AI Signatech S4000/S6000 Interface Protocol

1. Overview

The AI Signatech S4000/S6000 controller provides an RS232 interface for asynchronous serial communications between a host computer (PC) and target controller (remote) allowing for configuration of various timing and current parameters in the light head assembly.

There are only four commands available to communicate with the Signatech controller, Get Firmware Revision, Get Target Status, Read Parameters and Write Parameters. These commands provide the basic tools for configuring the current and timing parameters stored in the Signatech controllers.

2. Setup

Communications between the host and remote occurs at 19200 baud, 8 data bits, 1 stop bit and no parity. The host controller initiates ALL communications by sending a binary command sequence to the target. In response, the target controller will process the command upon complete reception, and send an acknowledgement back to the host controller indicating the command was processed successfully.

3. Command / Reponses

Commands are comprised of a command header, *target id*, and parameters. The command header is 8 bytes of data. The preamble to the header is always 4 bytes of 00h followed by 4 bytes of the command number. For example, the command header for the Get Status command is

00h 00h 00h 00h 33h 33h 33h 33h

All data bytes following the command header are transmitted as two bytes per byte of data. The first byte is the actual data byte and the second byte is the one's complement of the first byte.

The target id byte is the first field after the command header. This field is the desired output to configure. There are two outputs on S6000/S6000-AS controllers. To configure Output 1, set *target id* to 00h. For Output 2, *target id* is set to 01h. On S4000 controllers, *target id* is always set to 00h.

Some commands (Read Parameters and Write Parameters) require an extra byte after the *target id* byte. This byte is the *address* byte and indicates the memory address to begin reading from or writing to. In all commands requiring an *address* byte, it should be set to 00h (see Table 1 – Signatech S4000/S6000 Controller Commands and Responses).

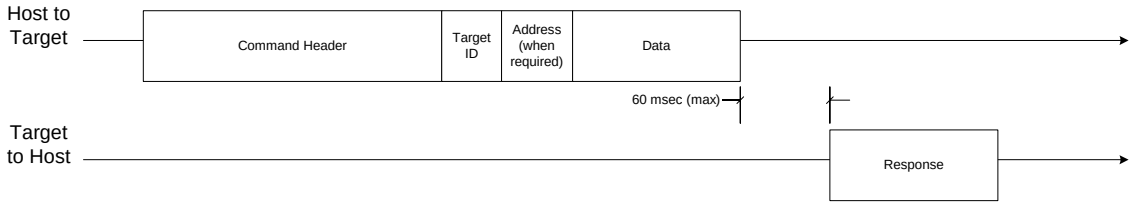
The Write Parameters command For Data is transmitted in a binary format with two bytes transmitted per byte of data. The first byte is always the original data byte and the second is the one's complement of the original data byte.

4. Timing

The host computer has up to one second to transmit between successive bytes in the command sequence. If more than one second elapses between bytes received by the target, the target discards the bytes already received and begins waiting for a new command header. Likewise, if an error condition (i.e. framing error) is detected on the serial interface, the bytes received from the host up to the time the error is detected are discarded and the target begins waiting for another command header from the host. In both aforementioned cases, the target does NOT respond to the host controller with any type of acknowledgement.

The maximum latency between command reception and acknowledgement is approximately 60 msec* (this latency is 500 msec maximum on firmware revisions earlier than revision 206). This latency occurs when the Write Parameters command is processed.

Figure 1 - Command/Response Timing



5. Light Head Identification

Each light head assembly contains a unique arrangement of LEDs which have different voltage and current requirements. Signatech controllers can identify attached light head assemblies by reading the voltage present at both the ID1 and ID2 inputs on the light head connector.

Each output of a Signatech controller connects to a light head assembly which consists of up to four channels of LED strings with each channel able to control up to 4 Amps of current. Connected to each each channel, can be 2 to 5 strings of LEDs with each string consisting of 4 to 7 LEDs. The value returned in *id1 pointer* identifies how many LEDs per string and how many strings per channel are in the current attached light head. The value returned by *id2 pointer* provides information on the type of LEDs connected on a per channel basis (please see Figure 2 - ID Pointer Definitions).

Figure 2 - ID Pointer Definitions

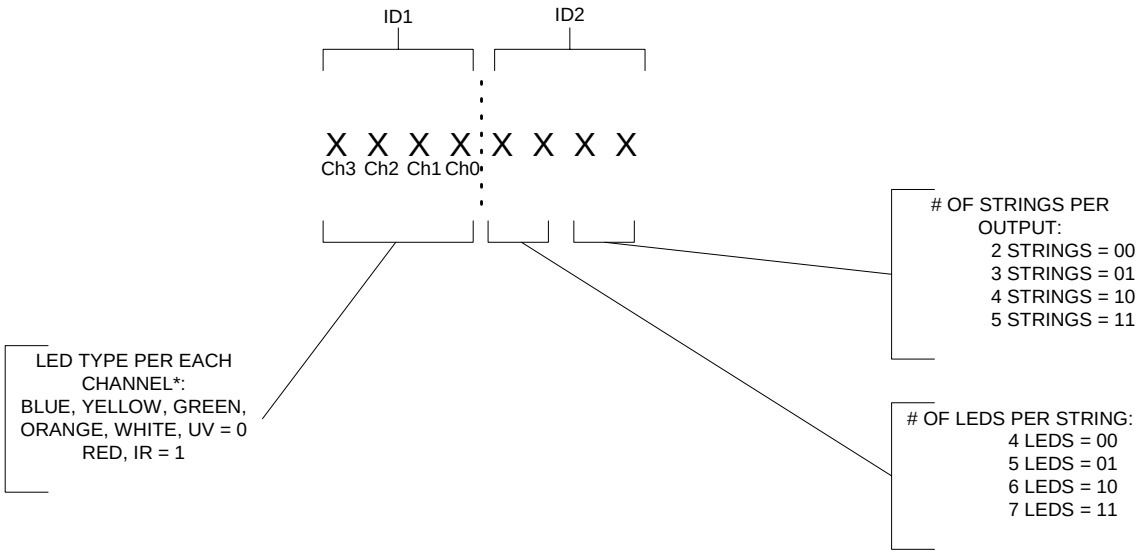


Table 1 – Signatech S4000/S6000 Controller Commands and Responses

Command/Request	**Format**	**Description**
*Get Firmware ID	<00h> <00h> <00h> <00h> <22h> <22h> <22h> <22h> <target id > <~target id> [firmware id byte 0] [~firmware id byte 0] [firmware id byte 1] [~firmware id byte 1] . . [firmware id byte 15] [~firmware id byte 15]	Return target firmware revision string ***Command*** *target id*: 0 (output 1) 1 (output 2) * for S4000 controller *target id* is always 0. ***Response*** *firmware id bytes 0-15:* 16 byte ASCII string containing the firmware part number/revision. The string returned should be “8200-000058-xxx” where xxx is the firmware revision number, i.e. 206.
Get Target Status	<00h> <00h> <00h> <00h> <33h> <33h> <33h>	Get the lighthead identification from the target controller. ***Command*** *target id*: 0 (output 1) 1 (output 2)

	<33h> <target id > <~target id> [head id 1] [~head id 1] [id 1 pointer] [~id 1 pointer] [head id 2] [~head id 2] [id 2 pointer] [~id 2 pointer] [status] [~status] [high voltage] [~high voltage] [firmware revision] [~firmware revision] [hardware revision] [~hardware revision]	* for S4000 controller *target id* is always 0. ***Response*** *head id 1:* A/D value of voltage read at HEAD_ID1 A/D input. *id 1 pointer:* Light head identifier 1 (see Figure 2 - ID Pointer Definitions). *head id 2:* A/D value of voltage read at HEAD_ID2 A/D input. *id 2 pointer:* Light head identifier 2 (see see Figure 2 - ID Pointer Definitions). *status:* target status. If the corresponding bit is set, then the error condition has occurred. bit 0: EEPROM read/write error bit 1: Strobe 0: target is s6000 1: target is s4000 bit 2: RS232 error bit 3: DAC error bit 4: I2C Bus Busy error bit 5: High Voltage error bit 6: ID error Lighthead parameters are not compatible with current attached lighthead. bit 7: Checksum error Stored EEPROM checksum does match computed checksum. *high voltage:* A/D value of high voltage read at HV_SENSE A/D input. *firmware revision:* Firmware revision number (no longer used) *hardware revision:* Hardware revision number
Read Parameters	<00h> <00h> <00h> <00h> <55h> <55h> <55h> <55h> <target id >	Read controller parameters from the target EEPROM. ***Command*** *target id:* 0 (output 1) 1 (output 2) * for S4000 controller *target id* is always 0. *address:*

	<~target id> <address> <~address> [target id] [~target id] [address] [~address] [current setpoint channel 0] [~current setpoint channel 0] [current setpoint channel 1] [~current setpoint channel 1] [current setpoint channel 2] [~current setpoint channel 2] [current setpoint channel 3] [~current setpoint channel 3] [current flags/high voltage pointer] [~current flags/high voltage pointer] [on time MSB] [~on time MSB] [on time LSB] [~on time LSB] [off time MSB] [~off time MSB] [off time LSB] [~off time LSB] [delay time MSB] [~delay time MSB] [delay time LSB] [~delay time LSB] [hold time MSB] [~hold time MSB] [hold time LSB] [~hold time LSB] [stop after pulse count] [~stop after pulse count] [id pointer 2/id pointer 1] [~id pointer 2/id pointer 1] [checksum] [~checksum]	always set to 00h. ***Response*** *target id:* 0 (output 1) 1 (output 2) * for S4000 controller *target id* is always 0. *address:* always set to 00h. *current setpoint channel 0:* 8-bit value representing the desired current setpoint for channel 0. If bit 4 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 4 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current setpoint channel 1:* 8-bit value representing the desired current setpoint for channel 1. If bit 5 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 5 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current setpoint channel 2:* 8-bit value representing the desired current setpoint for channel 2. If bit 6 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 6 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current setpoint channel 3:*

		8-bit value representing the desired current setpoint for channel 3. If bit 7 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 7 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current flags/high voltage pointer:* If the desired output current is greater than 200mA for a given channel, set the appropriate high current enable flag. bits 3-0: Pointer to 16 byte high voltage table bit 4: channel 0 high current enable flag bit 5: channel 1 high current enable flag bit 6: channel 2 high current enable flag bit 7: channel 3 high current enable flag *on time:* The amount of time in microseconds the lighthead is on (0 to 64000d). A value of zero indicates the lighthead is on continuously. *off time* The amount of time in microseconds the lighthead is off (0 to 64000d). *delay time:* The time delay in microseconds (0 to 64000d) between trigger pulse detection and the enabling of the lighthead. *hold time:* The amount of time in microseconds after the lighthead is turned off that additional triggers are ignored (0 to 64000d). *stop after pulse count:* The number of pulses generated on a single trigger event (0 to 255d). *id pointer 2/id pointer 1:* bits 3-0: Light head identifier 1 (see see Figure 2 - ID Pointer Definitions). bits 7-4: Light head identifier 2 (see

		see Figure 2 - ID Pointer Definitions). *checksum:* checksum is 8-bit sum of last 15 data bytes excluding the one's complement data bytes.
Write Parameters	<00h> <00h> <00h> <00h> <AAh> <AAh> <AAh> <AAh> <target id > <~target id> <address> <~address> <current setpoint channel 0> <~current setpoint channel 0> <current setpoint channel 1> <~current setpoint channel 1> <current setpoint channel 2> <~current setpoint channel 2> <current setpoint channel 3> <~current setpoint channel 3> <current flags/high voltage pointer> <~current flags/high voltage pointer> <on time MSB> <~on time MSB> <on time LSB> <~on time LSB> <off time MSB> <~off time MSB> <off time LSB> <~off time LSB> <delay time MSB> <~delay time MSB> <delay time LSB> <~delay time LSB> <hold time MSB> <~hold time MSB> <hold time LSB> <~hold time LSB> <stop after pulse count> <~stop after pulse count> <id pointer 2/id pointer 1> <~id pointer 2/id pointer 1> <checksum> <~checksum> [target id]	Write controller parameters to the target EEPROM. ***Command*** *target id:* 0 (output 1) 1 (output 2) * for S4000 controller *target id* is always 0. *address:* always set to zero. ***Response*** *target id:* 0 (output 1) 1 (output 2) * for S4000 controller *target id* is always 0. *address:* always set to 00h. *current setpoint channel 0:* 8-bit value representing the desired current setpoint for channel 0. If bit 4 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 4 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current setpoint channel 1:* 8-bit value representing the desired current setpoint for channel 1. If bit 5 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 5 is set, the equation used to compute the setpoint value is

	[~target id] [address] [~address] [current setpoint channel 0] [~current setpoint channel 0] [current setpoint channel 1] [~current setpoint channel 1] [current setpoint channel 2] [~current setpoint channel 2] [current setpoint channel 3] [~current setpoint channel 3] [current flags/high voltage pointer] [~current flags/high voltage pointer] [on time MSB] [~on time MSB] [on time LSB] [~on time LSB] [off time MSB] [~off time MSB] [off time LSB] [~off time LSB] [delay time MSB] [~delay time MSB] [delay time LSB] [~delay time LSB] [hold time MSB] [~hold time MSB] [hold time LSB] [~hold time LSB] [stop after pulse count] [~stop after pulse count] [id pointer 2/id pointer 1] [~id pointer 2/id pointer 1] [checksum] [~checksum]	*setpoint = (5000 / 255) * current (in mA)* *current setpoint channel 2:* 8-bit value representing the desired current setpoint for channel 2. If bit 6 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 6 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current setpoint channel 3:* 8-bit value representing the desired current setpoint for channel 3. If bit 7 in the *current flags/high voltage pointer* is clear, the equation used to compute the setpoint value is *setpoint = (250 / 255) * current (in mA)* Otherwise, if bit 7 is set, the equation used to compute the setpoint value is *setpoint = (5000 / 255) * current (in mA)* *current flags/high voltage pointer:* If the desired output current is greater than 200mA for a given channel, set the appropriate high current enable flag. bits 3-0: Pointer to 16 byte high voltage table bit 4: channel 0 high current enable flag bit 5: channel 1 high current enable flag bit 6: channel 2 high current enable flag bit 7: channel 3 high current enable flag *on time:* The amount of time in microseconds the lighthead is on (0 to 64000d). A value of zero indicates the lighthead is on continuously. *off time* The amount of time in microseconds the lighthead is off (0 to 64000d).

		delay time: The time delay in microseconds (0 to 64000d) between trigger pulse detection and the enabling of the lighthead. *hold time:* The amount of time in microseconds after the lighthead is turned off that additional triggers are ignored. *stop after pulse count:* The number of pulses generated on a single trigger event (0 to 255d). *id pointer 2/id pointer 1:* bits 3-0: Pointer to 16 byte ID1 table bits 7-4: Pointer to 16 byte ID2 table *checksum:* checksum is 8-bit sum of last 15 data bytes excluding the one's complement data bytes.

Notes:

* This command is only supported in firmware revision 206 and higher.

1. < > denotes command and command data originating from host computer.
2. [ ] denotes response data originating from target (S4000/S6000 controller).

6. Appendix

a. Read Parameters Command example code

```
AI_API Read_Controller_szPort (
        char                                    *portName,
                // ==>      "COM1", "COM2", etc.
        unsigned char                   *outputSelect,  // ==>      1 or 2   (Output 1 or Output 2)
        unsigned char                   *dominance,                     // ==>  0 or 1   (Dominant
Timing or Dominant Current)
        unsigned short          *ch1Curr,               // ==>  0 - 4000  milliamps
        unsigned short          *ch2Curr,               // ==>  0 - 4000  milliamps
        unsigned short          *ch3Curr,               // ==>      0 - 4000  milliamps
        unsigned short          *ch4Curr,               // ==>  0 - 4000  milliamps
        unsigned long                   *onTime,                        // ==>  0 - 64000 microseconds
        unsigned long                   *offTime,                       // ==>  0 - 64000 microseconds
        unsigned long                   *delayTime,                     // ==>  0 - 64000 microseconds
        unsigned long                   *holdTime,                      // ==>  0 - 64000 microseconds
        unsigned char                   *trigRepeat,            // ==>  0 - 255   pulses
        unsigned char   *id1,           // <==  light head ID no. 1
        unsigned char   *id2            // <==  light head ID no. 2
)
{
        static unsigned char receive_buffer[18];
        unsigned char i, trial, number_of_bytes, error = 0;
        LPSTR           szPortName;

        /* Get the sz string from the VB BSTR crap */
        szPortName = (LPSTR) portName;

        if ((*outputSelect < 1) || (*outputSelect > 2))
                return AI_OUTPUT_PARAM_ERR;
```

```
	/* ask for data */
	if ((error = InitComm (szPortName, *outputSelect - 1, receive_buffer)) != 0)
		return error;

	trial=1;
	do {
		for(i=0; i<18; i++)
			receive_buffer[i]=0;

		number_of_bytes=18;

		if ((error = Read_Data(*outputSelect - 1, 0, receive_buffer)) == 0)
		{
			*ch1Curr    = (((receive_buffer [4 +2] >> 4) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)receive_buffer [0 +2] + 0.5f;
			*ch2Curr    = (((receive_buffer [4 +2] >> 5) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)receive_buffer [1 +2] + 0.5f;
			*ch3Curr    = (((receive_buffer [4 +2] >> 6) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)receive_buffer [2 +2] + 0.5f;
			*ch4Curr    = (((receive_buffer [4 +2] >> 7) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)receive_buffer [3 +2] + 0.5f;
			*onTime     = receive_buffer [5 +2] * 256 + receive_buffer [6 +2];
			*offTime    = receive_buffer [7 +2] * 256 + receive_buffer [8 +2];
			*delayTime  = receive_buffer [9 +2] * 256 + receive_buffer [10+2];
			*holdTime   = receive_buffer [11+2] * 256 + receive_buffer [12+2];
			*trigRepeat = receive_buffer [13+2];
			*id2        = receive_buffer [14+2] >> 4;
			*id1        = receive_buffer [14+2] & 15;
			break;
		}

		trial++;

	} while(trial<3);


	if(trial==3)
		return error;

 // sanity check all values returned by controller, since we indicating no error
	if (*ch1Curr > 4000)		*ch1Curr = 4000;
	if (*ch2Curr > 4000)		*ch2Curr = 4000;
	if (*ch3Curr > 4000)		*ch3Curr = 4000;
	if (*ch4Curr > 4000)		*ch4Curr = 4000;


	return AI_NO_ERR;
}
```

b. Write Parameters Command example code

```
	unsigned int max_direct[2] = { 20, 30 };		/* max dc */
	double Qjmax[2] = { 1.0E2, 1.0E2 };				/* max junction temp */
	double Qamax[2] = { 4.5E1, 4.5E1 };				/* max ambient temp */
	double Rf[2] =    { 5.5E0, 4.0E0 };				/* diode resistance */
	double Vf[2] =    { 3.6E0, 1.9E0 };				/* Vf @ 20mA */
	double Rthja[2] = { 7.0E2, 7.0E2 };				/* thermal resistance */
	double Cth[2] =   { 5.2E-5, 5.2E-5 };		/* thermal capacitance */
	double Iabsmax[2]={ 1.0E3, 4.0E3 };				/* absolute max current */


	double Pmax[2];			/* Pmax[i]=(Qjmax[i]-Qamax[i])/Rthja[i] */
	double RCth[2];			/* RCth[i]=Rthja[i]*Cth[i] */
	double MaxOnMul[2];		/* MuxOnMul[i]=Pmax*(1-exp((-6.4E-2)/RCth[i])) */
```

```
AI_API Program_Controller_szPort (
        char                                                    *portName,                              // ==>
        "COM1", "COM2", etc.
        unsigned char           *outputSelect,          // ==>   0 or 1    (Output 1 or Output 2)
        unsigned char           *dominance,                     // ==>  0 or 1    (Dominant Timing or
Dominant Current)
        unsigned short      *ch1Curr,                           // ==>  0 - 4000  milliamps
        unsigned short      *ch2Curr,                           // ==>  0 - 4000  milliamps
        unsigned short      *ch3Curr,                           // ==>       0 - 4000  milliamps
        unsigned short      *ch4Curr,                           // ==>  0 - 4000  milliamps
        unsigned long           *onTime,                        // ==>  0 - 64000 microseconds
        unsigned long           *offTime,                       // ==>  0 - 64000 microseconds
        unsigned long           *delayTime,                     // ==>  0 - 64000 microseconds
        unsigned long           *holdTime,                      // ==>  0 - 64000 microseconds
        unsigned char           *trigRepeat,                    // ==>  0 - 255   pulses
        unsigned char   *id1,                                   // <==  light head ID no. 1
        unsigned char   *id2                                    // <==  light head ID no. 2
)
{
        unsigned char data_buffer[16];
        unsigned char receive_buffer[18];

        unsigned int  time[4];
        unsigned int  current[4];


        unsigned char i, j,  trial, number_of_bytes,
                                                block_address, word_address;
        LPSTR           szPortName;
        unsigned char timmy, led_type, number_of_strings;
        static unsigned char tempstr [255], temptry = 0;
        unsigned char ecode;


        /* Strobe 0 Block Address == 0x00 */
        /* Strobe 1 Block Address == 0x01 */

 if ((*outputSelect < 1) || (*outputSelect > 2))
                return AI_OUTPUT_PARAM_ERR;

        block_address = *outputSelect - 1;
        word_address = 0;

        /* Mode == 0 => Currents have higher priority */
        /* Mode == 1 => Time has higher priority */

 if ((*dominance < 0) || (*dominance > 1))
                return AI_DOMINANCE_PARAM_ERR;


   /* Check current values */
        current [0] = *ch1Curr;
        current [1] = *ch2Curr;
        current [2] = *ch3Curr;
        current [3] = *ch4Curr;
        for (j = 0; j < 4; j++)
                if ((current [j] < 0) || (current [j] > 4000))
                        return AI_CURRENT_PARAM_ERR;

   /* Check times */
        time [0] = *onTime;
        time [1] = *offTime;
        time [2] = *delayTime;
        time [3] = *holdTime;
        for (j = 0; j < 4; j++)
                if ((time [j] < 0) || (time [j] > 64000))
                        return AI_TIME_PARAM_ERR;

        /* Check Trigger Repeat (Stop After) */
        data_buffer [13] = *trigRepeat;
```

```
/* Adjust for DC, etc. */
        if((time[0]==0)||(time[1]==0)){          /* OnTime or OffTime == 0 => DC */
                time[0]=0;                                              /* OnTime */
                data_buffer[13]=0;                                      /* StopAfter */
                time[1]=0;                                              /* OffTime */
                time[3]=0;                                              /* HoldTime */
        }

        if(time[2]==0){                                 /* DelayTime==0 => Immediate */
                time[3]=0;                              /* HoldTime */
                data_buffer[13]=0;          /* StopAfter */
        }

        if(data_buffer[13]==0)                          /* StopAfter == 0 => Continuous */
                time[3]=0;                              /* HoldTime */

        for(i=0; i<2; i++)
                Pmax[i]=(Qjmax[i]-Qamax[i])/Rthja[i];

        for(i=0; i<2; i++)
                RCth[i]=Rthja[i]*Cth[i];

        for(i=0; i<2; i++)
                MaxOnMul[i]=Pmax[i]*(1-exp((-6.4E-2)/RCth[i]));

        szPortName = (LPSTR) portName;

        if ((ecode = InitComm (szPortName, *outputSelect - 1, receive_buffer)))
          return ecode;

        /* Retrieve light head IDs from controller if we don't have any */
        trial=1;
        do {
                for(i=0; i<18; i++)
                        receive_buffer[i]=0;

                number_of_bytes=8;

                if ((ecode = Request_ID(block_address)))
                {
                        return ecode;
                }
                if((ecode = Receive_Data(receive_buffer, number_of_bytes)) == AI_NO_ERR)
                {
                        break;
                }
                trial++;

        } while(trial<3);

        if(trial==3)
        {
                return ecode;
        }


        // fill in details
        ucFwRev                         = receive_buffer [6];
        ucHwRev                         = receive_buffer [7];
        ucFwFlags               = receive_buffer [4];
        ucHV                              = receive_buffer [5];
        ucID1                             = receive_buffer [1];
        ucID2                           = receive_buffer [3];

        temptry = trial;  // save the number of ID trials for later display

        data_buffer[14]=receive_buffer[3]<<4;                   /* ID2Pointer */
        data_buffer[14]=data_buffer[14]+receive_buffer[1];      /* ID1Pointer */
```

```
/* keep BYGOW to a max of 1 A per string */
number_of_strings=(data_buffer[14]&0x03)+2;
for (j = 0; j < 4; j++)
{
	timmy = j + 4;
	led_type=(data_buffer[14]>>(timmy))&0x01;
	if ((led_type == 0) && ((current [j] / number_of_strings) > 1000))
		current [j] = number_of_strings * 1000;
}

/* make sure everything is kosher */
if(!time[0])
	DC_Requested(data_buffer, time, current);
else if (*dominance)				/* Dominant Timing */
	Dominant_Timing(data_buffer, time, current);
else 							/* Dominant Current */
	Dominant_Current(data_buffer, time, current);

/* Compute Checksum */
data_buffer[15]=0;

for(i=0; i<15; i++)
data_buffer[15]=data_buffer[15]+data_buffer[i];

trial=1;
do
{
	/* Send data */
	for(i=0; i<18; i++)
		receive_buffer[i]=0;

	number_of_bytes=18;

	FlushFileBuffers (hCom);
	Write_Data(block_address, word_address, data_buffer);
	if ((ecode = Receive_Data(receive_buffer, number_of_bytes)) == AI_NO_ERR)
	{
		for(i=0; i<16; i++)
			if(receive_buffer[i+2]!=data_buffer[i])
				ecode = AI_DATA_PROGVERIFY_ERR;

		/* Update the data (may be modified) */
		if(ecode == AI_NO_ERR)
		{
			*ch1Curr    = (((data_buffer [4] >> 4) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)data_buffer [0] + 0.5f;
			*ch2Curr    = (((data_buffer [4] >> 5) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)data_buffer [1] + 0.5f;
			*ch3Curr    = (((data_buffer [4] >> 6) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)data_buffer [2] + 0.5f;
			*ch4Curr    = (((data_buffer [4] >> 7) & 1) ? (double)(5000.0 / 255.0) : (double)(250.0 /
255.0)) * (double)data_buffer [3] + 0.5f;
			*onTime     = data_buffer [5] * 256 + data_buffer [6];
			*offTime    = data_buffer [7] * 256 + data_buffer [8];
			*delayTime  = data_buffer [9] * 256 + data_buffer [10];
			*holdTime   = data_buffer [11] * 256 + data_buffer [12];
			*trigRepeat = data_buffer [13];
			*id2        = data_buffer [14] >> 4;
			*id1        = data_buffer [14] & 15;
			if (*ch1Curr > 4000) *ch1Curr = 4000;
			if (*ch2Curr > 4000) *ch2Curr = 4000;
			if (*ch3Curr > 4000) *ch3Curr = 4000;
			if (*ch4Curr > 4000) *ch4Curr = 4000;
			break;
		}
	}

	trial++;

} while(trial<3);
```

```
	if(trial==3)
		return ecode;

	return (GetFwError(ucFwFlags));
}



/* For each output the maximum Direct Current is determined and requested */
/* current is adjusted down if necessary */
void DC_Requested(unsigned char *data_buffer, unsigned int *time, unsigned int *current)
{

	unsigned char i, led_type, dc_limit, number_of_strings;
	double temp_current;
	static char timmy;

	number_of_strings=(data_buffer[14]&0x03)+2;
	data_buffer[4]=0x01;		/* CurrentFlags = 0, HVPointer = 1 */

	for(i=0; i<4; i++) {
		timmy = i + 4;
		led_type=(data_buffer[14]>>(timmy))&0x01;
		dc_limit=max_direct[led_type];

		temp_current=(double)current[i]/number_of_strings;

		if(current[i])
			if(!temp_current)
				temp_current=1;

		if(temp_current>dc_limit)
			temp_current=dc_limit;

		current[i]=temp_current*number_of_strings;

		data_buffer[i]=(unsigned char)((double)(((((double)current[i])*255.0)/250.0+0.5f));

	}

	for(i=5; i<13; i++)
		data_buffer[i]=0;

	data_buffer[10]=time[2];				/* DelayTime LSB */
	data_buffer[9]=time[2]>>8;				/* DelayTime MSB */
	data_buffer[12]=time[3];				/* HoldTime LSB */
	data_buffer[11]=time[3]>>8;			/* HoldTime MSB */

	return;

}


/* Mode 1 */
/* OnTime and OffTime are as requested. For each output the maximum current */
/* for the specified OnTime and OffTime is computed and requested current is */
/* adjusted down if necessary */
void Dominant_Timing(unsigned char *data_buffer,unsigned int *time, unsigned int *current)
{

	unsigned char i, k, led_type, number_of_strings, string_length;
	unsigned int /*temp_current, */voltage;
	double OnTime, OffTime, Pdmax, Imax, MaxVoltage, Voltage, temp_current;
	unsigned int max_current[2];
	static unsigned char timmy;

	data_buffer[4]=0;	/* CurrentFlags and HVPointer */
	k=0x10;
```

```
MaxVoltage=0;

string_length=((data_buffer[14]>>2)&0x03)+4;
number_of_strings=(data_buffer[14]&0x03)+2;

OnTime=time[0]*1.0E-6;
OffTime=time[1]*1.0E-6;

for(i=0; i<2; i++) {
        Pdmax=Pmax[i]*(1-exp(-OffTime/RCth[i]))/(1-exp(-OnTime/RCth[i]));
        Imax=(1.0E3)*(-Vf[i]+sqrt(Vf[i]*Vf[i]+4*Rf[i]*Pdmax))/(2*Rf[i]);
        max_current[i]=(unsigned int)Imax;
        if(max_current[i]<max_direct[i])
                max_current[i]=max_direct[i];
}

for(i=0; i<4; i++) {
         timmy = i + 4;
        led_type=(data_buffer[14]>>(timmy))&0x01;
        temp_current=(double)current[i]/number_of_strings;

        if(current[i])
                if(!temp_current)
                        temp_current=1;

        led_type, current[i], max_current [led_type], 0xFF >> (i + 4));

        if(temp_current>max_current[led_type])
                temp_current=max_current[led_type];

        Voltage=Vf[led_type]+temp_current*Rf[led_type]*1.0E-3;

        if(Voltage>MaxVoltage)
                MaxVoltage=Voltage;

        current[i]=temp_current*number_of_strings;

        if(current[i]>200) {
                data_buffer[4]=data_buffer[4]|(k<<i);

                Imax=current[i]*5.1E-2 + 0.5f;
                data_buffer[i]=(unsigned char)Imax;

        }
        else
                data_buffer[i]=(unsigned char)((current[i]*255.0)/250.0+0.5f);
}

MaxVoltage=MaxVoltage*string_length+15.5;

voltage=(unsigned int)MaxVoltage;

if(voltage>100)
        voltage=100;

if(voltage<30)
        voltage=30;

voltage=voltage/5-5;
k=(unsigned char)voltage;
data_buffer[4]=data_buffer[4]|k;

data_buffer[6]=time[0];                                         /* OnTime LSB */
data_buffer[5]=time[0]>>8;                                      /* OnTime MSB */
data_buffer[8]=time[1];                                         /* OffTime LSB */
data_buffer[7]=time[1]>>8;                                      /* OffTime MSB */
data_buffer[10]=time[2];                                        /* DelayTime LSB */
data_buffer[9]=time[2]>>8;                                      /* DelayTime MSB */
data_buffer[12]=time[3];                                        /* HoldTime LSB */
data_buffer[11]=time[3]>>8;                             /* HoldTime MSB */
```

```
	return;

}


/* Mode 0 */
/* Currents are as requested. For each output the OffTime needed for */
/* requested current and OnTime is computed. If OffTime>64,000 us, the */
/* minimum OnTime is computed */
void	Dominant_Current(unsigned char *data_buffer, unsigned int *time, unsigned int *current)
{
	unsigned char i, k, led_type, number_of_strings, string_length, adjust;
	unsigned int voltage;
	double MaxOnTime, MinOffTime, Pd, Temp, MaxVoltage, Voltage, temp_double;
	 static char timmy;

	data_buffer[4]=0;	/* CurrentFlags and HVPointer */
	k=0x10;
	MaxVoltage=0;

	string_length=((data_buffer[14]>>2)&0x03)+4;
	number_of_strings=(data_buffer[14]&0x03)+2;

	MaxOnTime=time[0]*1.0E-6;
	MinOffTime=time[1]*1.0E-6;

	adjust=0;

	for(i=0; i<4; i++) {
		timmy = i + 4;
		led_type=(data_buffer[14]>>(timmy))&0x01;

		temp_double = (double)current[i]/number_of_strings;
		if(current[i])
		{
			if (temp_double == 0)
				temp_double = 1;
		}
		Voltage=Vf[led_type]+temp_double*Rf[led_type]*1.0E-3;

		if(Voltage>MaxVoltage)
			MaxVoltage=Voltage;

		Pd=Voltage*temp_double*1.0E-3;

		if(Pd>Pmax[led_type]) {

			Temp=-RCth[led_type]*log(1-MaxOnMul[led_type]/Pd);

			if(MaxOnTime>Temp) {
				MaxOnTime=Temp;
				MinOffTime=6.4E-2;
			}
			else {
				Temp=-RCth[led_type]*log(1-Pd*(1-exp(-
				MaxOnTime/RCth[led_type]))/Pmax[led_type]);
				if(Temp>MinOffTime)
					MinOffTime=Temp;
			}

		}

		current[i]=temp_double * number_of_strings;//temp_int*number_of_strings;

		if(current[i]>200) {
			data_buffer[4]=data_buffer[4]|(k<<i);
			// Dac Value * 5Vref / 255 = Current in mA * 0.95238 ohms / 1000

			Temp=current[i]*5.1E-2 + 0.5f;
```

```
                        data_buffer[i]=(unsigned char)Temp;
                }
                else
                        // Dac Value * 5Vref / 255 = Current in mA * 20ohms / 1000
                        data_buffer[i]=(unsigned char)((current[i]*255.0)/250.0 + 0.5f);
        }

        MaxVoltage=MaxVoltage*string_length+15.5;

        voltage=(unsigned int)MaxVoltage;

        if(voltage>100)
                voltage=100;

        if(voltage<30)
                voltage=30;

        voltage=voltage/5-5;

        k=(unsigned char)voltage;

        data_buffer[4]=data_buffer[4]|k;


        MaxOnTime=MaxOnTime*1.0E6+.5;
        time[0]=(unsigned int)MaxOnTime;

        MinOffTime=MinOffTime*1.0E6+.5;
        time[1]=(unsigned int)MinOffTime;

        data_buffer[6]=time[0];                                         /* OnTime LSB */
        data_buffer[5]=time[0]>>8;                                      /* OnTime MSB */
        data_buffer[8]=time[1];                                         /* OffTime LSB */
        data_buffer[7]=time[1]>>8;                                      /* OffTime MSB */
        data_buffer[10]=time[2];                                        /* DelayTime LSB */
        data_buffer[9]=time[2]>>8;                                      /* DelayTime MSB */
        data_buffer[12]=time[3];                                        /* HoldTime LSB */
        data_buffer[11]=time[3]>>8;                             /* HoldTime MSB */

        return;
}
```