CCII Systems (Pty) Ltd
Communications • Computer Intelligence • Integration

User Manual

for the

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

Linux Software Driver

C²I² Systems Document No.	CCII/DIO/6-MAN/003
Document Issue	1.1
Issue Date	2015-12-03
Print Date	2015-12-03
File Name	W:\DIO\TECH\MAN\CDIMAN03.wpd
Distribution List No.	

© C²I² Systems The copyright of this document is the property of C²I² Systems. The document is issued for the sole purpose for which it is supplied, on the express terms that it may not be copied in whole or part, used by or disclosed to others except as authorised in writing by C²I² Systems.

Document prepared by C²I² Systems, Cape Town

Signature Sheet

Name	Signature	Date
Completed by L. KORTE	Project Engineer Board Level Products C²I² Systems	2015-12-04
Accepted by W. DE WAAL	Project Manager Board Level Products C²I² Systems	2015-12-04
Accepted by H. METCALF	Quality Assurance C²I² Systems	2015-12-04

Amendment History

Issue	Description	Date	ECP No.
1.0	First release	2015-06-09	-
1.1	Minor updates	2015-12-03	CCII/DIO/6-ECP/005

Contents

Abbreviations and Acronyms

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

List of Tables

1. Scope

1.1 Identification

This document is the user manual for the Linux 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 (I/O) 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 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 amounts of input and output channels. Please enquire with CCII Systems for different build options.

1.3 Document Overview

This document gives an overview of the DIO Linux 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 2015-12-03.

3. DIO Linux Software Driver Distribution

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

ccDIOLnxSrcV<version>.tar.gz	DIO Linux Software Driver source code. <version> - Software version is a 3 digit integer : • 1st digit : version number • 2nd digit : revision number • 3rd digit : beta number
dioReadme.txt	General information and installation notes.
dioTest.c	Sample C code for accessing the DIO Linux Software Driver.
dioTest.txt	Test procedure for verifying the DIO Linux Software Driver and Firmware.

4. Installation Procedure

This paragraph describes the installation procedure for the DIO Linux Software Driver.

4.1 Compiling the DIO Linux Software Driver Module

Unpack the DIO Linux Software Driver source file using *tar -xvzf ccDIOLnxSrcV<version>.tar.gz*. A directory named ‘dio' will be created. Change to *‘dio/builds/linuxX86’* to build the DIO Linux Software Driver module by typing the following commands :

```
make clean
make all
```

Note : The DIO Linux Software Driver is only supported on Linux kernel version 2.6 and above.

4.2 Loading the DIO Linux Software Driver

Within the *‘dio/builds/linuxX86‘* directory is a script file to help load the DIO Linux Software Driver.

The driver may be loaded by using the script (*dio*), which can be invoked from the system’s *rc.local* file or be called manually whenever the module is needed (e.g. *./dio start*)

The DIO Linux Software Driver supports up to four DIO Boards on one host system. The script files will detect all DIO Boards present and setup all devices. All DIO devices are created in */dev and* have the following naming convention :

dio_<X>	X	- [A - D] indicating which DIO Board this device belongs to.

The file */proc/devices* lists each DIO Board.

Note : In order to load the DIO Linux Software Driver, the user must have *root* privileges.

5. Linux Driver

5.1 Open DIO

Before any device may be accessed, it must be opened with the *open* system call. The *open* system call returns a file descriptor, which is used as a handle to the device for subsequent accesses.

The DIO Linux Software Driver allows only one instance of the device to be open at any time. Hence subsequent *open* calls will return an error. The user application should share the file descriptor between processes to access the device at any time. *dio_A* refers to the first device, *dio_B* refers to the second and so on.

The following example shows how to open the DIO device A :

```
int fd;
fd = open("/dev/dio_A", O_RDWR);
if(fd < 0)
{
   printf("Error opening device: %s - %s\n", "/dev/DIO_A", strerror(errno));
   return 1;
}
```

Can return error codes: -ENODEV; -ERESTARTSYS; -EBUSY.

5.2 Input / Output Control

After the DIO device has been opened, the *ioctl()* function allows the user to :

- get the device information
- set the I/O pin type
- set the interrupt masks
- set the watchdog timer delay
- set the debug options
- and read the registers

All functions are used on the message control block. This must be created and 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;
```

Can return error codes: -ERESTARTSYS; -ENOTTY; -EFAULT; -ENOMEM; -EBUSY; -EFAULT.

5.2.1 Get device information

The user must pass the DIO_IOCTL_DEVICE_INFO parameter into the *ioctl()* function to find the device information.

This will return the device information (*device_info*) for the device with “fd” as its 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 these can be read in and displayed :

```
dioDeviceInfo device_info;
status = ioctl(fd, DIO_IOCTL_DEVICE_INFO, &device_info);
if(status != 0)
{
   printf("Could not get device info.\n");
   close(fd);
   return 1;
}
else
```

```
{
   printf("Successfully performed ioctl on fd, return value = %i.\n", status);
}

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.

5.2.2 Set Input / Output Pin Type

The *ioctl()* function is used to set the type of the Input / Output pins. This must be done once during the setup of the DIO Board after the *open()* function has been called.

The user must pass the DIO_IOCTL_SET_OUT_DRV parameter into the *ioctl()* function to set the I/O pin type.

The MCB message type must be set to :

- DIO_OUT_DRV

The default setup of each DIO Board is specific to its part number, these are described in Paragraph 7. Once the correct values for the *MCB.data* array have been determined according to the DIO Board part number, these must be used with the *ioctl()* command.

The I/O pin type is set by writing to the *MCB.data* array. Each bit of the array corresponds to an I/O channel on the DIO Board, as described in Paragraph 7. A bitwise 1 sets the I/O pin to an output and a bitwise 0 sets the I/O pin to an input.

The following example shows how to set the I/O pin types for the DIO Board with part number CCII/DIO/3UCPCI/64C/FP/COM :

```
dioMessageControlBlock MCB;
MCB.messageType = DIO_OUT_DRV;
for (i=0; i<6; i++) MCB.data[i] = 0x0;
MCB.data[0] = 0x00ff00ff;
MCB.data[1] = 0x00ff00ff;
MCB.data[2] = 0x00000000;
status = ioctl(fd, DIO_IOCTL_SET_OUT_DRV, &MCB);
if(status != 0)
{
   printf("Could not set out_drv_en\n");
   close(fd);
   return 1;
}
else
{
   printf("Performed ioctl on fd, return value = %i.\n", status);
}
```

Note : The read-back channels are connected directly to the DIO Board output pins in hardware. All read-back channels must be configured as inputs. All read-back channels function similarly to input channels.

5.2.3 Set Interrupt mask

The *ioctl()* function is used to set the interrupt mask for the DIO input channels.

The user must pass the DIO_IOCTL_SET_INTMASK parameter into the *ioctl()* function to setup the interrupt masks.

Valid MCB message types are :

- DIO_INTMASK_FALL
- DIO_INTMASK_RISE

The interrupt mask is set by writing to the *MCB.data* array. A bitwise 1 enables the interrupt mask and a bitwise 0 disables the interrupt mask for the corresponding channel of the interrupt type. The interrupt mask has no affect on channels that have been configured as an output.

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

```
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;
status = ioctl(fd, DIO_IOCTL_SET_INTMASK, &MCB);
if(status != 0)
{
   printf("Could not set rise flags\n");
   close(fd);
   return 1;
}
else
{
   printf("Successfully performed ioctl on fd, return value = %i.\n", status);
}
```

5.2.4 Set Watchdog Timer

The *ioctl()* function is used to set the watchdog timer delay. The board will generate a repeating interrupt after every time period, where the length is equal the value set to the watchdog timer in milliseconds.

The user must pass the DIO_IOCTL_SET_WDT parameter into the *ioctl()* function to set the watchdog timer period.

Valid MCB message types are :

- DIO_WDT

The following example shows how to set the watchdog timer period to 1 000 ms :

```
long WDTinterval = 0x3e8; //0x3e8 = 1 000 ms
MCB.messageType = DIO_WDT;
for (i=0; i<6; i++) MCB.data[i] = WDTinterval;
status = ioctl(fd, DIO_IOCTL_SET_WDT, &MCB);
if(status != 0)
{
   printf("Could not set WDT\n");
   close(fd);
   return 1;
}
else
{
   printf("Successfully performed ioctl , return value = %i.\n", status);
}
```

5.2.5 Read registers

The ioctl() function allows the user to read any of the registers of the DIO device. This can be done by first setting the message type of the message control block. The message type determines which register will be read.

The user must pass the DIO_IOCTL_GET_REGISTER parameter into the *ioctl()* function to read the registers.

The message type may be set to one of the following :

- DIO_IN
- DIO_INTMASK_FALL
- DIO_INTMASK_RISE
- DIO_INT_STATUS_FALL
- DIO_INT_STATUS_RISE
- DIO_WDT

Once the *ioctl()* function has been called, the relevant register values according to the MCB message type will be saved in the *MCB.data* array.

The following example shows how the input channel status register is read :

```
for (i=0; i<6; i++) MCB.data[i] = 0x0; //clear MCB data
MCB.messageType = DIO_IN;
status = ioctl(fd, DIO_IOCTL_GET_REGISTER, &MCB);
if(status < 0)
{
   printf("Could not write to device: %s - %s\n", dev, strerror(errno));
}
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.

5.3 Write

To set the status of the output pins on the DIO Board, the *write()* function must be called.

The *MCB.data* array represents each channel of the DIO Board. This is 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 all the output channels being activated on the DIO Board :

```
for (i=0; i<6; i++) MCB.data[i] = 0x0; //inputs are unaffected
status = write(fd, (char *)&MCB, sizeof(dioMessageControlBlock));
if(status < 0)
{
   printf("Could not write to device: %s - %s\n", dev, strerror(errno));
}
```

If successful, the function will return the size of the MCB (104 Bytes).

Can return error codes: -ENOMEM; -ERESTARTSYS; -EINVAL; -EFAULT; -ENODEV; -EBUSY.

5.4 Read

The DIO *read()* function is setup as a blocking read. This means that the function will only return the MCB once an interrupt has occurred from the DIO Board. This enables the user to read from the device once an interrupt has occurred (such as in a callback function) or wait for the device to send an interrupt before the user reads the data.

A bitwise 0 read for an input channel means that the input is high (above 5 V) and a bitwise 1 read for an input channel means that the input is low (below 1,5 V).

The feedback channels are designed to follow what is set to the output channels. A bitwise 0 read from a feedback channel means that the corresponding output channel is enabled (sinking current) and a bitwise 1 read from a feedback channel means that the corresponding output channel is disabled (open).

The following example shows how the *read()* function is used :

```
status = read(fd, (char *)&MCB, sizeof(dioMessageControlBlock));
if(status < 0)
{
    printf("Could not read from device: %s - %s\n", dev, strerror(errno));
}
printf("DATA = ");
for(i = 2; i >=0; i--) printf("%08lX ", MCB.data[i]);
printf("RISE = ");
for(i = 2; i >=0; i--) printf("%08lX ", MCB.data_rise[i]);
printf("\nFALL = ");
for(i = 2; i >=0; i--) printf("%08lX ", MCB.data_fall[i]);
printf("\nWDT = ");
for(i = 2; i >=0; i--) printf("%08lX ", MCB.data_wdt[i]);
printf("\n");
```

The data returned in the MCB are :

- The current status of the I/O pins - MCB.data
- The rise flags that occurred since last *read()* operation - MCB.data_rise
- The fall flags that occurred since last *read()* operation - MCB.data_fall
- The current set value of the WDT - MCB.data_wdt

The MCB is described in more detail in Paragraph 6.2.

Another example can be seen in the *dioCallback()* function in Paragraph 5.5.

Can return error codes: -ENOMEM; -ERESTARTSYS; -EFAULT.

5.5 Setting up the interrupt

The DIO Linux Software Driver is able to notify the user if an interrupt has been sent from the DIO Board. This is accomplished with asynchronous notification and is enabled as follows :

- The user program needs to specify a process as the owner of the device. This is necessary so that the kernel knows which process to notify of an event.
- For the Call-back function to determine the file descriptor that caused this signal, the user program must set the SIGIO flag in the device by means of the F_SETSIG *fcntl* command.
- To actually enable asynchronous notification, the user program must set the FASYNC flag in the device by means of the F_SETFL *fcntl* command.

The above steps are implemented in software as follows :

```
#include <fcntl.h>

/* setup this process as the owner of the file descriptor */
fcntl(fd, F_SETOWN, getpid());

/* set the SIGIO signal */
fcntl(fd, F_SETSIG, SIGIO);

/* setup async notification */
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | FASYNC);

struct sigaction action;
```

```
    /* setup DIO callback */
    memset(&action, 0, sizeof(action));
    action.sa_sigaction = dioCallback;
    action.sa_flags = SA_SIGINFO | SA_NOMASK;

    sigaction(SIGIO, &action, NULL);
```

Note : For more information on the *sigaction* system call, consult its man page.

The Call-back function may be setup as follows :

```
    #include <signal.h>

    void dioTest_callback(int signo, siginfo_t *siginfo, void *what)
    {
        /* read data */
        status = read(fd, (char *)&MCB, sizeof(dioMessageControlBlock));
    }
```

The following example shows how the call-back function is used to display all the DIO Board data after an interrupt has occurred :

```
void dioTest_callback(int signo, siginfo_t *siginfo, void *what)
{
    int i, status;
    dioMessageControlBlock MCB;
    if(signo == SIGIO)
    {
        status = read(fd, (char *)&MCB, sizeof(dioMessageControlBlock));
        if(status < 0)
        {
              printf("Could   not   read   from   device:   %s   -   %s\n",   dev,
              strerror(errno));
        }
        else
        {
              printf("DATA = ");
              for(i = 2; i >=0; i--) printf("%08lX ", MCB.data[i]);
              printf("RISE = ");
              for(i = 2; i >=0; i--) printf("%08lX ", MCB.data_rise[i]);
              printf("\nFALL = ");
              for(i = 2; i >=0; i--) printf("%08lX ", MCB.data_fall[i]);
              printf("\nWDT = ");
              for(i = 2; i >=0; i--) printf("%08lX ", MCB.data_wdt[i]);
              printf("\n");  intflag = 1;
        }
    }
    fflush(stdout);
}
```

5.6 Close

The user must call the close function to remove the DIO device :

```
fd = open("/dev/dio_A", O_RDWR);
if(fd < 0)
{
    printf("Error opening device: %s - %s\n", "/dev/DIO_A", strerror(errno));
    return 1;
}
// close the DIO Board with device id "fd"
close(fd);
```

5.7 Description of Error codes

ENOMEM	- No memory available
ERESTARTSYS	- The task has been killed
EINVAL	- Invalid MCB size
EFAULT	- There was a fault reading the data
ENODEV	- No device can be found
EBUSY	- The device is currently being used
ENOTTY	- Inappropriate IOCTL

6. DIO Linux Software Driver Interface

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

- dioLnxDriver.h - IOCTL command definitions.
- ccDefs.h - Required for dioDriverIfc.h.
- dioDriverIfc.h - Definition of structures and types for the driver of the DIO Board.
- other header files - Other header files are only used to compile the software driver module and are not necessary for user applications.

6.1 DIO Linux Software Driver System Calls

6.1.1 Open System Call

Function : **open**

Purpose : Open the DIO device and return a file descriptor.

Arguments :

<pathname> - The path and name of device. Usually **"/dev/dio_X"**, where X = [A - D].

<flags> - Always **O_RDWR**. If no blocking is desired, specify **O_NONBLOCK** as well (bitwise-ORed with previous parameter). Do not specify **O_ASYNC** here, rather setup asynchronous notification as described in 5.5.

Returns :

fd - File descriptor.
-1 - Error occurred.

Errors :

-ENODEV - Incorrect device specified or device cannot be found.
-EBUSY - The device is already open or the adapter is busy.
-EINTR - The open system call was interrupted by a signal.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
```

6.1.2 Close System Call

Function : **close**

Purpose : Close the DIO device and release the file descriptor.

Arguments :

<fd> - The file descriptor to be closed.

Returns :

0	-	On success.
-1	-	Error occurred.

Errors :

-EBADF	-	Incorrect file descriptor specified.
-ENODEV	-	Incorrect device specified or device cannot be found.
-EBUSY	-	The adapter is busy.
-EINTR	-	The close system call was interrupted by a signal.

```
#include <unistd.h>

int close(int fd);
```

6.1.3 Read System Call

Function : **read**

Purpose : A blocking read from the device. Will return the current input and output pin status, rise flags, fall flags and the watchdog timer value. These are returned in the buffer (MCB).

Arguments :

<fd>	-	File descriptor to read from.
<buf>	-	Buffer to read bytes into.
<count>	-	Number of bytes to read from device.

Returns :

num_bytes	-	Number of bytes read. This will always be the size of the Message Control Block.
-1	-	Error occurred.

Errors :

-EBADF	-	Incorrect file descriptor specified.
-EFAULT	-	There was a problem copying data into the user specified buffer.
-EAGAIN	-	Non blocking has been specified and no data was immediately available for reading.
-EINTR	-	The close system call was interrupted by a signal.

```
#include <unistd.h>

ssize_t read(int fd, char *buf, size_t count);
```

6.1.4 Write System Call

Function : **write**

Purpose : Write the data stored the in the MCB (*MCB.data* array) to the output pins. Does not affect input pins.

Arguments :

<fd>	-	File descriptor to write to.
<buf>	-	Buffer containing the data to be written to device. Must always be a Message Control Block
<count>	-	Number of bytes to write to device. Must always be the size of the Message Control Block.

Returns :

num_bytes	-	Number of bytes written. Will always be size of the Message Control Block.
-1	-	Error occurred.

Errors :

-EBADF	-	Incorrect file descriptor specified.
-ENODEV	-	Incorrect device specified or device cannot be found.
-EINVAL	-	The maximum number of bytes to be written has been exceeded.
-EFAULT	-	There was a problem copying data from the user specified buffer.
-EINTR	-	The close system call was interrupted by a signal.

```
#include <unistd.h>

ssize_t write(int fd, char *buf, size_t count);
```

6.1.5 Ioctl System Call

Function : **ioctl**

Purpose : Configure the DIO device or obtain device specific information.

Arguments :

<fd>	-	File descriptor to configure.
<request>	-	Command to be performed. One of :

- DIO_IOCTL_DEVICE_INFO
- DIO_IOCTL_SET_WDT
- DIO_IOCTL_SET_INTMASK
- DIO_IOCTL_SET_OUT_DRV
- DIO_IOCTL_GET_REGISTER

<buf>	-	Buffer containing the data to be written to or read from the device. Must always be a Message Control Block

Returns :

0	-	On success.
-1	-	Error occurred.

Errors :

-EBADF	-	Incorrect file descriptor specified.
-ENODEV	-	Incorrect device specified or device cannot be found.
-EINVAL	-	Incorrect protocol specified for specific device or port.
-EFAULT	-	The command specific argument references an inaccessible memory area.
-ENOTTY	-	The specified request does not exist.
-ENOMEM	-	Internal temporary kernel memory allocation failed.
-EBUSY	-	DIO Adapter may be busy.

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, char *buf);
```

6.2 Message Control Block Data Structure

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

BIT Structures :

```
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_INT_STATUS_FALL
- DIO_INT_STATUS_RISE
- DIO_WDT

6.2.3 Data Arrays

The following arrays hold the information carried by the Message Control Block between the user and the driver. The mapping of I/O pins to bits in the data arrays are 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	Holds the data for the rise flags that have occurred since the last *read()* operation. Is only used during a *read()* operation.
data_fall [6]	array of 6 32 bit unsigned longs	Holds the data for the fall flags that have occurred since the last *read()* operation. Is only used during a *read()* operation.
data_wdt [6]	array of 6 32 bit unsigned longs	Holds the value set for the watchdog timer. Each of the 6 unsigned longs will hold this value. Is only used during a *read()* operation.

Table 2 : Message Control Block Data Structure

7. Input / Output Pin to Bit Mapping

The pin mapping described in Table 2 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/RUG

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

- 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 Linux Software Driver according to Paragraph 4, test it by following the test procedure given in dioTest.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).