Owner's Manual for the

PulseBlasterDDS Ultra™

Complete Digital Excitation System:

Direct Digital Synthesis (DDS),

Arbitrary Waveforms, and TTL

SpinCore Technologies, Inc.
3525 NW 67th Avenue
Gainesville, Florida 32653, USA
Phone: (352)-271-7383

http://www.spincore.com

Congratulations and THANK YOU for choosing a design from SpinCore Technologies, Inc. We appreciate your business. At SpinCore we try to fully support the needs of our customers, so if you ever need assistance please contact us and we will strive to provide the necessary help.

SpinCore Technologies, Inc. makes every effort to verify the correct operation of the equipment. This equipment should NOT, however, be used in system where the failure of a SpinCore device will cause serious damage to other equipment or harm to a person

Contents

Section I: Introduction

1. Quick Product Overview

The PulseBlasterDDS(tm) series of Intelligent Pattern and Waveform Generation boards from SpinCore Technologies, Inc., couples SpinCore's unique Intelligent Pattern Generation processor core, dubbed PulseBlaster(tm), with Direct Digital Synthesis (DDS) AND Arbitrary Waveform Generation (AWG) technology for use in system control and waveform generation.

The PulseBlaster's(tm) state-of-the-art timing processor core, implemented in programmable logic, provides all the necessary timing control signals required for overall system control and waveform synchronization. By adding DDS and arbitrary waveform generation features, PulseBlasterDDS(tm) can now provide not only digital (TTL) but also analog output signals, meeting high-performance and high-precision complex excitation/stimuli needs of demanding users.

PulseBlasterDDS(tm) provides users the ability to control their systems through the generation of fully synchronized (digital and analog) excitation waveforms from a small form factor PC board, providing users a compelling price/performance proposition unmatched by any other device on the market today. Figure 1 presents sample capabilities of the board.

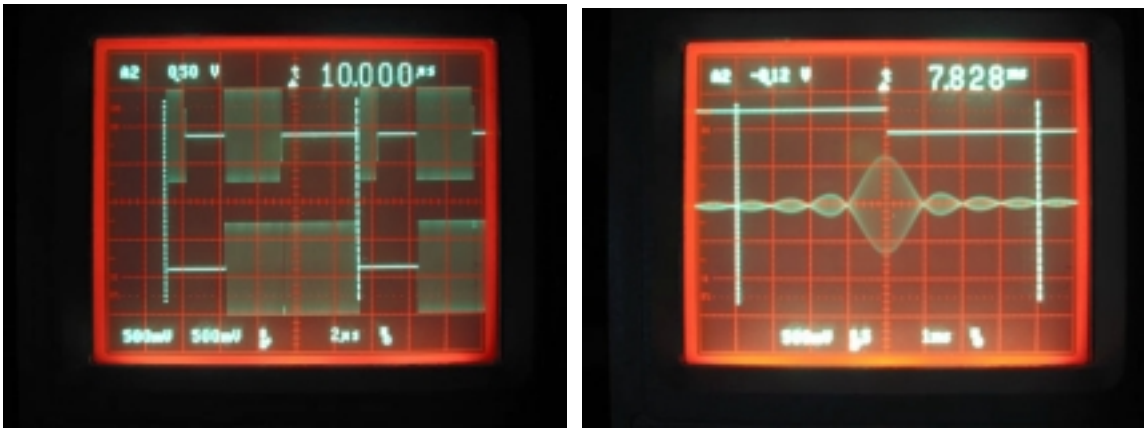

Figure 1. Sample PulseBlasterDDS(tm) output capabilities

2. Board Architecture

Block diagram

Figure 2 presents the general architecture of the PulseBlasterDDS(tm) board. The DDS, Modulator (optional), Pulse Timing and Gating cores have been integrated onto a single silicon chip. The internal DDS core has four frequency registers that are under the pulse program control. Prior to gating, the internal DDS signal can be amplitude-modulated with arbitrary waveforms (up to 32k samples) stored on on-board memory, also under pulse program control. Optionally, PulseBlasterDDS(tm) boards can be equipped with two independent off-chip DDS generators.

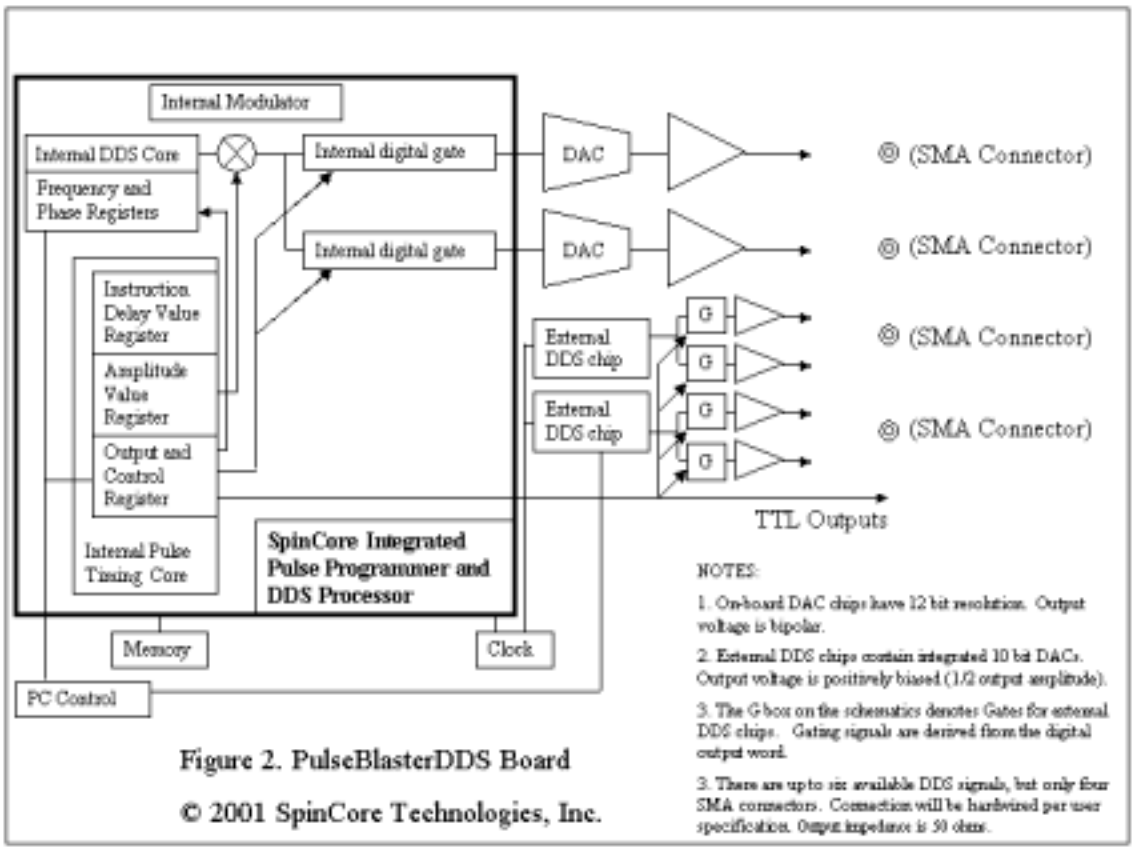

Fig. 2. PulseBlasterDDS(tm) board architecture.

- PulseBlasterDDS Ultra does not support Amplitude Control or external DDS chips

Output signals

This exciting product comes with up to two analog output channels that can be configured to output radio-frequency (RF/IF) waveforms, arbitrary waveforms, or a combination of both, and up to 10 digital output signal lines. The frequency and phase of the RF waveforms generated by the DDS output channels are under the complete control of the user and are specified through software programming. The arbitrary waveform patterns are also under the control of the user and are stored in on-board memory. PulseBlasterDDS(tm) also provides the ability to gate the output of the DDS channels allowing for independent pulsed RF operation of all output channels. With digital sampling rate of 100 MHz (max. clock frequency, internal DDS core only), analog signals up to approx. 50 MHz can be generated. Both analog output signals are available on on-board SMA connectors. The output impedance of the analog signals is 50 ohms.

The individually controlled digital (TTL/CMOS) output bits are capable of delivering +-25 mA per bit. Twenty-four of the output lines can be set to either the 5 V or 3.3V I/O TTL logic standard. Up to 16 digital TTL output lines are available on the PC bracket-mounted DB-25 connector; the remaining TTL signals are available on internal, flat-cable headers. The actual number of the available output bits depends on the model of PulseBlasterDDS(TM) board/design.

Timing characteristics

PulseBlasterDDS™' timing controller can accept either an internal (on-board) crystal oscillator or an external frequency source of up to 100 MHz. The innovative architecture of the timing controller allows the processing of either simple timing instructions (delays of up to 4,294,967,296 clock cycles), or double-length timing instructions (up to 2^52 clock cycles long - nearly 2 years with a 100 MHz clock!). Regardless of the type of timing instruction, the timing resolution remains constant for any delay – just one clock period (e.g., 10 ns for a 100 MHz clock, or 100 ns for a 10 MHz clock).

The core timing controller has a very short minimum delay cycle – only five clock periods for internal memory (512 words) models. This translates to a 50 ns pulse/delay/update with a 100 MHz clock. The external memory models (up to 32k words) have a nine-clock period minimum instruction cycle.

Instruction set

PulseBlasterDDS™' design features a set of commands for highly flexible program flow control. The micro-programmed controller allows for programs to include branches, subroutines, and loops at up to 16 nested levels – all this to assist the user in creating dense pulse programs that cycle through repetitious events, especially useful in numerous multidimensional spectroscopy and imaging applications.

For detailed description of the instruction set and programming, please consult the PulseBlaster's Owner Manual, available on SpinCore's web site (http://www.spincore.com/support).

Programming architecture and word definition

The PulseBlaster™ processor implements an 80 bit wide Very-Long Instruction Word (VLIW) architecture. The VLIW memory words have specific bits/fields dedicated to specific purposes, and every word should be viewed as a single instruction of the micro-controller. The maximum number of instructions that can be accommodated on on-board memory is 32k. The execution time of instructions can be varied and is under (self) control by one of the fields of the instruction word. All instructions have the same format and bit length, and all bit fields have to be filled. Figure 3 shows the fields and bit definitions of the 80-bit instruction word.

Bit Definitions for the 80-bit Very-Long Instruction Word (VLIW)

Output Pattern:79-56 **| Data Field**:55-36 **| OP Code**:35-32 **| Delay Count**:31-0

(24 bits) (20 bits) (4 bits) (32 bits)

Figure 3. Bit definitions of the VLIW instruction/memory word.

Several bits of the output pattern are dedicated to selecting the available frequency/phase registers and gating the RF/AWG signals, and are design/model specific. For example, for the design featuring a single internal DDS generator and two independently gated outputs (denoted, for convenience, TxDDS and RxDDS), the bit assignment is as follows:

#	Bit # (out of 24 bit of output pattern word)	Function
1	Bit # 0	Output Connector DB25, pin # 13
2	Bit # 1	Output Connector DB25, pin # 25
3	Bit # 2	Output Connector DB25, pin # 24
4	Bit # 3	Output Connector DB25, pin # 11
5	Bit # 4	Output Connector DB25, pin # 10
6	Bit # 5	Output Connector DB25, pin # 22

7	Bit # 6	Output Connector DB25, pin # 21
8	Bit # 7	Output Connector DB25, pin # 8
9	Bit # 8	Output Connector DB25, pin # 7
10	Bit # 9	Output Connector DB25, pin # 19
11	Bit # 10	Output Connector DB25, pin # 18
12	Bit # 11	Unavailable
13	Bit # 12	Unavailable
14	Bit # 13	Unavailable
15	Bit # 14	Unavailable
16	Bit # 15	Unavailable
17	Bit # 16	Unavailable
18	Bit # 17	Unavailable
19	Bit # 18	Unavailable
20	Bit # 19	Controls RxDDS Gate (0 = On, 1 = Off)
21	Bit # 20	Controls TxDDS Gate (0 = On, 1 = Off)
22	Bit # 21	Unavailable
23	Bit # 22	Frequency Select Bit 0
24	Bit # 23	Frequency Select Bit 1

For models featuring the internal modulator (AWG), the instruction word is 96 bit wide, and the additional bit-field codes for the signal amplitude of the analog output. Every word can have different signal amplitude, and words can be updated every nine clock cycles (for 32k memory word models).

Using the provided C functions, programming of the board can be accomplished by using calls to functions of the general form (delay, output control word).

Example:

```
output_control_word = FREQ0 | TX_DDS_GATE_ON | RX_DDS_GATE_OFF | 0xFFFF;
address = delay(1*us,output_control_word,program); //a 1 us interval in pulse sequence

output_control_word = FREQ3 | TX_DDS_GATE_OFF | RX_DDS_GATE_OFF | 0x0000;
address = delay(2*us, output_control_word,program); //a 2 us interval in pulse sequence
```

Please consult the provided Appendix for more information on programming the PulseBlasterDDS Ultra.

Programming of DDS registers

The registers of the internal DDS core(s) are programmed directly over the system bus. The following is an example use of the C function that was developed at SpinCore to aid in programming the available DDS registers:

```
/* Program DDS internal frequency registers */
   program_dds(1, 0, 6.25, dds);
   program_dds(1, 1, 0.1, dds);
```

In the above example, two different frequency registers (0 and 1) of the internal DDS core (#1) are loaded with two different frequency values, 6.25 MHz and 0.1 MHz.

For details on programming the DDS section of the board, please consult the appendix.

External triggering

PulseBlasterDDS™ can be triggered and/or reset externally via dedicated hardware lines. The two separate lines combine the convenience of triggering (e.g., in cardiac gating) with the safety of the "stop/reset" line. The required control signals are “active low” (or short to ground).

Summary

PulseBlasterDDS™ is a versatile, high-performance pulse/pattern TTL and RF/IF/AWG generator operating at speeds of up to 100 MHz and capable of generating pulses/delays/intervals ranging from 50 ns to over 2 years per instruction. It can accommodate pulse programs with highly flexible control commands of up to 32k program words. Its high-current output logic bits are independently controlled and some bits are 5/3.3 V user-selectable. Up to four analog channels (50 ohm impedance) are available from a single board.

2. Quick Installation Guide

PulseBlasterDDS™ boards are ready to use out of the box. After unpacking, they can be installed on your computer in any available ISA slot. Please shut down your computer and turn the power off when installing the board, and use a screw to fasten the bracket.

PulseBlasterDDS™ boards are factory pre-configured to operate with the following default settings:

ISA Base Address:	0x340.
Clock Oscillator:	Internal, installed on board; clock frequency as per customer specification
Output levels:	TTL 3.3 V

These settings can be changed using on-board jumpers. Please consult Chapter 6 in Section II for details regarding the jumpers' information and location.

The board can be used on computers running any operating system that supports the Industry Standard Architecture (ISA) bus, including DOS, Windows, QNX, and Linux. Third party drivers are available for protected operating systems (Win2000 and WinNT). Section III of this manual, "Test and Application Programs," describes sample programs that can be used *to program the board for operation* under Microsoft DOS/Windows95 operating systems. The C code described in this manual can also be compiled under most other operating systems as well. SpinCore's web site http://www.spincore.com/support serves as a repository of the software described in this manual.

2. PulseBlasterDDS™ Specific Information on C and Machine Language programming.

WE for Peripherals: This register is used to select the peripheral that is to be programmed. The value of this register that is used to select program memory is always zero and this is the default value for the register. A complete listing of the values and the associated hardware that can be programmed when appropriately set.

WE Register	Value (hex)
Program Memory	0
Integrated DDS 1	1

Table 3: Peripheral List

CLEAR ADDRESS COUNTER: The Address Counter is used to manufacture the memory address. The Address Counter is not loadable; it can only be cleared and started at zero. It is not possible to load a particular section of memory. All loads must start from either the beginning of memory, or wherever the Address Counter left off.

Flag Initialization Strobe: The output flags of the PulseBlasterDDS can be programmed while the device is in a reset state. This is useful to initialize flags after powering-up and to reset flags to a known state if a program must be aborted. Writing to the Flag Initialization Strobe register will toggle the line used to clock data into the output latches. Appendix B provide more information on how to use the Flag Initialization Strobe to program the output flags while the PulseBlasterDDS is in a reset state.

LOAD_MEMORY: This instruction is used to specify data that should be used to program the memory used by the device. Since the ISA data is taken only one byte at a time, the IBC must reconstruct the data word to be programmed. The data word is reconstructed in the IBC most significant byte first.

PROGRAMMING FINISHED: This instruction enables the pattern generator of the PulseBlaster™. This instruction prevents the pattern generator from accepting a hardware trigger or software start command before the device has been programmed. Once the design has been programmed, the **PROGRAMMING FINISIHED** command must be sent to arm the device for operation. After the pattern generator has been armed, any hardware trigger or software start command will cause the system to start operation. The PulseBlaster™ can be reset by issuing the **DEVICE_RESET** command. This will internally clear the **PROGRAMMING FINISHED** instruction and prevent the pattern generator from operating again until the IBC has been re-initialized.

4. ISA Readback Features

PulseBlaster products have the ability to provide status information to the user through a group of registers that are readable through the ISA Bus. The registers provide information on the status of the PulseBlaster series device as well as the firmware version number.

The register numbers provided below are offsets from the ISA port base address assigned to the PulseBlaster series card. All registers are 8 bits and are read only. If the users attempts to write to the port address associated with one of the registers, it will overwrite data in one of the IBC (ISA Bus Controller) control registers.

Only Register 0 and Register 1 are guaranteed to remain the same in future releases.

Register Definitions

Register 0 – Control Signal Register

- Bit 7: Reserved (Testing := Memory_Mode)
- Bit 6: IBC_Error – indicates if there has been an error programming the IBC
- Bit 5: Programming_Finished – indicates that the PulseBlaster series device has been programmed and is armed. The next trigger (either Hardware or Software) will start execution of the pulse programmer code.
- Bit 4: Idle -
- Bit 3: Waiting – The PulseBlaster series device has encountered a WAIT Op Code and is currently waiting for the next trigger (either Hardware or Software) to resume operation.
- Bit 2: Running – indicates that the PulseBlaster series device is current executing a program.
- Bit 1: Reset – the PulseBlaster series device is in a RESET state and must be reprogrammed before code execution can begin again.
- Bit 0: Stop – indicates that the PulseBlaster series device has encountered a STOP Op Code during program execution and has entered a stopped state.

Register 1 – Version Control Register

Bits[7..4]: Product Type Number

- PulseBlaster (external memory) = 0
- PulseBlaster (internal memory) = 1
- PulseBlasterDDS (PP only external memory) = 2
- PulseBlasterDDS (PP only internal memory) = 3
- PulseBlasterDDS (single channel DDS) = 4
- PulseBlasterDDS (dual channel integrated DDS) = 5
- PulseBlasterDDS (2 integrated, 2 external DDS) = 6
- PulseBlasterDDS (2 AWG, 2 DDS) = 7

Bits[3..0]: Release Version Number

Register 2 – Pulse Programmer Core Address LSB’s
Bits[7..0] : PP Core Current_Address[7..0]

Register 3 – Pulse Programmer Core Address MSB’s
Bits[7..0]: PP Core Current_Address[15..8]

Register 4 – PP Core Delay Counter Value (LSB)
Bits[7..0]: PP Core Delay_Count[7..0]

Register 5 - PP Core Delay Counter Value (MSB)
Bits[7..0]: PP Core Delay_Count[31..24]

Register 6
Bits[7]: PP Core Loop_Done
Bits[6..4]: PP Core Control_Mux[2..0]
Bits[3..0]: PP Core Control_Codes[3..0]

Register 7
Bits[7..0]: PP Core Branch_Address[7..0]

5. PulseBlaster™ Board Initialization

Initialization of the PulseBlasterDDS™ Board for operation involves a minimum of four steps. The steps are as follows:

1) Send **LOAD NUMBER OF BYTES PER WORD** instruction.
2) Send **SELECT PERIPHERAL DEVICE** instruction.
3) Send **CLEAR ADDRESS COUNTER** instruction.
   3.A. (Optional) loading data to memory.
4) Send **PROGRAMMING FINISHED** instruction.

If these four commands are not sent from a PC, the PulseBlasterDDS™ board will not run as desired. All four instructions are required as an attempt to ensure that the device has been programmed before it can be armed. The first time the board is used, the loading of the memory with data has to be performed between steps three and four, step 3.A above. Upon reset, all four instructions must be executed to restart the device again.

A Sample C code that implements the above commands is presented in the appendix.

5. ISA Bus Programming Issues

In order for the embedded intelligent pattern generator to operate, the memory it utilizes needs to be programmed, and appropriate control bytes have to be sent over the ISA Bus. To accomplish these tasks, a special controller, called IBC (ISA Bus Controller), was designed as the interface between a PC and the PulseBlaster™ Pulse/Pattern Generator.

The IBC handles programming the system memory for the pattern generator, initializes the board, and controls its operation. Once the system memory has been initialized, the IBC relinquishes control of the memory's data and address busses to the pattern generator. While the pattern generator is running, it has complete control of the memory buses. The IBC does have the power to reset the pattern generator and re-take control of the device. This allows for PC software to control the operation of the PulseBlaster™ Core Processor.

NOTE: The data taken off the ISA Bus is one byte wide - the 16-bit data capability of the ISA Bus was not used in order to conserve I/O pins on the microchip. Also, the IBC controller does not have the ability to write information to the bus. However, if necessary, three pins on PulseBlaster™'s board, namely RUNNING (J12-10), STOPPED (J12-7), and SYSTEM_RESET (J12-8) could be used to determine the status of the uPC.

ISA-Bus Base Port Address

Each device on the ISA Bus is mapped to a port address range. The port address is used to specify that data on the bus is for a particular peripheral device. The PulseBlaster™ board design has the ability to change its port address. This ability provides for the fact that other devices on the bus might have previously claimed certain port address ranges. Three control lines, running to the J4 header on the PulseBlaster™ card, allow one of eight port address ranges to be selected. The port address ranges are from the 'Base Address' to the 'Base Address + 7'. The Base Addresses that can be specified range from 0x260 to 0x360, see Table 3 in the next chapter "Header/Jumper Information."The factory pre-set value is 0x340.

ISA Bus Controller

The ISA Bus Controller uses three control signals from the ISA Bus: AEN, ~IOW, and Bus Clock. The control signals are used to decode the ISA Bus traffic. The Bus Clock signal is used by the IBC for timing, and it is completely independent of the system clock of the PulseBlaster™. The AEN signal is active high and indicates an address is on the bus, and that the address is from the DMA controller. In order to avoid traffic from the DMA controller, the IBC looks for this signal to be low. The ~IOW line specifies that a write (from the PC processors point of view) is occurring. A write indicates that the data on the bus is destined for a peripheral device. If both AEN and ~IOW are low, the data on the bus is being written to a peripheral device specified by the port address on the ISA Bus. The details of this hardware communication are hidden from the user standpoint if one uses the C language functions outp() or _outp().

Sending Control Commands over the ISA Bus

Once the on-chip ISA Bus Controller, IBC, finds the correct values for AEN and ~IOW, the address and data values are latched into control registers. The address is then decoded to determine if the bus traffic is addressed to the PulseBlaster™. If the address is in the defined range for the PulseBlaster™, then the address is used as a Control Command to drive the operation of the IBC. The IBC has eight distinct Control Commands - see Table 2. The Control Command values are specified in offsets from the Base Address. If the Control Command has data associated with it, the data latched off the ISA Bus is used; else the data buffer register is ignored.

6. Header/Jumper Information

The PulseBlasterDDS™ board is a configurable system. It allows the user to set jumpers on several headers on the PC card to select different modes of operation for the device.

Selecting ISA Bus Address: **Header J4, Pins 1-2, 3-4, and 5-6.**

The Base Addresses that can be specified range from 0x260 to 0x360. The default, factory pre-set value for the ISA PulseBlaster™ board is 0x340. This value can be changed, via jumpers on Header J4, according to the Table 3.

Base Address (in Hex)	Jumper Settings – Header J4 Pins 5-6	Pins 3-4	Pins 1-2
300	|	|	|
320	|	|	:
340	|	:	|
260	|	:	:
280	:	|	|
270	:	|	:
290	:	:	|
360	:	:	:

Table 3. Board's ISA-Bus Base Address Selection
(Legend: | jumper across pins, : no jumper)

(Default Value = 0x340, jumpers 1-2, 5-6).

Selecting output voltage levels: **Headers JPower1 and Jpower2**

The output signals are driven by latches/drivers capable of running off a 5.0-V or 3.3-V supply. The supply voltage for the drivers is selectable. Table 4, below, lists the configurations for 5.0-V and 3.3-V output driver operation.

5 V Operation
Jumper JPower1-1 across to JPower1-2
Jumper JPower2-1 across to JPower2-2
3.3 V Operation
Jumper JPower1-3 across to JPower1-4
Jumper JPower2-3 across to JPower2-4

Table 4. Output voltage selection

The JPower1 header selects the operating voltage for the output bits 0-15, and JPower2 independently selects the operating voltage for the output bits 16-23.

Output Bits **- Connector DB-25 (J10) and Header J9**

The following table lists the output bits for the PulseBlaster™ Pulse / Pattern Generator Board.

Signal	Location
Bit 0	J10-13
Bit 1	J10-25
Bit 2	J10-24
Bit 3	J10-11
Bit 4	J10-10
Bit 5	J10-22
Bit 6	J10-21
Bit 7	J10-8
Bit 8	J10-7
Bit 9	J10-19
Bit 10	J10-18
Bit 11	J10-5
Bit 12	J10-4
Bit 13	J10-16
Bit 14	J10-15
Bit 15	J10-1
Bit 16	J9-3
Bit 17	J9-5
Bit 18	J9-7
Bit 19	J9-9
Bit 20	J9-11
Bit 21	J9-13
Bit 22	J9--15
Bit 23	J9-17
Output Clock	J10-1
Running	J12-10
Stopped	J12-7
System Reset	J12-8

Table 5. Output bits and signals of the PulseBlasterDDS™ board.

Bits 15-0 are grouped on the external DB-25 connector (also marked as J10) provided for accessing the signals. The rest of the bits, Bits 23-16, are accessible on an internal IDC header J9. The table also lists several additional output signals that are available to the outside world, as described in the next subsection. All remaining pins on the DB-25 and all even pins of J9 connector are connected to the ground.

Using external trigger/reset lines - Header JTrigger.

HW_Trigger is a signal that is pulled high by default. When a falling edge is detected (e.g., when shorting pins 3-4), it initiates code execution. This trigger will also restart execution of a program from the beginning of the code if it is detected after the design has reached an idle state. The idle state could have been created either by reaching the STOP Op Code of a program, or by the detection of the HW_Reset signal.

The **HW_Reset** line is pulled high by a resistor. It can be used to halt the execution of a program by pulling it low (e.g., by shorting pins 5-6). When the signal is pulled low during the execution of a program, the controller resets itself back to the beginning of the program. Program execution can be resumed by either a software start command or by a hardware trigger.

Additional Output Signals – Connector DB-25 and Header J12.

The internal **Output Clock** signal is available to the outside world, as it is tied to the **DB25 connector, pin #1.** It is the clock signal used to latch patterns in the output buffers. This clock has been configured to have a relatively slow slew rate so as to avoid noise problems on a transmission line. This clock is **not** a 50% duty cycle clock. The width of the high part of the signal is one system clock period.

System Reset – Header J12 pin # is used to indicate (when low) to the external world that the uPC controller is in a reset state. It can be used in larger systems to monitor the state of the PulseBlaster™ design.

A signal that is similar to System Reset is the **Running** signal, **header J12 pin #.** It is driven high when the uPC is executing code. It is taken low when the uPC has entered either a reset of idle state.

The **Stopped** signal, **header J12 pin #,** is the last signal used to indicate the state of the uPC. Stopped is asserted when the uPC has encountered the stop command while normally executing code. This signal informs the external world that the uPC has successfully executed its program and has halted operation.

Header and Signal Locations

The location of the relevant headers and connectors on the PulseBlasterDDS™ board is presented in Figure 3.

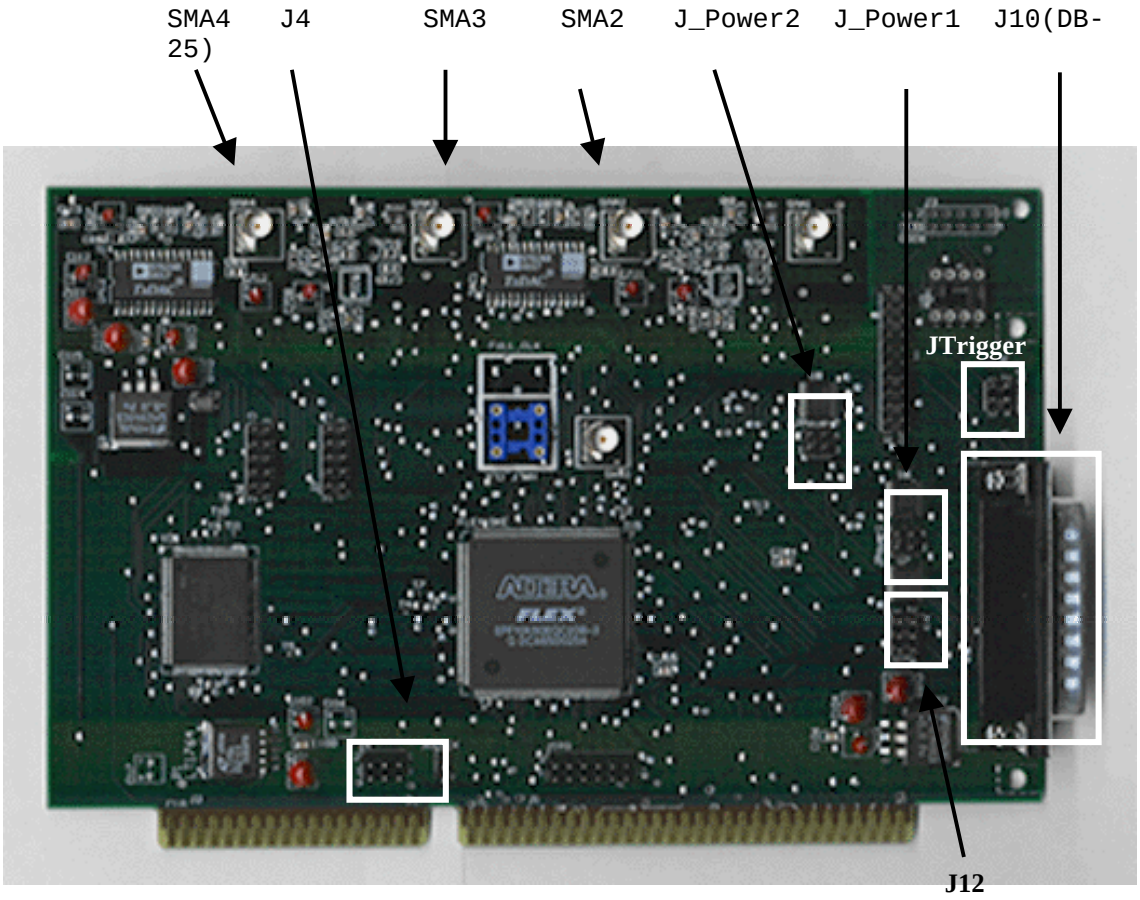

Figure 3. PulseBlasterDDS Board – header/connector locations.

Section III: DDS Basics

1. General Discussion of DDS Technology

Direct Digital Synthesis can be used to generate sinusoidal waveforms from their digital representations. The digital representation of a signal is discrete in both time and amplitude. To generate a waveform, the sample amplitudes of the waveform are calculated digitally and converted to a pseudo-analog waveform by a digital-to-analog converter (DAC). The DAC has a step-like output function, necessitating the use of an analog filter to smooth out the waveform and interpolate between sample values. The output of the analog filter approximates the desired analog waveform. A digital system with enough bits for representing the waveform accurately and a sampling rate that is high enough to allow rejection of higher order harmonics will produce an extremely accurate and harmonically pure signal. Figure 16 shows the signal generation line-up for a DDS system. The sections that follow will describe, in detail, the operation and performance characteristics of each of the blocks shown below.

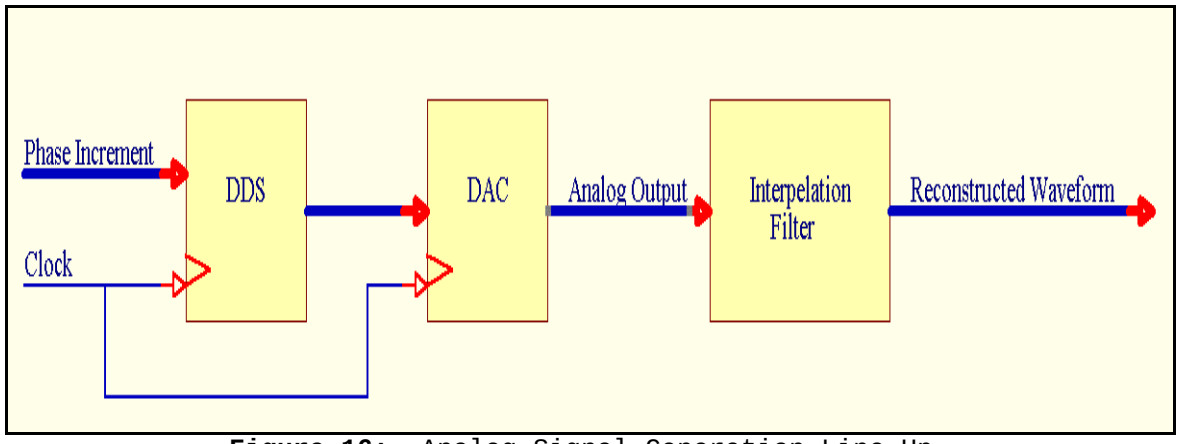

Figure 16: Analog Signal Generation Line-Up

Direct Digital Synthesizer (DDS) Architecture

The architecture of a DDS has three main blocks, see Figure 17. The first block is labeled the Frequency Control Register (FCR). The FCR is used to latch the Phase Increment (PI) value and synchronize it with the rest of the DDS. The PI controls the phase and frequency of the

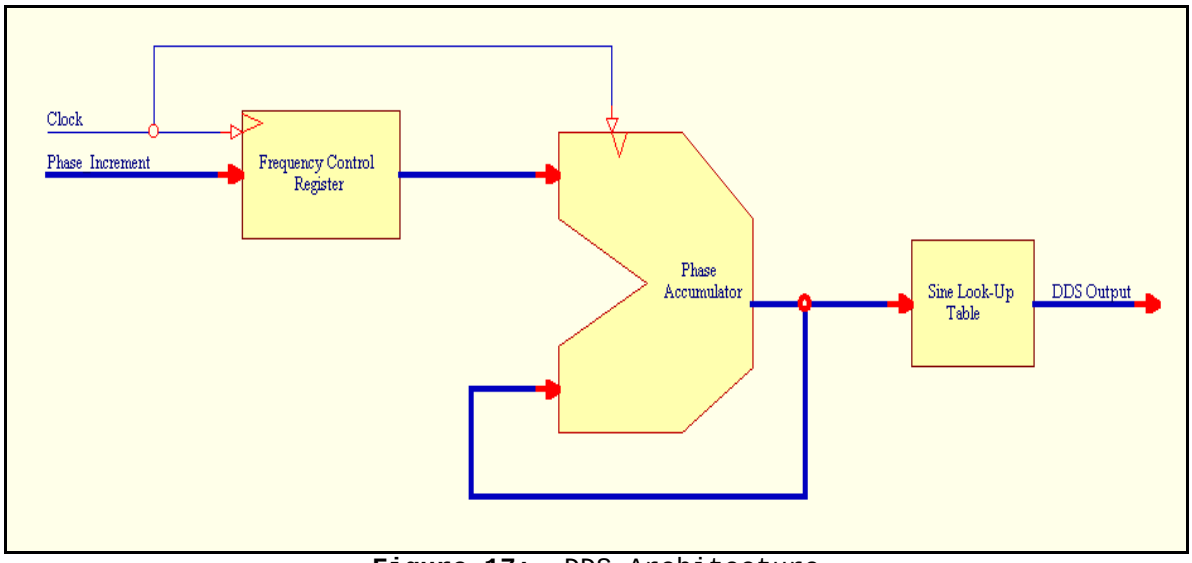

Figure 17: DDS Architecture

DDS output. The DDS uses phase accumulation to create sinusoidal signals of interest. It is the PI value that specifies the amount of phase to be accumulated between samples.

To better understand why phase accumulation is used, it should be noted that the phase difference between any two points of a constant frequency sinusoid, sampled at regular intervals, will be a constant value. In other words, the phase of a sinusoid is linear with respect to time. The linearity of a sinusoid's phase with respect to time is the key for generating sinusoidal waveforms using phase accumulation. All frequencies up to the Nyquist frequency can be created by accumulating a phase increment between sample times. Different frequencies are created by varying the size of the phase step between sample instances. Therefore, by controlling the phase steps in time, the frequency and phase of a signal can be controlled.

The next block in the DDS is the phase accumulator. The phase accumulator outputs a digital word representing a specific phase value. The accuracy of the digital word representing the phase is determined by the number of bits used in the digital representation.

$$\text{Phase Resolution} = \frac{2\pi}{2^N} \text{ (in radians),} \quad [4]$$

where N is the number of bits in the phase word. The accuracy of the digital representation can be increased by using a wider (more bits) phase accumulator. Since phase accumulation is used to generate frequencies of interest, the frequency resolution of the DDS is also directly controlled by the length of the PI input word. The frequency resolution can be expressed by

$$\text{Frequency Resolution} = \frac{F_{clk}}{2^N}, \quad [5]$$

where N is again the number of bits in the phase word and F_{clk} is the frequency of the clock driving the DDS system.

It should also be noted that the phase of a sinusoid repeats every 2π radians. This is beneficial for digital generation of sinusoidal signals. The phase accumulator will overflow every 2π radians and repeat itself in an exact analogy to the phase of a sinusoid.

The third block is the Look-Up Table (LUT). The LUT is used to convert a phase value to a corresponding amplitude in a normalized sinusoid. It is the output values from the LUT that are fed into a DAC. Generally, only part of the phase accumulator's output (the most significant bits) will be used by the LUT. This is done for several reasons. First, larger memories are slower than smaller memories using the same technology. The speed of the memory used to create the LUT is critical in the performance of a DDS system. Second, the performance characteristics of the DAC used to generate the analog waveform is the limiting factor in system performance. There is no need to have a large LUT if the corresponding amplitude resolution provided is greater than the resolution of the DAC generating the analog output.

Digital to Analog Converter (DAC)

There are many different types of DACs. All DACs are similar in that a DAC accepts a digital word as input and outputs an analog waveform. The digital word is used to represent a sample of the output waveform, specifying its magnitude. Since the digital input word specifies a discrete magnitude that is held constant for one clock cycle, the output of a DAC has a step-like response function.

Some key characteristic of all DAC's are settling time, glitch energy (switching transients), linearity, and precision. Settling time specifies how long it takes the DAC to achieve optimum performance after powering on. Glitch energy specifies how much energy is in the high frequency transients caused by switching on the outputs. Linearity describes the how uniform the output step sizes are between adjacent digital words. Precision is a measure of how closely the DAC is able to make the actual output match the theoretical output for a given digital input word.

Reconstruction (Interpolation) Filter

Interpolation filters come in many forms: active, passive, and electro-mechanical. All interpolation filters are used for one reason, to "smooth out" the step-like response of the DAC output. The interpolation filter can be either a lowpass or bandpass filter depending on the undesired frequencies introduced by the non-ideal sampling. When the interpolation filter "smoothes out" the DAC output, it is eliminating high frequency harmonics of the signal of interest. It can also be used to eliminate the high frequency glitch energy and switching transients of the DAC.

Filters are a complex design choice. Factors such as phase linearity, frequency selectivity, and insertion loss are just a couple of the considerations when choosing a filter. Filters also have noise figures, varying attenuation capabilities in the stopbands, varying transition bandwidth, and passband ripple characteristics. All these different characteristics make such a discussion outside the scope of this discussion.[1]

Sources of Error in Waveform Generation

There are several sources of error for a DDS Waveform Generation system. They include: phase and frequency error introduced by the sampling clock, glitch energy in the DAC, phase truncation in the Phase Accumulator, amplitude truncation in the LUT, the linearity of the DAC, and the linearity of the Reconstruction Filter. All these sources of error are unavoidable in real systems.

[1] For a complete discussion of filtering considerations refer to Horn, 1992 – (16)

The phase noise in the output introduced by the sampling clock is mitigated to some extent by the high sampling rates used by DDS's to construct waveforms. The phase noise improvement in the output waveform, in comparison to the phase noise of the clock source, will be (18):

$$\text{Phase Noise Improvement} = 20*\log_{10}\left(\frac{F_{clk}}{F_{out}}\right). \qquad [6]$$

The frequency noise introduced by the sampling clock is passed directly through the DDS system. Any frequency error seen in the clock changes the sampling rate of the DDS, moving the frequency response of the DDS output. A system requiring extremely accurate frequency representation should have a high stability clock as its reference. The DDS, however, can be "tuned" to any crystal used and the frequency error of the crystal can be compensated for in the DDS if necessary.

The phase truncation can be controlled by the number of bits used in the phase accumulator of the DDS. The more bits used in the phase accumulator the smaller the phase truncation error. In custom designs of DDS systems, the phase accumulator width can be controlled. If an off-the-shelf part is used, care must be taken to ensure proper width of the phase accumulator.

Amplitude truncation is a two fold problem. The first place where it can occur is in the LUT of the DDS. Amplitude truncation can be reduced by expanding the depth and/or width of the LUT memory. The second source of error in amplitude truncation is the DAC. The limiting factor in system performance will be the device, either the LUT or the DAC, which has the shortest digital word. In designing a system, care should be taken to properly match the LUT width and DAC width to achieve optimum performance.

The noise introduced by amplitude truncation is a function of the length of the digital word used to represent a sample's amplitude. The noise can be modeled statistically and is dependent on the signal variance and the full scale level of the DAC. The SNR of a signal with a given precision in amplitude is (13)

$$SNR = 10\log_{10}\left(\frac{\sigma_S^2}{\sigma_N^2}\right)$$

$$SNR = 10\log_{10}\left(\frac{12*2^{2B}\sigma_S^{\,2}}{X_M^2}\right)$$

$$SNR(dB) = 6.02B + 10.8 - 20\log_{10}\left(\frac{X_M}{\sigma_S}\right) \qquad [7]$$

where σ_S is the variance of the desired signal, σ_N is the variance of the noise, X_M is the full scale level of the DAC, and B is the number of non-sign bits in a two's complement number. There is an assumption made that the DAC's input is a two's complement number. Note the last term in Equation 7. It shows that the SNR is dependent on the relationship of the rms value of the signal amplitude with the full scale level of the DAC. It is important, therefore, to match the signal level with the full scale output level of the DAC. For full-scale sinusoidal signals the SNR can be reduced to (5),

$$SNR = (6.02B + 1.76)dB \qquad [8]$$

Several assumption are made in the following analysis. It is assumed that the error introduced by amplitude truncation is a stationary process, the error is uncorrelated with the desired output, and the error

has a uniform distribution over the range of quantization error. These assumptions are not always valid when generating fixed frequency signals, but the analysis is simple and yields worst case noise floor performance of the system. As the error becomes more correlated to the desired signal, spurs will begin to rise out of the noise floor, but the noise floor will start to drop at the same time. The basic tradeoff is that noise energy is transferred from the broadband noise into spurious noise. If the spurs produced can be filtered, concentrating noise energy in the spurs can be beneficial since the filter will remove the spurs and the noise floor has been reduced.

Summary of Direct Digital Synthesis Waveform Generation

In summary, the FRC is used to latch PI information and synchronize it with the rest of the DDS. The DDS accumulates phase to generate frequencies of interest. Frequency and phase changes are made by adjusting the amount of phase accumulated on each clock cycle. If a constant Phase Increment value is left in the Frequency Control Register, the DDS will generate a constant frequency sinusoid. The phase and frequency resolution of the DDS is dependent upon the size of the digital word used by the Phase Accumulator. The LUT uses only the MSB of the phase accumulator word to generate amplitude values corresponding to specific phase values. The output of the LUT is sent to a DAC. The DAC translates the digital word at its inputs to an analog value at its output. The output of the DAC has a step-like response that the Interpolation Filter will “smooth out”. Given the proper design considerations, the output of the Interpolation Filter will be an excellent representation of the desired signal of interest.

2. PulseBlasterDDS Specifics

This chip has four frequency registers with each register 32 bits wide. The frequency register to use is controlled by control line generated by the PulseBlaster PP core.

3. Calculating DDS Frequency

The calculation of the frequency word for the DDS units is relatively straight forward.

$$\text{DDS frequency word (32 bits)} = \frac{\text{Desired Freq (in MHz)} * 2^{32}}{\text{Clock Freq (in MHz)}}$$

The calculated value above can be used when programming the DDS units.

The frequency resolution of the default design is:

$$50e6 / 2^{32} = 0.012 \text{ Hz}$$

Appendix I: Programming Notes

Programming Notes for PulseBlasterDDS Ultra:

There are two methods of programming the PulseBlasterDDS Ultra. You can either use the included C functions to create the program for you by putting your pulse program in echo2.c and recompiling it, or you can generate the appropriate outputs directly to the device using the _outp command.

Method I. Using C functions

In order to program the board using a c program, you must first initialize the board by using the following functions. All of these steps are taken in the main routine of echo2.c. You can replace the pulse sequence lines in echo2.c to skip all of the initialization steps.

void* grab_image_memory(UINT32 length)

Used to allocate memory on the PC equal to the size of the memory on the PulseBlasterDDS. The memory on the PC will be programmed fully before being downloaded to the PulseBlasterDDS.

length – Always use INTERNAL_MEMORY

DDS_Unit* grab_dds_memory(void);

Used to allocate memory for the programming of the DDS registers.

HW_Parameters InitHW(UINT16 myPort, float clock_frequency, UINT32 memory_type);

Initializes hardware specific variables for the programming of the PulseBlasterDDS.

myPort – the base port address (e.g. 0x340)

clock_frequency – the clock frequency of the PulseBlasterDDS in MHz

memory_type – Always use INTERNAL_MEMORY

After initializing the hardware specific variables, you must then set the initial values of the TTL outputs and the values of the frequency registers.

INT8 program_dds(UINT8 dds, UINT8 reg, double frequency, DDS_Unit *dds_array);

Used to specify the values of each of the four frequency registers.

dds – specifies which dds unit you would like to program (Always set to 1)

reg - specifies which frequency register you would like to program (Valid range of values is 0-3)

frequency – specifies frequency you wish to program the register with (in MHz)

*dds_array – variable of type DDS_Unit to hold the information until it is to be written to the board. (Should be the same variable returned from grab_dds_memory)

void program_initial_flag_values(UINT32 flags);

Used to set the initial state of the TTL outputs.

flags – holds hex value of information to be output to the TTL outputs (only bits 9..0 are valid outputs)

After initializing the flags and frequency registers, you then need to generate your pulse program. This can be accomplished by using the following sequence of commands.

output_control_word = FREQ0 | TX_DDS_GATE_OFF | RX_DDS_GATE_OFF | 0xFFFF;

The first part of the output control word is used to specify the frequency to be used. Valid values are FREQ0, FREQ1, FREQ2, and FREQ3

The second part specifies whether the first DAC output is on or off. Valid values are TX_DDS_GATE_OFF and TX_DDS_GATE_ON

The third part specifies whether the second DAC output is on or off. Valid values are RX_DDS_GATE_OFF and RX_DDS_GATE_ON

The fourth part of the control word specifies what the TTL outputs should read during this pulse (Only bits 9..0 are valid)

Once the output control word has been generated, you must call a function that uses this information to generate the actual 80-bit instruction word.

ADDRESS delay(double time, UINT32 flags, Instruction_Unit *image);

Used to generate the specified pulse for the specified amount of time. By calling this function the instruction is automatically added to the copy of the internal memory pointed to by the *image variable.

time – length of pulse (in ns)

flags – this should be output_control_word as generated above

*image – this should be the variable returned from grab_image_memory;

These variables have the same meaning for each of the program functions;

ADDRESS stop(double time, UINT32 flags, Instruction_Unit *image);

Stops execution of pulse program.

ADDRESS loop(double time, UINT32 loop_count, UINT32 flags, Instruction_Unit *image)

Beginning of a loop. Loops this portion of the program for loop_count times

ADDRESS end_loop(double time, UINT32 top_of_loop, UINT32 flags, Instruction_Unit *image)

End of a loop. Returns to command specified by top_of_loop.

ADDRESS jump(double time, UINT32 next_addr, UINT32 flags, Instruction_Unit *image)

Jumps to a subroutine whose address is specified by next_addr

ADDRESS rts(double time, UINT32 flags, Instruction_Unit *image)

Returns from a subroutine

ADDRESS branch(double time, UINT32 next_addr, UINT32 flags, Instruction_Unit *image)

Unconditionally branches to next_addr

After creation of the pulse program, you must create the text file containing the programming information. The file will be used when actually programming the board.

INT16 create_bytecode_file_dds(Instruction_Unit *image, UINT32 image_size, DDS_Unit *dds, char *name, HW_Parameters hw);

*image – Memory image returned from grab_image_memory.

image_size – always use INTERNAL_MEMORY

*dds – DDS image returned from grab_dds_memory.

*name – name of text file to be created

hw – use variable returned by InitHW

After writing all relevant information to a file, call the following function to program the PulseBlasterDDS from this file.

INT16 program_pulseblaster(char *name);

*name – name of the text file as specified in create_bytecode_file_dds() function

Once the board has been programmed, call start_pb() when you are ready for the pulse program to run.

Other useful functions are listed below

void reset_pb(UINT16 port);

Used to stop the execution of the pulse program.

void restart_pb(UINT16 port);

Used to restart execution of the pulse program after a stop order has been issued.

Method II. Writing directly to the output port

The following is an example of the output sequence to program the PulseBlasterDDS board. Explanations are included in brackets in the middle of the code.

```
[ Initialization: ]

Output "0x00" to port base + 0 (Issue device reset)
Output "0x04" to port base + 2 (Select number of bytes per
word)
Output "0xFF" to port base + 3 (Select device to program (Flag
initial values))
Output "0x00" to port base + 4 (Reset address counter)

[ Set initial flag values ]
[ value for this example are "0x00aaf0f0" ]

Output "0x00" to port base + 6 (Data transfer)
Output "0xAA" to port base + 6 (Data transfer)
Output "0xF0" to port base + 6 (Data transfer)
Output "0xF0" to port base + 6 (Data transfer)

Output "0x00" to port base + 5 (Clock data into external
buffer)
Output "0x00" to port base + 5 (Return clock signal to low)

[ Set up DDS frequency registers ]

Output "0x00" to port base + 0 (Issue device reset)
Output "0x04" to port base + 2 (Select number of bytes per
word)
Output "0x01" to port base + 3 (Select device to program (DDS
Frequency Registers))
Output "0x00" to port base + 4 (Reset address counter)

[ DDS Register Values  ]
[ Reg0 = 051EB852 (1 MHz) ]
[ Reg1 = 0A3D70A4 (2 MHz) ]
[ Reg2 = 0F5C28F6 (3 MHz) ]
[ Reg3 = 147AE148 (4 MHz) ]

[ Formula for finding these values: ]

[ REG0 = DESIRED_FREQUENCY * 2^32 / PBDDS_CLOCK   ]
[  = 1 MHz * 2^32 / 50 MHz = 858993459.2 = 0x051EB852 ]
```

```
Output "0x05" to port base + 6 (Data Transfer - Byte 3 of
Reg0)
Output "0x1E" to port base + 6 (Data Transfer - Byte 2 of
Reg0)
Output "0xB8" to port base + 6 (Data Transfer - Byte 1 of
Reg0)
Output "0x52" to port base + 6 (Data Transfer - Byte 0 of
Reg0)

Output "0x05" to port base + 6 (Data Transfer - Byte 3 of
Reg1)
Output "0x3D" to port base + 6 (Data Transfer - Byte 2 of
Reg1)
Output "0x70" to port base + 6 (Data Transfer - Byte 1 of
Reg1)
Output "0xA4" to port base + 6 (Data Transfer - Byte 0 of
Reg1)

Output "0x0F" to port base + 6 (Data Transfer - Byte 3 of
Reg2)
Output "0x5C" to port base + 6 (Data Transfer - Byte 2 of
Reg2)
Output "0x28" to port base + 6 (Data Transfer - Byte 1 of
Reg2)
Output "0xF6" to port base + 6 (Data Transfer - Byte 0 of
Reg2)

Output "0x14" to port base + 6 (Data Transfer - Byte 3 of
Reg3)
Output "0x7A" to port base + 6 (Data Transfer - Byte 2 of
Reg3)
Output "0xE1" to port base + 6 (Data Transfer - Byte 1 of
Reg3)
Output "0x48" to port base + 6 (Data Transfer - Byte 0 of
Reg3)

[ Pulse Program Setup ]

Output "0x00" to port base + 0 (Issue Device Reset)
Output "0x0A" to port base + 2 (Select number of bytes per
word)
Output "0x00" to port base + 3 (Select device to program
(RAM))
Output "0x00" to port base + 4 (Reset address counter)
```

```
Output "0x18" to port base + 6 (Byte 9 of first instruction)
Output "0xFF" to port base + 6 (Byte 8 of first instruction)
Output "0xFF" to port base + 6 (Byte 7 of first instruction)
Output "0x00" to port base + 6 (Byte 6 of first instruction)
Output "0x00" to port base + 6 (Byte 5 of first instruction)
Output "0x00" to port base + 6 (Byte 4 of first instruction)
Output "0x00" to port base + 6 (Byte 3 of first instruction)
Output "0x00" to port base + 6 (Byte 2 of first instruction)
Output "0x00" to port base + 6 (Byte 1 of first instruction)
Output "0x07" to port base + 6 (Byte 0 of first instruction)

Output "0xXX" to port base + 6 (Byte 9 of second instruction)

[    Continue this process for all instructions.  The
     explanation of ]
[ how to create the 80 bit instruction words is included
below. ]
[ When finished with all instructions, continue with the
sequence ]
[ below.            ]

Output "0x00" to port base + 7 (Programming Finished)


[    Only execute the following command when you are ready for
     the ]
[    program to start running.        ]

Output "0x00" to port base + 1 (Start pulse program)
```

Breakdown of 80 bit instruction word

Instruction Bits 79...0 are broken up into 4 sections

2. Output Pattern - 24 bits (Instruction Bits 79..56)
3. Data Field - 20 bits (Instruction Bits 55..36)
4. OP Code - 4 bits -(Instruction Bits 35..32)
5. Delay Count - 32 bits - (Instruction Bits 31..0)

Output Pattern:

Instruction Bit #	Function
79	Selects one of four frequency registers (MSb)
78	Selects one of four frequency registers (LSb)
77	None
76	Turns on/off output of DDS #1
75	Turns on/off output of DDS #2

74	None
73	None
72	None
71	None
70	None
69	None
68	None
67	None
66	None
65	Output Connector DB25 pin 19
64	Output Connector DB25 pin 7
63	Output Connector DB25 pin 8
62	Output Connector DB25 pin 21
61	Output Connector DB25 pin 22
60	Output Connector DB25 pin 10
59	Output Connector DB25 pin 11
58	Output Connector DB25 pin 24
57	Output Connector DB25 pin 25
56	Output Connector DB25 pin 13

Data Field (Bits 55 – 36) and Op Code (Bits 35 – 32):
The data field's function is dependent on the OpCode.

OpCode #	OpCode Meaning	Data Field used for
0	Continue	Ignored
1	Stop	Ignored
2	Loop	Number of desired Loops - 1
3	End Loop	Address of Instruction originating loop
4	Jump to Subroutine	Address of first subroutine instruction
5	Return From Subroutine	Ignored
6	Branch	Address of next instruction
7	Long Delay	Number of desired loops - 2
8	Wait	Ignored

Delay Count:
How long the current instruction should be executed. The smallest possible delay is 6 clock cycles (120ns). The formula for determining its value is below

DELAY_VALUE = (Desired Delay (ns) – 60 ns) / 20 ns

Ex. for delay of 1000 ns

= (1000ns – 60ns) / 20ns = 940ns / 20ns
= 47
= 0x2F

Appendix II: Sample C program (echo2.c)

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<conio.h>

#include<math.h>

#include<malloc.h>

#include "spincore.h"


/*  SPINCORE TECHNOLOGIES, INC

    GAINESVILLE, FL

    www.spincore.com



    This file is being distributed as community software free
of charge.  SpinCore

    Technologies, Inc retains ownership of the software but
does not make any

    claims as to its functionaliy or warranty it in any way.
Any modifications

    to this code must be provide to SpinCore along with a
description of what

    was changed as well as the motivation behind the changes.
The modifications
```

```
    will be reviewed by SpinCore and made availible to others
if the changes are

    deemed positive by SpinCore.


    This is an beta release of code.  As such

    there may be bugs in the code. */



//  MAIN PROGRAM  //

void main(int argc, char *argv[])

{

   UINT16 port_addr = 0x340;

   UINT32 temp;

   UINT32 output_control_word;

   INT16 temp2;

   Instruction_Unit *program;

   DDS_Unit *dds;

   HW_Parameters hw_spec;

   ADDRESS address;

   UINT8 unit, reg, i, j, *dds_word;

   double freq;


   if(argc != 1 && argc != 2)

     {

       printf("\nCommand line ERROR.\n");
```

```
       printf("Command line usage is \"driver filename\"\n");

       exit(0);

     }


   if(argc == 2)
   {
    if(strcmp(argv[1],"-stop")==0){  /* command line option to
issue stop */

   reset_pb(port_addr);           /* reset command to the NMR
Controller */

   printf("\nStop Command has been issued to NMR
Controller.\n");

   exit(0);

  }


  if(strcmp(argv[1],"-start")==0){ /* command line option to
issue start */

   arm_pb(port_addr);

   start_pb(port_addr);               /* start command to the
NMR Controller */

   printf("\nStart Command has been issued to NMR
Controller.\n");

   exit(0);

  }
   }



   /* Initialize Code */

   program = grab_image_memory(INTERNAL_MEMORY);

   dds = grab_dds_memory();
```

```
   hw_spec = InitHW(port_addr,50,INTERNAL_MEMORY);


   /* Program DDS frequency registers */

   program_dds(1, 0, 1, dds);

   program_dds(1, 1, 2, dds);

   program_dds(1, 2, 3, dds);

   program_dds(1, 3, 4, dds);


   program_initial_flag_values(0x00AAF0F0);


   /* Create program for Pulse Programmer Core;

   Label definitions are in spincore.h

   (you may define your own labels as well)

 */

// begin your program here

   output_control_word = PHASE1 | FREQ0 | TX_DDS_GATE_OFF |
RX_DDS_GATE_OFF | 0xFFFF;

   address = delay(0.2*us, output_control_word,program);  //
synchronization state


   output_control_word = PHASE1 | FREQ0 | TX_DDS_GATE_ON |
RX_DDS_GATE_ON | 0x0000;

 address = delay(5*us, output_control_word,program);  // first
pulse

  output_control_word = PHASE1 | FREQ0 | TX_DDS_GATE_OFF |
RX_DDS_GATE_OFF | 0x0000;

 address = delay(100*us, output_control_word,program);  //
first delay
```

```
  output_control_word = PHASE1 | FREQ1 | TX_DDS_GATE_ON |
RX_DDS_GATE_ON | 0x0000;

 address = delay(2.5*us, output_control_word,program);  //
second pulse

  output_control_word = PHASE1 | FREQ1 | TX_DDS_GATE_OFF |
RX_DDS_GATE_OFF | 0x0000;

 address = delay(100*us, output_control_word,program);  //
second delay

  output_control_word = PHASE1 | FREQ2 | TX_DDS_GATE_ON |
RX_DDS_GATE_ON | 0x0000;

 address = delay(1.5*us, output_control_word,program);  //
third pulse

  output_control_word = PHASE1 | FREQ2 | TX_DDS_GATE_OFF |
RX_DDS_GATE_OFF | 0x0000;

 address = delay(100*us, output_control_word,program);  //
third delay

  output_control_word = PHASE1 | FREQ3 | TX_DDS_GATE_ON |
RX_DDS_GATE_ON | 0x0000;

 address = delay(1.25*us, output_control_word,program);  //
fourth pulse


   output_control_word = PHASE0 | FREQ0 | TX_DDS_GATE_OFF |
RX_DDS_GATE_OFF | 0x0000;

   address = branch(1.6*ms,0,output_control_word,program);
//repetition delay, loops back)


   output_control_word = PHASE2 | FREQ3 | TX_DDS_GATE_OFF |
RX_DDS_GATE_OFF | 0xFFFF;

   address = stop(500*ns,output_control_word,program);  //
never executed, if all OK


   /* Create file to be used by driver function */
```

```
   /* This file can be very useful in  debugging errors since
it allows

      the programmer to see what is actually being programmed
to the

      board. */

   if (argc == 1)
   {
    temp2 = create_bytecode_file_dds(program, address, dds,
DEFAULT_OUTPUT_FILE, hw_spec);
   }
   else
   {
    temp2 = create_bytecode_file_dds(program, address, dds,
argv[1], hw_spec);
   }

   if(temp2 != 0){

     printf("Error create bytecode file, error code
%d\n",temp2);

     exit(1);

   }



   /* Program PulseBlaster/PulseBlasterDDS series board with
info

      contained in file created by function
create_bytecode_file */

   if (argc == 1)
   {
    temp2 = program_pulseblaster(DEFAULT_OUTPUT_FILE);
   }
   else
   {
    temp2 = program_pulseblaster(argv[1]);
   }

   if(temp2 != 0){
```

```
      printf("Error programming board, error code %d\n",temp2);

      exit(1);

   }


   /* start execution of program */

   start_pb(port_addr);          /* command to start
PulseBlaster */

}
```