TGAM1 Communications Protocol

September 29, 2014

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 wearers' brainwaves. It includes the sensor that touches the forehead, the contact and reference points located on the ear pad, and the onboard chip that processes all of the data and provides this data to software and applications in digital form. Both the raw brainwaves and the eSense Meters (Attention and Meditation) are calculated on the ThinkGear chip.

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

- How to **parse** the serial data stream of bytes to reconstruct the various types of brainwave data sent by the ThinkGear
- How to **interpret and use** the various types of brainwave data that are sent from the ThinkGear (including Attention, Meditation, and signal quality data) in a BCI application
- How to **send** reconfiguration Command Bytes to the ThinkGear, for on-the-fly customization of the module's behavior and output

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

The ThinkGear Packets chapter describes the ThinkGear Packet format used to deliver the ThinkGear Data Values.

The ThinkGear Command Bytes chapter is for advanced users and covers how to send Command Bytes to the ThinkGear in order to customize its configuration (change baud rate, enable/disable certain Data Value outputs, etc).

ThinkGear Data Values

POOR_SIGNAL Quality

This unsigned one-byte integer value describes how poor the signal measured by the ThinkGear is. It ranges in value from 0 to 200. Any non-zero value indicates that some sort of noise contamination is detected. The higher the number, the more noise is detected. A value of 200 has a special meaning, specifically that the ThinkGear contacts are not touching the user's 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:

- Sensor, ground, or reference contacts not being on a person's head (i.e. when nobody is wearing the ThinkGear).
- Poor contact of the sensor, ground, or reference contacts to a person's skin (i.e. hair in the way, or headset which does not properly fit a person's head, or headset not properly placed on the head).
- Excessive motion of the wearer (i.e. moving head or body excessively, jostling the headset).
- Excessive environmental electrostatic noise (some environments have strong electric signals or static electricity buildup in the person wearing the sensor).
- Excessive non-EEG biometric noise (i.e. EMG, EKG/ECG, EOG, etc)

A certain amount of noise is unavoidable in normal usage of ThinkGear, and both NeuroSky's filtering technology and eSense™ algorithm have been designed to detect, correct, compensate for, account for, and tolerate many types of non-EEG noise. Most typical users who are only interested in using the eSense values, such as Attention and Meditation, do not need to worry too much about the `POOR_SIGNAL` Quality value, except to note that the Attention and Meditation values will not be updated while `POOR_SIGNAL` is detected. 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.

Poor Quality Detailed Definitions & Integrations

Poor Quality Flags and Values

The table below lists all the characteristic of the brainwaves that are being monitored by the *product* and the associated flag values.

Wave Characteristic	Value
Signal Flatness	25
Signal Excessiveness	26
Power Ratio	27
Off-Head Detection	29

Multiple flags can be notified at the same time; for example Poor Quality = 51 means the brainwave failed both the flatness and excessiveness criteria. The flag values were chosen to make the sum of every combination a unique number.

One additional value to take note of is 200. This value only occurs after there are 4 seconds of off-head.

Poor Quality Interpretations

Signal flatness: Checks the amount of EEG signal in the sensed data. Flatness will be flagged if the EEG signal is lacking. This is usually flagged when there is an extreme amount of motion that causes saturations when the headset is worn.

Signal excessiveness: Checks the amount of noise in the signal. This is usually flagged when there is an extreme amount of EMGs or headset movement.

Power Ratio: This is a check done in the frequency domain. This is usually flagged when there is excessive periodic environmental noise in the signal.

Off-Head Detection: Checks if the headset is actually on a head. This can be triggered if the headset is not worn properly or not worn at all.

Poor Quality Behavior

The Poor Quality value output is derived by applying several quality matrices on the measured EEG signal. It is a number from 0 to 200 with 0 being the best quality, and 200 the worst. When the signal quality is poor, the calculated eSense Attention and Meditation values are not accurate. An extended period of poor signal quality can further confuse the adaptive eSense algorithm and can cause inaccuracies long after the signal quality improves. Thus, an automatic reset mechanism is built into the *product* to reset the eSense meters after an extended period of off-head or poor signal quality.

When a *product* is initially powered, it will go through an initialization of the eSense. The eSense values will stay at 0 until the *product* senses 4 seconds of data with a Poor Quality value of 0. After the *product* is initialized, the eSense values will start at 40 and varies according to the user's emotional state after that.

Under regular operation the eSense values are only updated when the Poor Quality value is 0. When the Poor Quality value is nonzero, the eSense value is not updated and stays at the previous known good value. When the Poor Quality value exceeds 50 for 7 consecutive seconds, the eSense will reset to zero. The *product* will go back to the initialization step and look for 4 seconds of Poor Quality value of 0 before getting back into normal operation.

When the off-head (value = 29) happens for 4 consecutive seconds, the eSense will reset to zero and the Poor Quality value will become 200 for this special case. Once again, the *product* will go back to the initialization step and look for 4 seconds of Poor Quality value of 0 before getting back into normal operation.

Poor Quality Integration

Since the eSense values are closely related to the Poor Quality value, the poor quality value can give you extra information on why the *product* (headset) is malfunctioning. The signal quality can be used to notify the user the headset is not being worn or used correctly. For example, if the signal flatness is flagged, the application can notify the user to check if the headset is loose or reduce his movement.

Another example of using the signal quality is to make the application stop as soon as the headset is taken off. Currently the eSense meter does not reset to zero, until it collects 4 consecutive seconds of off-head. Before that, the eSense values just freezes. You can design your application to react to off-head condition more quickly by checking all the Poor Quality values that includes the value for off-head (value=29), which are: 29, 54, 55, 56, 80, 81, 82, and 107. After one or two consecutive values of the list above, the application can take action to stop the game before the eSense drops down to zero.

eSense™ Meters

For all the different types of eSenses (i.e. Attention, Meditation), the meter value is reported on a relative eSense scale of 1 to 100. On this scale, a value between 40 to 60 at any given moment in time is considered "neutral", and is similar in notion to "baselines" that are established in conventional EEG measurement techniques (though the method for determining a ThinkGear baseline is proprietary and may differ from conventional EEG). A value from 60 to 80 is considered "slightly elevated", and may be interpreted as levels being possibly higher than normal (levels of Attention or Meditation that may be higher than normal for a given person). Values from 80 to 100 are considered "elevated", meaning they are strongly indicative of heightened levels of that eSense.

Similarly, on the other end of the scale, a value between 20 to 40 indicates "reduced" levels of the eSense, while a value between 1 to 20 indicates "strongly lowered" levels of the eSense. These levels may indicate states of distraction, agitation, or abnormality, according to the opposite of each eSense.

An eSense meter value of 0 is a special value indicating the ThinkGear is unable to calculate an eSense level with a reasonable amount of reliability. This may be (and usually is) due to excessive noise as described in the `POOR_SIGNAL` Quality section above.

The reason for the somewhat wide ranges for each interpretation is that some parts of the eSense algorithm are dynamically learning, and at times employ some "slow-adaptive" algorithms to adjust to natural fluctuations and trends of each user, accounting for and compensating for the fact that EEG in the human brain is subject to normal ranges of variance and fluctuation. This is part of the reason why ThinkGear sensors are able to operate on a wide range of individuals under an extremely wide range of personal and environmental conditions while still giving good accuracy and reliability. Developers are encouraged to further interpret and adapt these guideline ranges to be fine-tuned for their application (as one example, an application could disregard values below 60 and only react to values between 60-100, interpreting them as the onset of heightened attention levels).

ATTENTION eSense

This unsigned one-byte value reports the current eSense Attention meter of the user, which indicates the intensity of a user's level of mental "focus" or "attention", such as that which occurs during intense concentration and directed (but stable) mental activity. Its value ranges from 0 to 100. Distractions, wandering thoughts, lack of focus, or anxiety may lower the Attention meter levels. See eSense\texttrademark Meters above for details about interpreting eSense levels in general.

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

MEDITATION eSense

This unsigned one-byte value reports the current eSense Meditation meter of the user, which indicates the level of a user's mental "calmness" or "relaxation". Its value ranges from 0 to 100. Note that Meditation is a measure of a person's **mental** levels, not **physical** levels, so simply relaxing all the muscles of the body may not immediately result in a heightened Meditation level. However, for most people in most normal circumstances, relaxing the body often helps the mind to relax as well. Meditation is related to reduced activity by the active mental processes in the brain, and it has long been an observed effect that closing one's eyes turns off the mental activities which process images from the eyes, so closing the eyes is often an effective method for increasing the Meditation meter level. Distractions, wandering thoughts, anxiety, agitation, and sensory stimuli may lower the Meditation meter levels. See "eSense Meters" above for details about interpreting eSense levels in general.

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.

Each ThinkGear model reports its raw wave information in only certain areas of the full -32768 to 32767 range. For example, TGAM1 reports raw waves that fall between approximately -2048 to 2047.

By default, output of this Data Value is disabled. When enabled, the TGAM1 outputs this value 512 times a second, or approximately once every 2ms.

Note: Because of the high rate at which this value is output, and the number of bytes of data involved, it is only possible to output the `16-bit RAW Wave Value` on the serial communication stream at 57,600 baud and above.

ASIC_EEG_POWER_INT

This Data Value represents the current magnitude of 8 commonly-recognized types of EEG (brain-waves). This Data Value is output as a series of eight 3-byte unsigned integers in big-endian format. The eight EEG powers are output in the following order: delta (0.5 - 2.75Hz), theta (3.5 - 6.75Hz), low-alpha (7.5 - 9.25Hz), high-alpha (10 - 11.75Hz), low-beta (13 - 16.75Hz), high-beta

(18 - 29.75Hz), low-gamma (31 - 39.75Hz), and mid-gamma (41 - 49.75Hz). These values have no units and therefore are only meaningful compared to each other and to themselves, to consider relative quantity and temporal fluctuations.

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

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 includes new 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 it 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 (0-255)
0	0x04	–	ATTENTION eSense (0 to 100)
0	0x05	–	MEDITATION eSense (0 to 100)

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	0x81	32	EEG_POWER: eight big-endian 4-byte IEEE 754 floating point values representing delta, theta, low-alpha high-alpha, low-beta, high-beta, low-gamma, and mid-gamma EEG band power values
0	0x83	24	ASIC_EEG_POWER: eight big-endian 3-byte unsigned integer values representing delta, theta, low-alpha high-alpha, low-beta, high-beta, low-gamma, and mid-gamma EEG band power values

(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.

Whenever a ThinkGear module is powered on, it will always start in a standard configuration in which only some of the Data Values listed above will be output by default. To enable or disable the types of Data Values output by the ThinkGear, refer to the advanced chapter ThinkGear Command Bytes.

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]: 0x20  // [PLENGTH] (payload length) of 32 bytes
[ 3]: 0x02  // [POOR_SIGNAL] Quality
[ 4]: 0x00  // No poor signal detected (0/200)
[ 5]: 0x83  // [ASIC_EEG_POWER_INT]
[ 6]: 0x18  // [VLENGTH] 24 bytes
[ 7]: 0x00  // (1/3) Begin Delta bytes
[ 8]: 0x00  // (2/3)
[ 9]: 0x94  // (3/3) End Delta bytes
[10]: 0x00  // (1/3) Begin Theta bytes
[11]: 0x00  // (2/3)
[12]: 0x42  // (3/3) End Theta bytes
[13]: 0x00  // (1/3) Begin Low-alpha bytes
[14]: 0x00  // (2/3)
[15]: 0x0B  // (3/3) End Low-alpha bytes
[16]: 0x00  // (1/3) Begin High-alpha bytes
[17]: 0x00  // (2/3)
[18]: 0x64  // (3/3) End High-alpha bytes
[19]: 0x00  // (1/3) Begin Low-beta bytes
[20]: 0x00  // (2/3)
[21]: 0x4D  // (3/3) End Low-beta bytes
[22]: 0x00  // (1/3) Begin High-beta bytes
[23]: 0x00  // (2/3)
[24]: 0x3D  // (3/3) End High-beta bytes
[25]: 0x00  // (1/3) Begin Low-gamma bytes
[26]: 0x00  // (2/3)
[27]: 0x07  // (3/3) End Low-gamma bytes
[28]: 0x00  // (1/3) Begin Mid-gamma bytes
[29]: 0x00  // (2/3)
[30]: 0x05  // (3/3) End Mid-gamma bytes
[31]: 0x04  // [ATTENTION] eSense
[32]: 0x0D  // eSense Attention level of 13
[33]: 0x05  // [MEDITATION] eSense
[34]: 0x3D  // eSense Meditation level of 61
[35]: 0x34  // [CHKSUM] (1's comp inverse of 8-bit Payload sum of 0x)
```

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]`.
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 );

        /* Compute [PAYLOAD...] chksum */
        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 [PAYLOAD...] chksum against [CKSUM] */
        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, 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_BATTERY_CODE         0x01
#define PARSER_POOR_SIGNAL_CODE     0x02
#define PARSER_ATTENTION_CODE       0x04
#define PARSER_MEDITATION_CODE      0x05
#define PARSER_RAW_CODE             0x80
```

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]: ATTENTION eSense */
            case( 0x04 ):
                printf( "Attention Level: %d\n", value[0] & 0xFF );
                break;

            /* [CODE]: MEDITATION eSense */
            case( 0x05 ):
                printf( "Meditation Level: %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_2.2_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.

ThinkGear Command Bytes

Upon power-on, TGAM1s always start in their factory-programmed default state. After power-on however, the TGAM1 can be sent Command Bytes in order to change its configuration, such as switching to 57.6k baud and enabling raw wave values output. ThinkGear Command Bytes are intended as an advanced feature for performing some customization of the behavior ThinkGear hardware.

ThinkGear Command Bytes are sent to the ThinkGear hardware through the same UART interface used to receive Packet bytes. A Command Byte is a single byte (8-bit) value with certain bits set. This section describes which bits to set in order to change the configuration of a ThinkGear module.

Note that after being power cycled, the TGAM1 returns to its factory default settings.

Also note that, before an application sends any command bytes to the ThinkGear, it should first make sure it has read at least one complete, valid Packet from the ThinkGear, to ensure it sends Command Bytes at the proper baudrate. Sending Command Bytes at the wrong baudrate may result in the ThinkGear being rendered inoperable until it is power-cycled (reset back to default configuration).

Command Byte Syntax

A Command Byte is formed by 8 bits, each of which are either set or unset. The lowest (least significant) four bits are used to control modes, such as 9600 vs. 57.6k baud mode, attention output enabled or disabled mode, etc. The upper (most significant) four bits, known as the "Command Page", define the meaning of the lower four bits.

For example, a Command Byte of `0x0E` has the bit pattern of `0000 1110`. The upper (most significant) four bits are `0000`, which refers to Command Page zero.

TGAM1 Command Byte Table

Important: TGAM1 only recognizes Command Bytes from below. Any other Command Bytes may put it into an inoperable state until it is power cycled.

```
      (0x0_):  STANDARD/ASIC CONFIG COMMANDS*
      00000000 (0x00): 9600 baud, normal output mode (POOR_SIGNAL, ATTENTION, MEDITATION,
ASIC_EEG_POWER_INT)
      00000001 (0x01): 1200 baud, normal output mode (POOR_SIGNAL, ATTENTION, MEDITATION,
ASIC_EEG_POWER_INT)
      00000010 (0x02): 57.6k baud, normal+raw output mode (POOR_SIGNAL, ATTENTION, MEDITATION,
ASIC_EEG_POWER_INT, RAW)
      00000011 (0x03): 57.6k baud, FFT output mode (FFT)
```

*: After sending this Page byte, the application itself must change itself to begin communicating at the new requested baud rate, and then wait at that new requested baud rate for a complete, valid

Packet to be received from the TGAM1 before attempting to send any other command bytes to the TGAM1. Sending another command byte before performing this check may put the TGAM1 into an indeterminate (and inoperable) state until it is power-cycled.