BiasDAC comms protocol
MacArthur
Revision History:
11/29/00 First release
1/29/01 Clarified position of C/S bit
2/8/01 Added Set Limit commands, program mode examples
3/9/01 Added Device Busy status, warnings about writing to NVRAM
Added Store Macro, Run Macro commands
4/13/01 Added Recal DAC Channel, Update DAC Curve commands
6/26/01 Added Frequency Counter, Event Generator commands
11/02/01 BiasDAC Rev6 updates, expands functionality of
Read From Memory, Write To Memory. Adds Reset Status
4/8/02 Added BabyDAC commands
5/14/02 More BabyDAC commands
9/18/02 Added Block Read, Block Write commands
11/7/02 added AnaBaby read command

Topology:

The link is set up as a ring with a single host and up to 61 target devices. Typically, each microcontroller represents a device, so if you have two 4-channel DAC boards in the same enclosure, they represent two independent devices. The serial output of the host is connected to the first device in the chain. The output of the first device goes to the input of the second device, etc. The output of the last device goes back to the host. Thus, the host can control up to 61 devices with a single serial I/O port.

Electrical Interface:

Devices may support any or all of the following interfaces:

RS-232 (no hardware handshaking)
RS-488
Fiber-optic (840 nm, ST termination)

These interfaces may be mixed and matched throughout the ring. For example, a PC's RS232 output could go through a fiberoptic adapter, then through fiber to the first device. From the first device, it could go through regular logic level signals to a second device located in the same enclosure, and from there through RS488 or fiber to another device in the same rack, and from there through fiber back to the PC.

Repeating:

For all devices, the input data stream goes into a UART, then into a microprocessor, then is retransmitted through a UART to the next device. In other words, input serial streams are not simply electrically buffered and sent along. This has two benefits:
1) It allows very large rings because it doesn't allow timing slop to accumulate.
2) It guarantees that the data that the microprocessor read is the same data it passed along, removing a possible source of communication error.

Serial Protocol:

Standard Asynchronous, 8 data bits, 1 stop bit, no parity.
Supported bauds: 9600, 19200, 38400, 57600. Others possible. All devices must be running at the same baud.
Eventually, the device's baud will be stored in EEPROM, but for now it will be configured with a dipswitch.

Handshaking:

The host controls the data flow. Each device waits to receive a byte, then retransmits either the same byte, or a different byte, depending on the protocol (see below). Regardless, every device always transmits a single byte for every byte it receives, with the exception of the No Echo byte (see below).

Sync Bit:

The MSB of every byte is reserved for a Sync Bit. This bit, when set, indicates either the start of a command from the host, or a status byte returned from a target. All other transmitted bytes must have their MSB set to 0.

C/S Bit:

When the Sync Bit is set, the C/S bit (next bit after MSB) indicates whether the byte is the start of a command (C/S = 1) or a status byte (C/S = 0).

Device ID:

Each Device has a 6-bit ID which is unique on a physical network. IDs 00 and 63 are reserved.
Eventually, Device IDs will be stored in the Device's EEPROM, but for now it will be configured with a dipswitch

No Echo Byte:

In order to prevent buffer overruns in target devices, the host may transmit No Echo bytes, which have a value of 0xFF, between commands. When a device receives a "No Echo" byte, it absorbs the byte and takes no further action. It does not pass it down along the serial chain.

No Echo bytes allow a host to pack a series of commands into a transmit buffer and still allow an adjustable amount of space between commands to prevent the device’s buffers from overrunning. Conservative programming practice suggests adding a No Echo byte at the end of every command packet, if you can afford the reduced command throughput.

Command Format:

All commands consist of at least two bytes, followed by a checksum byte and a zero pad byte:

BYTE 1 = ID byte
11dddddd, where dddddd is the target Device ID.
Note that the sync bit (bit 7) is set to “1”, indicating the start of a command or status message.
The C/S bit (bit 6) is also set to “1”, indicating that the following message is a command to device dddddd.

BYTE 2 = command byte
00000000 = reserved
00000001 - 00111111 = Universal commands supported by all Devices
01000000 - 01111111 = Device-specific commands

BYTES 3-N = data
0-31 bytes of argument data, depending on the command.
If the host is sending information to the device, it sends it here.
If the host is requesting information, it sends a string of 0's, which the device replaces with the requested data.
A command may both send and request data in the same string.

BYTE N+1 = parity
The host creates this byte by XORing every preceding byte in the command (and setting the MSB to 0). The target checks parity by XORing all of the command bytes, including the parity byte (and setting the MSB to 0). The result should be 0. The target replaces this byte with a parity byte generated from its outgoing data stream (which may be different from the incoming stream).

BYTE N+2 = zero-pad, status.
The host sends a byte set to 0. The selected target replaces it with a status byte (defined below).

Universal Commands:

Here is the map of currently implemented Universal Commands. The commands are defined below.

00000001	Clear Error
00000010	Read From Memory
00000011	Write to Memory
00000100	Stop Program
00000101	Run Program
00000110	Change Device ID
00000111	Change Baud
00001000	Backup Settings
00001001	Set Mode Flags
00001010	Set Interrupt Period
00001011	Store Program
00001100	Store Macro
00001101	Run Macro
00001110	Block Read
00001111	Block Write
0001xxxx	Unspecified
001nnnnn	Get Device Info

Clear Error: 00000001
Clears "sticky" errors resulting from power-on self-tests, etc. Allows host to test device operation even if, for example, one of the device's channels is bad.

Read from Memory: 00000010
Diagnostic used to read one byte of device-specific data from memory
The next byte is the 7 high address bits.
The next byte is the 7 low address bits.
The next two bytes are set to zero by the host, and filled with data by the target, as follows:
The next byte has the high nybble of data in its low nybble
The next byte has the low nybble of data in its low nybble

Write to Memory: 00000011
Diagnostic used to write one byte of device-specific data to memory
The next byte is the 7 high address bits.
The next byte is the 7 low address bits.
The next byte has the high nybble of data in its low nybble
The next byte has the low nybble of data in its low nybble

In the case of the BiasDAC, reads and writes to addresses 0x0000 – 0x01FF will access the microprocessor's internal register space. Writing to this space has a high probability of causing mischief.
Reads and writes to 0x0200 – 0x027F will access the non-volatile memory used to store programs (see the Store Program command below). While the Store Program command is more convenient for storing programs, reading this space will tell a user what programs are stored there.
Reads and writes to 0x0280 – 0x02C0 will access 64 bytes of uncommitted non-volatile memory. Users may use this space to store board-specific parameters such as calibration constants, configuration information, etc.
Refer to the section on Writing to Non-Volatile Memory for warnings on performing NVRAM writes.
Reads and writes to 0x0300 – 0x033F will access the internal registers of the four D/A chips. Bits 5-4 select the D/A and bits 3-0 select the register within the D/A. The D/A is a Burr-Brown DAC1220, and its registers are described in:

http://www-s.ti.com/sc/psheets/sbas082/sbas082.pdf

Stop Program: 00000100
Stops execution of preprogrammed commands

Run Program: 00000101
Runs pre-stored program, starting at following byte location
The next byte is the starting address of the program (0-127)

Change Device ID: 00000110
Changes the current Device ID, and the copy stored in non-volatile memory.
The next byte is the new ID. Note that this can be overridden by dipswitch settings on the device.
Refer to the section on Writing to Non-Volatile Memory for warnings on using this instruction.

Change Baud: 00000111
Changes the copy of the baud stored in non-volatile memory, but not the current baud. Therefore, the change doesn't take effect until a reset.
The next byte is the new baud code, defined as follows:

00000000 =	9600 baud
00000001 =	14.4K
00000010 =	19.2K
00000011 =	38.4K
00000100 =	57.6K

Note that this can be overridden by dipswitch settings on the device.
Refer to the section on Writing to Non-Volatile Memory for warnings on using this instruction.

Backup Settings: 00001000
Copies all device-specific volatile parameters into NV memory. For example, in the case of the BiasDAC, all current DAC settings get backed up. If power is then cycled, the DAC outputs will then default to the new settings.
Refer to the section on Writing to Non-Volatile Memory for warnings on using this instruction. Because multiple bytes are written to NVRAM, the user must wait at least 250 msec before performing another write to non-volatile memory.

Set Mode Flags: 00001001
Sets or clears the seven device-independent mode flags.
The next byte has the new flag settings.
At the moment, the only defined flag is BootToProgram, bit 0. When set to one, the target will attempt to execute a program, starting at location 0, after power-up.
This command causes the changed mode flags to be stored in the device’s non-volatile memory. Refer to the section on Writing to Non-Volatile Memory for warnings on using this instruction.

Set Interrupt Period: 00001010.
Sets the period of the target’s internal interrupt, in microseconds.
The next byte has the 7 MSBs of the period.
The next byte has the 7 LSBs of the period.
It is up to the host to stay within min and max period settings. At the moment, BiasDAC’s minimum period is 500 usec, and maximum is 10,000 usec.

The default interrupt period is 500 usec, and some caution should be taken when changing it. As all of the program mode timing is based on interrupt periods, changing the period will change the program mode timing. Extremely long interrupt periods may also affect the temperature control loop.

Store Program: 00001011
Stores one instruction byte in the specified location, for later execution in program mode.
The next byte has the location to be stored into.
The next byte has the instruction to be stored.
The instructions used in program mode are defined later in this spec.
Refer to the section on Writing to Non-Volatile Memory for warnings on using this instruction.

Store Macro: 00001100
Identical to Store Program, except that it stores instructions into "macro memory", a volatile SRAM area 64 bytes long. Use the Store Macro and Run Macro commands when creating and running temporary programs that don't need to survive a power cycle. This preserves the life of the non-volatile memory.

Run Macro: 00001101
Identical to Run Program, except that it starts execution from macro memory. See the "Store Macro" instruction above.

Block Read: 00001110
Reads up to 127 bytes from the device.
The next three bytes have 21 bits of address information, high byte first.
The next byte has the number of bytes to read, called N.
The next N bytes are padding, and replaced with data read from the device.
The format of this data is device-specific.

Block Write: 00001111
Writes up to 127 bytes from the device.
The next three bytes have 21 bits of address information, high byte first.
The next byte has the number of bytes to write, called N.
The next N bytes are the data to write to the device.
The format of this data is device-specific.

Commands 00001110 – 00011111 are currently unspecified

Get Device Info: 001nnnnn
Requests nnnnn bytes of device information. The command byte should be followed by nnnnn bytes of zeros, which the target replaces with device information.
For all devices, the first byte should be the model number
The second byte will be the revision number.
The remaining bytes will be an ASCII string with more device information.
Current model numbers:
BiasDAC = 1
Frequency Counter = 2
Event Generator = 3

BiasDAC/BabyDAC-Specific commands:

Here is the map of currently implemented BiasDAC-Specific Commands. The commands are defined below.

010000cc	Update DAC channel
010001cc	Update High DAC channel (BabyDAC only)
010010cc	Update DAC mask
010011xx	undefined digibaby readback
010100cc	Update DAC slope
01011Sff	Set/Clear flags
011000cc	Get temperature (BiasDAC only, cc = 00)
	Get ADC value (BabyDAC only)
011001cc	DigiBaby write (BabyDAC only)
011010cc	Update DAC curve
011011cc	Recalibrate DAC channel
011100cc	Set DAC lower limit
01110100	Read AnaBaby (BabyDAC only)
01110101	undefined
0111011x	undefined
011110cc	Set DAC upper limit
011111xx	undefined

Update DAC channel: 010000cc
Followed by three bytes of DAC data, which update channel cc.
The first byte of data is 00aaaaaa
The second byte is 0bbbbbbb
The third byte is 0ccccccc
They get combined into 20-bit data of the form aaaaaabbbbbbbccccccc
This is an unsigned number, meaning that 00000000000000000000 is the bottom of the DAC's range and 11111111111111111111 is the top.
On power-up, the DACs are set to the values that were stored in non-volatile memory by the last Backup command.

Update High DAC channel: 010001cc
BabyDAC only. Identical to Update DAC channel, except that it updates on of four channels on the daughterboard. Note that as of BabyDAC rev 2, the daughterboard DACs were NOT set to the values stored in NV memory.

Update DAC mask: 010010cc
Followed by two bytes of mask data. The first byte has the high nybble,
And the second byte has the low nybble.
DAC mask data is used in conjunction with the Update DAC Slope command to create ramped DAC waveforms. Because each interrupt only has time to update one DAC, the user programs the four DAC masks to determine which DAC gets updated when. The 8 bits of each mask correspond to 8 consecutive interrupt slots.
For example, if you set
DAC0 mask to 01010101,
DAC1 mask to 10000000,
DAC2 mask to 00100000, and
DAC3 mask to 00001000, then DAC0 would be updated every other interrupt, and DACs 1, 2, and 3 would be updated every 8 interrupts.
On power-up, all DAC masks are set to 0.

Update DAC Slope: 010100cc
Followed by 4 bytes of DAC slope data, which update channel cc.
The first byte is 0aaaaaaa
The second byte is 0bbbbbbb
The third byte is 0ccccccc
The fourth byte is 0ddddddd
They get combined into 28-bit data of the form aaaaaaabbbbbbbcccccccddddddd
This is a 2's comp. number which is added to the existing DAC value on every interrupt for which the selected DAC's mask bit is set. In other words, if the interrupt has been programmed with a period of 500 usec, and DAC0's mask is set to 00010001, then DAC0's value will be incremented by the slope number every 4th interrupt, or every 2 msec.
The slope is a left-justified value. To calculate the value for the slope, first determine the desired change in voltage as a fraction of full scale. Multiply this fraction by 4294967296 (2^32). Divide this result by the total number of DAC updates that will take place during the slope.
As a specific example, if we have a DAC whose range is –5V to +5V, and we want to ramp from –2V to +1V, that represents a 3V rise, or 0.3 * full scale. Suppose we've set the interrupt time to 500 usec, and the DAC mask to 00010001, so that it updates every 2 msec, or 500 times/sec. If we want the slope to last for 10 seconds, that makes 5000 updates. So the DAC value = (4294967296 * 0.3) / 5000, or 257698. In hex, that's 0x0003EEA2. We drop the 4 LSBs. The final 28-bit binary number is 0000 0000 0000 0011 1110 1110 1010

Note that if the DAC update exceeds the upper or lower DAC value limit (see Set Lower Limit, below), then the slope is set to 0, and the DAC value is set to the limit.
On power-up, the initial slope value is 0.

Set/Clear flags: 01011SFF
S=0 clears the flag and S=1 sets it.
Sets or clears one of four hardware flags. FF selects the flag.
For BiasDAC:
00 sets/clears PORTB.4, which is connected to pins 1 and 2 of driver IC U7
01 sets/clears PORTB.2, which is connected to pins 6 and 7 of driver IC U7
Note that U7 is an inverting driver. A high on the input turns on the output transistor, which pulls the output pin to ground.
10 sets/clears UART_CTS
For BabyDAC,
00 sets/clears PORTD.0 (pin 23 of DB37 connector)
01 sets/clears PORTD.1 (pin 5 of DB37 connector)
10 sets/clears PORTD.2 (pin 24 of DB37 connector)
11 sets/clears PORTD.3 (pin 6 of DB37 connector)
These bits are readable as the four LSBs of memory address 08.

Get Temperature: 01100000 (BiasDAC Only)
Followed by two bytes of 0, which are replaced with temp data from the target.
The first byte has the 6 high bits of temp data in its 6 LSBs, and the second byte has the 7 low bits of temp data in its 7 LSBs.
The data is two's complement, with 0 = 0 Celcius, and 0.0625 degrees C per LSB.

Get ADC Value: 011000bc (BabyDAC Only)
Followed by three bytes of 0, which are replaced with ADC data from the selected channel.
The first byte of data is 0aaaaaaa
The second byte is 0bbbbbbb
The third byte is 0ccccccc
They get combined into 21-bit data of the form aaaaaaabbbbbbbccccccc
This is an unsigned number, meaning that 000000000000000000000 is the bottom of the ADC's range and 111111111111111111111 is the top.
Setting the "b" bit 0 selects the ADC on the main BabyDAC board. Setting "b" to 1 selects the ADC on the daughterboard. The "c" bit selects one of the two channels on each board.

DigiBaby Write: 011001bc (BabyDAC Only)
Followed by three bytes of digibaby data, which update register bc on the digibaby.
The first byte of data is 00aaaaaa
The second byte is 0bbbbbbb
The third byte is 0ccccccc
They get combined into 20-bit data of the form aaaaaabbbbbbbccccccc
If Digibaby is the first daughterboard, set "b" to 0. If it is the second daughterboard, set "b" to 1. The "c" bit selects between two 20-bit registers on each Digibaby.

Update DAC Curve: 011010cc
Followed by 4 bytes of DAC curve data, which update channel cc.
The first byte is 0aaaaaaa
The second byte is 0bbbbbbb
The third byte is 0ccccccc
The fourth byte is 0ddddddd
They get combined into 28-bit data of the form aaaaaaabbbbbbbcccccccddddddd
This is a 2's comp. number which is first divided by 16, then added to the existing DAC slope on every interrupt for which the selected DAC's mask bit is set. This allows the operator to program quadratic curves into the BiasDAC.

Recalibrate DAC channel: 011011cc (BiasDAC Only)
Sends a recalibrate instruction to the selected D/A chip. While BiasDAC performs a DAC calibration on power-up, it is suggested that a recalibration be performed on all channels once the unit has reached thermal equilibrium.

NOTE: the recalibration can take as long as 500 msec. During that time, no commands which update DAC values should be sent to the selected DAC channel (this includes ramps and commands sent under program control). Other DAC channels are unaffected by the command.

Set DAC Lower Limit: 011100cc
Set DAC Upper Limit: 011110cc
Followed by three bytes of DAC data, which update channel cc.
These bytes are in the same format as in the Update DAC Channel command.
Once the upper and lower limits are set, if the user attempts to set the DAC channel outside these limits, either with an Update DAC Channel or an Update DAC Slope command, the DAC value will be set to the nearer limit, and the slope will be set to 0.
On power-up, the upper limit is set to FFFFF, and the lower limit is set to 00000.

Read Anababy: 01110100
Followed by 4 bytes.
The first byte is the channel number
The second byte is set to 0, and is replaced by the high 2 bits of the returned ADC data.
The third byte is set to 0, and replaced by bits 7-13 of the ADC data
The fourth byte is set to 0, and replaced by bits 0-6 of the ADC data.

Frequency Counter-Specific commands:

Read Counter: 01000ccc
ccc = counter channel

This instruction stops the count on the selected channel, reads the 5-byte master counter and the 4-byte frequency counter, returns them, then resets and restarts the counters.

The host follows the command byte with 9 bytes of 00. The target replaces these bytes with counter information as follows:

```
Byte 1: 0000aaaa
Byte 2: 0bbbbbbb
Byte 3: 0ccccccc
Byte 4: 0ddddddd
Byte 5: 0eeeeeee
Byte 6: 00000fff
Byte 7: 0ggggggg
Byte 8: 0hhhhhhh
Byte 9: 0iiiiiii
```

where aaaabbbbbbbcccccccdddddddeeeeeee is the 32-bit master count and fffggggggghhhhhhhiiiiiii is the 24-bit frequency count. To determine the frequency, multiply the master frequency (50 MHz at the moment) by the frequency count and divide the result by the master count.

The msb of each count is an overflow indicator. If either bit is set, then the respective counter has overflowed, which means too much time has elapsed since the last Read Counter instruction.

Event Generator-Specific commands:

Set Channel Delay: 010ccccc
ccccc = channel (0-23)
Followed by four bytes of DAC data, which update channel ccccc.
The first byte of data is 00aaaaaa
The second byte is 0bbbbbbb
The third byte is 0ccccccc
They get combined into 20-bit data of the form aaaaaabbbbbbbccccccc
The last byte has channel configuration information, of the form 000000TP
"T" is the termination bit. When cleared, the selected channel's output goes active the programmed delay after the trigger, and stays active until the next trigger. When set, the output is cleared when the channel that numerically follows the selected channel goes active. In other words, if you program channel 19 to 100 msec, and channel 20 to 150 msec, you will create a pulse on channel 19 which is set at 100 msec, and cleared at 150 msec.
"P" is the polarity bit. When cleared, active means logical high. When set, active means logical low.

Initialize Event Generator: 01100000
Followed by 1 byte of initialization data, defined as 000RRREP

Bits RRR select the counter resolution

000 = 10 usec
001 = 100 usec
010 = 1 msec
011 = 10 msec
100 = 100 msec

Bits EP select the trigger source:
00 = software trigger (from the Trigger Event Generator command)
01 = software trigger
10 = External trigger, falling edge
11 = External trigger, rising edge

Trigger Event Generator: 01100001
Performs a software trigger of event generator

Status bytes:

A selected target may, at any time during the transmission of a command block, replace any byte with a status byte of the format 10ssssss, indicating an error. If there is no error in the execution of the command, it replaces the last byte of the command (the zero pad) with 1000000, indicating normal status.

Error messages:

0x80 = normal status, command implemented
0x81 = parity error
0x82 = non-supported command
0x83 = argument out of range
0x84 = device busy
0x85 = device recovered from reset

Program Mode Commands:

As mentioned above, the target can be programmed to run autonomously, thus generating waveforms without host intervention. The Stop Program, Run Program, and Store Program commands described above facilitate this. To create a program, the user stores program commands, followed by their arguments, in the target's non-volatile 128-byte command space. Note that the user doesn't need to store sync bytes, stop bits, no-echo bytes, etc. There is some overlap between commands supported in Program Mode and those supported in normal mode. However, some normal mode commands don't make sense in Program Mode, and there are special commands that only run in Program mode.

Stop Program: 00000100
Stops execution of preprogrammed commands and returns to normal mode.

Run Program: 00000101
Runs pre-stored program, starting at following byte location
The next byte is the starting address of the program (0-127)
In the context of Program Mode, this acts like a GOTO command.

Run Macro: 00001101
Identical to Run Program, except that it starts execution from macro memory.

Set Timeout: 00010000
Initializes an interrupt counter, which decrements once per interrupt. After the counter expires, a timeout flag is set, which can be used by the Wait For Timeout command, described below.
The next byte has the 7 high bits of the timeout value
The next byte has the 7 middle bits of the timeout value
The next byte has the 7 low bits of the timeout value
The maximum timeout duration is 2097151 (0x1FFFFF) interrupts.

Wait For Timeout: 00010001
Pauses execution until the timeout counter has counted out.

Wait for Trigger: 00010010
Pauses execution until a hardware trigger condition is met.
The next byte has the trigger condition: 0000TTEP
TT specifies which trigger
TT = 00: PORTB.0
TT = 01: PORTB.1
TT = 10: PORTB.5 (UART_RTS)
TT = 11: undefined
When E = 1 and P = 1, trigger is a positive edge.
When E = 1 and P = 0, trigger is a negative edge
When E = 0 and P = 1, trigger is a positive level
When E = 0 and P = 0, trigger is a negative level

Update DAC channel: 010000cc
Followed by three bytes of DAC data

Update DAC mask: 010010cc
Followed by two bytes of mask data

Update DAC Slope: 010100cc
Followed by 4 bytes of DAC slope data.

Update DAC Curve: 011010cc
Followed by 4 bytes of DAC curve data.

Set/Clear flags: 01011SFF
Sets or clears one of four hardware flags. FF selects the flag.
S=0 clears the flag and S=1 sets it.

Set DAC Lower Limit: 011100cc
Set DAC Upper Limit: 011110cc
Followed by 3 bytes a DAC limit data. This is a good way to terminate a ramp if you are trying to ramp to a specific value.

Program Mode Example #1: Power-on flag

In many applications, it is handy to have an indication as to when the DACs have finished initializing and stabilizing after power-up. This example runs immediately after the BiasDAC powers up and initializes its DACs. It pauses one second, then turns on an output flag. This flag may be used to drive an LED, a small relay, or both.
(Please refer to the Usage Note concerning the mislabeling of the flags in BiasDAC's schematic.)

```
ADDR BINARY    HEX  COMMAND

00   00010000  10   Set Timeout to 2000 interrupts
01   00000000  00
02   00001111  0F
03   01010000  50
04   00010001  11   Wait for timeout
05   01011100  5C   Set flag 0
06   00000100  04   Stop program
```

In order for this to work properly, the user must have previously used the Set Mode Flags command to set the BootToProgram bit. BiasDAC's microcontroller examines this bit after initializing the DACs, and if it is set, executes the program starting at address 00.

The first command, Set Timeout, initializes a timeout counter to the value stored in the next three bytes. To set the correct timeout duration, calculate the number of interrupts in the desired duration, in this case 1 second, or 2000 interrupts. Convert to binary:
2000 = 0x7D0 = 11111010000.
Now counting from the right, insert a zero every seven bits:

```
11111010000
1111 1010000
111101010000
```

Right-justify, and parse into bytes:

```
             1111 01010000
00000000 00001111 01010000 = 00, 0F, 50
```

The next command is Wait for Timeout. It suspends execution of the next instruction until the timer has counted to 0. It does not interfere with commands coming in on the serial port, nor does it suspend ramps, etc. It is perfectly permissible to put as many instructions as you wish between the command that sets up the timer and that which waits for timeout.

The next command sets flag 0, which activates an open-collector driver on BiasDAC, which can be used to drive an LED or relay.

The next command stops execution of program mode, returning the box to normal mode. At this point, the user is free to send direct commands to the BiasDAC or run another program located above address 06 in program space.

Program Mode Example #2: Trapezoids

Now for something a bit trickier. This program generates a one-second ramp from –3V (assuming the DAC range is –5V to +5V) to +3V, pauses at +3V for 500 msec, then ramps back down to –3V over a second, pauses there for 500 msec, then repeats the cycle. This program illustrates more than anything the necessity of automating the code generation when dealing with all but the simplest programs. Doing the math and bit manipulations can take many hours when done by hand.

Here is the program, located at address 0x10 to avoid bumping into the power-on code.

```
ADDR BINARY    HEX  COMMAND

10   01110000  70   Set low limit of DAC0
11   00001100  0C   to 2/10 full scale
12   01100110  66
13   00110011  33
14   01111000  78   Set high limit of DAC0
15   00110011  33   to 8/10 full scale
16   00011001  19
17   01000100  44
18   01010000  50   Set slope of DAC0
19   00000000  00   to 0
1A   00000000  00
1B   00000000  00
1C   00000000  00
1D   01001000  48   Set mask of DAC0
1E   00000101  05   to every other interrupt
1F   00000101  05
20   01000000  40   Set DAC0 value
21   00001100  0C   to 2/10 full scale
22   01100110  66
23   00110011  33
```

```
Loop:
24   00010000  10   Set timeout
25   00000000  00   to 1500 msec (3000 interrupts)
26   00010111  17
27   00111000  38
28   01010000  50   Set slope of DAC0
29   00000000  00   to 6/10 full scale in 1000 msec
2A   00001001  09
2B   01101010  6A
2C   00100101  25
2D   00010001  11   Wait for timeout
2E   00010000  10   Set timeout
2F   00000000  00   to 1500 msec (3000 interrupts)
30   00010111  17
31   00111000  38
32   01010000  50   Set slope of DAC0
33   01111111  7F   to –6/10 full scale in 1000 msec
34   01110110  76
35   00010101  15
36   01011010  5A
37   00010001  11   Wait for timeout
38   00000101  05   Goto
39   00100100  24   Loop
```

When generating a ramp that you want to stop at a set voltage, it is often easier (and more accurate) to set the limit voltage of the DAC to the termination voltage of the ramp, rather than trying to stop the ramp by setting a timeout. For our application, we set up the top and bottom of the trapezoid at +3V and –3V. The math for that is the same as for setting the DAC values. On a +/-5V full scale, -3V is 0.2 * full scale. Full scale = 2^20, or 1048576. So –3V corresponds to 209715 or 33333 or

```
00110011001100110011
```

We parse the bits into 6:7:7, and add leading zeros

```
  001100  1100110  0110011
00001100 01100110 00110011
```

We repeat the process to set the high limit to +3V, or 8/10 of full scale. Then we set the slope to 0, which keeps it from running away when we turn it on by setting the mask.

Next we set the interrupt mask to 01010101, which means that the DAC will be updated every other interrupt. If the interrupt is the default 500 usec, this means that the DAC is updated every 1 msec. Avoid the temptation to set the mask to 11111111. While this will work, it will also lock out other DACs – not just those being ramped, but also those which are set through direct program control.

Once we've set the mask, the DAC will add the slope value to the DAC value until the mask is set to 0. However, at the moment, the slope is 0, so the DAC value doesn't move. At this point, we initialize the DAC value –3V, the bottom of the trapezoid.

Now we're ready to run the loop. We want a ramp which rises for 1 second, then pauses for 500 msec before falling. We'll do this by programming the ramp to rise 0.6 * full scale in 1 second. At 1 second, the ramp will run into the high limit, which will set the DAC value at the high limit value and reset the slope to 0.

But before we do this, we'll set the timeout to 1.5 seconds, which will tell us when to start the falling ramp. We do this before setting the slope because it gives us a bit more timing accuracy. It takes a finite amount of time to execute the program instructions, so it's good practice to place the Set Timeout instruction immediately after the Wait For Timeout instruction (which is at instruction location 37). Even doing that, the maximum delay between the timer timing out and the Set Timeout instruction reinitializing it is unspecified. For that reason, if you need precise synchronization, it is best to use an accurate external clock going to one of BiasDAC's input flags, then use the Wait for Trigger command to synchronize to it.

1500 msec corresponds to 3000 interrupts (assuming 500 usec per interrupt). Note that the timeout counter sees all interrupts, not just those that update the selected DAC.
3000 is 0xBB8 is

```
101110111000
```

We right-justify, parse into 7-bit chunks (working right-to-left), and add zeros to complete the bytes:

```
            10111  0111000
00000000 00010111 00111000
```

Note, if we wanted to find the maximum possible timeout duration, we could work backwards:

`01111111 01111111 01111111` turns into
`111111111111111111111` or
`11111 11111111 11111111` or 0x1FFFFF or 2097151 interrupts or 1048575 msec at 500 usec/interrupt. That's about 17.5 minutes.

Next, we will set the slope to 6/10 full scale (from 2/10 full scale to 8/10 full scale) in 1 second. 1 second = 1000 DAC update cycles, so start with a full scale equal to 4 unsigned bytes, or 2^32, or 4294967296, and multiply by 0.6 to get 2576980377. Next, divide by the number of update cycles (1000) to get 2576980, or 0x275254 or

```
00000000 00100111 01010010 01010100
```

(remember to right-justify to make a 32-bit number)

Now parse into 7-bit bytes, working left to right, which means that we're going to drop the 4 LSBs.

```
 0000000  0001001  1101010  0100101  0100
00000000 00001001 01101010 00100101
```

As soon as the Set Slope command is processed, the DAC will start rising at 0.6FS/sec until it hits the upper limit. This will cap the DAC value and set the slope to zero.

500 msec later, the timer will time out. In the meantime, BiasDAC will continue executing program step # 2D. While BiasDAC is waiting for the timeout, it will continue to execute any new commands coming in over the serial port, allowing you to set DAC levels, abort the program, or whatever.

When the timer times out, the program will continue, setting a new 1.5 second timeout, then setting a new slope of –0.6FS/sec. It then waits for the second timeout, then executes what in program mode is equivalent to a GOTO command. This jumps the execution back to the top of the loop.

Writing to Non-Volatile Memory

The target devices contain non-volatile memory (in the form of EEPROM) which retains data even with the power removed. The following commands perform writes to this memory:

Write to Memory (certain addresses)
Change Device ID
Change Baud
Backup Settings
Set Mode Flags
Store Program

The non-volatile memory has a specified life of 100,000 cycles, so none of the above commands should be executed in a tight loop.

In addition, it takes the microprocessor 10 msec to store data into non-volatile memory, so after executing any of the above instructions, the user must wait at least 10 msec before executing another of the above instructions. Otherwise, the device will abort the command and return busy status.