CCII Systems (Pty) Ltd

Communications • Computer Intelligence • Integration

User Manual

for the

3U CPCI and VPX 64-Channel Isolated Digital Input / Output Board

VxWorks Software Driver

C²I² Systems Document No.	CCII/DIO/6-MAN/007
Document Issue	1.0
Issue Date	2017-01-24
Print Date	2017-01-24
File Name	W:\DIO\TECH\MAN\CDIMAN07.wpd
Distribution List No.	

Document prepared by C²I² Systems, Cape Town

Signature Sheet

Name	Signature	Date
Completed by L. KORTE	Project Engineer Board Level Products C²I² Systems	2017-01-24
Accepted by L DELUNA	Project Manager Board Level Products C²I² Systems	2017-01-24
Accepted by H. METCALF	Quality Assurance C²I² Systems	2017-01-24

Amendment History

Issue	Description	Date	ECP No.
1.0	Initial Release	2017-01-24	-

Contents

1. **Scope** 1

1.1 Identification 1
1.2 System Overview 1
1.3 Document Overview 1

2. **Applicable and Reference Documents** 2

2.1 Reference Documents 2

3. **DIO Linux Software Driver Distribution** 3

4. **Installation Procedure** 4

4.1 To Build the DIO VxWorks Software Driver into the VxWorks Kernel 4
4.2 To Load the DIO VxWorks Software Driver Separately 4

5. **VxWorks Software Driver** 5

5.1 Overview 5
5.2 DIO Message Control Block 5
5.3 Create Device 6
5.4 Get Device Information 6
5.5 Read Registers 6
5.6 Set Output Channel Status 8
5.7 Interrupts 9
5.8 Destroy Device 11
5.9 Description of Error codes 11

6. **DIO VxWorks Software Driver Interface** 12

6.1 DIO VxWorks Software Driver System Calls 12
6.2 Message Control Block Data Structure 21

7. **Input / Output Pin to Bit Mapping** 23

8. **Getting Started** 24

9. **Contact Details** 25

9.1 Contact Person 25
9.2 Physical Address 25
9.3 Postal Address 25
9.4 Voice and Electronic Contacts 25
9.5 Product Support 25

Abbreviations and Acronyms

API	Application Program Interface
CPCI	Compact Peripheral Component Interconnect
DIO	Digital Input / Output
FPGA	Field-Programable Gate Array
ID	Identification
I/O	Input / Output
MCB	Message Control Block
ms	millisecond
PCI	Peripheral Component Interconnect
RMS	Root Mean Square
V	Volts

List of Tables

Table 1 : DIO Device Information Data Structure 21

Table 2 : Message Control Block Data Structure 22

Table 3 : Message Control Block Data Arrays to Channel Mapping 23

List of Figures

Figure 1 : Software Flow Diagram for the DIO VxWorks Software Driver 5

1. Scope

1.1 Identification

This document is the User Manual for the VxWorks Software Driver for the 64 Channel Digital Input/Output (I/O) Board. It describes how to create and run code to control the Digital Input / Output Board.

1.2 System Overview

The 64-Channel Digital Input/Output Board provides 32 opto-isolated digital output channels, each with internal output status feedback, plus 32 opto-isolated digital input channels on a single 3U CompactPCI (CPCI) Board. A Field-Programmable Gate Array (FPGA) is used to provide access to the digital data over the PCI bus.

I/O channel to System isolation is 2 500 V RMS.

The DIO Board may be configured to achieve different numbers of input and output channels. Please enquire about different build options.

1.3 Document Overview

This document gives an overview of the DIO VxWorks Software Driver Installation Procedure and its Application Program Interface (API).

2. **Applicable and Reference Documents**

2.1 Reference Documents

2.1.1 CCII/DIO/6-MAN/004, *Hardware Design Manual for the 3U CPCI and VPX64-Channel Isolated Digital Input / Output Board*, Rev 1.1 dated 2015-12-03.

3. DIO Linux Software Driver Distribution

The DIO Linux Software Driver software distribution consists of (at least) the following files :

ccDIOLib<arch>V<version><long>.a	Host-architecture specific, driver object file : *cc* - CCII Systems (Pty) Ltd *DIOLib* - DIO Board VxWorks Software Driver <arch> - Host for which the binary is built : • X86 • PPC <version> - Software version is a 3 digit integer : • 1st digit : version number • 2nd digit : revision number • 3rd digit : beta number <*long*> - VxWorks Software Driver compiled with -mlongcall flag (only for a PPC host) e.g. “ccDIOLibPPCV100.a” for version 1.0.0 of the DIO software driver, built for a PPC processor.
dioReadme.txt	General information and installation notes.
dioSample.c	Sample C code for accessing the DIO VxWorks Software Driver.
dioSample.txt	Test procedure for verifying the DIO VxWorks Software Driver and Firmware.

4. Installation Procedure

This paragraph describes the installation procedure for the DIO VxWorks Software Driver. (The examples given are for a PowerPC host.)

4.1 To Build the DIO VxWorks Software Driver into the VxWorks Kernel

- Assume the BSP directory is given as : BSP_DIR = /tornado/target/config/dy4181.
- Copy ccDIOLibPPCV<version>.a to your $(BSP_DIR)/lib directory as ccDIOLib.a.
- In the Builds section of the Project Workspace, change the Kernel properties to include the ccDIOLib.a library file in the Macros LIBs option.
- Rebuild all VxWorks images.

4.2 To Load the DIO VxWorks Software Driver Separately

Note : This step is not required if the software driver was built into the BSP.

If the software driver is not built into the BSP, a user can load it separately :

- Copy ccDIOLibPPCV<version>.a to your present working directory as ccDIOLib.a.
- From the VxWorks shell type :

```
ld < ccDIOLib.a
```

5. VxWorks Software Driver

5.1 Overview

The following flow chart shows the main functions of the DIO VxWorks Software Driver :

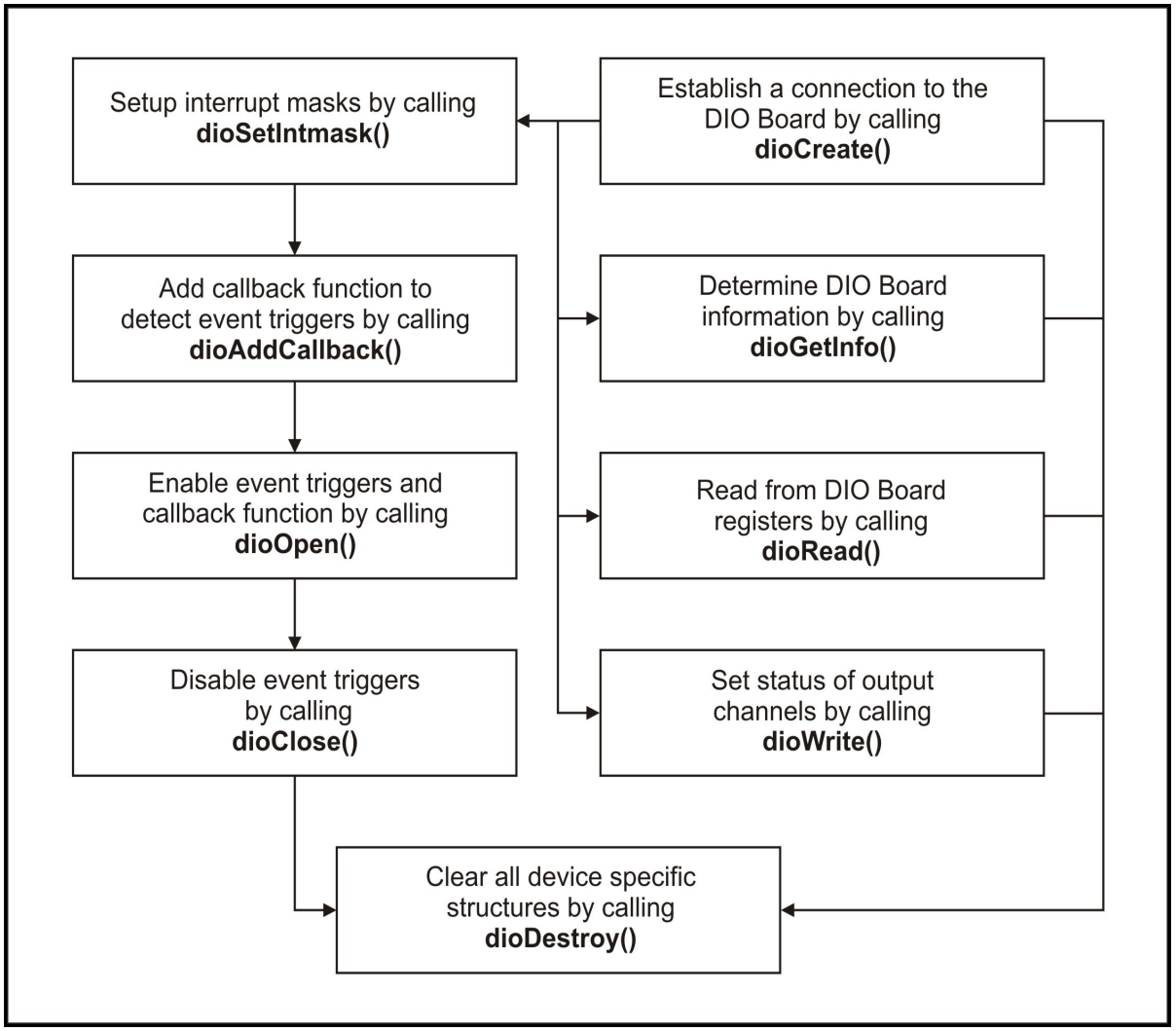

Figure 1 : Software Flow Diagram for the DIO VxWorks Software Driver

5.2 DIO Message Control Block

The *dioRead*, *dioWrite*, *dioSetIntmask* and user defined callback functions all use a Message Control Block (MCB) that acts as a carrier of information between the software driver and the user. The DIO MCB is described in more detail in Paragraph 6.2.

The MCB uses a member called *messageType* that determines what type of data is stored in the *data* array member during a *dioRead* and *dioSetIntmask* operation. Valid options for the *messageType* member are :

DIO_IN	-	I/O Status Register
DIO_INTMASK_WDT	-	Watchdog Timeout Register
DIO_INTMASK_RISE	-	Interrupt Rise Register
DIO_INTMASK_FALL	-	Interrupt Fall Register

Each bit of the *data* array member corresponds to a channel on the DIO Board. A detailed description of this bit to channel mapping is described in Paragraph 7. For input channels a bitwise 1 corresponds to a logic low input and a bitwise 0 corresponds to a logic high input. For output channels a bitwise 0 activates the output, meaning that it is sinking current and a bitwise 1 deactivates the output, meaning that the output pin is disconnected from ground.

Note that each bit of the *data_rise* and *data_fall* array members also correspond to a channel of the DIO Board but are only valid during a user defined callback routine. The *data_wdt* array member stores the watchdog timeout period in element 0 and element 1 determines if a watchdog timeout has occured but is only valid during a user defined callback routine.

5.3 Create Device

The DIO VxWorks Software Driver supports multiple DIO boards on a single host. To establish a connection and construct all the device specific structures, a user must create each of the devices separately, using the device ID to identify it.

The device ID starts at 0 and increments by 1 for each of the devices. Device 0 refers to the device in the lowest slot. The DIO VxWorks Software Driver can not be used until the user has created the device.

The below example shows how to create device 0 :

```
/* Create one DIO device */
dioCreate_device(1, 0);
```

The device ID is used in all calls to the DIO VxWorks Software Driver to identify the correct device.

5.4 Get Device Information

The user may use the *dioGetInfo()* function to find the device information. This will return the device information for the DIO Board with the specified device ID. The information will include the device type (0-2), hardware ID (0-15) and the switch ID (0-15).

The below example shows how the device information can be found and displayed for device 0 :

```
int devid = 0;
dioDeviceInfo device_info;
status = dioGetInfo(devid, &device_info);
if(status != DIO_OK)
{
    printf("Could not get device info.\n");
    return 1;
}

printf("Device info: type = %i, hardware ID = %x, switch ID = %x\n",
       device_info.device_type,
       device_info.device_hardware_ID,
       device_info.device_switch_ID);
```

The *dioDeviceInfo* structure is described in Paragraph 6.2.1.

5.5 Read Registers

The internal registers of the DIO Board may be read by using the *dioRead()* function. The following registers may be read :

I/O Status Register
Watchdog Timeout Register
Interrupt Rise Register
Interrupt Fall Register

The *dioRead()* function uses a MCB that acts as the carrier of information between the user space and the driver space. The full MCB structure is described in Paragraph 6.2.

```
dioMessageControlBlock MCB;
```

The *messageType* attribute of the MCB determines the register that the *dioRead()* function will return data from. The register data will be returned in the *data* array of the MCB. Each bit of the *data* array represents each channel of the DIO Board as described in Paragraph 7.

5.5.1 Read I/O Status

The current status of the input and output channels may be read from the DIO Board by setting the *messageType* attribute of the MCB to DIO_IN and using the *dioRead()* function.

MCB.messageType must be set to :

DIO_IN

The following example shows how the I/O Channel Status Register is read from device 0 :

```
int devid = 0;
for (i=0; i<6; i++) MCB.data[i] = 0x0; /*clear MCB data*/
MCB.messageType = DIO_IN;
status = dioRead(2, devid, &MCB);
if(status < 0)
{
   printf("Could not read from device: %i\n", devid);
}
printf("I/O Channel Status = ");
for(i = 2; i >=0; i--) printf("%08x ", MCB.data[i]);
printf("\n");
```

Note : The status for all the I/O channels (whether set as an input or output) are stored in the DIO_IN register. A channel set as an input will have its current status returned, while a channel set as an output will have the current user set value for the output channel returned.

5.5.2 Read Watchdog Timeout

The current value of the Watchdog Timeout Register may be read from the DIO Board by setting the *messageType* attribute of the MCB to DIO_WDT and using the *dioRead()* function. The DIO Board will generate a repeating interrupt after every time period where the length of the period is equal to the value set to the Watchdog Timeout Register in milliseconds.

MCB.messageType must be set to :

DIO_WDT

The following example shows how the Watchdog Timeout Register is read from device 0 :

```
int devid = 0;
for (i=0; i<6; i++) MCB.data[i] = 0x0; /*clear MCB data*/
MCB.messageType = DIO_WDT;
status = dioRead(2, devid, &MCB);
if(status < 0)
{
   printf("Could not read from device: %i\n", devid);
}
printf("Watchdog Timeout Register = %08x ", MCB.data[i]);
```

Note : The Watchdog Timeout Register is returned in element 0 of the *MCB.data* array.

5.5.3 Read Rise Interrupt Mask

The current value of the Interrupt Rise Register may be read from the DIO Board by setting the *messageType* attribute of the MCB to DIO_INTMASK_RISE and using the *dioRead()* function. The Interrupt Rise Register determines which input channels will trigger an interrupt on a rising edge.

MCB.messageType must be set to :

DIO_INTMASK_RISE

The following example shows how the Interrupt Rise Register is read from device 0 :

```
int devid = 0;
for (i=0; i<6; i++) MCB.data[i] = 0x0; /*clear MCB data*/
MCB.messageType = DIO_INTMASK_RISE;
status = dioRead(2, devid, &MCB);
if(status < 0)
{
   printf("Could not read from device: %i\n", devid);
}
printf("Interrupt Rise Register = %08x ", MCB.data[i]);
```

5.5.4 Read Fall Interrupt Mask

The current value of the Interrupt Fall Register may be read from the DIO Board by setting the *messageType* attribute of the MCB to DIO_INTMASK_FALL and using the *dioRead()* function. The Interrupt Fall Register determines which input channels will trigger an interrupt on a falling edge.

MCB.messageType must be set to :

DIO_INTMASK_FALL

The following example shows how the Interrupt Fall Register is read from device 0 :

```
int devid = 0;
for (i=0; i<6; i++) MCB.data[i] = 0x0; /*clear MCB data*/
MCB.messageType = DIO_INTMASK_FALL;
status = dioRead(2, devid, &MCB);
if(status < 0)
{
   printf("Could not read from device: %i\n", devid);
}
printf("Interrupt Fall Register = %08x ", MCB.data[i]);
```

5.6 Set Output Channel Status

The *dioWrite()* function may be used to set the status of the output channels on the DIO Board.

The *MCB.data* array must be set before calling the *dioWrite()* function and each bit represents each channel of the DIO Board as described in Paragraph 7. A bitwise 0 activates the output, meaning that it is sinking current. A bitwise 1 deactivates the output, meaning that the output pin is disconnected from ground. Writing to a channel configured as an input will have no effect on that channel. No specific message type is required.

The following example shows how to activate all the output channels on device 0 :

```
int devid = 0;
for (i=0; i<6; i++) MCB.data[i] = 0x0; /*inputs are unaffected*/
status = dioWrite(3, devid, &MCB, sizeof(dioMessageControlBlock));
if(status < 0)
{
   printf("Could not write to device: %i\n", devid);
}
```

5.7 Interrupts

The DIO VxWorks Software Driver is able to notify the user when an event occurs by calling a user defined callback function. The callback function will be called when one or more of the following events occur :

Watchdog Timeout	-	The Watchdog Timeout period has elapsed.
Input Rising Edge Trigger	-	A rising edge was detected on one or more of the inputs.
Input Falling Edge Trigger	-	A falling edge was detected on one or more of the inputs.

The user must first set the various interrupt masks for each input channel of the DIO Board. These include the rising edge and falling edge interrupt masks and the watchdog timeout period. Following this, the user must add a user defined callback function which will handle the data during an interrupt. Finally the *dioOpen()* function must be called to enable the message queue which will allow the user defined callback function to be called when an interrupt occurs.

5.7.1 Set Interrupt Masks

The user may set the interrupt masks for each channel by calling the *dioSetIntmask()* function. The *dioSetIntmask()* function requires a pointer to a DIO MCB which will store the values being set to the various interrupt mask registers. The *messageType* member of the DIO MCB determines which interrupt mask register will be written to. Valid inputs for the *messageType* member are :

DIO_INTMASK_WDT	-	Watchdog Timeout Register
DIO_INTMASK_RISE	-	Interrupt Rise Register
DIO_INTMASK_FALL	-	Interrupt Fall Register

The following example shows how the rise interrupt mask is enabled for each input and read-back channel :

```
int devid = 0;
MCB.messageType = DIO_INTMASK_RISE;
for (i=0; i<6; i++) MCB.data[i] = 0x0;
MCB.data[0] = 0xff00ff00;
MCB.data[1] = 0xff00ff00;
MCB.data[2] = 0xffffffff;
dioSetIntmask(devid, &MCB);
if(status != 0)
{
   printf("Could not set rise flags\n");
   return 1;
}
else
{
   printf("Successfully set Interrupt Rise Flags”);
}
```

Note : The Watchdog Timeout Register is set to the value written to the first element of the *MCB.data* array (*MCB.data[0]*). The value written is the watchdog timeout period in milliseconds (ms).

5.7.2 Add User Defined Callback Function

The user may add a user defined callback function by calling the *dioAddCallback()* function. Only one callback function may be set at a time. The callback function will be called when an interrupt occurs and will be given a pointer to a DIO MCB which will hold all the information of the event that occured.

The data returned in the MCB is :

The current status of the I/O pins	- MCB.data
The status of the Watchdog Timer	- MCB.data_wdt
The rising edge triggers that occured	- MCB.data_rise
The falling edge triggers that occured	- MCB.data_fall

Note : The value stored in *MCB.data_wdt[1]* determines if a watchdog timeout has occured. A value of 1 means that a watchdog timeout has occured and a value of 0 means that a watchdog timeout has not occured.

The MCB is described in more detail in Paragraph 6.2.

Callback function prototype :

```
void dioCallback (int num_args, int dev_id, int user_id,
dioMessageControlBlock* MCB, int size)
```

- *num_args* shall always be set to 4
- *dev_id* will be set to the device number of the device that triggered an interrupt
- *MCB* will be a pointer to a DIO MCB that stores the triggered data
- *size* will be set to the size of the DIO MCB

The below example shows how to create a user defined callback function that displays the triggered data on an interrupt :

```
void myCallback(int num_args, int dev_id, int user_id, dioMessageControlBlock*
MCB, int size)
{
  static int i;
  printf("\nDIO Device = %i", dev_id);
  printf("\nDIO_IN = ");
  for(i = 2; i >=0; i--) printf("%08X ", MCB->data[i]);
  printf("\nDIO_rise = ");
  for(i = 2; i >=0; i--) printf("%08X ", MCB->data_rise[i]);
  printf("\nDIO_fall = ");
  for(i = 2; i >=0; i--) printf("%08X ", MCB->data_fall[i]);
  if(MCB->data_wdt[1]==1) printf("\nWatchdog Timeout Occured");
  printf("\n");
}
```

The below example shows how to add the user defined callback function for device 0 :

```
int dev_id = 0;
if((status = dioAddCallback(3, dev_id, (dioCallback)myCallback, 0)) != DIO_OK)
{
  printf("\ndioAddCallback failed, %i\n", status);
}
```

5.7.3 Enable Message Queue

Once the interrupt masks and user defined callback function has been setup, the user may call the *dioOpen()* function to enable the message queue. This will enable the interrupt functionality and the user defined callback function will be called whenever an interrupt from the DIO Board occurs.

The following example shows how to enable the message queue for device 0 :

```
int dev_id = 0;
dioOpen(3, dev_id, 0, 11);
```

5.7.4 Disable Message Queue

The user may disable the message queue (and interrupt functionality) by calling the *dioClose()* function.

The following example shows how to disable the message queue for device 0 :

```
int dev_id = 0;
dioClose(2, dev_id, 0);
```

5.8 Destroy Device

When the device is no longer required it should be destroyed to free system resources.

The following example shows how to destroy device 0 :

```
int dev_id = 0;
dioDestroy(1, dev_id);
```

5.9 Description of Error codes

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_PCI_INIT_FAIL	-	PCI initialisation failed.
DIO_MEM_ALLOC_FAILED	-	If DIO device structure could not be created in memory.
DIO_DEVICE_NOT_FOUND	-	If DIO device <dev_id> was not found on the PCI bus.
DIO_ERROR	-	Error has occurred.
DIO_DEVICE_FAULTY	-	Device is faulty.

6. DIO VxWorks Software Driver Interface

The DIO VxWorks Software Driver source files contains the following header files (in '*dio/src/h*'), which should always be included in user applications :

- dioDriver.h - Function definitions.
- ccDefs.h - Variable definitions.

6.1 DIO VxWorks Software Driver System Calls

6.1.1 Create Device

Function : **dioCreate**

Purpose : Create and initialise the DIO device specific structures.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 1.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_PCI_INIT_FAIL	-	PCI initialisation failed.
DIO_MEM_ALLOC_FAILED	-	If DIO device structure could not be created in memory.
DIO_DEVICE_NOT_FOUND	-	If DIO device <dev_id> was not found on the PCI bus.

```
dioStatus dioCreate(int num_args, int dev_id);
```

Note : This function has to be called (once per device) before any other function call to the specified device will be valid.

6.1.2 Destroy Device

Function : **dioDestroy**

Purpose : Destroy the DIO device specific structures.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 1.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_PCI_INIT_FAIL	-	PCI initialisation failed.
DIO_ERROR	-	Device has already been destroyed.

```
dioStatus dioDestroy(int num_args, int dev_id);
```

Note : After this function is called, no other function call to the specified device will be valid, except for dioCreate_device().

6.1.3 Get Device Information

Function : **dioGetInfo**

Purpose : Return the DIO Board type, hardware ID and switch ID

Arguments :

<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<device_info>	-	Pointer to *dioDeviceInfo* structure that will hold the device information.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_DEVICE_FAULTY	-	Device is faulty.

```
dioStatus dioGetInfo(dioDeviceId dev_id, dioDeviceInfo *device_info);
```

6.1.4 Read Registers

Function : **dioRead**

Purpose : Read the DIO Board internal registers.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 2.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<MCB>	-	Pointer to *dioMessageControlBlock* structure that will hold the data returned.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_DEVICE_NOT_FOUND	-	Device not found.

```
dioStatus dioRead(int num_args, dioDeviceId dev_id,
                            dioMessageControlBlock * MCB);
```

Note : The messageType variable of the DIO MCB determines which register will be accessed. The following options are valid :

DIO_IN	-	I/O Status Register
DIO_INTMASK_RISE	-	Interrupt Rise Register
DIO_INTMASK_FALL	-	Interrupt Fall Register
DIO_INTMASK_WDT	-	Watchdog Timeout Register

6.1.5 Set Output Channel Status

Function : **dioWrite**

Purpose : Set the status of each output channel.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 3.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<MCB>	-	Pointer to *dioMessageControlBlock* structure that will hold the data being set.
<size>	-	Size in Bytes of *dioMessageControlBlock* structure.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_DEVICE_NOT_FOUND	-	Device not found.

```
dioStatus dioWrite(int num_args, dioDeviceId dev_id,
                           dioMessageControlBlock * MCB, int size);
```

6.1.6 Set interrupt Masks

Function : **dioSetIntmask**

Purpose : Set the interrupt masks and watchdog timeout period.

Arguments :

<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<MCB>	-	Pointer to dioMessageControlBlock structure that will hold the data being set.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_DEVICE_FAULTY	-	Device is faulty.
DIO_DEVICE_NOT_FOUND	-	Device not found.

```
dioStatus dioSetIntmask(dioDeviceId dev_id, dioMessageControlBlock *MCB)
```

Note : The messageType variable of the DIO MCB determines which register will be accessed. The following options are valid :

DIO_INTMASK_RISE	-	Interrupt Rise Register
DIO_INTMASK_FALL	-	Interrupt Fall Register
DIO_INTMASK_WDT	-	Watchdog Timeout Register

6.1.7 Add User Defined Callback Function

Function : **dioAddCallback**

Purpose : Add a user defined callback function that is called when an event triggers.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 3.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<callback>	-	User function.
<user_id>	-	User identifier. This identifier will be passed to the callback function when it is called.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.

```
dioStatus dioAddCallback(int num_args, dioDeviceId dev_id,
                         dioCallback callback, int user_id)
```

6.1.8 Enable Message Queue

Function : **dioOpen**

Purpose : Enable the message queue so that the user defined callback function is called on each interrupt.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 3.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<port_id>	-	Reserved for future use. Set this argument to 0.
<rx_priority>	-	Message queue priority. Should be set to 11.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_MEM_ALLOC_FAILED	-	Memory allocation failed.
DIO_ERROR	-	Semaphore ID is invalid.

```
dioStatus dioOpen(int num_args, dioDeviceId dev_id, int port_id,
                              int rx_priority)
```

6.1.9 Disable Message Queue

Function : **dioClose**

Purpose : Disable the message queue blocking all event triggers.

Arguments :

<num_args>	-	The number of arguments to follow. Must be set to 2.
<dev_id>	-	Device ID on the PCI bus. The DIO device in the lowest PCI slot : <dev_id> = 0, next DIO device : <dev_id> = 1, etc.
<port_id>	-	Reserved for future use. Set this argument to 0.

Returns :

DIO_OK	-	On success.
DIO_INVALID_PARAM	-	Invalid <dev_id> supplied or the function has been called with the incorrect number of variables.
DIO_ERROR	-	Semaphore ID is invalid.

```
dioStatus dioClose(int num_args, dioDeviceId dev_id, int port_id)
```

6.2 Message Control Block Data Structure

The following describes the structure of the MCB for the DIO driver (as defined in *dioDriverIfc.h*) :

```
struct dioMCB_struct
{
    dioDeviceInfo deviceInfo;
    dioMessageType messageType;
    ccUINT32 data[6];
    ccUINT32 data_rise[6];
    ccUINT32 data_fall[6];
    ccUINT32 data_wdt[6];
};
typedef struct dioMCB_struct dioMessageControlBlock;
```

6.2.1 Device Information Member

```
struct dioDeviceInfo_struct
{
    ccUINT8 device_type;
    ccUINT8 device_hardware_ID;
    ccUINT8 device_switch_ID;
};
typedef struct dioDeviceInfo_struct dioDeviceInfo;
```

Name	Options	Description
device_type	0 ... 2	The device types are: 0 = CCII/DIO/6UCPCI/256C/FP 2 = CCII/DIO/3UCPCI/64C/FP CCII/DIO/3UVPX/64C/FP CCII/DIO/3UCPCI/64C/BP
device_hardware_ID	0 ... 7	The hardware ID set by resistors on the DIO Board. Returns 0 in 3U DIO Boards.
device_switch_ID	0 ... 7	The switch ID set by the 4 way switch (U5).

Table 1 : DIO Device Information Data Structure

6.2.2 DIO Message Type

The DIO Message Type describes what type of data is currently held in the *data* array.

```
dioMessageType messageType;
```

Valid options for the message type include :

- DIO_IN
- DIO_INTMASK_FALL
- DIO_INTMASK_RISE
- DIO_WDT

6.2.3 Data Arrays

The following arrays hold the information carried by the MCB between the user and the driver. The mapping of I/O pins to bits in the data arrays is described in Paragraph 7.

```
ccUINT32 data[6];
ccUINT32 data_rise[6];
ccUINT32 data_fall[6];
ccUINT32 data_wdt[6];
```

Name	Type	Description
data [6]	array of 6 32 bit unsigned longs	Holds the data corresponding to the current DIO message type.
data_rise [6]	array of 6 32 bit unsigned longs	Shows which input channels have triggered a rising edge event. Only valid in a user defined callback routine.
data_fall [6]	array of 6 32 bit unsigned longs	Shows which input channels have triggered a falling edge event. Only valid in a user defined callback routine.
data_wdt [6]	array of 6 32 bit unsigned longs	*data_wdt[0]* holds the value set for the watchdog timer period. data_wdt[1] returns a 1 during a callback routine if a watchdog timeout occured, otherwise this is set to 0.

Table 2 : Message Control Block Data Structure

7. Input / Output Pin to Bit Mapping

The channel to bit mapping described below applies to the following part numbers :

- CCII/DIO/3UCPCI/64C/FP/COM
- CCII/DIO/3UCPCI/64C/FP/IND
- CCII/DIO/3UCPCI/64C/FP/RGD
- CCII/DIO/3UCPCI/64C/BP/COM
- CCII/DIO/3UCPCI/64C/BP/IND
- CCII/DIO/3UCPCI/64C/BP/RGD
- CCII/DIO/3UCPCI/64C/BP/CC
- CCII/DIO/3UVPX/64C/FP/COM
- CCII/DIO/3UVPX/64C/FP/IND
- CCII/DIO/3UVPX/64C/FP/RGD

The “MCB Array Element” refers to the following members of the MCB :

- MCB.data [6]
- MCB.data_rise [6]
- MCB.data_fall [6]

MCB Array Element	Bit	I/O Channel	I/O Type
[0]	Bit [0:7] Bit [8:15] Bit [16:23] Bit [24:31]	Channel [0:7] Channel [8:15] Channel [16:23] Channel [24:31]	Output Input Output Input
[1]	Bit [0:7] Bit [8:15] Bit [16:23] Bit [24:31]	Channel [32:39] Channel [40:47] Channel [48:55] Channel [56:63]	Output Input Output Input
[2]	Bit [0:7] Bit [8:15] Bit [16:23] Bit [24:31]	Read-back of Channel [0:7] Read-back of Channel [16:23] Read-back of Channel [32:39] Read-back of Channel [48:55]	Read-back Read-back Read-back Read-back
[3]	Unused	Unused	
[4]	Unused	Unused	
[5]	Unused	Unused	

Table 3 : Message Control Block Data Arrays to Channel Mapping

Please refer to the Hardware Reference Manual [2.1.1] for the connector pin to channel mapping.

8. Getting Started

After installing the DIO VxWorks Software Driver according to Paragraph 4, test it by following the test procedure given in dioSample.txt.

9. Contact Details

9.1 Contact Person

Direct all correspondence and / or support queries to the Project Manager at C²I² Systems.

9.2 Physical Address

CCII Systems (Pty) Ltd (C²I² Systems)
Real-Time House
Block T Greenford Office Estate
Punters Way
7708 Kenilworth
Cape Town
Republic of South Africa

9.3 Postal Address

C²I² Systems
P.O. Box 171
Rondebosch
7701
South Africa

9.4 Voice and Electronic Contacts

Tel	:	(+27) (0)21 683 5490
Fax	:	(+27) (0)21 683 5435
Email	:	info@ccii.co.za
Email	:	support@ccii.co.za
URL	:	http://www.ccii.co.za/

9.5 Product Support

Support on C²I² Systems products is available telephonically between Monday and Friday from 09:00 to 17:00 CAT. Central African Time (CAT = GMT + 2).