Owner's Manual for the

PulseBlaster™

Intelligent General Purpose

Pulse/Pattern/RF Generation Board

SpinCore Technologies, Inc.
4623 NW 53rd 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.

Contents

Section I: Introduction

1. Quick Product Overview

We are proud to present PulseBlaster™ - our Intelligent Pulse Programmer-Pattern Generator Board for NMR/MRI/NQR/ICR/Solids and related resonance and test applications. The entire logic design (excluding output drivers) is contained in a single silicon chip, qualifying it as a system-on-a-chip (SOC) design – the industry's first and only design of this kind. Implemented on a 0.22 micron, state-of-the-art programmable silicon, its innovative Very-Long Instruction Word (VLIW) design has many unique and attractive features such as high speed, small size, low power consumption, increased functionality, and ease of programming.

Output signals

PulseBlaster™'s standard model has 24 individually controlled output bits that are capable of delivering +-25 mA per bit. The outputs can be set to either the 5 V or 3.3V I/O logic standard. Output signals are available both on the PC bracket-mounted DB-25 connector and on an internal, flat-cable header.

Timing Characteristics

PulseBlaster™'s 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 controller has a very short minimum delay cycle – only five clock periods. This translates to a 50 ns pulse/delay with a 100 MHz clock. It is so fast that the board can actually be used to generate rf pulses – the

20 MHz (max) update rate can be used for IF testing of some systems, and even as the rf Larmor frequency of many low-field systems!

Instruction Set

PulseBlaster™'s 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.

Optional Memory Chip

The internal, on-chip, memory space for the system-on-a-chip design is 512 80-bit words. In addition, the board features an optional memory chip to allow for execution of much larger programs. The optional memory chip provides 32k words of program space - a number that is much larger than necessary for even the most complex pulse programs today. The trade-off for using the additional on-board memory is that the minimum delay increases from five to seven clock periods.

External triggering and cascading

PulseBlaster™ 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). The design also allows for multiple boards to be synchronized, easily extending the 24-bit output pattern to 48 bits or more. If a larger number of output bits is necessary, e.g., 64, 128 or 256, SpinCore can customize the design to suit the individual needs. Please contact SpinCore for details.

Summary

PulseBlaster™ is a versatile, high-performance pulse/pattern generator operating at speeds of up to 100 MHz and capable of generating delays 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 24 high-current output bits are independently controlled and 5/3.3 V user-selectable.

2. Quick Installation Guide

PulseBlaster™ 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.

PulseBlaster™ 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
Memory select:	Internal, on-chip.

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

No software or drivers of any kind are required to *install* the board. 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. 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/Windows 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/ serves as a repository of the software described in this manual.

Section II: PulseBlaster™ Design

1. PulseBlaster™ Design Overview

The PulseBlaster™ device is an intelligent pulse/pattern generation unit. The intelligence of the PulseBlaster™ comes from an embedded microprogrammed controller core (uPC). The uPC is able to execute instructions that allow it to control program flow. This means that the PulseBlaster™ understands Operational Control Codes, Op Codes, and will execute them much the same way a general-purpose microprocessor does. The PulseBlaster™'s microcontroller is different from the general-purpose microprocessor in that it does not contain an arithmetic logic unit (ALU) and is incapable of doing mathematical or logical calculations.

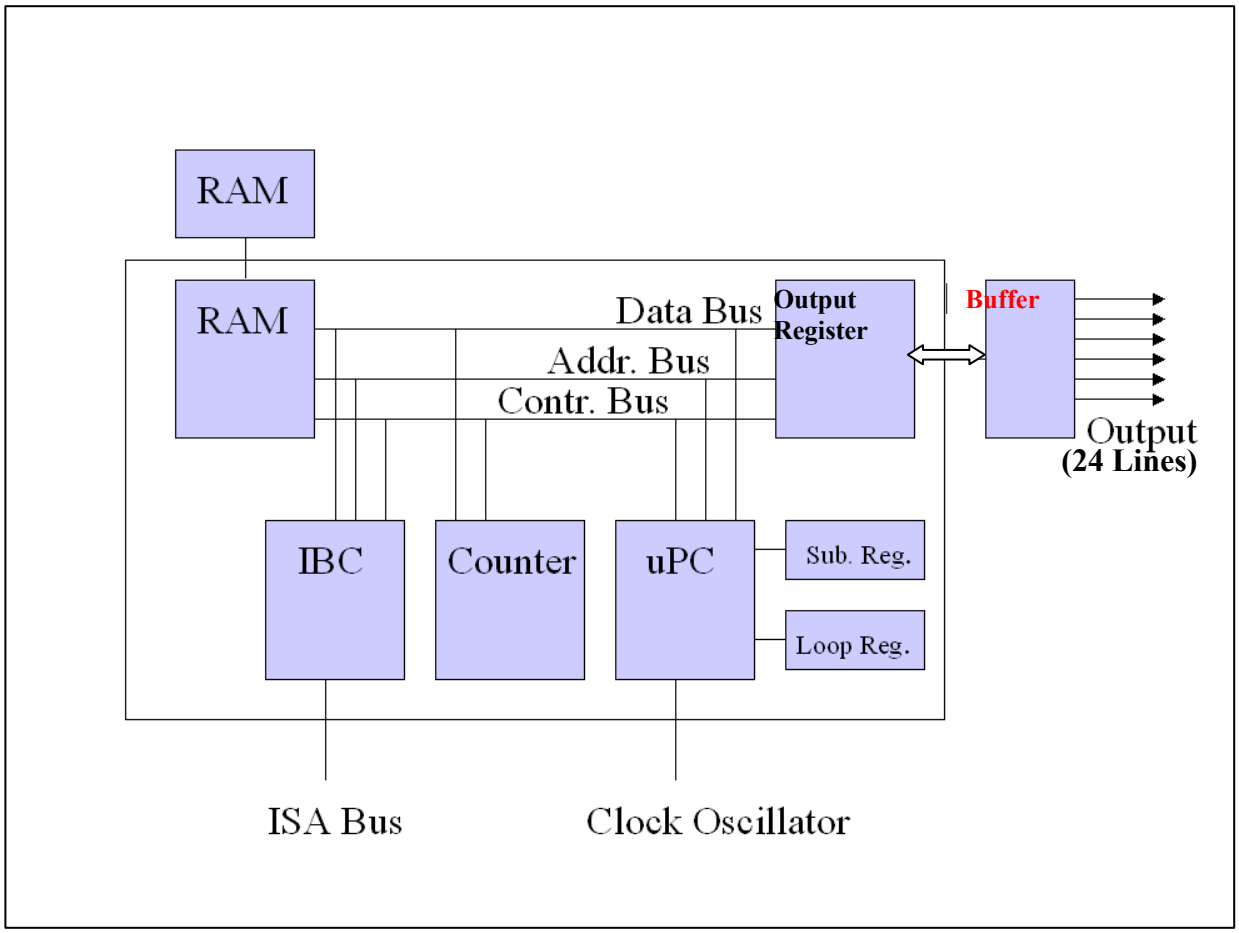

Figure 1. Block diagram of PulseBlaster™ design.

Figure 1 presents a block diagram of PulseBlaster™ processor and the board. The major blocks are the memory, both internal to the processor chip and external to the chip (optional, on board), the ISA Bus Controller, the Delay Counter, the Output Register, and the micro-programmed controller uPC. The clock oscillator and the output buffers are external to the processor. The entire board is in the PC-XT form factor.

The benefit of the uPC core is the increased code density achieved when a program outputs a repetitious pattern. A potential disadvantage of a uPC design is that it takes time for the uPC to make decisions on program flow and forces a longer minimum delay cycle time over some other types of designs. However, by carefully designing the uPC and retaining only the most critical Op Codes, the latency of the uPC can be kept to a minimum. In the PulseBlaster™ design, the minimum delay cycle latency is 5 system clock cycles. The maximum system clock speed is, currently, 100 MHz. Note: When external memory is used, the minimum delay cycle is 7 clock cycles long.

The uPC core uses two stacks[1] in order to control program flow. The first stack, the Subroutine Return Address Stack, is used to hold the return address of the currently executing subroutine. This stack is 16 addresses deep, implying that there can be no more than 16 embedded subroutine calls. This does not mean that there can be no more than 16 subroutine calls in a program. It means that the user may have at most 16 consecutive subroutine calls without ever returning from a subroutine. Every time a call to a subroutine is encountered, the next address in memory is pushed[2] onto this stack, and whenever a return from subroutine command is encountered by the uPC the return address is popped[3] off this stack.

The second stack used is the Loop Count Stack. This stack is used to hold the loop count for any loop command encountered by the uPC. This stack also has 16 locations and this means that no more than 16 embedded loops can occur at any given time. It should be noted that Long Delay instructions are implemented by using the loop control structure. For programs using very long delay values, the greatest number of embedded loops is 15, since the Long Delays instruction will use one location on the loop stack. Any long delay value that is not prime (and a couple of other special cases) can be implemented with the one instruction loop. All other rare cases must be implemented in a 2-instruction set.

Timing control is an integral part of pattern generation. The PulseBlaster™ features an embedded 32 bit delay counter. This counter provides the delay tracking capabilities needed and generates an I/O strobe that clocks data through the output latches at the correct time. Maximum delay times for a given clock rate be calculated by multiplying the clock period by 2^32. When using a Long Delay instruction the maximum delay value is the clock period multiplied by 2^52.

The minimum delay cycle for the design depends on whether internal or external memory is being used for program memory. The minimum delay possible when using internal memory is 5 clock cycles. There is an

[1] A stack is a first-in-last-out (FILO) buffer.
[2] Pushing an item onto the stack simply means that it is being added to the buffer.
[3] Popping an item from the stack simply means that it is being removed from the buffer.

increased latency of 2 clock cycles when using external memory. If the internal memory is used, the maximum number of instruction words is 512. When using extremely long delay values, the worst case minimum number of instruction words is around 256 (implying that every delay takes 2 instruction words). The penalty in minimum delay latency incurred by using external memory is offset by the depth that the memory provides for program space. Using the external memory provides the user with 32k (32 * 1024) words of program memory space. The external memory depth is required when long non-repeating patterns require generation.

The ISA Bus Controller, IBC, serves as an interface between a PC and the PulseBlaster™ Processor. It handles programming the system memory, arming the device trigger, and can generate trigger and reset signals for the board.

2. PulseBlaster™ Machine Language

Programming architecture and word definition

The PulseBlaster™ processor implements an 80-bit wide Very-Long Instruction Word (VLIW) architecture. The 80-bit word has specific bits/fields dedicated to specific purposes, and every word should be viewed as a single instruction of the micro-controller. 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 2 shows the instruction word's fields and bit definitions.

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 2. Bit definitions for instruction (machine) word of PulseBlaster™.

The individual fields, starting from the least-significant bit, are interpreted as follows:

Delay Count – the lowest order bits of the command word - 32 bits long. This value is specified in clock cycles. The internal counter used in the PulseBlaster design counts down from the value specified in this field down to zero. There are two basic rules that need to be observed when specifying the delay count in clock cycles.

1) The minimum delay value entered must be greater than or equal to 2 cycles; and
2) The delay value generated equals the number of clock cycles entered *PLUS* three cycles. The extra three cycles come from overhead in the timing control logic.

OP Code - is 4 bits long. The following Op codes are allowed:

binary	hex	Mnemonic
0000	0	Continue
0001	1	Stop
0010	2	Loop
0011	3	End Loop
0100	4	Jump SR (Jump to Subroutine)
0101	5	Return SR (Return from Subroutine
0110	6	Branch (or just Jump, unconditionally)
0111	7	Long Delay

Table 1. Operational Codes (OP Codes) for the PulseBlaster microcontroller

Most of the **Op Codes** are similar to instructions for general-purpose processors and have the same effect on program flow as in a general-purpose processor. In the following subsection, all PulseBlaster™ Op Codes will be explained in details.

The **CONTINUE** instruction has no effect on program flow. The next instruction executed will be the one immediately following the current instruction in memory. The basic effect of the CONTINUE command is to delay a specified amount of time and generate a particular output pattern.

The **STOP** command is used to tell the uPC that the end of a program has been reached and that it should remain in an idle state until it has been reset or received another trigger. This instruction does not take any data.

The **LOOP** command denotes the beginning of a loop structure in a program and forces the uPC to push the data field of the instruction word onto the Loop Stack the first time it is encountered in a program. The user does not have to define register locations for the loop count values because the uPC automatically allocates resources if they are available. If too many loops are nested together, the outside loop will not function correctly and the design fails. There is no checking in hardware for the error condition caused by pushing too many loop count values onto the loop stack. It is the user's (or program compiler's) responsibility to ensure that no more than 16 loops are ever nested together.

The **END LOOP** command forces the uPC to decrement the most recent loop count in the Loop Stack. It also redirects the program counter[4] to the top of the loop if the loop has not been completed. If the loop has been completed, the program continues to the next instruction in memory. The address of the top of the loop is specified as a data field associated with the END LOOP Op Code.

The **JUMP to SUBROUTINE** command forces the uPC to modify the program counter and redirects the execution of the next command to be at the location specified in the data field of the instruction word. This command also forces the uPC to push the return address (next address after the JUMP SR command) onto the Subroutine Return Address Stack.

[4] The program counter is a register that tracks the address of the next command to be executed.

Allocation of all resources for saving the return address is handled by the uPC and the user does not need to specify how resources should be allocated. The number of nested subroutines can not exceed 16 calls. If more the 16 calls are made in a row, the outer subroutine call(s) will return to address 0. No error checking is done in hardware to determine if the stack has overflowed. It is the user's responsibility to ensure that no more than 16 subroutines are active at the same time.

The **RETURN from SUBROUTINE** command instructs the uPC to pop the most recent return address off the Subroutine Return Address Stack and execute that address as the next program instruction. There is no data field associated with this command.

The **BRANCH** instruction changes the program counter's value. It specifies a new location in memory to start execution from on the next instruction. The location is specified as a data field associated with the BRANCH instruction. Note: this case is similar to the JUMP to SUBROUTINE command except no return address is pushed onto the stack.

The **LONG DELAY** instruction implements a true zero overhead loop instruction. This command will generate a single particular output pattern for long periods of time by looping and executing the same instruction many times. The delay period can be calculated by multiplying the instruction 'delay length' by the ('loop length' + 2). The pipelined nature of the uPC adds two to the loop length. It should also be noted that the minimum number of loops is three and the minimum value that can be loaded into the loop counter is one.

Data Field – is 20 bits wide. Its meaning depends on the OP Code, as follows:

- If the OP Code is BRANCH, the corresponding data field must contain the exact address of the instruction to be executed next, i.e., where the program jumps to. NOTE: the first line of the machine code has the address zero, i.e., binary 00000; the second line has the binary address 00001, etc.

- If the OP Code is LOOP, the corresponding data field entered must equal the number of loops desired *MINUS* one. Programming the Data Field with this value in conjunction with the LOOP Op Code provides the desired number of loops in the output.

- If the OP Code is END LOOP, the corresponding data field entered must be the exact address of the instruction where the originating LOOP instruction resides. NOTE: the first line of the machine code has the address zero, i.e., binary 00000; the second line has the binary address 00001, etc.

- If the OP Code is JUMP SR, the corresponding data field must contain the exact (Note as above) address of the instruction where the subroutine starts.

- If the OP Code is LONG DELAY, the corresponding Data Field must contain the (number of loops desired – 2). This instruction is a

zero-overhead loop, and its execution time equals the {delay} times the (Data_field+2).

- All other instructions ignore the Data Field.

Output Pattern - this field specifies the 24 output bits. The output pattern is *maintained for the entire duration of the* ***current*** *machine word,* as specified in the delay count field (plus three cycles).

3. PulseBlaster™ Control Commands

Table 2 lists the control commands that are used by a host computer in order to program and use the PulseBlaster™ Pulse/Pattern Generator Board over the ISA Bus. The explanation of the individual control codes follows. The control code values are specified in offsets from the board's Base Address. The program in Section III, Chapter 3 "Programming with C/C++: An Example" can be consulted for information on how to implement control commands.

Control Code	Function
0	Device Reset
1	Start Trigger
2	Load Number of Bytes per Word
3	Select Memory Device
4	Clear Address Counter
5	Not Currently Used
6	Load Memory
7	Programming Finished

Table 2. PulseBlaster™ Control Commands

DEVICE_RESET: This command allows the PC software to reset the PulseBlaster™. After reset, the pattern generator can not be re-armed until the IBC has been re-initialized. The **DEVICE_RESET** command does not have any data associated with it

START_TRIGGER: This command can be used to start execution of a machine program residing in PulseBlaster™'s memory. *NOTE: the PulseBlaster™ always starts program execution from address zero.* The **START_TRIGGER** signal is only accepted, however, after the board has been initialized. The **START_TRIGGER** command does not have any data associated with it.

LOAD_NUMBER_OF_BYTES_PER_WORD: This command is used to facilitate programing of the memory used by the PulseBlaster™. The ISA Bus

Controller (IBC) accomplishes this task by using the Bytes per Word Counter to keep track of memory device width. It can handle memory devices from 2 to 15 bytes wide. The counter is loaded with the data byte sent over the ISA Bus with the Control Code to load the counter. The data value used to load the counter is also saved in a control register to allow the Bytes per Word Counter to be reloaded automatically. The Bytes per Word Counter ensures the correct number of bytes are used to reconstruct the memory word. By using this implementation, memory widths of any practical size can be used without changing the firmware design. It also allows for programming internal memory of unusually large width to facilitate the use of VLIW architecture, a design approach that is used in the PulseBlaster™.

SELECT MEMORY DEVICE: This instruction is used to specify which memory device to write to, either the embedded or external memory. In the PulseBlaster™ design, writing a zero data byte with this instruction programs the internal memory of the PulseBlaster™. If a 'one' is sent in the associated data byte, the external memory is programmed.

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.

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. PulseBlaster™ Board Initialization

Initialization of the PulseBlaster™ 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 MEMORY 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 PulseBlaster™ 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. 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 Section III of this Manual, Chapter 3, "Programming with C/C++: An Example."

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 PulseBlaster™ 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 Master Clock Oscillator:

The **Clock Select** signal is used to specify where the clock signal for the system will be coming from. The PulseBlaster™ Board allows for use of either an on board clock or an externally supplied clock source. If the signal is left tied high, the on board clock will be selected, else if the Clock Select line is pulled low through a jumper to ground, the externally applied clock will be used. (Default Value is the on board Clock).

The external clock source should be connected to the on-board SMA5 connector. The connector is terminated with a 50 Ω resistor. The input signal must also have a DC bias and the lowest voltage generated by the clock must not be lower than ground. Driving the clock to negative voltages will damage the input pin on the microchip. The device can accept 5.0 V volt peak input signals.

Selecting external memory: **Header J12, Pins 1-2**

The internal, on-chip embedded memory can accommodate up to 512 machine words. To accommodate larger programs, up to 32k machine words, the board has provision for using an optional memory chip. If equipped, the external, on-board chip can be selected by removing the jumper across pins #1-2 on header J12. If a board is not equipped with external memory, these pins must be jumpered.

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 JP9

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	J10-2
Bit 16	JP9-2
Bit 17	JP9-4
Bit 18	JP9-6
Bit 19	JP9-8
Bit 20	JP9-10
Bit 21	JP9-12
Bit 22	JP9--14
Bit 23	JP9-16
Output Clock	J10-1
Running	J12-10
Stopped	J12-7
System Reset	J12-8

Table 5. Output bits and signals of the PulseBlaster™ 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 JP9. 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 JP9 connectors are connected to the ground.

Using external trigger/reset lines - Header J5, Pins #3 and #5.

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 #8 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 #10.** 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 #7,** 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 PulseBlaster™ board is presented in Figure 3.

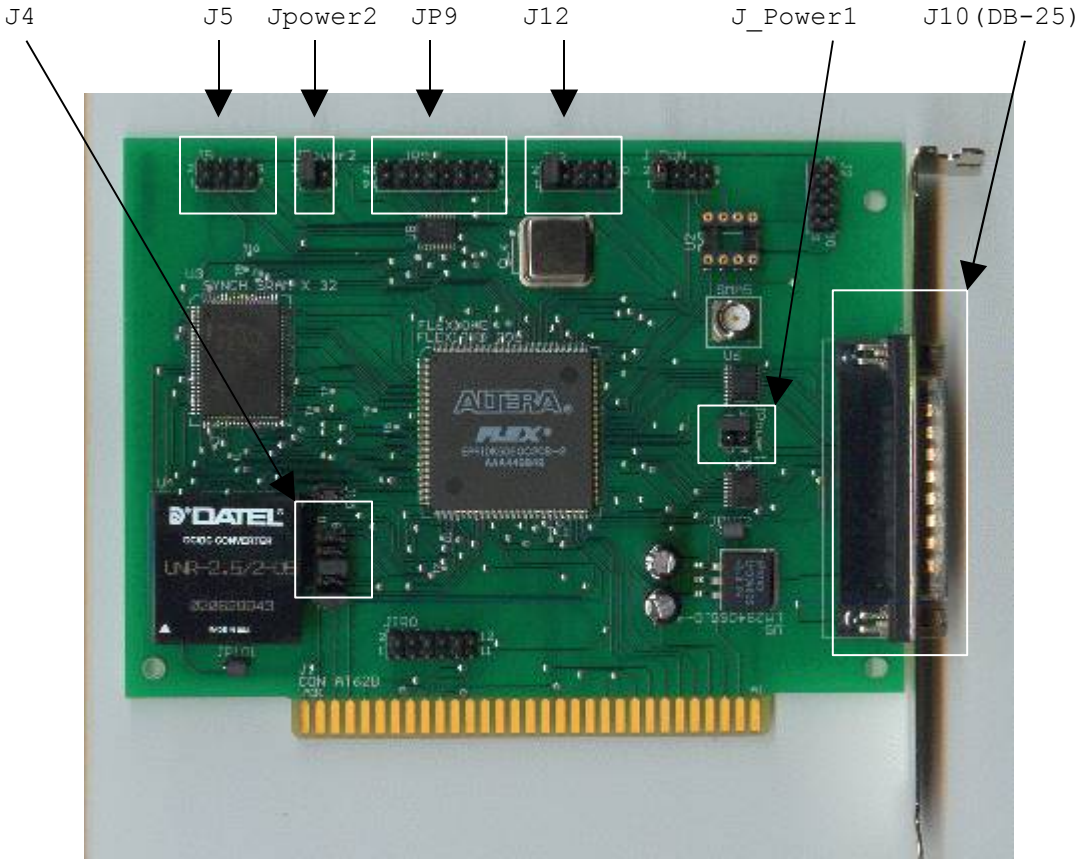

Figure 3. PulseBlaster™ Board – header/connector locations.

Section III: Test and Application Software

1. Test Board Loader and Controller

SpinCore has developed a simple program that can be used under Windows™ to quickly evaluate the PulseBlaster board under Windows95/98. The program, called NewISA, can load and execute several embedded test pulse programs. It can load any hex-coded pulse program from a text file. The screenshots below show the simple user interface to the board. The program can be launched from the command line or by double-clicking the corresponding icon.

Upon program's launch, the first screen prompts the user to enter the board's base address and clock frequency:

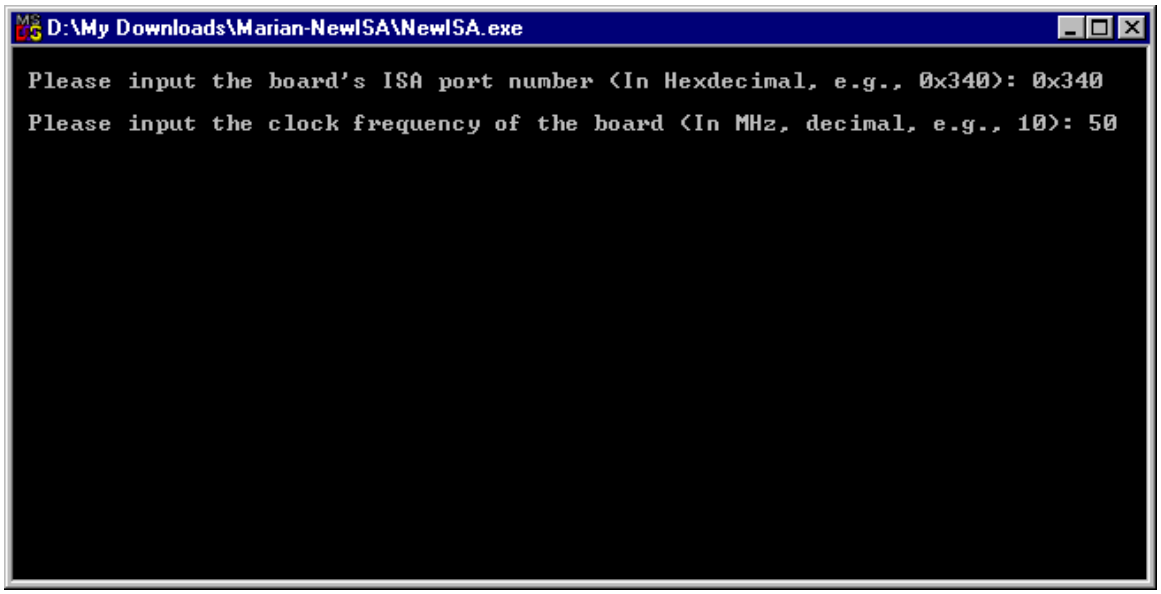

Figure 4. The initial screenshot of the simple test board loader and controller program called NewISA.

Upon entering the base-port address and the clock frequency, the program allows the user to execute several default pulse programs/waveforms. They include a 1 MHz waveform and a train of three 0.5 us pulses.

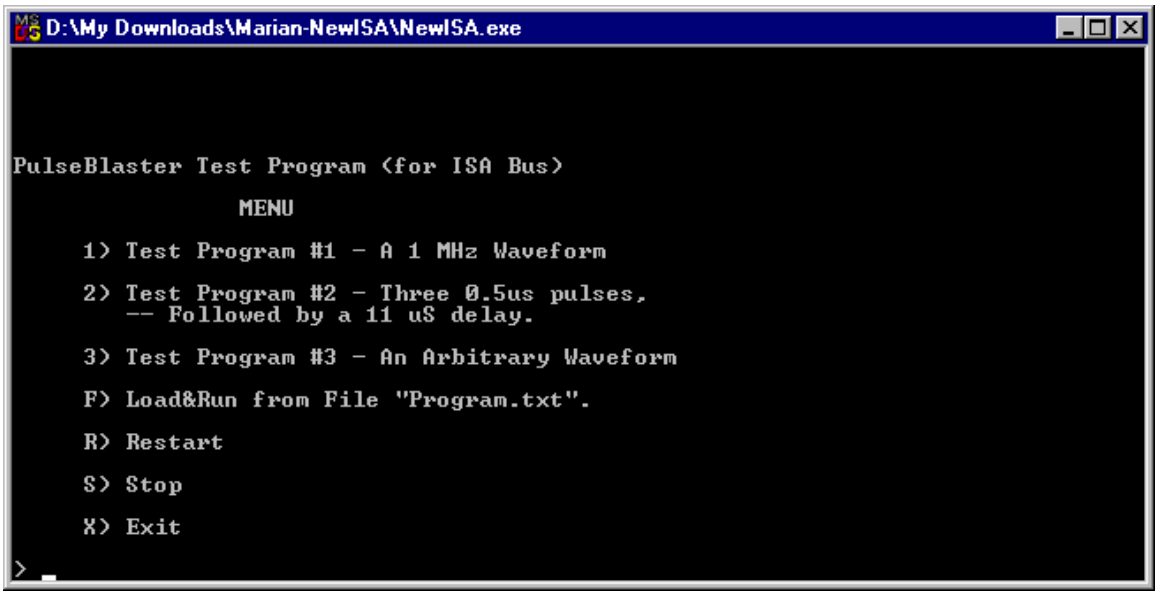

Figure 5. The main screenshot of the NewISA program for quick test and control of PulseBlaster™.

The pulse programs execute automatically upon selection. Any running pulse program can be stopped and restarted again with the S and R command, respectively. The NewISA program also allows the user to load and execute any hexadecimally-coded pulse program contained in the text file called (by default) Program.txt. An example Program.txt file could contain

0xffffff 0x000000 0x00000007
0x000000 0x000006 0x000000f1

This simple pulse program will execute a single pulse (the first line) followed by a delay (the second line). For more on machine programming, please see the next chapter "Understanding Machine Code: Examples."

2. Understanding Machine-Code: Examples

To use PulseBlaster™, no machine-code knowledge is necessary if one relies on application programs that take care of generating the appropriate byte streams. The test and application programs that are supplied by SpinCore and distributed with PulseBlaster™ can be used to generate nearly any pulse program that can be imagined, without knowing anything about Op Codes, Data Fields, word length, etc. However, knowledge of the machine-code programming is essential in writing a custom pulse program *compiler*.

To help understand the major machine-programming concepts, this chapter presents two simple machine-code programs that illustrate some of the information pertinent to machine programming. Those programs can be loaded and executed using the loader program described in the preceding chapter of this Manual, "Test Board Loader and Controller." They can also be loaded with the C/C++ program described in the next chapter.

Sample 1 – A simple two-line program in machine code

Just two machine instructions can generate a useful waveform - a single pulse followed by a delay. If the delays are identical in both instructions, a square waveform will result, as in the following code, with explanation:

```
Output | Data Field | OpCode | Delay

ffffff    00000        0      00000007  <- first instruction (80 bits)
000000    00000        6      00000007   <- second instruction (80 bits)
```

Delays: The number of clock "ticks" in both instructions is 0x00000007, decimal seven, resulting in the total duration of each instruction equal to ten clock cycles (three clock cycles are always inserted, automatically, to account for timer controller overhead). Thus, if the clock frequency is 10 MHz, each instruction will last 1 us, generating a 0.5 MHz waveform on all 24 outputs of the PulseBlaster™

Opcodes: The two Op Codes used are Continue (in the first instruction, the value equals 0x0) and Branch (second instruction, value 0x6).

Data Fields: The data field in the first instruction is irrelevant, since the associated Op Code is Continue (0x0). The data field in the second instruction is the address 0x00000, i.e., the address pointing to

the first instruction in the program (program lines are counted starting from 0, 1, 2, etc.).

Output bits: The 24 output bits are "all ones" during the first machine word (0xffffff) – they form the first half of the square wave (or the pulse), and "all zeros" (0x000000) during the second machine word.

Sample 2 – A simple three-line program in machine code

A simple three-line program can generate a number of useful patterns. For example, we will illustrate a sequence of 11 pulses followed by a delay, running continuously.

```
Output | Data Field | OpCode | Delay

ffffff    0000a        2      00000007  <- (start loop, loop 11 times)
000000    00000        3      00000007   <- (end the current loop)
000000    00000        6      00000030  <-  (go back to the beginning)
```

Delays: The first and second instructions: 7 clock "ticks," (plus three inserted) – they will correspond to a train of equidistant pulses (square wave-like). The third instruction will last longer, for 0x30 clock "ticks" (plus three inserted).

Op Codes: The first Op Code is LOOP (0x2), the Op Code in the second instruction is END_LOOP (0x3), and the Op Code in the third instruction is BRANCH (0x6). Thus, the program will loop first (lines one and two), then it will continue to line three, and then it will branch (jump) back to the first line.

Data Fields: The data field in the first instruction specifies the number of loops. The entered value is 0x0a, i.e., decimal 10. Thus, the program will execute 11 loops (this is because the accompanied loop counter counts down to 0). The data field in the second line is the top address of the current loop structure, i.e., where the corresponding LOOP command resides (0x00000). The data in the third instruction is the address where the program execution should continue next, i.e., the first line (0x00000).

Output bits: The 24 output bits are "all ones" during the first machine word (0xffffff) – they form the pulses that can be seen on an oscilloscope. The second line is "all zeros" (0x000000) – these will be the gaps between the pulses. The third line is "all zeros" again, for the duration of the last interval representing the delay between the groups of 11 pulses.

3. Programming with C/C++: An Example

The C source code below illustrates how the programming and loading of the board can be accomplished. It can be used to quickly test the PulseBlaster™ board. The program initiates the board, loads two 80-bit wide machine words, and starts the pulse program. NOTE: Under Linux, the _outp(address, value) commands have to be replaced with outb(value,address) pairs, and the program has to be run by root.

Example Code

```
//////////////////////////////////////////////////////////////////////
//
// This script can be used to test the pulse programmer board.
// First, it initiates the on-chip ISA Bus Controller.
// Then it programs the board's memory with two machine words.
// Afterwards, it issues one more control word - programming completed.
// And finally, it executes the start command.
// The resulting waveform should be a square wave.
//
// The script assumes that the board is at the address 0x340
//
// NOTE: Each machine word is 80-bit wide. Word organization:
//
// Output bits | Address/Loop counter | Op Code   | Delay
//  (24 bits)      (20 bits)             (4 bits)   (32 bits)

// Copyright 1999 SpinCore Technologies, Inc. http://www.spincore.com
//
//////////////////////////////////////////////////////////////////////
```

```
#include<stdio.h>
#include<conio.h>

#define port 0x340 // This is the default base address of the board

void main(){

 // Initialize the PulseBlaster™ board

 _outp(port,0);          // Device Reset
 _outp(port+2,0x0A);     // Load Number of Bytes per Word (10)
 _outp(port+3,0x00);    // Select Memory Device
 _outp(port+4,0x55);     // Clear Address Counter (value irrelevant)

 // Load two 80-bit-long machine word instructions

 // Instruction 1 - will last 4+3 clock ticks -------------------------
 _outp(port+6,0xff);// First eight bits of the output pattern
 _outp(port+6,0xff);// Output bits continue
 _outp(port+6,0xff); // Output bits - total of 24 bits - all "ones"

 _outp(port+6,0x00);  // First part of the data field
 _outp(port+6,0x00);  // value irrelevant for the Op Code Continue
 _outp(port+6,0x00);  // Op_Code - 4 LSBs is 0000 = Continue


 _outp(port+6,0x00); // First eight bits of the Delay Value
 _outp(port+6,0x00);// Delay Value bits continue
 _outp(port+6,0x00); // Delay Value bits continue
 _outp(port+6,0x04); // Delay Value - in clock ticks - total 32 bits


 // Instruction 2 - will last 4+3 clock ticks ---------------------------
 _outp(port+6,0x00);  // Output pattern
 _outp(port+6,0x00);  // Output pattern
 _outp(port+6,0x00); // Output pattern - total of 24 bits, all "zeros"

 _outp(port+6,0x00);
 _outp(port+6,0x00);  // Branch Address - points to first line, numbered 0
 _outp(port+6,0x06);  // Op_Code is 4 LSBs - will cause to program to jump


 _outp(port+6,0x00);
 _outp(port+6,0x00);// Delay Value
 _outp(port+6,0x00);
 _outp(port+6,0x04);

 // Signal the End of programming

 _outp (port+6, 0x00); //
 _outp(port+7,7); // Programming Finished

 // Put the device in run mode
 _outp(port+1,7); //Trigger - starts the pulse program execution

}
```

4. Pulse-Program Compiler in Java

Quick Product Overview

SpinCore Technologies has developed a simple pulse programming language where the commands are in the general form (delay, output_pattern). The accompanied Java-written program, **ISACompiler2.exe** reads, parses, and compiles a pulse program (text) file, generates machine code, and sends the data to the PulseBlaster™ board through the ISA bus. A sample program file, Sample.nmr is included with this distribution.

Program Flow Chart

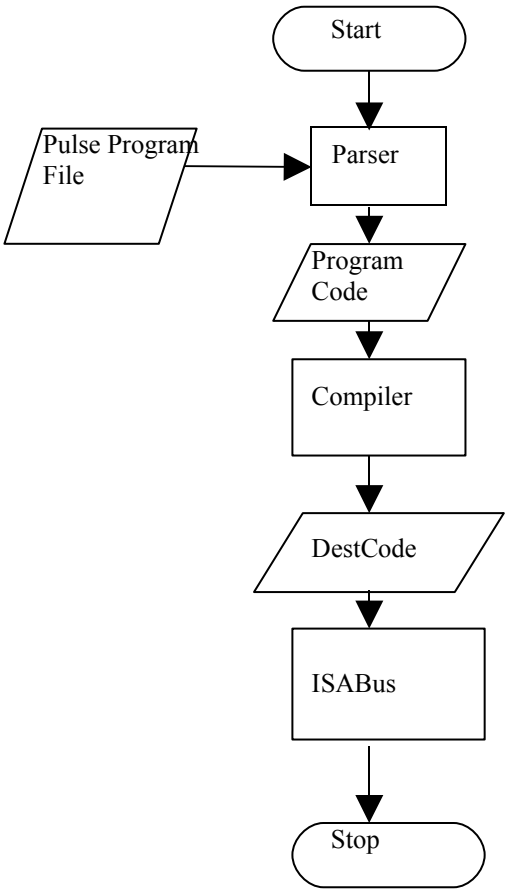

Figure 4. Flowchart of the Java Pulse Program Compiler

ProgramFile: A disk text file that stores the pulse program (e.g., Sample.nmr).

Parser: It reads in the program file, parses it, and stores the result in ProgramCode.

ProgramCode: A data structure (class) that stores the pulse program in a formatted way.
Compiler: It transfers pulse program into machine code of the board.
DestCode: A data structure (class) that stores the destination machine code.
ISABus: It sends machine code to the board through the ISA bus.

Native Windows Program

Because a typical Java program can not directly access hardware ports, we implemented the function of writing to a hardware port in the C language, and created a dynamic link library which can be called by the Java program through JNI. This associated library file is ISABus.dll.

Configuration File

In order for the program to work with boards configured with different ISA-bus port numbers and base clock frequencies, the ISACompiler2.exe program automatically reads the current ISA-bus port number and the base clock frequency from the disk file "Config.txt". A typical "Config.txt" contains:

```
configuration of ISACompiler2:

PortAddress 0x340 // ISA bus port number
BaseFrequency 10// in MHz, base frequency of the board
```

NOTE: If the "Config.txt" file does not exist, the program will ask user to input the ISA-bus port number and base clock frequency via keyboard every time the program is evoked.

Java Files

When the Java source code is compiled by javac of JDK, it generates appropriate classes, which will run on Java virtual machine. The main class is ISACompiler2.class.

Files:

```
ISACompiler2.class,
Comm.class,
Compiler. class,
PrgCode. class,
PrgCodeLine. class,
DestData. class,
DestDataLine. class,
ISABus. class,
ISACompiler. class,
MyException. class,
Parser. class
```

Usage

1. The JDK should be correctly installed on the machine.

2. Always put "ISABus.dll", "Config.txt" and all above ".class" files in the same directory.(e.g., C:\PulseBlaster\).

3. Go to the above directory where all files reside.

4. (a) To load and run from a program (text) file:

   At system prompt, on the command line,
   type "java ISACompiler2 ProgramFileName", which is case sensitive.
   The ProgramFileName is the name of the file you want to compile and run.

   (b) To stop running program:

   At command line, type "java ISACompiler2 -stop"

   (c) To restart a loaded program:

   At command line, type "java ISACompiler2 -restart"

Format of the Program File

```
----- Comments block 1: Compiler just ignores it. -------
This is a test header and comment section. I will use it to test
compiler; DECLARATIONS as it appears below is a keyword that
can not be used in the comment section

----- Delay Definitions: up to 10 delays can be defined as constants.
They can be referenced as D1, D3, … in the program below
They are expressed in microseconds, assuming a 10 MHz clock-------
```

Delay Declarations

```
D0 = 20 ;
D1 = 5 ;
D2 = 2.2 ;
D3 = 2.3 ;
D4 = 2.4 ;
D5 = 500 ;
D6 = 2.6 ;
D7 = 2.7 ;
D8 = 2.8 ;
D9 = 5000 ;

----- Comments block 2: Compiler just ignores it. -------
SECTION is also a keyword when spelled as below and can not be used in
a comment section Start and Stop commands for the controller are
implicit at the beginning and end of experiment.

Code will have following structure:
 "Delay Output;"
 OR "Opcode Address;"
```

```
----- Program. -------
                              Program Section

     D1       0xffffff;
     D5       0x000000;
     LOOP 4
      D2       0xffffff;
      D3       0x000000;
    ENDLOOP
    Branch 0;


                                End Program
```

5. Pulse Program Compiler in Perl

SpinCore Technologies provides an open source code compiler for their Intelligent Pattern Generator designs. The compiler is broken up into two sections, a front-end text-parsing engine, and a back-end, low-layer driver engine. The text parser is written in the Perl programming language and allows for the software to run on virtually any major operating system[5]. The low layer driver code is written in C and is OS dependent. SpinCore currently supports Linux and Windows95/98 operating systems. The driver code is used to send byte code information generated by the text parsing engine over the ISA bus to the embedded hardware.

Modification of the source code is permitted, provided that any modifications are submitted back to SpinCore to be made available to other users. Submission of code to SpinCore can be made through our website at www.spincore.com.

If your institution requires customization of the compiler please contact your SpinCore account manager for assistance.

Nuts and Bolts

The PulseBlaster™ system is programmed through the use of text files. The text files describe the program flow control as well as the operation of the output flags generated. The compiler can be invoked at the command prompt of the operating system used. The format of the command is "perl compile [programming file name]". 'Compile' is the name of the Perl script that implements the compiler. Note: It is possible to configure systems to execute the Perl script without the need to invoke the Perl command, but implementation of this feature is OS dependent. The method listed above works for all OS's

The compiler expects several formatting conventions. All delay variables must be created in a name space denoted by the prefix 'd_' and all the flag variables must be created in a name space denoted by the prefix 'f_'. Any alphanumeric text that follows the prefix is a valid name. There are no length limitations to the names so the names can be made descriptive and help to self-document the program. **All lines must end in a semicolon.** This is generally the most overlooked mistake and should be your first consideration when debugging a program. The compiler also recognizes comments, which are preceded by the standard C++ comment marker '//'. Any text after the comment marker to the end of the current line is ignored.

[5] For more information on Perl, please refer to www.perl.com or any one of the many reference books written on the subject.

There are several key words that the parsing engine looks for in addition to the name spaces for the variables. The key words are used for program flow control and are as follows:

1. Branch
2. Jump
3. RTS
4. Loop
5. End Loop

All keywords take an additional input field except RTS. The text string immediately following the Loop and End Loop commands is considered a label for that loop. It is used to identify which Loop and End Loop commands should be grouped together. The string following the Jump command instructs the compiler on what address to jump to and is associated with another label somewhere within the file. There are no limitations on how far a jump can be made. The RTS instruction returns from the current subroutine being performed by a previous call to Jump. Since, the return address from a subroutine call is pushed onto a stack, no label is needed to identify the return address. The most current entry in the stack must be from the subroutine just exited by the RTS. The maximum number of nested subroutine calls currently possible is hardware dependent. Please see previous sections for the subroutine stack limitations of your hardware.

Flag labels define groups of bits and assign them to a readable identifier. Each label is associated with a particular bit pattern. The labels are listed in the command line and concatenated to form an output word by joining the labels with the '+' sign. The leftmost label represents the MSB of the output word while the rightmost label represents the LSB. Each flag variable has two fields that must be specified. First, a bit pattern specified in hexadecimal, followed by a comma and the flag variable width. The width specifies the number of bits to be used when generating a number. This way 0,8 can be used to specify that eight bits should be set to zero. Bit definitions do not need to end on byte or nibble boundaries, either. A 3-bit definition is just as valid as an 8-bit definition.

The compiler also supports flag lists that are stored in a file. The format for using this functionality is as follows:

```
f_test  => [filename],7;.
```

The flag variable on the left provides a reference label to the file with the stored information. The '=>' symbol notifies the compiler that the flag variables values will be stored in a file. Every time the compiler comes across the label in a programming file, it reads in the next entry in the flag file and inserts its bit pattern in the programming file. Note that the number of bits that the flag value represents is still specified in the variable declaration and it will be checked for every value that is entered from the file.

The Program Compiler needs to know the clock frequency that is being used so that it can correctly generate the programming file. This is because some hardware models support externally supplied user clocks that can vary in frequency. Assignment of the clock frequency is provided with the following command format: "**Clock Frequency = 10 MHz;**".

The ISA Bus Address of the hardware design is also variable and must be specified to the compiler so that it can properly program the hardware.

The format for specifying the ISA Bus Address is as follows: "**ISA Card Address = 340;**". Notice that the port address is specified in hexadecimal.

The number of flags is also specified for the compiler since this can vary between designs. The flag number is used to ensure that no flag field has too many bits specified. The format of the instruction is as follows: "**Number of Flags = 24;**". The compiler will not complain if less than the specified number of flags are present. It will make the assumption that these lines were intended to be zero and will fix their values automatically.

It is also important to note that the compiler does no error checking for the number of embedded loops and subroutines. It is expected that the user ensure the maximum number of embedded subroutines and loops is never exceeded.

Writing a Program

When writing a program there are a few simple rules to follow and the rest is easy. All program instructions require a delay and flag output specification. Instruction lines that do not contain a delay and flag output specification are grouped with either the line directly after or before them. The outputs and delay for the Loop, Branch, and Jump commands are all specified on the line immediately following them. The End Loop and RTS commands are linked with the delay and outputs on the line specified immediately before them. Labels can be associated with program lines by placing them before a delay value. The labels must be in front of a delay specification line and not a program flow control command. All lines that are not associated with a special program flow instruction will be assigned a continue Op Code[6].

Loop Command Syntax
Loop [loop label] [number of time] – all fields required

End Loop Command Syntax
End Loop [loop label] – all fields required

Jump Command Syntax
Jump [subroutine label] – all fields required

RTS Command Syntax
RTS – no extra fields

Branch Command Syntax
Branch [branch to label] – all fields required

Delay and Output Specification Syntax
Label d_name f_name1 + f_name2 + ... – the label is optional and number of flag fields is optional

[6] Long Delay instructions are inserted as needed by the compiler for both program flow control (Jump, Branch, etc.) and Continue instructions.

Program Code Example

```
// This is a test header and comment section.
// DECLARATIONS as it appears below is a keyword that can not be
// used in the comment section.


// Units: Hz, kHz, MHz
Clock Frequency = 10 MHz;

Number of Flags = 24;

// Value specified in hexadecimal
ISA Card Address = 340;

//Delay Declarations
// Units: ns, us, ms, sec, min, hr
D_0 = 0.144 ms; // test the function of comments
D_1=5000 ns;
D_6 = 11 min;

f_on = FF,8;
f_off = 00,8;
f_dac = 3,7;
f_test =>test.dat,7;
f_sample1 = 1,1;
f_sample2 = 0,1;


////////////////// Program Section ////////////////////////

Top  D_1  f_off + f_on + f_dac + f_sample1;
     D_0  f_on + f_off + f_dac + f_sample2;

     Loop One 12;
       D_1  f_sample1 + f_on + f_dac + f_test;

       Loop Two 16;
         D_1  f_sample1 + f_on + f_dac + f_test;
         D_0  f_sample1 + f_off + f_dac + f_test;
       End Loop Two;

       D_0  f_sample1 + f_off + f_dac + f_test;
     End Loop One;

     Branch Top;
     D_6  f_sample1+f_on + f_dac + f_test;
////////////////// End Program ////////////////////////////
```