources. All significant changes to the text since the December 1997 edition are indicated by change-bars in the right-hand margin, as shown. The changes are summarised in Appendix D.

---

## Document History

July 31, 1996	'Alpha' Revision
November 8, 1996	Second Revision
February 5, 1997	Third Revision
September 15, 1997	Fourth Revision
September 25, 1997	Review Draft
December 17, 1997	Release
October 2, 2000	Update for Release 11

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Industrial versus Data Acquisition IO . . . . .	2
1.2	Configuring Software for Hardware . . . . .	2
1.3	Commercial IO Software Libraries . . . . .	3
1.4	The PIRAT CSDB Software . . . . .	4
1.5	The IIO Development . . . . .	4
<b>2</b>	<b>Concepts and Nomenclature</b>	<b>5</b>
2.1	Configuration File . . . . .	5
2.2	Modules . . . . .	5
2.2.1	Model and Module Ident Codes . . . . .	5
2.2.2	Module Drivers . . . . .	6
2.2.3	Module Parameters . . . . .	6
2.3	Channels . . . . .	6
2.3.1	Channel Type . . . . .	7
2.3.2	Channel Names . . . . .	7
2.3.3	Bitwise Digital Channels . . . . .	9
2.3.4	Channel Ranges . . . . .	10
2.3.5	Bitwise-digital Ranges . . . . .	10
2.4	Channel and Module Aliases . . . . .	10
2.5	Why so many channel name options? . . . . .	11
2.6	Operations on Channels . . . . .	12
2.6.1	Operation Codes and Channel Type . . . . .	12
2.6.2	Operation Function Data Types . . . . .	14
2.7	Channel Properties . . . . .	15
2.7.1	‘Real Unit’ Linear Scaling . . . . .	15
2.7.2	Output Value Limiting . . . . .	16
2.7.3	Channel Logging . . . . .	16
<b>3</b>	<b>The IIO Configuration File</b>	<b>17</b>
3.1	Syntax . . . . .	17
3.2	The <code>module</code> Directive . . . . .	17
3.3	The <code>alias</code> Directive . . . . .	18
3.4	The <code>channel</code> Directive . . . . .	18
3.5	An Example Configuration File . . . . .	19
3.6	Writing Configuration Files . . . . .	20
<b>4</b>	<b>Using the IIO Library</b>	<b>21</b>
4.1	Include File . . . . .	21
4.2	Initialisation . . . . .	21
4.3	Opening Channels . . . . .	22
4.4	Channel Operations . . . . .	22
4.4.1	Integer Operations . . . . .	23
4.4.2	Real Operations . . . . .	23
4.4.3	Address Operations . . . . .	23
4.5	Closing Channels . . . . .	24
4.6	Finished using IIO . . . . .	24
4.7	Error Handling . . . . .	24
4.8	Multi-Threaded Applications . . . . .	24
4.9	Customising the Driver List . . . . .	25
4.10	Channel Types and Operations . . . . .	25
4.10.1	Analogue Channels . . . . .	26
4.10.2	Digital Channels . . . . .	26
4.10.3	Address Space Channels . . . . .	27
4.10.4	Encoder Counter Channels . . . . .	27

4.10.5	Servo Controller Channels . . . . .	28
<b>5</b>	<b>Writing Module Drivers</b>	<b>31</b>
5.1	Overview . . . . .	31
5.2	Identification Function . . . . .	32
5.3	Installation Function . . . . .	33
5.3.1	Register Structure . . . . .	33
5.3.2	State Structure . . . . .	34
5.3.3	Decoding Configuration Parameters . . . . .	35
5.3.4	Resolving and Mapping Addresses . . . . .	38
5.3.5	Registering Channels . . . . .	42
5.4	Initialisation Function . . . . .	43
5.5	Operation Function . . . . .	45
5.6	Integrating a Driver into IIO . . . . .	48
<b>6</b>	<b>More on Modules and Module Drivers</b>	<b>51</b>
6.1	Chip Drivers . . . . .	51
6.1.1	When to Write a Chip Driver . . . . .	51
6.1.2	Chip Driver Interface . . . . .	52
6.1.3	Chip Driver Code . . . . .	53
6.1.4	Integrating Chip Drivers into IIO . . . . .	54
6.2	Generic Driver Code . . . . .	54
6.3	Proxy Drivers . . . . .	54
6.4	Endianism and Module Registers . . . . .	55
6.5	ISA Bus Errors . . . . .	56
6.6	Module Drivers for IndustryPacks . . . . .	56
6.7	Module Drivers for Address Spaces . . . . .	57
6.7.1	Invoking Address Space Operations . . . . .	58
6.7.2	Installation Function . . . . .	58
6.7.3	Initialisation Function . . . . .	59
6.7.4	Operation Function . . . . .	59
6.8	Module Drivers for CPU Modules . . . . .	60
6.9	Interrupts . . . . .	60
6.10	Adam Module Channels . . . . .	61
6.10.1	Adam Module Interfaces . . . . .	61
6.10.2	Adam Module Drivers . . . . .	61
6.11	Adding New Channels and Operations . . . . .	62
6.11.1	Adding New Channel Types . . . . .	62
6.11.2	Adding New Operation Codes . . . . .	63
6.12	Errors . . . . .	63
<b>7</b>	<b>Inside the IIO Library</b>	<b>65</b>
7.1	General Practices . . . . .	65
7.1.1	Processes and Systems . . . . .	65
7.1.2	Dynamic Allocation . . . . .	66
7.1.3	Data Structure Conventions . . . . .	66
7.1.4	Coding . . . . .	66
7.1.5	Portability . . . . .	67
7.2	Operational Phases . . . . .	67
7.3	Data Structures and the Initialisation Phase . . . . .	67
7.3.1	State block, <code>IIO_STATE</code> . . . . .	69
7.3.2	Module Information list, <code>IIO_MINFO</code> . . . . .	69
7.3.3	Configuration File Parsing . . . . .	69
7.3.4	Installed Module list, <code>IIO_MODULE</code> . . . . .	70
7.3.5	Memory Map list, <code>IIO_MAP</code> . . . . .	71
7.3.6	Channel Node list, <code>IIO_CHNODE</code> . . . . .	72
7.3.7	Channel Info arrays, <code>IIO_CHINFO</code> . . . . .	73
7.3.8	Alias list, <code>IIO_ALIAS</code> . . . . .	73
7.4	Data Structures and the Open Function . . . . .	73
7.4.1	Open Channel Structure, <code>IIO_OPEN</code> . . . . .	73

7.4.2	Operation Node Structure, IIO_OPNODE	74
7.5	The Operate Function	75
7.6	Operating System Interactions	77
7.6.1	Initialisation	77
7.6.2	Cleanup	78
7.6.3	Mappings	78
7.6.4	Shared and Process Memory	79
7.6.5	Mutual Exclusion Semaphores	80
7.6.6	Register Probes	80
7.6.7	File Interface	81
7.6.8	Serial Device Interface	81
7.6.9	Miscellaneous Functions	81
<b>8</b>	<b>Conclusion</b>	<b>83</b>
8.1	Assessment	83
8.2	Critique	83
8.3	Future Development	85
<b>A</b>	<b>Standard Module Drivers</b>	<b>87</b>
A.1	Module <code>adam4011</code> , <code>adam4012</code> , <code>adam4013</code>	88
A.2	Module <code>adam4017</code> , <code>adam4018</code>	90
A.3	Module <code>adam4520</code>	91
A.4	Module <code>atc10</code> , <code>atc30</code> , <code>atc40</code>	92
A.5	Module <code>bvmipadc</code>	93
A.6	Module <code>dmm32at</code>	94
A.7	Module <code>ipdac</code>	96
A.8	Module <code>ipdigital24</code>	97
A.9	Module <code>ipdualpit</code>	98
A.10	Module <code>ipquadrature</code>	99
A.11	Module <code>ipserial</code>	100
A.12	Module <code>ipservo</code>	101
A.13	Module <code>ipwatchdog</code>	103
A.14	Module <code>isapc</code>	104
A.15	Module <code>mvme162</code> , <code>mvme162lx</code> , <code>mvme167</code>	105
A.16	Module <code>mvme1603</code> , <code>mvme1604</code>	106
A.17	Module <code>null</code>	107
A.18	Module <code>pcgp</code>	108
A.19	Module <code>pcpp</code>	109
A.20	Module <code>phantom</code>	111
A.21	Module <code>tews850</code>	112
A.22	Module <code>vipc610</code>	113
A.23	Module <code>vmivme2532a</code> , <code>vmevme2534</code>	114
A.24	Module <code>vmivme4100</code>	115
<b>B</b>	<b>Using the IIO Interactive Interface</b>	<b>117</b>
B.1	The <code>alias</code> Command	118
B.2	The <code>chnode</code> Command	118
B.3	The <code>map</code> Command	118
B.4	The <code>minfo</code> Command	118
B.5	The <code>module</code> Command	119
B.6	The <code>operate</code> and <code>oinfo</code> Commands	119
B.7	The <code>execute</code> Command	121
B.8	The <code>script</code> Command	121
<b>C</b>	<b>Installing and Maintaining the IIO Library</b>	<b>123</b>
C.1	IIO CVS Archive	123
C.2	Distribution Usage	123
C.3	Platform Script	123
C.4	Distribution Tree	124
C.4.1	Sub-Directories	124

C.4.2	Sources . . . . .	124
C.4.3	Documentation . . . . .	124
C.5	The IIO Makefile . . . . .	125
C.6	An IIO Application Makefile . . . . .	125
<b>D</b>	<b>Changes</b>	<b>127</b>
<b>E</b>	<b>IIO Proposal Document</b>	<b>129</b>
<b>F</b>	<b>Header Files</b>	<b>135</b>
F.1	Header File <code>iiio.h</code> . . . . .	135
F.2	Header File <code>internal.h</code> . . . . .	137
F.3	Header File <code>types.h</code> . . . . .	149
	<b>Index</b>	<b>151</b>



# List of Figures

- 4.1 IIO Digital Channel Types . . . . . 26
- 4.2 IIO Servo Controller Channel Model . . . . . 27
- 4.3 IIO Servo Controller Trajectory Generator . . . . . 28
  
- 5.1 Example VMEbus processor, bus, IndustryPack carrier and module. 38
  
- 7.1 Relationship between Major IIO Data Structures . . . . . 68
  
- C.1 IIO Distribution Tree and Sub-directories . . . . . 124



# List of Tables

2.1	iIO Channel Types . . . . .	7
2.2	Example Relationship between the Four Forms of an iIO Channel Name . . . . .	8
2.3	Example Relationship between Digital and Bitwise-Digital Channels	9
2.4	Channel Operation Codes versus Channel Type . . . . .	13
3.1	Options and Arguments to the <b>channel</b> Directive . . . . .	18
4.1	Flags for Function <b>iio_init()</b> . . . . .	21
4.2	Flags for Function <b>iio_open()</b> . . . . .	22
4.3	iIO Function Return Codes . . . . .	24
5.1	Module Driver Argument Types for <b>iio_arg()</b> and Variants . . . .	36
5.2	Address Space Channel Operation Codes versus Channel Type. . .	40
5.3	Register Width Argument to <b>iio_resolve()</b> . . . . .	42
A.1	ADAM 4000-series Input Range Codes. . . . .	89
A.2	Parallel Port D-25 connector pin assignments . . . . .	110



# Section 1

## Introduction

For a control engineer, there is little point in a real-time computer system that cannot be connected to ‘the real world’ through practical industrial-grade sensors and actuators. A computer with the standard forms of IO (Input Output) devices—such as a keyboard, disc drive and graphical screen—may be useful in the *design* of a control system, but not in its *implementation*.

This is not because of a lack of suitable IO hardware. There is a huge range of industrial-grade interface hardware, which can be used to couple virtually any digital or analogue sensor or actuator onto any common computer system, from an industrial VMEbus rack, to a engineering workstation, to a Linux laptop PC.

The main problem is the operating system, which supports the traditional forms of IO very well, but hardly ever supports this *industrial* IO. It is easy to write software which uses the traditional devices, because the operating system deals with all the details of the device hardware, and presents a neat, virtualised software interface to the user’s program. When it comes to industrial IO, however, users are frequently on their own. They must write software routines (‘drivers’) to directly access the device hardware in whatever way the application requires.

Unfortunately, operating systems such as UNIX actively prevent user software accessing hardware directly, as this is considered a risk to the system’s integrity (which it is). Users are instead expected to write a kernel device driver and install it in the system. This is not always easy, and means users must deal with what is usually the least documented part of the system. They must also choose between implementing the driver as a stream device, like a serial port or network interface, or as a block device, like a disc drive. Sometimes neither is appropriate.

The smaller real-time kernels, such as vxWorks or RTEMS, are somewhat better in this respect. While they usually provide a UNIX-like IO system for traditional devices, they do not force the programmer to use it, and do not obstruct direct access to the hardware. Real-time UNIX systems, such as LynxOS, fit in the middle somewhere, having UNIX-style kernel drivers, but allowing user programs to access hardware, with a little extra work. Still, none of these more industrially-oriented operating systems greatly assist the industrial IO programmer.

This means that the driver code for accessing the IO hardware tends to get included in application programs, where it does not really belong. Ideally, the application programmer should be able to deal with industrial IO devices in a generic way, through a neat, virtualised software interface, just like the traditional devices. The programmer should not know or care what brand of hardware is used to implement the IO, any more than they care what brand of disc drive or serial board is installed.

This practice of writing driver code into application programs also tends to discourage both portability and software re-use. Firstly, there is little impetus to make the driver code portable, because the application may be a one-off, and so certain peculiarities of the operating system may get written into the driver. In turn, the application itself may end up having certain peculiarities of the driver or hardware written into it. In the end the system becomes bound to a certain model of hardware, and little of the software can be re-used in a new context.

If a clear distinction can be made between the application program and the driver software, as occurs with traditional forms of IO, many of these problems will disappear. If the driver software can be amalgamated in the form of a portable library, code re-use becomes simpler. If the programmer’s interface to the library can be standardised, higher-level generic program modules become feasible. These are the principal purposes of the IIO library.

## 1.1 Industrial versus Data Acquisition IO

The term ‘industrial IO’ in the IIO sense includes:

- analogue-to-digital converters (ADCs), which measure voltage, current, temperature or other scalar quantity and supply the result to the computer
- digital-to-analogue converters (DACs), which produce scalar voltage or current on demand from the computer
- digital or binary inputs, such as from relay contacts, switches, proximity detectors, external logic, and so on
- digital or binary outputs, to drive lamps, solenoids, relays, external logic, and so on
- digital counters and timers, such as for incremental encoders, event counters, interrupters, and so on
- motor servo controls.

The term ‘industrial IO’ tends to suggest professional VMEbus hardware and the like, but the type of construction is not important. The style of operation, from both the electronic and the software programming viewpoint, is more significant. Firstly, the hardware tends to be a component in a complete system built for some purpose, and, once installed and configured, is generally not altered frequently. Secondly, the hardware is almost always operated by the software in a ‘sample-on-demand’ mode. In other words, the timing of inputs or outputs is controlled by the application software (using a real-time kernel).

This distinguishes industrial IO from ‘data acquisition IO’. The latter generally consists of a multi-channel selectable-gain ADC coupled with a time-base and sample buffer memory. This kind of hardware is more often used in laboratory or temporary settings, and acquires large sets of samples on a regular (and possibly quite fast) time-base. The software merely starts and stops the hardware, and usually does not process or act on the data, except perhaps to graph or store it after acquisition. This style of hardware (or at least, its operation in such modes) is beyond the current IIO scope.

## 1.2 Configuring Software for Hardware

The driver software must provide a set of function calls that perform actual input and output operations on the industrial IO hardware. Often the relationship between the functions and the low-level register reads and writes is quite simple. This apparent simplicity tends to obscure other issues, particularly those related to configuration, which have more bearing on the re-usability or convenience of the driver and application code.

Most hardware modules have a number of switches of shorting-jumpers which configure various features of the hardware’s operation. Most important is the *base address*, the location in the computer’s address space at which the hardware registers can be read or written. However, other important factors, such as the voltage ranges of ADCs or DACs, are often configured by jumpers.

These configuration details need to be conveyed to the driver software for it to work properly. The most common method is to encode the configuration in `#define` pre-processor directives which are compiled into the driver object code. This means the driver needs to be re-compiled and the application re-linked whenever the hardware configuration changes. It also makes it very difficult to use more than one of a given type of hardware in a system.

A slightly better solution is to add a configuration function to the driver, so that the base address and other details can be supplied to the driver by the application at initialisation time. However, this means that the application program must obtain and deal with hardware-specific data, which compromises its portability.

A complication to both of these approaches is that the base address setting of the hardware is not necessarily going to be the address the program will need to use to access the registers. An address that is written into the driver or supplied through the application will have an implicit assumption about the processor hardware and operating system the application is running on, because a logical or virtual address used in a program is usually subjected to a number of transformations before it emerges as a physical address on the hardware's bus. Often this difference is ignored or simply accepted by the programmers and system integrators.

Another issue is the identification of particular input or output units (*channels*, in IIO parlance). The application program obviously needs to know which channel is connected to which device in the system. Again, `#define` directives are frequently used to assign symbolic names to channels.

Clearly, then, any generic approach to industrial IO will have to deal with issues such as the hardware configuration details, the naming of channels, and the resolution of physical to logical addresses. These are issues that will apply to almost any piece of industrial IO hardware, which suggests a software design comprising a central core of generic interface, addressing and configuration functions, surrounded by an expandable set of hardware module-specific drivers.

## 1.3 Commercial IO Software Libraries

Manufacturers of industrial IO hardware often include sample driver code—often written in assembly language—along with their products. In the author's experience, these routines are sometimes so badly written or so narrow in their applicability that they are useless. Frequently code examples depend on non-standard C compiler peculiarities, or even complete misunderstandings of the language. Usually it is safer to start from the hardware manual and write a driver from scratch.

A few larger manufacturers sell separate software libraries for their products, which are presumably of higher quality. Some of these are intended for execution on a particular operating system. For example, VMIC, a producer of VMEbus hardware, sells IOWorks, which is intended for use with vxWorks, although ports to other operating systems are promised.

IOWorks supports most of all of VMIC hardware range, although support for third-party hardware can be written and integrated. It appears to be a Windows NT application that emits hardware interface code, which is then compiled into applications to run on the real-time system. It also features a Windows NT server application, allowing graphical interfaces to access IO devices on hosts on a network.

Similarly, GreenSpring, originator of the IndustryPack IO mezzanine modules, has released QuickPacks, software libraries that support ranges of IndustryPacks in a reasonably generic manner. For instance, the ADC-oriented library allows the application program to read ADC channels in real units without having to know the particular model of ADC module. This package comes as source, and the user compiles it into the application, and provides the operating system support the packages requires. It is not clear how the IP configuration information is handled.

Both of these solutions are acceptable if the hardware in the system is all of the same brand, but this is rarely the case. An application may end up using a number of these libraries at once, which will at best be untidy and at worst will cause conflict.

## 1.4 The PIRAT CSDB Software

The general issue of IO driver software, and the desire for a more generic solution, tended to emerge each time a new project began. This was particularly true when it came to write the IO hardware drivers, which is a fairly unrewarding activity. Usually, however, deadlines were such that there was no time to consider anything better than a quick fix to get the system running.

Within the PIRAT project, however, the opportunity was taken to test some ideas. The PIRAT Command/Sensor Data Board (CSDB) was the computer interface to a fibre-optic data link from the tele-operated in-sewer vehicle to the surface support van. It provided an array of up- and down-channels, each connected to to a sensor or actuator in the vehicle.

Instead of assigning symbolic names to the channels using `#define`, or using the channel numbers directly in the application programs, the names, types, bit-widths, scale factors and channel numbers were assigned in a configuration file, read during system start-up. Application programs that operated channels first ‘opened’ the channel by name, and were returned a ‘channel descriptor’, in a similar way to opening a file by name and receiving a file descriptor. The descriptor was used for subsequent read or writes to the channel, until the program was finished with the channel, when it closed it.

The configuration file and late name-binding approach was particularly useful. When channel wiring or sensor calibrations or assignments changed, only the configuration file needed to be altered. But further, it allowed the system sensor logger, the safety limit monitor, and the sensor and command displays on the operator interfaces to become almost completely generic. All they required was the list of channel names to be logged, monitored or displayed. This delivered considerable economies of code and conveniences in operation.

## 1.5 The IIO Development

Many of these ideas from the PIRAT CSDB software, and the concerns about module configuration information and address resolution, mentioned in Section 1.2, went into the design of the IIO library.

The opportunity to develop a first version of IIO was provided within the context of the CSIRO DMT Mining Equipment Group. Following a proposal accepted in May 1996, early versions of the library were working by September of that year, and significant improvements were made up until February 1997.

Work concentrated on designing and implementing the framework or structure of the system, rather than supporting a large number of hardware modules. Module drivers were only completed, or partly completed, for modules immediately required in the Dragline Automation project, or for testing purposes.

This report is the sole documentation of the library and that work.



## Section 2

# Concepts and Nomenclature

The IIO system is based on a number of reasonably simple concepts, which are described in this section. This is done independently from descriptions of the C language application program interface to the IIO library, which is the subject of Section 4. Allied with these concepts is a nomenclature, comprising common words with more specific meanings in the IIO context. Defining instances of this nomenclature are introduced in *italics* in this section.

### 2.1 Configuration File

The complete configuration of an industrial IO system is defined by the single IIO *configuration file*. The file tells IIO:

- what hardware *modules* are in the system
- how the hardware modules are *configured*
- the *properties* of individual IIO *channels*.

The configuration file is a plain text file with a simple syntax, described in full in Section 3. It is read by the IIO library as it is initialised after an IIO-using application is started.

An important point is the *all* the configuration information is in the configuration file: there is no other source of configuration details. Furthermore, this configuration is *constant*. IIO-using applications cannot change the configuration while they are running. This is essential so all the running IIO applications have the same ‘view’ of the system, so they can safely share the necessary state data and exclusion locks, and so they will agree about the name and data they read or write from a given channel.

### 2.2 Modules

A *module* in the IIO nomenclature refers to a distinct hardware unit, board, or box. A VMEbus board is a module, as is an IndustryPack (IP), an IP carrier board, a PC motherboard, or a serially-addressed remote data-acquisition unit. In other words, a module is something a computer can use to do industrial IO through.

The use of the term *device* is avoided, because this term is commonly associated with traditional forms of IO hardware, such as serial ports and disk drives, and also their representation in the UNIX operating system.

#### 2.2.1 Model and Module Ident Codes

Modules are represented by *model ident* codes. Each distinct module of hardware has such a code, which uniquely identifies it, at least within the IIO system. A model ident is a short alphanumeric string, which should not contain spaces, punctuation, or any other non-alphanumeric characters. Case is significant but generally lower-case letters are used. The code is usually derived from the manufacturer and/or model number of the hardware.

For instance, `mvme162` is the model ident for a Motorola MVME-162 CPU board, `vmivme2534a` is that for a VMIC VMIVME-2534A digital IO interface board, and `ipdac` is that for a GreenSpring IP-DAC digital-to-analogue IndustryPack module.

Each module installed in a system must be listed in the configuration file, using a **module** directive, its model ident code, and the *module parameters* (see below). IIO assigns a *module sequence number* for each module of a given model as it encounters it in the file, so that that module can be identified later. The sequence number starts from zero and is attached to the model ident code with a period '.' to form the *module ident code*.

Thus, the first module with model ident code **vmivme2534a** will be assigned the module ident **vmivme2534a.0**, the second will be assigned **vmivme2534a.1**, the third **vmivme2534a.2**, and so on, up to the total number of **vmivme2534a** modules in the system. Similarly, the first (or only) **ipdac** module will have module ident **ipdac.0**, the second **ipdac.1**, and so on.

In other words, the model ident code refers to a particular kind of hardware module, while the module ident code identifies a specific instance of that module in the system. The module sequence numbers are assigned in the order that the modules of each model are encountered in the configuration file. In particular, they are not related to the base addresses of the modules, as they often are for, say, UNIX device numbers.

### 2.2.2 Module Drivers

Each module has a *module driver*. The module driver is the piece of software which deals with the intricacies of each individual module, and interfaces it to a standard, internal IIO format. The model ident tells IIO which module driver to use to access the module hardware.

IIO cannot use a module which does not have a module driver, but it is reasonably straightforward to add new drivers to the IIO library (see Section 5). Application writers are encouraged to do so, rather than bypassing IIO if there is no IIO driver.

### 2.2.3 Module Parameters

The *module parameters* appear with the model ident code in the **module** directives of the IIO configuration file. Module parameters tell the IIO module driver how the module has been set up—the base address the board is set to, what configuration jumpers are fitted, and so on.

Some modules permit some of these parameters to be read directly from the hardware (although not usually the base address). IIO however requires them to be specified in the configuration file, partly because IIO module installation must be done without accessing hardware, and partly for consistency. Where the specified parameters can be *checked* against the hardware, the IIO module drivers will do so.

The parameters are specified in the **module** directive using UNIX-command style *options* (with a leading '-') followed by the *argument* values. Some options are mandatory, while most are optional and have appropriate default arguments. The module description pages in Appendix A list the parameters a module can have.

## 2.3 Channels

The word *channel* refers to a single, addressable, input output unit, through which a single scalar value can be read or written on demand (in IIO parlance, with a *channel operation*). A single analogue-to-digital converter is a channel, as is a digital IO port, or a counter, timer, or interrupter.

Each module provides a certain number of channels to IIO. The channels are named, numbered and organised into several structured lists, so that application programs can access the channels in a variety of ways, as described below.

Application programs *open* channels by name, perform *operations* on the open channel to do IO, and then *close* the channel. The scenario is very similar to the way programs use ordinary files: they open them, read or write to the open file, then close it. Section 4 deals with how application programs use IIO.

Channels are also used to identify address spaces, such as an IndustryPack slot or a VMEbus. While application programs will generally not use these channels directly, they are important to module drivers in the IIO *address resolution* mechanism. This type of channel is described in more detail in Section 5.3.4.

### 2.3.1 Channel Type

Channels have a *generic type* and a *specific type*. The generic type indicates what the channel is: an analogue-to-digital converter, a digital IO port, and so on. Each channel type known to IIO has an alphabetical acronym. The current list of channel types appears in Table 2.1. Each channel type has an model of its behaviour, which is described in Section 4.10.

The *specific type* supplements this generic type with the *channel width*—the width of the channel in bits. Thus, a 12-bit analogue-to-digital converter has generic type `adc` and specific type `adc12`: a 32-bit digital output channel has generic type `do` and specific type `do32` (note there is no dot between the type and the width).

The general idea of channel type is to indicate whether channels can be interchanged, at least logically. In principle, one `adc12` is the same as another `adc12`, is similar to a `adc8`, but definitely not the same as a `adc14`. Applications, when they open channels, can choose whether they specifically need a 12-bit ADC (by using a specific type) or any ADC (using a generic type), because the type is encoded in the *channel names*.

### 2.3.2 Channel Names

*Channel names*, in a similar way to module idents, are strings comprising the channel type code and a *channel sequence number*, separated by a period ‘.’. Thus, the analogue-to-digital converters in a system have channel names `adc.0`, `adc.1`, `adc.2` and so on, up to the total number of ADCs present. Analogue-to-digital converters are used here as an example, but the same system applies to all channel types.

Unlike modules, however, each channel actually has at least *four* names, that is, it fits into in four different numbering sequences. The simplest sequence is called the *global generic* sequence, because it is the global (IIO system-wide) sequence of channels with the generic type `adc`. All ADCs of all widths appear in this list.

There is also a sequence for each specific type. Each channel that appears on the generic sequence will also appear on exactly one of the specific sequences.

Type	Description	Type	Description
<code>di</code>	Digital in port	<code>bi</code>	Bitwise digital in port
<code>do</code>	Digital out port	<code>bo</code>	Bitwise digital out port
<code>dio</code>	Digital in/out port	<code>bio</code>	Bitwise digital in/out port
<code>oco</code>	Open-collector output port	<code>ocio</code>	Open-collector in/out port
<code>boco</code>	Bitwise open-collector out port	<code>bocio</code>	Bitwise open-collector in/out port
<code>rdio</code>	Reversible digital in/out port	<code>enc</code>	Incremental encoder
<code>adc</code>	Analogue-to-digital converter	<code>dac</code>	Digital-to-analogue converter
<code>sc</code>	Servo controller	<code>vme</code>	VMEbus address spaces
<code>ip</code>	IndustryPack IP spaces	<code>isa</code>	ISA address space
<code>adam</code>	Adam 4000 serial module address	<code>null</code>	Null device

**Table 2.1** IIO Channel Types

Thus, all the 8-bit ADCs will be in one such sequence, and will have *global specific* names `adc8.0`, `adc8.1`, `adc8.2` and so on, up to total number of 8-bit ADCs in the system. All the 12-bit ADCs will be in the sequence `adc12.0`, `adc12.1`, and so on.

The relationship between the generic and specific sequence numbers of a given channel depends on the particular modules in the system. Different modules will contribute different numbers of different channel types and widths into the global pool of channels. IIO arranges the channels into contiguous sequences in the same order as the contributing modules appear in the configuration file. Thus, `adc12.9` (the tenth 12-bit ADC) might happen to be `adc.35` (the 36<sup>th</sup> ADC in the system).

There are two other sequences a channel will form part of, giving it two further names: the *local generic* name and the *local specific* name. The local sequences contain the channels that are provided by each particular module in the system. The names comprise the module ident, a colon ':', and the generic or specific channel type and sequence number. The sequence numbers start from zero for each module. Thus, the ADCs on (say) the second `xyzz1234` module would have local generic names `xyzz1234:adc.0`, `xyzz1234:adc.1`, `xyzz1234:adc.2` and so on, up to the total number of ADCs on that module. Similarly, there is a sequence for each specific channel type the module provides, with names such as `xyzz1234:adc12.0`, `xyzz1234:adc12.1`, and so on.

The example in Table 2.2 should help to clarify the relationship between the four forms of channel name. Each row in the table represents an ADC channel, and the four columns the four forms of its name. The system, for the sake of argument, comprises two `xyzz1234` modules (providing four 8-bit and two 12-bit ADCs) and one `zog43` module (providing three 10-bit and five 12-bit ADCs). In the configuration file, the `zog43` module appears in between the two `xyzz1234` modules.

The list in Table 2.2 is shown in the order of the global generic sequence number, but it is possible to show it in three other orderings. Again, analogue-to-digital converters are used, but the same system applies to each channel type. However, the simple digital IO channels (types `di`, `do`, `dio`, `oco` and `ocio`) and the bitwise channels (types `bi`, `bo`, `bio`, `boco` and `bocio`) have an additional, special relationship, described next.

Global		Local	
Generic	Specific	Generic	Specific
<code>adc.0</code>	<code>adc8.0</code>	<code>xyzz1234.0:adc.0</code>	<code>xyzz1234.0:adc8.0</code>
<code>adc.1</code>	<code>adc8.1</code>	<code>xyzz1234.0:adc.1</code>	<code>xyzz1234.0:adc8.1</code>
<code>adc.2</code>	<code>adc12.0</code>	<code>xyzz1234.0:adc.2</code>	<code>xyzz1234.0:adc12.0</code>
<code>adc.3</code>	<code>adc12.1</code>	<code>xyzz1234.0:adc.3</code>	<code>xyzz1234.0:adc12.1</code>
<code>adc.4</code>	<code>adc12.2</code>	<code>xyzz1234.0:adc.4</code>	<code>xyzz1234.0:adc12.2</code>
<code>adc.5</code>	<code>adc12.3</code>	<code>xyzz1234.0:adc.5</code>	<code>xyzz1234.0:adc12.3</code>
<code>adc.6</code>	<code>adc10.0</code>	<code>zog43.0:adc.0</code>	<code>zog43.0:adc10.0</code>
<code>adc.7</code>	<code>adc10.1</code>	<code>zog43.0:adc.1</code>	<code>zog43.0:adc10.1</code>
<code>adc.8</code>	<code>adc10.2</code>	<code>zog43.0:adc.2</code>	<code>zog43.0:adc10.2</code>
<code>adc.9</code>	<code>adc12.4</code>	<code>zog43.0:adc.3</code>	<code>zog43.0:adc12.0</code>
<code>adc.10</code>	<code>adc12.5</code>	<code>zog43.0:adc.4</code>	<code>zog43.0:adc12.1</code>
<code>adc.11</code>	<code>adc12.6</code>	<code>zog43.0:adc.5</code>	<code>zog43.0:adc12.2</code>
<code>adc.12</code>	<code>adc12.7</code>	<code>zog43.0:adc.6</code>	<code>zog43.0:adc12.3</code>
<code>adc.13</code>	<code>adc12.8</code>	<code>zog43.0:adc.7</code>	<code>zog43.0:adc12.4</code>
<code>adc.14</code>	<code>adc8.2</code>	<code>xyzz1234.1:adc.0</code>	<code>xyzz1234.1:adc8.0</code>
<code>adc.15</code>	<code>adc8.3</code>	<code>xyzz1234.1:adc.1</code>	<code>xyzz1234.1:adc8.1</code>
<code>adc.16</code>	<code>adc12.9</code>	<code>xyzz1234.1:adc.2</code>	<code>xyzz1234.1:adc12.0</code>
<code>adc.17</code>	<code>adc12.10</code>	<code>xyzz1234.1:adc.3</code>	<code>xyzz1234.1:adc12.1</code>
<code>adc.18</code>	<code>adc12.11</code>	<code>xyzz1234.1:adc.4</code>	<code>xyzz1234.1:adc12.2</code>
<code>adc.19</code>	<code>adc12.12</code>	<code>xyzz1234.1:adc.5</code>	<code>xyzz1234.1:adc12.3</code>

**Table 2.2** Example Relationship between the Four Forms of an IIO Channel Name

### 2.3.3 Bitwise Digital Channels

Industrial IO modules frequently provide digital IO channels. These input or output a scalar binary *number* (as opposed to a voltage or current, as does an analogue-to-digital or digital-to-analogue converter). Sometimes this binary number is significant in its own right, and connects to a sensor or actuator which has a parallel binary interface (such as an absolute position encoder or a seven-segment display).

More often than not, however, individual *bits*, or groups of bits, from a digital IO channel are connected to unrelated sensors or actuators. Applications thus need to conveniently address individual bits, or ranges of them, in digital IO channels, without disturbing any adjacent bits. It is for this purpose IIO provides the bitwise-digital channels, which operate in parallel with the digital channels.

Bitwise channels are always one bit wide. For each real digital channel provided by a module, IIO provides a set of virtual bitwise-digital channels, one for each bit of the underlying digital channel. In effect, all the bits of the digital channels are laid out side-to-side, and each bit in the resultant large array is given its own sequence number. It is very similar to the way all channels of a given type from all modules are laid out side-by-side, and assigned a global generic sequence number.

This is done for the five simple digital channel types: `do.*` channels correspond to the bitwise-digital channels `bo.*`, `di.*` channels to `bi.*`, `dio.*` to `bio.*`, `oco.*` to `boco.*`, and `ocio.*` to `bocio.*`.

Thus, the individual bits of a digital channel `do8.5` might be otherwise opened as eight separate bitwise-digital channels `bo.30`, `bo.31`, `bo.32` and so on up to `bo.37`.

An application program can access the bits through either the digital channel `do8.5` (or any of its other names), or the bitwise-digital channels, or both (although the latter is confusing and is not recommended). When a bitwise-digital channel is altered, the other bits in the underlying digital channel are unaffected.

The same rules pertaining to global and local sequence numbers also apply to the bitwise channels, so applications can refer to bits in the global sequences or local sequences. (There is no effective distinction between generic and specific bitwise-digital channel names, as bitwise-digital channels are always 1-bit wide). Table 2.3 illustrates the relationship between several `dio3` and `dio5` channels and the global and local `bio` channels, from different modules (the sequence continues at both ends).

It could be argued that bitwise-digital channels could be handled by extending the channel name system for digital channels (or indeed generally), rather than

dio3.4	...	bio.13	crel50.0:bio.4
		bio.14	crel50.0:bio.5
		bio.15	crel50.0:bio.6
		bio.16	crel50.0:bio.7
dio3.5		bio.17	crel50.0:bio.8
		bio.18	crel50.0:bio.9
		bio.19	zog200.0:bio.0
		bio.20	zog200.0:bio.1
dio5.0		bio.21	zog200.0:bio.2
		bio.22	zog200.0:bio.3
		bio.23	zog200.0:bio.4
		bio.24	zog200.0:bio.5
dio5.1		bio.25	zog200.0:bio.6
		bio.26	zog200.0:bio.7
		bio.27	zog200.0:bio.8
		bio.28	zog200.0:bio.9
dio5.2		bio.29	zog200.1:bio.0
		bio.30	zog200.1:bio.1
		bio.31	zog200.1:bio.2
		bio.32	zog200.1:bio.3
		bio.33	zog200.1:bio.4
	...		

**Table 2.3** Example Relationship between Digital and Bitwise-Digital Channels

introducing a new, parallel channel type. Perhaps a syntax like `do16.5#4`, with the `#4` indicating bit 4 could be used. However, the bitwise-digital types permitted re-use of much of the IIO library mechanism relating to sequence number assignment and decoding. They also fitted far better into the *channel range* idea described next.

### 2.3.4 Channel Ranges

A very important feature of IIO is the ability to access channels in arrays called *channel ranges* as simply as they are individually. Reading or writing a channel range is exactly the same as doing so on a simple channel, except the data for the read, write, or other operation is in vector, rather than scalar, form.

Generally, the IIO module driver code will repeat the operation for each simple channel in the range. However, some module drivers, such as those for simultaneous-sample ADC and simultaneous-output DAC modules, can perform the operation on all members of a range at once, with attendant benefits.

Channel ranges are indicated using a dash ‘-’ and an extra sequence number in the channel name. Thus, `dac.4-7` refers to a range of four DACs, with global generic sequence numbers 4 through 7. Nonsense ranges such as `dac.7-4` are not permitted; the range `dac.4-4` is equivalent to the simple (single) channel `dac.4`.

Ranges can be used in all the channel name forms. However, they are especially useful with the global forms, because the range may then straddle modules. Thus, with reference to Table 2.2, the range `adc.4-11` refers to ADCs on both `xyzy1234.0` and `zog43.0`. Note that this channel range contains a mixture of 12-bit and 10-bit ADC channels. Alternatively, the range `adc12.4-11` comprises only 12-bit ADCs, now spread over three modules.

It should be noted that even if modules feature simultaneous-sample hardware, IIO cannot guarantee the simultaneous capture across the whole range if it straddles modules, although it will be simultaneous for the sub-ranges within by each module. It does guarantee that the operation is atomic, that is, will be completed for the whole range before any other operation on the range, but only if the same *channel descriptor* (see Section 4.3) is used by the potentially conflicting operators. Atomicity is guaranteed for the sub-ranges contained within modules.

### 2.3.5 Bitwise-digital Ranges

Ranges can also be used on the bitwise-digital channel types, where they refer to bit-fields within or straddling the underlying digital channels. This is a very useful feature for sensors or actuators that require only a few bits, as several can be packed into one digital channel yet accessed separately. It avoids the ‘mask and roll’ bit-twiddling usually used in such cases (in fact, the IIO library does the twiddling).

There is an important difference in the way user data for bitwise-digital channel operations is presented, compared to other channel types. The data is bit-packed into a single unsigned integer, instead of a vector or array. Bit 0 of the data integer corresponds to the lowest-numbered bit of the bitwise-digital channel range. This is a more natural format for bit-fields, but it follows that a bitwise-digital channel range cannot have more bits than an unsigned integer, generally 32.

## 2.4 Channel and Module Aliases

As well as the four standard ways of naming a channel, *channel name aliases* may be defined in the configuration file. An alias is a text string, containing

any non-space characters except a period ‘.’ or a colon ‘:’. Each channel name alias has an *alias value*, which is substituted for the alias whenever it appears as the name of a channel, or part thereof. Aliases are used to give useful, symbolic names to channels, so that the arcane channel name syntax or the installation wiring numbering system does need not be written into application code.

There are actually three kinds of aliases:

- *global aliases*, which stand for complete channel names. The alias value, which may be a global or a local channel name, is simply substituted then interpreted as normal. Thus, a global alias **outside-temp** might be an alias for **adc12.56**: if channel **outside-temp** is opened, it is equivalent to having opened **adc12.56**. Global aliases, which are the most common kind, are created in the configuration file with **channel** or **alias** directives (Sections 3.4 and 3.3).
- *module aliases*, which are aliases for the module ident part of a local channel name, and thus can only appear to the left of the colon ‘:’. Thus, a module alias **upper** can be made for the module **xyzz1234.1**, and channels on that module opened using names like **upper:adc.2**. Module aliases are created in the configuration file using the **module** or the **alias** directives (Sections 3.2 and 3.3).
- *local aliases*, which are aliases for the channel part of a local channel name. These can only appear to the right of the colon ‘:’. If **left-temp** is a local alias for **adc.2**, then that channel on the **xyzz1234.1** can be opened as **xyzz1234.1:left-temp**. Combining it with the module alias described above, the same channel can be accessed as **upper:left-temp**. Local aliases can only be created in the configuration file using the **alias** directive (Section 3.3).

Alias substitution is performed only once *for each type* of alias. In other words, an alias cannot be made for another alias. However, the result from a global alias substitution, if it contains a colon, will be subject to alias lookups for the module and the local channel halves. So, if a global alias **important-temp** expands to **upper:left-temp**, this will further expand to **xyzz1234.1:adc.2**.

Global and local aliases may also stand for channel ranges, as well as for simple channels.

## 2.5 Why so many channel name options?

It can be seen from the preceding sections that IIO features a quite rich channel naming scheme. The need for such a complicated approach might be questioned. It was implemented because of the observation that there are a variety of approaches to wiring up industrial IO systems, each of which is useful in different situations.

For instance, in a laboratory situation, a computer might be wired up to an experimental rig, and a small application program written to operate the rig. The experimenter knows which channel number connects to what, and which module he or she is using, so the program uses local channel names. This minimises changes to the configuration file—other modules in the system can be simply left alone, and the program can be certain it is accessing the correct module.

Alternatively, a generic data-logger might have several input modules connected internally to a set of numbered input sockets (starting from 0, of course). In this case, the programmer will not be interested in which module the inputs are being accessed through, but will be interested in having the channel names corresponding with the numbers on the sockets. Thus, global generic channel names would be used.

In large industrial systems, with many hundreds of channels, it is not uncommon to number IO channels in a similar way, with the numbers following the conductors through the wiring loom to remote IO panels. Again, global generic channel names may be appropriate, particularly if it matches the traditional numbering scheme used in an installation.

However, in most installations it is usually worthwhile to use aliases to give channels symbolic names related to the function of the sensor or actuator the channels connects to. Thus, a channel like `adc.14` can be given the global alias `fishtank-temp`, and `fishtank-temp` can be written into the application program. This makes the program easier to understand, and if the wiring was reorganised and the fish-tank temperature sensor was moved to `adc.17`, only the alias definition in the configuration file need be altered to make the application work again.

Local aliases may be useful when the same ‘mini-installation’ is repeated a number of times, with a 1:1 correspondence between the mini-installations and a module. For instance, perhaps the computer is to monitor many fish-tanks using a custom front-end for an `ipadc` module: there is one `ipadc` and front-end per fish-tank. In the configuration file, local aliases `temp` for `adc.0`, `pressure` for `adc.1`, and `bubbles` for `adc.2` and other quantities could be defined. The application could then open the various quantities using names like `ipadc.2:bubbles`, `ipadc.7:temp`, and so on. Module aliases could also be made for each fish-tank, making the names `tank2:bubbles` or `tank7:temp`. If global aliases were used, rather than local alias combinations, many aliases would have to be defined for all the combinations of module (fish-tank) and input (quantity), to get a similar effect.

More often than not, however, sensors and actuators from various sections and sub-sections of an installation are wired into a set of modules that don’t match the logical structure very well. Global aliases can be used to sort these into a more structured, hierarchical arrangement. Sections, sub-sections and so on down to individual sensors and actuators can be given descriptive names. These names are concatenated with a suitable separator character (such as ‘\_’, ‘-’, ‘!’ or ‘/’) to form the full channel alias, in the same way as a file-name in a hierarchical file-system is prepended with its various parent directory names to form the full file-name.

It is possible even to have several naming schemes at once, as many aliases can refer to a single channel. Thus, a control application on a system might open channels using symbolic names, while another application, perhaps a generic channel monitoring program, might open the same channels using a numeric alias system, or even the real channel names. The actual scheme chosen is completely up to the system integrator.

## 2.6 Operations on Channels

The names by which a channel can be opened for use in an IIO-using application have been described. Having opened and obtained a descriptor for it, what can be *done* with it?

IIO *operations* refer to the simple IO actions a program can perform on an open channel. IIO attempts to carry out the operation as quickly and efficiently as possible, in the context of the calling task or thread. There is no buffering or queueing of operations or data. Operations involve either zero or one data elements per channel, and are either basically *write* operations, in which the data flows out of the system, or *read* operations, where data flows in.

### 2.6.1 Operation Codes and Channel Type

Traditional file and stream IO has several basic functions, `read()`, `write()`, and `ioctl()`. IIO has a single *operation function*, which accepts a channel, the user’s



		Generic Operation Codes															
		io.op.noop	No operation	io.op.read	Read input from the channel	io.op.readback	Read current output from the channel	io.op.write	Write output to channel	io.op.clear	Write zero to channel	io.op.show	Log status of driver/module				
		Address Space Operation Codes															
		io.space.io	Resolve input/output space	io.space.id	Resolve identity space	io.space_int	Resolve interrupt space	io.space_mem	Resolve memory space	io.space_mem16	Resolve 16-bit memory space	io.space_mem24	Resolve 24-bit memory space	io.space_mem32	Resolve 32-bit memory space	io.space.port	Resolve non-mappable port space
		Trajectory Generator Operation Codes															
		io.sc.start	Start servo with new target/settings	io.sc.stop	Stop motion	io.sc.free	Servo to instantaneous position	io.sc.read_target	Read target servo value	io.sc.write_target	Write target position	io.sc.write_target_dt	Write trapezoidal dt (e.g. velocity)	io.sc.write_target_ddt	Write trapezoidal ddt (e.g. acceleration)		
		Servo Controller Operation Codes															
		io.sc.write_current	Write (calibrate) current position	io.sc.read_current	Read current servo value	io.sc.read_index	Read last index value	io.sc.read_gain_p	Read loop proportional gain	io.sc.read_gain_d	Read loop derivative gain	io.sc.read_gain_i	Read loop integral gain	io.sc.write_gain_p	Write loop proportional gain	io.sc.write_gain_d	Write loop derivative gain
		io.sc.write_gain_i	Write loop integral gain	Serial Addressable Operation Codes													
		io.adam_message	Exchange ADAM command/reply packet														
null	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
adam	■	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
isa	■	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
ip	■	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
vme	■	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
sc	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
enc	■	■	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■
rdio	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
bocio	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
boco	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■	■	■
bio	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
bo	■	■	□	■	■	■	■	□	■	■	■	■	■	■	■	■	■
bi	■	■	■	□	□	□	□	□	■	■	■	■	■	■	■	■	■
ocio	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
oco	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■	■	■
dio	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
do	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■	■	■
di	■	■	■	□	□	□	□	□	■	■	■	■	■	■	■	■	■
dac	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■	■	■
adc	■	■	■	□	□	□	□	□	■	■	■	■	■	■	■	■	■

**Table 2.4** Channel Operation Codes versus Channel Type. The symbol ■ indicates the operation should be implemented for the channel type. □ indicates the operation should not be implemented. ☒ indicates the operation is optional. ⊞ indicates the operation may be required in future.

data, and an *operation code*, which tells IIO (or more specifically, the module driver) what to do. This was done because the IIO interface had to cope with a wide range of operations without suffering an endless proliferation of operation functions. As new types of IO hardware is supported, new operation codes can be added as required, without disturbing the form of the programmatic interface.

The codes thus range from generic `iio_op_read` and `iio_op_write` operations through to fairly channel type-specific, like `iio_sc_read_gain_p`. All the current operation codes are listed in Table 2.4. Section 4.10 describes the meaning of the operation codes—although most are fairly obvious—in the context of the models for each channel type.

This table divides the operation codes into various groups, which roughly align with some of the channel types. The table precisely indicates which operations apply to which channel types.

The *generic operation codes* are by far the most commonly used, and apply to most of the channels types. `iio_op_read` reads input data from sampling-style channels, such as analogue-to-digital converters (type `adc`) or digital input latches (`di` or `bi`). `iio_op_write` writes output data to output-style channels, such as digital-to-analogue converters (`dac`) or digital output latches (`do` or `bo`). `iio_op_read` does not apply to these channels, but `iio_op_readback` will return the current *output* value. `iio_op_clear` is equivalent to writing a zero value. `iio_op_nop` and `iio_op_show` are for testing module drivers only.

The *address space operation codes* are used with the address space channels mentioned in Section 2.3 to perform *address mapping and resolution*, the translation of register physical addresses to the logical or virtual addresses needed to actually access them. Section 5.3.4 discusses address mapping and resolution and these operation codes in detail.

The distinction between the *trajectory generator* and *servo controller* operation codes is somewhat arbitrary, particularly as these codes both apply to the servo controller type channel `sc`. These channels attempt to control an output value (such as a position) using a feedback loop, with a reference input derived from a trapezoidal trajectory generator. This type of channel and its operation codes are discussed in detail in Section 4.10.5.

### 2.6.2 Operation Function Data Types

There are three basic forms of the operation function, one for each of the three basic data types that IIO deals in. The actual C-language operation functions a program would call are described in Section 4.4.

**Integer.** The primary data format is a signed integer. This is the same format as the module drivers deal in, and is frequently the data is read or written to the module hardware without alteration. The only transformations the driver will apply to a datum will be:

- zero-shifting for ‘offset 800’-style ADCs and DACs, so that a zero datum always corresponds to zero voltage
- sign-extension to the normal word-width for channel types where this makes sense (currently `adc`, `dac`, `enc`, and `sc`)
- application of the factory calibration correction parameters, where this is feasible.

The only transformations the IIO library will apply will be:

- on write-style operations only, limiting the integer value to the maximum and minimum *channel properties*, described in Section 2.7.2 below.

Integer is the fastest and most precise format, and should be used wherever practical.

**Real.** The ‘real unit’ format is the integer data converted to floating-point and into the actual units of measurement. For instance, an ADC can be read and the result obtained in Volts. The conversion is done by the IIO library, not by the module driver. Output values are also limited as with integer format. Section 2.7 details the *channel properties* that apply to real unit format.

**Address.** The final form of operation function is for memory address data. This form is generally only used for the address space channels, and generally only within module drivers. No transformations or limitations are applied to the data at all.

It is up to the module driver to decide what to do with erroneous data from the application program. In general, these errors are not returned to the user. DACs and similar output devices limit their output to their inherent maximum or minimum, as appropriate, for out-of-range output data, whereas digital outputs and bitwise-digital outputs would tend to mask the data to their own width, and ignore out-of-range bits.

## 2.7 Channel Properties

Aside from having at least four possible names, and being able to be addressed in ranges, and operated upon by application programs, all channels have a set of *channel properties*, which can affect the results of operations.

These properties are initially assigned to the channel by the module driver, and can be subsequently altered by a **channel** directive in the IIO configuration file. The **channel** directive, explained more fully in Section 3.4, can alter the channel properties of a single channel or a channel range.

The current set of channel properties relate to:

- the *linear scaling* between the integer and floating-point user data formats (the channel scale and offset properties)
- the *channel limits*, the minimum and maximum values that can be written to the channel
- *channel logging* properties, in particular whether channel operations are logged, and the real units of the channel.

### 2.7.1 ‘Real Unit’ Linear Scaling

Linear scaling and conversion to or from floating-point format is applied to all values when channels are accessed using the ‘real unit’ forms of the operate function (Section 2.6.2 above, and Section 4.4). The idea is that the application need not worry about the details of the sensors and actuators connected to the channels, as these details can be encoded in the configuration file. Channels can be accessed using values expressed in the real units of the quantity being measured or controlled.

When reading a channel, the integer datum from the driver is converted to floating point, multiplied by the scale value, and the offset value is added. When writing, the inverse is applied: the offset value is subtracted, the result is divided by the scale factor, and the dividend rounded to the nearest integer datum, which goes to the driver. (The driver may also obtain the un-rounded dividend if it requires it, which is rare). The need for the inverse to exist has discouraged implementation of a more general scaling scheme, such as a polynomial, to cope with sensors with non-linear characteristics, such as thermocouples and some ultra-sonic devices.

Usually, the module driver pre-sets the scale and offset values to sensible initial values. For an analogue-to-digital converter, it would set them so that real-unit

operations work out in Volts. The scale and offset values from a **channel** directive in the configuration file are *pre-multiplied* into these initial properties to form new scale and offset properties: they do not replace them.

The purpose of this is to simplify configuring channels for use with sensors or actuators. For instance, in connecting a sensor to a voltage ADC it is only necessary to specify the slope and offset of the sensor in real units per Volt. There is no need to know the Volts per bit of the ADC, since this is already in the initial scale and offset properties supplied by the driver. When choosing sensor and actuator scale and offset properties, always use basic SI units, such as m, kg, A, V, N, °C, radians, and so on.

For channel types where scaling to real units does not usually make sense, such as digital and bitwise-digital channels, scale and offset properties of 1.0 and 0.0 apply respectively. This means there will be no numeric difference in using integer or real operation functions with these channels. However, scale and offset properties can still be attached to these channels in the configuration file if appropriate.

### 2.7.2 Output Value Limiting

Output integer data, whether direct from the application program or computed from real data, is subject to simple limiting before it is used by the driver. There are minimum and maximum channel properties, outside which the output datum is not permitted to go. Limiting does not cause an error.

The limits are only enabled by setting the maximum and minimum channel datum properties in the configuration file. If the minimum exceeds the maximum, the limits are ignored. At present, only module drivers can set the limit properties.

### 2.7.3 Channel Logging

The IIO library will print out a line or two to the standard error stream (or logging stream) for each channel operation, if the channel has the logging property enabled. The logged output shows the name of the channel and the operation code for the overall operation, plus a line for each sub-operation (such as each channel in a channel range, or the components of a bitwise-digital operation). These lines show the full local specific channel name, the data conversions being applied, the integer format data, and the real format data, including units.

## Section 3

# The IIO Configuration File

The IIO configuration file specifies the model and configuration of the input output modules in the system to the IIO library. It allows all the information about the modules and the external system they connect to to be drawn to a single point. Much of this information was described in the previous section.

The file does not apply to any particular IIO application program, but to *all* IIO application programs, or more correctly, systems on which IIO applications can run. While some programs might use certain IO modules and others other modules, all programs read the same file and have the same ‘view’ of the system.

The configuration file is `/etc/iio.conf` on LynxOS systems. On vxWorks systems it is `/vw/iio.conf`, or the configuration can also be read from a static string in memory. This latter method will probably always be the used with RTEMS. On UNIX systems other than LynxOS, IIO does not presently operate system-wide, and the file is `./iio.conf` in the working directory of the application.

### 3.1 Syntax

The configuration file is a plain ASCII line-oriented file, with space-separated tokens. It may be created and edited using a standard text editor. The syntax is as follows:

**Lines.** Lines consist of a series of white-space separated *tokens*. Logical lines can continue over more than one physical line, provided the previous line ends with a backslash character ‘\’.

**Comments.** The hash character ‘#’ introduces comments, can appear anywhere a token can, and also after the line continuation backslash ‘\’. The comment continues to the end of the physical line (i.e., the ‘\’ does not continue a comment).

**Directives.** The first token on a line that is not a hash character ‘#’ should be one of three configuration file *directives*: **module** (Section 3.2), **alias** (Section 3.3) or **channel** (Section 3.4).

**Options.** A token starting with a dash ‘-’ character is an *option*, and the token following it is usually its *argument*.

**Strings.** Spaces can be preserved inside string tokens if the whole token is enclosed in a pair of double-quotes ‘”’.

In other words, the configuration file follows a similar syntax to command languages such as shell-scripts or Tcl. In a sense the configuration file is ‘executed’, as it is parsed from top to bottom, with each directive initiating various actions. An example configuration file appears in Section 3.5. The following sub-sections will describe the three configuration file directives.

### 3.2 The module Directive

The **module** directive *installs* and possibly *initialises* a module. In other words, it tells IIO that there is a module with model ident `<ident>` in the system. The **module** directive syntax is as follows:

```
module <ident> [ -<option> [ <argument> ] ] ...
```

This directive searches for the model ident code `<ident>` in the list of known models, and if located, the module driver code (see Section 2.2.2) is activated to *install* the module, and then *initialise* it.

The *module parameters* (Section 2.2.3) which follow the model ident code comprise *options* and *arguments*. Boolean options (flags) do not have arguments. The module parameters are passed to the module driver, which interprets them as part of the installation process. The parameters indicate the configuration of the module, such as its base address, output range, and other settings. The set of options and default arguments for each module model are given in the module documentation pages in Appendix A.

There are two module directive options that are always available, `-alias <alias>` and `-log`. The former adds a module alias (Section 2.4) into the alias list for the module being installed. The latter makes IIO produce log messages for accesses to all channels on the module (Section 2.7.3).

The module driver *registers* the channels that the module provides. As IIO works through the `module` directives, the list of available channels grows. These channels can be referenced by subsequent `module`, `alias` and `channel` directives in the configuration file, as well as in the application program. Thus, the order of the `module` directives in the file is important.

As well as installing the module in this way, the module driver will initialise the module hardware, if it has not already been initialised. This means output channel values may change.

### 3.3 The alias Directive

This directive adds an alias `<alias>` to the extant channel or module `<extant>`:

```
alias [ -global | -local | -module ] <alias> <extant>
```

The qualifier option `-global`, `-local` or `-module` indicates the type of alias; if omitted, the alias is global. The alias type indicates the context in which the alias will be expanded when referenced. `<extant>` *need not* be defined at the point the alias is made, but it must be by the time the alias is expanded. IIO aliases are explained in Section 2.4.

Note that module aliases are more commonly made using the `-alias` option of the `module` directive, and global channel aliases by the `-alias` option of the `channel` directive. Local aliases can only be made using the `alias` directive.

### 3.4 The channel Directive

This directive alters channel properties (Section 2.7), or adds channel aliases, either for a simple channel, or for a channel range:

Option	Description
<code>-alias &lt;alias&gt;</code>	Add a global alias for <code>&lt;channel&gt;</code>
<code>-scale &lt;scale&gt;</code>	Pre-multiply <code>&lt;scale&gt;</code> into the channel scale factor
<code>-offset &lt;offset&gt;</code>	Add <code>&lt;offset&gt;</code> into the channel offset value
<code>-unit &lt;unit&gt;</code>	Make <code>&lt;unit&gt;</code> the new user unit for the channel
<code>-log</code>	Switch on logging for this channel
<code>-no-log</code>	Switch off logging for this channel

**Table 3.1** Options and Arguments to the `channel` Directive

```
channel <channel> [ -<option> [ <argument> ] ] ...
```

The given channel or channel range *<channel>* must exist. It can be expressed in any of the four channel name forms (Section 2.3.2) or as an alias. The allowable options are shown in Table 3.1.

The *<scale>* and *<offset>* factors specified in the configuration file will in most cases be those of the sensors or actuators connected to the channel. The **channel** directive multiplies these into the scale and offset factors initially installed by the module driver. For ADC, DAC and similar channels, these will be the factors relating the channel's integer value with the sensor or actuator's natural units. Section 2.7.1 describes this in detail. The *-unit* option is used to specify these units, which are used when logging channel operations.

Channel output limits (Section 2.7.2) will also be specified using the **channel** directive, but this is not yet implemented.

## 3.5 An Example Configuration File

Here is an example IIO configuration file, illustrating the use of **module**, **alias** and **channel** directives.

```
# My system's IIO configuration file
# Nibor Mahkrik, 19 August 1997
#
module mvme167
module vmivme2534 -address 0x3600      # another comment
module vipc610 -address 0x6000
module ipdac -slot ip.1 -range 4 -range.2 3 -log
module bvmipadc -slot vipc610.0:ip.0 -alias zog

channel bio.21 -alias fishtank-pump
channel dac.0 -alias fishtank-bubbles \
    -scale 2.243 -unit "bubble/s"
channel zog:adc.0 -scale 0.2 -offset 2.3 -unit C
alias fishtank-temp zog:adc.0

alias -local seconddac dac.1
channel ipdac.0:seconddac -alias fishtank-zap
```

This system uses a Motorola MVME-167 CPU module, which must appear first in the configuration file. The MVME-167, as the module documentation in Appendix A.15 shows, provides one *vme.0* channel, representing the VMEbus.

The next module is a VMIC VMIVME-2534 digital IO module, which provides two *dio16* channels. As Section 2.3.3 explained, they are also accessible as thirty-two *bio* channels. This module plugs in the VMEbus, so it must come after the MVME-167 which provides the VMEbus channel. The *-address 0x3600* option and argument indicate the base address the module has been configured for.

Similarly, another VMEbus module, a GreenSpring VIPC-610 is installed at address 0x6000. This module provides four IndustryPack slots, and so four IndustryPack address space channels, *ip.0-3*. Two of these are used by IndustryPacks.

An GreenSpring IP-DAC is installed in slot B (*ip.1*). Note there is no base address to specify, only the physical location (IndustryPack are slot-addressed). Note that the slot could also have been specified as *vipc610.0:ip.1*. The output voltage ranges of the DACs have been configured differently from the factory default, and the configurations indicated in the module parameters: *-range 4* indicates all the channels are jumpered for range 4 ( $\pm 10$  V) except for channel 2, where the *-range.2 3* parameter indicates it is set to range 3 ( $\pm 5$  V). The

final parameter `-log` indicates IIO will log a message each time a channel on this module is used.

Similarly, a BVM IP-ADC is installed in slot A. This module is given a module alias `zog`. If `zog` is used instead of a module ident (as is done a little later) the real module module ident (`bvmipadc.0`) is substituted.

Some of the channel properties are then configured. One digital bit is used to control a fish-tank pump, so the channel is given the alias `fishtank-pump`. The first DAC channel is controlling the bubbles, using some actuator that produces 2.243 bubbles/s for each volt from the DAC. An ADC channel on the BVM IP-ADC is similarly configured to read a thermometer with a rate of 0.2°C/V and a zero offset of 2.3°C. Note that the `zog` module alias is used: in this case, the global `adc.0` or `adc12.0` would also have done. Then, a global alias `fishtank-temp` is given to the channel, although a `-alias fishtank-temp` on the previous line would have had the same effect.

Finally, a local alias `seconddac` is created, and used to create a further alias `fishtank-zap`, which is really channel `ipdac.0:dac.1`.

## 3.6 Writing Configuration Files

A configuration file for a serious application would be more regular and consistent than the example above. There would be more use of local generic channel names and/or module aliases, since this makes configuration problems less likely, as the module sequence numbers and global channel sequence numbers change when modules are added or removed in the middle. Sections 2.4 and 2.5 discuss different approaches to channel naming in a system.

Writing a configuration file should follow on directly from the system wiring diagram or design. Choose meaningful aliases wherever possible, as the aliases may be used for an interactive channel display or some similar purpose. For the same reasons, always enter the scale and offset factors of the sensor or actuator, even if you expect the application programs to access them using integer operation functions.

The IIO interactive interface, described in Appendix B, is handy for testing configuration files.



# Section 4

## Using the IIO Library

The IIO library has been designed to have a simple C language application interface, which is described in this section. It builds on the concepts and definitions expounded in Section 2. This section also includes detailed descriptions of the channel types and the operation codes that go with them.

### 4.1 Include File

```
#include "iio.h"
```

Programs should always include the file `iio.h`. This file contains function prototypes, enumerated type declarations, and macros necessary to use the library properly. Normally the application `Makefile` will set the compiler search path so this file can be included as shown. A copy of `iio.h` appears in Appendix F.1.

### 4.2 Initialisation

A call to the function `iio_init()` must precede any other calls to the IIO library. This function reads the configuration file, constructs the IIO internal data structures, locates any other IIO-using applications, and, if necessary, initialises the IO hardware. Like almost all IIO functions, it returns a non-zero value if an error was detected, or zero if the call succeeded:

```
if (iio_init(iio_standard, iio_iflag_log)) {  
    fprintf(stderr, "%s: %s\n", argv[0], iio_essage_get());  
    exit(1);  
}
```

Function `iio_init()` accepts two arguments. The first is a pointer to an array of module driver identification function pointers. This allows applications to specify a list of module drivers for IIO to use. Most will specify the ‘standard’ module driver list `iio_standard` (more about this in Section 4.9). The second argument is an OR-ed set of initialisation flags from Table 4.1.

Errors that occur in `iio_init()` mean the application cannot continue, and should call `exit()` like the example above. Errors are generally caused by syntax errors in the configuration file, or more commonly the configuration file not corresponding to the hardware actually installed in the system. The error message obtained from `iio_essage_get()` should be helpful in locating the cause.

Function `iio_init()` should be called only once, and will return an error on the subsequent calls. In protected memory environments, such as LynxOS and UNIX, ‘once’ means once *per application program*. In shared memory environments, such as vxWorks, it means once in total. See Section 4.8 for more information on IIO in multi-threaded situations.

Name	Value	Meaning
<code>iio_iflag_none</code>	0x00	Nothing special
<code>iio_iflag_log</code>	0x01	Log certain major events

**Table 4.1** Flags for Function `iio_init()`

## 4.3 Opening Channels

Channels and channel ranges are represented by objects of type `IIO`, referred to as a channel descriptor. (This object is actually a pointer to an internal `IIO` data structure). Channels are opened using the `iio_open()` function:

```
IIO channel;
...
if (iio_open("dio.23", iio_oflags_none, &channel)) {
    fprintf(stderr, "%s: %s\n", argv[0], iio_emessage_get());
    exit(1);
}
```

Opening a channel is analogous to opening a file, although the call syntax is a little different to `fopen()`. The first argument is the character string *name* of the channel (see Section 2.3.2). This can be in any of the four standard forms (Section 2.3.2), or name aliases may be used (Section 2.4).

The second argument is an OR-ed set of open flags from Table 4.2. The logging flag controls logging of the channel. Operations on the channel are logged if any of the following are true:

- the `module` directive in the configuration file installing the module that provides channel had a `-log` flag
- a `channel` directive in the configuration file referring to the channel had a `-log` flag
- the channel was opened with the `iio_oflag_log` flag.

The third argument to `iio_open()` is the address of the user's channel descriptor variable. The descriptor value is only written if the open is successful. Like a file descriptor, the channel descriptor is used for subsequent operations on the channel.

The same channel can be opened more than once: the two (or more) channel descriptors are not identical but are interchangeable. The same channel may also be opened by another program or process in the system. Section 4.8 discusses multi-threaded issues further.

## 4.4 Channel Operations

There are six channel operation functions, which perform the channel operations described in Section 2.6 on channels. In other words, they actually perform IO through the channel or channel range represented by the channel descriptor. The only difference between the functions is the format in which the user's data is accepted (in the case of output) or produced (in the case of input). There are three such formats, with each format having two operation functions. Section 2.6.2 describes the data formats in detail.

Table 2.4 on page 13 shows the matrix of operation codes and the channel types that implement them.

Name	Value	Meaning
<code>iio_oflag_none</code>	0x00	Nothing special
<code>iio_oflag_log</code>	0x01	Log channel operations

**Table 4.2** Flags for Function `iio_open()`

### 4.4.1 Integer Operations

The basic data type is an `int`. This is because most IO hardware uses a binary representation and so data can be transferred to and from an `int` quickly and without loss of precision.

```
extern IIO_STATUS
    iio_operate(IIO channel, IIO_OP operation, int data[]),
    iio_operate_in(IIO channel, IIO_OP operation, int data);
```

Function `iio_operate()` accepts a channel descriptor, an operation code (from Table 2.4), and a pointer to integer user data.

The user data array *must* be the correct size for the channel. For a simple channel, a single `int` will do; for channel arrays, there must be one element in the array for each channel in the range. There is no way for the IIO library to tell if the data array is the correct size, and only the easily overridden C-language type system to ensure the type is correct.

There is one important exception: if the channel is a bitwise-digital range, the data is bit-packed into a single integer. In other words, bit 0 of the data corresponds to the first channel in the range, bit 1 to the second, and so on. The size of bitwise-digital ranges is presently limited to the number of bits in an `int`.

For output operations on simple channels, there is a convenience function `iio_operate_in()` (the ‘in’ refers to the direction of data into the function, not the direction of the channel). Instead of a data pointer, a single integer argument is passed directly. This is useful for writing constant values to channels, for instance:

```
/* switch on the bubbles */
if (iio_operate_in(bubbles, iio_op_write, 1)) {
    ...
}
```

In both functions the output value is subject to limiting, as described in Section 2.7.2. Input values are never limited.

### 4.4.2 Real Operations

Real form is integer form scaled and offset by floating-point factors supplied by the module driver and the configuration file, as described in Section 2.7.1. Normally these factors convert the integral value to and from one in real units, such as volts, amperes or degrees. As with the integer form, the output values are limited.

```
extern IIO_STATUS
    iio_operate_real(IIO channel, IIO_OP operation, double data[]),
    iio_operate_inreal(IIO channel, IIO_OP operation, double data);
```

Function `iio_operate_real()` is identical to its integer counterpart, except the data the data pointer points to must be of type `double`. Again, it must point to an array of at least the same number of elements as the number of channels in the channel range. `iio_operate_inreal()` is for simple output channels channels, and accepts a single `double` argument.

### 4.4.3 Address Operations

Address space channels and operations (Section 5.3.4) should be invoked using the functions `iio_operate_addr()` or `iio_operate_inaddr()`. These two forms are identical to their integer and real counterparts, except that no processing at all is performed on the data, which should be of address (`void*`) type.

```
extern IIO_STATUS
    iio_operate_addr(IIO channel, IIO_OP operation, void *addr[]),
    iio_operate_inaddr(IIO channel, IIO_OP operation, void *addr);
```

## 4.5 Closing Channels

```
iio_close(channel);
```

Channels are closed using `iio_close()`. Generally, channels are opened, used for the life of the application, and closed when the application cleans up prior to exiting. Alternatively, applications may choose to open, operate, and close channels as required. This is not efficient (`iio_open()` is slow compared to the operation functions) but may be justified where channel use is sparse. Closing a channel does not affect its output value.

## 4.6 Finished using IIO

```
iio_done();
```

When the application program has finished using IIO, it should call `iio_done()`. This closes any remaining channels, dismantles the IIO data structures and releases resources, such as virtual memory maps. UNIX applications should use the `atexit` mechanism, or catch fatal signals to make sure this function is called, as some operating systems (including LynxOS) *do not* automatically release these resources on process exit. Calling `iio_done()` will not affect the output value of any channels.

## 4.7 Error Handling

As mentioned, IIO functions generally return a zero or non-zero error indication. More specifically, they return a value of enumerated type `IIO_STATUS` (Table 4.3).

Fatal errors indicate an internal inconsistency within the IIO library, which should not in theory occur, or exhaustion of a vital system resource, such as virtual memory.

In the examples on the previous pages, the function `iio_essage_get()` is called. This function returns a pointer to a static error string, indicating the source of the error. If the error originated in the operating system, a pointer to the operating system's error message is returned instead.

Within the IIO library, this status-return mechanism is used throughout, aided by macros for calling functions and returning on error status (`iio_eret()` and `iio_fret()`). These macros allow for stack-collapse traces to be printed for fatal errors, which helps pin-point bugs in the library. Refer to Section 6.12 for further discussion of these macros.

## 4.8 Multi-Threaded Applications

IIO is thread-safe. Each module and each open channel has a mutex semaphore associated with it which is taken whenever the object is accessed. Where available, these mutexes have priority inversion protection enabled. IIO and its module drivers create no threads or tasks, and all processing and hardware accesses occur in the context of the calling thread.

Name	Value	Meaning
<code>iio_status_ok</code>	0	No error
<code>iio_status_error</code>	-1	A recoverable error occurred
<code>iio_status_fatal</code>	-2	A serious internal error in IIO was detected

**Table 4.3** IIO Function Return Codes

On multi-processing (protected memory) systems, the module mutexes operate both process-wide and system-wide, by placing the mutex in a shared-memory segment. Module state information is also stored in the shared segment. This means all IIO-using threads and processes in the system will agree about the datum of a channel, and simultaneous access will not corrupt the shared state.

Open channel descriptors may be shared freely between threads within a process (or tasks in a system, in the case of a shared-memory multi-tasking system). Channels can be opened by one thread, used by another, and closed by a third. The same channel can be safely opened multiple times. Channel descriptors *cannot* however be shared between processes (such as through a shared-memory segment). If two processes both need to access the one channel, they need to independently open the channel.

Generally, an IIO-using application will initialise IIO (using `iio_init()`) before going multi-threaded. Any IIO-using threads in the application should be deleted or caused to exit before `iio_done()` is called. `iio_done()` does not need to be called in the same context as `iio_init()` was.

## 4.9 Customising the Driver List

The list of module drivers which IIO can use is stored in an array of pointers to module identification functions. A pointer to the array must be passed to `iio_init()`. Generally, a built-in list of all drivers is specified, `iio_standard`.

This means that executables will be linked with every driver, whether they will need it or not. By and large, this does not matter, but it may unnecessarily increase executable size and linking time.

On occasions when it is necessary to thin down the number of drivers, a separate driver list can be specified instead of `iio_standard`. This is usually put in a separate file within the application sources. The file should resemble this example:

```
#include "iio.h"

extern IIO_STATUS iio_atc40(void);
extern IIO_STATUS iio_isapc(void);
extern IIO_STATUS iio_bvmipadc(void);

IIO_INFOFN myDriverList[] = {
    iio_atc40,
    iio_isapc,
    iio_bvmipadc,
    NULL
};
```

Note that the list is terminated with a `NULL`. Order is not important, although alphabetical listing is customary. When the application starts IIO, it would supply `myDriverList` instead of `iio_standard` to `iio_init()`. Obviously, the configuration file can then only contain module directives for `atc40`, `isapc` or `bvmipadc`.

## 4.10 Channel Types and Operations

Each channel type has a functional ‘model’ which defines what the channel can do and what operations do it. All IIO module drivers which provide that channel should conform to it, so that applications do not need to be written around a particular model of hardware. These informal ‘models’ are described here, for the channel types application programs will typically use.

### 4.10.1 Analogue Channels

Analogue-to-digital converters (channel type `adc`) sample a continuous scalar quantity, such as voltage, current, temperature, pressure, and so on, in response to a `iio_op_read` operation. The module driver returns a two's complement sign-extended integer datum, where:

- zero input corresponds to zero datum, or minimum input corresponds to zero datum if the input range does not include zero
- one unit change in datum should correspond to the smallest detectable input change
- numerically increasing input means increasing datum, and vice-versa
- input higher or lower than the input range should be represented by the highest or lowest datum respectively.

Most ADC hardware already satisfies these requirements.

The module driver should initialise the channel properties so that the units of real data operations are correct. These units will be Volts or sometimes Amperes, except where the ADC is *permanently* connected to a sensor of some kind (such as for temperature), where the properties should correctly reflect the sensor.

Digital-to-analogue converters create a quantised approximation of a continuous scalar quantity, essentially the reverse of an ADC. The `iio_op_write` operation outputs new data, while `iio_op_readback` returns the current output. The output datum interpretation should therefore be the same as outlined above, except reversed. Identical rules regarding channel properties, including the output limits, also apply.

### 4.10.2 Digital Channels

Digital channels are groupings of individual bits, each of which can be only on or off. The channel input or output is the binary representation of the *unsigned* integer datum. There are five types of simple digital channels, `do`, `di`, `dio`, `oco` and `ocio`, and the more complicated `rdio` channel. Figure 4.1 shows the electronic circuits typical of the bits in the digital channel types.

- `di` Digital in channels read a 1 when the terminal voltage is 'high', that is, higher than some threshold voltage, and 0 otherwise.
- `do` Digital out channels drive the terminal 'high', or above a threshold, when a 1 is written to them, otherwise they drive it low. `do` channels correspond to TTL 'totem-pole' outputs, or other circuits that act this way.
- `dio` Digital in/out channels are a `do` and a `di` channel connected to the same terminal. The reading from the `di` channel should always correspond to the setting of the `do` channel, if the channel is functioning correctly. This channel type is not common. *In previous IIO releases, `ocio` channels were known as `dio` channels.*
- `oco` Open-collector digital out channels drive the channel low when a 1 is written to them (assuming NPN transistor drivers). *This is the reverse of the*

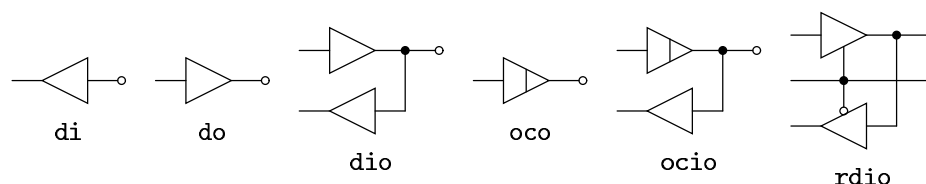


Figure 4.1 IIO Digital Channel Types

*do channel.* `oco` channels frequently operate current loads (lamps, relays, solenoids) wired between positive supply and the terminal, so a 1 corresponds to the on or energised state.

`ocio` Open-collector digital in/out channels are an `oco` channel connected to a `di` channel. This can serve two purposes:

- the input channel can be used to monitor the output channel, possibly detecting transistor failure, load disconnection or loss of supply voltage
- the output channel can be left off (by writing a 0 to it) and the input channel used as a normal `di` channel.

Because of this flexibility, `ocio` channels are widely used. *In previous IIO releases, `ocio` channels were known as `dio` channels.*

`rdio` Reversible digital in/out channels are driven by tri-state totem-pole drivers. They can be regarded as more like a data bus than a data port, because input and output data may only be at the terminals fleetingly, and additional control signals (strokes and direction indicators) are needed to effect data transfer.

Data for digital channels is subject to the channel scale and offset properties, if a real-unit operation function is used (by default, these are 1.0 and 0.0). The output data is subjected to the channel limits, if any, and is then masked to the width of the channel or bit-range. The output datum can be read back from the output channels using `iio_op_readback`.

The bit-arrays formed from concatenating all channels of the simple digital types can be accessed using the parallel `bo`, `bi`, `bio`, `bcoo` and `bocio` channels, as described in Section 2.3.3. `rdio` channels are not concatenated.

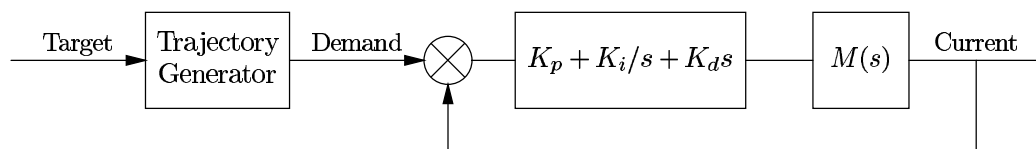
Bitwise-digital channels and channels ranges also have channel properties. These are not very useful for simple bitwise channels, but they may be for bitwise-digital ranges. The channel properties pertaining the the first (zeroth) bit in the range are used for the scaling and limiting of the data.

### 4.10.3 Address Space Channels

Address space channels represent addressing hardware, such as a VMEbus, an ISA bus, an IndustryPack slot or an ADAM serial-addressable unit address. These channels and their operations are not generally directly used by application programs. They are described fully in the context of address resolution in module drivers (Section 5.3.4), module drivers for address space channels (Section 6.7), and ADAM module drivers (Section 6.10).

### 4.10.4 Encoder Counter Channels

Incremental encoders produce quadrature signals which are decoded by encoder interface hardware into up or down count pulses. These accumulate into counters, represented by channels of type `enc`. The datum represents the relative rotational or linear position of the encoder shaft.



**Figure 4.2** IIO Servo Controller Channel Model

Operations `iio_op_read` and `iio_sc_read_current` return the current encoder position. Operations `iio_op_write` and `iio_sc_write_current` set the position (or *calibrate*) the encoder. Further movement of the encoder will be relative to the new datum.

Encoder counters are also usually equipped with an index register, which captures a copy of the main counter when the encoder passes its index position. This index datum is read using `iio_sc_read_index`. The index datum will not be consistent with the current datum if the current datum has been written and the index position has not been passed again.

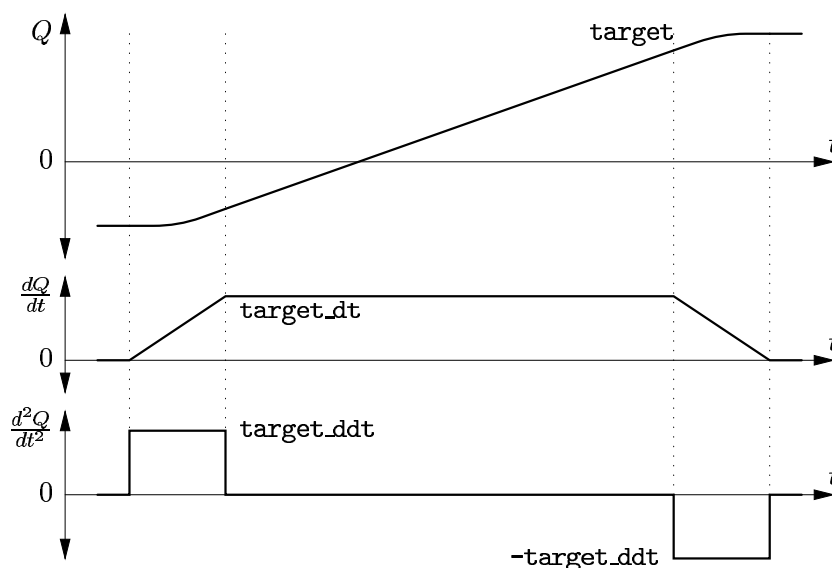
#### 4.10.5 Servo Controller Channels

The servo controller module is much more complicated than other IIO channel modules. The controller attempts to keep the *current* servo value (usually, but not limited to, a position) as close as possible to an *command* value, using a traditional PID control loop, or equivalent. The command value is generated by a trapezoidal (or higher-order) trajectory generator, based on a *target* data and rate limits from the application program. Figure 4.2 shows the model. Many of the motion parameters will be constant, and will be specified in the configuration file.

There are two kinds of operation codes: those that read or write motion parameters and current data, and those that start or stop motion.

The PID servo loop characteristics are set using `iio_sc_write_gain_p` for the proportional gain  $K_p$ , `iio_sc_write_gain_d` for the derivative gain  $K_d$ , and `iio_sc_write_gain_i` for the integral gain  $K_i$ . There are three matching operations for reading back these data. The value and units of these quantities will depend on the design of the servo controller, particularly its interface with the actuator and any intervening servo amplifiers: these effects are lumped in  $M(s)$  in Figure 4.2. New data does not take effect until an `iio_sc_start` operation is issued.

The current servo datum can be read using `iio_sc_read_current` or more simply `iio_op_read`. The current datum can be set using `iio_sc_write_current`, which calibrates the feedback element, usually an incremental encoder. Operations `iio_sc_read_current`, `iio_sc_write_current` and `iio_sc_read_index` should behave identically to the incremental encoder channel type, `enc`, described in Section 4.10.4.



**Figure 4.3** IIO Servo Controller Trajectory Generator



The trajectory generator is driven by the target datum, which is changed using the operation codes `iio_sc_write_target` or `iio_op_write`. The target datum can be read back using `iio_sc_read_target` or `iio_op_readback`.

Apart from the target datum, the trajectory is controlled by two rate parameters, set using `iio_sc_write_target_dt` and `iio_sc_write_target_ddt`. These are shown in Figure 4.3. The first is the *maximum* change of the target per second. (If the target is a position, then it refers to the maximum velocity). The second is the maximum rate of change of the first (or the maximum acceleration). Both these parameters must be greater than zero, or no motion can result.

A *motion segment* is initiated by setting the target data, and then issuing a `iio_sc_start` operation. This operation promulgates the motion and feedback parameters, and makes the servo move, if it is not moving already. The servo will move until the target datum is reached: the time the segment will take is a function of the initial difference between the current and target data and the maximum rates of change. The target, or any of the other motion parameters can be changed at any time, and in any order: they do not take effect until a further `iio_sc_start`.

Motion can be stopped by using the `iio_sc_stop` operation, which replaces the target datum with the current datum at the instant the operation was issued. The servo can be disabled using `iio_sc_free`, which makes it servo to the *instantaneous* current position.



## Section 5

# Writing Module Drivers

This section is intended for people who want to add support for new hardware to the IIO library. It describes the structure of a module driver's code, the things it must do, the things it can optionally do, and how to integrate it into IIO.

### 5.1 Overview

A module driver is the interface between the core part of the IIO library and the module hardware itself. There is one module driver for each distinct hardware model, or variants on a basic model.

The driver must conform to the quite rigid format described here. This is because the IIO core performs a number of services for the module drivers, such as managing data structures and providing mutual exclusion locks, in order to minimise the amount of code in the drivers. This only works, however, when the driver performs as the core expects.

The driver must restrict itself to calling only the IIO functions described in this section, and *not* use any system calls or C library functions. This is all in the name of module driver portability: as soon as a programmer uses an operating system call in a driver in IIO, the whole library becomes potentially non-portable.

The driver must handle *all* the resources provided by a module (or, at least, all resources that are going to be made available to the IIO user). Two drivers may not share access to a module. This is because the mutual exclusion mechanism assumes an exact correspondance between a module driver, the driver's state, and its hardware.

Furthermore, access to the module cannot be shared with a non-IIO driver, except in cases where there is no possibility of conflict. The same applies to direct access to the module by the user program outside the IIO structure. Cases such as this can be handled by using IIO *proxy drivers*, described in Section 6.3.

While they are generally self-contained, drivers may share code. An example is where a *chip driver* is used by the module driver. Chip drivers are written to support specific components, usually peripheral chips, that appear, or might appear, in otherwise unrelated modules. Section 6.1 explains chip drivers in detail.

The driver must contain at least four specific functions:

- The *identification function*, which is called by IIO core *before* the configuration file is read, and even if the module is not installed in a given system. The function registers various details about the driver with IIO by *calling back* the library (a style of operation that is used throughout the module driver interface).
- The *installation function*, which is called if the module is specified in the configuration file with a `module` directive. The function parses the module parameters, initialises the module data structures, and registers the channels the module is going to provide to the IIO core. It does *not* access the module hardware itself.
- The *initialisation function*, which is called directly after the installation function *only* if the IIO code knows the module hardware has not been initialised. The function must perform the initialisation and also initialise the module state structure, if there is one.

- The *operation function*, which is called when the user calls `iio_operate()` or one of its variants. The function must obtain the user's data and perform the desired operation. There may be more than one operation function, usually when a module has channels of more than one type.

The first three functions are called by IIO as a result of the user program calling `iio_init()`. The operation function is called when the user calls `iio_operate()` or a variant. (There is no driver function which corresponds to `iio_open()`).

There are also two data structures defined by the module driver:

- The *register structure*, which is established by the installation function. It contains pointers to the module hardware registers, and any configuration information digested from the configuration parameters. Once established, it should not be changed, and in particular it should *not* contain any module state.
- The *state structure*, which is established by the initialisation function, *does* contain module state, and is shared amongst all IIO processes and tasks using IIO. It is protected by the same mutual exclusion mechanism as the module itself. It typically contains shadows of write-only register values. The state structure is optional, as simple modules may not require it.

The following four sub-sections (5.2 to 5.5) describe each of the four functions and the two data structures, what they should do, and hopefully why they should do it. Each sub-section also describes the IIO core functions that should or can be used by the driver at each stage.

Section 5.6 deals with more practical aspects of writing a driver module and integrating it into the IIO library. Section 6.1 describes how to write and integrate chip drivers and other generic driver components.

## 5.2 Identification Function

The module identification function is the only public symbol the driver must provide. The function, which is called by IIO once only during the initialisation phase, must register certain details about the module driver with the IIO library, by calling the function `iio_minfo()`, as in this example:

```
IIO_STATUS iio_xyzzy1234(void) {
    return iio_minfo(
        "xyzzy1234",
        "XYZCom XYZZY-1234 Thingy Interface",
        "$Revision: 1.6 $",
        iio_multi_yes,
        iio_xyzzy1234_install,
        iio_xyzzy1234_init
    );
}
```

The first argument is the model ident for this module, as described in Section 2.2.1. It is a short string containing letters and numbers which uniquely identifies the module; usually, the module's model number is used. The model ident is the name used to identify the module in the configuration file.

The second argument is a fuller description of the module; this should include the name of the manufacturer (or a common abbreviation), the full model number, and a quick description of the module. The next argument is the version of the module driver software; normally the CVS/RCS revision number keyword is used, as shown.

The `iio_multi_yes` argument indicates the module driver may be installed more than once, if there is more than one such module in the system. This will

be the case with most modules, but modules with fixed bus addresses, or modules such as CPU boards, will typically specify `iio_multi_no`.

The last two arguments are pointers to the installation and initialisation functions, which are described in the next two sections. As mentioned, the identification function is the only public symbol the module driver must provide; all the other functions are called through pointers passed back to the IIO core. This is purely to simplify the process of patching in new module drivers.

The identification function can call `iio_minfo()` more than once, in order to register the module driver with different model ids. This permits a module driver to support a number of similar but distinct models. For instance, the `xyzy1234` might be very similar to the `xyzy1235`, such that the same driver can handle both. Normally, the same installation and initialisation functions are specified for all variants: it is possible for the installation function to establish which variant was specified in the configuration file later. If the differences between the variants turn out to be any more than minor, you should probably be writing separate module drivers for them.

## 5.3 Installation Function

The installation function is called indirectly by the IIO core when the module's model id is given in the configuration file in a `module` directive. It should conform to the prototype

```
IIO_STATUS iio_xyzy1234_install(IIO_MODULE *module, char *argv[]) {  
    ...  
}
```

The `module` argument is a pointer to the IIO data structure that represents an installed module. This pointer should be passed to the IIO core functions called by the installation functions, but not otherwise used. `argv` is the tokenised list of parameters from the module directive in the configuration file. There is no `argc`: the list is terminated by a NULL pointer.

The installation function is frequently the largest function in the module driver. In short, it must perform all the initialisation that can be done *without* actually accessing the module hardware. The function is expected to:

- allocate the register structure (Section 5.3.1)
- allocate the state structure (Section 5.3.2)
- destructively parse the module parameters (Section 5.3.3)
- map the module and resolve the register addresses (Section 5.3.4)
- register the channels the module provides (Section 5.3.5)

Should anything go wrong in the installation function—or any of the others—an error status, with accompanying static error message string, should be returned, using the `iio_error()` or `iio_fatal()` macros. The error string is ultimately returned to the application program. The installation function will typically only generate error returns for bad or missing configuration parameters.

Similarly, an error status should be returned immediately should any of the IIO functions the driver calls fail. An easy way to do this is to enclose these calls in an `iio_eret()` macro, as is shown in the examples. Section 6.12 describes the error handling system further.

### 5.3.1 Register Structure

The installation function should first allocate a module-specific register structure. This structure, always of type `IIO_MREG`, should be defined at the top of the module driver code. It contains pointers to each of the module's hardware registers,

plus any of the configuration details decoded from the arguments that may be required later. A pointer to the register structure is passed to the subsequent calls to the initialisation and operation functions.

For our example `xyzzzy1234` module, the register structure might be defined like this:

```
struct IIO_MREG {
    volatile iio_uint8_t *csr;    /* control/status register */
    volatile iio_uint16_t *dr[4]; /* four data registers */

    IIO_BOOL invertflag;          /* data inversion flag */
    int range;                    /* output range setting */
};
```

The first element is a pointer to the module's 8-bit control register, and the second is an array of pointers to the module's four 16-bit data registers. Always use *pointers to each individual register* and choose the pointer type to match the width of the register. Never attempt to lay a C structure or array over the device registers, as is sometimes suggested. While a little neater, the results of this practice are compiler and architecture dependent, and therefore not portable: the pointer approach is. (For similar reasons, C bit-field structures also cannot be used: this is a great shame).

Note also the use of the `volatile` declaration qualifier. This is essential to force the compiler to actually access the register when the program demands, and not optimise away accesses that it thinks are redundant. It is also necessary to force the compiler to access the register with the specified bus cycle width.

The actual values of the pointers (i.e., the register addresses) are determined from the configuration parameters, parsed using the argument decoding functions in Section 5.3.3, and the address mapping/resolution functions in Section 5.3.4. Section 6.4 deals further with register addressing issues.

The register structure can also contain flags or variables which indicate configuration of the module to the initialisation and operation functions. These values for these are also decoded from the configuration parameters, using the argument decoding functions. In this example, there is a flag that indicates that output stages are to be inverted (or perhaps that inverting output stages are fitted to the board), and a number that indicates an output range setting. Only constant configuration information that is actually required by the initialisation and operation functions should go in the register structure.

The structure itself must be allocated using a call to `iio_module_reg()`, which accepts the `module` argument passed to the installation function, the size of the register structure, and a pointer to the register structure pointer:

```
IIO_MREG *reg;
...
iio_erec( iio_module_reg(module, sizeof(IIO_MREG), &reg) );
```

The new structure can then be initialised using `reg`.

### 5.3.2 State Structure

The installation function must also allocate the module-specific state structure. This structure, always of type `IIO_MSTATE`, is also defined at the top of the module code. It contains device state and configuration details that cannot be decoded from the configuration file, or read back from the hardware itself. The state structure is described further in Section 5.4.

The structure is allocated by calling `iio_module_state()`:

```
iio_erec( iio_module_state(module, sizeof(IIO_MSTATE)) );
```

This structure is shared amongst all processes or tasks using IIO, so that all will agree about the state of shared hardware. As with the module hardware, the installation function should not access or initialise the state structure: this is done later by the initialisation function.

To enforce this, the state structure pointer is not returned to the caller (the register structure pointer is); instead, it is passed to the initialisation function. Some modules will not require a state structure, as the module state can be read directly from the hardware registers as required. In this case, there is no need to define `IIO_MSTATE` and `iio_module_state()` should not be called at all.

### 5.3.3 Decoding Configuration Parameters

In the IIO configuration file (Section 3, the module driver configuration parameters are specified using a UNIX shell command-line style, with *options* (indicated by a leading `-` character) introducing each *argument*. For instance, the hypothetical XYZZY-1234 module might appear in the configuration file like this:

```
module xyzzy1234 -address 0x4600 -range 23 -no-invert
```

The IIO core simply tokenises the line into a NULL-terminated string-pointer array, which it passes when it calls the module installation function. `argv[2]` will point to the string "xyzzy1234", `argv[3]` to "-address", `argv[4]` to "0x4600", and so on. Pointer `argv[8]` (the one following "-no-invert") will be NULL, indicating the end of the list. `argv[0]` contains a file-name/line-number reference of the line in the configuration file, which is supplied by the IIO core to assist generating an error message if the function returns an error status.

**Argument Decoding Functions.** The installation function should use the `iio_arg()` function and its variants to decode the parameters. These functions scan the `argv[]` list, looking for selected options. If found, they remove the option and argument from the list (replacing them with empty strings) and return the argument value (using the argument type to check its syntax). The process is repeated for each possible option.

This approach is slightly different from traditional way of parameter list parsing, where each parameter is compared in turn to a list of possible options. There are several advantages in the IIO context:

- it reduces the amount of code in the module drivers
- a partially decoded argument list can be passed to the installation function of a chip driver (Section 6.1) for further decoding
- it standardises the style of the configuration file.

The IIO core checks the argument list after the initialisation function has returned: any parameters remaining (i.e., not cleared by the argument parsing functions) must be illegal options, which are reported to the user.

All of the argument parsing functions accept the parameter list, option names, argument types, and argument value pointers. For instance, our example module might have options for the module base address and for output inversion:

```
iio_uint16_t base = 0x5000;
...
reg->invertflag = iio_bool_false;
iio_eret( iio_arg(argv, "address", iio_arg_uint16_t, &base) );
iio_eret( iio_arg(argv, "invert", iio_arg_bool, &reg->invertflag) );
```

Here, the base address is a 16-bit number. `iio_arg()` will search `argv[]` for a "-address" option, and attempt to convert the following token into a 16-bit number. If successful, the number will be written into `base`; otherwise, there is

probably a syntax error in the configuration file, so `iio_arg()` returns an error. If the `iio_eret()` macro is used as shown, the installation function will also return an error, and an appropriate message will be printed.

If there is no `"-address"` option in the parameter list, `iio_arg()` does not return an error, but also does not alter the value of the `base`. Its value will remain the pre-set `0x5000`.

Boolean options (type `iio_arg_bool`), such as `"-invert"` are unique in that function `iio_arg()` and variants do not look for a following argument. Instead, the result variable is set to a non-zero value (`iio_bool_true`) if the option appears anywhere in the parameter list, or is cleared to zero (`iio_bool_false`) if its negation `"-no-invert"` appears. If neither `"-invert"` or `"-no-invert"` is found, the value is unchanged.

**Default and Mandatory Arguments.** If any option does not appear in the argument list, its result variable is *not changed at all*. It is important that all such variables are pre-set to sensible default values, as in the example above. These values should reflect the factory configuration of a module.

Some module options will be mandatory, and have no sensible default. In this case, the argument variable should be pre-set to a nonsense value, and the value checked after the call to `iio_arg()`. For instance, a `-slot` option for IndustryPack module drivers is always mandatory:

```
IIO slot = NULL;
...
iio_eret( iio_arg(argv, "slot", iio_arg_channel, &slot) );
if (! slot) {
    iio_log("%s: IP slot not specified", argv[0]);
    return iio_error;
}
```

**Argument Types.** The full set of argument types that `iio_arg()` and its variants will search for is shown in Table 5.1. The programmer must ensure that the argument result pointer points to an object of the correct type. There is no way the compiler can enforce this.

Most of the argument types are self-explanatory. For integer arguments, `iio_arg()` accepts either decimal, octal and hexadecimal parameters in the conventional form, and always properly checks their syntax. The floating-point forms are similarly checked.

Type	Description	Result type
<code>iio_arg_bool</code>	boolean	<code>IIO_BOOL</code>
<code>iio_arg_int8</code>	8-bit signed integer	<code>iio_int8_t</code>
<code>iio_arg_int16</code>	16-bit signed integer	<code>iio_int16_t</code>
<code>iio_arg_int32</code>	32-bit signed integer	<code>iio_int32_t</code>
<code>iio_arg_int64</code>	64-bit signed integer	<code>iio_int64_t</code>
<code>iio_arg_uint8</code>	8-bit unsigned integer	<code>iio_uint8_t</code>
<code>iio_arg_uint16</code>	16-bit unsigned integer	<code>iio_uint16_t</code>
<code>iio_arg_uint32</code>	32-bit unsigned integer	<code>iio_uint32_t</code>
<code>iio_arg_uint64</code>	64-bit unsigned integer	<code>iio_uint64_t</code>
<code>iio_arg_float</code>	floating point	<code>float</code>
<code>iio_arg_double</code>	double floating point	<code>double</code>
<code>iio_arg_addr</code>	address	<code>void *</code>
<code>iio_arg_string</code>	dynamic allocated string	<code>char *</code>
<code>iio_arg_channel</code>	IIO channel descriptor	<code>IIO</code>
<code>iio_arg_file</code>	file-descriptor	<code>IIO_FILE</code>

**Table 5.1** Module Driver Argument Types for `iio_arg()` and Variants



As well as decoding simple types, `iio_arg()` accepts string, file and channel arguments. String arguments are dynamically duplicated, and a pointer returned into the result. As described in Section 3, strings that contain white-space should be enclosed in double-quotes `'(".")'`.

File type arguments expect a file-name. `iio_arg()` opens the file in read-write-create mode, and returns a *file descriptor* (type `IIO_FILE`) of the open file, or an error if the file could not be opened. The module driver should close the file when it has finished with it. Few module drivers will need to open files, but if they do, they should do it this way, as file-names are likely to be system-dependent and so should be in the configuration file.

Similarly, IIO channels can be opened, and an IIO channel descriptor is returned (`iio_arg()` simply calls `iio_open()`). This feature is frequently used, because IIO channels that represent *address spaces*, as opposed to input-output channels, are central to the IIO module mapping and register address resolution system, described in the following section.

**Argument Parsing Function Variants.** There are also a few variants to `iio_arg()`, which are convenient when there are many options, or when options for individual channel numbers are needed.

Function `iio_arg_list()` can be used to process a list of arguments in one call, and is equivalent to a sequence of calls to `iio_arg()`. The function accepts the parameter list pointer, then any number of triplets of option name, argument type, and result pointer arguments, terminated by a `NULL`. For instance, the example could have used:

```
iio_eret(
    iio_arg_list(
        argv,
        "address", iio_arg_uint16_t, &base,
        "invert", iio_arg_bool, &reg->invertflag,
        NULL
    )
);
```

Where the same option needs to be individually specified for a number of individual channels, such as the gains to be used on a set of ADC channels, the function `iio_arg_index()` can be used. This is the same as `iio_arg()`, but accepts an additional index argument, and searches for options of the form `-gain.<index>`, instead of just `-gain`. For example, the code

```
for (index = 0; index < NCHAN; ++index) {
    iio_eret(
        iio_arg_index(
            argv, "gain", index, iio_arg_uint8, &reg->gain[index]
        )
    );
}
```

will search for the arguments `-gain.0`, `-gain.1`, `-gain.2` and so on, up to `NCHAN`.

These two variants are combined by function `iio_arg_index_list()`, which is equivalent to `iio_arg_list()` but includes the index argument, as in the following example:

```
for (index = 0; index < NCHAN; ++index) {
    iio_eret(
        iio_arg_index(
            argv, index,
            "range", iio_arg_uint8, &reg->range[index],
            "gain", iio_arg_uint8, &reg->gain[index],
        )
    );
}
```

```

        "invert", iio_arg_bool, &reg->invert[index],
        0
    )
};

```

This will look for `-range.0`, `-gain.0` and `-invert.0`, then `-range.1`, `-gain.1`, `-invert.1`, and so on.

### 5.3.4 Resolving and Mapping Addresses

After decoding its arguments and possibly storing them (or something derived from them) in the register structure, the next thing the installation function must do is *map* the full address space occupied by the module, and then *resolve* the addresses of the device registers, that is, compute them and put them in the register pointers in the register structure. In other words, it must

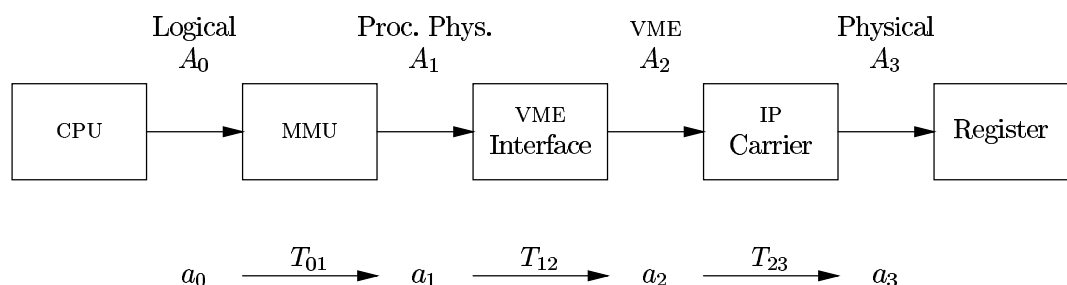
- make the module hardware registers ‘visible’ to the program (mapping), and
- find out the logical addresses of the registers (resolution).

Address mapping and resolution is a more complicated issue in the IIO system than for normal IO drivers, because the drivers must be portable, and because IIO must run in protected (i.e., virtual) address environments like UNIX user processes. ‘Standard’ or ‘well known’ mappings between logical and physical address spaces must never be written into module drivers, because such mappings will almost certainly not be the same on all systems. Instead, the IIO address mapping and resolution mechanism, described below, must be used.

**The IIO Address Mapping and Resolution System.** The need for address mapping, and, more particularly, address resolution, can be understood by considering the transformations applied to an address as it makes its way from the CPU, through the computer system and its sometimes numerous busses, to actually accessing a particular module register.

In Figure 5.1, which represents a typical VMEbus processor board accessing a register on a IndustryPack module, a program running on the CPU references an address  $a_0$ . Address  $a_0$  is known as a *logical address* (or a virtual address) because the address is relative to the program’s address space  $A_0$ . It is the address (or, if preferred, pointer value) the *program* must use to actually access the register. As it travels through the computer system’s hardware and busses, the address is subject to transformation  $T_{01}$  through the memory management unit (MMU), and  $T_{12}$  through the bus interface unit, in becoming a VMEbus A16 address  $a_2$ . From the VMEbus there is a further transformation  $T_{23}$  to the register space of the given IP slot,  $A_3$ , where  $a_3$  is the register’s *physical address*.

These address transformations are usually simple arithmetic or linear functions related to the base addresses each of the hardware modules.  $T_{01}$  (the MMU) may be more complex, and is usually controlled by the operating system.



**Figure 5.1** Example VMEbus processor, bus, IndustryPack carrier and module.

As well, the number of these translations involved will vary from situation to situation. Clearly, these transformations cannot—and should not—be encoded into the module driver code, or it will not be portable.

Instead, IIO module drivers only encode the physical addresses of registers with respect to the appropriate address space—in other words,  $a_3$  with respect to  $A_3$ . Thus, the address of a register on an IndustryPack is encoded as an address in an IP register space, not as a VMEbus address or anything else. Drivers use the IIO address mapping and resolution functions to determine *at run-time* what logical address  $a_0$  they should use in order to access the physical address  $a_3$ .

Address mapping, performed by the IIO core function `iio_map()`, is called by drivers to ensure that the transformations  $T_{01}$ ,  $T_{12}$ ,  $T_{23}$  and so on actually exist (especially  $T_{01}$ , which usually requires operating system calls to achieve). Address resolution, performed by `iio_resolve()`, is the evaluation of the overall inverse mapping for a given register  $(T_{23}T_{12}T_{01})^{-1}a_3$  to obtain the corresponding logical address  $a_0$ .

**Address Space Channels.** While the IIO core deals with the address translations through the MMU ( $T_{01}$ ) through the operating system, the user effectively defines the relationship between the other address spaces using the IIO configuration file, by indicating which modules plug into which. This is done, implicitly or explicitly, using *address space channels*.

Address space channels represent addressing devices, like busses, slots, and addressable serial networks. For example, the channel `vme.0` represents a system's VMEbus, while the channel `ip.5` represents the sixth IndustryPack slot in a system. While address space channels represent addressing devices, rather than ordinary IO channels, they are named, numbered, opened, operated upon and closed like any other channel. However, they have their own sub-set of operations, called *address space operations*, which are described shortly.

Every module is 'on', that is, plugged into, exactly one of these address space channels. Often this channel is implicit, and not usually specified in the configuration file, such as with VMEbus modules (most systems have only one VMEbus, `vme.0`, so this is a sensible default for VMEbus module drivers). Other modules require this channel to be specified, such as those for IndustryPacks, because the channel effectively identifies the physical slot the module is plugged into.

Modules that allow other modules to be accessed through them, such as bus adaptors, IP carriers and CPU modules, *provide* the address space channels the other modules are on, in the same way as ordinary modules provide IO channels. For instance, a PC CPU module always provides an address space channel `isa.0`, because it can access other modules through its ISA bus. A four-slot Industry-Pack carrier module would provide four `ip` address space channels, since four IP modules can be accessed through it.

Thus, as it reads the configuration file, the IIO library builds up a hierarchy of address spaces and modules that link them. This is the reason the order of modules in the configuration file is important, as mentioned in Section 3: modules that are on a particular address space must come after the module that provides that space.

**Address Space Operations.** Address space channels generally respond only to a subset of IIO operation codes called *address space operation codes*. Table 5.2 lists the current address space operation codes and shows which address space channel types implement them. (This information also appears in Table 2.4 on page 13).

Each code represents a sub-space of an address space channel, generally related to the width of the address word. For instance, the VMEbus has three basic non-overlapping address spaces, A16, A24 and A32, which are selected by

the operations `iio_space_mem16`, `iio_space_mem24`, `iio_space_mem32`. Operation `iio_space_io` is a synonym for `iio_space_mem16`, since A16 space is traditionally used for IO modules on the VMEbus. Proposed revisions of the VMEbus standard allow for an ‘identity PROM’ space, which would be accessed through `iio_space_id`.

On IndustryPack address spaces, the codes `iio_space_io`, `iio_space_id` and `iio_space_mem` select the register, ID-PROM and memory spaces of the IP respectively. (The `iio_space_int` code is intended for resolving interrupt vector numbers, and remains experimental at present). The ISA bus provides a single 24-bit address space, through either `iio_space_mem` or `iio_space_mem24`. 16-bit address ISA modules (those with the single edge connector) fit into the same address space.

The ISA port space addresses are represented by `iio_space_port`, but this space is somewhat different the others. It is available only on 80x86 processors, and it cannot be mapped into logical memory. Registers in this space are accessed through function calls (Section 5.5) rather than dereferencing register pointers. However, the address translation issue described above still applies, and the IIO address mapping and resolution system is used in the same way.

Address space channels with their attendant address space operations may be compared to UNIX device-special files such as `/dev/mem` or `/dev/vme16`, but they are used a different way. The UNIX special files are used to actually access the devices they represent, through their attendant kernel device drivers. The IIO address space channels and operations are used by `iio_map()` and `iio_resolve()` to *compute* logical addresses, so the driver can directly access the device later.

**Mapping the Module.** So, after decoding its configuration parameters, the module driver installation function should first map its registers or memory. For example:

```
if (!chan)
    iio_eret( iio_open("vme.0", 0, &chan) );
iio_eret( iio_map(chan, iio_space_mem16, base, 0x200) );
```

The first argument to `iio_map()` is an open channel descriptor representing the address space. Generally this descriptor is obtained through `iio_arg()`, as described in Section 5.3.3, and if not a default is opened, as shown. Once all the mapping and resolution is done, the channel should be closed.

The second argument to `iio_map()` is the address space operation code, from Table 5.2. This must be an operation that the given channel will implement. The example is for an A16 VMEbus module. The third argument is the physical base address of the board, expressed in that address space. The base address is usually set by switches or jumpers on the module, and the value given in the

vme	ip	isa	Address Space Operation Codes	
■	□	□	<code>iio_space_io</code>	Input/output Space
⊞	■	□	<code>iio_space_id</code>	Identity Space
⊞	⊞	⊞	<code>iio_space_int</code>	Interrupt Space
■	■	■	<code>iio_space_mem</code>	Memory Space
■	□	□	<code>iio_space_mem16</code>	16-bit Memory Space
■	□	■	<code>iio_space_mem24</code>	24-bit Memory Space
■	□	□	<code>iio_space_mem32</code>	32-bit Memory Space
□	□	■	<code>iio_space_port</code>	Non-mappable Port Space

**Table 5.2** Address Space Channel Operation Codes versus Channel Type. ■ indicates the address space operation code is supported; □ indicates it is not; and ⊞ indicates it may be in future.

configuration file, so the base address will have been obtained into `base` using `iio_arg()`.

The final argument is the size of the mapping required, expressed in the smallest addressable units of the space, usually bytes. Here, the size is `0x200` bytes, which means the module hardware will respond to A16 accesses from address `base` through to `base + 0x1ff`. Note that hardware often responds to a wider range than suggested by the register map, so check the hardware manual.

`iio_map()` tells the IIO to ensure a mapping exists from the program's logical memory space to the physical space used by the module. The function uses the given address space channel and operation code to resolve the bottom and top addresses (that is, `base` and `base + 0x1ff`) to processor *physical* memory addresses. It then looks up these addresses in a physical-to-logical map, and if a suitable mapping covering the whole range is not available, requests the operating system create one. The mapping it creates may be somewhat larger than requested.

Mapping is required even for the non-mappable port space `iio_space_port`. This is because some operating systems, such as Linux, require ports to be 'opened' before they can be accessed from user processes. `iio_map()` does this.

On operating systems like vxWorks, or on computer hardware that does not feature a memory management unit, there is no physical-to-logical map for IIO to worry about. However, the module driver must still call `iio_map()`, as it may later be used on an operating system or computer that does.

Some modules respond to more than one address sub-space. For instance, VMEbus frame-grabbers often have a number of control registers in A16 space and the frame buffer itself in A24 or A32 space. Function `iio_map()` should be called for each sub-space the module uses.

Some modules, in particular IndustryPacks, plug into carrier modules which themselves have module drivers. In these cases, the mapping is done by the carrier's module driver, as this reduces the number of individual physical-to-logical mappings (some systems limit this number). Thus, module drivers for IP's need not call `iio_map()`, although there is not harm done if they do.

**Resolving Register Addresses.** Once a mapping exists, the addresses of the individual registers can be resolved, using `iio_resolve()`:

```
iio_eret(  
    iio_resolve(  
        chan, iio_space_mem16,  
        iio_size_8, base + 0x10, &reg->csr  
    )  
);
```

The first two arguments should be the same as the call to `iio_map()`, identifying the address space/sub-space where the register is located, here VMEbusA16.

The next argument indicates the width of the register, chosen from Table 5.3. The specified width, 8-, 16-, 32- or 64-bits, must match the actual width of the hardware register *and* the pointer in the register structure to be used to access it. Do not confuse address width, which is usually part of the address space operation code, with data width. They are often independent.

The following argument is the physical address of the register, which will always be relative to the given address space/sub-space, and is usually a fixed offset from the module hardware base address. The final argument is a pointer to the register pointer, normally in the register structure, which is where the register's resolved logical address is put.

It is important that each individual register address be separately resolved. This is because it cannot be assumed that the same byte spacing will be maintained over the physical to logical mapping, which must also take into account

bus size and processor-endian issues. In other words, it is not acceptable to resolve just the base address, and index from there. Section 6.4 deals further with register addressing issues.

Usually a number of registers in the same address space need to be resolved *en masse*. Function `iio_resolve_list()` is provided for this:

```
iio_eret(  
    iio_resolve_list(  
        chan, iio_space_mem16,  
        iio_size_8, base + 0x10, &reg->csr,  
        iio_size_16, base + 0x0, &reg->dr[0],  
        iio_size_16, base + 0x2, &reg->dr[1],  
        0  
    )  
);
```

The first two arguments of `iio_resolve_list()` are the same as `iio_resolve()`, but they can be followed by any number of size/physical-address/register pointer triplets, terminated by a zero. In this example, an 8-bit control register and two 16-bit data registers are resolved.

### 5.3.5 Registering Channels

The last act of the installation function is to register the channels the module provides, and configure them as appropriate. Registration done using the function `iio_chnode()`:

```
IIIO_CHNODE *chnode;  
...  
iio_eret(  
    iio_chnode(  
        module,  
        iio_ctype_dac, 16, 4,  
        iio_xyzzy1234_dac,  
        &chnode  
    )  
);
```

This function accepts the module argument to the install function, then the type, width (in bits) and number of channels of that type. In this case, the module provides four 16-bit digital-to-analogue converters. Then follows a pointer to the operate function (described in the next section) for these channels, and finally a pointer to an `IIIO_CHNODE` pointer (if not later required, this argument can be `NULL`). The `IIIO_CHNODE` structure (a *channel node*) represents the range of channels just registered, and can be used later to configure the properties of these channels).

There must be a 1:1 correspondence between calls to `iio_chnode()` and the operation functions: that is, the operation function passed to one call cannot be passed to any other. This is because the channel numbering scheme, as far as the module driver is concerned, is based on channel nodes, not channel types or modules.

Code	Register Data Width		Pointer Type
<code>iio_size_8</code>	8-bit	byte register	<code>iio_int8_t</code> , <code>iio_uint8_t</code>
<code>iio_size_16</code>	16-bit	word register	<code>iio_int16_t</code> , <code>iio_uint16_t</code>
<code>iio_size_32</code>	32-bit	long-word register	<code>iio_int32_t</code> , <code>iio_uint32_t</code>
<code>iio_size_64</code>	64-bit	quad-word register	<code>iio_int64_t</code> , <code>iio_uint64_t</code>

**Table 5.3** Register Width Argument to `iio_resolve()`

**Channel Properties.** Once a group of functions is registered, the `IIO_CHNODE` pointer can be used to configure the properties of individual channels (Section 2.7).

At present, the only properties of interest are the channel scaling and offset values used by IIO when the channel is accessed through the `iio_operate_real()` function.

The linear scale and offset relate the real-world quantities present at the input or outputs of the module to the integer values read or written by software. Scaling is only really applicable to channels like analogue-to-digital or digital-to-analogue converters, or possibly timers.

The factors must reflect the configuration of the module. For instance, if a module features selectable gains or output ranges, the setting must be obtained from the configuration parameters, and used to compute the scale and offset factors.

The factors are set using `iio_chnode_linear()`. For a 12-bit DAC with a  $\pm 10$  V range:

```
for (seqno = 0; seqno < 4; ++seqno) {
    iio_eret(
        iio_chnode_linear(
            chnode, seqno,
            20.0/4096, 0.0, "V"
        )
    );
}
```

After the `IIO_CHNODE` argument, the `seqno` argument identifies the channel, the arguments are the linear scale factor (real units per integer unit), the offset (real units), and the real units as a string. Note the use of the exact scale ratio `20.0/4096`. Always choose the scale factors to refer to basic SI units, such as V, m or kg, rather than multiples like kV or mm.

Channel limits can be set using `iio_chnode_limits()`:

```
for (seqno = 0; seqno < 4; ++seqno) {
    iio_eret(
        iio_chnode_limits(
            chnode, seqno,
            0, -2040, 2040
        )
    );
}
```

The first two arguments are the same as `iio_chnode_linear()`. The next argument is the channel initial value property, which is currently ignored. The other two are the minimum and maximum integer channel values. The IIO core will not supply a value outside these inclusive limits, unless the minimum is greater than the maximum, which switches off limiting altogether (the default).

Note that while the channels are registered *en masse*, channel properties are configured one by one. If `iio_chnode_linear()` is not called for a given channel, the default scale and offset factors of 1.0 and 0.0 apply. The scale and offset factors may subsequently be multiplied by a user scale and offset specified by a `channel` directive in the configuration file.

## 5.4 Initialisation Function

If the installation function did not return error status, IIO may then call the initialisation function. This function actually accesses and initialises the module hardware, as well as the module state structure, if any.

The initialisation function is called only once in the life of the system. This is fairly easy to arrange on shared-memory systems like vxWorks where the life of the system is generally the life of the single application program. On LynxOS systems, IIO provides an elaborate system to ensure the initialisation function is called only once even if there are several IIO-using processes running, and that the state structure is properly preserved and shared amongst all of these processes.

**Probing for Hardware.** The `iio_probe()` function is used to see if there is hardware actually at the register addresses resolved in the installation function:

```
iio_eret( iio_probe(reg->csr, iio_size_8, iio_ptype_read) );
```

This function returns error status if there is nothing at the probed address (i.e., a bus error or segmentation fault occurred). The most likely cause of this is a wrong address in the configuration file, or the hardware is not plugged in.

`iio_probe()` accepts a resolved register address as the first argument, a register width argument (Table 5.3), and the probe type (`iio_ptype_read` or `iio_ptype_write`). Generally, it is enough to read-probe only one register (but choose one that will not cause the hardware to do something potentially dangerous!)

Note that certain computers (such as ISA bus systems) do not detect bus errors, and on these systems `iio_probe()` should not be used. In particular, it should never be used for register addresses that are in the non-mappable port space (that is, were resolved and mapped using the `iio_space_port` address space operation code).

Probing will not detect that the *wrong* module has been configured to the *right* address, if the module happens to have a register at the probed address. However, it will catch many module set-up errors. If the module is self-identifying, test the identity of the module against what the driver expects. If it does not match, the initialisation function should return an error status, using `iio_error()` (Section 6.12).

IndustryPacks are always self-identifying, and because they are common, module probing and identity checking have been combined into the one function. `iio_ipinfo_ident()` probes the ID-PROM which exists on all IndustryPacks, and checks the manufacturer and product identity codes against those that the module driver supports:

```
iio_eret( iio_ipinfo_ident(reg->slot, 0xf0, 0x16) );
```

The first argument is the address space channel which would have been used for register address mapping and resolution in the installation function. Normally, for an IP, this is saved in the register structure for use by `iio_ipinfo_ident()`. The second and third arguments are the manufacturer (0xf0 for GreenSpring) and product (0x16 for an IP-DAC) identity codes. These are unique to each IP and should be found in the module's documentation. `iio_ipinfo_ident()` copes with both the old-style and the new-style ID-PROM formats.

**The State Structure.** The state structure duplicates whatever module state cannot be read when required from the module hardware itself. It will contain everything the module driver needs to know about the module *except* any configuration information in the register structure. Typically, the state structure will contain shadows of write-only registers, so that the read-back operations can be supported:

```
struct IIO_MSTATE {
    iio_uint16_t dr[4];      /* shadows of data registers */
};
```



Register shadows should be declared to be the same width as the register, but of course should be real variables, not pointers. They do not need to be declared `volatile`.

The state structure is defined, like the register structure, at the top of the C code for the module driver, and is also allocated in the installation function. However, it must be initialised, like the module hardware, in the initialisation function.

**Hardware and State Initialisation.** Finally, the hardware can be initialised to correspond to the configuration stored in the register structure, and any outputs cleared to safe or zero values. In the case of the XYZZY-1234, the inversion flag indicates certain control register values:

```
*reg->csr = (reg->invertflag) ? 0x03 : 0x01;
for (i = 0; i < 4; ++i)
    *reg->dr[i] = state->dr[i] = 0x0;
```

As described, IIO only calls the initialisation function if it believes the hardware is not initialised. On UNIX systems, this means the first run of an IIO program only. On subsequent runs, IIO assumes the hardware is configured and running, and so the next thing that might happen in the module driver after the installation function is a call to the operation function.

## 5.5 Operation Function

The operate function is called in response to user program calls to one of the operate functions. While the processes involved in installing modules and opening channels in IIO are fairly complicated, the operation functions are (hopefully) quite short, simple and fast.

The operation function only has to deal with single channel range of a single type at once. (There is generally one operation function for each type of channel a module provides, and certainly only one operation function for each call of `iio_chnode()` in the installation function). It must have the same prototype as this example:

```
HIDDEN IIO_STATUS iio_xyzzy1234_dac(
    IIO_MSTATE *state, IIO_MREG *reg, IIO_OPNODE *opnode,
    IIO_OP op, unsigned first, unsigned number
)
```

The function receives the state and register structure pointers, an `IIO_OPNODE` pointer, the IIO operation code, the local sequence number of the first channel in the range and the number in the range. These are all the variables the operation function should need directly; other values used by IIO are encoded in the `IIO_OPNODE` structure. The operation function's job is to actually carry out the operation on the module hardware.

**Channel Sequence Numbers.** Normally, the operation function will contain a `for`-loop to deal with each channel in the range, which will enclose a `switch`-statement on the operation code:

```
unsigned seqno;
...
for (seqno = first; seqno < first + number; ++seqno) {
    ...
}
```

Some hardware can be more efficiently accessed in ranges by different means. For instance, there may be a similar overhead for accessing one channel or twenty, so it is useful for the module driver to know how many channels are involved. Modules with simultaneous capture inputs or output may also need to know this. In other cases, the `for`-loop is merely a nuisance.

The channel sequence numbers start from zero for each channel node. Normally, all the channels of a given type on a given module are in the same channel node, because they were registered by a single call of `iio_chnode()`, and are handled by the one operation function. If the channels are not all registered together, there has to be a separate operation function for each group. If the same function was used for both groups, there would be no way for it to tell which group the channel belongs to.

**Accessing User Data.** The operation function calls one of several IIO core functions to obtain the data the user passed to the operation, and return the operation's results to the user. The IIO core scales, offsets and limits the data according to the properties registered with the channels, so that the driver need only apply with the simplest data transformations, if any.

Usually the module driver operation function needs the user data converted to an integer format, since that is the form most IO hardware requires. The `iio_data_get()` function is used in this case:

```
int val;
...
val = iio_data_get(opnode, seqno);
*reg->dr[seqno] = (iio_uint16_t)((val & 0xffff) + 0x800);
```

The first argument is the `IIO_OPNODE` pointer, and the second is the local sequence number of the channel whose user datum is required. The `iio_data_get()` function locates the correct user datum, performs whatever scaling, offsetting, limiting or logging is required, and returns the result. The driver must do the final transformation of the datum before inserting it into the hardware, such as adding an offset, as in the example above, which is an 'offset-800' style DAC.

Similarly, when returning a datum to the user, the `iio_data_set()` function is used. For instance:

```
int val = *reg->dr[seqno] & 0xffff;
iio_data_set(opnode, seqno, val - 0x800);
```

The third argument to the function is the new signed integer datum.

The module driver may access the user data in two other forms. The get and set functions `iio_data_get_real()` and `iio_data_set_real()` are identical to the functions `iio_data_get()` and `iio_data_set()`, but return or accept double data. These functions are used when the rounding-off that occurs in the latter functions becomes a problem. It should be noted that the function still perform the scaling, offsetting, limiting or logging as required: the only difference is that rounding-off does not occur.

User data that represents pointers (address data) should be accessed using `iio_data_get_addr()` and `iio_data_set_addr()`. These functions access the user data without modification. They will only work if the user has used `iio_operate_addr()` to do the operation. These are generally only used in address space channel drivers (Section 6.7).

**Implementing the Operations.** It is up to the module driver writer to make sure the correct operation codes are implemented for the channel type, and that the `iio_data_get()` and `iio_data_set()` functions are called appropriately. These are listed in Table 2.4 on page 13.

Typically, the operation code is used as the selector in a `switch`-statement, inside the channel loop (although the reverse is perhaps slightly faster):

```
for (seqno = first; seqno < first + number; ++seqno) {
    switch (op) {
        case iio_op_read:
            ...
        case iio_op_write:
            ...

            ...
        default:
            return iio_error("Operation code not supported");
    }
}
```

There should always be a `default` clause that returns this error should an illegal operation code be given. The operation codes are not validated by the IIO core.

**Modules in Port Space.** The example so far has assumed the module has been memory mapped, which is the most common way of accessing modules in IIO. However, many ISA modules are port-mapped, and so the 80x86 processors found in ISA systems cannot map them into logical memory. Instead, special `inp` and `outp` instruction must be used to access registers.

Unfortunately this means module for port-mapped modules must use functions to access registers, rather than dereferencing the register pointer directly. These functions are declared:

```
extern void
    iio_port_set8(volatile iio_uint8_t *addr, iio_uint8_t val),
    iio_port_set16(volatile iio_uint16_t *addr, iio_uint16_t val),
    iio_port_set32(volatile iio_uint32_t *addr, iio_uint32_t val);

extern iio_uint8_t
    iio_port_get8(volatile iio_uint8_t *addr);
extern iio_uint16_t
    iio_port_get16(volatile iio_uint16_t *addr);
extern iio_uint32_t
    iio_port_get32(volatile iio_uint32_t *addr);
```

The first three functions are for writing to registers. The first parameter is the register address, and the second is the value to write. The register address must have been obtained through the IIO address resolution and mapping mechanism using the `iio_space_port` address space. The second three functions are for reading registers.

Separate functions are provided for each data width, to ensure the correct access instruction is used, and to allow the functions to be re-implemented as macros or inline functions if necessary.

**Write-Only Registers.** Data written to write-only registers should be cached in the state structure. For instance, if the hardware has a number of write-only data registers accessed through a pointer array `reg->dr[]`, then there would typically be a matching array of equal size integers `state->dr[]` containing the cached register values:

```
*reg->dr[seqno] = state->dr[seqno] = (iio_uint16_t)(val & 0xffff);
```

or if the module was port-mapped,

```
iio_port_set16(
    reg->dr[seqno],
    state->dr[seqno] = (iio_uint16_t)(val & 0xffff)
);
```

The cached values would normally be returned to the user in an `iio_op_readback` operation.

The module driver need not worry about shared resource issues. Each module has its own system-wide mutual-exclusion semaphore, which is taken by the IIO library before any of the module driver functions are called. Thus, assuming these register shadows are initialised and maintained properly by the driver code, they should remain consistent with the actual value in the hardware.

## 5.6 Integrating a Driver into IIO

When new module hardware is to be used in a system, it is better to write the driver into the IIO library than into the application. It is generally more convenient to fit it into IIO, and it means the driver is available to others who may wish to use that hardware. The user can also benefit from any generic tools that are based on IIO.

This section outlines the procedure for patching in a new driver. In short, the procedure is:

- write the source code
- put it in the `iio/src/module` subdirectory
- alter the files `iio/src/standard.c` and `iio/src/Makefile`
- in directory `iio/src`, type `make`
- once the driver works, document it in `iio/doc/manual/module`.

It assumes a writable set of sources is available.

**Source Code.** Normally, the four required functions (identification, installation, initialisation, and operation), the definition of the two data structures (`IIO_MREG` and `IIO_MSTATE`) and all the code and declarations required for the module hardware are written into a single C file. This makes inclusion of the driver into the IIO library straightforward. There is no need for a separate header file, since there should be nothing that will need to include it. (Chip drivers, as explained in Section 6.1, are an exception, and do have header files.)

The driver module must include the file `internal.h`, which contains declarations of the internal IIO functions and data structures needed by the driver. A copy of `internal.h` appears in Appendix F.2.

There is no ‘driver skeleton’. It is often best to start with an existing driver for a similar kind of module, edit out the unwanted code, and edit in the code for the new hardware. It is usually a good idea to implement the optional `iio_op_show` operation, or alternate debugging infrastructure first, to prove that all the register accesses work: initialisation can be done next, and the remainder of the operations last.

You should ensure that the driver implements the correct set of operation codes for each of the channel types it provides. Table 2.4 on page 13 is the master reference, while Section 4.10 describes what the operation codes are supposed to do for common channel types. As well, make sure that only the initialisation and operation functions access hardware; that no pointer or process-specific data is in the state structure; that the driver has no global or static variables (except read-only look-up tables); and that it allocates no memory.

The examples given in this section show how to write a driver for a reasonably straightforward IO module, but less straightforward modules will be encountered. These may entail the use of chip drivers, may require address space channels to be implemented, or even new channel types or operations codes to be defined. Section 6 covers these issues and more.

**Module Driver Directory.** The model ident code (in the previous section's running example, `xyzy1234`), should be used as the source filename. Module driver source files should be placed in the sub-directory `iio/src/module`. Since `internal.h` is in `iio/src` (refer to Figure C.1 on page 124), it is included in the module driver source using:

```
#include "../internal.h"
```

There is no need to include `iio.h`, as it is already included by `internal.h`.

**Driver List and Makefile.** The standard IIO driver list is in `standard.c` in `iio/src`. This file consists only of an external declaration of each module driver's identification function, and an array of pointers to these functions. A pointer to this array, `iio_standard`, is generally passed to `iio_init()`. Equivalent references for the identification function of the new driver, `iio_xyzy1234()`, should be inserted.

The **Makefile** must also be altered. There is a list of drivers defined near the top of the file (variable `MODULENAMES`). Simply add the model ident of the driver to this list in its alphabetical position.

**Making and Testing.** The IIO **Makefile** takes care of architecture and operating system dependencies, segregating object modules and libraries. Usually it is necessary only to type `make` in the main source directory `iio/src`.

The **Makefile** always builds the IIO library and the interactive test program `iio` (Appendix B), which is useful for testing the new driver. First check to see if the module appears on the `minfo` list. Then try a safe channel operation, like a channel read.

All module drivers are compiled for the given architecture/operating system combination (platform), even on platforms that don't usually permit direct user access to IO modules, such as SunOS and Linux. Appendix C described this further. With care, much of a module driver can be tested on such platforms, simply by temporarily removing the register accesses in the initialisation and operation functions, and replacing them with `printfs`. This is sometimes convenient if software development on the target platform (with the module hardware) is inconvenient or unavailable.

**Documentation.** Each module driver should have a documentation page prepared using  $\text{\LaTeX}$  and incorporated into Appendix A of this manual. Use one of the existing pages as a guide. Macros are used to present a consistent format. If the new module involved the addition of a new channel type or operation code, then more complicated additions to this document are required.



## Section 6

# More on Modules and Module Drivers

The previous section attempted to give a step-by-step description of writing module drivers, without pursuing too many side-issues. This section, by contrast, deals with a range of issues related to IIO module drivers, but not necessarily related to each other.

### 6.1 Chip Drivers

Many IO modules are built from basic integrated circuits, such as as latches or analogue-to-digital converter chips. For this run-of-the-mill hardware, the module drivers only have to deal with a number of quite simple interfaces. Such modules are generally sufficiently different from each other to justify different drivers for each.

The situation is less clear where the module is built around one of the more complicated peripheral chips, such as the MC68230 parallel interface/timer, the Am9513 timer/counter, the LM628 servo controller, or one of a host of others. Arguably these chips are modules in themselves, in that they provide IO channels to the computer, and more than one of the same kind may appear in a system. However, they are also only really components and they must be *part of* a module: they need extra things like address decoders, buffers, and so on to work in a practical system.

These chips frequently do not exhibit a simple interface, and may have many registers, or use a serial register file accessed through a single register address. Accessing, configuring and operating them may require quite a lot of code. This code can, of course, be written into the module driver, but this means that if the chip is encountered in a different module the code must be duplicated, or even re-written. Eventually this may lead to inconsistent behaviour—operations performed on channels provided through the chip will have different results depending on which module they happen to be on.

The purpose of chip drivers is to allow the code that supports particular peripheral chips to be isolated from the module drivers, creating a suite of ‘sub-drivers’. Modules that contain these chips can then call chip driver functions to install, initialise and operate the chip, rather than doing it themselves. In this way, chip operation is consistent, and improvements to the chip driver benefits all client modules.

#### 6.1.1 When to Write a Chip Driver

The programmer should consider writing a chip driver when:

- a module has a chip which requires more than several lines of code to initialise or operate it, especially if the module has an array of these chips
- the chip provides some or all of the channels of the module in its own right
- the operation of a chip took a while to understand, and so is worth encapsulating in a separate piece of code

- a module driver is getting large and complicated, and needs to be modularised. Doing so along the lines of the component chips usually makes sense.

On the other hand, the programmer might decide *not* to implement a chip driver, if, for instance:

- the chip is essentially a module-specific ASIC or gate-array, and there is little chance that the chip will be encountered on any other module
- the chip is not used to provide IO channels in its own right, but is part of the module's infrastructure. For instance, a common counter/timer chip might be used to generate specific timing signals used only within a module.

### 6.1.2 Chip Driver Interface

There is no chip driver interface standard: drivers may be free-form. But they must still fit in with the general IIO module driver approach, because they have to be *used* by module drivers. This imposes a general form on them, with the differences being in the details. Thus, a chip driver will typically have register and state structures, and installation, initialisation, and operation functions. These functions can be close in form to the module driver ones, and accept similar arguments, or they can be different.

It is essential, of course, that the chip drivers do not violate any of the requirements of module drivers: they must be completely reentrant; they must only access hardware through pointers in their register structures; they must only store legitimate chip state in their state structures; they must not meddle with any other hardware; and so on.

They should also conform with the general principles of IIO module drivers: they should be as generic as possible; they should support as many features of the hardware as practical; they should be as fast as possible, especially in those parts invoked by operation functions; and so on.

Two chip drivers used within the current set of IIO module drivers exemplify the possible differences chip driver interface design.

**A Chip Driver Example.** The GreenSpring IP-WATCHDOG module contains a DS1620 thermometer chip. Essentially, this chip is an analogue-to-digital converter with a few extra features. It can be read to obtain the current temperature, and temperature thresholds can be written into it so it can operate as a thermostat. On the face of it, a chip driver would hardly be justified, except that all communications with the chip are conducted through a peculiar two-wire bit-serial interface, and the chip must be fed certain bit sequences so that it may work properly.

The DS1620 chip driver thus only vaguely resembles an IIO module driver. There are separate functions to initialise the chip, and to start temperature conversions, read the results, read or write the thresholds, and so on. These in turn depend on primitive functions that send and receive words through the two-wire interface to the chip. However, the chip driver cannot assume the chip is actually built into an IP-WATCHDOG module, and so it cannot know how to toggle the interface wires into the chip.

The problem in this case is solved by the calling module driver passing a *call-back function* pointer to the chip driver functions. The chip driver calls this function to toggle the two-wire chip interface wires. In other words, the logical operation of the chip is dealt with by the chip driver, but the module-specific part of the interface is still dealt with by the calling module driver.



**A Different Example.** At the other end of the scale, the chip driver for the LM628 servo-controller chip very strongly resembles an IIO module driver.

There is a register and a state structure, which serve exactly the same purposes as they would in a module driver. There is an installation function, which accepts module parameters, fills out its register structure, and registers channels with the IIO core, just like a standard installation function. On the other hand, it does not allocate the register and state structures, as this is done by the calling module on its behalf. While it parses the module parameter list, it only decodes the parameters that pertain directly to itself: parameters that apply to the whole module are decoded by the module driver beforehand, and the decoded values, where relevant, are passed to the installation function.

There is an initialisation function and an operation function, again very similar to the module driver counterparts. The operation function is passed the standard IIO operation codes, IIO\_OPNODE, channel sequence numbers, and so on.

Like the DS1620, however, the chip interface is fairly complicated, this time a byte-serial arrangement. Thus, this chip driver also has two layers, with communication primitives at the bottom. Unlike the DS1620, however, the chip presents conventional addressable registers, so there is no need for the communications primitives to use a call-back arrangement as described in the previous example.

### 6.1.3 Chip Driver Code

Chip driver code resides in the `iio/src/chip` directory. Unlike module drivers, chip drivers *do* have header files. The header files should define the register and state structures, where required, as well as prototyping the chip driver functions.

If the chip is, for instance, a FZ3344, the header file would be called `fz3344.h` and the chip driver proper would be `fz3344.c`. The register structure would be called `IIO_FZ3344_MREG` and the state structure `IIO_FZ3344_MSTATE`. Functions named `iio_fz3344_whatever()` and so on.

Module drivers that used the chip driver would thus include the chip driver header file, after `internal.h`, like this:

```
#include "../internal.h"
#include "../chip/fz3344.h"
```

The register and state structures for the *module* incorporate one instance of the chip register and state structures for each chip on the module. For instance, if a module used four of these FZ3344 chips, the structures might look like:

```
struct IIO_MREG {
    /* four FZ3344 chips */
    struct IIO_FZ3344_MREG reg[4];

    /* a module register */
    iio_uint16_t *csr;
};

struct IIO_MSTATE {
    struct IIO_FZ3344_MSTATE state[4];
};
```

Note that non chip-specific registers or state is simply included in the register and state structures as normal, such as with `csr`.

When the chip driver functions are called, the module driver passes pointers to the particular chip's register and state structures. Chip drivers should not allocate their own register and state structures.

### 6.1.4 Integrating Chip Drivers into IIO

Chip drivers are normally integrated into IIO at the same time as the module drivers that use them. Integration is much simpler than for a module driver. All that is required is that the name of the chip driver module be added to the variable `CHIPNAMES` in `iio/src/Makefile`.

## 6.2 Generic Driver Code

Generic driver code refers to routines that are used by a number of drivers, usually of the same general class, but which aren't generic enough to be regarded as part of the IIO library core. Two such driver classes have emerged so far: IndustryPacks and ADAM serial-addressable units.

IndustryPacks feature an ID-PROM which encodes the identity of the module, along with other information which may be useful to the driver, such as factory calibration constants. There is a group of functions which decodes this information which are called by module drivers for IPs: these are described in Section 6.6.

Similarly, there is a group of functions for manipulating range codes and hexadecimal strings related to the ADAM serial-addressable units. This is discussed in Section 6.10.

These class-specific generic modules are in the `iio/src/module` directory, along with the module drivers they support. However, their associated declarations are in `internal.h`.

## 6.3 Proxy Drivers

In a realistic system, some of the IO hardware may not be under the control of IIO. The reasons for this may include:

- the hardware is of a type that falls outside the general ambit of IIO
- the hardware required the use of a driver only available as a binary library, and so cannot be integrated.

In these cases a compromise solution may be available through the use of a *proxy driver*. Proxy drivers represent the module to IIO, but do little if anything to the hardware, which is handled outside of the IIO system. The most useful thing IIO can offer in these cases is module installation through the IIO configuration file, and address resolution.

A proxy driver is listed in the configuration file like any other. Instead of installing and initialising the hardware, however, it will call some other function or execute some other routine to actually install or initialise the hardware, or to start some external software system, such as installing a kernel module. The parameters for this call (and the name of the call itself) should be derived from the module parameters in the configuration file.

The call may also pass the virtual or processor-physical addresses of module registers, obtained through the IIO address resolution mechanism (Section 5.3.4). This is particularly useful for IndustryPacks where register addresses depend on the slot they are in. In the case of IPs and other self-identifying modules, the proxy driver initialisation function can also check the module's identity. The proxy driver will generally not introduce any channels, so it will have no operation function.

The module driver for the IP-SERIAL (page 100) is of this type. In this case, the serial ports it provides are outside the ambit of IIO and are better dealt with by the operating system. The installation function decodes the module

arguments, and resolves to processor-physical addresses the four registers of the module. The initialisation function checks the IP's identity, then executes an external program (whose name is a module parameter) to make the operating system use the module for normal serial ports. The external program, probably a script, receives the register addresses as arguments.

This arrangement is obviously operating system specific, but at least the system-specific code is not written into the IIO proxy module driver. This driver can be compiled and used on different systems: however each system will have a different script, or it may not even have a means of installing serial port dynamically.

## 6.4 Endianism and Module Registers

The ugly term *endianism* refers to the ordering of the bytes of integers stored in memory. So-called 'big-endian' processors store the big end (that is, the highest order bytes) in a word first, or at lower memory addresses. This is sometimes referred to as Motorola ordering. 'Little-endian' processors store the bytes in the reverse order, with the little-end first, or at lower memory addresses. This is sometimes known as Intel ordering.

Many portable device drivers contain ugly solutions to the problem of endianism, usually involving C macros which re-arrange bytes in a word as they are read or written. Alternatively, register bit-patterns are defined in big- and little-endian forms, which are selected by pre-processor symbols. This is in fact unnecessary, and none of the current IIO module drivers contain macros of this sort, or indeed any references to the issue at all, yet they work correctly on both big- and little-endian processors.

Endianism is only really an issue where the same data is accessed as sequences of different sized elements. For instance, a 32-bit register can usually be accessed as a single 32-bit long word, as two 16-bit words, or as four 8-bit bytes at adjacent addresses. The solution to the problem is simply to *not* access the register as separate bytes or words. Make all accesses to a given register the same size (and the same size as the register), and almost all of the endianism 'problem' vanishes.

This approach is enforced to a degree by the IIO module driver register structure (Section 5.3.5). Each register has its own pointer, and the pointer has a particular type, chosen to match the size of the register. An 8-bit (byte) register would have a `iio_uint8_t` pointer, a 16-bit (word) register would have a `iio_uint16_t` pointer, or a 32-bit (long word) register would have a `iio_uint32_t` pointer.

Each of the register addresses (and there may be a large number of them) must be resolved separately, using `iio_resolve()` or a variant. Any endian effects in address resolution are dealt with by this function, or by the address space module drivers that it invokes. This is the reason the size of the register is needed when its address is resolved.

Before resolution, however, the driver must know the local physical addresses of each of the registers. The hardware manual will have these, but sometimes they are misleading because different register addresses are given for different processors. This is not so much because of endian effects, but because of different conventions about whether single-byte registers should be connected to the upper or lower halves of a 16-bit data bus. This is a slightly different issue from endianism, but they are frequently blurred together.

Usually a little investigation (and sometimes even experiment) is required to tell which convention the module manufacturer really uses, and so which values to use as local physical addresses in IIO module drivers.

## 6.5 ISA Bus Errors

The ISA bus lacks a bus cycle acknowledge signal, and so there is no feedback to the processor as to whether a read or write was completed or not. This is quite different to the VMEbus, where accessing an address where no hardware responds causes a *bus error*. The bus probing function `iio_probe()` accesses module addresses and uses the bus errors to determine whether there is hardware present or not.

This function does not work in ISA systems (it always succeeds), which argues for it to be omitted from ISA module drivers. This should not be done, however, as a workable probing technique may one day be found and incorporated into `iio_probe()`. In the interim, it has to be accepted that IIO is less able to check the correctness of module addresses given in the configuration file.

## 6.6 Module Drivers for IndustryPacks

Module drivers for IndustryPacks differ only slightly from the drivers for other kinds of modules. The the first two differences were mentioned in Sections 5.3 and 5.4.

**Module Mapping.** There is no need to map the module by calling `iio_map()`, since the IP carrier module drive will have already done so. This is a pragmatic convention made because it may reduce the number of virtual-to-physical mappings required for the system: many operating systems limit the number of mappings a program may create.

A similar convention will probably apply to other IO carrier/mezzanine products, should they come to be supported.

There is no harm done if an IP module driver does map itself with `iio_map()`, as IIO will ignore the duplicate mapping request. The driver must still resolve the addresses of its individual registers, using `iio_resolve()`.

**Module Probing.** Instead of using `iio_probe()` to test if the module (or a module) is installed at the given address, use `iio_ipinfo_ident()` (Section 5.4). This probes for the IP's ID-PROM, and if found, checks the manufacturer and product identity codes against those the driver expects (a driver can, of course, support a number of similar IPs at once).

**Register Physical Addresses.** Section 6.4 mentioned the general confusion about processor endianism. Evidence of this is the GreenSpring IP manuals, which give different register addresses depending on what bus the *carrier* module plugs into.

In IIO this does not matter, as the address resolution mechanism, and more specifically the module driver for the carrier module, should deal with the address translations through the module. In general, you should use the addresses given for VMEbus carriers, no matter what kind of IP carrier bus you are using. You should of course ignore references to the the carrier base address and slot offset: these are all dealt with automatically. The numbers that should go into `iio_resolve()` are simply the address offsets from the base of the register address space.

**Module Information.** The ID-PROM can contain information other than just the manufacturer and module code. Often factory calibration data for individual IPs is stored in the ID-PROM, which is useful to the driver. This information can be accessed by resolving individual elements in the ID-PROM (just like registers,

except using `iio_space_id` address space code). It is messy to access because the size of the inter-byte gaps in the ID-PROM vary with the carrier bus type.

A simpler way is to use the `iio_ipinfo_read()` function:

```
IIO_STATUS iio_ipinfo_read(IIO slot, IIO_IPINFO *ipinfo);
```

This probes the IP at the given channel, and returns the contents of the ID-PROM into a user-supplied `IIO_IPINFO` structure. `IIO_IPINFO` is defined in `internal.h` as:

```
struct IIO_IPINFO {
    iio_uint32_t mid;           /* manufacturer code */
    iio_uint32_t pid;           /* product code */
    iio_uint32_t rev;           /* module revision number */
    iio_uint32_t did;           /* driver ID code */
    iio_uint32_t flg;           /* flags word */

    /* the original data from the IDPROM */
    iio_uint16_t prom[0x20];
};
```

The structure field `prom` contains the original data from the ID-PROM, in an easily addressable format. Old-style 8-bit ID-PROMs will only use the bottom byte in each word, with a zero top byte. The locations of module-specific data in this array should be found in the module manual.

Some of the standard data from `prom[]` is extracted into the other fields. `mid` is the manufacturer code and `pid` is the product code: these are the two quantities the driver should use to check the identity of an IP. (If `iio_ipinfo_read()` is called and these two checked, there is no need to also use `iio_ipinfo_ident()`). `rev` is the module revision number, which might be useful if IP design changes affect drivers. `did` is the driver identity code and `flg` is a flags word: the purpose of these is unclear.

There are some other fields in `IIO_IPINFO` that are not shown, and which may be removed later. Only use the fields shown above.

## 6.7 Module Drivers for Address Spaces

Address space channels were introduced in Section 5.3.4, in the context of address mapping and resolution. Address space channels represent addressing hardware, such as a VMEbus or an IP slot. These channels are provided by modules just like ordinary channels, although the channel operations the module driver must implement are different: the address space operation codes are shown in Table 5.2 on page 40.

The module driver in these cases is structured exactly like other module drivers, and the same four basic functions should be provided. Such modules can of course provide other ordinary IO channels as well, if the module hardware has them, but this is unusual. In general, address space module drivers are simpler than their counterparts.

Address space modules only have to deal with address resolution, not address mapping (although they are invoked as part of the mapping process as well). They do not need to deal with the virtual-to-physical map, as this is managed by the IIO core, in the form of `iio_map()` and `iio_resolve()`. What they do have to do is deal with that part of address resolution that thus done be they module hardware they represent.

### 6.7.1 Invoking Address Space Operations

Operations on address space channels can be invoked using `iio_operate_addr()` (Section 4.4.3). The channel descriptor should refer to an open address space device. The operation code should be one of the address space operation codes of Table 5.2 arithmetically ORED with the register width code (type `IIO_SIZE`, Table 5.3 on page 42). This is different from other IIO operation codes, which are used by themselves, and comes about because operations can have only one argument, whereas two are required for this purpose. The register width is essentially a qualifier for the operation code, which specified the address sub-space.

Note that `iio_operate_addr()` accepts a *pointer* to the address to be resolved, not the address itself. This is because the result of the operation is overwritten on the input data (this is less of a problem than it might seem).

The input address is expressed in the space of the address space channel and the address sub-space given by the operation code. The output is an address resolved to a *processor physical* address. A processor physical address is the address that would be output from the memory management unit (MMU). Is is the *physical* address that is as close to the processor as possible:  $a_1$  in Figure 5.1 on page 38. Such addresses are sometimes known as local addresses, because they are addresses on the CPU module's local or internal bus.

Now, each address space module driver can only know how to resolve an address as far as the address space it is plugged into. To resolve the address to a processor-physical address, it must itself invoke `iio_operate_addr()` on the channel it connects to. Thus, a user resolving an address will transparently invoke a chain of recursive module driver calls, all the way back to the CPU module. Each module will perform its little part of the resolution process, usually adding an offset related to its configured base address.

The final part of address resolution, from processor-physical to logical, is handled by the function `iio_resolve()`, which is the way all address resolutions should be invoked, *except* by address space module drivers. Function `iio_resolve()` calls `iio_operate_addr()` using the supplied address space channel, operation code, register width and physical address. The resulting processor-physical address is then looked up in its table of physical-to-logical maps, to obtain the corresponding logical address that it returns to its caller.

Function `iio_map()` also uses `iio_operate_addr()` to invoke address space operations, but instead uses the resulting processor-physical address as the argument for an operating system call to *establish* a logical-to-physical map, if one is not in the list. The new mapping is added to the list, so mappings are not duplicated, and so `iio_resolve()` does not need to query the operating system.

### 6.7.2 Installation Function

The installation function for an address space module driver is of the same form as one for an ordinary module. It must process the module parameters from the configuration file, allocate the register and state structures, and register the channels it provides to IIO using `iio_chnode()`. These will of course be address space channels.

There will always be a register structure, because the channel descriptor representing the address space the module is plugged into is needed by the operation function, and the correct place to store it is the register structure. Usually other module parameters (like the base address) are also needed and go in the register structure.

The installation function should not use `iio_map()` to map the address spaces they provide, *except* for IndustryPack modules and similar products, where the address spaces are quite small.

### 6.7.3 Initialisation Function

The initialisation function is frequently empty, unless the module has control registers that can be probed or self-identifying features of the module that can be checked. The addresses in the address spaces should not be probed, because it is quite legal to have an address space module plugged in but with nothing plugged into it. If an address space module is configured to the wrong address (either on the module or in the configuration file) it should become apparent when modules plugged into it are initialised.

### 6.7.4 Operation Function

As described, the address space operations must add the address space module base address, or whatever other address transformation the module hardware performs, to the argument address, and then invoke the same operation on the module it is plugged into. The argument address should be obtained from the caller using `iio_data_get_addr()`, and the result returned using `iio_data_set_addr()` (the other data access functions will mangle the address data by applying limits or channel scale factors).

The core of the operation function will resemble this example, which is for a VMEbus IndustryPack carrier:

```
for (slot = first; slot < first + number; ++slot) {

    iio_uint32_t offs = (iio_uint32_t)iio_data_get_addr(opnode, slot);
    IIO_SIZE size = (IIO_SIZE)(op & IIO_SIZE_MASK);
    void *result;

    switch (op & IIO_SPACE_MASK) {

    case iio_space_io:
        if (offs > 0x7f)
            return iio_error("IPIO address out of range");
        result = (void *) (reg->a16addr + 0x100 * slot + 0x00 + offs);
        iio_eret(
            iio_operate_addr(reg->bus, iio_space_mem16|size, &result)
        );
        iio_eret( iio_data_set_addr(opnode, slot, result) );
        break;

    case iio_space_id:
        if (offs > 0x7f)
            return iio_error("IPID address out of range");
        result = (void *) (reg->a16addr + 0x100 * slot + 0x80 + offs);
        iio_eret(
            iio_operate_addr(reg->bus, iio_space_mem16|size, &result)
        );
        iio_eret( iio_data_set_addr(opnode, slot, result) );
        break;

    case iio_space_mem:
        ...

    default:
        return iio_error("Space code not supported by channel");
    }
}
return iio_status_ok;
```

Recall that the address space operation codes are qualified by (OR'ed with) the register width code. After using `iio_data_get_addr()` to obtain the address

argument, the size is extracted using the macro `IIO_SIZE_MASK`, since it will be needed below. Similarly, the operation code `switch`-statement selector needs to exclude the size code.

Each of the arms of the `switch` should check that the argument address is within the allowable size of the address space. Then the address is partially resolved, by adding the module's base address direct from the configuration file (in this example, stored by the installation function in `reg->a16addr`) and a further offset depending on the slot number (which is the same as the local channel sequence number) and the particular address space. The result is then resolved on the VMEbus A16 space (note how the size code is combined with the operation code), and the result from that (overwritten into `result`) is returned to the caller using `iio_data_set_addr()`.

## 6.8 Module Drivers for CPU Modules

Module drivers for CPU modules usually provide just one address space channel for the bus interface they provide: `vme.0` in the case of VMEbus module, `isa.0` (or perhaps one day `pci.0`) in the case of PCI ISA systems, and so on.

CPU modules do not usually plug into other address space devices. Module drivers for CPU modules are thus even simpler than other address space module drivers, since they do not need to further resolve the address on the channel they plug into. (They could, for the sake of symmetry, resolve the address using the `null.0` channel, but this would not really achieve anything).

On VMEbus processors, there is usually a bus interface chip that maps parts of the three main VMEbus address spaces onto the processor physical space. The address space operations implemented by the operation function in this case must follow the operation of this chip. Usually such chips are initialised by the operating system, which would subsequently expect it to be left alone. The IIO module driver must thus only read this chip, not write to it, to avoid conflict with the operating systems. (The current drivers in fact only implement the built-in processor-to-VMEbus mappings, and do not even read the chip).

The same goes for any similar chips in the CPU module. It should be noted that IIO, as with most device driver arrangements, will be undermined by operating systems that change address maps one the fly: this includes the logical-to-physical maps implemented by the MMU, at least the ones established by IIO.

On ISA systems, there is generally no address translation between the ISA bus and the local bus—they are logically the same thing. The module driver for ISA systems, `isapc`, is thus very simple indeed. The same applies to the PC-104 bus, which is logically equivalent to the ISA bus.

## 6.9 Interrupts

IIO does not currently support interrupts. This is a considerable restriction, because it means that modules that depend on rapid service by the processor after external events cannot be used. It also means that channels that are inherently time-based, such as timers, or inherently asynchronous, like interrupters, cannot really be implemented.

There are several reasons for the restriction. Firstly, the best way of specifying the mapping between the hardware interrupt, interrupt level, vector number and so on is not clear, although something not unlike the address mapping resolution and resolution mechanism is envisaged.

Secondly, how this would translate into real, portable module driver code is also not clear. On vxWorks, for instance, functions can be directly connected to interrupts, so little is required. On UNIX the interrupt would have to be handled



by a kernel module, and translated into a signal to be delivered to an IIO process. If there are several processes using IIO, which one should it choose? The first IIO process to run in these systems is the one that does the module hardware initialisation, so it is the obvious choice, but what if it this initial process exits?

Thus the handling of interrupts within IIO raises some thorny issues which will require some further consideration.

## 6.10 Adam Module Channels

The ADAM serial-addressable modules are quite different from the bus-addressable modules normally dealt with by IIO module drivers. ADAM units connect to an RS-485 serial bus, and the computer reads and writes to the channels on the units by exchanging short ASCII messages with the units. Despite the differences, the ADAM units integrated quite neatly into the IIO system. Other commercial serial-addressable systems should fit in in a similar way.

### 6.10.1 Adam Module Interfaces

The RS-485 twisted-pair serial bus can be driven by the computer directly (although few have RS-485 serial devices built in), or through an ADAM interface unit, such as the ADAM 4520 (Appendix A.3), which interfaces the bus to a standard RS-232C serial port. Each RS-485 network can connect to a string of 256 ADAM units, each of which must be programmed to respond to a unique address between 0 to 255. The computer sends request messages with this address to the network, and the selected unit sends its result back.

Each serial network has an IIO module driver, which provides 256 `adam` channels, one for each possible address. This is a bit like the `ip` channels provided by IP carrier modules. The `adam` channels have a single operation, `iio_adam_message`, which exchanges a message with the unit they represent.

The `iio_adam_message` operation involves prepending the network address and appending a checksum, and writing the result to the serial network. The interface drivers do not access the serial device directly, but do so through the operating system, using the IIO serial calls to open, read, write and configure the port (see Section 7.6.8). They then read the response message from the unit, test the checksum, and return the message to the caller.

### 6.10.2 Adam Module Drivers

Each ADAM unit connected to the network, except for the repeater units, is also a module in the IIO sense, and each provides a number of real IO channels to the channel pool. For instance, the ADAM 4017 provides eight `adc16` channels, which are indistinguishable from `adc16` channels provided by a bus-addressable module, except they take slightly longer to access.

The modules are installed through the configuration file in the normal manner, and must follow the installation of the module driver for the serial interface. The address configuration of the ADAM unit is implicit in the `adam` address channel, which is a mandatory module parameter.

**Message Format.** The message formats are defined in the ADAM documentation. They vary somewhat across from unit to unit, but all consist of a leader character, a two-digit hexadecimal network address, command codes and/or data, and an optional two-digit hexadecimal checksum. The data can be in a number of ASCII formats, but hexadecimal is preferred.

Function `iio_operate_addr()` is used by the module driver to invoke the operation, because the user data is the address of a pointer to a message buffer.

The module provides the buffer, and writes in the body of the message, in the correct positions. The interface module driver adds the address and checksum parts of the message, and delivers it as described. The reply message is received and overwritten onto the same buffer, which the caller must decode.

**Installation Function.** The installation function for an ADAM module is much the same as those for other modules. The address channel is stored in the register structure, because it must be used later to communicate with the unit. Often a single module driver will support several similar units, and so the unit sub-type must also be stored in the register file. ADAM units appear to use a common input range code sequence across the whole product range, which is shown in Table A.1 on page 89. An array in the generic ADAM module driver code contains scale factor and unit information, which is used for pre-setting the channel properties.

**Initialisation Function.** ADAM units are self-initialising. They are also self-identifying, and the initialisation function should check the unit's identity against that in the configuration file. This is done by sending a `$AAM` message to the unit, which should return its model number, which can be compared against what it should be.

This is consistent with the general IIO approach of using the configuration file as the primary source of configuration information, confirming it using the self-identifying features of modules, where these exist.

**Operation Function.** The operation function simply assembles the appropriate message for the operation code, and decodes the result. There are several helper functions in the generic ADAM module code used to simplify the hexadecimal conversions required.

## 6.11 Adding New Channels and Operations

The current set of channel types and operation codes follows on from the set of modules currently supported by IIO, and is by no means closed. New modules will eventually need new channel types, and operation codes to go with them, and these can be added quite easily.

On the other hand, endless proliferation must be avoided, to prevent 'same but different' channel types or operation codes appearing. The IIO channel types are based around the kinds of IO hardware that is available, not the things it can be connected to. For instance, there is no point adding a new channel type `temp` for temperature sensors, since the sensor would almost certainly connect to the computer through an analogue-to-digital converter, so the existing `adc` type is sufficient.

If a new channel type is considered, a generic 'model' for the channel type should be developed, describing what the channel should do for each operation, extant or new. This should be generalised as much as possible, not based on the particular hardware module that has is being interfaced, so that other hardware offering the new channel type will also fit into the scheme.

### 6.11.1 Adding New Channel Types

The channel types are defined by the enumerative `IIO_CHTYPE`, which is defined in `internal.h`. The new channel can be added anywhere, but the dummy element `IIO_NCHTYPES` (the number of channel types) should be last.

Follow the same naming scheme: if the name of the new channel is `zog` then the new entry is `iio_ctype_zog`. The name of the new channel must be added to `iio_ctype_string[]`, in the matching position of this string array. The name is used for operation logging and channel list displays.

### 6.11.2 Adding New Operation Codes

Adding new operation codes is similar to adding channel types. The codes are defined in the `II0_OP` enumerative in `iio.h`.

The matching operation name list is `iio_opinfo[]` in `opinfo.c`. This is a structure array, with fields for the operation name, a symbol for the data direction, and the argument type (inward, outward or bi-directional). These codes should be self-explanatory.

After adding either new channel types or operation codes, the IIO library must be re-compiled. If the library has been re-compiled, most IIO applications will also need to be re-compiled.

## 6.12 Errors

IIO uses a simple error-return code system, similar to the conventions of many other libraries and operating systems. There are three return statuses, already described in Table 4.3 on page 4.3, indicating either successful completion, an error, or a fatal error.

**Function Calls.** Within the library, the return status of all function calls are checked. If a non-zero status is detected, the calling function immediately returns this status to its caller, and so on, collapsing the call stack and eventually returning the response to the application program, which should also check and act on the status. The only difference between errors and fatal errors is that fatal errors print messages during the stack collapse, indicating the code filenames and line numbers. This is not quite as good as the information a debugger might furnish, but most of the time the source of the error can be traced.

This useful feature is provided by the `iio_eret()` macro. This encloses all function calls, and tests and act upon the return status, and at the same time avoids cluttering the code. It is defined, in `internal.h`, as:

```
#define iio_eret(S) \
    switch ((S)) { \
        case iio_status_ok: \
            break; \
        case iio_status_error: \
            return iio_status_error; \
        case iio_status_fatal: \
            iio_log( \
                "IIO: called from: file %s, line %d\n", \
                __FILE__, __LINE__); \
            return iio_status_fatal; \
    }
```

Note that the macro has no side-effects (apart from a potential function return). Function `iio_log()` is a `printf()`-like function that logs messages to the system logging stream, or `stderr`, as appropriate. There is a variant of the macro, `iio_fret()`, which ignores errors, but still returns in the case of fatal errors. This macro is used in cases where an error can be handled by the caller, and should not cause a stack collapse. (The C pre-processor replaces the special macros `__FILE__` and `__LINE__` with the filename and line number of the code file being compiled where the macro is expanded).

The only situations where these macros are not used, and the error handling code written out, is where resources such as held mutexes must be released first. Failure to do this may lead to a system lock-up.

**Originating Errors.** Where errors or fatal errors occur, either of the macros `iio_error()` or `iio_fatal()` should be used, in conjunction with a `return` statement. These macros are defined as:

```
#define iio_error(S) iio_return_error((S), __FILE__, __LINE__)
#define iio_fatal(S) iio_return_fatal((S), __FILE__, __LINE__)
```

The macros accept a static string, which should indicate the actual error. They are used like this:

```
if (addr > 0xffff)
    return iio_error("Address out of range");
```

This approach has been found to be simpler than using an ‘error number’, since there is no need to maintain a separate array of number-to-message-strings. The arrangement integrates into the UNIX error number scheme, however. If the error string is `NULL`, the UNIX error number `errno` is used to select the system error string, instead of the user one. This method should be employed when checking the status of operating system calls:

```
if (! (file = fopen("fred", "r")))
    return iio_error(NULL);
```

The error string (or a pointer to it), concatenated with the file and line number strings, is stored by the functions `iio_return_error()` and `iio_return_fatal()`. These functions always return error and fatal error status respectively. The message is, where practical, stored as per-thread data.

# Section 7

## Inside the IIO Library

This section is a guide to the internal workings of the IIO core library, intended for those who need to alter, add to or port it. It is not essential reading for someone wanting to add a new module driver: Sections 5 and 6 cover that.

Instead, the aim is to provide sufficient context for a programmer to effectively read the sources, and as such is a commentary to the IIO sources. Descriptions of most of the important data structures and functions are given. If the sources are not handy, the IIO header files are a reasonable substitute. They can be found in Appendix F.

### 7.1 General Practices

The overriding general practice in the IIO library is to pre-allocate, pre-compute, and pre-index. This follows normal real-time design practice, where resources are obtained ahead of time, so that the delay incurred does not affect critical real-time activities. It performs the maximum amount of work in system initialisation and in opening channels, so that the minimum need be done by the operation functions.

IIO also uses a generally object-oriented approach, in so far as this is practical using C. The data structures are to the fore: each structure has an associated set of functions (methods) that do things with or to it. On the other hand, data encapsulation is not carried through to the extent it might be if a real object-oriented language was being used. Functions ‘belonging’ to one type routinely access structure members of other types, at least for simple purposes. This avoids a plethora of data access functions which, in C at least, are very clumsy.

Another programming idiom heavily used in IIO is the registration and callback style, where function pointers are passed as function parameters, stored, and are subsequently called back. This technique is used, for instance, in the module driver interface (Section 5), to reduce the number of public symbols that the driver must provide: the rest of the entry points are registered at run-time.

#### 7.1.1 Processes and Systems

The biggest difference between the real-time operating systems IIO runs on is their approach to addressing contexts. The smaller real-time kernels, such as vxWorks or RTEMS, have a single, system-wide addressing context. All threads of control, known as *tasks*, share the same addressing space, so all global, static and dynamic variables are implicitly shared (only automatic variables are not). These systems are referred to as *shared-memory* systems.

UNIX-style systems, on the other hand, provide one addressing context per thread of control, which is known as a *process*. These systems are sometimes known as *protected memory* systems. Two processes running the same program do not share variables, unless explicit steps are taken to do so (using a *shared memory* block). Hardware registers must also be explicitly mapped into the process address space. Many modern UNIX systems, such as Solaris and LynxOS, extend the process model to permit multiple threads of control within each process.

IIO must run in all of these situations. On shared-memory systems, there is only one copy of each data structure, and hardware access is generally easy,

so there are few problems on these platforms. All that is required is standard mutual exclusion locks on non read-only shared structures.

On protected memory systems, each IIO-using process builds its own set of data structures. Because these are derived entirely from information in the configuration file, they will be identical. A few data structures, most importantly the module driver state structures, are placed in a shared memory block. Steps are taken to ensure that all IIO processes cooperating in this way use the same version of the configuration file and IIO library code.

It could be argued that all IIO variables could be shared in this way, but this is not practical. While the data in shared memory blocks is the same for all processes, the address of the block in each process is not necessarily the same. The ‘block’ may also be a number of discontinuous blocks. Thus, pointers cannot be shared, which eliminates many useful data structure types, such as linked lists.

### 7.1.2 Dynamic Allocation

The IIO library is generally dynamic in its approach. All important data structures are allocated off the process or system heap, or from a shared memory block. That said, it is not a continuous user of `malloc()` and `free()`, because it constructs all its data structures during initialisation, which are thereafter left unaltered. String data is almost always duplicated into heap memory, and linked to the relevant data structure.

IIO generally does not de-allocate its data structures. On protected memory systems, this does not matter: when the IIO-using process exits, the system reclaims the memory. On shared-memory systems, such as vxWorks or RTEMS, the lifetime of the application is generally the lifetime of the whole system, so it does not matter either.

### 7.1.3 Data Structure Conventions

Almost all IIO data structures form part of homogeneous singly-linked lists. The first element in these structures is always the `next` pointer, which is `NULL` for the last element in the list. The lists are always ordered, by using a common search-and-insert function `iio_sll_insert()` to expand the list. This function accepts a pointer to the list head pointer, a pointer to the new element, and a criteria function, which must return a value similar to `strcmp()`.

Most data structures have a magic number, generally the second element. This is of an enumerative type `IIO_MAGIC`, whose values are formed from four ASCII bytes, with unique values for each data structure type. Functions that allocate data structures set this value. Functions that accept pointers to these structures usually test the magic number, and return a fatal error if it is wrong. The values are formed from ASCII characters so that the data structure type can be identified in a memory dump, a feature that has happily not been required to date.

### 7.1.4 Coding

The IIO library is coded in C. Symbol-space pollution is avoided by prepending `iio_` to all external and static symbols (functions, variables, function-like macros and enumerative values), which are all in lower case and use underscores between words. Similarly, `IIO_` is prepended to all types, tags and preprocessor constants, which are in capitals.

In general, symbols follow a left-to-right precedence rule. The second word is usually the name of the source module and/or the data structure to which the symbol belongs or is associated with, the third is the operation, and following words are usually qualifiers. Thus, `iio_init()` will be found in `init.c` in `iio/src`. There are a few exceptions, however, in the interests of keeping the length of commonly-used functions reasonable.

### 7.1.5 Portability

Every effort has been made to make the IIO library and its module and chip drivers portable across computers, compilers and operating systems. The library has been proven on big-and little endian processors, on UNIX systems and on small real-time kernels.

Portability across operating systems is achieved by routing all calls to operating system and C library functions through interface functions. There is one set of such functions for each operating system, limiting the extent of system specificity to a reasonably small module. Frequent use of preprocessor directives is avoided. Section 7.6 discusses this in detail.

Portability across processor types is achieved by fairly strict ANSI C usage. Processor endianness, in the few cases where it matters, is dealt with by the address resolution mechanism (Sections 5.3.4 and 6.4). There may be some implicit assumptions that integers are not less than 32 bits wide.

IIO has not been seriously tested on compilers other than the GNU C compiler, `gcc`. Few `gcc`-specific features are used, they are not critical, and are enclosed in `gcc`-exclusive preprocessor directives. In any case, on many systems nowadays `gcc` is the approved, or sometimes only, ANSI C compiler.

## 7.2 Operational Phases

There are two distinct operational phases of the IIO library, and thus IIO-using applications: *initialisation*, and everything else.

Initialisation is performed by the function `iio_init()`, and is by far the most complicated thing done by the IIO library. The process is driven completely by the IIO configuration file, and not at all by the application program. In general terms, it involves:

- collating module driver information structures
- obtaining a system-wide exclusion lock to cover the installation phase
- creating or attaching to shared memory structures
- opening and parsing the configuration file
- calling installation functions of the modules
- collating lists of available channels
- calling initialisation functions of the modules
- releasing the exclusion lock.

Once initialisation is complete, control returns to the application program. Activity in the IIO library is then driven by the program, which will open channels by calling `iio_open()`, and use them with `iio_operate()` or a variant.

Opening channels can of course be interspersed with operations on already open channels. However, an application will typically open all the channels it requires during *its* initialisation phase, and operate on them during its running phase. Thus, IIO arguably has three phases of operation, with the second two, channel opening and channel operations, rather less distinct.

The following sections will work through these two or three phases in more detail, with particular reference to the data structures that are constructed. Figure 7.1 attempts to show all these data structures and their overall relationships.

## 7.3 Data Structures and the Initialisation Phase

The initialisation phase is commenced by the user calling `iio_init()`. This only calls `iio_osinit()`, which must perform operating system specific initialisation, and then call `iio_init_iio()`, which does the majority of library initialisation.

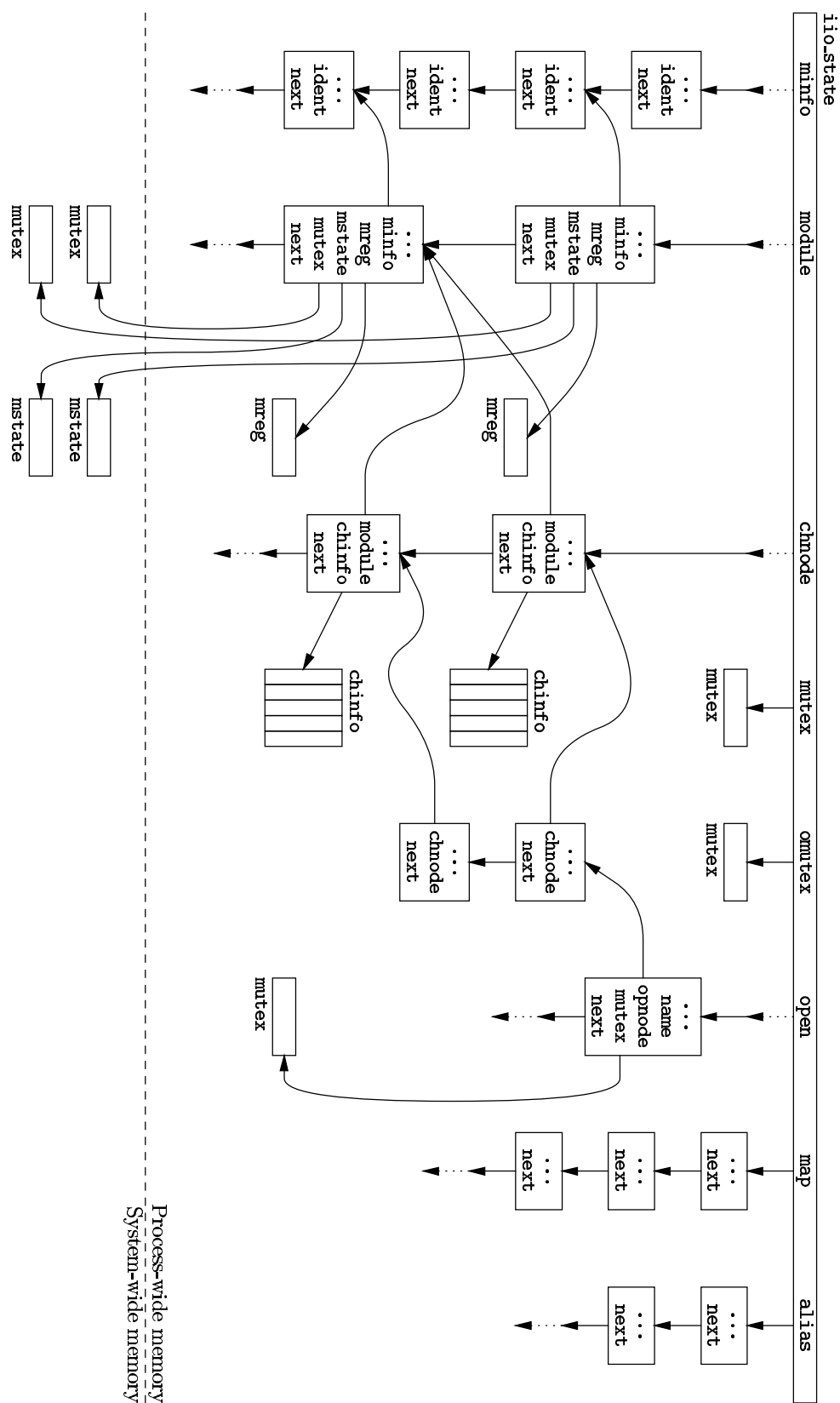


Figure 7.1 Relationship between Major IIO Data Structures



The details of operating system specific initialisation are dealt with in Section 7.6: these vary somewhat, but encompass establishing a system-wide exclusion lock, attaching or creating the shared memory block, and determining if this is an subsequent or initial IIO process (i.e., on protected memory systems, if there is another IIO-using process in the system or not).

### 7.3.1 State block, IIO\_STATE

Function `iio_init_iio()` first creates one state block, `iio_state`, one of the few global variables. ‘State block’ is something of a misnomer, because it has nothing to do with the module state structures (Section 5.3.2). This block contains the head pointers for all the major data structure linked lists, `iio_state->map` (the logical/physical map), `iio_state->minfo` (the module information list), `iio_state->module` (the installed module list), `iio_state->chnode` (the channel node list), and `iio_state->open` (the open channel list). It also contains two process-wide mutexes `iio_state->mutex` and `iio_state->omutex`, which are used to protect the linked lists (`iio_state->omutex` is for the open channel list `iio_state->open`, and `iio_state->mutex` is for the rest).

Function `iio_state_init()` creates and initialises the state block. It returns a fatal error if called more than once per process.

### 7.3.2 Module Information list, IIO\_MINFO

The module information list is a linked list of `IIO_MINFO` structures, one for each model of module the library has a driver for, and headed by `iio_state->minfo`. This list is built by `iio_minfo_call()`.

This function accepts the pointer to the driver array passed to `iio_init()`, frequently the array `iio_standard` (Sections 4.9 and 5.6). This NULL-terminated array contains pointers to the identification functions of each module driver (Section 5.2). `iio_minfo_call()` calls these functions; in turn they call back `iio_minfo()`, which inserts new `IIO_MINFO` structures into the module information list using the information they pass. Drivers which support a number of models call `iio_minfo()` more than once, and so have more than one `IIO_MINFO` block. Once the module information list is built, the driver array is no longer required.

The `IIO_MINFO` structures contain the model ident code, the manufacturer and module number, the RCS revision number of the module driver, and pointers to the installation and initialisation functions of the module driver. The module information list can be viewed from the IIO interactive shell (Appendix B) using the `minfo` command.

### 7.3.3 Configuration File Parsing

The configuration file is then opened (if it was not previously) and parsed using `iio_tfile()` or a variant. ‘`tfile`’ stands for ‘token file’. The parser accepts an open file pointer (type `IIO_FILE`), a filename (which it opens and parses), or a string (which it parses directly).

Each line of the file (excluding comments introduced by ‘`#`’, but including continuation lines introduced by ‘`\`’) is split into white-space tokens. Quoted strings ‘`"..."`’, are a single token, although the quotes are removed. Pointers to the zero-terminated token strings are assembled into an argument array `argv[]`. The first token in the line is pointed to by `argv[1]`. `argv[0]` points to a special token containing the filename of the file being parsed, and the current line number, which can be used for generating error messages.

Once each line is tokenised in this manner, the parser calls back an ‘execution function’ provided by the caller, which actually interprets the tokenised argument

array. This decouples the parser from the rest of the library, so it could be used to read other files of similar syntax.

Usually the execution functions, or the functions they call, use the `iio_arg()` function and its variants, described in Section 5.3.3. These functions blank out arguments that they have interpreted, so that the argument array can be partially interpreted in different parts of the library. The parser's buffer space is overwritten for each new line, so arguments that need to be preserved should be duplicated.

`iio_init()` invokes `iio_tfile()` with `iio_config_exec()` as the execution function. This function tests the first token, and calls `iio_module()` in the case of a `module` directive (Section 3.2), `iio_channel()` in the case of `channel` (Section 3.4), or `iio_alias()` in the case of `alias` (Section 3.3).

### 7.3.4 Installed Module list, `IIO_MODULE`

Function `iio_module()` adds a new module to the installed module list, which is a linked list of `IIO_MODULE` structures headed by `iio_state->module`. It is invoked in response to a configuration file `module` directive, and is passed the `argv[]` array built by the parser.

The model ident of the module to be installed should be in the second argument (`argv[2]`). The module information list `iio_state->minfo` is searched for a module of this name, and if found, a new `IIO_MODULE` structure is allocated. The existing installed module list is searched to find out how many modules of this model are already installed, so that the module sequence number can be assigned. All this is done by `iio_module_create()`.

The `IIO_MODULE` structure contains a pointer to the module information structure, pointers to the register and state structures of the driver, the module sequence number, and the system-wide module mutex, which is later used to protect the module hardware and state structures from simultaneous access by different tasks or processes.

The new module structure is not actually inserted into the installed module list until the module installation and initialisation has been successfully completed. The search and insert function `iio_sll_insert()` is used, with `iio_module_cmp()` as the criteria function. This orders the list in alphabetical then sequence number order. The installed module list can be viewed from the IIO interactive shell (Appendix B) using the `module` command.

**Module Installation and Initialisation.** The installation function of the module driver is called, using the pointer in the module information structure. A pointer to the new module structure is passed, along with the argument array. The installation function, as described in detail in Section 5.3, decodes the module parameter arguments, and proceeds to call back functions in the IIO core.

One of the things it does is to allocate the register and state structures for the module, using functions `iio_module_reg()` and `iio_module_state()`. The register structure is allocated from per-process memory, while the state structure comes from system-wide shared memory. Pointers to both of these are stored in the module structure.

The installation function also calls back functions to map and resolve addresses, open channels, and register the channels the module provides. The data structures these create are discussed shortly.

If the installation function returns successfully, `iio_module()` then calls the initialisation function, but only if this process is an initial IIO process, or this is a shared-memory system. This status is determined in `iio_osinit()` and passed through `iio_init_iio()`, eventually to be stored as a flag in `iio_state`. The initialisation function actually accesses the module hardware, resets it as necessary, and fills out and synchronises the module state structure, if any.

At the end of `iio_module()`, `iio_arg_remnants()` is called, accepting the argument array. The array should be filled with blanks (pointers to the empty string `iio_arg_blank`, or `""`) because all the arguments should have been interpreted. Anything still there is probably a misspelt option, and `iio_arg_remnants()` prints an error message.

**Module Logging and Aliases.** The two standard module directive options, `-log` and `-alias` (Section 3.2), are actually decoded by `iio_module()` before the module installation function is called. The former sets a boolean flag in the module structure. The latter causes a module alias to be inserted into the alias list, using `iio_alias_insert()`.

### 7.3.5 Memory Map list, `IIO_MAP`

The IIO address mapping and resolution system has been described in detail, at least from the point of view of its users, module drivers (Section 5.3.4). Section 6.7 described it from the point of view of the address space module drivers that implement an important part of it. Here, the core library functions `iio_map()` and `iio_resolve()` are described.

The previous discussions suggested that mapping, that is, making module hardware registers visible to the program, comes before resolution, which is finding out what address to use to access a mapped register. In fact, the functions are interdependent, as `iio_map()` uses address resolution to work out what addresses to map.

**Mapping.** `iio_map()` accepts the physical base address and size of the segment encompassing the module registers, along with an open IIO channel descriptor and address space operation code identifying the physical space. It adds the size of the segment to the base to obtain the top and bottom addresses, and calls `iio_operate_addr()` to resolve these separately through the chain of address space drivers described in Section 6.7. This results in two processor-physical addresses, resolved as far as the memory management unit (or  $A_1$  in Figure 5.1 on page 38). The difference in the two addresses gives the size of the partially-resolved segment, which may be different to the physical size.

It then calls `iio_map_new()`, which accepts the partially-resolved base and size, and first calls `iio_map_ptov()` to see if there is already a mapping covering the given range and of the same type (memory or port). `iio_map_ptov()` does so by searching the mapping list, a linked list of `IIO_MAP` structures headed by the pointer `iio_state->map`. If a map exists, nothing more needs to be done and the function returns.

If there is no mapping covering the full range, a new one is requested from the operating system. If the address space operation code indicates the mapping is to be into memory (determined by function `iio_map_type()`), `iio_map_new()` uses `iio_shmap_alloc()`. This takes the requested processor-physical address and size, and returns the actual physical address and size, and the corresponding logical (virtual) address. The actual physical base and size may be different because MMUS map memory in pages of a certain size. The returned information is stored in a new `IIO_MAP` structure, which is inserted in the mapping list.

If the address space operation code indicates a non-mappable address space, `iio_map_new()` calls the operating system function `iio_port_alloc()`. This makes the port addresses visible to the process, if the operating system actually requires it (although this is not ‘mapping’ in the strict sense, it is an analogous operation). The mapped addresses can be later accessed by module drivers the `iio_port_set()` and `iio_port_get()` functions (Section 5.5). A new `IIO_MAP` structure is also inserted in the mapping list.

The memory mapping list can be viewed from the IIO interactive shell (Appendix B) using the `map` command.

**Resolution.** `iio_resolve()` accepts a register physical address and in the same way as `iio_map()`, partially resolves it to a processor physical address, and then calls `iio_map_ptov()` to turn this into a logical address. `iio_map_ptov()` searches the mapping list, and finds a previously created mapping that covers the physical address required (the size of the register is counted as well). The offset of the physical address from the base physical address of the mapping will be the same as the offset of the logical address from the base logical address, so the fully resolved logical address can be determined by simple arithmetic.

Note that `iio_resolve()` does not invoke an operating system request: all it uses is the results of previous mapping requests. This technique is used because operating systems often do not provide an equivalent to `iio_map_ptov()`, only a logical-to-physical map.

These functions still operate in the same way on shared-memory systems, where there is frequently no MMU, or where the operating system effectively disables it. The map request function `iio_shmap_alloc()` simply returns its input arguments, mirroring the 1:1 map the MMU implements in these cases.

### 7.3.6 Channel Node list, `IIO_CHNODE`

The other important function called by the module driver installation function is `iio_chnode()`. This function registers the channel ranges that module provides. Each call of `iio_chnode()` inserts a new `IIO_CHNODE` structure into the list headed by `iio_state->chnode`. Each 'channel node' represents a contiguous range of channels of the same type and width, from the same module and with the same operation function. This is the information that is passed to the `iio_chnode()` function by the module driver.

The list is in global generic channel name order, but contains sufficient information to allow a channel name expressed in any of the four forms to be located. This is in the `seqno[]` structure member, which contains the sequence number of the first channel in the channel node in each of the four numbering sequences. (The array is indexed using the `IIO_NFORM` enumerative). These sequences are compiled by `iio_chnode_new()`, which searches the current members of the channel node list. These numbers are used when a channel is opened.

The channel node list can be viewed from the `IIO` interactive shell (Appendix B) using the `chnode` command.

**Bitwise Digital Channels.** `iio_chnode()` singles out digital channels of types `do`, `di`, `dio`, `oco` and `ocio` for additional treatment. These are inserted into the channel node list by `iio_chnode_new()` in the same way as other types.

As mentioned in Section 2.3.3, the channel name decoding mechanism is reused to resolve the bit-channels onto their underlying real digital channels. For function `iio_chnode()`, this means extra channel nodes are inserted, and two extra fields of `IIO_CHNODE` come into play. For each digital *channel*, a bitwise digital *channel node* is also inserted, using `iio_chnode_new()`.

This node has as many 1-bit channels as the original channel has bits. The channel type is selected to match the original channel. Thus, for a `di16` digital channel, a node of sixteen `bi` channels is inserted. (It is not permitted for a module driver to directly insert channels of these types).

The additional fields `rchnode` and `rseqno` of these additional channel nodes point to the underlying 'real' channel node and the local sequence number of the underlying real digital channel. Thus, when the bitwise channel is opened, the underlying channel, module and operation function can be easily located.

### 7.3.7 Channel Info arrays, `IIO_CHINFO`

Each channel node (including those for bitwise-digital channels) also contains a pointer to a channel information array, with one element of type `IIO_CHINFO` for each simple channel in the channel node. This structure contains the channel properties mentioned in Section 2.7.

The properties of individual channels can be set using functions such as `iio_chnode_linear()`, `iio_chnode_limits()`, and so on. The channel properties are initialised by `iio_chnode_new()` to sensible defaults. Module drivers subsequently alter these to comply with module hardware configuration.

Users can further alter them using the `channel` directive in the configuration file. This is handled by `iio_channel()`, which calls `iio_chnode_linear()` and so on as required.

### 7.3.8 Alias list, `IIO_ALIAS`

The alias list is a list of `IIO_ALIAS` structures, headed by `iio_state->alias`. The structures contain name-value string pairs and an alias type enumerative `IIO_ATYPE`.

The `alias` directive configuration file invokes `iio_alias()`, which accepts the argument token array. This calls `iio_alias_insert()` which actually creates a new `IIO_ALIAS` structure, and inserts it in the list using `iio_sll_insert()` as normal.

## 7.4 Data Structures and the Open Function

Following the completion of configuration file parsing, the system-wide exclusion lock, however implemented, is released. Function `iio_init()` returns to the application program, which can proceed to open and use channels.

Function `iio_open()` opens a channel. ‘Opening’ in the IIO sense involved constructing a set of data structures which link to information that will be needed when the channel (or channel range) is operated upon, so that operations can be fast. Opening a channel does *not* involve accessing the module driver: module drivers do *not* (and should not) know what channels then user has open.

### 7.4.1 Open Channel Structure, `IIO_OPEN`

This structure, and at least one `IIO_OPNODE` structure (described below), is allocated when the application program calls `iio_open()`. The channel descriptor, type `IIO`, is a pointer to this structure. All the `IIO_OPEN` structures link into the open channel list, which is headed by `iio_state->open`, and is ordered by the channel name used to open the channel (which may be an alias). Since the open channel list is not read-only after the end of the initialisation phase, it has its own process-wide mutex, `iio_state->omutex`.

The open channel structure contains the channel mutex, which is used to protect operations if the open channel is a channel range. The `number` element gives the number of channels in the range. If the open channel is just a simple channel, `number` is 1 and the mutex is not used. The structure also contains a duplicate of the name by which the channel was opened: this is used when logging channel operations.

**Name expansion.** Before constructing the `IIO_OPNODE` list, described next, `iio_open()` expands the channel name. This is done by calling `iio_namex()`, which returns the component parts of the channel name in an `IIO_NAMEX` structure. This structure is not dynamically allocated, and exists simply to tidy up the interface to `iio_namex()`.

Firstly, `iio_namex()` looks up the given channel name in the alias list, looking for a matching global name alias using `iio_alias_find()`. The result, or if none, the original name, is then copied to a buffer for destructive parsing.

If the string contains a `:` character, it is in a local channel name form, so it is split in two. The alias list is then searched for a module alias matching the first half, and the result, if any, is substituted. This should be in the form of a module ident (model ident plus module sequence number). The installed module list is then searched for this combination, and if found the `IIO_MODULE` pointer is started in the `IIO_NAMEX` structure. Similarly, the alias list is searched for the second half of the name, and the result, if any, substituted. Then, `iio_namex_chan()` is called to break this local channel name into the channel type, width (if specified) and local sequence number.

If the name string did not contain a `:` character, it must be a global channel name, and it is broken up using `iio_namex_chan()`. In this cases, the module the channel comes from is not known until later.

Note that `iio_namex()` does not test for the existence of the channel; it merely determines the name's form and splits it into its constituent parts, which are returned in the `IIO_NAMEX` structure.

#### 7.4.2 Operation Node Structure, `IIO_OPNODE`

The `IIO_OPNODE` structure contains the information that relates the channel or channels the user has opened to the channels provided by the modules, through the `IIO_CHNODE` structures described previously. The operation node list is headed by the `opnode` member of `IIO_OPEN`.

There is exactly one `IIO_OPNODE` structure created for each `IIO_CHNODE` structure that covers, in full or in part, the channel range of the open channel. Simple channels will require one operation node; channel ranges require one or more. The `IIO_OPNODE` structure contains a pointer to its `IIO_CHNODE`, and a number of sequence number indices: **first** is the local sequence number of the first channel of this operation node in the corresponding channel node; **number** is the number of channels in the operation node, and cannot be larger than the channel node's **number**. **index** is the index into the user's data array (the array passed to `iio_operate()`) of the first channel in corresponding to this operation node.

**Operation Node Assignment.** Once `iio_open()` has successfully split the name into its constituent parts using `iio_namex()`, it searches the channel node list to find the node or nodes which match the desired channel. The first match is against channel type, and then channel width, or module, or both, depending on the form of channel name originally given to `iio_open()`.

Matching channel nodes are then tested to see if their sequence numbers intersect those desired. The start sequence number used in this comparison comes from the channel node `seqno[]` array, which was computed when the channels were registered. The end sequence number is simply that number plus the number of channels in the channel node.

These start and end numbers are compared with the *desired* start and end sequence numbers. There are thus four comparisons with four binary results: these results are combined into four bits of an integer to form the code **which**. **which** indicates the relative disposition of the two sub-ranges. There may be no intersection at all or several kinds of intersection.

which is sorted out in a **switch**-statement, which creates operation nodes with different **first**, **number** and **index** parameters for the different kinds of intersection with the current channel node. These are appended to a list that eventually attaches to the **IIO\_OPEN** structure.

The intersection kinds also indicate whether *any* operation nodes have been created (flag **started**), and when all the required operation nodes for the given range have been created (flag **finished**). When **started** and **finished** are both true, the searching of channel nodes can cease.

**Channel Logging.** At the end of **iio\_open()**, the **log** flags in all the operation nodes of the newly opened channel are assigned. The flag is in **IIO\_OPNODE** because this structure is the only one passed to the module driver data access functions (Section 5.5) which actually do the logging. The flag's value is the logical OR of the logging flags from **iio\_open()**, any of the modules, or any of the individual simple channels involved in the new channel.

Finally, if the new open channel has more than one operation node, a process-wide channel mutex is created (if there is only one node, the module mutex suffices). The **IIO\_OPEN** structure is inserted into the open channel list (protected by the **iio\_state->mutex** mutex), and is then returned to the application program (where its type is the equivalent **IIO**).

## 7.5 The Operate Function

The operation function, **iio\_operate()** and its variants, do not create or significantly alter any data structures. Operations essentially involve taking the channel and module mutexes, then walking through the operation node list, calling the module driver operation function once for each operation node. It is also responsible for taking the channel and module mutexes, so that operations are thread-safe.

Bitwise-digital channels are much the same, only the manipulation of the user data is somewhat different.

**Operation Function.** The six forms of the operation function

- **iio\_operate()** (integer user data)
- **iio\_operate\_real()** (real user data)
- **iio\_operate\_addr()** (address user data)
- **iio\_operate\_in()** (in-only integer user data)
- **iio\_operate\_inreal()** (in-only real user data) and
- **iio\_operate\_inaddr()** (in-only address user data)

are simply front-ends for the main operation function, **iio\_operate\_call()**.

**iio\_operate\_call()** first logs a message, if logging is on, takes the channel mutex, if any, and then checks if there is an **rchnode** element in the first operation node's channel node. If so, it calls **iio\_operate\_bitfield()**, described later, to do the operation, as this must be a bitwise-digital channel.

**Operation Node Scratch-pad.** Otherwise, it loops through the operation node list. Before calling the module driver operation function, however, it sets up a few scratch-pad variables in the operation node structure, **udata**, **base**, and **op**.

This scratch-pad is a bit ugly, but is the simplest way of transmitting this data through the module driver and into the data access functions, which is where it is actually needed. The 'right' way would be to pass them as parameters to the module driver, along with the **IIO\_OPNODE**, and expect the driver to pass

them on to the data access functions. Using part of the operation node for these avoids this clutter. It is thread-safe, because this data is protected by the module mutex.

Note that the parameters for the module driver call are all, except for the operation code `op`, derived from the operation node anyway. Arguably these functions could have as few as two arguments, `opnode` and `op`. This would mean, however, that the drivers could not treat `IIO_OPNODE` as opaque, but would have to pull the fields they required out from it. It probably would not deliver much speed improvement.

**Data Access Functions.** The six data access functions

- `iio_data_get()` (get datum as integer)
- `iio_data_get_real()` (get datum as real)
- `iio_data_get_addr()` (get datum as address)
- `iio_data_set()` (set datum from integer)
- `iio_data_set_real()` (set datum from real) and
- `iio_data_set_addr()` (set datum from address)

are called back by the module driver operation function to access the user's data. `iio_operate()` and its variants do not do anything to this data, except pass the pointer to it through. All the data transformations (array indexing, the scale and offset, limiting, and logging) are performed by the data access functions, on the demand of the operation function.

The type qualifiers (`_real`, `_addr` and the implicit `_int`) of the data access functions indicate the form the *driver* wants (or has) the datum in. This contrasts with the similar qualifiers of `iio_operate()`, which indicate the form the *user* has (or wants) the data in. This latter type is indicated by the `IIO_UDATA` enumerative type, which is given to `iio_operate_call()` which puts it into the `udata` scratch-pad member of `IIO_OPNODE`.

Thus, the data access functions know the user data type, so they can correctly index the user data array to get or set the datum, and they can apply the correct data conversion. Address data, of course, cannot be meaningfully converted, and so the data access functions return an error if this is attempted.

**Bitwise-Digital Channels.** Function `iio_operate_bitfield()` is used for bitwise-digital channels. It is similar to `iio_operate_call()`—in that it works through the list of operation nodes—but unlike `iio_operate_call()`, it *does* transform the user data, and it does interpret the operation code. It effects the desired operation by performing a sequence of read and write operations on the underlying ordinary digital channels to which the bitwise channel or channel range corresponds. Bitwise-digital operations are essentially a layer on top of real digital channel operations: this makes module driver simpler, as they do not need to have any bit-twiddling code in them.

For a bitwise channel operation, the user data is a single unsigned integer, rather than an array (it is really a bit-array, of course). So, for write operations, `iio_operate_bitfield()` gets the user data once, by calling `iio_data_get()`, just like a module driver operation function would do. At the end of a read operation, it sets it using `iio_data_set()`. This means the bit-field datum may be subject to the normal scale and offset factors, if the application program called `iio_operate_real()` to invoke the operation.

Each operation node represents part of the datum bit-field that must be read from or written to part of a real digital channel. This is so because the channel nodes from which the operation nodes were generated are this way. For each operation node, different mask and roll factors are computed from the sequence number usually used to index the user data array.



For read operations, the underlying digital channel is read, by calling the module driver operation function in the *real* channel node. The scratch-pad data in the operation node was set so the the data access function that the driver will call will deposit the data unmodified into a local variable `cdata`. The data is masked and rolled into the return value.

For write operations, things are a little more complicated, because the new data must be written into some of the underlying channel bits, without disturbing adjacent bits. Thus, the channel has to be read first, again into `cdata`, the new data masked and rolled in, and the result written back.

## 7.6 Operating System Interactions

Unfortunately, operating systems do not provide a common interface to application programs, notwithstanding standards such as POSIX. In any case, there are differences in operating style and philosophy that fall well outside the ambit of such standards. In order to minimise the impact of such differences IIO uses an absolute minimum of operating system and C library facilities, and funnels all such calls through an operating system specific module.

This approach eliminates the `#ifdef` directives that frequently make low-level interface code almost unreadable. There are no operating-system header files included the by IIO library code or its header files (except, of course, the operating system specific modules themselves, and for `stdarg.h`). The approach is arguably unnecessarily strict, and it is not necessary for IIO-using applications to do this, but it has been very helpful in revealing operating system portability issues.

The operating system specific modules are found in the `iio/src/os` sub-directory. There is one for each operating system IIO provides support for: `lynxos.c`, `vxworks.c`, `solaris.c`, and so on. The `Makefile` selects the correct module using the operating-system name from the platform script (Appendix C.3).

These modules are expected to provide all the operating system functions, which are prototyped in `internal.h` (Appendix F.2), and described in the following sub-sections. However, many of these functions are merely wrappers for fairly generic operating system or C library functions. In these cases, where the same code goes for all or most operating systems, the functions are in separate C files, which are `#included` into the main operating system specific module where required.

Some operating systems do not easily provide all the facilities IIO requires. This mostly the case with the shared memory/hardware mapping functions on standard UNIX system, such as SunOS, Solaris and Linux: to uses these features the user would require access permissions not normally given to ordinary users. Thus, these functions simply do nothing, which results in a protection violation later. Given this, it would seem there is little point in providing operating system specific modules for these systems at all. However, it is worthwhile, because much testing can be done on there system even without hardware accesses, and in any case IO modules that do not have memory mapped hardware registers, such as serial-addressable modules, can still be used on these systems.

### 7.6.1 Initialisation

As described in Section 7.3, `iio_osinit()` is supposed to carry out any operating system specific initialisation, and then call `iio_init_iio()` within system-wide exclusion lock, informing it if this is an initial IIO-using process or not.

**LynxOS.** On LynxOS, the exclusion lock is obtained by opening the IIO configuration file (using the IIO file access functions described in Section 7.6.7) and

then using `flock()` to lock it. Within this system-wide lock, the modification time of the file is obtained.

Then, the shared-memory block is established, first by attempting to *remove* it. If this fails with a ‘device busy’ error, then the block exists and there is another IIO-using program running (attempting to remove a busy block causes no harm). If the block is successfully removed, or there was not one in the first place, then is the only (initial) IIO process. The flag `iio_first` indicates this status: the flag is only valid during the life of the exclusion lock.

In either case, the first structure in the shared block is the sentinel, type `IIO_SENTINEL`. This is allocated using `iio_shmem_alloc()`. This structure contains the modification time of the IIO configuration file and the build date of the IIO library. The latter comes from `iio_timestamp`, which is re-computed by the IIO `Makefile` every time any part of the library is built (Appendix C.5). For an initial IIO process, the time-stamps and a magic number are written into the sentinel; for subsequent processes they are tested, and any discrepancy causes an error return. After calling `iio_init_iio()`, the exclusion lock on the configuration file is dropped.

**Other Systems.** On shared-memory systems, such as vxWorks, there is no sentinel block and there is no need for any other operating system specific initialisation. On these systems, the configuration ‘file’ may be a string in memory, as sometimes files are not available. Other UNIX systems also do not have much beyond the requisite call of `iio_init_iio()`, for the user privilege reasons given above. In these cases, where process-shared structures are not provided, there is no point in a shared configuration or locked file either, so the file `./iio.conf` (that is, in the process’s working directory) is used instead.

### 7.6.2 Cleanup

Function `iio_done()`, which is supposed to be called when the user has finished with IIO, invokes `iio_osdone()` to perform operating system specific cleanup. Symmetrically, this should invoke `iio_done_iio()` which dismantles the process’s IIO data structures.

**LynxOS.** On LynxOS, the configuration file is once again opened and used as an exclusion lock. However, the file is not parsed, and nothing depends on whether this is the initial process or not. After calling `iio_done_iio()` the shared structures are dismantled with `iio_shmap_done()` and `iio_shmem_done()`. These eventually detach the process from the shared memory blocks, but do not delete them (other processes may be using them). When the last process detaches, the LynxOS kernel will delete the shared blocks.

### 7.6.3 Mappings

Mappings of areas of processor-physical memory into process memory (where there is a distinction) are provided by `iio_shmap_alloc()`, which is called by `iio_map()` (Section 7.3.5). They are discarded using `iio_shmap_free()`, and `iio_shmap_done()` discards all such mappings.

**LynxOS.** On LynxOS, shared mappings and shared memory (discussed below) are essentially the same thing. The native mapping calls are used (`smem_create()` and so on) rather than the POSIX ones, as the latter do not seem to work on earlier versions of LynxOS.

Function `iio_shmap_alloc()` rounds down the processor-physical base address, and rounds up the size of the segment, to the nearest multiple of the page

size (obtained previously by `getpagesize()`). A name for the mapping is generated, using a string and a serial number, so that each mapping has a unique name, but so that each IIO process will come up with the same sequence of names. `smem_create()` is used to create the mapping. If the mapping does not exist, it is created it and attached it to this process: if it does already exist, it is simply attached.

Port space does not need to be mapped on LynxOS, so `iio_port_alloc()` simply checks that the process is owned by the super-user. The six port access functions

- `iio_port_set8()` (8-bit port write)
- `iio_port_set16()` (16-bit port write)
- `iio_port_set32()` (32-bit port write)
- `iio_port_get8()` (8-bit port read)
- `iio_port_get16()` (16-bit port read) and
- `iio_port_get32()` (32-bit port read)

are simply wrappers around the inline functions `_outb`, `_outw`, `_outl`, `_inb`, `_inw` and `_inl`.

**Shared Memory Systems.** On shared-memory systems, such as vxWorks, all memory is implicitly shared between tasks, and no special actions are required to ‘map’ particular areas to make them visible. Thus, the shared mapping functions are all no-ops.

If vxWorks is used with the vxVMI (virtual memory) option, this isn’t completely true: some very high addresses in processor-physical memory aren’t mapped, so that there is some virtual space left to make mappings of A32 VMEbus areas (or equivalent) into. At present IIO does not issue the vxVMI calls if presented with such an address. In practice this has never presented difficulties, because industrial IO modules are generally wired for the addresses that are always mapped.

**Unix Systems.** On these systems the shared mapping and port functions are no-ops.

#### 7.6.4 Shared and Process Memory

Function `iio_shmem_alloc()` allocates system-wide shared memory blocks, which are used for module driver state structures, and other shared objects. Such memory is not mapped to specified processor-physical addresses, like the shared mappings described above. Function `iio_mem_alloc()`, by contrast, allocates process-wide memory, at least on systems where there is a distinction. In other words, it is a wrapper for `malloc()`.

Both of these have corresponding discard functions, `iio_shmem_free()` and `iio_mem_free()`. Shared memory has an additional `iio_shmem_done()`, which detaches all segments.

**LynxOS.** Shared memory requests of various sizes are allocated from shared memory blocks obtained from the system using the LynxOS call `smem_get()`. Each block is a multiple of `iio_pagesize` in size, and is given a name and serial number in the same way as the shared mappings. When each block is fully allocated, a new one is obtained. The base addresses of each block are recorded in an array (which is currently fixed in size) so that the blocks can be deallocated later.

The whole shared state scheme of IIO on LynxOS depends on all processes executing the same configuration file with the same library code, and so exactly the

same sequence of shared memory (and shared map) segments will be requested. This is guaranteed by the sentinel structure check.

Note that freeing an allocation from a shared memory block is not actually supported. IIO never actually needs to do this, so it is not an issue.

**Other Systems.** On shared-memory systems, there is no distinction between the two kinds of memory, so both allocation functions simply call `malloc()`. On UNIX systems, only process memory is provided, so again, both allocators call `malloc()`.

### 7.6.5 Mutual Exclusion Semaphores

IIO defines mutual exclusion semaphore objects (also known as a mutexes or monitors) `IIO_MUTEX` and `IIO_SHMUTEX`. As with memory, there are two forms of mutexes, system-wide and process-wide, where there is a distinction. The IIO mutex types are simply pointers to the operating system's mutex objects. Priority inversion-safe mutexes should be used if available.

Shared or non-shared mutexes are obtained with `iio_shmutex_alloc()` or `iio_mutex_alloc()`; taken using `iio_shmutex_grab()` or `iio_mutex_grab()`; released using `iio_shmutex_drop()` or `iio_mutex_drop()`; and discarded using `iio_shmutex_free()` or `iio_mutex_free()`.

**LynxOS.** POSIX thread mutexes are used for non-shared mutexes. In principal the same mutexes, allocated from a shared-memory block, should work as system-wide mutexes. However, on LynxOS 2.5 at least, this does not work, as the necessary mutex attribute code is not implemented.

For the time being, POSIX unnamed semaphores, allocated from a shared memory block using `iio_shmem_alloc()`, are substituted. Hopefully, future versions of LynxOS will properly implement shared mutexes.

**vxWorks.** The `IIO_MUTEX` and `IIO_SHMUTEX` types are simply cast to and from the vxWorks semaphore type `SEM_ID`, which is also a pointer, and the vxWorks semaphore primitives used. There is of course no distinction between system-wide and process-wide mutexes.

**Other Systems.** Where no mutex types exists, such as on UNIX systems, the mutex functions are all no-ops.

### 7.6.6 Register Probes

Function `iio_probe()` is used by module driver initialisation functions (Section 5.4) to test if a register (or something) is at the address that has been resolved for it.

**vxWorks.** The vxWorks function for probing addresses, `vxMemProbe()` is used directly by `iio_probe()`.

**Other Systems.** On UNIX-like systems, including LynxOS, a signal handler for bus errors and segmentation faults is established, and a read or write of the given address and width attempted. If the signal handler is invoked, a static flag is set, and `iio_probe()` returns an error. This is thread-safe on LynxOS, because `iio_probe()` is only used during module initialisation, which is inside the initial exclusion lock. There appears to be no way of probing for the existence of a register at an address in port space.

### 7.6.7 File Interface

Functions `iio_file_open()`, `iio_file_getc()`, and `iio_file_close()` are simple wrappers for the standard file access calls. Arguably, these functions are standard enough to use directly in IIO library code, but unfortunately the header files are not. These functions are only used to access the configuration file.

### 7.6.8 Serial Device Interface

The IIO serial device interface functions are somewhat more than simple wrappers, because operating systems are notoriously inconsistent about serial device setup, and even the standard procedures for doing so are fairly complicated and worth encapsulating anyway.

Serial devices can be opened in raw mode using `iio_tty_raw()`, or in line mode using the function `iio_tty_line()`. The serial port settings, such as baud rate, bits per character, and so on are specified using arguments to these opening functions, rather than later using an `ioctl()`-like wrapper.

Messages are written using `iio_tty_send()` and read using `iio_tty_recv()`. A read timeout would be a useful addition to `iio_tty_recv()`. The serial functions are used by the serial-addressable module drivers, such as those for the ADAM units.

**vxWorks.** Unfortunately vxWorks does not feature all the serial port settings expected by `iio_tty_raw()` and `iio_tty_line()`. An error is returned if such settings are requested.

### 7.6.9 Miscellaneous Functions

The remaining functions in the operating system specific modules are generally simple wrappers around C library functions. Their purposes are self-explanatory and their declarations are found in `internal.h` (Appendix F.2).



# Section 8

## Conclusion

The IIO library, while internally fairly complicated, presents a fairly simple and straightforward front to both the application program writer, through its C interface, and to the system integrator, through its single configuration file.

The library and application programs using it have been proven on real-time operating systems as diverse as LynxOS, which has a UNIX-like design, and vx-Works, which is a shared-memory kernel system. It has operated on hardware ranging from industrial VMEbus systems to laptop PCs.

### 8.1 Assessment

Any fair assessment of this project must do so in the light of the original proposal document, which is included verbatim in Appendix E.

Essentially all the project's objectives were met. The application program interface is more or less as proposed, and the configuration file contains essentially the same elements. Some of the less central elements suggested by the proposal, such as remote network access, have not been provided, because they are better implemented by an application that *uses* the IIO library.

The only important omission is support for hardware interrupts. While such an addition is relatively straightforward for shared-memory systems, such as vx-Works, where the operating system takes a minor role in interrupt handling, it is not easy for systems such as LynxOS. Section 6.9 discusses some of the reasons why: they are strongly related to the decision, taken fairly early on, to implement IIO as a user-level library, rather than a kernel module. A solution to this problem is required before IIO can claim to be capable of supporting all industrial IO hardware.

Nevertheless, the library has become an important tool in the Mining Automation projects conducted by the Division in Brisbane. It has already been deployed in the Dragline Automation system, where all industrial IO is routed through it. At the Preston site, where it was developed, no use has been made of it, as robotic and 'mechatronic' work there has essentially ceased.

The proposal document was vague on the duration of the project. However, early working versions of the library were delivered and tested by Brisbane staff within about four months of the proposal being accepted. Basic documentation was also provided at the time. Improvements and extensions were made up until February 1997 in a reasonably timely manner, and it is believed no other work was delayed through the lack of a promised IIO feature or bug. At about that time, development work ceased because the author was directed to other areas, and as a result the final documentation has been considerably delayed.

As a body of code, the library is written in a consistent and fairly clear style, is thoroughly commented, and is reasonably well documented. It should not prove a burden for anyone charged with its maintenance and extension. The procedure for expanding the library of module drivers is very detailed, and with luck this expansion will occur, making the library a more useful tool.

### 8.2 Critique

Given the assessment above, any serious critique of the IIO library as it presently stands must also criticise the proposal as well. This is fair enough, since the

library and the proposal were written by the same person (as was, of course, the assessment, and the critique that follows).

The difficulty with interrupt handling, and the design decisions that may have contributed to it, have already been mentioned. Other more fundamental design issues deserve some critical attention, in the light of the experience with the current implementation.

**The Form of the Interface.** The interface presented by the library to the application programmer is essentially an object-oriented one. Channels, which are initially extant but latent, can be opened, essentially constructing an access object. These objects have a set of methods (channel operation codes) which depend on the type of channel.

Unfortunately, C is not an object-oriented language, so any attempt to implement an object-oriented interface with it tends to lead to the writing of a lot of ‘mechanism’ code—code that does things like object instantiation and virtual methods—and code that an object-oriented compiler would emit automatically. Additionally, the tools available in C to implement these things, such as void pointer casts and variable argument lists, trade off poorly against type safety, and thus program reliability.

In IIO these issues revolve around the operation function `iio_operate()`. Originally, there was to be only one such function, accepting a channel descriptor, operation code, and a single data pointer, providing a fairly neat interface. Expansion of the library would be handled by the addition of more operation codes. This was proposed because of a desire to avoid a proliferation of operation functions, as this would complicate the module driver interface. Additionally, it was initially thought the library would have to be a kernel module on LynxOS systems, which also argued for a narrow interface.

However, in order to win back a degree of type safety, different operation functions were provided for different user data types. Thus one became six (Section 4.4). This was not really much of a win for type safety anyway, simply because there is no way to guarantee that the size of a destination array for channel range operation is big enough—and there is no way for the program to determine what size it should be anyway.

Furthermore, the servo controller channel model (Section 4.10.5) places further strain on the operation function, because some operations logically required triplets of user data (such as position, velocity and acceleration) to be read or written at once. Because only one datum per channel could be passed per operation, three calls have to be made and the data buffered by the module driver, and a further operation code to promulgate the data into action had to be added.

So, it may well have been worthwhile to consider and model the IIO programmatic interface more before its implementation began. It is probable that a larger suite of operation functions, essentially replacing the operation codes, would not cause the trouble at driver level that was feared, and would improve safety at the application program level.

**Type of Channel Data.** A related issue is the type of channel data. The application can read and write the channel in either an integer form, very close to the hardware format, or a floating point form, scaled to match the real units of the sensor or actuator the channel connects to. It is not clear what the former form is for: precision and efficiency are the reasons given, but most floating point formats have more mantissa bits than the average channel, and floating arithmetic is still very fast (although it remains a serious issue on micro-controllers, which usually lack floating point co-processors).

The dual formats, with the additional exceptions for the address data type, considerably complicate the operation function data access functions, as well as decreasing type safety, or at least increasing the possibilities for error, at the application program level.



**Type of Channel.** The distinction between the different types of channel—`adc`, `dac`, `enc`, and so on—is maintained throughout the library. It must be asked if this really necessary. In the end, channels are either inputs (sensors) or outputs (actuators); should an application care whether a position datum is obtained through an encoder, a digital input channel, or an ADC?

**Duplicate Channel Access.** Any channel can be accessed by four different names (Sections 2.3.2), plus any number of aliases defined in the configuration file. It must also be asked whether this is really necessary. Should a program be able to access the same object by more than one name?

Assuming the answer is no, the scope of all the automatically numbered channel names could be limited to the configuration file, where non-overlapping ranges of channels could be given channel names, not unlike the current aliases. These names would become the *only* names by which a channel or channel range could be opened in the application program. Such a change would considerably simplify the channel opening code, at the expense of a slight complication of the configuration file syntax and interpreter code.

**Library Size.** By UNIX or even vxWorks standards the IIO library, with about 70k of code, is not large. While it constructs a fairly complicated set of data structures, it is also not profligate in its use of dynamic data memory. However, on both counts it is a bit large for microcontroller systems, which typically have 128 or 64k of memory, or even less.

Part of the problem is the fact that IIO maintains a lot of data derived from the configuration file and the module identification functions that is not actually required. The initialisation and file parsing code is also not used after startup. If the library is to be truly useful in this domain, then these issues need to be addressed.

## 8.3 Future Development

Initially, future developments of the current version of the library should concentrate on rectifying some of the criticisms raised above, addressing the interrupt matter, and expanding the range of hardware supported by module drivers.

Many of the criticisms relate partly to the well-known deficiencies in the C language. While it is likely that a slightly better interface was possible using C, it once again raises the issues of programming language. It is time for the Division, or at least those in it who are attempting to engineer commercial quality systems, to abandon this aging language, take advantage of nearly a generation of development in software engineering practice and embrace a stronger language?

The author's view is that it is, and there are a number of choices for a new language (and C++ is not among them). If such a break is made, then there will be an immediate need for generic foundation libraries, including something like IIO, implemented in the new language. The future development of this library should be seen in that light.



# Appendix A

## Standard Module Drivers

The following pages describe the module drivers in the standard list. Each module, or family of related modules, has its own page in a standard format. The pages are organised in the alphanumeric order of the model ident codes. The sections are as follows:

### Module Description

A short description of the module, which specifically details its IO capabilities, its interface bus, significant components, relationship to other modules, and so on.

### Configuration Options

The list of module parameters that may be given to a `module` directive using the module in the configuration file (Section 3). Parameters are option/argument pairs, or binary option flags. Mandatory options must be specified, and have no default argument values. The argument default value of non-mandatory options is shown.

### Driver Status

The version of the module driver code that the documentation refers to. The actual version of the driver can be obtained using the `minfo` command in the IIO interactive shell, Appendix B.

### Module Usage

Any caveats, restrictions, qualifications or advice regarding practical operation of the module.

### Author

The author or origin of the driver code.

## A.1 ADAM 4011, 4012 and 4013 Analogue to Digital Units

Module ident code `adam4011`, `adam4012`, `adam4013`

### Module Description

The ADAM 4012 provides one 16-bit analogue to digital convertors (channel type `adc16`). The ADAM 4011 provides one millivolt-level voltage input or one thermocouple input, calibrated for a variety of thermocouple types. Both the 4011 and 4012 provide one 1-bit digital input channel (type `di1`) and one 2-bit digital output channel (type `do2`). The ADAM 4013 provides a single Resistance-Temperature Device (RTD) input only, calibrated for a variety of RTD types.

### Configuration Options

- `address <channel>` Mandatory. The channel representing the ADAM network and network address of the unit. These channels are provided by ADAM network interface units, such as the `adam4520` (page 91).
- `range <range>` Default `0x08`. Set the input range of the analogue channel on the unit. The codes are listed in Table A.1 on page 89. Note that the 4011, 4012 and 4013 accept different range codes. The code should be written in hexadecimal *with* the leading `0x`.

### Driver Status

This document relates to revision 1.3 of the driver. The event counter and alarm functions are not yet supported, and should be disabled when the unit is configured. Support for the ADAM 4011 and 4013 is not tested.

					Code	Sensor	Range
adam4011	adam4012	adam4013	adam4017	adam4018	0x00	Voltage	$\pm 15\text{ mV}$
■	□	□	□	■	0x01	Voltage	$\pm 50\text{ mV}$
■	□	□	□	■	0x02	Voltage	$\pm 100\text{ mV}$
■	□	□	□	■	0x03	Voltage	$\pm 500\text{ mV}$
■	□	□	□	■	0x04	Voltage	$\pm 1\text{ V}$
■	□	□	□	■	0x05	Voltage	$\pm 2.5\text{ V}$
■	□	□	□	■	0x06	Current †	$\pm 20\text{ mA}$
□	■	□	■	□	0x08	Voltage	$\pm 10\text{ V}$
□	■	□	■	□	0x09	Voltage	$\pm 5\text{ V}$
□	■	□	■	□	0x0A	Voltage	$\pm 1\text{ V}$
□	■	□	■	□	0x0B	Voltage	$\pm 500\text{ mV}$
□	■	□	■	□	0x0C	Voltage	$\pm 150\text{ mV}$
□	■	□	■	□	0x0D	Current †	$\pm 20\text{ mA}$
■	□	□	□	■	0x0E	Thermocouple Type J	0 to $670^{\circ}\text{C}$
■	□	□	□	■	0x0F	Thermocouple Type K	0 to $1000^{\circ}\text{C}$
■	□	□	□	■	0x10	Thermocouple Type T	$-100$ to $400^{\circ}\text{C}$
■	□	□	□	■	0x11	Thermocouple Type E	0 to $1000^{\circ}\text{C}$
■	□	□	□	■	0x12	Thermocouple Type R	500 to $1750^{\circ}\text{C}$
■	□	□	□	■	0x13	Thermocouple Type S	500 to $1750^{\circ}\text{C}$
■	□	□	□	■	0x14	Thermocouple Type B	500 to $1800^{\circ}\text{C}$
□	□	■	□	□	0x20	RTD Platinum 385	$-100$ to $100^{\circ}\text{C}$
□	□	■	□	□	0x21	RTD Platinum 385	0 to $100^{\circ}\text{C}$
□	□	■	□	□	0x22	RTD Platinum 385	0 to $200^{\circ}\text{C}$
□	□	■	□	□	0x23	RTD Platinum 385	0 to $600^{\circ}\text{C}$
□	□	■	□	□	0x24	RTD Platinum 392	$-100$ to $100^{\circ}\text{C}$
□	□	■	□	□	0x25	RTD Platinum 392	0 to $100^{\circ}\text{C}$
□	□	■	□	□	0x26	RTD Platinum 392	0 to $200^{\circ}\text{C}$
□	□	■	□	□	0x27	RTD Platinum 392	0 to $600^{\circ}\text{C}$
□	□	■	□	□	0x28	RTD Nickel	$-80$ to $100^{\circ}\text{C}$
□	□	■	□	□	0x29	RTD Nickel 392	0 to $100^{\circ}\text{C}$
□	□	□	□	□	0x30	Current Loop	0 to 20 mA
□	□	□	□	□	0x31	Current Loop	4 to 20 mA
□	□	□	□	□	0x32	Voltage	0 to 10 V

**Table A.1** ADAM 4000-series Input Range Codes. ■ indicates the unit supports the range code. □ indicates it does not. (The difference between the two current ranges marked † is unclear).

## A.2 ADAM 4017 and 4018 Analogue to Digital Units

### Module ident code adam4017, adam4018

#### Module Description

The ADAM 4017 provides eight 16-bit analogue to digital convertors (channel type `adc16`). The ADAM 4018 provides eight millivolt-level voltage inputs or eight thermocouple inputs, calibrated for a variety of thermocouple types.

#### Configuration Options

- `address` <*channel*> Mandatory. The channel representing the ADAM network and network address of the unit. These channels are provided by ADAM network interface units, such as the `adam4520` (page 91).
- `range` <*range*> Default `0x08`. Set the input range of the analogue channel on the unit. The codes are listed in Table A.1 on page 89. Note that the 4017 and 4018 accept different range codes, and that the range affects all input channels on the unit. The code should be written in hexadecimal *with* the leading `0x`.

#### Driver Status

This document relates to revision 1.4 of the driver. The driver is complete. Support for the ADAM 4018 is not tested.

## A.3 ADAM Module Network

### Module ident code adam4520

#### Module Description

The ADAM 4520 unit interfaces a standard RS232C serial device with an ADAM 4000 RS285 twisted-pair network. This module provides 256 ADAM 4000-series unit address channels (channel type `adam`). For instance, channel `adam.23` refers to the address of an ADAM unit with address 23 decimal (note that the ADAM literature frequently uses hexadecimal representation).

#### Configuration Options

- `-tty <file>` Mandatory. The filename of the serial port to which the network of ADAM 4000-series units are connected (such as `/dev/ttya`).
- `-baud <baud>` Default 9600. The baud rate of the ADAM network. The jumpers inside the ADAM 4520 unit must reflect this rate, and all ADAM units on the network must be configured for this rate. The serial port will be configured to this speed.
- `-checksum/-no-checksum` Default `-no-checksum`. Compute the hexadecimal checksum digits and append to messages sent to ADAM units; test the checksum of messages returned from the units. All units on the network must be configured for checksum operation.
- `-50hz/-no-50hz` Default `-no-50hz`. For analogue ADAM units specify a 50 Hz integration time, rather than the default 60 Hz (recommended).

#### Module Usage

The line specifying this module should appear in the configuration file *before* those for the ADAM units connected on the network.

The ADAM module drivers never change the configuration of the ADAM units. The module options specified in the configuration file must match the configuration of the units, and where possible this is checked. To configure the units, use the ADAM unit configuration software.

#### Driver Status

This document relates to revision 1.4 of the driver.

The serial driver at present does not implement a read timeout, so attempting to install an ADAM unit (other than the 4520) that is not connected, switched off, or configured with different data communications settings will hang the IIO process indefinitely.

## A.4 CSIRO and GreenSpring IP Carriers

### Module ident code atc10, atc30, atc40

#### Module Description

The GreenSpring ATC-40 and GreenSpring ATC-30 are ISA bus IP carriers which provide four and three IP slots respectively (channel type `ip`). The CSIRO/MST ATC-10 is a PC/104 bus carrier which provides one IP slot.

#### Configuration Options

- `address <address>` Default `0xfc0000`. The 24-bit ISA module base address.
- `bus <channel>` Default `isa.0`. The channel representing the ISA bus. This only needs to be specified where there are multiple ISA busses, which is unlikely.
- `irq <irq-number>` Default `0`. The ISA bus interrupt request line the board is jumpered to use, one of 3, 4, 5, 6, 7, 9, 10, 11, 12, 13 or 15, or 0 to indicate none. This option currently has no effect.

#### Driver Status

This is a bare-bones implementation. Memory IPs are mapped only to their default 2k segments, and vectored interrupts are not supported at all. This document relates to revision 1.5 of the driver.



## A.5 BVM IP-ADC Analogue to Digital Convertor

### Module ident code bvmipadc

#### Module Description

The BVM IP-ADC (different to the GreenSpring IP-ADC) provides sixteen isolated, differential 12-bit ADCs. A built-in timer covering the channel select settling period simplifies the driver considerably. Input is 0–5 V or 0–20 mA if the 250  $\Omega$  current loop sense resistors are added.

#### Configuration Options

**-slot <channel>** Mandatory. Specifies the slot in which the IP is installed.

**-gain <gain>** Default 1.0. Specifies the gain of the pre-amplifier stage. This should be computed from the value of resistor R1 according to

$$G = \frac{49400}{R1} + 1$$

In the factory configuration R1 is open, so the default (and minimum) gain is unity.

**-cloop/-no-cloop** Default **-no-cloop**. Indicates the 250  $\Omega$  current-loop resistors are installed, and the channel range is 0–20 mA. This option applies to all channels at once.

**-poll/-no-poll** Default **-no-poll**. Indicates the module driver should poll-wait for ADC conversions, rather than use the conversion timer, which can produce long bus cycles.

#### Driver Status

The ADC correction factors stored in the ID-PROM are not as yet used, otherwise this driver is complete. This document relates to revision 1.10 of the driver.

## A.6 Diamond MM-32-AT Multi-IO PC/104 board

### Module ident code dmm32at

#### Module Description

The Diamond DMM-32-AT is a PC/104 module providing highly configurable 16-bit analogue-to-digital, 12-bit digital-to-analogue and 8-bit digital IO facilities. This driver supports most configurations.

#### Configuration Options

`-bus <channel>` Default `isa.0`. The bus in which the module is installed (IIO regards PC/104 and ISA busses as equivalent).

`-address <address>` Default `0x300`. Module address in port space, set by J7. Must be one of `0x100`, `0x140`, `0x180`, `0x200`, `0x280`, `0x300`, `0x340` or `0x380`.

`-adcconfig <config>` Default `0`. Analogue-to-digital input configuration code, which selects the number and type (single-ended or differential) of the ADC channels. This must correspond to the setting of jumpers set J1–6. See the Diamond DMM-32-AT instruction manual, page 9. In the table below, an installed jumper is indicated by ■, and a removed jumper by □. Note that for practical reasons configuration 2 is not supported.

<code>&lt;config&gt;</code>	Channels	1 2 3 4 5 6	Configuration and Header
0	32	■ ■ □ □ □ □	0–31 SE A
1	16	□ □ ■ ■ ■ ■	0–15 DI B
3	24	■ □ □ ■ □ ■	0–7 SE, 8–15 DI, 16–23 SE D

`-adcrange <range>` Default `0`. Analogue-to-digital input reference and gain code, presently for all channels. Note that codes 4 to 7 are not allowed, and that some ranges are duplicates. See the manual, page 22.

<code>&lt;range&gt;</code>	Input Range	<code>&lt;range&gt;</code>	Input Range
0	±5.0 V	1	±2.5 V
2	±1.25 V	3	±0.625 V
8	±10.0 V	9	±5.0 V
10	±2.5 V	11	±1.25 V
12	0–10.0 V	13	0–5.0 V
14	0–2.5 V	15	0–1.25 V

`-adcscan <scan>` Default `2`. Analogue-to-digital input settling time code. The hardware waits at least this time after channel and gain changes. A `<scan>` of 0 corresponds to 20  $\mu$ s, 1 to 15  $\mu$ s, 2 to 10  $\mu$ s, and 3 to 5  $\mu$ s. The manual recommends against changing this value.

`-dacrange <range>` Default `0`. Digital-to-analogue output reference and gain code, for all channels. See manual, page 10. `<range>` must match the setting of J8 as shown below.

<code>&lt;range&gt;</code>	$\frac{1}{0}$ 5 P B R	Output Range and Note
0	□ ■ □ ■ □	±5 V
1	□ ■ □ □ ■	0–5.0 V
2	■ □ □ ■ □	±10.0 V
3	■ □ □ □ ■	0–10.0 V
4	□ □ ■ ■ □	± $M$ V specify $M$ using <code>-dacmax</code>
5	□ □ ■ □ ■	0– $M$ V specify $M$ using <code>-dacmax</code>

**-dacmax** *<max>* Mandatory. Must be specified if an argument of 4 or 5 has been given to **-dacrange**. *<max>* is the maximum voltage the digital-to-analogue convertor has been adjusted to (the quantity *M* in the previous option). It must be between 0 and 10.0V. It is adjusted using a separate DMM-32-AT calibration program.

**-dioconfig** *<config>* Default 0. Controls the configuration of the DMM-32-AT's 8254 digital input-output chip emulation. All Mode 0 configurations which do not split the chip's Port C are supported. Modes 1 (latched) and 2 (bi-directional bus) are not supported. The table below shows how I/O **di8** and **do8** channels are mapped onto the the three 8-bit ports (A, B and C).

<i>&lt;config&gt;</i>	Port A	Port B	Port C
0	di.0	di.1	di.2
1	di.0	di.1	do.0
2	di.0	do.0	di.1
3	di.0	do.0	do.1
4	do.0	di.0	di.1
5	do.0	di.0	do.1
6	do.0	do.1	di.0
7	do.0	do.1	do.2

**-auxdi/-no-auxdi** Default **-auxdi**. Enables the use of the four auxiliary digital input bits, which are otherwise connected to the 8255 timer/counter chip. The inputs can be read through a **di4** channel.

**-auxdo/-no-auxdo** Default **-auxdo**. Enables the use of the three auxiliary digital output bits, which are otherwise connected to the 8255 timer/counter chip. The outputs are controlled by a **do3** channel.

## Driver Status

The 8254 timer/counter chip is not currently supported. Preamplifier gains other than 1 appear not to work on modules with revision numbers 2L. This document relates to revision 1.3 of the driver.

## A.7 GreenSpring IP-DAC Digital to Analogue Unit

### Module ident code ipdac

#### Module Description

The GreenSpring IP-DAC Digital to Analogue Convertor IP provides size independent 12-bit DAC channels with individual jumper-selectable output ranges.

#### Configuration Options

- `-slot <channel>` Mandatory. Specifies the slot in which the IP is installed.
- `-range <code>` Default 0. Specifies the *default* output voltage range the module is jumpered to use, when not overridden by a `-range.<chan>` for a particular channel.
- `-range.<chan> <code>` Default 0. Specifies the output voltage range the module is jumpered to use for local channel number `<chan>`.

The following table gives the option arguments for both `-range.<chan>` and `-range`:

<code>&lt;code&gt;</code>	Range
0	0 to +5 V
1	0 to +10 V
2	-2.5 to +2.5 V
3	-5 to +5 V
4	-10 to +10 V

#### Driver Status

The current output and dual-DAC capabilities of the IP-DAC are not supported. The driver does not use the ID-PROM factory calibration information. This document relates to revision 1.11 of the driver.

## A.8 GreenSpring IP-Digital 24

### Module ident code ipdigital24

#### Module Description

The GreenSpring IP-DIGITAL 24 provides 24 digital IO lines, organised as three banks of 8-bit registers. The output drivers are open-collector and are permanently linked to the input channels, making them `ocio8` type channels.

#### Configuration Options

`-slot <channel>` Mandatory. Specifies the slot in which the IP is installed.

#### Driver Status

This driver is complete. This document relates to revision 1.12 of the driver.

## A.9 GreenSpring IP-Dual PI/T

### Module ident code ipdualpit

#### Module Description

The GreenSpring IP-DIGITAL 48 and IP-DUAL PI/T provide two MC68230 PI/T chips. The only practical difference between the two is which PI/T pins are brought out to the 50-way connector; for the Digital 48 the Port C/dual function pins are available (and are intended for use as digital inputs or outputs), but for the IP-DUAL PI/T the digital IO handshaking lines are provided. The software implication is that some of the chip modes are made unavailable in certain combinations, as the mode is not useful if certain pins cannot be connected.

At the moment, configuration of the PI/T chips is fixed to provide **only** one 32-bit digital input port, type `di32`. This ignores most all the capability of the Motorola MC68230, which is a complicated multi-modal affair. Because we have to use the handshaking pins, we cannot use port C of either chip.

#### Configuration Options

`-slot <channel>` Mandatory. Specifies the slot in which the IP is installed.

`-ipdigital48/-no-ipdigital48` Default `-no-ipdigital48`. Indicates this IP is really an IP-DIGITAL 48 jumpered to operate as a IP-DUAL PI/T. There are sixteen soldered jumpers located under the PI/T chips.

#### Module Usage

To function with this driver, the IP-DUAL PI/T requires external wiring. The handshake output pin H2 of PI/T X (pin 20 on the 50-way connector) must be connected to handshake input H3 of PI/Ts X and Y (pins 22 and 47). Avoid using long ribbon cables for this wiring, as the high capacitance has been observed to cause malfunctions.

The data bits are arranged as follows:

Data Bits	Port	Pins
D00 to D07	PI/T X port A	1 to 8
D08 to D15	PI/T X port B	9 to 16
D16 to D23	PI/T Y port A	26 to 33
D24 to D31	PI/T Y port B	34 to 41

#### Driver Status

I must recommend against using this IP for just about anything. It is difficult to understand and use, has poor output drive capability, and in any case you don't really get 48 digital input/outputs in most of its modes.

This document relates to revision 1.5 of the driver.

## A.10 GreenSpring IP-Quadrature Quadrature Decoder

### Module ident code ipquadrature

#### Module Description

The GreenSpring IP-QUADRATURE is an IndustryPack compatible module providing four independent 24-bit quadrature decoder channels. Channels may also be used as general purpose up/count counters. Inputs are configured as either single-ended (TTL/CMOS) or differential (RS-422) by removing the appropriate 120  $\Omega$  resistive terminator on the IP.

#### Configuration Options

`-slot <channel>` Mandatory. Specifies the slot in which the IP is installed.

`-prescaler <factor>` Mandatory. Specifies the quadrature prescaler gain. Valid `<factor>`s are 1, 2, or 4 times quadrature count.

#### Driver Status

The hardware counter range of the IP-QUADRATURE has been extended in software from 24 to 32 bits by decoding the carry, borrow, and sign status bits.

The IP-QUADRATURE has a wide variety of programmable input polarity and counter mode operations which are not currently implemented as command line options.

This document relates to revision 1.3 of the driver.

#### Author

Jonathon Ralston, CSIRO Division of Exploration and Mining

## A.11 GreenSpring IP-Serial

### Module ident code ipserial

#### Module Description

The GreenSpring IP-SERIAL provides a Zilog SCC chip, which provides two standard asynchronous serial channels.

Serial devices are not directly supported by IIO. Instead, this proxy driver calls an operating system-specific program specified using the `-init` option. This program should initialise the SCC hardware and insert the necessary entries into the operating system kernel data structures or driver tables, so that the hardware appears to the user as an ordinary serial device. Other programs, or IIO drivers that require serial devices (such as the `adam4520` units), can then access and configure them using the normal operating system mechanisms.

In this way, the IIO configuration file, module identity checking, and address resolution mechanisms can be used, without duplicating the serial device support that exists within most operating systems.

#### Configuration Options

- `-slot <channel>` Mandatory. Specifies the slot in which the IP is installed.
- `-init <program>` Mandatory. Specifies the operating system-specific initialisation program to be called when the module is initialised. See below for details.
- `-argument.<chan> <args>` Default `""`. Additional arguments to be passed to the operating system-specific initialisation program. `<arg>` is not interpreted in any way. Channel 0 refers to SCC channel A, and channel 1 refers to SCC channel B. In the case of UNIX-like operating systems, these options might be used to specify the names of the character-special files the serial devices should be installed as.

#### Module Usage

The operating system-specific initialisation program should be an executable program or script in the case of UNIX-like operating systems, or a C function in the case of shared-memory multi-tasking systems. The name of this program or function should be given as `<program>`.

The program or function `<program>` is executed once for each SCC channel. The arguments are the physical addresses of the control and data registers respectively (as hexadecimal numbers), then any additional arguments specified using `-argument.<chan>`.

The program may be executed more than once in the lifetime of a system, so it should first check if the serial device has already been installed. The program should print a message to a logging stream and return a non-zero status if it cannot install the device. Unfortunately, there is no easy way to return the message to the IIO program that called it.

To view the arguments being passed to `<program>`, specify it to be `/bin/echo`; to discard the arguments, specify `/bin/true`.

#### Driver Status

This driver is complete. This document relates to revision 1.5 of the driver.



## A.12 GreenSpring IP-Servo

### Module ident code ipservo

#### Module Description

The GreenSpring IP-SERVO provides twin LM628 closed-loop position or velocity servo chips, each with 32-bit incremental encoder inputs and 12-bit  $\pm 5$  V range DAC outputs. These generally connect to the velocity drive of a servo amplifier and motor. The digital PID servo loop operates at  $256\ \mu\text{s}$  or 3906 Hz. The driver implements two servo controller channels, type **sc**, as described in Section 4.10.5.

#### Configuration Options

- slot <channel>** Mandatory. Specifies the slot in which the IP is installed.
- lines.<chan> <lines>** Default 1. Specifies the number of encoder lines (or counts) per *user unit* (for example, 4000 lines/m) for channel <chan>. This is equivalent to subsequently specifying the user unit scale factor for the channel using the **channel** configuration file directive, except the factor is also used when other servo parameters (such as loop gains) are specified. In the following options the user unit is U.
- unit.<chan> <unit>** Default (*none*). Specifies the *user unit* name of U. This is equivalent to subsequently specifying the user unit name for the channel using in the **channel** configuration file directive.
- pgain.<chan> <pgain>** Default 0. Specify the proportion gain of the LM628 digital servo loop. The units are V/U (volts per user unit error). In designing the control system and determining the proportional gain, the gain of the stages following the voltage output from the IP-SERVO must be taken into account.
- dgain.<chan> <dgain>** Default 0. Specify the derivative gain of the LM628 digital servo loop. The units are V/U/s (or Vs/U), volts per user unit per second error.
- igain.<chan> <igain>** Default 0. Specify the integral gain of the LM628 digital servo loop. The units are V/Us, volts per user unit seconds.
- imax.<chan> <imax>** Default . Specifies the maximum value of the integral feedback component in Us. The default limit is the largest the LM628 chip will accept (equivalent to 32767 chip units).
- dperiod.<chan> <period>** Default 256e-6. Specifies the period of the derivative feedback computation.  $256\ \mu\text{s}$  is the minimum (same rate as the main feedback loop). The maximum is 65.536 ms.
- bipolar.<chan>/-no-bipolar.<chan>** Default **-no-bipolar.<chan>**. Indicates the IP-SERVO is jumpered for bipolar operation ( $\pm 5$  V) or unipolar (0–5 V).
- 4mhz/-no-4mhz** Default **-no-4mhz**. Indicates the IP-SERVO operates at 4 MHz instead of 8 MHz (only very old IP-SERVO should need this option).

#### Module Usage

While every effort has been made to re-package the characteristics of the LM628 chip, including real-unit scaling of the several loop parameters, many foibles remain. Familiarity with the chip manuals and application notes is therefore an advantage.

## **Driver Status**

This document relates to revision 1.7 of the IP-SERVO driver, and revision 1.8 of the LM628 chip driver.

## A.13 GreenSpring IP-Watchdog

### Module ident code ipwatchdog

#### Module Description

The GreenSpring IP-WATCHDOG provides an 8-bit digital IO port (channel type `ocio8`), a programmable periodic or watchdog timer/interrupter, external control of VMEbus `SYSRESET`, power supply range monitors (+5 V, +12 V, -12 V and a selectable  $\pm 24$  V or  $\pm 48$  V, channel type `di8`), and a programmable temperature monitor/thermostat (channel type `adc9`).

#### Configuration Options

- slot <slot> Mandatory. Specifies the slot in which the IP is installed.
- temphi <th> Default (*previous*). Specifies in degrees celsius the upper temperature limit. If the temperature exceeds this value the temperature alarm output bit 5 of the `di8` channel will be asserted. The default value is the previous setting (it is stored in non-volatile memory).
- templo <th> Default (*previous*). Specifies in degrees celsius the lower temperature limit. If the temperature falls below this value the temperature alarm output bit 5 of the `di8` channel will be asserted. The default value is the previous setting (it is stored in non-volatile memory).

#### Module Usage

The interpretation of the digital in port (`di8.0`) is as follows:

Bit	Channel	Meaning
0	<code>bi.0</code>	+5 V power supply out of acceptable range
1	<code>bi.1</code>	+12 V power supply out of acceptable range
2	<code>bi.2</code>	-12 V power supply out of acceptable range
4	<code>bi.4</code>	$\pm 24$ V or $\pm 48$ V supply out of acceptable range
5	<code>bi.5</code>	temperature out of specified range

The IIO bitfield channels can be used to conveniently read these bits. The local channel names are given above.

#### Driver Status

Currently, only the digital IO and the monitors are implemented in this driver. Interrupts are not supported. This document relates to revision 1.7 of the driver.

## A.14 Generic ISA PC

### Module ident code isapc

#### Module Description

This module must be installed first on generic ISA ('Industry Standard Architecture') machines (i.e., PCs). It installs a single ISA bus channel `isa.0` which is used by other ISA bus modules for address resolution.

#### Configuration Options

`-mbytes <size>` Default 0. Indicates the PC is fitted with `<size>` megabytes of memory. This causes all IIO memory address resolution operations on the `isa.0` channel with physical addresses below `<size>` to fail. It is intended to catch the installation of ISA modules at addresses that would clash with memory.

#### Module Usage

The ISA bus provides two address sub-spaces, a 24-bit memory space and a 16-bit port space. The port space is not memory mapped, and so drivers for modules with registers in this space must use the `iio_space_port` address space operation code for address resolution and mapping, and the `iio_port_get()` and `iio_port_set()` functions to access them.

The 24-bit ISA bus memory space is shared with on-board RAM, so usually only accesses to the segment *above* the top of system RAM will cause ISA bus memory cycles. If more than 16 Mb of RAM is installed, the ISA bus is essentially unavailable. On some systems, however, it may be possible to exploit a 'hole' in between the top of video adaptor memory and the bottom of the BIOS ROM, at addresses such as 0x0d0000, to install small ISA memory mapped modules.

The memory cache should be disabled for those addresses used to access ISA modules. The cache is believed to have caused difficulties in some cases. The BIOS set-up menus can usually be used to exclude selected address ranges from being cached.

There are two forms the ISA bus modules and slots. The original PC form has an 8-bit data bus and a 16-bit address range. These are sometimes referred to as '8-bit ISA', although ISA terminology is not always consistent. The second PC/AT form has a 16-bit data bus and a 24-bit address space, with an extra edge connector to carry the additional signals. These are known as '16-bit ISA' modules. The 24-bit address space is a superset of the 16-bit space.

There is a third form, called EISA or Extended ISA. These have 32-bit data and address busses, and use peculiar double-row edge connectors, for backward compatibility with normal ISA modules. EISA cards are rare, and the term is often thought to refer to 16-bit ISA. EISA has been supplanted by the PCI bus.

#### Driver Status

This document relates to revision 1.6 of the driver.

## A.15 Motorola MVME160 series MPU boards

### Module ident code mvme162, mvme162lx, mvme167

#### Module Description

This module driver should appear first in the configuration file for systems based on Motorola MVME-160 series MC68040 MPU boards (MVME-162 and MVME-167). It provides one channel `vme.0` for use by other VMEbus boards for address resolution.

#### Configuration Options

There are no module-specific options.

#### Driver Status

This driver only performs address resolutions using the two fixed maps (A16/D16 and A24/D16) of the VMEchip2 ASIC. The other four mappings are usually set up by the operating system. Eventually, all mappings should be used for resolutions. The driver does not yet provide the two or four IP channels (type `ip`) on the MVME-162LX and MVME-162 respectively.

This document relates to revision 1.9 of the driver.

## **A.16 Motorola MVME1600 series MPU boards**

### **Module ident code mvme1603, mvme1604**

#### **Module Description**

This module driver should be installed first on MVME-1600 series of PowerPC MPU boards (MVME-1603 and MVME-1604). It provides one channel `vme.0` for use by other VMEbus boards for address resolution.

#### **Configuration Options**

There are no module-specific options.

#### **Driver Status**

This driver only performs address resolutions using the two fixed maps (A16/D16 and A24/D16) of the VMEchip2 and the VME2PCI ASICs. The other four mappings are usually set up by the operating system.

This document relates to revision 1.11 of the driver.

## A.17 Null Module

### Module ident code null

#### Module Description

The `null` module is used internally by the IIO library. It installes a single channel `null.0`. Operations on this channel always succeed. Read operations on this channel always return zero, and address resolution operations always return the original address.

#### Configuration Options

There are no module options. The module is automatically installed, and should not appear in configuration files.

#### Driver Status

It is possible that a re-design of the address resolution mechanism will make this driver unnecessary.

## A.18 Generic PC Game Port

### Module ident code pcgp

#### Module Description

The generic PC game port module provides four potentiometer and four switch inputs. Only the switch inputs are supported by this driver. Game ports are often integrated into ISA motherboards or sound-cards.

#### Configuration Options

`-bus <channel>` Default `isa.0`. Specifies the bus in which the module is installed.

This module always has address `0x201`. At most one may be installed.

#### Driver Status

The principal value of the game port that it provides a convenient +5 V DC supply outlet. The driver is a simple example of one for an ISA port-mapped module.

The potentiometer inputs cannot be read except with the aid of a software timing loop, which is not practical in multi-tasking systems. This document relates to revision 1.3 of the driver.



## A.19 Generic PC Printer Port

### Module ident code pcpp

#### Module Description

The PC printer port module supports generic and IEEE-1284 compatible parallel interfaces in several modes.

#### Configuration Options

- address <address> Default 0x378. Must be one of 0x378 (for hardware configured to be MS-DOS LPT1:), 0x278 (LPT2:), 0x3bc (LPT3: or PRN:).
- irq <irq> Default 7. Interrupt setting. Currently ignored.
- bus <channel> Default isa.0. The bus in which the module is installed.
- mode <mode> Default spp. Printer port mode. Must be one of spp, sppi, epp or eppa, described below. <mode> determines what IO resources are available.

#### Module Usage

Printer port hardware varies widely, particularly that which pre-dates the IEEE-1284 standard. For instance, some 'standard' hardware uses open-collector drivers instead of totem-pole drivers. Some hardware features a bi-directional data port, while other hardware does not.

The IEEE-1284 standard more clearly defined the parallel port design, and formalised a number of modes of operation. This driver supports the IEEE-1284 Compatibility (SPP), Byte, and EPP modes, which appear most useful for IIO-style applications, and ignores the Nibble and ECP modes.

**spp** Standard Parallel Port mode. This assumes the Compatibility mode defined by IEEE-1284 and models pre-IEEE-1284 hardware, with the caveats mentioned above. Most old parallel ports should work with this mode.

The data lines provide a **do8** channel, and the four printer control lines a further **oco4** channel. The five status lines provide a **di5** channel. The assignment of channel bits to connector pins is shown in Table A.2.

**sppi** Standard Parallel Port input mode. In this mode the data lines are used as a **di8** channel. This is possible only if the parallel port hardware has a bidirectional capability which works in the standard manner. The **oco4** and **di5** channels remain.

It would be possible, but not practical, to have used the data lines as a reversible **rdio** channel, but there is no hardware indicator of the direction of data flow which external hardware could use. For this sort of application the **epp** mode should be used.

**epp** Enhanced Parallel Port mode. This bidirectional mode provides two **rdio8** channels ('address' and 'data') which share the data lines. The status and control lines are used for hardware data transfer strobes, which are defined in the Intel/Xircom/Zenith/IEEE EPP standard.

A full description of EPP operation is beyond the ambit of this document, but briefly, data direction is controlled by the  $\overline{\text{WRITE}}$  output, and data is latched on the rising edge of the  $\overline{\text{ADDRSTROBE}}$  or  $\overline{\text{DATASTROBE}}$  output as appropriate. The 'address' port is IIO channel **rdio8.0**, and 'data' **rdio8.1**. The strobe output must be acknowledged by the rising edge of the  $\overline{\text{WAIT}}$  input to complete the cycle.

Note that there are two versions of the EPP standard, 1.7 and 1.9. The difference lies in the value of  $\overline{\text{WAIT}}$  needed at the start of each cycle.

**eppa** Enhanced Parallel Port addressable mode. *Experimental.* This uses the EPP's addressing capability to provide up to 128 **di8** and 128 **do8** channels, assuming the EPP address port is used as a latch/buffer selector. This mode requires external hardware conforming to a specific (although fairly simple) design.

**Warning.** The parallel port hardware was not intended for industrial use, and, in **spp** and **sppi** modes at least, should not be connected to any potentially hazardous actuator. This is because the outputs change state once, and sometimes more, during PC reset, BIOS initialisation, operating system booting, and I/O startup.

A more robust approach would use **epp** mode, as the hardware is unlikely to initiate a spurious EPP data cycle on startup.

**Configurable Hardware.** Modern parallel ports allow the hardware to be configured (often using the BIOS menus) to emulate various parallel port standards. For maximum flexibility an 'ECP+EPP' setting is preferred. ECP mode usually permits Compatibility and Byte modes (used by the I/O **spp** and **sppi** modes) and well as ECP, and the ECP mode is used by the I/O **epp** and **eppa** modes.

## Driver Status

This document relates to revision 1.3 of the driver.

Channel ( <b>spp</b> /i)	Bit	Signal (SPP)	Pin	Signal (EPP)	Channel ( <b>epp</b> )	Bit
oco4	3	$\overline{\text{SEL}}$	17	$\overline{\text{ADDRSTROBE}}$		
oco4	2	$\overline{\text{INIT}}$	16	$\overline{\text{RESET}}$		
oco4	1	$\overline{\text{AFD}}$	14	$\overline{\text{DATASTROBE}}$		
oco4	0	$\overline{\text{STROBE}}$	1	$\overline{\text{WRITE}}$		
do8 or di8	7	DATA7	9	DATA7	rdio8.0-1	7
do8 or di8	6	DATA6	8	DATA6	rdio8.0-1	6
do8 or di8	5	DATA5	7	DATA5	rdio8.0-1	5
do8 or di8	4	DATA4	6	DATA4	rdio8.0-1	4
do8 or di8	3	DATA3	5	DATA3	rdio8.0-1	3
do8 or di8	2	DATA2	4	DATA2	rdio8.0-1	2
do8 or di8	1	DATA1	3	DATA1	rdio8.0-1	1
do8 or di8	0	DATA0	2	DATA0	rdio8.0-1	0
di5	4	BUSY	11	$\overline{\text{WAIT}}$		
di5	3	$\overline{\text{ACK}}$	10	INTERRUPT		
di5	2	PE	12			
di5	1	SELIN	13			
di5	0	$\overline{\text{ERR}}$	15			
		GROUND	18-25	GROUND		

**Table A.2** Parallel Port D-25 connector pin assignments in I/O **spp** and **sppi** modes (left) and **epp** mode (right).

## A.20 Phantom Module

### Module ident code phantom

#### Module Description

The **phantom** module driver inserts arbitrary channel nodes into the channel node list, creating in effect ‘phantom’ channels. These channels may be opened and operated upon like any other channel; the generic operation codes `iio_op_read`, `iio_op_readback`, `iio_op_write`, and `iio_op_clear` are implemented. Operations do nothing, except that the data from `iio_op_write` operations is stored, and is returned one a subsequent `iio_op_read` or `iio_op_readback`.

#### Configuration Options

- `-type <type>` Default `do32`. The type and width of the phantom channels. For instance, `adc16` refers to a 16-bit ADC channel. Any legal channel type or width may be specified.
- `-number <number>` Default 1. The number of channels of the given `<type>` to insert.

#### Module Usage

There are two main uses for this module:

- where hardware in a system has failed or has been removed, the **phantom** module can be used to put the equivalent number of channels back in to the system, so that existing programs can continue to run.
- where a new program, or the IIO library itself, is being tested, and it may be unsafe to connect real hardware. In this case, channel logging is generally enabled, using the `-log` switch in the configuration file.

#### Driver Status

The appropriateness of module operations is not checked, so it is possible, for example, to write to input devices without error.

## A.21 Tews TIP-850-11 ADC/DAC

### Module ident code tews850

#### Module Description

The Tews DatenTechnik TIP-850-11 is an Industry Pack compatible module providing eight differential or sixteen single-ended 12-bit ADCs (software selectable), as well as four independent 12-bit DACs. The TIP-850-11 provides input voltage ranges of  $\pm 1.25$ ,  $\pm 2.5$ ,  $\pm 5$  and  $\pm 10$  V, and output range of  $\pm 10$  V. Minimum single channel throughput is 100 kHz.

#### Configuration Options

- slot <channel> Mandatory. Specifies the slot in which the IP is installed.
- gain <gain> Default 1.0. Specifies the gain of the pre-amplifier stage. Valid options for the TIP-850-11 are 1, 2, 4, or 8, corresponding to  $\pm 10$ ,  $\pm 5$ ,  $\pm 2.5$ , or  $\pm 1.25$  V respectively.

#### Driver Status

The ADC correction factors stored in the ID-PROM are not used. The gain factor currently applies to all channels, although it is possible to have different gain settings for each individual channel. The TIP-850-11 supports interrupt generation on analogue-to-digital conversion (e.g., cf. the GreenSpring IP-PRECISION-ADC, but the TIP-850-11 auto-settle option does not delay bus cycles (e.g., as in the bvm IP-ADC). As a consequence, a conversion-complete flag is polled to ascertain conversion status.

This document relates to revision 1.5 of the driver.

#### Author

Jonathon Ralston, CSIRO Division of Exploration and Mining

## A.22 GreenSpring VIPC610 VME IP Carrier

### Module ident code vipc610

#### Module Description

The GreenSpring VIPC-610 and VIPC-616 are IP carriers providing four IP slot addressed from the VMEbus. The VIPC-616 is an updated model with more features.

#### Configuration Options

- address <address> Default 0x6000. Specifies the address in VMEbus short (A16) space where the module is located.
- bus <channel> Default vme.0. Specify the channel representing the VMEbus in which the module is plugged. This option is only useful in multi-VMEbus systems, which are rare.
- memory/-no-memory Default -no-memory. Indicates that IP memory is mapped to the VMEbus A24 space.

#### Driver Status

The features of the VIPC-616 which are not identical to the VIPC-610 are not supported. Interrupts are not supported in either case.

This document relates to revision 1.12 of the driver.

## A.23 VMIC Digital IO Modules VMIVME-2532A and VMIVME-2534

Module ident code `vmivme2532a`, `vmevme2534`

### Module Description

The VMIC VMIVME-2532A and VMIVME-2534 provide two 16-bit high-voltage digital IO channels (type `ocio16`) on the VMEbus. The only difference between them is the connector arrangement (front panel on the VMIVME-2532A and via VMEbus P2 on the VMIVME-2534).

### Configuration Options

- `address <address>` Default `0xff60`. Specifies the address in VMEbus short (A16) space where the board is located.
- `bus <channel>` Default `vme.0`. Specify the channel representing the VMEbus in which the module is plugged. This option is only useful in multi-VMEbus systems, which are rare.

### Driver Status

This document relates to revision 1.14 of the driver.

## A.24 VMIC VMIVME-4100 VME Digital to Analogue Module

### Module ident code `vmivme4100`

#### Module Description

The VMIC VMIVME-4100 provides 16 12-bit DAC channels on the VMEbus (channel type `dac12`).

#### Configuration Options

- address** <*address*> Default `0x0`. Specifies the address in VMEbus short (A16) space where the module is located. The factory default is silly.
- bus** <*channel*> Default `vme.0`. Specify the channel representing the VMEbus in which the module is plugged. This option is only useful in multi-VMEbus systems, which are rare.

#### Module Usage

The board should be set up for update under program control (jumpers JC1 to JC4 are in), rather than immediate mode (the factory default). This permits the driver to load a range of channels into the DAC preload registers, then promulgate them all at once, for simultaneous update.

If the jumpers are out, the individual output will probably update immediately, and the attempted promulgate will do nothing. Similarly, if the jumpers are in, and the external update jumper (JC5) is in as well, the promulgate will do nothing and promulgation will be caused by a low-pulse inserted into the external trigger input.

Therefore, there are no driver options for these two setups.

#### Driver Status

This driver has not yet been tested. This document relates to revision 1.7 of the driver.





# Appendix B

## Using the IIO Interactive Interface

While the IIO library is intended for linking with specific-purpose application programs, a general purpose interactive shell program `iio` is also provided. This can be used to check configuration files, manually operate channels, and print lists of available modules and channels. The program uses the GNU ‘readline’ command-line history and editing library, where this is available.

The program is started by typing `iio` at the UNIX command line. It responds with a preamble and prompt

```
% iio
This is an initial IIO process
Library timestamp 970205123355
iio>
```

As described in Section 7.6.4, IIO-using UNIX processes use a common shared-memory block for module state, where this is possible. The initial IIO process establishes the block, and subsequent processes attach to it. Only programs linked with IIO libraries with identical time-stamps can attach.

The `help` command produces the following information:

```
iio> help
Commands:
  alias      print alias list
  chnode     print channel node list
  map        print virtual/physical map
  minfo      print module info list
  module     print installed module list
  operate    operate on a channel
  execute    execute UNIX command
  help       print this message
  quit       quit program
Commands can be abbreviated
```

Each of the commands, except `help` and `quit`, will now be described, assuming the the following `iio.conf` file:

```
# config file for dline
module mvme1603                                # -log

module vipc610 -address 0x6000                  # -log
module vipc610 -address 0x6800                  # -log

module bvmipadc -slot vipc610.0:ip.0            # -log
module ipdac -slot vipc610.0:ip.1               # -log
module ipdigital24 -slot vipc610.0:ip.2         # -log
module ipdigital48 -slot vipc610.0:ip.3         # -log
module ipwatchdog -slot vipc610.1:ip.1         # -log
module ipservo -slot vipc610.1:ip.2            # -log

alias hoist_volt bvmipadc.0:adc.0
alias hoist_amp bvmipadc.0:adc.1
alias hoist_reg bvmipadc.0:adc.2
```

## B.1 The alias Command

This lists the aliases specified in the configuration file. Aliases are described fully in Section 3.3.

```
iio> alias
type                name    value
global              hoist_amp = bvmipadc.0:adc.1
global              hoist_reg = bvmipadc.0:adc.2
global              hoist_volt = bvmipadc.0:adc.0
```

## B.2 The chnode Command

This lists the names and number of channels associated with each of the modules defined in the configuration file (the *channel nodes*):

```
iio> chnode
num    name(gg)      name(gs)          name(lg)          name(ls)
16     adc.0-15      adc12.0-15        bvmipadc.0:adc.0-15 bvmipadc.0:adc12.0-15
6      dac.0-5       dac12.0-5         ipdac.0:dac.0-5   ipdac.0:dac12.0-5
3      dio.0-2       dio8.0-2          ipdigital24.0:dio.0-2 ipdigital24.0:dio8.0-2
4      ip.0-3        ip0.0-3           vipc610.0:ip.0-3   vipc610.0:ip0.0-3
1      null.0-0      null10.0-0        null.0:null.0-0    null.0:null10.0-0
1      vme.0-0       vme0.0-0          mvme167.0:vme.0-0  mvme167.0:vme0.0-0
```

Each channel node comprises a group of channels of exactly the same type. The number of channels is given in the first column. The four forms of the channel names are given in the remaining columns: global generic, global specific, local generic, and local specific. Channel names are described in Section 2.3.2.

## B.3 The map Command

This prints out the physical-to-logical mappings the IIO library is using:

```
iio> map
      paddr      vaddr      size      size
0xffff6000  0x80000000  0x00001000    4096
```

`paddr` refers to processor physical address of start of mapped segment: `vaddr` is the logical address of start of mapped segment. The size of the segment is given in hexadecimal and decimal. These mappings are created by the calls to `iio_map()` performed by module drivers during their installation, as described in Section 5.3.4.

## B.4 The minfo Command

This provides information on all the supported modules (the *module info blocks*):

```
iio> minfo
ident  install  init  mul  model  rev
atc30  0x00008768  0x0000872a  yes  GreenSpring ATC30 IP Carrier  1.1
atc40  0x00008768  0x0000872a  yes  GreenSpring ATC40 IP Carrier  1.1
bvmipadc  0x000057c2  0x000056ba  yes  BVM IP-ADC  1.6
ipdac  0x00005eb8  0x00005d54  yes  GreenSpring IP-DAC  1.6
ipdigital24  0x00006620  0x0000657c  yes  GreenSpring IP-Digital 24  1.7
ipservo  0x0000ab8e  0x0000aa44  yes  GreenSpring IP-Servo  1.1
ipwatchdog  0x000093ca  0x000092ec  yes  GreenSpring IP-Watchdog  1.1
isapc  0x00008ec2  0x00008e88  no   ISA PC  1.1
```

mvme1603	0x00006dbe	0x00006d8c	no	Motorola MVME1603 MPU	1.7
mvme1604	0x00006dbe	0x00006d8c	no	Motorola MVME1604 MPU	1.7
mvme162	0x00006a12	0x000069e0	no	Motorola MVME162 MPU	1.5
mvme162lx	0x00006a12	0x000069e0	no	Motorola MVME162LX MPU	1.5
mvme167	0x00006a12	0x000069e0	no	Motorola MVME167 MPU	1.5
null	0x0000d97e	0x0000d970	no	<null>	1.3
phantom	0x0000db9c	0x0000db00	yes	<phantom>	1.3
vipc610	0x000072fa	0x000072d2	yes	GreenSpring VIPC610 IP Carrier	1.8
vmivme2532a	0x000078d2	0x000077d0	yes	VMIC VMIVME-2532A Digital IO	1.9
vmivme2534	0x000078d2	0x000077d0	yes	VMIC VMIVME-2534 Digital IO	1.9
vmivme4100	0x00009a3c	0x00009924	yes	VMIC VMIVME-4100 16-chan DAC	1.3

As described in Section 5.2, each module driver identifies itself to IIO using its *identification function*. The information from this process is stored in the module information (`minfo`) list.

The first column, `ident`, is the unique model identifier, described in Section 2.2.1. The following two columns are the addresses of the module driver installation and initialisation functions described in Sections 5.3 and 5.4. The `mul` column indicates if the module can be installed more than once. The remaining columns give a fuller description of the module and the RCS revision number of the module driver.

## B.5 The module Command

This prints the *module list*, the modules that are actually installed:

```
iio> module
  ident seq      state      reg      log  model
  bvmipadc.0 0x00017a60 0x00017610 no    BVM IP-ADC
  ipdac.0    0x00017cc8 0x00017a98 no    GreenSpring IP-DAC
  ipdigital24.0 0x00000000 0x00017d08 no    GreenSpring IP-Digital 24
  mvme167.0   0x00000000 0x00000000 no    Motorola MVME167 MPU
  null.0      0x00000000 0x00000000 no    <null>
  vipc610.0   0x00000000 0x00017478 no    GreenSpring VIPC610 IP Carrier
```

While parsing the configuration file, IIO builds this list of installed modules. The first column is the *module ident*, including the module sequence number (Section 2.2.1). The following two are the addresses of the register and state structures of the module driver, described in Sections 5.3.1 and 5.3.2. The `log` column indicates if logging of module operations is enabled. The `module` column is a duplicate of that in the `minfo` list.

## B.6 The operate and oinfo Commands

The command takes the form

```
operate <channel> <operation> [ <data> ...]
```

and actually performs the given operation on the channel or channel range. **Always check it is safe to perform the operation.** In other words, make sure that equipment connected to the IO system will not cause a hazard when operated. Take particular care with output channels, such as servo systems or power relay controls.

The command `oinfo` gives information on the range of available operations:

```
iio> oinfo
  name      type
00  nop
```

01	show	
02	read	out data
03	readback	out data
04	write	in data
05	clear	
06	space-io	in out addr
07	space-id	in out addr
08	space-int	in out addr
09	space-mem	in out addr
10	space-mem16	in out addr
11	space-mem24	in out addr
12	space-mem32	in out addr
13	sc-start	
14	sc-stop	
15	sc-free	
16	sc-read-current	out data
17	sc-read-target	out data
18	sc-read-index	out data
19	sc-write-current	in data
20	sc-write-target	in data
21	sc-write-target-dt	in data
22	sc-write-target-ddt	in data

For `write`-type operations with the `operate` command, there must be as many `<data>` numbers as channels, so if the operation is on, say, `adc.0-3` than there should be four numbers.

The type of user data is determined from the syntax of the data. If there is a period `.` in any of the data, the `iio` program calls `iio_operate_real()` to perform the operation, and the channel scale and offset factors for real data apply. Otherwise, for decimal, octal or hexadecimal integer data, `iio_operate()` is used.

Here are some examples of the use of `operate`:

```
iio> op do.0 write 1
do.0: write[1]:
adam4012.0:do2.0: ir[0]: 00000001 <- 1

iio> op adc.25 read
adc.25: read[1]:
adam4012.0:adc16.0: ir[0]: 000006de -> 1.758 V

iio> op adc.25 write 2.3
adc.25: write[1]:
operate: adc.25: Operation code not supported by channel

iio> op do.0 readback
do.0: readback[1]:
adam4012.0:do2.0: ir[0]: 00000001 -> 1

iio> op dio.0-3 read
dio.0-3: read[4]:
ipwatchdog.0:dio8.0: ir[0]: 000000ff -> 255
ipdigital24.0:dio8.0: ir[1]: 000000ff -> 255
ipdigital24.0:dio8.1: ir[2]: 000000ff -> 255
ipdigital24.0:dio8.2: ir[3]: 000000ff -> 255
```

Here, the value 1 is written to a digital output, the ADC number 25 is read (showing 1.758 V), then the ADC is written to (in error, as IIO knows an ADC is an input device), and then the digital output channel is read back. Finally, a range of four `dio` channels is read.

The confirmatory output from the `operate` command is the same as that produced when logging is enabled on a channel or module. The first line indicates

the channel, the operation, and the size of the user data array. The channel name is the one by which it was opened, so it may actually be an alias name.

The subsequent lines indicate the actual simple channels being operated on. A two-letter acronym indicates the data conversions occurring (for example, `ir` means integer to real) and the number in square brackets is the index into the user data array. The (usually) hexadecimal data is the integer data supplied by the core IIO library to the module driver (it is not necessarily the actual data going into the hardware device). Finally the user data, generally in real format with units, is shown.

## B.7 The execute Command

```
execute <command> [ <arg> ...]
```

The `execute` command runs a UNIX command, for example, `ls`:

```
iio> execute ls
i486-lynxos      m68040-lynxos   ppc-lynxos      sparc-sunos
iio              platform      sparc-solaris
```

## B.8 The script Command

This reads and executes a file of commands `<script>` into the `iio` as if they had been typed.

```
script <script>
```

This is handy for frequently-used sequences of commands, but again, **always check the script will be safe to execute**. (Use `exec vi <script>` to edit the script without exiting the `iio` program).



# Appendix C

## Installing and Maintaining the IIO Library

### C.1 IIO CVS Archive

The sources to the complete IIO system, including documentation, are maintained in the cvs (Concurrent Version System) repository at CMST Preston. You can obtain the current version of IIO by typing

```
% cvs checkout iio
```

which will create a copy of the IIO distribution tree in the current directory. This copy is not ‘locked’ in any way, and may be modified, moved or deleted without damage to the master in the cvs repository. Alterations to any of the controlled files in the tree can be merged into the master if required.

### C.2 Distribution Usage

The IIO distribution may be used in several ways. Firstly, a distribution might be installed permanently in a public or project-group directory as a central resource. Application programs would compile and link against the IIO header files and library in this tree. The tree can be shared amongst different computers, because object files and libraries are segregated according to machine architecture and operating system, as described in later sections.

Alternatively, a copy of the distribution can be obtained and included alongside the sources of a particular application program. This has the advantage of insulating the application from unexpected changes to the library. However, because the shared-memory time-stamp check on system such as LynxOS, such applications probably will not coexist with others built using a different copy of IIO.

For working on IIO itself, such as writing a new module driver, it is definitely a good idea to obtain a fresh copy of the distribution and work from there. The changes or new files can be submitted for inclusion in the master sources. Once the work is complete, the distribution can be deleted.

### C.3 Platform Script

The platform script, **platform**, prints a string that succinctly identifies a computer system by its CPU architecture and its operating system. Examples of its output are **sparc-solaris** and **i486-lynxos**. These strings are similar to the target strings used to specify system types to GNU **configure**.

**platform** invokes the standard program **uname** on UNIX systems, and tidies its output into a consistent format. The IIO **Makefile** uses the identifier to segregate object code, so that the IIO library can be compiled for a number of different computer systems at once in the one source tree.

The **platform** script is also used for run-time executable selection. If a link, for example, **xyzyy**, is made to the script and executed on a Sparc system, the script will in turn execute **sparc-solaris/xyzyy**. Multi-architecture

`bin` directories can be implemented by placing the real binaries for each architecture/operating system combination in sub-directories `bin/sparc-solaris`, `bin/i486-linux`, and so on. The `platform` script is placed in the `bin` directory, and a symbolic links to it are installed for each real program in each sub-directory. Users then need only to place the `bin` directory in their executable search path: invoking the name of a program causes the `platform` script to execute the correct executable program. (Architecture-independent programs, such shell and Tcl scripts, still go in the `bin` directory).

## C.4 Distribution Tree

### C.4.1 Sub-Directories

The distribution tree, as it comes from the repository, contains only two sub-directories, `iio/src` and `iio/doc`. The building process, carried out mostly by `iio/src/Makefile` (Section C.5), creates further subdirectories `iio/bin`, `iio/h`, and `iio/lib`, as well as directories for object files under `iio/src/object`. The tree is illustrated in Figure C.1.

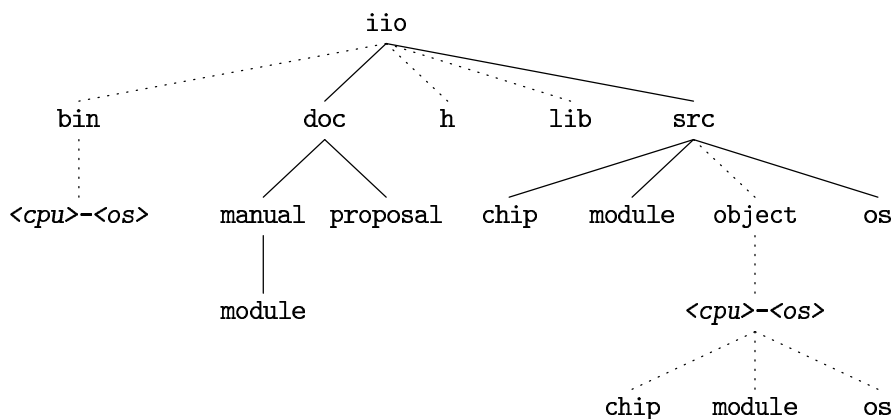
All these have the traditional meanings. Directory `iio/bin` should be placed in the command search environment variable `PATH`. It has subdirectories for each platform type, as described above. `iio/h` contains public header files, and `iio/lib` compiled object archives. This directory has no sub-directories for platform type; instead, the archive files themselves bear the platform type.

### C.4.2 Sources

The sources of the IIO core library, interactive shell, scripts, and test programs are in `iio/src`. There are two subdirectories: `iio/src/module`, which contains all the module drivers and generic module support code, and `iio/src/chip`, which contains the chip drivers and their header files.

### C.4.3 Documentation

The IIO library is documented using L<sup>A</sup>T<sub>E</sub>X. Sources for the manual (the document you are reading now) are in `iio/doc/manual`. The original IIO proposal document is in `iio/doc/proposal`. The documentation can be built by typing `make` in this directory.



**Figure C.1** IIO Distribution Tree and Sub-directories. Solid lines indicate the directories that are part of the original IIO distribution, while dotted lines indicate those created by `iio/src/Makefile`.



## C.5 The IIO Makefile

Building the IIO library is controlled by `iio/src/Makefile`. This file uses the platform script (Section C.3) and the facilities of GNU `make` to build IIO for a variety of self- and cross-hosted targets *without* the need for `configure` and automatically generated `Makefile` hierarchies.

Installers should not need to modify `iio/src/Makefile` at all to build IIO from the CVS distribution tree. They should simply change to `iio/src` and typing `make` (or `gmake`, if this is the name GNU `make` was installed under):

```
% cd iio/src
% make
```

The `Makefile` compiles first the IIO library, and installs it in, for instance, `iio/lib/libiio-i486-linux.a`, on a Linux system. It then builds the IIO interactive interface program `iio` (Appendix B) in `iio/bin/i486-linux/iio`, and installs a link `iio/bin/iio` to the platform script, `iio/bin/platform`, as described in Section C.3.

IIO can be compiled for cross-hosted targets, such as vxWorks, by typing

```
% cd iio/src
% make 68040-vxworks
```

although this assumes the presence of the vxWorks cross-development environment.

Each time IIO is made, a new library time-stamp is created and compiled into the library. This is essential to ensure that memory segments shared between IIO-using processes are match correctly. Unfortunately, it means that is IIO is re-made, then all IIO -using applications must also be re-made.

## C.6 An IIO Application Makefile

Application `Makefiles` should also use the `platform` script, as it makes them more portable. The following is a simplistic example:

```
PLATFORM := $(shell platform)
IIO = /usr/aa/iio

INCLUDES = -I$(IIO)/h
DEFINES =
LIBS = -L$(IIO)/lib -liio-$(PLATFORM)
CFLAGS = -ansi -Wall -g
CC = gcc

default: application

%: %.c
$(CC) $(CFLAGS) $(INCLUDES) $(DEFINES) -o $@ $< $(LIBS)
```



# Appendix D

## Changes

Release 11 of the IIO library incorporated the changes summarised below. Related changes in this document are highlighted by change-bars, as illustrated.

- The types `uint8_t`, `uint16_t` ..., defined in `types.h`, have been changed to `iio_uint8_t`, `iio_uint16_t` ... to avoid clashes with operating system header files.
- The number of digital channel types was expanded to discriminate between totem-pole and open-collector style output drivers and eliminate inconsistency in the use of `do` and `dio` channels. Channel types `oco` and `ocio` have been introduced. Module drivers for the GreenSpring IP-DIGITAL 24 and IP-WATCHDOG, and the VMIC VMIVME-2532A and VMIVME-2534 were modified to change their `dio` channels to `ocio`. The reversible digital in/out channel type `rdio` was also introduced. See Section 4.10.2.
- A system for handling modules with registers in non-mappable port address spaces was added, primarily for ISA 80x86 systems. Register addresses are resolved and mapped using the existing mechanism (except using the `iio_space_port` address space operation code), but the registers themselves must be accessed using function calls rather than the usual pointer dereferences (Section 5.5). Support is limited to 80x86 LynxOS systems.
- The module driver for the GreenSpring ATC-40 was extended to support the CSIRO/MST ATC-10 PC/104 bus carrier, which provides one IP slot.
- A module driver for the generic PC game port has been added.
- A module driver for the generic PC printer port has been added.
- A module driver for the Diamond DMM-32-AT PC/104 multi-IO module has been added.



# Appendix E

## IIO Proposal Document

### A Uniform Interface to Industrial IO Hardware

R. J. Kirkham

May 1996

## 1 Introduction

Many projects within the Industrial Automation Programme in recent years have highlighted the need for a uniform approach to controlling industrial input/output (IO) hardware from application programs. Industrial IO hardware refers to devices such as analogue-to-digital converters (ADCs), digital-to-analogue converters, bit-wise digital IO such as lamp or solenoid drivers, incremental encoder counters, timers, interrupters, and so on.

This ‘industrial’ IO differs from the more traditional computer IO hardware, such as terminals, video displays, or disk drives in several ways. Firstly, traditional IO devices are accessed by an application program in a ‘logical’ fashion through the computer’s operating system. The application neither cares about or directly accesses the underlying hardware of the device, but uses high-level operations such as `open()`, `read()`, and `write()`.

Industrial IO, however, is almost never supported within a standard operating system, and industrial IO hardware, with the exception of laboratory-style IO for MS-DOS systems, frequently comes without any supporting software. This means application programmers must write a ‘device driver’ for the hardware, often from scratch. These device drivers tend to implement only the capability or style of software interface required for the application, because they are viewed as being part of that application. This limits their portability and usefulness to others.

A second difference between traditional and industrial IO is the way in which it is used. Devices such as terminals and disks, and their associated software, tend to handle large amounts of data, so throughput is emphasised over delay. Buffering is commonly used, as the real-time response from application program through to device is generally not important.

On the other hand, industrial IO tends to deal in much smaller data rates, and delay must be minimised. This is because the application is frequently real-time in nature, and so the timing of the IO is controlled by, or influences, the application program itself.

Ideally, a unified industrial IO interface would provide a separation between logical, generic industrial IO devices, and the actual hardware, while retaining the assured real-time performance of a bespoke application program resident device driver.

## 2 Proposed IO Model

### 2.1 A Flavour of the Interface

From the application programmer’s point of view, all industrial IO channels would appear organised in named arrays according to generic type. Thus, there might

be an array of bit-wise digital input channels, digital output channels, and ADC channels. Each channel within each type would be sequentially numbered, and be given names such as `do.0`, `di.34` or `adc.3`.

An application program would then ‘open’ a channel it needed to use using a function call like:

```
io chan;  
chan = io_open("di.34");
```

where `chan` is a ‘handle’ for the open file, essentially like a file descriptor in UNIX. It would then use the handle for subsequent operations on the channel:

```
int result;  
io_operate(chan, IIO_READ, &result);
```

When finished with the channel, it would close it:

```
io_close(chan);
```

These function would, of course, return some status indication, which a real application would examine and act upon.

## 2.2 Channels, Devices and Names

Channels would be the smallest individually operable unit. For bit-wise digital IO, a channel would be 1 bit wide; for an ADC it might be 12 or 14 bits. Each channel would be available through a number of names. As well as the type grouping suggested above, perhaps other type groupings could be used. Hence, `adc12.3`, a 12-bit ADC, might also be known as `adc.11`, if, for instance, the first eight ADCs were not 12-bits.

Of more direct use would be to associate meaningful names (or aliases) to the channels. Thus, `adc.3` might also be opened as `fuel-level` and `do.12` as `fuel-solenoid`. This means that applications become more readable, and don’t need to be changed if the channel for a particular external device is re-wired.

Channels are ultimately implemented by hardware devices, which might also have names. This would permit operations on all the channels on a device, or alternatively provide yet another naming scheme, so that `ipadio.2:dio.12` might refer to digital IO channel 12 on IP-ADIO 2.

## 2.3 Channel Ranges

Channels could be opened in ranges. This would allow a contiguous (or possibly even discontinuous) set of channels of the same type to be opened and operated on with the one handle:

```
int position[3];  
io chan;  
chan = io_open("adc.0-2");  
io_operate(chan, IIO_READ, position);  
io_close(chan);
```

Here the range of three ADCs, perhaps representing the  $(x, y, z)$  position of something, are opened as a range, and read as a set into an array `position`. Slightly different semantics would apply for bit-wise digital IO, where bit-packing into a single integer variable is more useful.

---

## 2.4 Interface System Structure

The interface system would have a two-tiered internal structure: a generic core containing the application programmer's interface (`io_open()`, and so on), and a set of device driver modules, one for each type of industrial IO device required.

The core would contain the mechanism for interpreting the channel names and ranges, and calling the appropriate device driver module(s) for each operation. Many operations, especially range operations, will need to be broken down into a sequence of simpler ones, which may be distributed over a number of drivers.

The device drivers would be written to conform to a well-defined interface standard, so that the IO system can be easily expanded. A driver would have entry points for opening, operating, and closing (matching the three main application level functions), plus an initialisation function. The driver initialisation function is called once for each device of that type in the system, and establishes a device state structure, pointers to the hardware registers, and so on.

## 2.5 Configuration File

On its initialisation, the interface system core would read a configuration file, which tells it exactly what IO devices are available for use, and what names by which they will be known. The information would include the model of the device, its physical base address, and jumper or other configuration.

A typical configuration file might look like this:

```
# a configuration file
device vmivme4100 vme16 0x1280
device xyx230 vme24 0x400000 j23 j34
device vipc600 vme16 0x4000

alias fuel-level vmivme4100.0:adc.3
alias fuel-solenoid vipc600.0:do.6
```

This indicates a VMIVME-4100 board on the VMEbus A16 address-space at address 0x1280, an XYZ-230 board at A24 0x400000, and a VIPC-600 Industry-Pack carrier board at A16 0x4000. The options on the second line indicate to the XYZ-230 driver that jumpers J23 and J24 are on, which might be something the driver needs to know.

As the configuration file is read, the interface module core would call the initialisation function for the device, which would return the number and type of the actual IO channels provided by it. The core would then create channel name entries for the channels, and link them back to the driver.

Self-identifying modules, such as IndustryPacks, would be not be entered into the configuration file. The carrier module only needs to be entered, and its device driver would initialise the appropriate device drivers for the modules it found on the carrier. A similar scheme would operate for self-identifying VMEbus boards (a recent addition to the VMEbus specification), serial bus-addressable remote devices, or any other self-identifying interface.

Alias names would be contained in the configuration file as well, as shown in the example. So that shared-memory operating systems, or computers without file-systems can use the system, the configuration file could also be parsed from a memory string.

## 2.6 Channel Operations

While most operations on channels are generic (essentially read and write), the wide variety of industrial IO devices suggests allowing an extendible framework of operations. New operations for specific device types will need to be added

as required, and device drivers will need to check if a requested operation is permitted.

For this reason a single operation function `io_operate()` is proposed, with an integer operation code. In extending the operation repertoire, however, care needs to be exercised to prevent a proliferation of ‘the same but different’ operation codes. As each new device type is added, a generic model for that type of device needs to be developed, so that other devices of the same type will share the same operation codes.

Even with an extendible set of operations, some complex devices will not fit neatly into such a structure. In the end, it may be necessary for a special ‘cover all’ device driver to allow applications direct access to the device registers. The interface system would then act only as a rather complicated address resolution mechanism.

The most difficult type of device is one that is reconfigurable or modal. Timers and parallel digital IO chips are often like this. Possibly the selection of mode can be done within the configuration file, as an option to the `device` line, with some other solution for self-identifying devices.

## 2.7 Channel Scaling

Since each channel’s data passes through a device-specific driver, the driver can perform scaling or processing so that the application deals only in real-world units. For instance, an ADC driver could report results as a floating-point number in volts, rather than the raw count (although access to the raw values should always be retained). Calibration offset and rate factors could also be included by some means.

Scaling or other processing could also be performed by the interface system core, on a channel by channel basis, using parameters defined in the configuration file. In this way, an LVDT sensor connected to an ADC channel could be read by the application program in millimetres, simply by including the LVDT rate (millimetres per volt) in the file.

## 2.8 Fake Devices and Channels

A difficulty often arises with application software development for systems that involve IO devices of this kind. Frequently there is only one fully built system available, but several programmers want to test the application software at once.

To permit software testing, the hardware could be simulated by substituting ‘fake devices’ for the real ones in the configuration file. In this way the applications need not be changed to run on systems without the actual hardware, or with only part of it.

These fake devices could also be linked with temporary application code to simulate the expected behaviour of the device and the system it is connected to. This could even extend to simulating the global behaviour of the system, including the response of fake inputs.

## 2.9 Interrupts, Events and Exceptions

There could also be a channel type for events, such as interrupts or hardware exceptions. This would permit easy linking of sections of application code to the IO system. While getting close to the realm of the operating system, it would be appropriate for devices like the IP-OPTO interrupter.



---

## 2.10 Channel Directories and Monitoring

The channel name-space would be available to application programs through some functions, to allow listings of the available devices and channels to be produced. This would then allow fairly generic monitoring programs to be written, which would display all the channels and their values on a screen, perhaps using Tcl/Tk, and allow interactive command of outputs.

## 2.11 Remote IO

A simple extension to the system is to permit computers on a network to transparently access IO devices on another computer. A stateless network protocol, such as RPC, would be suitable. This would be implemented within the core part of the interface system.

The configuration file on each computer would declare which IO devices would be available to specified other computers (i.e. *export* a device, or perhaps a channel of a device), and also which devices are *imported* from which host, and what their local name alias is.

To maintain reasonable levels of safety it may be necessary to limit the exporting of devices to certain user-names as well (as was done with the PIRAT Remote Tool Control system). Of course, performance through the network would be comparatively poor, but considerably more convenient for many applications.

# 3 Implementation

## 3.1 Location

A system along these lines is a quite self-contained module which could be separately developed in Melbourne, and subsequently brought back to CMTE Brisbane for proving and eventual use. Hopefully, the system would find use within Melbourne Laboratory as well.

Once a clear definition of the device driver interface was developed, writing of this code could be conducted in both locations.

## 3.2 Operating Systems and Devices

LynxOS is the principal real-time operating system used with the CMTE, so the initial version of the system will run on LynxOS. However, Melbourne Laboratory projects still typically use vxWorks, and may in the future use other shared-memory kernels, especially for micro-controller work, although LynxOS may fall into use there as well.

The system should then be in large part portable between the two operating systems. If this is achieved, portability to other systems is reasonably straightforward. Given the authors current experience, it is the LynxOS aspects of the project that may be the more time-consuming. However it is a LynxOS implementation that is required by CMTE.

Specifically for the Drag-line Automation project, an initial batch of device drivers are required. These are, in priority order, GreenSpring IP-PRECISION ADC, IP-DIGITAL 48, IP-WATCHDOG, IP-SERIAL, IP-16DAC, IP-DIGITAL 24, and IP-TIMER. There are also ADAM RS-485 addressable modules which may need to be integrated into the structure.

### **3.3 Timing**

Initially, a period of a few weeks would be required to flesh out the details of the interface specification (for both applications programs and device drivers) and refine aspects of the model. At that stage a reasonably firm idea of completion to a certain defined point (perhaps without the remote access or fancy monitoring facilities) could be given.

### **3.4 Facilities Required**

A PC or similar running LynxOS will be needed reasonably early on in Melbourne, so the general approach to the system implementation can be determined (e.g. memory mapping, loadable kernel drivers, ...). This would later require an IP carrier board, plus all the IP modules mentioned above. Facilities for testing on vxWorks exist already in Melbourne.

# Appendix F

## Header Files

### F.1 Header File iio.h

```
#ifndef _iio_h_
#define _iio_h_
/*
 * This file is part of IIO, the Industrial IO Library
 * CSIRO Division of Manufacturing Technology
 * $Id: iio.h,v 1.9 2000/09/11 04:32:39 kir092 Exp $
 *
 * iio.h -- user level function header
 * Robin Kirkham, June 1996
 */

/* global channel or channel range identifier */
typedef struct IIO_OPEN * IIO;

/* the config code, used by iio_init() */
typedef enum {
    iio_config_none,           /* no configuration */
    iio_config_stdfile,       /* read configuration from standard file */
    iio_config_file,          /* read configuration from a file */
    iio_config_string         /* read configuration from a string */
} IIO_CONFIG;

/* the operation code, used by iio_operate() */
typedef enum {
    /* generic operation codes */
    iio_op_nop,               /* no operation */
    iio_op_show,              /* log status of driver/module */
    iio_op_read,              /* read input from the channel */
    iio_op_readback,          /* read previous output from the channel */
    iio_op_write,             /* write output to channel */
    iio_op_clear,             /* write zero to channel */

    /* space resolution codes, always ORed with an IIO_SIZE */
    iio_space_io,             /* resolve input/output space */
    iio_space_id,             /* resolve identity space */
    iio_space_int,            /* resolve interrupt space */
    iio_space_mem,            /* resolve memory space */
    iio_space_mem16,          /* resolve 16-bit memory space */
    iio_space_mem24,          /* resolve 24-bit memory space */
    iio_space_mem32,          /* resolve 32-bit memory space */

    /* servo controller codes. These are a bit LM628-specific */
    iio_sc_start,             /* start servo with new target/settings */
    iio_sc_stop,              /* stop motion */
    iio_sc_free,              /* disable servo */
    iio_sc_read_current,      /* read current servo value */
    iio_sc_read_target,       /* read target servo value */
    iio_sc_write_current,     /* write (calibrate) current position */
    iio_sc_write_target,      /* write target position */
    iio_sc_write_target_dt,   /* write trapezoidal dt (e.g. velocity) */
    iio_sc_write_target_ddt,  /* write trapezoidal ddt (e.g. acceleration) */
    iio_sc_read_index,        /* read last index value */
    iio_sc_read_gain_p,       /* read loop proportional gain */
    iio_sc_read_gain_d,       /* read loop derivative gain */
    iio_sc_read_gain_i,       /* read loop integral gain */
    iio_sc_write_gain_p,      /* write loop proportional gain */
    iio_sc_write_gain_d,      /* write loop derivative gain */
    iio_sc_write_gain_i,      /* write loop integral gain */

    /* Adam module codes */
    iio_adam_message,         /* exchange command/reply packet */

    /* more space resolution codes, always ORed with an IIO_SIZE */
    iio_space_port,           /* resolve non-mappable port space */

    IIO_NOPs
} IIO_OP;
```

```
/* init flag type */
typedef enum {
    iio_iflag_none = 0x0,      /* no flags */
    iio_iflag_log = 0x1       /* log some major events */
} IIO_IFLAG;

/* opening flag type (EXAMPLES, NOT IMPLEMENTED YET) */
typedef enum {
    iio_oflag_none = 0x0,      /* no flags */
    iio_oflag_log = 0x1,      /* log operations on this channel */
    iio_oflag_excl = 0x2      /* open exclusively (NOT IMPLEMENTED) */
} IIO_OFLAG;

/* return status values */
typedef enum {
    iio_status_ok = 0,         /* all is well */
    iio_status_error = -1,     /* an error */
    iio_status_fatal = -2      /* a serious error */
} IIO_STATUS;

/* type of module information function */
typedef IIO_STATUS (* IIO_INFOFN)(void);

/* the user functions */
extern IIO_STATUS
iio_init(IIO_INFOFN list[], IIO_IFLAG flags),
iio_open(char *name, IIO_OFLAG flags, IIO *channel),
iio_close(IIO channel),
iio_done(void);

/* family of operate functions */
extern IIO_STATUS
iio_operate(IIO channel, IIO_OP operation, int data[]),
iio_operate_real(IIO channel, IIO_OP operation, double data[]),
iio_operate_addr(IIO channel, IIO_OP operation, void *addr[]);

/* family of operate functions for inwards simple channels */
extern IIO_STATUS
iio_operate_in(IIO channel, IIO_OP operation, int data),
iio_operate_inreal(IIO channel, IIO_OP operation, double data),
iio_operate_inaddr(IIO channel, IIO_OP operation, void *addr);

/* error message access function */
extern char
*iio_emessage_get();

/* the standard module driver list */
extern IIO_INFOFN iio_standard[];

#endif
```

## F.2 Header File internal.h

```

#ifndef _internal_h_
#define _internal_h_
/*
 * This file is part of IIO, the Industrial IO Library
 * CSIRO Division of Manufacturing Technology
 * $Id: internal.h,v 1.23 2000/09/11 04:32:39 kir092 Exp $
 *
 * internal.h -- IIO internal data structures and functions
 * Robin Kirkham, June 1996
 */

#include "iio.h"
#include "types.h"
#include <stdarg.h>

/*
 * ----- ENUMERATIVE TYPES
 */

/* module argument parser types */
typedef enum {
    iio_arg_bool,           /* boolean "-flag" or "-no-flag" */
    iio_arg_int8,           /* 8-bit signed */
    iio_arg_int16,          /* 16-bit signed */
    iio_arg_int32,          /* 32-bit signed */
    iio_arg_int64,          /* 64-bit signed */
    iio_arg_uint8,          /* 8-bit unsigned */
    iio_arg_uint16,         /* 16-bit unsigned */
    iio_arg_uint32,         /* 32-bit unsigned */
    iio_arg_uint64,         /* 64-bit unsigned */
    iio_arg_float,          /* float */
    iio_arg_double,         /* double */
    iio_arg_addr,           /* any address */
    iio_arg_string,         /* a dynamic string */
    iio_arg_channel,        /* an IIO channel descriptor */
    iio_arg_file,           /* a file-descriptor */

    IIO_NARG
} IIO_ARG;

/* type of channel/module alias */
typedef enum {
    iio_atype_none,         /* not an alias */
    iio_atype_global,       /* global alias */
    iio_atype_local,        /* local alias */
    iio_atype_module        /* module alias */
} IIO_ATYPE;

/* the four forms of channel name */
/* boolean flag type */
typedef enum {
    iio_bool_false = 0,
    iio_bool_off = 0,
    iio_bool_no = 0,
    iio_bool_true = 1,
    iio_bool_on = 1,
    iio_bool_yes = 1
} IIO_BOOL;

/* the channel type */
typedef enum {
    iio_ctype_di,           /* digital in */
    iio_ctype_do,           /* digital out */
    iio_ctype_dio,          /* digital in/out */
    iio_ctype_oco,          /* open collector out */
    iio_ctype_ocio,         /* open collector in/out */

    iio_ctype_bi,           /* bitwise digital in */
    iio_ctype_bo,           /* bitwise digital out */
    iio_ctype_bio,          /* bitwise digital in/out */
    iio_ctype_boco,         /* bitwise open collector out */
    iio_ctype_bocio,        /* bitwise open collector in/out */

    iio_ctype_adc,          /* analogue to digital */
    iio_ctype_dac,          /* digital to analogue */
    iio_ctype_enc,          /* positional encoder */
    iio_ctype_rdio,         /* reversible digital in/out */

```

```

        iio_chtype_isa,          /* ISA space */
        iio_chtype_ip,          /* IP space */
        iio_chtype_null,        /* the null device */
        iio_chtype_vme,         /* VMEbus space */
        iio_chtype_sc,          /* servo controller */
        iio_chtype_adam,        /* Adam 4000 address */
        IIO_NCHTYPES            /* LAST */
} IIO_CHTYPE;

/* file open attributes */
typedef enum {
    iio_fattr_rdonly = (1 << 0), /* open for read */
    iio_fattr_wronly = (1 << 1), /* open for write */
    iio_fattr_rdwr = (1 << 2),   /* open for read and write */
    iio_fattr_append = (1 << 3), /* start at end of file */
    iio_fattr_creat = (1 << 4),  /* create if not there */
    iio_fattr_trunc = (1 << 5),  /* truncate on open */
    iio_fattr_excl = (1 << 6),   /* access exclusively */
} IIO_FATTR;

/* structure magic numbers */
typedef enum {
    iio_magic_minfo = 0x4d494e46, /* "MINF" */
    iio_magic_module = 0x4d4f444c, /* "MODL" */
    iio_magic_chnode = 0x43484e4f, /* "CHNO" */
    iio_magic_chinfo = 0x4348494e, /* "CHIN" */
    iio_magic_ipinfo = 0x4d50494e, /* "IPIN" */
    iio_magic_map = 0x564d4150,    /* "VMAP" */
    iio_magic_name = 0x4e414d45,   /* "NAME" */
    iio_magic_open = 0x4f50454e,   /* "OPEN" */
    iio_magic_opnode = 0x4f504e4f, /* "OPNO" */
    iio_magic_sentinel = 0x4d4d4f4c, /* "IIOL" */
    iio_magic_state = 0x53544154,  /* "STAT" */
} IIO_MAGIC;

/* mapping type */
typedef enum {
    iio_maptype_memory,
    iio_maptype_port
} IIO_MAPTYPE;

/* how many times a module can be installed */
typedef enum {
    iio_multi_no,
    iio_multi_yes
} IIO_MULTTI;

/* format of a channel name */
typedef enum {
    iio_nform_gg,                /* global generic form */
    iio_nform_gs,                /* global specific form */
    iio_nform_lg,                /* local generic form */
    iio_nform_ls                 /* local specific form */
} IIO_NFORM;

/* kind of operation code argument */
typedef enum {
    iio_oparg_none = 0x00,        /* no arguments */
    iio_oparg_in = 0x01,          /* inward argument */
    iio_oparg_out = 0x02,         /* outward argument */
    iio_oparg_inout = 0x03,       /* in and out argument */
    iio_oparg_data = 0x10,        /* data argument */
    iio_oparg_addr = 0x20         /* address argument */
} IIO_OPARG;

/* type of probing */
typedef enum {
    iio_ptype_read = 0x1,         /* try reading */
    iio_ptype_write = 0x2,        /* try writing */
    iio_ptype_rdwr = 0x3          /* try reading and writing */
} IIO_PTYPE;

/* mask/roll of IIO_SIZE when rolled into an operation code */
#define IIO_SPACE_MASK ((1 << 8) - 1)
#define IIO_SIZE_MASK (IIO_SPACE_MASK << 8)

/* probe/resolve register/bus access size types */
typedef enum {
    iio_size_8 = (1 << 8),        /* byte access */
    iio_size_16 = (2 << 8),       /* word access */
    iio_size_32 = (4 << 8),       /* long word access */

```

```

    iio_size_64 = (8 << 8)                /* really long word access */
} IIO_SIZE;

/* tfile access method */
typedef enum {
    iio_tfile_file,                        /* file descriptor */
    iio_tfile_filename,                   /* file name */
    iio_tfile_string                       /* string pointer */
} IIO_TFILE;

/* TTY control tags */
typedef enum {
    iio_ttypar_none = 0x0,
    iio_ttypar_even = 0x1,
    iio_ttypar_odd = 0x2,
    iio_ttypar_ignore = 0x10,
    iio_ttypar_mark = 0x20
} IIO_TTYPAR;

typedef enum {
    iio_ttyflow_none = 0x0,
    iio_ttyflow_rtscts = 0x1,
    iio_ttyflow_xonxoff = 0x2,
    iio_ttyflow_xonany = 0x4,
    iio_ttyflow_hupcl = 0x10
} IIO_TTYFLOW;

typedef enum {
    iio_ttymap_instrip = 0x1,
    iio_ttymap_incrnl = 0x2
} IIO_TTYMAP;

typedef enum {
    iio_ttyecho_none = 0x0,
    iio_ttyecho_echo = 0x1
} IIO_TTYECHO;

typedef enum {
    iio_udata_int,                         /* integer user data */
    iio_udata_real,                       /* real (double) user data */
    iio_udata_addr,                       /* address user data */
    iio_udata_bit                          /* bitfield data */
} IIO_UDATA;

/*
 * ----- FORWARD DECLARATIONS
 */

typedef struct IIO_ADAMINFO IIO_ADAMINFO;
typedef struct IIO_ALIAS IIO_ALIAS;
typedef struct IIO_OPEN IIO_OPEN;
typedef struct IIO_CHNODE IIO_CHNODE;
typedef struct IIO_CHINFO IIO_CHINFO;
typedef struct IIO_OPNODE IIO_OPNODE;
typedef struct IIO_IPINFO IIO_IPINFO;
typedef struct IIO_MAP IIO_MAP;
typedef struct IIO_MINFO IIO_MINFO;
typedef struct IIO_MODULE IIO_MODULE;
typedef struct IIO_MREG IIO_MREG;
typedef struct IIO_MSTATE IIO_MSTATE;
typedef struct IIO_NAMEEX IIO_NAMEEX;
typedef struct IIO_OPINFO IIO_OPINFO;
typedef struct IIO_SENTINEL IIO_SENTINEL;
typedef struct IIO_STATE IIO_STATE;
typedef struct IIO_SLL IIO_SLL;

/* opaque mutex and file types */
typedef void IIO_MUTEX;
typedef void IIO_SHMUTEX;
typedef int IIO_FILE;

/*
 * ----- FUNCTION POINTER TYPES
 */

/* type of a module install function */
typedef IIO_STATUS (* IIO_INSTALLFN)(
    IIO_MODULE *module, char *argv[]
);

/* type of a module init function */

```

```
typedef IIO_STATUS (* IIO_INITFN)(
    IIO_MREG *reg, IIO_MSTATE *state
);
/* type of a module operate function */
typedef IIO_STATUS (* IIO_OPFN)(
    IIO_MSTATE *state, IIO_MREG *reg, IIO_OPNODE*,
    IIO_OP op, unsigned first, unsigned last
);
/* compare function type */
typedef int (* IIO_CMPFN)(IIO_SLL*, IIO_SLL*);

/* tfile callback type */
typedef IIO_STATUS (* IIO_TEXEC)(char *argv[]);

/*
 * ----- STRUCTURE DEFINITIONS
 */

struct IIO_ADAMINFO {
    /*
     * Information on a particular range setting for an ADAM module.
     * A table of these, indexed by the range code, is used to look up
     * the scaling constants where required
     */
    double
        escale,                /* scale factor (eng mode) */
        hscale;                /* scale factor (hex mode) */
    char *unit, *range;        /* user unit, range/type string */
};

struct IIO_ALIAS {
    /*
     * Linked list of aliases of either whole channels or channel ranges,
     * modules (not ranges) or local channels and ranges. An alias is
     * actually stored as name-value strings, and is resolved only when
     * invoked in a channel open
     */
    IIO_ALIAS *next;

    IIO_ATYPE type;           /* type of alias */
    char *name;               /* the alias itself */
    char *value;              /* the thing aliased */
};

struct IIO_CHNODE {
    /*
     * List of available channels. Each element identifies a group of
     * channels of the same type and width on a particular module, and
     * contains the pointer to the operation function. This list is built
     * as the modules are installed, and is searched when modules are
     * opened
     */
    IIO_CHNODE *next;        /* linked list pointer */
    IIO_MAGIC magic;         /* magic number */

    unsigned int number;     /* number of these channels in group */
    IIO_CHTYPE type;         /* channel type */
    unsigned int width;      /* channel width in bits */

    /* sequence numbers of first channel in this group */
    unsigned int seqno[4];   /* one for each of the four name forms */

    IIO_MODULE *module;      /* the module that provides them */
    IIO_OPFN operate;        /* the operation function pointer to use */

    IIO_CHINFO *chinfo;      /* array of information on each channel */

    /* extra data for the bitfield pseudo-driver */
    IIO_CHNODE *rchnode;     /* real chnode */
    unsigned int rseqno;     /* real seqno */
};

struct IIO_CHINFO {
    /*
     * Specific information on a single channel. There is an array of these
     * structures in each IIO_CHNODE, one per local channel. There is a
     * default linear scaling, and later perhaps provision for other
     * conversions. The limits and initial values of the channel are also
     * stored here
     */
}
```



```

        IIO_MAGIC magic;           /* magic number */

        double scale, offset;      /* standard linear conversion */
        char *unit;                /* units after linear conversion */

        int initial;               /* initial channel value */
        int upper, lower;          /* limits apply if upper>lower */
        IIO_BOOL log;              /* logging flag */
};

struct IIO_IPINFO {
    /*
     * This structure is used by IndustryPack Carrier cards. It contains a
     * duplicate of the IP's IDPROM (read as 16-bit words), plus reconstructed
     * forms of the manufacturer/product code and the driver ID code.
     */
    IIO_MAGIC magic;              /* magic number */

    iio_uint32_t mid;              /* manufacturer code */
    iio_uint32_t pid;              /* product code */
    iio_uint32_t rev;              /* module revision number */
    iio_uint32_t did;              /* driver ID code */
    iio_uint32_t flg;              /* flags word */

    iio_uint16_t *pprom;           /* pointer to pack data in data[] */
    iio_uint16_t *uprom;           /* pointer to user data in data[] */

    /* the original data from the IDPROM */
    iio_uint16_t prom[0x20];
};

struct IIO_MINFO {
    /*
     * The minfo structure contains information about a particular
     * module driver. Minfos are added to the minfo list with iio_minfo(),
     * which is the first (and usually only) act of a module information
     * function
     */
    IIO_MINFO *next;              /* linked list pointer */
    IIO_MAGIC magic;              /* magic number */

    char *ident;                   /* the ident code of the module */
    char *model;                   /* manufacturer, module and description */
    char *version;                 /* module driver RCS version, date etc */
    IIO_MULTI multi;               /* how many times this module can be used */
    IIO_INSTALLFN install;         /* the install function pointer */
    IIO_INITFN init;               /* the init function pointer */
};

struct IIO_MODULE {
    /*
     * The module list contains one element for each installed module.
     * The module list is extended by iio_module()
     */
    IIO_MODULE *next;              /* linked list pointer */
    IIO_MAGIC magic;              /* magic number */

    IIO_MINFO *minfo;              /* module information block */
    unsigned int seq;              /* module sequence number */
    IIO_BOOL log;                  /* log operations on this module */

    IIO_MSTATE *state;             /* global shared state */
    IIO_MREG *reg;                 /* module register pointers */
    IIO_SHMUTEX *mutex;            /* module/state mutex */
};

struct IIO_NAMEX {
    /*
     * The namex structure is filled out (not created) by iio_namex, and
     * contains the broken-down (expanded) form of a channel name. The
     * type field indicates the type of the original name. The module
     * ident and sequence numbers are only valid for lg and ls forms
     */
    IIO_MAGIC magic;              /* magic number */
    IIO_NFORM form;                /* form of name */

    IIO_MODULE *module;            /* module pointer */
    IIO_CHTYPE type;               /* channel type */
    unsigned int width;            /* channel width */
    unsigned int seq1, seq2;       /* channel sequence number range */
};

```

```

struct IIO_OPEN {
    /*
     * Open channel descriptor. There is one of these for each open
     * channel or channel range, linked into a single list. A channel may
     * represent a single simple channel on a module, or a contiguous range
     * of channels (but only of the same type) spread over several modules
     * (one IIO_OPNODE for each)
     */
    IIO_OPEN *next;           /* main linked list pointer */
    IIO_MAGIC magic;          /* magic number */

    unsigned number;          /* number of simple channels in range */
    char *name;               /* name under which this channel was opened */
    IIO_MUTEX *mutex;         /* channel mutex (if a range) */
    IIO_OPNODE *opnode;       /* list of channel nodes */
};

struct IIO_OPNODE {
    /*
     * A list of these structures hangs off the channel descriptor blocks.
     * When a channel is opened by iio_open(), one of these gets
     * created for each chnode associated with the open channel
     */
    IIO_OPNODE *next;         /* main linked list pointer */
    IIO_MAGIC magic;          /* magic number */

    IIO_CHNODE *chnode;       /* channel info block */
    unsigned int first;        /* first channel seqno in range (ls form) */
    unsigned int number;       /* number in the range (local) */
    unsigned int index;        /* index of corresponding user data element */
    IIO_BOOL log;              /* log operations on this channel */

    /*
     * This information changes each operation function call,
     * but is protected by the module semaphore
     */
    IIO_UDATA udata;          /* type of user data */
    void *base;               /* base of user data */
    IIO_OP op;                 /* the current operation code */
};

struct IIO_OPINFO {
    /*
     * Operation info structure. There is an array iio_opinfo[] of these,
     * indexed by the IIO_OP operation code. It is used to tell how many
     * arguments of what type and what direction
     */
    char *name;               /* the name of the operation */
    char *symbol;              /* a symbol for the log message */
    IIO_OPARG arg;             /* kind and direction of the argument */
};

struct IIO_SENTINEL {
    /*
     * The sentinel structure is the first IIO strincture in shared memory.
     * It contains a timestamp from the IIO library itself (output from the
     * date command) and from the configuration file. If both of these match
     * this the remaining shared memory should be consistent between processes
     */
    IIO_MAGIC magic;          /* magic number */

    double library_time;       /* "mtime" of library */
    unsigned long config_time; /* mtime of config file */
};

struct IIO_SLL {
    /*
     * All the linked list structures (i.e., most of the above) are
     * ordered, and the ordered insert is done using iio_sll_insert().
     * This requires the *next element TO BE FIRST in these structures.
     * struct IIO_SLL is a dummy structure, with a *next element only
     */
    IIO_SLL *next;
};

struct IIO_STATE {
    /*
     * The state structure contains the heads of the main list data
     * structures, and the protection mutex for them all
     */

```

```

    IIO_MAGIC magic;          /* magic number */

    IIO_ALIAS *alias;         /* head of alias list */
    IIO_OPEN *open;          /* head of open channel list */
    IIO_CHNODE *chnode;       /* head of available channel list */
    IIO_MAP *map;             /* head of available map list */
    IIO_MODULE *module;       /* head of installed module list */
    IIO_MINFO *minfo;         /* head of available module drivers list */

    IIO_Mutex *mutex;         /* per-process core state mutex */
    IIO_Mutex *omutex;        /* per-process open-channel list mutex */

    IIO_BOOL init;            /* initialise, as well as install modules */
    IIO_BOOL log;             /* generate some messages */
};

struct IIO_MAP {
    /*
     * IIO_MAP structures represent a mapping from physical memory to
     * virtual (logical, or process) memory, usually to access module
     * registers. The mappings are created through the OS function accessed
     * using iio_shmap_alloc(), but the OS generally won't reverse-lookup
     * physical addresses, so we need to keep a record of the mappings
     */
    IIO_MAP *next;           /* linked list pointer */
    IIO_MAGIC magic;         /* magic number */

    IIO_MAPTYPE type;        /* mapping type (memory or port) */
    void *paddr;             /* physical address of base of mapping */
    void *vaddr;             /* virtual address of base of mapping */
    unsigned size;           /* size of mapping in bytes */
};

/*
 * ----- IIO LIBRARY FUNCTIONS/GLOBALS
 */

extern IIO_STATUS
    iio_adam_dread(char *dex, int digits, int *result),
    iio_adam_hread(char *hex, int digits, int *result),
    iio_adam_hwrite(int value, int digits, char *string);
extern IIO_ADAMINFO
    iio_adam_info[];

extern IIO_STATUS
    iio_alias(char *argv[]),
    iio_alias_insert(IIO_ATYPE type, char *name, char *value),
    iio_alias_find(char *name, IIO_ATYPE *atype, char **value),
    iio_alias_show(void);

extern IIO_STATUS
    iio_arg(char *argv[], char *option, IIO_ARG type, ... ),
    iio_arg_index(char *argv[], char *option, unsigned ind, IIO_ARG type, ... ),
    iio_arg_list(char *argv[], ... ),
    iio_arg_index_list(char *argv[], unsigned index, ... ),
    iio_arg_remnants(char *argv[]);
extern char
    *iio_arg_blank;

extern IIO_INFOFN
    iio_builtin[];

extern IIO_STATUS
    iio_channel(char *argv[]);

extern IIO_STATUS
    iio_chnode(
        IIO_MODULE *module, IIO_CHTYPE type, unsigned int width,
        unsigned int number, IIO_OPFN operate, IIO_CHNODE **chnode
    ),
    iio_chnode_linear(IIO_CHNODE *, unsigned, double sc, double of, char *un),
    iio_chnode_limits(IIO_CHNODE *, unsigned, int initial, int low, int high),
    iio_chnode_log(IIO_CHNODE *, unsigned, IIO_BOOL),
    iio_chnode_show(IIO_BOOL longform);
extern int
    iio_chnode_cmp(IIO_SLL *s1, IIO_SLL *s2);

extern IIO_STATUS
    iio_ctype_find(char *string, IIO_CHTYPE *type, unsigned int *width);
extern char

```

```
*iio_chtype_string[];

extern IIO_STATUS
    iio_config_exec(char *argv[]);

extern int
    iio_data_get(IIO_OPNODE *opnode, unsigned seqno);
extern double
    iio_data_get_real(IIO_OPNODE *opnode, unsigned seqno);
extern void
    *iio_data_get_addr(IIO_OPNODE *opnode, unsigned seqno);
extern IIO_STATUS
    iio_data_set(IIO_OPNODE *opnode, unsigned seqno, int value),
    iio_data_set_real(IIO_OPNODE *opnode, unsigned seqno, double value),
    iio_data_set_addr(IIO_OPNODE *opnode, unsigned seqno, void *value);

extern IIO_STATUS
    iio_done_iio(void);

extern IIO_STATUS
    iio_init_iio(
        IIO_INFOFN list[], IIO_IFLAG flags,
        IIO_BOOL init, IIO_TFILE method, ...
    );

extern IIO_STATUS
    iio_ipinfo_read(IIO slot, IIO_IPINFO *ipinfo),
    iio_ipinfo_ident(IIO slot, iio_uint32_t mid, iio_uint32_t pid),
    iio_ipinfo_show(IIO slot);

extern IIO_STATUS
    iio_map(IIO_OPEN *chan, IIO_OP space, unsigned lbase, unsigned lsize),
    iio_map_new(IIO_OP space, void *paddr, unsigned psize),
    iio_map_ptov(IIO_MAPTYPE type, void *paddr, unsigned size, void **vaddr),
    iio_map_vtop(IIO_MAPTYPE type, void *vaddr, unsigned size, void **paddr),
    iio_map_type(IIO_OP space, IIO_MAPTYPE *type),
    iio_map_done(void),
    iio_map_show(void);

extern IIO_STATUS
    iio_minfo(
        const char *ident, const char *name, const char *version,
        IIO_MULTI multi, IIO_INSTALLFN install, IIO_INITFN init
    ),
    iio_minfo_call(IIO_INFOFN infon[]),
    iio_minfo_find(char *ident, IIO_MINFO **minfo),
    iio_minfo_show(void);

extern IIO_STATUS
    iio_module(char *argv[]),
    iio_module_state(IIO_MODULE *module, unsigned int size),
    iio_module_reg(IIO_MODULE *module, unsigned int size, IIO_MREG **reg),
    iio_module_find(char *string, IIO_MODULE **module),
    iio_module_show(void);

extern IIO_STATUS
    iio_namex(char *name, IIO_NAMEEX *namex);

extern IIO_STATUS
    iio_null(void);

extern IIO_STATUS
    iio_open_show(IIO_OPEN *chan);

extern IIO_OPINFO
    iio_opinfo[];
extern IIO_STATUS
    iio_opinfo_lookup(char *string, IIO_OP *operation),
    iio_opinfo_show(void);

extern IIO_STATUS
    iio_phantom(void);

extern IIO_STATUS
    iio_resolve(
        IIO_OPEN *chan, IIO_OP space, IIO_SIZE size,
        unsigned laddr, void **vaddr
    ),
    iio_resolve_physical(
        IIO_OPEN *chan, IIO_OP space, IIO_SIZE size,
```

```

        unsigned laddr, void **paddr
    ),
    iio_resolve_list(IIO_OPEN *chan, IIO_OP space, ...);

extern IIO_STATUS
    iio_return_error(char *emessage, char *file, unsigned int lineno),
    iio_return_fatal(char *emessage, char *file, unsigned int lineno);

extern IIO_STATUS
    iio_sll_insert(IIO_SLL **head, IIO_SLL *new, IIO_CMPFN cmp);

extern IIO_STATUS
    iio_state_init(void);
extern IIO_STATE
    *iio_state;

extern double
    iio_timestamp;

extern IIO_STATUS
    iio_tfile(IIO_TEXEC texec, IIO_TFILE method, ...),
    iio_tfile_stdarg(IIO_TEXEC texec, IIO_TFILE method, va_list ap);

/*
 * ----- FUNCTIONS IN OS MODULE
 *
 * There is one module per operating system which should contain all of the
 * functions below, which implement, emulate or re-package certain system calls
 * or standard library calls, so that the remainder of the library only
 * accesses the system through this module. The module is generally named
 * after the system, e.g. vxworks.c, lynxos.c, and so on.
 */

extern char
    *iio_configname;

extern IIO_STATUS
    iio_osinit(IIO_INFOFN list[], IIO_IFLAG iflags),
    iio_osdone(void);

extern IIO_STATUS
    iio_emessage_set(char *message);

    /* the old GCC does not understand __attribute__ */
#if (__GNUC__ == 2)
extern IIO_STATUS
    iio_log(char *fmt, ...) __attribute__((format(printf, 1, 2)));
#else
extern IIO_STATUS
    iio_log(char *fmt, ...);
#endif

extern IIO_STATUS
    iio_shmap_alloc(
        void *paddr, unsigned int psize,
        void **pactual, void **vaddr, unsigned int *vsize
    ),
    iio_shmap_free(void *vaddr, void *paddr, unsigned size);

extern IIO_STATUS
    iio_shmem_alloc(unsigned int size, void **new),
    iio_shmem_free(void *old);

extern IIO_STATUS
    iio_shmutex_create(IIO_SHMUTEX **mutex),
    iio_shmutex_grab(IIO_SHMUTEX *mutex),
    iio_shmutex_drop(IIO_SHMUTEX *mutex),
    iio_shmutex_free(IIO_SHMUTEX *mutex);

extern IIO_STATUS
    iio_mutex_create(IIO_MUTEX **mutex),
    iio_mutex_grab(IIO_MUTEX *mutex),
    iio_mutex_drop(IIO_MUTEX *mutex),
    iio_mutex_free(IIO_MUTEX *mutex);

extern IIO_STATUS
    iio_sem_wait(IIO_MUTEX *mutex),
    iio_sem_post(IIO_MUTEX *mutex);

extern IIO_STATUS

```

```
    iio_port_alloc(
        void *paddr, unsigned int psize,
        void **pactual, void **vaddr, unsigned int *vsize
    );
extern void
    iio_port_set8(volatile iio_uint8_t *addr, iio_uint8_t val),
    iio_port_set16(volatile iio_uint16_t *addr, iio_uint16_t val),
    iio_port_set32(volatile iio_uint32_t *addr, iio_uint32_t val);
extern iio_uint8_t
    iio_port_get8(volatile iio_uint8_t *addr);
extern iio_uint16_t
    iio_port_get16(volatile iio_uint16_t *addr);
extern iio_uint32_t
    iio_port_get32(volatile iio_uint32_t *addr);

extern IIO_STATUS
    iio_probe(volatile void *vaddr, IIO_SIZE size, IIO_PTYPE ptype);

extern IIO_STATUS
    iio_roughdelay(unsigned milliseconds);

extern int
    iio_round(double x);

extern IIO_STATUS
    iio_tty_line(
        IIO_FILE tty,
        unsigned baud, unsigned nchar, unsigned nstop,
        IIO_TTYPAR parity, IIO_TTYFLOW flow, IIO_TTYMAP map,
        IIO_TTYECHO echo
    ),
    iio_tty_raw(
        IIO_FILE tty,
        unsigned baud, unsigned nchar, unsigned nstop,
        IIO_TTYPAR parity, IIO_TTYFLOW flow, IIO_TTYMAP map,
        unsigned char min, unsigned char time
    ),
    iio_tty_send(IIO_FILE tty, char *buffer, unsigned len),
    iio_tty_recv(
        IIO_FILE tty, char *buffer, unsigned maxlen,
        unsigned *len, unsigned ms
    );

/*
 * ----- FUNCTIONS IN LIBC MODULE
 *
 * The module libc.c contains wrappers for C library (or standard system call)
 * functions that are used within the rest of the library. If the library is
 * only ever used at user level (in Unix/LynxOS), or on vxWorks and RTEMS
 * (which have a standard C library interface too), then we can probably get
 * rid of all these wrappers
 */

extern IIO_STATUS
    iio_mem_alloc(unsigned int size, void **new),
    iio_mem_free(void *old);

extern IIO_STATUS
    iio_string_lookup(char *token, char **table, int *index),
    iio_string_dup(char *string, char **new),
    iio_string_cpy(char *dst, char *src);
extern char
    *iio_string_chr(char *s, char c),
    *iio_string_pbrk(char *s, char *c);
extern int
    iio_string_cmp(char *s1, char *s2),
    iio_stringncmp(char *s1, char *s2, int len),
    iio_string_len(char *string);

extern IIO_STATUS
    iio_decode_long(char *string, long int *result),
    iio_decode_double(char *string, double *result);

    /* the old GCC does not understand __attribute__ */
#if (__GNUC__ == 2)
extern IIO_STATUS
    iio_slog(char *out, char *fmt, ...)
        __attribute__((format(printf, 2, 3)));
#else
extern IIO_STATUS
    iio_slog(char *out, char *fmt, ...);
#endif
```

```

#endif
extern IIO_STATUS
    iio_vslog(char *out, char *fmt, va_list ap);

#if (__GNUC__ == 2)
extern IIO_STATUS
    iio_exec(char *fmt, ...)
        __attribute__((format(printf, 1, 2)));
#else
extern IIO_STATUS
    iio_exec(char *fmt, ...);
#endif

extern IIO_STATUS
    iio_file_open(char *name, IIO_FATTR fattr, IIO_FILE *file),
    iio_file_close(IIO_FILE file);
extern char
    iio_file_getc(IIO_FILE file);

/*
 * ----- MISC
 */

/* use HIDDEN for local objects */
#define HIDDEN static

#ifndef NULL
#define NULL ((void *)0)
#endif

/* ----- ERROR MACROS
 *
 * This is a slightly embellished error-return-collapse arrangement. All
 * functions return IIO_STATUS, defined in iio.h to be an enumerative:
 *
 *      iio_status_ok           function executed successfully
 *      iio_status_error       function found non-fatal error
 *      iio_status_fatal       function found fatal error
 *
 * There are macros to generate the error returns, and macros to call functions
 * and test the result. This makes the code much more readable, as it is not
 * covered in messy if(...) statement (actually it is, but they are done by
 * macros so they aren't seen). It also allows cunning things to be done, like
 * printing file/line numbers and subroutine call traces for fatal errors.
 *
 * To return an error from a function, use iio_error() or iio_fatal():
 *
 *      return iio_error("Bad thingy whatsit");
 * or
 *      return iio_fatal("Really bad thingy whatsit");
 *
 * The string argument should be a string constant (as shown), or a global
 * or static string pointer; it is stored for later printing. It can be
 * retrieved using iio_essage_get(). If the error is from a system call
 * or operating system function, use iio_error_system(), or iio_fatal_system(),
 * to generate the message from the system errno or equivalent:
 *
 *      if ((fd = open( ... )) < 0)
 *          return iio_error_system;
 *
 * To call functions, you can use iio_eret(), or iio_fret()
 *
 *      iio_eret( function(arg, ...) );
 * or
 *      iio_fret( function(arg, ...) );
 *
 * iio_eret() calls function(), and if it returns fatal error status, logs
 * a message, and returns fatal error status. If it returns non-fatal error
 * status, no message is printed, and it returns error status; if ok status,
 * nothing is done.
 *
 * iio_fret() is the same, but only returns for fatal errors; non-fatals
 * DO NOT cause a return and it is up to the code at this point to deal with
 * the error (using a variable to catch the status).
 */

```

```
    * This macro is used when generating errors, usually used with
    * a return statement. The S argument should be a static string pointer.
    */
#define iio_error(S) iio_return_error((S), __FILE__, __LINE__)

/*
 * This macro is for generating FATAL errors; as for the above, but logs
 * a message indicating where it was detected, as well as printing S
 */
#define iio_fatal(S) iio_return_fatal((S), __FILE__, __LINE__)

/*
 * This macro evaluates S once (usually a function call) and returns from
 * the calling function if S is an error. This is used to pass up errors
 * at lower levels. If S indicates a fatal error, a tracing message is
 * logged as well
 */
#define iio_eret(S) \
switch ((S)) { \
case iio_status_ok: \
break; \
case iio_status_error: \
return iio_status_error; \
case iio_status_fatal: \
iio_log( \
    "IIO: called from: file %s, line %d\n", \
    __FILE__, __LINE__); \
return iio_status_fatal; \
}

#define iio_fret(S) \
switch ((S)) { \
case iio_status_ok: \
break; \
case iio_status_error: \
break; \
case iio_status_fatal: \
iio_log( \
    "IIO: called from: file %s, line %d\n", \
    __FILE__, __LINE__); \
return iio_status_fatal; \
}

#endif
```



## F.3 Header File types.h

```

#ifndef _types_h_
#define _types_h_
/*
 * This file is part of IIO, the Industrial IO Library
 * CSIRO Division of Manufacturing Technology
 * $Id: types.h,v 1.3 2000/06/20 08:28:53 kir092 Exp $
 *
 * types.h -- known size integer types
 * Robin Kirkham, June 1996
 */

/*
 * I hate this file, because we are second-guessing the compiler. But I see
 * no real alternative at present: only vxWorks defines certain size integer
 * types. Hence, this nonsense. We assume gcc. The #predicate() stuff is
 * apparently not ANSI, but the compiler only complains if we use -pedantic
 * as well (but it still works). You can test this file by compiling and
 * running debris/test.c
 */

#ifndef __GNUC__
#error This should be compiled by GCC
#endif

#if defined(m68000) || defined(m68010) || defined(m68020) || \
    defined(m68030) || defined(m68040) || defined(m68060) || \
    defined(i386) || defined(i486) || defined(i586) || \
    defined(ppc) || defined(powerpc) || \
    defined(sparc) || defined(sparclite)

    /* gcc does these all like this */
    typedef char iio_int8_t;
    typedef short int iio_int16_t;
    typedef int iio_int32_t;
    typedef long long int iio_int64_t;

    typedef unsigned char iio_uint8_t;
    typedef unsigned short int iio_uint16_t;
    typedef unsigned int iio_uint32_t;
    typedef unsigned long long int iio_uint64_t;

#else
#error Integer sizes not defined
#endif

    /* is this right? */
    #define widthof(typ) (sizeof(typ) * 8)

#endif

```



# Index

- CSIRO/MST
  - ATC-10, 92, 127
- adam, 133
  - 4017, 61, 90
  - 4018, 90
  - 4520, 61
- ADAM, 13, 27, 54, 61, 62, 81, 88–91
- ADC, 2, 3, 7, 8, 10, 14–16, 19, 20, 26,
  - 37, 85, 93, 94, 111, 112, 120,
  - 129, 130, 132
- address
  - base, 36
  - endianism, 55
  - probing, 44, 56, 80
  - resolution, 7
- address space
  - CPU modules, 60
  - channels, 39, 40
  - mapping, 14, 38, 38, 40, 58, 71, 78,
    - 79
  - module driver, 57
  - operation, 14, 39, 58, 59
  - port, 40, 41, 60
  - resolution, 14, 38, 38, 41, 57, 72
- alias, 11
  - global, 11
  - local, 11
  - module, 11
- ANSI, 67
- argument decoding, 35
  - boolean, 36
  - functions, 37
  - mandatory, 36
  - types, 36
- ASCII, 17, 61, 66
- ASIC, 52
- big-endian, 55
- BIOS, 104, 110
- bus error, 56
- bvm
  - IP-ADC, 20, 93, 112
- C, iii, 3, 5, 21, 55, 63, 65, 67, 77, 83–85
- C++, 85
- channel, 5, 6
  - adding new, 62
  - alias, 73
  - bitwise, 72, 76
  - close, 7
  - descriptor, 10
  - generic type, 7
  - information, 73
  - logging, 75
  - name, 7, 7, 19, 74, 85
    - alias, 11, 18, 19
    - global generic, 7
    - global specific, 8
    - local generic, 8
    - local specific, 8
  - node, 72, 74
  - open, 7, 73
  - operation, 6, 7, 12
    - code, 14
    - read, 12
    - write, 12
  - operations
    - adding new, 63
  - properties, 14, 15, 15, 43
    - limit, 19
    - limits, 15, 16, 43
    - linear scaling, 15, 15, 19, 43
    - logging, 15, 16, 19
    - units, 15, 15, 19, 43
  - range, 10, 10
  - registering, 42
  - sequence number, 7, 45
  - specific type, 7
  - type, 85
    - adding new, 62
- chip driver
  - register structure, 53
  - state structure, 53
- chip drivers, 51
  - interface, 52
  - source code, 53
- CMST, 123
- CMTE, 133
- command
  - chnode, 72
  - map, 71
  - minfo, 69
  - module, 70
- Concurrent Version System, *see* CVS
- configuration file, 5, 17, 35, 69, 83
  - alias directive, 18
  - channel directive, 18
  - module directive, 17
  - example, 19
  - parser, 70
  - syntax, 17
  - writing, 11, 20
- conventions, 65
  - coding, 66
  - data structures, 66
    - de-allocation, 66
  - object-oriented, 65
  - portability, 67
  - registration-callback, 65
- CPU, 5, 19, 33, 38, 39, 58, 60, 123
- cross compiling, 125
- CSDB, 4
- CSIRO, 4, 99, 112
- CVS, 32, 123, 125
- D-25, 110
- DAC, 2, 10, 14, 15, 19, 20, 43, 46, 96,
  - 101, 112, 115
- data structures, 67
- DC, 108

- DI, 94
- Diamond
  - DMM-32-AT, 94, 95, 127
- directory
  - iio/bin, 124
  - iio/doc, 124
  - iio/doc/manual, 124
  - iio/doc/manual/module, 48
  - iio/doc/proposal, 124
  - iio/h, 124
  - iio/lib, 124
  - iio/src, 48, 49, 66, 124, 125
  - iio/src/chip, 53, 124
  - iio/src/module, 48, 49, 54, 124
  - iio/src/object, 124
  - iio/src/os, 77
- distribution tree, 123, 124
- DMT, 4
- DS1620, 52
- ECP, 109, 110
- ECP+EPP, 110
- EISA, 104
- endianism, 55, 56
- enumerative
  - iio\_arg\_bool, 36
  - iio\_bool\_false, 36
  - iio\_bool\_true, 36
  - IIO\_CHTYPE, 62
  - iio\_chtype\_zog, 62
  - iio\_iflag\_log, 21
  - iio\_iflag\_none, 21
  - iio\_multi\_no, 33
  - iio\_multi\_yes, 32
  - iio\_oflag\_log, 22
  - iio\_oflag\_none, 22
  - IIO\_OP, 63
  - iio\_ptype\_read, 44
  - iio\_ptype\_write, 44
  - iio\_space\_id, 57
  - iio\_status\_error, 24
  - iio\_status\_fatal, 24
  - iio\_status\_ok, 24
- EPP, 109, 110
- error, 63
  - macros, 63
  - originating, 64
  - return string, 64
- file
  - ./iio.conf, 17, 78
  - /dev/mem, 40
  - /dev/ttya, 91
  - /dev/vme16, 40
  - /etc/iio.conf, 17
  - /vw/iio.conf, 17
  - iio.conf, 117
  - iio.h, viii, 21, 49, 63, 135
  - iio/bin/i486-linux/iio, 125
  - iio/bin/iio, 125
  - iio/bin/platform, 125
  - iio/lib/libiio-i486-linux.a, 125
  - iio/src/Makefile, 48, 54, 124, 125
  - iio/src/standard.c, 48
  - init.c, 66
  - internal.h, viii, 48, 49, 53, 54, 57, 62, 63, 77, 81, 137
  - lynxos.c, 77
  - Makefile, 21, 49, 77, 78, 123, 125
  - opinfo.c, 63
  - solaris.c, 77
  - standard.c, 49
  - stdarg.h, 77
  - types.h, viii, 127, 149
  - vxworks.c, 77
- fish tanks, 20
- function
  - exit(), 21
  - flock(), 78
  - fopen(), 22
  - free(), 66
  - getpagesize(), 79
  - iio\_alias(), 70, 73
  - iio\_alias\_find(), 74
  - iio\_alias\_insert(), 71, 73
  - iio\_arg(), 35–37, 40, 41, 70
  - iio\_arg\_index(), 37
  - iio\_arg\_index\_list(), 37
  - iio\_arg\_list(), 37
  - iio\_arg\_remnants(), 71
  - iio\_channel(), 70, 73
  - iio\_chnode(), 42, 45, 46, 58, 72
  - iio\_chnode\_limits(), 43, 73
  - iio\_chnode\_linear(), 43, 73
  - iio\_chnode\_new(), 72, 73
  - iio\_close(), 24
  - iio\_config\_exec(), 70
  - iio\_data\_get(), 46, 76
  - iio\_data\_get\_addr(), 46, 59, 76
  - iio\_data\_get\_real(), 46, 76
  - iio\_data\_set(), 46, 76
  - iio\_data\_set\_addr(), 46, 59, 60, 76
  - iio\_data\_set\_real(), 46, 76
  - iio\_done(), 24, 25, 78
  - iio\_done\_iio(), 78
  - iio\_emessage\_get(), 21, 24
  - iio\_file\_close(), 81
  - iio\_file\_getc(), 81
  - iio\_file\_open(), 81
  - iio\_init(), 21, 25, 32, 49, 66, 67, 69, 70, 73
  - iio\_init\_iio(), 67, 69, 70, 77, 78
  - iio\_ipinfo\_ident(), 44, 56, 57
  - iio\_ipinfo\_read(), 57
  - iio\_log(), 63
  - iio\_map(), 39–41, 56–58, 71, 72, 78, 118
  - iio\_map\_new(), 71
  - iio\_map\_ptov(), 71, 72
  - iio\_map\_type(), 71
  - iio\_mem\_alloc(), 79
  - iio\_mem\_free(), 79
  - iio\_mininfo(), 32, 33, 69
  - iio\_mininfo\_call(), 69
  - iio\_module(), 70, 71
  - iio\_module\_cmp(), 70
  - iio\_module\_create(), 70

- `iio_module_reg()`, 34, 70
  - `iio_module_state()`, 34, 35, 70
  - `iio_mutex_alloc()`, 80
  - `iio_mutex_drop()`, 80
  - `iio_mutex_free()`, 80
  - `iio_mutex_grab()`, 80
  - `iio_namex()`, 74
  - `iio_namex_chan()`, 74
  - `iio_open()`, 22, 24, 32, 37, 67, 73–75
  - `iio_operate()`, 23, 32, 67, 74–76, 84, 120
  - `iio_operate_addr()`, 23, 46, 58, 61, 71, 75
  - `iio_operate_bitfield()`, 75, 76
  - `iio_operate_call()`, 75, 76
  - `iio_operate_in()`, 23, 75
  - `iio_operate_inaddr()`, 23, 75
  - `iio_operate_inreal()`, 23, 75
  - `iio_operate_real()`, 23, 43, 75, 76, 120
  - `iio_osdone()`, 78
  - `iio_osinit()`, 67, 70, 77
  - `iio_port_alloc()`, 71, 79
  - `iio_port_get16()`, 79
  - `iio_port_get32()`, 79
  - `iio_port_get8()`, 79
  - `iio_port_set16()`, 79
  - `iio_port_set32()`, 79
  - `iio_port_set8()`, 79
  - `iio_probe()`, 44, 56, 80
  - `iio_resolve()`, 39–42, 55–58, 71, 72
  - `iio_resolve_list()`, 42
  - `iio_return_error()`, 64
  - `iio_return_fatal()`, 64
  - `iio_shmap_alloc()`, 71, 72, 78
  - `iio_shmap_done()`, 78
  - `iio_shmap_free()`, 78
  - `iio_shmem_alloc()`, 78–80
  - `iio_shmem_done()`, 78, 79
  - `iio_shmem_free()`, 79
  - `iio_shmutex_alloc()`, 80
  - `iio_shmutex_drop()`, 80
  - `iio_shmutex_free()`, 80
  - `iio_shmutex_grab()`, 80
  - `iio_sll_insert()`, 66, 70, 73
  - `iio_state_init()`, 69
  - `iio_tfile()`, 69, 70
  - `iio_tty_line()`, 81
  - `iio_tty_raw()`, 81
  - `iio_tty_recv()`, 81
  - `iio_tty_send()`, 81
  - `iio_xxyzyy1234()`, 49
  - `ioctl()`, 12, 81
  - `malloc()`, 66, 79, 80
  - `printf()`, 63
  - `read()`, 12
  - `smem_create()`, 78, 79
  - `smem_get()`, 79
  - `strcmp()`, 66
  - `vxMemProbe()`, 80
  - `write()`, 12
- GNU, 67, 117, 123, 125
- GreenSpring, 3, 44, 56
- ATC-30, 92
  - ATC-40, 92, 127
  - IP-16DAC, 133
  - IP-ADC, 93
  - IP-ADIO, 130
  - IP-DAC, 5, 19, 44, 96
  - IP-DIGITAL 24, 97, 127, 133
  - IP-DIGITAL 48, 98, 133
  - IP-DUAL PI/T, 98
  - IP-OPTO, 132
  - IP-PRECISION-ADC, 112
  - IP-PRECISION ADC, 133
  - IP-QUADRATURE, 99
  - IP-SERIAL, 54, 100, 133
  - IP-SERVO, 101, 102
  - IP-TIMER, 133
  - IP-WATCHDOG, 52, 103, 127, 133
  - VIPC-610, 19, 113
  - VIPC-616, 113
- ID-PROM, 40, 44, 54, 56, 57, 93, 96, 112
- identification function, 31, 32, 69
- IEEE, 109
- IIO, 26, 27
- IndustryPack, *see* IP
- initialisation, 5
- initialisation function, 31, 43, 59, 70
- probing, 44
- initialising function, 62
- installation function, 31, 33, 58, 62, 70
- interrupts, 60, 83, 84
- IP, 3, 5, 7, 38–41, 44, 54–57, 61, 92, 93, 96–101, 103, 105, 112, 113, 127, 134
- ISA, 7, 27, 39, 40, 44, 47, 56, 60, 92, 94, 104, 108, 127
- library size, 85
- little-endian, 55
- local alias, 11
- LVDT, 132
- LynxOS, 1, 17, 21, 24, 78, 79, 123, 133
- shared mutex problem, 80
- macro
- `__FILE__`, 63
  - `__LINE__`, 63
  - `iio_eret()`, 24, 33, 36, 63
  - `iio_error()`, 33, 44, 64
  - `iio_fatal()`, 33, 64
  - `iio_fret()`, 24, 63
  - `IIO_SIZE_MASK`, 60
- memory
- dynamic, 66
  - mapping, 71
  - protected, 65, 79
  - shared, 65, 79
- MMU, 38, 39, 58, 60, 71, 72
- model ident, 5, 17, 32, 70
- module, 5, 5, 17, 31
- alias, 11
  - driver, 6, 31
  - ADAM modules, 61
  - CPU modules, 60

- address space, 57
- directory, 49
- documentation, 49
- generic code, 54
- IndustryPack, 56
- proxy, 54
- source code, 48
- testing, 49
- ident, 7, 70
- ident code, 6
- initialisation, 45
- installation, 17
- list, 70
- mutex, 48, 70, 75
- parameters, 6, 6, 35, 70, 71
- self-identifying, 44, 57, 62
- sequence number, 6
- Motorola
  - MC68040, 105
  - MC68230, 98
  - MVME-160, 105
  - MVME-1600, 106
  - MVME-1603, 106
  - MVME-1604, 106
  - MVME-162, 5, 105
  - MVME-162LX, 105
  - MVME-167, 19, 105
- MPU, 105, 106
- MS-DOS, 109, 129
- mutex, 80
- NPN, 26
- object-oriented, 65, 84
- operating system, 77
  - files, 81
  - serial ports, 81
- operation, 12
  - code, 14, 46
    - generic, 14
  - function, 12, 32, 45, 59, 62, 75, 84
  - node, 74
  - read, 12
  - write, 12
- operational phases, 67
- PC, 1, 5, 39, 83, 92, 104, 108–110, 127, 134
- PC-104, 60
- PC/104, 94, 127
- PC/AT, 104
- PCI, 60, 104, 106
- PI/T, 98
- PID, 28, 101
- PIRAT, 4, 133
- POSIX, 77, 78, 80
- PowerPC, 106
- program
  - /bin/echo, 100
  - /bin/true, 100
  - configure, 123, 125
  - gcc, 67
  - gmake, 125
  - io, 49, 117, 120, 121, 125
  - ls, 121
  - make, 48, 49, 125
  - platform, 123–125
  - uname, 123
- PROM, 40
- RAM, 104
- RCS, 32, 69, 119
- register
  - endianism problem, 55
  - write-only, 48
- register structure, 32, 34, 70
  - chip driver, 53
- registration-callback, 65
- ROM, 104
- RPC, 133
- RS-232C, 61
- RS-422, 99
- RS-485, 61, 133
- RTD, 88, 89
- RTEMS, 1, 17, 65, 66
- SCC, 100
- SE, 94
- servo controller, 14, 84
- SI, 16, 43
- SPP, 109, 110
- state block, 69
- state structure, 32, 45, 48, 70
  - allocating, 34
  - chip driver, 53
- Tews DatenTechnik
  - TIP-850-11, 112
- trajectory generator, 14
- TTL, 26
- TTL/CMOS, 99
- type
  - char \* type, 36
  - double type, 36, 46
  - float type, 36
  - IIO type, 22, 36, 73, 75
  - IIO\_ALIAS type, 73
  - IIO\_ATYPE type, 73
  - IIO\_BOOL type, 36
  - IIO\_CHINFO type, 73
  - IIO\_CHNODE type, 42, 43, 72, 74
  - IIO\_FILE type, 36, 37, 69
  - io\_int16\_t type, 36
  - io\_int32\_t type, 36
  - io\_int64\_t type, 36
  - io\_int8\_t type, 36
  - IIO\_IPINFO type, 57
  - IIO\_MAGIC type, 66
  - IIO\_MAP type, 71
  - IIO\_MINFO type, 69
  - IIO\_MODULE type, 70, 74
  - IIO\_MREG type, 33, 48
  - IIO\_MSTATE type, 34, 35, 48
  - IIO\_Mutex type, 80
  - IIO\_NAMEX type, 74
  - IIO\_NFORM type, 72
  - IIO\_OPEN type, 73–75
  - IIO\_OPNODE type, 45, 46, 53, 73–76
  - IIO\_SENTINEL type, 78
  - IIO\_SHMUTEX type, 80

- II0\_SIZE type, 58
- II0\_STATUS type, 24
- II0\_UDATA type, 76
- ii0\_uint16\_t type, 36, 55, 127
- ii0\_uint32\_t type, 36, 55
- ii0\_uint64\_t type, 36
- ii0\_uint8\_t type, 36, 55, 127
- int type, 23
- SEM\_ID type, 80
- void type, 84
- void \* type, 36
- type safety, 84
- units, 16
  - SI, 16
  - channel property, 15
- UNIX, 1, 5, 6, 17, 21, 24, 35, 38, 40, 45,  
60, 64, 65, 67, 77, 78, 80, 83,  
85, 100, 117, 121, 123, 130
- user data, 46, 76, 84
- variable
  - argv[], 69, 70
  - CHIPNAMES, 54
  - errno, 64
  - ii0\_arg\_blank, 71
  - ii0\_ctype\_string[], 62
  - ii0\_first, 78
  - ii0\_opinfo[], 63
  - ii0\_pagesize, 79
  - ii0\_standard, 21, 25, 49, 69
  - ii0\_state, 69, 70
  - ii0\_state->alias, 73
  - ii0\_state->chnode, 69, 72
  - ii0\_state->map, 69, 71
  - ii0\_state->minfo, 69, 70
  - ii0\_state->module, 69, 70
  - ii0\_state->mutex, 69
  - ii0\_state->omutex, 69, 73, 75
  - ii0\_state->open, 69, 73
  - ii0\_timestamp, 78
  - MODULENAMES, 49
  - myDriverList, 25
  - PATH, 124
  - stderr, 63
- VMEbus, 1–3, 5, 7, 19, 27, 38–41, 56, 57,  
59, 60, 79, 83, 103, 105, 106,  
113–115, 131
- vmic, 3
  - VMIVME-2532A, 114, 127
  - VMIVME-2534, 19, 114, 127
  - VMIVME-2534A, 5
  - VMIVME-4100, 115
- vxWorks, 1, 3, 17, 21, 41, 44, 60, 65, 66,  
78–81, 83, 85, 125, 133, 134
- XYZZY-1234, 35, 45