BMD100 Communications Protocol

March 19, 2012

NeuroSky®
Brain-Computer Interface Technologies

The NeuroSky® product families consist of hardware and software components for simple integration of this biosensor technology into consumer and industrial end-applications. All products are designed and manufactured to meet consumer thresholds for quality, pricing, and feature sets. NeuroSky sets itself apart by providing building block component solutions that offer friendly synergies with related and complementary technological solutions.

Contents

Introduction

ThinkGear™ is the technology inside every NeuroSky product or partner product that enables a device to interface with the user's bio-signals.

The BMD100™ device incorporates ThinkGear technology in a convenient, stylish device form factor.

This BMD100 Communications Protocol document defines, in detail, how to communicate with the BMD100. In particular, it describes:

- How to **connect** to the Bluetooth serial data stream to receive a stream of bytes.
- How to **parse** the serial data stream of bytes to reconstruct the ECG data sent by the BMD100
- How to **interpret and use** the ECG data that are sent from the ThinkGear (including raw wave, heart rate, and signal quality data) in a BCI application

The ThinkGear Data Values chapter defines the types of Data Values that can be reported by ThinkGear in a BMD100. It is highly recommended that you read this section to familiarize yourself with which kinds of Data Values are (and aren't) available from BMD100 before continuing to later chapters.

The ThinkGear Packets chapter describes the ThinkGear Packet format used to deliver the ThinkGear Data Values over the serial I/O stream.

Bluetooth Interface

The BMD100 transmits ThinkGear Data Values, encoded within ThinkGear Packets, as a serial stream of bytes over Bluetooth via a standard Bluetooth **Serial Port Profile (SPP)**:

- Bluetooth Profile: Serial Port Profile (SPP)
- Baud Rate: 57600
- Authentication key: 0000

Please refer to the BMD100 Quick Start Guide and/or BMD100 Instruction Manual that accompanied your BMD100 for instructions on how to pair the BMD100 to your Windows or Mac computer via SPP using Bluetooth drivers and Bluetooth stacks available for those platforms. For information on pairing the BMD100 via SPP on other platforms, please refer to your platform's documentation, and to the SPP specifications that can be found on the Web.

ThinkGear Data Values

POOR_SIGNAL Quality

This unsigned one-byte integer value describes how poor the signal measured by the ThinkGear is. It is either 0 or 200. A value of 0 indicates that there is no contact between the electrode and skin. Conversely, a value of 200 indicates that there is contact between the electrode and skin. This value is typically output every second, and indicates the poorness of the most recent measurements.

Poor signal may be caused by a number of different things. In order of severity, they are:

- Electrode contacts not being on a person's skin (i.e. when nobody is touching the electrodes).
- Poor contact of the electrode to a person's skin (i.e. hair or dirt in the way)
- Excessive motion of the wearer (i.e. moving fingers excessively, jostling the device).
- Excessive environmental electrostatic noise (some environments have strong electric signals or static electricity buildup in the person touching the sensor).
- Excessive non-ECG biometric noise (i.e. EMG, EEG, EOG, etc)

A certain amount of noise is unavoidable in normal usage of ThinkGear, and NeuroSky's filtering technology has been designed to detect, correct, compensate for, account for, and tolerate many types of non-ECG noise. The `POOR_SIGNAL` Quality value is more useful to some applications which need to be more sensitive to noise (such as some medical or research applications), or applications which need to know right away when there is even minor noise detected.

By default, output of this Data Value is enabled. It is typically output once a second.

HEART_RATE Value

This unsigned one-byte integer value provides the heart rate value based on the past one second of data. This value is outputted even when the POOR_SIGNAL Quality value is zero. Therefore, it is important to disregard this value when the POOR_SIGNAL Quality is zero.

Occasionally, the HEART_RATE Value is excessively high (such as 200) or excessively low (such as 20). This may be caused by a number of different things. In order of severity, they are:

- Electrode contacts not being on a person's skin (i.e. when nobody is touching the electrodes).
- Poor contact of the electrode to a person's skin (i.e. hair or dirt in the way)
- Excessive motion of the wearer (i.e. moving fingers excessively, jostling the device).
- Excessive environmental electrostatic noise (some environments have strong electric signals or static electricity buildup in the person touching the sensor).

- Excessive non-ECG biometric noise (i.e. EMG, EEG, EOG, etc)

By default, output of this Data Value is enabled. It is typically output once a second.

RAW Wave Value (16-bit)

This Data Value consists of two bytes, and represents a single raw wave sample. Its value is a signed 16-bit integer that ranges from -32768 to 32767. The first byte of the Value represents the high-order bits of the twos-compliment value, while the second byte represents the low-order bits. To reconstruct the full raw wave value, simply shift the first byte left by 8 bits, and bitwise-or with the second byte:

```
short raw = (Value[0]<<8) | Value[1];
```

where `Value[0]` is the high-order byte, and `Value[1]` is the low-order byte.

In systems or languages where bit operations are inconvenient, the following arithmetic operations may be substituted instead:

```
raw = Value[0]*256 + Value[1];
if( raw >= 32768 ) raw = raw - 65536;
```

where `raw` is of any signed number type in the language that can represent all the numbers from -32768 to 32767.

BMD100 reports raw waves that fall between -32768 to 32767.

By default, output of this Data Value is enabled, and is outputed 512 times a second, or approximately once every 2ms.

Config Byte

This Data Value consists of 1 byte. After the application reads the Config Byte, the application must send the Config Byte back to the BMD100 for internal configuration.

The Config Byte is flagged by the code, 0x08. When parsing data, the Config Byte will appear immediately following the 0x08 code. The Config Byte will be a integer between 0 and 255.

For more details on parsing packets, please see the **Example Packet** section located later on in this document.

For more details on sending Command Bytes, please see the **Sending Command Byte to BMD100** application note.

Debug Output 1

This Data Value consists of 5 bytes. This value is outputted for NeuroSky's debugging purposes. Please ignore this value.

Debug Output 2

This Data Value consists of 3 bytes. This value is outputted for NeuroSky's debugging purposes. Please ignore this value.

ThinkGear Packets

ThinkGear components deliver their digital data as an asynchronous serial stream of bytes. The serial stream must be parsed and interpreted as ThinkGear Packets in order to properly extract and interpret the ThinkGear Data Values described in the chapter above.

A ThinkGear Packet is a packet format consisting of 3 parts:

1. Packet Header
2. Packet Payload
3. Payload Checksum

ThinkGear Packets are used to deliver Data Values (described in the previous chapter) from a ThinkGear module to an arbitrary receiver (a PC, another microprocessor, or any other device that can receive a serial stream of bytes). Since serial I/O programming APIs are different on every platform, operating system, and language, it is outside the scope of this document (see your platform's documentation for serial I/O programming). This chapter will only cover how to interpret the serial stream of bytes into ThinkGear Packets, Payloads, and finally into the meaningful Data Values described in the previous chapter.

The Packet format is designed primarily to be robust and flexible: Combined, the Header and Checksum provide data stream synchronization and data integrity checks, while the format of the Data Payload ensures that new data fields can be added to (or existing data fields removed from) the Packet in the future without breaking any Packet parsers in any existing applications/devices. This means that any application that implements a ThinkGear Packet parser properly will be able to use newer models of ThinkGear modules most likely without having to change their parsers or application at all, even if the newer ThinkGear hardware includes new data fields or rearranges the order of the data fields.

Packet Structure

Packets are sent as an asynchronous serial stream of bytes. The transport medium may be UART, serial COM, USB, bluetooth, file, or any other mechanism which can stream bytes.

Each Packet begins with its Header, followed by its Data Payload, and ends with the Payload's Checksum Byte, as follows:

```
[SYNC] [SYNC] [PLENGTH]  [PAYLOAD...]     [CHKSUM]
_____________________    _____________    ___________
^^^^^^^^(Header)^^^^^^^  ^^(Payload)^^    ^(Checksum)^
```

The `[PAYLOAD…]` section is allowed to be up to 169 bytes long, while each of `[SYNC]`, `[PLENGTH]`, and `[CHKSUM]` are a single byte each. This means that a complete, valid Packet is a minimum of 4 bytes long (possible if the Data Payload is zero bytes long, i.e. empty) and a maximum of 173 bytes long (possible if the Data Payload is the maximum 169 bytes long).

A procedure for properly parsing ThinkGear Packets is given below in Step-By-Step Guide to Parsing a Packet.

Packet Header

The Header of a Packet consists of 3 bytes: two synchronization [SYNC] bytes (0xAA 0xAA), followed by a [PLENGTH] (Payload length) byte:

```
[SYNC] [SYNC] [PLENGTH]
_______________________
^^^^^^^^(Header)^^^^^^^
```

The two [SYNC] bytes are used to signal the beginning of a new arriving Packet and are bytes with the value 0xAA (decimal 170). Synchronization is two bytes long, instead of only one, to reduce the chance that [SYNC] (0xAA) bytes occurring within the Packet could be mistaken for the beginning of a Packet. Although it is still possible for two consecutive [SYNC] bytes to appear *within* a Packet (leading to a parser attempting to begin parsing the middle of a Packet as the beginning of a Packet) the [PLENGTH] and [CHKSUM] combined ensure that such a "mis-sync'd Packet" will never be accidentally interpreted as a valid packet (see Payload Checksum below for more details).

The [PLENGTH] byte indicates the length, in bytes, of the Packet's Data Payload [PAYLOAD…] section, and may be any value from 0 up to 169. Any higher value indicates an error (PLENGTH TOO LARGE). Be sure to note that [PLENGTH] is the length of the Packet's **Data Payload**, NOT of the entire Packet. The Packet's complete length will always be [PLENGTH] + 4.

Data Payload

The Data Payload of a Packet is simply a series of bytes. The number of Data Payload bytes in the Packet is given by the [PLENGTH] byte from the Packet Header. The interpretation of the Data Payload bytes into the ThinkGear Data Values described in Chapter 1 is defined in detail in the Data Payload Structure section below. Note that parsing of the Data Payload **typically should not even be attempted** until **after** the Payload Checksum Byte [CHKSUM] is verified as described in the following section.

Payload Checksum

The [CHKSUM] Byte must be used to verify the integrity of the Packet's Data Payload. The Payload's Checksum is defined as:

1. summing all the bytes of the Packet's Data Payload
2. taking the lowest 8 bits of the sum
3. performing the bit inverse (one's compliment inverse) on those lowest 8 bits

A receiver receiving a Packet must use those 3 steps to calculate the checksum for the Data Payload they received, and then compare it to the [CHKSUM] Checksum Byte received with the Packet. If the calculated payload checksum and received [CHKSUM] values do not match, the entire Packet should be discarded as invalid. If they do match, then the receiver may procede to parse the Data Payload as described in the "Data Payload Structure" section below.

Data Payload Structure

Once the Checksum of a Packet has been verified, the bytes of the Data Payload can be parsed. The Data Payload itself consists of a continuous series of Data Values, each contained in a series of bytes called a DataRow. Each DataRow contains information about what the Data Value represents, the length of the Data Value, and the bytes of the Data Value itself. Therefore, to parse a Data Payload, one must parse each DataRow from the Data Payload, until all bytes of the Data Payload have been parsed.

DataRow Format

A DataRow consists of bytes in the following format:

```
([ EXCODE]...) [ CODE]   ([ VLENGTH])   [ VALUE...]
_____________________ _____________ _____________
^^^^(Value Type)^^^^ ^^(length)^^ ^^(value)^^
```

Note: Bytes in parentheses are conditional, meaning that they only appear in some DataRows, and not in others. See the following description for details.

The DataRow may begin with zero or more [EXCODE] (**Ex**tended **Code**) bytes, which are bytes with the value 0x55. The number of [EXCODE] bytes indicates the Extended Code Level. The Extended Code Level, in turn, is used in conjuction with the [CODE] byte to determine what type of Data Value this DataRow contains. Parsers should therefore always begin parsing a DataRow by counting the number of [EXCODE] (0x55) bytes that appear to determine the Extended Code Level of the DataRow's [CODE].

The [CODE] byte, in conjunction with the Extended Code Level, indicates the type of Data Value encoded in the DataRow. For example, at Extended Code Level 0, a [CODE] of 0x04 indicates that the DataRow contains an eSense Attention value. For a list of defined [CODE] meanings, see the CODE Definitions Table below. Note that the [EXCODE] byte of 0x55 will never be used as a [CODE] (incidentally, the [SYNC] byte of 0xAA will never be used as a [CODE] either).

If the [CODE] byte is between 0x00 and 0x7F, then the the [VALUE...] is implied to be 1 byte long (referred to as a Single-Byte Value). **In this case, there is no [VLENGTH] byte, so the single [VALUE] byte will appear immediately after the [CODE] byte.**

If, however, the [CODE] is greater than 0x7F, then a [VLENGTH] ("Value Length") byte immediately follows the [CODE] byte, and this is the number of bytes in [VALUE...] (referred to as a Multi-Byte Value). These higher CODEs are useful for transmitting arrays of values, or values that cannot be fit into a single byte.

The DataRow format is defined in this way so that any properly implemented parser will not break in the future if new CODEs representing arbitrarily long DATA... values are added (they simply ignore unrecognized CODEs, but do not break in parsing), the order of CODEs is rearranged in the Packet, or if some CODEs are not always transmitted in every Packet.

A procedure for properly parsing Packets and DataRows is given below in Step-By-Step Guide to Parsing a Packet and Step-By-Step Guide to Parsing DataRows in a Packet Payload, respectively.

CODE Definitions Table

Single-Byte CODEs

Extended Code Level	[CODE]	(Byte) [LENGTH]	Data Value Meaning
0	0x02	–	POOR_SIGNAL Quality (0 or 200)
0	0x03	–	HEART_RATE Value (0 to 255)
0	0x08	–	CONFIG_BYTE (0 to 255)

Multi-Byte CODEs

Extended Code Level	[CODE]	(Byte) [LENGTH]	Data Value Meaning
0	0x80	2	RAW Wave Value: a single big-endian 16-bit two's-compliment signed value (high-order byte followed by low-order byte) (-32768 to 32767)
0	0x84	5	DEBUG_1
0	0x85	3	DEBUG_2
Any	0x55	–	NEVER USED (reserved for [EXCODE])
Any	0xAA	–	NEVER USED (reserved for [SYNC])

(any Extended Code Level/CODE combinations not listed in the table above have not yet been defined, but may be added at any time in the future)

For detailed explanations of the meanings of each type of Data Value, please refer to the chapter on ThinkGear Data Values.

Example Packet

The following is a typical packet. Aside from the [SYNC], [PLENGTH], and [CHKSUM] bytes, all the other bytes (bytes [3] to [34]) are part of the Packet's Data Payload. Note that the DataRows within the Payload are **not** guaranteed to appear in every Packet, nor are any DataRows that do appear guaranteed by the Packet specification to appear in any particular order.

```
byte: value // [CODE] Explanation

[ 0]: 0xAA  // [SYNC]
[ 1]: 0xAA  // [SYNC]
[ 2]: 0x12  // [PLENGTH] (payload length) of 18 bytes
[ 3]: 0x02  // [POOR_SIGNAL]
[ 4]: 0x00  // Poor signal detected (0)
[ 5]: 0x03  // [HEART_RATE]
[ 6]: 0xAA  // Heart Rate value of 170 beats per minute
[ 7]: 0x84  // [DEBUG_1]
[ 8]: 0x05  // [VLENGTH] 5 bytes
[ 9]: 0x00  // (1/5) Begin DEBUG_1
[10]: 0xF9  // (2/5)
```

```
[11]: 0x00  // (3/5)
[12]: 0x03  // (4/5)
[13]: 0x44  // (5/5) End DEBUG_1
[14]: 0x08  // [CONFIG_BYTE]
[15]: 0x39  // CONFIG_BYTE value
[16]: 0x85  // [DEBUG_2]
[17]: 0x03  // [VLENGTH] 3 bytes
[18]: 0xFF  // (1/3) Begin DEBUG_2
[19]: 0xFF  // (2/3)
[20]: 0xFF  // (3/3) End DEBUG_2
[21]: 0xC1  // [CHKSUM] (1's comp inverse of 8-bit Payload sum)
```

Step-By-Step Guide to Parsing a Packet

1. Keep reading bytes from the stream until a `[SYNC]` byte (`0xAA`) is encountered.
2. Read the next byte and ensure it is also a `[SYNC]` byte
   - If not a `[SYNC]` byte, return to step 1.
   - Otherwise, continue to step 3.
3. Read the next byte from the stream as the `[PLENGTH]`.
   - If `[PLENGTH]` is 170 (`[SYNC]`), then repeat step 3.
   - If `[PLENGTH]` is greater than 170, then return to step 1 (`PLENGTH TOO LARGE`).
   - Otherwise, continue to step 4.
4. Read the next `[PLENGTH]` bytes of the `[PAYLOAD…]` from the stream, saving them into a storage area (such as an `unsigned char payload[256]` array). Sum up each byte as it is read by incrementing a checksum accumulator (`checksum += byte`).
5. Take the lowest 8 bits of the checksum accumulator and invert them. Here is the C code:

   ```
   checksum &= 0xFF;
   checksum = ~checksum & 0xFF;
   ```

6. Read the next byte from the stream as the`[CHKSUM]` byte.
   - If the `[CHKSUM]` does not match your calculated chksum (`CHKSUM FAILED`).
   - Otherwise, you may now parse the contents of the Payload into DataRows to obtain the Data Values, as described below.
   - In either case, return to step 1.

Step-By-Step Guide to Parsing DataRows in a Packet Payload

Repeat the following steps for parsing a DataRow until all bytes in the payload[] array (`[PLENGTH]` bytes) have been considered and parsed:

1. Parse and count the number of `[EXCODE]` (`0x55`) bytes that may be at the beginning of the current DataRow.
2. Parse the `[CODE]` byte for the current DataRow.

3. If [CODE] >= 0x80, parse the next byte as the [VLENGTH] byte for the current DataRow.
4. Parse and handle the [VALUE…] byte(s) of the current DataRow, based on the DataRow's [EXCODE] level, [CODE], and [VLENGTH] (refer to the Code Definitions Table).
5. If not all bytes have been parsed from the payload[ ] array, return to step 1. to continue parsing the next DataRow.

Sample C Code for Parsing a Packet

The following is an example of a program, implemented in C, which reads from a stream and (correctly) parses Packets continuously. Search for the word TODO for the two sections which would need to be modified to be appropriate for your application.

Note: For simplicity, error checking and handling for standard library function calls have been omitted. A real application should probably detect and handle all errors gracefully.

```
#include <stdio.h>

#define SYNC   0xAA
#define EXCODE 0x55

int parsePayload( unsigned char *payload, unsigned char pLength ) {

    unsigned char bytesParsed = 0;
    unsigned char code;
    unsigned char length;
    unsigned char extendedCodeLevel;
    int i;

    /* Loop until all bytes are parsed from the payload[] array... */
    while( bytesParsed < pLength ) {

        /* Parse the extendedCodeLevel, code, and length */
        extendedCodeLevel = 0;
        while( payload[bytesParsed] == EXCODE ) {
            extendedCodeLevel++;
            bytesParsed++;
        }
        code = payload[bytesParsed++];
        if( code & 0x80 ) length = payload[bytesParsed++];
        else              length = 1;

        /* TODO: Based on the extendedCodeLevel, code, length,
         * and the [CODE] Definitions Table, handle the next
         * "length" bytes of data from the payload as
         * appropriate for your application.
         */
        printf( "EXCODE level: %d CODE: 0x%02X length: %d\n",
                extendedCodeLevel, code, length );
        printf( "Data value(s):" );
        for( i=0; i<length; i++ ) {
            printf( " %02X", payload[bytesParsed+i] & 0xFF );
        }
        printf( "\n" );

        /* Increment the bytesParsed by the length of the Data Value */
```

```
        bytesParsed += length;
    }

    return( 0 );
}

int main( int argc, char **argv ) {

    int checksum;
    unsigned char payload[256];
    unsigned char pLength;
    unsigned char c;
    unsigned char i;

    /* TODO: Initialize 'stream' here to read from a serial data
     * stream, or whatever stream source is appropriate for your
     * application.  See documentation for "Serial I/O" for your
     * platform for details.
     */
    FILE *stream = 0;
    stream = fopen( "COM4", "r" );

    /* Loop forever, parsing one Packet per loop... */
    while( 1 ) {

        /* Synchronize on [SYNC] bytes */
        fread( &c, 1, 1, stream );
        if( c != SYNC ) continue;
        fread( &c, 1, 1, stream );
        if( c != SYNC ) continue;

        /* Parse [PLENGTH] byte */
        while( true ) {
            fread( &pLength, 1, 1, stream );
            if( pLength ~= 170 ) break;
        }
        if( pLength > 169 ) continue;

        /* Collect [PAYLOAD...] bytes */
        fread( payload, 1, pLength, stream );

        /* Calculate [PAYLOAD...] checksum */
        checksum = 0;
        for( i=0; i<pLength; i++ ) checksum += payload[i];
        checksum &= 0xFF;
        checksum = ~checksum & 0xFF;

        /* Parse [CKSUM] byte */
        fread( &c, 1, 1, stream );

        /* Verify [CKSUM] byte against calculated [PAYLOAD...] checksum */
        if( c != checksum ) continue;

        /* Since [CKSUM] is OK, parse the Data Payload */
        parsePayload( payload, pLength );
    }

    return( 0 );
```

```
}
```

ThinkGearStreamParser C API

The ThinkGearStreamParser API is a library which implements the parsing procedure described above and abstracts it into two simple functions, so that the programmer does not need to worry about parsing Packets and DataRows at all. All that is left is for the programmer to get the bytes from the data stream, stuff them into the parser, and then define what their program does with the `Value[]` bytes from each DataRow that is received and parsed.

The source code for the ThinkGearStreamParser API is provided as part of the BMD100 Development Tools (MDT), and consists of a `.h` header file and a `.c` source file. It is implemented in pure ANSI C for maximum portability to all platforms (including microprocessors).

Using the API consists of 3 steps:

1. Define a data handler (callback) function which handles (acts upon) Data Values as they're received and parsed.
2. Initialize a `ThinkGearStreamParser` struct by calling the `THINKGEAR_initParser()` function.
3. As each byte is received from the data stream, the program passes it to the `THINKGEAR_parseByte()` function. This function will automatically call the data handler function defined in 1) whenever a Data Value is parsed.

The following subsections are excerpts from the `ThinkGearStreamParser.h` header file, which serves as the API documentation.

Constants

```
/* Parser types */
#define PARSER_TYPE_NULL       0x00
#define PARSER_TYPE_PACKETS    0x01 /* Stream bytes as ThinkGear Packets */
#define PARSER_TYPE_2BYTERAW   0x02 /* Stream bytes as 2-byte raw data */

/* Data CODE definitions */
#define PARSER_POOR_SIGNAL_CODE     0x02
#define PARSER_HEARTRATE_CODE       0x03
#define PARSER_RAW_CODE             0x80
#define PARSER_DEBUGA_CODE          0x08
#define PARSER_DEBUGB_CODE          0x84
#define PARSER_DEBUGC_CODE          0x85
```

THINKGEAR_initParser()

```
/**
 * @param parser              Pointer to a ThinkGearStreamParser object.
 * @param parserType          One of the PARSER_TYPE_* constants defined
 *                            above: PARSER_TYPE_PACKETS or
 *                            PARSER_TYPE_2BYTERAW.
 * @param handleDataValueFunc A user-defined callback function that will
 *                            be called whenever a data value is parsed
 *                            from a Packet.
 * @param customData          A pointer to any arbitrary data that will
 *                            also be passed to the handleDataValueFunc
 *                            whenever a data value is parsed from a
 *                            Packet.
```

```
 *
 * @return -1 if @c parser is NULL.
 * @return -2 if @c parserType is invalid.
 * @return 0 on success.
 */
int
THINKGEAR_initParser( ThinkGearStreamParser *parser, unsigned char parserType,
                      void (*handleDataValueFunc)(
                          unsigned char extendedCodeLevel,
                          unsigned char code, unsigned char numBytes,
                          const unsigned char *value, void *customData),
                      void *customData );
```

THINKGEAR_parseByte()

```
/**
 * @param parser Pointer to an initialized ThinkGearDataParser object.
 * @param byte   The next byte of the data stream.
 *
 * @return -1 if @c parser is NULL.
 * @return -2 if a complete Packet was received, but the checksum failed.
 * @return 0 if the @c byte did not yet complete a Packet.
 * @return 1 if a Packet was received and parsed successfully.
 *
 */
int
THINKGEAR_parseByte( ThinkGearStreamParser *parser, unsigned char byte );
```

Example

Here is an example program using the ThinkGearStreamParser API. It is very similar to the example program described above, simply printing received Data Values to stdout:

```
#include <stdio.h>
#include "ThinkGearStreamParser.h"

/**
 * 1) Function which acts on the value[] bytes of each ThinkGear DataRow as it is received.
 */
void
handleDataValueFunc( unsigned char extendedCodeLevel,
                     unsigned char code,
                     unsigned char valueLength,
                     const unsigned char *value,
                     void *customData ) {

    if( extendedCodeLevel == 0 ) {

        switch( code ) {

            /* [CODE]: HEARTRATE */
            case( 0x03 ):
                printf( "Heart Rate: %d\n", value[0] & 0xFF );
                break;

            /* Other [CODE]s */
            default:
                printf( "EXCODE level: %d CODE: 0x%02X vLength: %d\n",
```

```
                        extendedCodeLevel, code, valueLength );
                printf( "Data value(s):" );
                for( i=0; i<valueLength; i++ ) printf( " %02X", value[i] & 0xFF );
                printf( "\n" );
            }
        }
    }

    /**
     * Program which reads ThinkGear Data Values from a COM port.
     */
    int
    main( int argc, char **argv ) {

        /* 2) Initialize ThinkGear stream parser */
        ThinkGearStreamParser parser;
        THINKGEAR_initParser( &parser, PARSER_TYPE_PACKETS,
                              handleDataValueFunc, NULL );

        /* TODO: Initialize 'stream' here to read from a serial data
         * stream, or whatever stream source is appropriate for your
         * application.  See documentation for "Serial I/O" for your
         * platform for details.
         */
        FILE *stream = fopen( "COM4", "r" );

        /* 3) Stuff each byte from the stream into the parser.  Every time
         *    a Data Value is received, handleDataValueFunc() is called.
         */
        unsigned char streamByte;
        while( 1 ) {
            fread( &streamByte, 1, stream );
            THINKGEAR_parseByte( &parser, streamByte );
        }
    }
```

A few things to note:

- The `handleDataValueFunc()` callback should be implemented to execute quickly, so as not to block the thread which is reading from the data stream. A more robust (and useful) program would probably spin off the thread which reads from the data stream and calls `handleDataValueFunc()`, and define `handleDataValueFunc()` to simply save the Data Values it receives, while the main thread actually uses the saved values for displaying to screen, controlling a game, etc. Threading is outside the scope of this manual.
- The code for opening a serial communication port data stream for reading varies by operating system and platform. Typically, it is very similar to opening a normal file for reading. Serial communication is outside the scope of this manual, so please consult the documentation for "Serial I/O" for your platform for details. As an alternative, you may use the ThinkGear Communications Driver (TGCD) API, which can take care of opening and reading from serial I/O streams on some platforms for you. Use of that interface is described in the developer_tools_2.1_development_guide and TGCD API documentation.
- Most error handling has been omitted from the above code for clarity. A properly written program should check all error codes returned by functions. Please consult the `ThinkGearStreamParser.h` header file for details about function parameters and return values.