Introduction

The BaseBoard4, with the FPGA programmed, supports up to eleven different peripherals. Using the protocol and peripherals registers described in this manual; a host computer, typically running Linux, controls the peripherals on the BaseBoard4 through the USB link connecting the two. Of the eleven possible peripherals, nine, Peripheral #2 through Peripheral #11, are defined by the user (customer) and two, Peripheral #0 and Peripheral #1, are included in every build of the FPGA firmware. The creation of the FPGA code begins with the user choosing up to nine additional peripherals to install and concludes with the choices converted, through Verilog, into an the FPGA image file (the Demand Peripherals DPCore). The system with the protocol and the chosen peripherals is physically implemented once the DPCore image is downloaded into the FPGA.

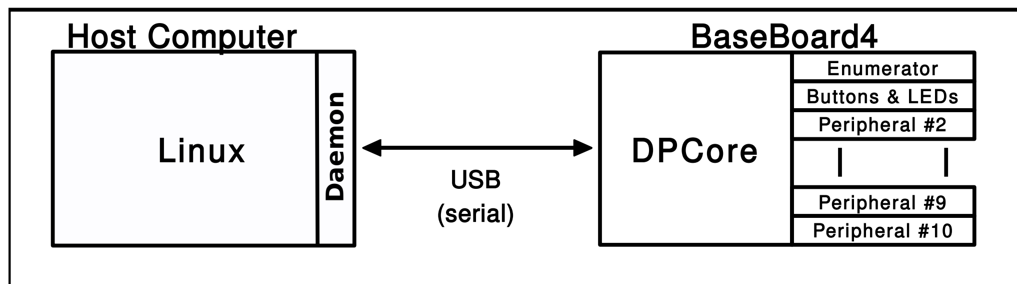


Figure 1. DPCore System Architecture

Overview

The BaseBoard4 loaded with DPCore is a major component of the system shown in Figure 1. The full system is composed of a daemon running on a Linux host that provides high level Application Programming Interfaces (APIs) to the controlling applications for each of the installed peripherals. DPCore, in the FPGA, provides critical timing and control at the FPGA pins of the BaseBoard4 connectors. The final component is the customer supplied hardware that comprises the cabling and electronics needed to implement the specific peripheral's function.

Consider, for example, a servo controller. A Linux application on the host defines the angle of the servo motor's horn and communicates this setting to the Demand Peripherals device driver daemon. The Demand Peripherals daemon converts the angle setting into a pulse width measured in increments of 50 nanoseconds. The daemon sends the pulse width (as a count) to the control register for the servo controller in DPCore. Hardware in the FPGA based servo controller uses the count to give an accurate pulse width to the FPGA output pin to which the servo is connected.

Each of the peripherals in a DPCore image is identified and referred to by a unique identifier called an address and each address has one or more control or status registers associated with it. For example, the address, Peripheral #0, refers to an internal logic construct, called the *enumerator*, that is a read-only memory block containing a list of the peripherals built into a specific DPCore image. When the Linux daemon starts, it reads the contents of Peripheral #0 and configures itself to match the peripherals it finds listed in the enumerator. Every build of DPCore has an enumerator and it is always at Peripheral #0. Another construct that is included in every DPCore is Peripheral #1 which manages the eight LEDs and three buttons installed on the BaseBoard4. This peripheral has two registers associated with it. A read of Register 0 in Peripheral #1 gives the state of the three buttons and a write to Register 1 sets the state of the eight LEDs.

© Copyright Demand Peripherals, Inc. All rights reserved.

Peripheral Pin Definitions

The physical FPGA pins used by a peripheral are determined by the address of the peripheral. With some constraints, the user is able to assign the peripheral address to meet his layout requirements. FPGA pins are referred to by the connector number on the BaseBoard4 and are associated to addresses as follows:

Peripheral	Component or Pins
0	Enumerator
1	BB4 buttons and LEDs
2	Connector SV1 pins 2, 4, 6, 8
3	Connector SV1 pins 10, 12, 14, 16
4	Connector SV2 pins 2, 4, 6, 8
5	Connector SV2 pins 10, 12, 14, 16
6	Connector SV3 pins 2, 4, 6, 8
7	Connector SV3 pins 10, 12, 14, 16
8	Connector SV4 pins 2, 4, 6, 8
9	Connector SV4 pins 10, 12, 14, 16
10	GPIO0, GPIO1, GPIO2

Note: Connector SVx pins 1, 5, 9 and 13 are ground.
Note: Connector SVx pins 3 and 11 are 3.3V power.
Note: Connector SVx pins 7 and 15 are 5.0V power.
Note: GPIOx center pin is 5V and outside one is ground.

Figure 2. BaseBoard4 Peripheral Slots.

The first constraint in assigning addresses is that some peripherals require twice the number of FPGA pins and so use all eight FPGA pins of a connector, not just the four associated with the address listed above. Since these peripherals span two peripheral addresses and therefore two sets of pins, they are referred to by the lower of the two addresses that they span and are never split between two connectors. This means that they will be referred to as Peripheral #2, #4, #6 or #8. A dual H-bridge controller, for example, is an 8 pin peripheral and might be at address #2 and use all the pins on connector SV1. The next peripheral would then be located at address #4 and would use the first four pins on connector SV2 if it were a 4 pin peripheral.

Another constraint applies to Peripheral #10. This peripheral has only three FPGA pins, 5.0V power and ground. The layout on the board was configured for servo motor connectors.

The DPCore Protocol

Features of the protocol include the following:

- The protocol is organized as SLIP encoded packets
- Each packet has a command byte with a command of Read, Write, or WriteRead
- A packet may read or write multiple values
- A packet may read or write the same or consecutive registers
- Registers may be either 8 or 16 bits wide
- Addresses are 16 bits (one byte for peripheral # and one for register address)
- A response packet indicates success or failure of a command packet
- Data can be sent automatically from DPCore to the host

Both commands to DPCore and responses to the commands are sent as SLIP encoded packets. Each packet is composed of a command byte, an address byte, a register byte, a transfer count byte , and, if a write command or a read response,the data word. All command responses echo back the command, the address, and the transfer count. The responses also have a byte after the transfer count to indicate how many words remain to be transferred if there is additional response data remaining to be sent.

© Copyright Demand Peripherals, Inc. All rights reserved

A command packet from the host to DPCore has the following format:

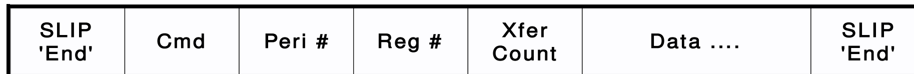


Figure 3. Command Packet Format

Serial Line Internet Protocol (SLIP) encapsulation is described by RFC 1055. SLIP packets end with an END character (0xC0). The DPCore implementation of SLIP uses Phil Karn's extension which starts packets with an END character as well. In a packet the END character is replaced by the two byte sequence 0xdb, 0xdc. The character 0xdb is defined as the ESC character. In a packet the ESC character is replaced by the two byte sequence 0xdb, 0xdd. It is a protocol violation to have an END character appear anywhere in a packet. It is also a violation to have an ESC character that is not followed by either 0xdc or 0xdd.

The command byte has the following bit definitions:

- BIT 7: Transparent data echoed back to the host
- BIT 6: Ignored in commands to DPCore. Cleared in read response packets if the packet is the result of a read command, and set if the packet has data that is being automatically sent from the peripheral to the host.
- Bit 5: Reserved for future use
- Bit 4: Reserved for future use
- Bit 3-2: 00 No operation
  01 Read data from the peripheral
  10 Write data to the peripheral
  11 Write-Read data to/from the peripheral. (Used for SPI peripherals)
- Bit 1: One to increment the register address for each word transferred. Zero to read or write words repeatedly to the same address.
- Bit 0: One to send a word size of 16 bits and Zero for a word size of 8 bits.

The second byte in the packet is the peripheral address. This address implicitly specifies the FPGA pins (or connector pins) associated with the peripheral. While generally consecutive, peripheral addresses do not need to be. For example, a peripheral that spans eight FPGA pins also spans two peripheral addresses.

The third byte specifies which register in the peripheral to read or write. If the command is for a multi-word transfer with incrementing register addresses, the third byte specifies the starting register number.

The fourth byte specifies how many words to transfer. A "word" may be either one or two bytes depending on Bit 0 in the command byte. Up to 255 words can be transferred by a single command.

Data is sent as a sequence of big-endian words. That is, the high byte of 16 bit data is sent first.

The packet format described here is used for all commands to the FPGA. Response packets from the FPGA add one more byte after the transfer count. This byte indicates how many words remain to be transferred. Usually this byte is zero, indicating that all words were transferred. However, when reading or writing to FIFO-based peripherals you might not know beforehand how many words are queued for sending. In this case it is best to send read requests with a request count of 255 and the peripheral can send words until it exhausts its queue. How many words are in the read response can be computed as 255 minus the value of the fifth byte. Writing data is similar. Send a write packet with as many words as you wish, but watch the remaining field in the response to see if the peripheral had to drop part of the data you sent. Use the remaining field to compute which data needs be resent in a later write packet.

Many systems require that the host poll peripherals for new data. This type of polling can place a heavy load on the host processor and can severely limit data rates to and from the peripherals. DPCore allows peripherals to respond to an external event by automatically building and sending a read response packet to the host. Bit 6 of the command byte is zero for packets sent automatically by DPCore. Peripherals can send a packet up to the host at most every 4 milliseconds. A typical aggregate data rate for a peripheral sending a packet every 4 milliseconds is on the order of 2500 bytes per second. Please contact Demand Peripherals if

© Copyright Demand Peripherals, Inc. All rights reserved

your application requires a DPCore build with an update rate faster than every 4 milliseconds.

A Demand Peripherals Protocol Testbed

The appendix to this manual contains two C programs that can help you explore the registers and peripherals in a DPCore system. The first program, `dpdump`, displays packets sent from DPCore to the host. It removes the SLIP encoding and displays each packet byte as a two digit hexadecimal number. The whole packet is displayed on one line and a newline character separates consecutive packets.

A typical sequence to compile and run `dpdump` might be:

```
gcc -o dpdump dpdump.c
/dpdump /dev/ttyUSB0 &
```

Install DPCore as you would any BaseBoard4 FPGA image, using the following sequence.

```
stty -opost < /dev/ttyUSB0
cat DPCore_XXXX.bin > /dev/ttyUSB0
```

All builds of DPCore use Peripheral #1 to manage the LEDs and buttons built onto the BaseBoard4. The buttons on the BaseBoard4 are edge sensitive and automatically send updates to the host on a button press or release. If you have loaded and are running `dpdump` a press of button 3 should give a packet dump of:

```
01: 06 01 00 01 04 00
```

As a reminder, 01: is the board number, 06 is the command byte, 01 is the peripheral address, 00 is the register address, 04 is the value assigned to button 3, and the final 00 indicates that all (one) of the requested registers are being returned in this packet. This packet is assembled and sent automatically for a button press (or release) event

The other program, `dpxmit`, SLIP encodes a packet for transmission to a BaseBoard4. Build and run `dpxmit` with the following commands:

```
gcc -o dpxmit dpxmit.c
./dpxmit >/dev/ttyUSB0
```

The input to `dpxmit` is a line of hexadecimal numbers separated by spaces. Each line is send as a complete SLIP encoded packet. The LEDs are accessed as a byte write to register 1 at peripheral #1. The packet to set the LEDs to 0x55 would be:

```
0a 01 01 01 55
```

The above packet generates the following response from the FPGA

```
01: 0a 01 01 01 00
```


© Copyright Demand Peripherals, Inc. All rights reserved

Peripheral Register Definitions

This section presents the low level register definitions for every peripheral offered by Demand Peripherals. An introduction to each peripheral is given before the register definitions, but the reader is assumed to have sufficient background in programming and electrical engineering to find the register definitions sufficient.

Enumerator

The enumerator is a read-only-memory that contains configuration information that is set at Verilog build time. The ROM contains a copyright notice, the email address of the person who accepted the license, the date of the build, and a list of the peripherals in the build. The ROM contains 19 null-terminated ASCII strings. The first 8 strings are for the licensing information and the last 11 strings are the list of peripherals in the build. The ROM has at most 2048 characters and is always Peripheral #0.

The enumerator has a single 8-bit register that is accessed as a FIFO. Typically you would read the ROM by issuing a series of 8-bit reads of 255 words from register 0. An internal counter keeps track of the reads so that subsequent reads pick up the next character to send. You may reset the internal counter by doing any 8-bit write to register 0.

The information in the enumerator is broken into strings of text with the following meaning assigned to each string:

String 0: Copyright
String 1: Licensee email address
String 2: Build date
String 3: (unused)
String 4: (unused)
String 5: (unused)
String 6: (unused)
String 7: (unused)
String 8: enumerator.1
String 9: bb4io.1
String 10: peripheral_slot_2_name
String 11: peripheral_slot_3_name
String 12: peripheral_slot_4_name
String 13: peripheral_slot_5_name
String 14: peripheral_slot_6_name
String 15: peripheral_slot_7_name
String 16: peripheral_slot_8_name
String 17: peripheral_slot_9_name
String 18 peripheral_slot_10_name

The string 'peripheral_slot_x_name' above is replaced by the name of the library for the peripheral the user has selected for that slot. Examples of typical library names include "dc2.1", "quadrature2.1", and "servo8.1". Note that the enumerator and BaseBoard4 I/O peripherals (Buttons and LEDs) are always assigned to slot zero and one respectively.

Each slot has four pins associated with it. Some peripherals use eight pins, not four, and we need a way to indicated that a slot is unused or is used by the previous eight-pin peripheral. A special peripheral named "null" fills this need. A null peripheral takes four pins but does not actually have any registers associated with it.

 © Copyright Demand Peripherals, Inc. All rights reserved

BaseBoard4 Buttons and LEDs

As indicated previously, the BaseBoard4 has three buttons and 8 LEDs installed on the board. Collectively these are the BaseBoard4 I/O peripherals and are always at address Peripheral #1.

An 8-bit read of Peripheral #1, Register 0 returns the current status of the buttons. The state of button 1 is in the LSB, button 2 is in bit 1, and button 3 is in bit 2. A set bit indicates that the switch is closed. The button inputs are edge triggered and a press or release of a button automatically sends a read packet response up to the host. You do not need to poll the buttons to get a press or release event.

An 8-bit write to Peripheral #1, Register 1 sets the LEDs. Bit 0 of the byte sent controls the bit 0 of the LEDs. A set bit turns on the LED. A read of Register 1 returns the current status of the LEDs.

The Buttons and LEDs peripheral has two 8-bit registers:

Register 0: Button status.
Register 1: LED control & status.

Quad Event/Frequency Counter

The quad counter has four independent counters that count the positive edges at the four FPGA input pins. The counts are visible in four unsigned 16-bit registers. Counts are automatically sent up to the host 250 times a second. The 4 millisecond sample window uses a crystal controlled timebase making the counts immediately useful for frequency estimation. The maximum input frequency per pin is 5 MHz.

Counter A is connected to the lowest numbered pin on the peripheral connector. Counter D is connected to the highest numbered pin.

The Quad Event/Frequency Counter has four 16-bit registers:

Register 0: Count from Counter A
Register 2: Count from Counter B
Register 4: Count from Counter C
Register 6: Count from Counter D

Dual DC Motor Controller

The dual DC motor controller peripheral provides direction, PWM speed control, and both brake and coast for two motors using four FPGA pins. The lowest numbered pin on the connector is the AIN1 input for motor A and the second pin is the AIN2 input. Pins 3 and 4 are the BIN1 and BIN2 inputs for the B motor. The modes of operation versus the IN pins is depicted in this table.

MODE	IN1	IN2	
(3) Brake	high	high	This is the power-on default
(2) Forward	low	(PWM)	
(1) Reverse	(PWM)	low	
(0) Coast	low	low	

The PWM control uses a base counter that increments from zero up to a terminal count called the "period" which is set by the low ten bits of register 0. The rate at which the counter increments is set by the clock source selected by the upper three bits of register 0.

Register 2 uses the high two bits for motor A mode and the low ten bits for the "on" count for the PWM output on A. Motor A PWM is set low at the start of the cycle and goes high when the count increments to the

© Copyright Demand Peripherals, Inc. All rights reserved

A on count. An A-on value of 0 turns the A motor on 100% and a value equal to the period turns motor A off completely.

The B output goes high at the start of the cycle. The B output turns off at the B-off count which is in the low ten bits of register 4. The high two bits of register 4 is the B motor mode. Full motor off is when B-off equals zero. Full motor on is B-off=period.

Register 6 is the watchdog control register. The idea of the watchdog is that if enabled (bit 15 == 1) the low four bits are decremented once every 100 millisecond. If the watchdog count reaches zero both PWM outputs are turned off. A four bit counter gives a minimum update rate of about once every 1.5 seconds. You reset the watchdog timer by writing a new timeout value to register 6. The power on default is to disable the watchdog.

The Dual DC Motor peripheral has five16-bit registers:

Register 0: PWM Control – Clock source (3 MSB) + PWM period (10 LSB).
Register 2: Motor A – Mode (2 MSB) + PWM on count (10 LSB).
Register 4: Motor B – Mode (2 MSB) + PWM off (10 LSB).
Register 6: Watch-dog Enable (Bit 15 ==1) + Watch-dog Timer (4 LSB).

The clock source is selected by the upper three bits of register 0 with the following meaning:

0: off
1: 20 MHz
2: 10 MHz
3: 5 MHz
4: 1 MHz
5: 500 KHz
6: 100 KHz
7: 50 KHz

H-Bridge Controller

The H-Bridge controller controls the four individual switches in an H-Bridge using four FPGA pins. Modes of operation include driving on the high side, driving on the low side, and driving on both high and low sides. Both high and low side dynamic braking is possible, as is a not-driven or coast mode. In ascending order on the pins, the four outputs are "Top Left" (TL), "Top Right" (TR), "Not Bottom Left" (BL_), and "Not Bottom Right" (BR_). Note well that the bottom half pins are inverted.

There are two safety features included in this controller. The default, power-up configuration is high side braking. The FPGA powers up with all pins tied to pull-up resistors. This turns on both high side transistors and OFF both low side transistors, putting the controller into the high side brake mode.

The second safety feature is inclusion of a watchdog timer. When enabled, a failure to write to the watchdog register within the timeout period turns off all PWM outputs.

The PWM for the motor controller uses a base counter that increments from zero up to the "period" count. The increment rate is selected by "clksel". The clksel[1:0] is the high two bits of register 0, and the period is the low ten bits. The four possible clock sources are:

Clksel 0: 20 MHz
Clksel 1: 1 MHz
Clksel 2: 100 KHz
Clksel 3: 10 KHz

Register 2 uses the low ten bits for the "on" count for the PWM output. Note that since the count is incrementing, the lower the value of register 2 the sooner the output goes high and the longer the PWM pulse. An on count of "period" is 0 percent duty cycle and an on count of 0 is 100 %.

© Copyright Demand Peripherals, Inc. All rights reserved

Register 4 has the motor direction in bit 7 and the mode in the low 3 bits. The modes are:

Mode 0: High side brake. All outputs high. (power on default)
Mode 1: Low side brake
Mode 2: Coast
Mode 3: PWM on bottom side transistors
Mode 4: PWM on high side transistors
Mode 5: PWM on both high and low side transistors
Mode 6: PWM switch the direction

Register 6 has the watchdog control register. If enabled (bit 8 == 1) the low 4 bits are decremented once every 100 milliseconds. If the watchdog count reaches zero all PWM outputs are turned off. A 4 bit counter gives a minimum update rate of about 1.5 seconds. A read of register 6 gives the time remaining until the timer expires. You reset the watchdog timer by writing a new timeout value to register 6.

Register 6 also has the anti-shoot-through timer in the high 4 bits. The time is measured in units of 50 nanoseconds and can vary between 0 nanoseconds and 15*50 = 750 nanoseconds. The anti-shoot-through timer forces all switches off for a short time when changing direction. Huge current flow, called "shoot through", can occur if both top and bottom switches are on at the same time. The anti-shoot-through timer switches all outputs off briefly, giving the transistor gate pins a chance to discharge and so fully turn off.

The H-Bridge Controller uses four 16-bit registers for control and status:

Register 0: Count decrement rate = clksel (2 MSB) + PWM period (10 LSB).
Register 2: PWM on count (10 LSB).
Register 4: Motor direction (Bit 7) + Motor mode (3 LSB).
Register 6: Anti-Shoot-Through (4 MSB) + Watch-dog Enable (Bit 8 ==1) + Watch-dog Timer (4 LSB).

The four possible clock sources are:

Clksel 0: 20 MHz
Clksel 1: 1 MHz
Clksel 2: 100 KHz
Clksel 3: 10 KHz

The modes are:

Mode 0: High side brake. All outputs high. (power on default)
Mode 1: Low side brake
Mode 2: Coast
Mode 3: PWM on bottom side transistors
Mode 4: PWM on high side transistors
Mode 5: PWM on both high and low side transistors
Mode 6: PWM switch the direction

NOTES:

- If dir==1 then V+ is applied to the left side of the motor. That is, output TL is closed, as is BR_.
- All switches are open while the anti-shoot-through timer is non-zero

© Copyright Demand Peripherals, Inc. All rights reserved

Dual H-Bridge Controller

The dual H-Bridge controller controls the four individual switches of two independent H-Bridges using eight FPGA pins. The advantage of the dual H-bridge controller over two instances of a single H-Bridge controller is that the dual controller schedules the PWM pulses such that only one motor is driven at a time. This extends battery life since it reduces the I-squared-R loss in the battery and its wires. Even when both motors are on with more than 50 percent duty cycle, the controller tries to limit how much overlap there is for both motors being driven simultaneously.

Modes of operation include driving on the high side, driving on the low side, and driving on both high and low sides. Both high and low side dynamic braking is possible, as it a not-driven or coast mode. In ascending order on the pins, the eight outputs are "Motor A Top Left" (ATL), "Motor A Top Right" (ATR), "Motor A Not Bottom Left" A(ABL_), "Motor A Not Bottom Right" (ABR_), "Motor B Top Left" (BTL), "Motor B Top Right" (BTR), "Motor B Not Bottom Left" (BBL_), and "Motor B Not Bottom Right" (BBR_). Note well that the bottom half pins are inverted for both motors.

There are two safety features included in this controller. The default, power-up configuration is high side braking. The FPGA powers up with all pins tied to pull-up resistors. This turns on both high side transistors and OFF both low side transistors, putting the controller into the high side brake mode.

The second safety feature is inclusion of a watchdog timer. When enabled, a failure to write to the watchdog register within the timeout period turns off all PWM outputs.

The PWM for the motor controller uses a base counter that increments from zero up to the "period" count. The increment rate is selected by "clksel". The clksel[1:0] is the high two bits of register 0, and the period is the low ten bits. The four possible clock sources are:

Clksel 0: 20 MHz
Clksel 1: 1 MHz
Clksel 2: 100 KHz
Clksel 3: 10 KHz

Register 2 uses the low ten bits for the "on" count for the first motor (Motor A) PWM output. Note that since the count is incrementing, the lower the value of register 2 the longer the PWM pulse. An on count of "period" is 0 percent duty cycle and an on count of 0 is 100 %.

Register 4 has the turn-off count for the second motor (Motor B). The B motor turn on at the start of the cycle and Register 4 specifies when to turn it off. An off count of 0 turns the motor off completely, and an B-off count of period turns it on completely.

Register 6 has the first motor (Motor A) direction in bit 7 and the mode in the low 3 bits. Bit 15 and bits 10-8 are the direction and mode for the second motor (Motor B). The modes are:

Mode 0: High side brake - All outputs high (power on default).
Mode 1: Low side brake
Mode 2: Coast
Mode 3: PWM on bottom side transistors
Mode 4: PWM on high side transistors
Mode 5: PWM on both high and low side transistors
Mode 6: PWM switch the direction

Register 8 has the watchdog control register. If enabled (bit 8 == 1) the low 4 bits are decremented once every 100 milliseconds. If the watchdog count reaches zero all PWM outputs are turned off. A 4 bit counter gives a minimum update rate of about 1.5 seconds. A read of register 10 gives the time remaining until the timer expires. You reset the watchdog timer by writing a new timeout value to register 10.

Register 8 also has the anti-shoot-through timer in the high 4 bits. The time is measured in units of 50 nanoseconds and can vary between 0 nanoseconds and 15*50 = 750 nanoseconds. The anti-shoot-through timer forces all switches off for a short time when changing direction. Huge current flow, called "shoot through",

© Copyright Demand Peripherals, Inc. All rights reserved

can occur if both top and bottom switches are on at the same time. The anti-shoot-through timer switches all outputs off briefly, giving the transistor gate pins a chance to discharge and so fully turn off.

The Dual H-Bridge Controller uses six 16-bit registers for control and status:

Register 0: Count decrement rate = clksel (2 MSB) + PWM period (10 LSB).
Register 2: PWM on count for Motor A (10 LSB).
Register 4: PWM off count for Motor B (10 LSB).
Register 6: Motor B Direction (Bit 15) + Mode (Bit 10:8) + Motor A Direction (Bit 7) + Mode (Bit 2:0)
Register 8: Anti-Shoot-Through (4 MSB) + Watch-dog Enable (Bit 8 ==1) + Watch-dog Timer (4 LSB).

The four possible clock sources are:

Clksel 0: 20 MHz
Clksel 1: 1 MHz
Clksel 2: 100 KHz
Clksel 3: 10 KHz

The modes are:

Mode 0: High side brake - All outputs high (power on default).
Mode 1: Low side brake
Mode 2: Coast
Mode 3: PWM on bottom side transistors
Mode 4: PWM on high side transistors
Mode 5: PWM on both high and low side transistors
Mode 6: PWM switch the direction

NOTES:

- If dir==1 then V+ is applied to the left side of the motors. That is, output ATL and BTL are closed, as is ABR_ and BBR_.
- All switches are open while the anti-shoot-through timer is non-zero

Dual Chain Maxbotix Ultrasonic Range Finder

The dual chain Maxbotix ultrasonic range sensor peripheral can read the distance from two chains of Maxbotix range sensors using four FPGA pins. The idea of a "chain" is that you give a start strobe to the first sensor on the chain and as it is reporting the distance it finds, it triggers the next sensor in the chain. The advantage of a chain is that one set of inputs can read up to 16 different sensors. Another advantage is that only one sensor on the chain transmits at a time which prevents interference between the sensors.

Both chains are synchronized to be sure that only one sensor on the peripheral is emitting at a time. This is accomplished by not strobing the B chain until all of the sensors in the A chain have had a chance to report in.

The sensors report a distance as pulse widths that increase in quantum steps of 147 microseconds per inch giving a range from 6 inches to 254 inches. Each sensor's measurement takes 50 milliseconds. The B chain strobe is sent after waiting 50 milliseconds for each of the sensors in the A chain. The process repeats with the A chain strobe being sent after the last B chain sensor measurement is made.

The reading of the pulse width is in units of 20 microseconds making a value of 147 into 160 and a value of 1890 into 1900. That is, the actual reading is rounded up to the next multiple of 20 microseconds. The value for a reading is sent immediately up to the host, so the host gets a separate SLIP packet for each sensor.

The output Register 0 contains the pulse time (in units of 20 microseconds) in an 11 bit number and the 4 bit ID of the sensor. The ID is just a zero-indexed count of the pulses received since the strobe. The MSB of this register identifies if it is the A sensor chain (MSB == 0) or the B sensor chain (MSB == 1) data. The

© Copyright Demand Peripherals, Inc. All rights reserved

configuration Register 2 has the count of sensors in the high and low 8 bits respectively for the A and B sensor chains.

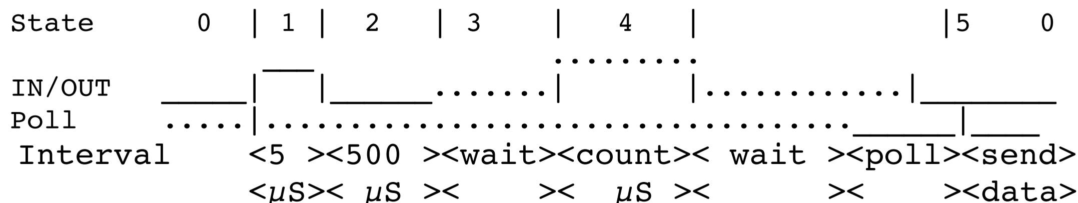
The Dual Chain Maxbotix Ultrasonic Range Finder uses two 16-bit registers:

Register 0:	Sensor ID and value	MSB=0 if A, 1 if B, 4 bit sensor ID, 11 bits of pulse width
Register 2:	Chain counts	Bits 8-11 is A chain length + Bits 0-3 is B chain length.

Quad Parallax Ultrasonic Range Finder

The quad PNG)))™ peripheral interfaces to as many as four Parallax PNG)))™ ultrasonic sensors using four FPGA pins. The sensors are numbered from 0 to 3 and a measurement is made from each one in turn. As each measurement is completed it is reported immediately to the host as a read response from the first two registers. The second register, the interface number, should be incrementing in each response and then wrap back to zero.

For each sensor the corresponding FPGA pin drives the PNG)))™ input high for 5 microseconds (State 1) and then holds the line low for 500 microseconds (State 2). It then switches to being an input and starts listening for a rising edge (State 3) coming back from the PNG)))™. When it finds the rising edge it counts the microseconds until the falling edge (State 4). With a complete sample, we wait for a poll and then send the sample up the host (State 5)



The above is repeated for each of the four input lines. If we're still in state 3 after 1024 microseconds we assume that no sensor is connected and go immediately to state 5 with a reading of zero.

The sensors may be individually turned on or off by a write to the low four bits of register #4. The LSB of register #4 corresponds to the lowest numbered pin on the BaseBoard4 connector and to the lowest numbered interface, number zero.

The Quad Parallax Ultrasonic Range Finder uses three 16-bit registers:

Register 0:	Echo time in microseconds with 15 bits of resolution
Register 2:	Interface number (1-4)
Register 4:	Enabled register (Bits 0-3 are a bit mask, 1==enabled)

Quad PWM Input

The quad PWM input peripheral can measure the high and low times for up to four inputs using four FPGA pins.

The registers store the values of the pins at the start of an interval and the duration, in clock counts, of the interval. At the end of a cycle the values are sent up to the host and a new cycle is started. A new cycles starts on the first input transition after sending up to the host. This state machine has three states: waiting for first transition, taking measurements, and waiting to send to host.

The transition out of "taking measurements" state can occur on either of two events: all inputs have made at

© Copyright Demand Peripherals, Inc. All rights reserved

least three transitions (so we get both high and low durations), or, there have been no transitions at all while the interval counter counted from 0 to 65535. We don't want a busy input to fill up the interval registers so an input is ignored after it has made three transitions.

The number of transitions recorded is in bits 4 to 7, and the clock source is in bits 0 to 3. Since the measurements start on the first transition, the first duration (register 0) is always zero. The duration registers record the clock count at the end of the interval and are zero indexed. That is a duration of 0x0001 represents a duration of two clock cycles.

The MSB of the value bits is the lowest numbered pin on the connector.

The Quad PWM Input uses twenty-five 16-bit registers:

Register 0: Interval 0 duration in clock counts (16 bits)
Register 2: Input pins at the start of the interval (4 bits)
Register 4: Interval 1 duration in clock counts (16 bits)
Register 6: Input pins at the start of the interval (4 bits)
Register 8: Interval 2 duration in clock counts (16 bits)
Register 10: Input pins at the start of the interval (4 bits)
Register 12: Interval 3 duration in clock counts (16 bits)
Register 14: Input pins at the start of the interval (4 bits)
Register 16: Interval 4 duration in clock counts (16 bits)
Register 18: Input pins at the start of the interval (4 bits)
Register 20: Interval 5 duration in clock counts (16 bits)
Register 22: Input pins at the start of the interval (4 bits)
Register 24: Interval 6 duration in clock counts (16 bits)
Register 26: Input pins at the start of the interval (4 bits)
Register 28: Interval 7 duration in clock counts (16 bits)
Register 30: Input pins at the start of the interval (4 bits)
Register 32: Interval 8 duration in clock counts (16 bits)
Register 34: Input pins at the start of the interval (4 bits)
Register 36: Interval 9 duration in clock counts (16 bits)
Register 38: Input pins at the start of the interval (4 bits)
Register 40: Interval 10 duration in clock counts (16 bits)
Register 42: Input pins at the start of the interval (4 bits)
Register 44: Interval 11 duration in clock counts (16 bits)
Register 46: Input pins at the start of the interval (4 bits)
Register 48: Number of transitions recorded (Bits 4 to 7) + Clock source (Bits 0 to 3)

The Clock Source, in the 4 LSB of Register 48, is specified by the following assignments

0: Off
1: 20 MHz
2: 10 MHz
3: 5 MHz
4: 1 MHz
5: 500 KHz
6: 100 KHz
7: 50 KHz
8: 10 KHz
9 5 KHz
10 1 KHz
11: 500 Hz
12: 100 Hz
13: 50 Hz
14: 10 Hz
15: 5 Hz

Note that there is no global sync pulse for the inputs. This means that the output can start on any edge. If

© Copyright Demand Peripherals, Inc. All rights reserved

all four inputs are active then you may see eight different output patterns for the same input pattern (four pulses times two edges per pulse). Parsing the output might be easies if you make several passes over the output getting the high and low times for just one pin.

Quad / Octal PWM Output

The PWM output peripherals can generate four or eight bits of precise PWM output using four or eight FPGA pins. You can specify the values at the output pins and the amount of time from one transition to the next. The duration counts are zero indexed. That is, a duration count of 0x0001 represents two clock cycles.

As an example, let's consider an application in which the intensities of four LEDs are controlled by a quad PWM output peripheral at slot #2. One LED is on at 100 percent, one at 75 percent, one at 50 percent and one at 25 percent. The PWM frequency is to be 1 KHz. The clock frequency is set to 1 MHz. All LEDs are turned on initially. One LED is turned off at a count of 250. Another is turned off at a count of 500, and a third LED is turned off at a count of 750. The `dpxmit` commands would be:

```
0b 02 40 01 00 44          //  Register 64 = 4 steps, 1 MHz
0b 02 00 02 00 f9 00 0f    //  Register 0,2 = 249 ; all bits on
0b 02 04 02 00 f9 00 0e    //  Register 4,6 = 249 ; toggle bit 1
0b 02 08 02 00 f9 00 0c    //  Register 8,10 = 249 ; toggle bit 2
0b 02 0c 02 00 f9 00 08    //  Register 12,14 = 249 ; toggle bit 3
```

The Quad PWM Output uses thirty-three 16-bit registers:

Register 0: Interval 0 duration in clock counts	(16 bits)
Register 2: Output pins at the start of interval 0	(4/8 bits)
Register 4: Interval 1 duration in clock counts	(16 bits)
Register 6: Output pins at the start of interval 1	(4 bits)
Register 8: Interval 2 duration in clock counts	(16 bits)
Register 10: Output pins at the start of interval 2	(4/8 bits)
Register 12: Interval 3 duration in clock counts	(16 bits)
Register 14: Output pins at the start of the interval	(4/8 bits)
Register 16: Interval 4 duration in clock counts	(16 bits)
Register 18: Output pins at the start of the interval	(4/8 bits)
Register 20: Interval 5 duration in clock counts	(16 bits)
Register 22: Output pins at the start of the interval	(4/8 bits)
Register 24: Interval 6 duration in clock counts	(16 bits)
Register 26: Output pins at the start of the interval	(4/8 bits)
Register 28: Interval 7 duration in clock counts	(16 bits)
Register 30: Output pins at the start of the interval	(4/8 bits)
Register 32: Interval 8 duration in clock counts	(16 bits)
Register 34: Output pins at the start of the interval	(4/8 bits)
Register 36: Interval 9 duration in clock counts	(16 bits)
Register 38: Output pins at the start of the interval	(4/8 bits)
Register 40: Interval 10 duration in clock counts	(16 bits)
Register 42: Output pins at the start of the interval	(4/8 bits)
Register 44: Interval 11 duration in clock counts	(16 bits)
Register 46: Output pins at the start of the interval	(4/8 bits)
Register 48: Interval 12 duration in clock counts	(16 bits)
Register 50: Output pins at the start of the interval	(4/8 bits)
Register 52: Interval 13 duration in clock counts	(16 bits)
Register 54: Output pins at the start of the interval	(4/8 bits)

© Copyright Demand Peripherals, Inc. All rights reserved

Register 56: Interval 14 duration in clock counts (16 bits)
Register 58: Output pins at the start of the interval (4/8 bits)
Register 60: Interval 15 duration in clock counts (16 bits)
Register 62: Output pins at the start of the interval (4/8 bits)
Register 64: Clock source in the low 4 bits and the number of intervals in use in bits 4 to 7

The Clock Source, in the 4 LSB of Register 64, is specified by the following assignments:

0: Off
1: 20 MHz
2: 10 MHz
3: 5 MHz
4: 1 MHz
5: 500 KHz
6: 100 KHz
7: 50 KHz
8: 10 KHz
9 5 KHz
10 1 KHz
11: 500 Hz
12: 100 Hz
13: 50 Hz
14: 10 Hz
15: 5 Hz

IR Receiver

The IR Receiver peripheral decodes consumer IR pulse streams from an NEC compatible IR remote control using four FPGA pins. The first pin is the input from the IR receiver to the FPGA. The second pin is an output that is high when an IR packet is being received. The second pin would usually be connected to an LED to show activity. The remaining two pins are used for general purpose I/O. The low two bits of register 17 are the value of the pins and bits 4 and 5 control the direction of the third and fourth connector pins respectively. A 1 in the data direction bits indicates an output. Reads and write to the low two bits of register 17 read and set the pins depending on the values in the data direction bits. The two general purpose pins default to inputs and are not event driven. That is, you must poll them to get their current states.

The receiver allows pulse streams to be up to 64 bits in length even though the most common format is only 32 bits in length. In the NEC protocol a packet starts with an AGC pulse 9.0 milliseconds long followed by a pause of 4.50 milliseconds. The next 32 pulses indicate either a one or a zero with a one being a 0.56 millisecond pulse followed by a 1.69 millisecond pause, and a zero being a 0.56 millisecond pulse followed by a 0.56 millisecond pause. The packet ends with a stop bit with a pulse 0.56 milliseconds long followed by a pause of at least 30 milliseconds.

The peripheral accepts up to 64 bits of IR packet data. Because of the way the shift registers work the data is stored in strips. That is the most recent bit is in the low bit register 0. The next most recent bit is in the low bit of register 1, and so on until the sixteenth most recent bit which is in the low bit of register 15. The seventeenth most recent bit is in bit 1 of register 0; the eighteenth most recent bit is in bit 1 of register 1, and so on. The sixty-fourth most recent bit is in bit 3 of register 15. The configuration register is just the high bit of register 16. It indicated the value of the input while an IR signal is being received. Since most IR receivers have an active low output the polarity bit defaults to zero. A write to register 16 affects only the MSB. The status register is the low 6 bits of register 16. These six bits indicate how many valid samples are stored in the IR data bit registers. Received IR packets are automatically sent up to the host as a read response to a read of 17 consecutive 8-bit registers starting with register number 0.

The program `irrec.c` converts the `dpdump` of an IR received packet into a string of ASCII ones and zeros.

 © Copyright Demand Peripherals, Inc. All rights reserved

Use this program as a guide to understanding how to read the IR packet from the peripheral registers.

The IR Receiver uses eighteen 8-bit registers:

Register 0: IR packet data (in low 4 bits)
Register 1: IR packet data (in low 4 bits)
Register 2: IR packet data (in low 4 bits)
Register 3: IR packet data (in low 4 bits)
Register 4: IR packet data (in low 4 bits)
Register 5: IR packet data (in low 4 bits)
Register 6: IR packet data (in low 4 bits)
Register 7: IR packet data (in low 4 bits)
Register 8: IR packet data (in low 4 bits)
Register 9: IR packet data (in low 4 bits)
Register 10: IR packet data (in low 4 bits)
Register 11: IR packet data (in low 4 bits)
Register 12: IR packet data (in low 4 bits)
Register 13: IR packet data (in low 4 bits)
Register 14: IR packet data (in low 4 bits)
Register 15: IR packet data (in low 4 bits)
Register 16: IR receiver status and configuration register
Register 17: General purpose I/O status and configuration register

IR Transmitter

The IR transmitter peripheral sends consumer IR data using the so-called "Japanese Format" (JF) which is used by NEC and many other manufacturers using four FPGA pins. Details of the format are given below. The peripheral can send up to 64 bits of IR packet data with the packet transmission triggered by a write to the control register. A write to the control register also turns on an output (usually tied to a visible LED) to indicate that a packet is being sent.

The first bit transmitted is the LSB of Register 0. The second bit sent is bit 1 of Register 0. The fifth bit is bit 0 of Register 1. This continues up to bit 3 of Register 15 which is the 64th bit sent. Note that the format of these registers differs from that of the IR receiver peripheral.

The high bit of the control register is the polarity of the output. A polarity value of one inverts the output and a zero does not. This is important so as not to leave the IR LEDs on when data is not being sent. The low 6 bits of the control register contain how many bits to send in the packet. Sending a packet with zero bits is allowed and is usually interpreted as a "repeat" packet at the receiver. The IR data and the trigger write are usually sent as a single command but this is not a requirement.

The JF format starts by sending 9.2 milliseconds of one as an AGC pulse, followed by 4 milliseconds of off, and then the data bits. Each data bit has an on time pulse of 650 microseconds followed by an off time. The off time is 450 microseconds for a data bit of zero and 1600 microseconds for a data bit of one.

The lowest numbered pin on the FPGA connector is the output from the FPGA to the IR receiver. The second pin is an output that is high when an IR packet is being sent. This pin would usually be connected to an LED to show activity. The remaining two pins are used for general purpose I/O. The low two bits of register 17 are the value of the pins and bits 4 and 5 control the direction of the third and fourth connector pins respectively. A one in the data direction bits indicates an output. Reads and write to the low two bits of register 17 read and set the pins depending on the values in the data direction bits.

The program `irxmit.c` converts a string of ASCII ones and zeros into the `dpxmit` command string needed to send the IR packet corresponding to the input string. Use this program as a guide to understanding how to write the peripheral registers in order to send a packet.

© Copyright Demand Peripherals, Inc. All rights reserved

The IR Receiver uses eighteen 8-bit registers:

Register 0: IR packet data (in low 4 bits)
Register 1: IR packet data (in low 4 bits)
Register 2: IR packet data (in low 4 bits)
Register 3: IR packet data (in low 4 bits)
Register 4: IR packet data (in low 4 bits)
Register 5: IR packet data (in low 4 bits)
Register 6: IR packet data (in low 4 bits)
Register 7: IR packet data (in low 4 bits)
Register 8: IR packet data (in low 4 bits)
Register 9: IR packet data (in low 4 bits)
Register 10: IR packet data (in low 4 bits)
Register 11: IR packet data (in low 4 bits)
Register 12: IR packet data (in low 4 bits)
Register 13: IR packet data (in low 4 bits)
Register 14: IR packet data (in low 4 bits)
Register 15: IR packet data (in low 4 bits)
Register 16: IR transmit trigger and control register
Register 17: GPIO control and status

Dual Quadrature Decoder

The dual quadrature decoder provides two independent channels of quadrature decoding suitable for use with high speed wheel encoders or low speed user interface controls. The quadrature counts are given as 16-bit one's complement signed numbers. The maximum input clock frequency is 1.25 megahertz. The counter sample period is crystal controlled and is set to 4 milliseconds. A precise sample window means the quadrature count can be used directly as an estimate of the speed of the source.

The dual quadrature decoder has two 16-bit registers. Every 4 milliseconds the values of both counters is sent to the host as a read response to a read of two auto-incrementing words starting at Register 0. Register 0 has the count for the quadrature signals appearing on the two low numbered pins on the connector, and Register 2 has the count for the quadrature signals appearing on the two high numbered pins on the connector.

The Dual Quadrature Decoder uses two 16-bit registers:

Register 0: Quadrature count A as a 16-bit two's complement signed number from the two low number pins on the connector.
Register 2: Quadrature count B as a 16-bit two's complement signed number from the two high number pins on the connector.

Quad/Octal RAM-Based Pattern Generator

The RAM based pattern generator let you put an arbitrary string of ones or zeros on either four or eight FPGA pins. The host loads the pattern into one side of a dual port RAM. The other side of the dual port RAM is connected to the output pins. A timer steps through the addresses in the RAM to generate the changing bit pattern. The rate as which the address lines increment is controlled by a clock select line and by a period counter. The input from the host is 16 bits wide and the output from the FPGA is four or eight bits wide. The high bits of the host side data are output before the lower bits. A register specifies the maximum value that the output addresses are allowed to reach. The total memory is 16 K bits which gives the host up to 1 K words to write and the 4 bit output up to 4 K words to output or 2 K words for the 8 bit output.

© Copyright Demand Peripherals, Inc. All rights reserved

The host side is a FIFO. The user sets "address pointer" and can then write up to 255 words of data to the FIFO. Each write to register 0 increments the host side address pointer to the next RAM location.

As an example consider the following `dpxmit` commands to a quad pattern generator at slot #2.

```
0b 02 02 01 00 00
09 02 00 04 80 00 00 00 00 00 00 00
0b 02 04 01 00 0f
0b 02 06 01 c0 03
```

The first command sets the internal address pointer to a value of zero. The second command writes four 16-bit words to the RAM. The pattern is all zeros with a single set bit in word 0. Note the auto-increment bit is turned off when writing to the RAM (a command of 09 instead of 0b). This is because the RAM is seen as a FIFO. The third command sets the maximum output address to write to the output before looping. The output addresses are zero-indexed so to get all four 16 bit input words mapped to 4 bit output words we set the maximum output address to 0f. The final command selects the 100 Hertz clock as the clock source and sets the dwell time on each output word to 4 clocks (the step rate is also zero-indexed). So the above commands generate a 40 milliseconds high going pulse followed by a low time of 600 milliseconds (15 * 40).

The 4-bit Pattern Generator uses four 16-bit registers:

Register 0: Pattern data: FIFO. A read or write increments the address pointer. Typically set the pointer then write up to 256 words of pattern data.
Register 2: Address pointer: 10 bit address into the pattern data RAM. Each read or write of pattern data increments this to the next location.
Register 4: Output max addr: The output starts at zero and increments to this address and then repeats. The increment rate is controlled by the period clock below. Note that the input address has 10 bits (1Kx16) and that this the output address (this one) has 12 bits (4Kx4).
Register 8: Output period: High three bits select the clock source. Low 10 bits set the output step rate in terms of the selected clock source

The 8-bit Pattern Generator uses four 16-bit registers

Register 0: Pattern data: FIFO. A read or write increments the address pointer Typically set the pointer then write up to 256 words of pattern data.
Register 2: Address pointer: 10 bit address into the pattern data RAM. Each read or write of pattern data decrements this to the next location.
Register 4: Output max addr: The output starts at this address and decrements to zero and then repeats. The decrement rate is controlled by the period clock above. Note that the input address has 10 bits (1Kx16) and that this the output address (this one) has 11 bits (2Kx8).
Register 8: Output period: High three bits select the clock source. Low 10 bits set the output step rate in terms of the selected clock source.

Clock sources are as follows:

0: 20 MHz
1: 10 MHz
2: 1 MHz
3: 100 KHz
4: 10 KHz
5: 1 KHz
6: 100 Hz
7: 10 Hz

© Copyright Demand Peripherals, Inc. All rights reserved

RC Receiver

Radio control systems encode the channel data as the position or width of a string of pulses. A "frame" is a complete sequence of these pulses with a leading sync interval. The sync interval is always at least 3 milliseconds long. The first edge after the sync interval is the start of the pulse for channel #1. The user value for a channel is the time from the leading edge of the channel pulse to the leading edge of the next channel. The signal may be inverted so we look for edges and not specific values. This circuit records both the high and the low times for each pulse. This additional information can be used by the host to help determine if the signal is valid or not. The channel time is the sum of the high and low times for that channel. We send the data up to the host at an edge count of two times the number of channels if none of the intervals exceeded 3.2 milliseconds.

The pulse interval registers has two fields. The MSB is the value of the input during the interval being reported by the lower 15 bits. The lower 15 bits are the duration of the interval in units of 100 nanoseconds.

The low three bits of the configuration register specify the number of channels to expect in the received signal. The MSB of the configuration register controls the "PKTLED" line. It is up to the Linux driver to turn on or off the LED depending on whether or not a valid RC signal is being detected. The accumulated pulse widths are sent up to the host after the completion of reading the number of channels specified.

The first pin on the BaseBoard4 connector is the input from the RC receiver to the FPGA. The second pin is an output that is high when an RC packed is being received. The second pin would usually be connected to an LED to show activity. The remaining two pins are used for general purpose I/O. The low two bits of Register 34 are the value of the pins and bits 4 and 5 control the direction of the third and fourth connector pins respectively. A 1 in the data direction bits indicates an output. Reads and write to the low two bits of Register 34 read and set the pins depending on the values in the data direction bits.

The RC Receiver uses eighteen 16-bit registers:

Register 0: Pulse #1 interval (16 bits) Bit 15=polarity. Bits 14-0 are duration in 100 nS steps
Register 2: Pulse #1 interval (16 bits)
Register 4: Pulse #2 interval (16 bits)
Register 6: Pulse #2 interval (16 bits)
Register 8: Pulse #3 interval (16 bits)
Register 10: Pulse #3 interval (16 bits)
Register 12: Pulse #4 interval (16 bits)
Register 14: Pulse #4 interval (16 bits)
Register 16: Pulse #5 interval (16 bits)
Register 18: Pulse #5 interval (16 bits)
Register 20: Pulse #6 interval (16 bits)
Register 22: Pulse #6 interval (16 bits)
Register 24: Pulse #7 interval (16 bits)
Register 26: Pulse #7 interval (16 bits)
Register 28: Pulse #8 interval (16 bits)
Register 30: Pulse #8 interval (16 bits)
Register 32: RC receiver status and configuration register
Register 34: GPIO data direction and value registers

Quad Servo Motor Controller

The quad servo motor controller can control up to four servo motors using four FPGA pins. Each output has a pulse width between 0 milliseconds and 2.5 milliseconds. The repetition time for all four servos is 10 milliseconds.

© Copyright Demand Peripherals, Inc. All rights reserved

Each servo has a window of 2.5 milliseconds and the value in the register specifies the 50 nanosecond count at which the pin goes high. The pin stays high until the count reaches 2.5 milliseconds, a count of 50,000 50 nanosecond pulses. To get a pulse width of 1.0 milliseconds you would subtract 1.0 from 2.5 giving how long the low time should be. A low time of 1.5 milliseconds would give a count of 30,000 clock pulses, or a register value of 16'h7530.

The Quad Servo Motor Controller uses four 16-bit registers:

Register 0: Servo 0 low pulse width in units of 50 nS.
Register 2: Servo 1 low pulse width in units of 50 nS.
Register 4: Servo 2 low pulse width in units of 50 nS.
Register 6: Servo 3 low pulse width in units of 50 nS.

Octal Servo Motor Controller

The octal servo motor controller can control up to eight servo motors using eight FPGA pins. Each output has a pulse width between 0 milliseconds and 2.5 milliseconds. The repetition time for all eight servos is 20 milliseconds.

Each servo has a window of 2.5 milliseconds and the value in the register specifies the 50 nanosecond count at which the pin goes high. The pin stays high until the count reaches 2.5 milliseconds, a count of 50,000 50 nanosecond pulses. To get a pulse width of 1.0 milliseconds you would subtract 1.0 from 2.5 giving how long the low time should be. A low time of 1.5 milliseconds would give a count of 30,000 clock pulses, or a register value of 16'h7530.

The Octal Servo Motor Controller uses eight 16-bit registers:

Register 0: Servo 0 low pulse width in units of 50 nS.
Register 2: Servo 1 low pulse width in units of 50 nS.
Register 4: Servo 2 low pulse width in units of 50 nS.
Register 6: Servo 3 low pulse width in units of 50 nS.
Register 8: Servo 4 low pulse width in units of 50 nS.
Register 10: Servo 5 low pulse width in units of 50 nS.
Register 12: Servo 6 low pulse width in units of 50 nS.
Register 14: Servo 7 low pulse width in units of 50 nS.

Bipolar Stepper Motor Controller

The bipolar stepper motor controller uses four FPGA pins to control the A and B windings of a stepper motor with a bipolar configuration. The bipolar stepper motor controller has a read/write register for the number of steps to go, a read/write register for step rate (step period actually), a read/write flag to indicate full or half steps, and a write-only register that adds or removes steps from the target step count.

The output pins contain the A winding direction, the A winding On/Off control, the B winding direction, and the B winding OnOff control. Outputs are inverted to match the power-on state of the FPGA. The outputs are as follows for full and half steps (without output inverters):

© Copyright Demand Peripherals, Inc. All rights reserved

Outputs for Full Steps				
Step	A Dir	A On/Off	B Dir	B On/Off
0	x	0	N	1
1	N	1	x	0
2	x	0	S	1
3	S	1	x	0

Outputs for Half Steps				
Step	A Dir	A On/Off	B Dir	B On/Off
0	x	0	N	1
1	N	1	N	1
2	N	1	x	0
3	N	1	S	1
4	x	0	S	1
5	S	1	S	1
6	S	1	x	0
7	S	1	N	1

The Bipolar Stepper Motor Controller uses three 16-bit registers:

Register 0: 12 bit target step count, decremented to zero
Register 2: 12 bit value that is synchronously added to the target, write only
Register 4: High 5 bits are the setup, low 8 bits are the period

The setup portion of Register 4 has the following bit meanings:

Bit 4: On/Off - turns outputs off if bit is cleared
Bit 3: Direction
Bit 2 Half/Full - 1==half
Bit 1,0 Input clock select
00 period clock is 1 microsecond
01 period clock is 10 microseconds
10 period clock is 100 microseconds
11 period clock is 1 millisecond

Unipolar Stepper Motor Controller

The unipolar stepper motor controller uses four FPGA pins to control the four windings, A, B, C and D, of a stepper motor with a unipolar configuration. The unipolar stepper motor controller has a read/write register for the number of steps to go, a read/write register for step rate (step period actually), a read/write flag to indicate full or half steps, and a write-only register that adds or removes steps from the target step count.

© Copyright Demand Peripherals, Inc. All rights reserved

The four output pins contain the on/off status for all four motor coils. The output pins are assigned as coil A, coil B, coil C, and coil D. Outputs are inverted to match the power-on state of the FPGA. The outputs are as follows for full and half steps (without output inverters):

Outputs for Full Steps				
Step	A	B	C	D
0	1	1	0	0
1	0	1	1	0
2	0	0	1	1
3	1	0	0	1

Outputs for Half Steps				
Step	A	B	C	D
0	1	1	0	0
1	0	1	0	0
2	0	1	1	0
3	0	0	1	0
4	0	0	1	1
5	0	0	0	1
6	1	0	0	1
7	0	1	0	0

The Unipolar Stepper Motor Controller uses three 16-bit registers:

Register 0: 12 bit target step count, decremented to zero
Register 2: 12 bit value synchronously added to the target, write only
Register 4: High 5 bits are the setup, low 8 bits are the period

The setup portion of Register 4 has the following bit meanings:

Bit 4: On/Off - turns outputs off if bit is cleared
Bit 3: Direction - 1==abcd, 0=dcba
Bit 2 Half/Full - 1==half
Bit 1,0 Input clock select
00 period clock is 1 microsecond
01 period clock is 10 microseconds
10 period clock is 100 microseconds
11 period clock is 1 millisecond

© Copyright Demand Peripherals, Inc. All rights reserved

Quad Simple I/O Port

The Quad Simple I/O Port provides four FPGA pins for either input or output. When configured as an input, each pin may additionally be configured to send up to the host the new value of the input when the input changes.

The Simple4 peripheral has three 8-bit registers located at addresses 0, 1 and 2. A read of Register 0 returns the current values at all four pins and a write sets the values for those pins configured as outputs. Bit 0 in each register corresponds to the lowest numbered pin on the BaseBoard4 connector.

Register 1 is the direction register with a one indicating an output and a zero indicating an input. All four pins default to inputs after power up.

Register 2 is the interrupt-on-change configuration register. Here an "interrupt" refers to asynchronously sending to the host a USB packet that contains the most recent values available at the pins. Setting a bit to one enables interrupt-on-change for the corresponding pin. The power up default is to turn off interrupt-on-change. A change on an interrupt enabled pin causes the peripheral to send to the host a read response packet for a read of one 8-bit register starting at register 0.

The Quad Simple I/O Port uses three 8-bit registers:

Register 0: Read: Current values of all pins. Write: Sets values of pins configured as outputs.
Register 1: Pin Direction: Output ==1. Input == 0. Default == 0.
Register 2: Interrupt-on-Change: == 1. Default ==0.

Octal Simple I/O Port

The Octal Simple I/O Port provides eight FPGA pins for either input or output. When configured as an input, each pin may additionally be configured to send up to the host the new value of the input when the input changes.

The Simple4 peripheral has three 8-bit registers located at addresses 0, 1 and 2. A read of Register 0 returns the current values at all eight pins and a write sets the values for those pins configured as outputs. Bit 0 in each register corresponds to the lowest numbered pin on the BaseBoard4 connector.

Register 1 is the direction register with a one indicating an output and a zero indicating an input. All eight pins default to inputs after power up.

Register 2 is the interrupt-on-change configuration register. Here an "interrupt" refers to asynchronously sending to the host a USB packet that contains the most recent values available at the pins. Setting a bit to one enables interrupt-on-change for the corresponding pin. The power up default is to turn off interrupt-on-change. A change on an interrupt enabled pin causes the peripheral to send to the host a read response packet for a read of one 8-bit register starting at register 0.

The Octal Simple I/O Port uses three 8-bit registers:

Register 0: Read: Current values of all pins. Write: Sets values of pins configured as outputs.
Register 1: Pin Direction: Output ==1. Input == 0. Default == 0.
Register 2: Interrupt-on-Change: == 1. Default ==0.

Tone Generator

The Tone Generator peripheral is a simple, monophonic audio tone generator using four FPGA pins. It also offers 8-bit volume control and tone duration as short as 10 milliseconds.

© Copyright Demand Peripherals, Inc. All rights reserved

The lowest numbered pin on the BaseBoard4 connector is the output from the FPGA to the audio amplifier. The remaining three pins are used for general purpose I/O. The low three bits of register 2 are the values of the pins and bits 4-6 control the direction of the pins. A one in the data direction bits indicates an output. Reads and writes to the low three bits of register 2 read and set the pins depending on the values in the data direction bits. The three general purpose I/O pins default to inputs.

A "note" consists of four fields: a duration (in units of 10 milliseconds), a clock source (1 microsecond or 100 nanoseconds), an 8-bit volume, and the frequency. The first three fields are sent in a 16-bit write to Register 0 followed by a write of the frequency. The control information always has bit 15 set, and the frequency always has bit 15 clear. The duration is in bits 9 to 14, the clock source is bit 8, and the volume is bits 0 to 7. For example, here are the two 16-bit writes to generate a 1 KHz tone for one-tenth of a second. Duration = 10. Clock source = 100 nanoseconds. The volume is set to 8'h10. The frequency count is the length of a half period, or 500 microseconds, or 5,000 counts of the 100 nanosecond clock, or 15'h1388. The control word has bit 15 set; the frequency does not, and the two writes would look like:

```
{1'b1,duration,clksrc,volume}    = 1001010000010000
{1'b0,freq_count}                = 0001001110001000
```

Register 0 is the front end to a FIFO. You can write alternating control and frequency words until the FIFO is full. A write to a full FIFO returns how many words were not written. Use this information to send the notes again after the FIFO has had a chance to drain a little.

The duration is set by an input clock of 10 milliseconds. Since the write of a note from the host can occur anywhere inside the 10 millisecond clock period, the first note is usually less than 10 milliseconds. If this is a problem you can give the very first note a duration of 1 and a volume of 0. This will synchronize the duration timer to the 10 millisecond duration clock for the rest of your notes.

The Tone Generator uses two 16-bit registers:

Register 0: FIFO input for tone generator data
Register 2: General purpose IO control and status

Dual Watchdog Timer

The Dual Watchdog Timer peripheral has two independent watchdog timers with an active low output using four FPGA pins. The watchdog inputs are on the first and third pins of the peripheral, and the outputs are on the second and fourth pins.

The timeout period is specified as an 8-bit value and is in seconds. Thus the maximum watchdog timeout is about 255 seconds. If enabled, the counter is decremented to zero at which time the corresponding output goes low for one second. The outputs are active low so that the non-alarm state of the output matches the power-on state of the FPGA. Any edge, either positive or negative, at the timer input pin resets the watchdog timer to its initial count.

The enable is the MSB of the control register and is set to one to enable the watchdog timer. Both timers are disabled by default. Software may reset a timer by (re-)writing a the enable and timeout period.

When either watchdog timer expires a message is sent to the host to report the event. The message is a read response for a read of four consecutive 16-bit registers starting at register 0.

The Dual Watchdog Timer uses two 16-bit registers:

Register 0: Watchdog #0 enable and 8 bits of timeout period
Register 2: Watchdog #1 enable and 8 bits of timeout period
Register 4: Watchdog #0 current count (i.e. time until alarm)
Register 6: Watchdog #1 current count (i.e. time until alarm)

© Copyright Demand Peripherals, Inc. All rights reserved

Appendix A: dpdump.c

```
/*
 * Name: dpdump.c
 *
 * Description:  Dump SLIP encoded packets from any of the serial ports
 *               specified on the command line.  Each packet is put onto
 *               a single output line.  Bytes are displayed in hex.
 *               Each line starts with the number of the serial port as
 *               it appeared on the command line.  That is, the first
 *               specified has its lines displayed starting with "0: XXXX"
 *               and the second port as "1: XXXX".
 */

#include <stdio.h>
#include <stdlib.h>
#include <termio.h>
#include <ctype.h>
#include <sys/fcntl.h>
#include <sys/time.h>


// Define the SLIP characters and SLIP decoder states
#define SLIP_END      ((unsigned char) 192)
#define SLIP_ESC      ((unsigned char) 219)
#define INPKT_END     ((unsigned char) 220)
#define INPKT_ESC     ((unsigned char) 221)
// Waiting for a packet to start; in a a packet; or in an escape sequence
#define AWAITING_PKT  (1)
#define AWAITING_END  (2)
#define ESCAPED       (3)


/* Demand Peripherals protocol maximum packet size */
#define PKTLEN  514

/* Maximum number of connected FPGA boards */
#define MXBOARD 8

/* The maximum number of peripherals per board */
#define MXPERI  16
```

 © Copyright Demand Peripherals, Inc. All rights reserved

```
/* Forward and external references */
void process_packet(int);
extern int errno;


/* Define a data structure to hold the information about each board
 * connected via USB serial port passed in on the command line. */
typedef struct {
    char *name;                 // the board file name from the cmd line
    char  fd;                   // File descriptor. Non-negative if opened
    unsigned char rpkt[PKTLEN]; // the received packet
    int   rindx;                // number of occupied byte in rpkt
    int   slstate;              // state of the SLIP decoder on this board
} BOARD;

/* Allocate the board info structures */
BOARD board[MXBOARD];




int main (int argc, char *argv[])
{
    fd_set rfds;                /* bit masks for select statement */
    int    mxfd;                /* Maximum FD for the select statement */
    int    fd;
    int    rdret;               /* read() return value */
    int    nboard = 0;          /* number of working boards */
    int    i;                   /* loop variable for different boards */
    int    cndx;                /* loop variable for walking the received data */
    int    strtrindx;           /* where in rpkt we start processing */
    struct  termios  tbuf;

    if (argc <= 1) {
        printf("Usage: %s [USB tty port] [USB tty port] [USB tty port] ...\n", argv[0]);
        exit(1);
    }
    if (argc > MXBOARD) {
        printf("%s: At most %d input ports are allowed\n", argv[0], MXBOARD);
```

 © Copyright Demand Peripherals, Inc. All rights reserved

```
        exit(1);
    }

    /* Loop through and initialize all of the port names on the command line */
    for (i = 1; i < argc; i++) {
        if ((fd = open(argv[i], O_RDWR | O_NDELAY,0)) < 0 ) {
            printf("Unable to open USB tty port %s\n", argv[i]);
            exit(1);
        }
        tbuf.c_cflag = CS8|CREAD|B115200|CLOCAL;
        tbuf.c_iflag = IGNBRK ;
        tbuf.c_oflag = 0;
        tbuf.c_lflag = 0;
        tbuf.c_cc[VMIN] = 1; /* character-by-character input */
        tbuf.c_cc[VTIME]= 0; /* no delay waiting for characters */
        if (tcsetattr(fd, TCSANOW, &tbuf) < 0) {
            exit(1);
        }

        /* Update the Channel structure with the opened file info */
        board[i-1].name = argv[i];     // (-1) since i starts at 1
        board[i-1].fd = fd;
        board[i-1].rindx = 0;
        board[i-1].slstate = AWAITING_PKT;
        nboard++;
    }

    while(1) {
        FD_ZERO(&rfds);                     /* Set up the select() stuff */
        mxfd = 0;
        for (i = 0; i < nboard; i++ ) {
            if (board[i].fd >= 0) {
                FD_SET(board[i].fd, &rfds);
                mxfd = (board[i].fd > mxfd) ? board[i].fd : mxfd;
            }
        }
        (void) select(mxfd+1, &rfds, (fd_set *)NULL, (fd_set *)NULL,
                (struct timeval *) NULL);

        /* Loop through the return fd's to get SLIP packet data from boards */
```


© Copyright Demand Peripherals, Inc. All rights reserved

```
    for (i = 0; i < nboard; i++ ) {
        if ( ! FD_ISSET(board[i].fd, &rfds))
            continue;

        /* at this point we have an FPGA board that is sending us some */
        /* data.  Get the data and do SLIP decoding on it.... */

        rdret = read(board[i].fd, &(board[i].rpkt[board[i].rindx]),
                     (PKTLEN - board[i].rindx));

        if (rdret > 0) {
            /* Got valid data.  Walk and shift it to remove SLIP encoding */
            strtrindx = board[i].rindx;  // needed since rindx can change
            for (cndx = strtrindx; cndx < (strtrindx + rdret); cndx++) {
                if (board[i].slstate == AWAITING_PKT) {
                    if (board[i].rpkt[cndx] == SLIP_END)
                        board[i].slstate = AWAITING_END;   // now in a packet
                }
                else if (board[i].slstate == AWAITING_END) {
                    if (board[i].rpkt[cndx] == SLIP_END) {
                        if (board[i].rindx == 0)  // Oops, not in proper sync
                            continue;
                        process_packet(i);   // Packet on board i is ready.
                        board[i].slstate = AWAITING_PKT;
                        board[i].rindx = 0;
                    }
                    else if (board[i].rpkt[cndx] == SLIP_ESC)
                        board[i].slstate = ESCAPED;
                    else {
                        // Just a normal packet byte.  Move it, if needed
                        if (board[i].rindx != cndx) {
                            board[i].rpkt[board[i].rindx] = board[i].rpkt[cndx];
                        }
                        board[i].rindx++;
                    }
                }
                else {  // must be ESCAPED
                    if (board[i].rpkt[cndx] == INPKT_END) {
                        board[i].rpkt[board[i].rindx] = SLIP_END;
                        board[i].rindx++;
```

 © Copyright Demand Peripherals, Inc. All rights reserved

```
                        board[i].slstate = AWAITING_END;
                    }
                    else if (board[i].rpkt[cndx] == INPKT_ESC) {
                        board[i].rpkt[board[i].rindx] = SLIP_ESC;
                        board[i].rindx++;
                        board[i].slstate = AWAITING_END;
                    }
                    else {  // Protocol violation.  Report it.
                        printf("%s: Protocol error on  %s\n", argv[0], board[i].name);
                        //board[i].slstate = AWAITING_PKT;
                        //board[i].rindx = 0;
                    }
                }
            }
        }
        else {
            /* Got a close or a read error from the USB port. */
            fprintf(stderr, "%s: Error on  %s\n", argv[0], board[i].name);
            exit(1);
        }
    }
  }
}


/* process_packet --- dispatch packet to proper handler
 *     Use board number and the peripheral number in the
 *     packet to pass this packet off to the right routine.
 *
 *     Packet and its length are in the board structure
 *     indexed by the input board ID.
 */
void process_packet(int boardid)
{
    int   cndx;

    /* for now, just print it */
    printf("%d: ", boardid);
    for (cndx = 0; cndx <board[boardid].rindx; cndx++)
        //if (isprint(board[boardid].rpkt[cndx]))
```


© Copyright Demand Peripherals, Inc. All rights reserved

```
            //printf("_%c ", board[boardid].rpkt[cndx]);
        //else
            printf("%02x ", board[boardid].rpkt[cndx]);

    printf("\n");
    fflush(stdout);     // Send output to any post processors
    return;
}
```


© Copyright Demand Peripherals, Inc. All rights reserved

Appendix B: dpxmit.c

```
/*
 * Name: dpxmit.c
 *
 * Description:  Send the packets entered at stdin as SLIP encoded packets to stdout.
 *               A packet is a string of two digit hexadecimal numbers separated by
 *               spaces and terminated by a newline.  For example:
 *                   06 01 00 01
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

// Max line size is about max packet size times three
#define MXPKT  780

// States for our parsing of the input line
#define WHITE    0
#define NONWHITE 1

// Define the SLIP characters
#define SLIP_END             192
#define SLIP_ESC             219
#define INPKT_END            220
#define INPKT_ESC            221

int main (int argc, char *argv[])
{
    unsigned char x;
    char *retl, retv;
    char line[MXPKT];
    int  state = WHITE;
    int  i, len;

    while (1) {
        retl = fgets(line, MXPKT, stdin);
        if (retl == (char *) 0)
            exit(0);
```


© Copyright Demand Peripherals, Inc. All rights reserved

```
        // Send opening SLIP_END character
        fprintf(stdout, "%c", SLIP_END);

        // Walk the line doing conversion on any non-white characters
        // This a state machine with two state: in-white and in-non-white
        len = strlen(line);
        for (i = 0; i < len; i++) {
            if (state == WHITE) {
                if (! isalnum(line[i]))
                    continue;
                else {
                    retv = sscanf(&line[i], "%02x", &x);
                    if (retv == 1) {
                        if (SLIP_END == x)
                            fprintf(stdout, "%c%c", SLIP_ESC, INPKT_END);
                        else if (SLIP_ESC == x)
                            fprintf(stdout, "%c%c", SLIP_ESC, INPKT_ESC);
                        else
                            fprintf(stdout, "%c", x);
                    }
                    else
                        break;
                    state = NONWHITE;
                }
            }
            else {
                // Scan until we find a non-white character
                if (isalnum(line[i]))
                    continue;
                else
                    state = WHITE;
            }
        }

        // Send closing SLIP_END character
        fprintf(stdout, "%c", SLIP_END);
    }
}
```


© Copyright Demand Peripherals, Inc. All rights reserved

Appendix C: irrec.c

```
/*  irrec: Convert a dpdump of IR data to a string of ones and zeros
 *
 * Usage:
 *      dpdump /dev/ttyUSB0 | irrec 6
 *
 * The single parameter is the slot number of the IR receiver
 * peripheral.  Data from all other peripherals is ignored.
 * Output is a line containing a single string consisting of
 * ones and zeros.
 *
 * For example, an input of
 *    0: 46 03 00 11 01 03 03 01 03 01 03 03 02 00 00 00 00 02 02 00 20 00
 * gives an output of
 *    0110000111010110000000011111111
 */

#include "stdio.h"
#include "stdlib.h"
  // Number registers that have IR data
#define SZ 16
  // Maximum number of IR bits to send
#define MXBIT 64

main(int argc, char *argv[])
{
    int targetslot;   // Read from this slot only
    int board;        // The board ID
    int cmd;
    int slot;         // Slot number from packet
    int reg;          // The starting register number
    int count;        // This many registers (err if not 17)
    int data[SZ];     // IR data as an array
    int nbits;        // Number of IR data bits in packet
    int xfercnt;      // Number of bytes not fetched (==0)
    int err;          // Number of items read from input line
    char bitout[MXBIT +1];  // Output string of 1 and 0's.
    int i;            // loop counter

    /* Get the slot number */
```


© Copyright Demand Peripherals, Inc. All rights reserved

```
    if (argc != 2) {
        fprintf(stderr, "%s: Slot number required as argument\n", argv[0]);
        exit(1);
    }
    err = sscanf(argv[1], "%d", &targetslot);
    if ((err != 1) || (targetslot < 2) || (targetslot > 9)) {
        fprintf(stderr, "%s: Slot must be between 2 and 9\n", argv[0]);
        exit(1);
    }

    while (1) {
        /* Read all 23 entries on the input line */
        err = fscanf(stdin, "%1x : %x %x %x %x %x %x %x %x %x %x %x \
                  %x %x %x %x %x %x %x %x %x %x %x", &board, &cmd, &slot,
                  &reg, &count, &data[0], &data[1], &data[2], &data[3],
                  &data[4], &data[5], &data[6], &data[7], &data[8],
                  &data[9], &data[10], &data[11], &data[12], &data[13],
                  &data[14], &data[15], &nbits, &xfercnt);

        if ((err != 23) || (slot != targetslot)) {
            printf("Got %d numbers.  Slots=%d:%d\n", err, slot, targetslot);
            if (err < 0)
                exit(1);
            else
                continue;
        }

        // Number of IR bits is in low 6 bits of nbits
        nbits = 0x3f & nbits;
        bitout[nbits] = (char) 0;

        // Walk down bit 0 of each data element print a 1 or 0
        while (nbits) {
            for (i = 0; i < 16; i++) {
                nbits--;
                bitout[nbits] = (data[i] & 1) ? '1' : '0' ;
                data[i] = data[i] >> 1;
                if (nbits == 0)
                    break;
            }
```


© Copyright Demand Peripherals, Inc. All rights reserved

```
        }
        printf("%s\n", bitout);
        fflush(stdout);
    }

    exit(0);
}
/*
 * Name: dpdump.c
 *
 * Description:  Dump SLIP encoded packets from the serial port specified
 *               on the command line.  Each packet is put onto a single
 *               output line.  Bytes are displayed in hex.
 */
```


© Copyright Demand Peripherals, Inc. All rights reserved

Appendix D: irrec.c

```
/*  irxmit: Convert a string of ones and zeros into the dpxmit
 *      commands to send the data using an irxmit peripheral.
 *
 * Output is a series of dpxmit command that can be fed directly to
 * dpxmit.
 *
 * For example an input of
 *     echo 11111111000000000110101110000110 | irxmit 2
 * gives an ouppput of
 *     0a 02 00 11 0f 0f 00 00 06 0d 01 06 00 00 00 00 00 00 00 00 20
 *
 * The single argument is the BaseBoard slot number that has the
 * irxmit peripheral attached.
 */

#include "stdio.h"
#include "stdlib.h"
#include "string.h"
  // Number of data registers
#define SZ 16
  // Maximum number of IR bits to send
#define MXBIT 64

main(int argc, char *argv[])
{
    char xbit[1000];                // The input string of bits
    int  err;                       // error code / return status
    int  slot;                      // Slot # for irxmit peripheral
    int  nbits;                     // Number of IR bits to send
    int  data[SZ];                  // IR data to send
    int  i,j;

    /* Get the IR xmit peripheral slot number */
    if (argc != 2) {
        fprintf(stderr, "%s: Slot number required as argument\n", argv[0]);
        exit(1);
    }
    err = sscanf(argv[1], "%d", &slot);
    if ((err != 1) || (slot < 2) || (slot > 9)) {
```

 © Copyright Demand Peripherals, Inc. All rights reserved

```
        fprintf(stderr, "%s: Slot must be between 2 and 9\n", argv[0]);
        exit(1);
    }

    while (1) {
        /* Read the input IR bit pattern as a string */
        err = fscanf(stdin, "%s", xbit);
        if (err < 0) {
            fprintf(stderr, "%s: Error reading IR xmit bits\n", argv[0]);
            exit(1);
        }

        nbits = strlen(xbit);
        if (nbits > MXBIT) {
            fprintf(stderr, "%s: Invalid IR xmit packet\n", argv[0]);
            if (xbit)
                free(xbit);
            continue;
        }

        for (j = 0; j < SZ; j++)
            data[j] = 0;

        // Walk the bits building the data to send
        j = 0;
        for (i = 0; i < nbits; i++) {
            // printf("%d %d %c\n", i, j, xbit[i]);
            if (xbit[i] == '1')
                data[j] = data[j] + (1 << (i & 3));
            if ((i % 4) == 3)
                j++;
        }

        // Write the dpxmit command that sends the data
        printf("0a %02x 00 11", slot);  // command bytes
        for (j = 0; j < SZ; j++)        // data bytes
            printf(" %02x", data[j]);
        printf(" %02x\n", nbits);
    }
```


© Copyright Demand Peripherals, Inc. All rights reserved

```
    exit(0);
}
```


© Copyright Demand Peripherals, Inc. All rights reserved