TDIG CANbus High Level Protocol

Version 1.5 (10.13.04)

Justin Kennington

Communication between the MCU on TDIG and the CANbus host PC comprises six transactions:

```
DATA_TO_PC
SET_CONTROL
CONFIGURE_TDC
GET_STATUS
CHANGE_MCU_PROGRAM
THRESHOLD
DEBUG
```

The first of these, DATA_TO_PC, is a one-way communication initiated by the MCU.

The rest of the transactions are command sequences of two or more messages. SET_CONTROL, GET_STATUS, and DEBUG are initiated by a request for data from the PC, and completed with a single response from the MCU (containing the requested data). CONFIGURE_TDC and CHANGE_MCU_PROGRAM are more complicated, because more data than the 8 byte limit of each CAN message are required. In these cases, an 8 byte data packet is sent (by the PC) and acknowledged (by the MCU) before another 8 byte packet is sent. This continues until all data have been sent.

CANbus Packet ID

Each CANbus packet contains an 11-bit identifier field. The packet ID encapsulates three pieces of data: Message type, board number, and TDC number. The data is stored as follows:

```
MsgID[10:6] = Message type
MsgID[5:4]  = TDC Number
MsgID[3]    = Extra bit (will be specified per message type if applicable)
MsgID[2:0]  = Board number
```

Message type

Message type is a 5-bit field that specifies the type of command that the message is related to and whether the packet is a command from the control system (downstream) or response from a TDIG board (upstream). Table 1 shows the available message types and the corresponding codes.

Command	**Downstream Code**	**Upstream Code**	**Function**
DATA_TO_PC	N/A	00000	TDC data upload
SET_CONTROL	00100	00010	Apply new TDC control word
CONFIGURE_TDC	01000	00110	Apply new TDC configuration
GET_STATUS	01100	01010	Get TDC or TDIG status
CHANGE_MCU_PROGRAM	10000	01110	Change MCU firmware
DEBUG	10100	10010	Multifunction debugging code
THRESHOLD	11000	10110	Sets new discriminator threshold
ERROR	N/A	11110	Response to unrecognized message

Table 1

TDC Number

Some message types apply to a specific TDC on a board. These types include SET_CONTROL, CONFIGURE_TDC, and GET_STATUS. For these messages types, the TDC number is included in the message ID to save bits in the payload. The TDCs are numbered 1 to 4 and encoded as follows:

```
MsgID[5:4] = 00  => TDC1, Leading edge Hit[7:0]
MsgID[5:4] = 01  => TDC2, Leading edge Hit[15:8]
MsgID[5:4] = 10  => TDC3, Leading edge Hit[23:16]
MsgID[5:4] = 11  => TDC4, Trailing edge Hit[23:0]
```

Board Number

Each tray contains one TCPU card and eight TDIG cards numbered 0 to 7. Board Number is a 3-bit field indicating which of the eight boards a packet is to or from. The most upstream (closest to TCPU) board is number zero, while the most downstream (farthest from TCPU) is number seven. The board number is binary encoded. TCPU does not need a unique ID because it receives all messages transmitted by TDIG cards.

```
MsgID[3:0] = 0000  => Board 0 (Far upstream)
MsgID[3:0] = 0001  => Board 1
...
MsgID[3:0] = 0111  => Board 7 (Far downstream)
```

DATA_TO_PC

This command is used to push data upstream from the MCU. These messages only flow one direction, and the MCU assumes that the messages are received.

```
MsgID[10:6] = 00000
MsgID[5:4]  = 00
MsgID[3]    = 0
MsgID[2:0]  = Board number
```

Payload:

For bench testing and debugging, TDC data can be sent via the CANbus. Each TDC word is four bytes, so the MCU will typically pack two words per CAN message. However, if only one word is available, the MCU will send a CAN message with a single word, to avoid stalls. The resulting two payload formats are:

```
dddddddd dddddddd dddddddd dddddddd
  [four byte data packet from TDC]
```

```
dddddddd dddddddd dddddddd dddddddd  dddddddd dddddddd dddddddd dddddddd
[1st four byte data packet from TDC]  [2nd four byte data packet from TDC]
```

Information identifying whether the packet is a group header, TDC header, etc. is included in the data word itself. No guarantee is made regarding which TDC either data word will come from. The TDC ID is included in each data word, so the upstream entity will be able to keep this straight without use of the TDC number field in the message ID.

SET_CONTROL

This command sequence is used to set the 40-bit control word used by the TDC. See TDC data sheet section 17.6 for details on the control word. When the MCU receives a SET_CONTROL packet from upstream, it immediately programs the new control word into the TDC, then responds with a confirmation message.

Message ID

For commands issued to TDIG:

```
MsgID[10:6] = 00100
MsgID[5:4]  = TDC number
MsgID[3]    = ‘All TDCs’ bit
MsgID[2:0]  = Board number
```

For responses by TDIG:

```
MsgID[10:6] = 00010
MsgID[5:4]  = TDC number
MsgID[3]    = ‘All TDCs’ bit
MsgID[2:0]  = Board number
```

In a response packet, `MsgID[10:6]` are set appropriately, and the remaining bits are echoes of the command packet.

‘All TDCs’ bit

It is convenient sometimes to apply the same control word to all four TDCs at once. In this case, the commanding node should send the SET_CONTROL command with the ‘All TDCs’ bit set. The new control word will be applied to all four TDCs.

Payload:

The payload format for SET_CONTROL packets is the 40-bit (5 byte) new control word. The MSB (bit 39) is in payload word 0; the LSB (bit 0) is in payload word 5.

```
cccccccc cccccccc cccccccc cccccccc cccccccc
           [40-bit control word]
```

Once the TDC has been updated, the MCU returns a response packet for confirmation that the new control word has been copied to the TDC. The payload of the response packet matches the payload of the command packet.

CONFIGURE_TDC

This instruction uses a command and response technique to allow the PC and MCU to coordinate as the PC updates the MCU's local copy of the TDC configuration settings, and then the MCU reprograms the TDC with the updated configuration. For each command that the PC issues to the MCU, the MCU performs a particular action and then issues a confirmation response to the PC.

For commands issued by the PC to the MCU:

```
MsgID[10:6] = 01000
MsgID[5:4]  = TDC number
MsgID[3]    = Unused (don't care)
MsgID[2:0]  = Board number
```

For responses by the MCU to the PC:

```
MsgID[10:6] = 00110
MsgID[5:4]  = TDC number
MsgID[3]    = Unused (don't care)
MsgID[2:0]  = Board number
```

Payload:

In both cases, the first byte of the payload is a descriptor field that gives three pieces of information about the message and its contents. The descriptor field is laid out as follows:

```
DES[3:0] = packet_number
DES[4]   = Unused (don't care)
DES[5]   = error
DES[7:6] = sub_instruction
```

Each of these fields is further defined herein:

I. Packet number:

Each payload can carry 7 bytes of configuration data (8 byte payload – 1 byte descriptor). Each configuration string is 647 bits = 81 bytes. Therefore, the configuration string will be sent in twelve packets of up to seven bytes each. The first eleven packets will contain seven bytes of data, while the final (12th) packet will contain four bytes. Data beyond the fourth byte in the twelfth packet will be ignored. Each packet will be numbered sequentially, and the data will be written to MCU data memory sequentially based on packet numbers. The first data packet sent will have packet_number = 1 and will contain seven bytes of MSB's. The final data packet will have packet_number = 12 and contain bytes 78 – 81 (LSB's). The configuration string is left justified with MSB's first.

647 bits is one bit shy of 81 bytes. A 0 should be appended to the *end* (LSB) of the configuration data (in packet 12), and the very first bit sent (in packet 1) should be the MSB (parity) of the configuration data. Correctly setting the parity bit is optional, as the MCU will calculate the correct parity bit regardless.

As defined by the sub_instruction field, not every payload in the configure_TDC sequence will carry configuration data. For non-data payloads, packet_number should be given a value of zero.

II. Error:

This bit is set by the MCU to indicate that an error has occurred. If the MCU detects an error, it will **perform no action**, and instead will send back the message payload with the error bit set. The MCU will set the error bit under the following conditions:

- The MCU receives the start sub_instruction but is not ready to begin CONFIGURE_TDC
- The MCU detects a mismatch between sub_instruction and packet_number (*i.e.* non-zero packet_number with non-data sub_instruction, or zero packet_number with data sub_instruction)
- The MCU receives an out-of-order data packet (*e.g.* a packet_number is skipped)
- The MCU receives a CONFIGURE_TDC message for a particular TDC between start and program sub_instructions for another TDC. That is, while configuring a TDC, the entire configuration process must be complete before configuring another TDC.
- The MCU receives a config_end sub_instruction but the MCU does not have a full configuration.
- The MCU receives a program sub_instruction but the MCU has not received a config_end sub_instruction.

If the PC receives a packet with the error bit set, the safe response is to restart the configuration by sending a new start sub_instruction and re-sending all data. However, the MCU will also maintain its state following the most recent packet received without error. Continuing from that point by sending the appropriate next packet is guaranteed to work.

III. Sub-instruction:

The sub-instruction defines the command as 1 of 4 subcommands in the configure_TDC sequence. The four sub-instructions are:

Start: instructs the MCU to initialize the CONFIGURE_TDC sequence. Data will follow in later packets. Sending the start sub_instruction (for the TDC currently being configured) will always re-start the configuration sequence.

```
DES[7:6] = 00   (sub_instruction)
DES[3:0] = 0000 (packet_number)
```

Data: a data-containing message. As described above, the payload will contain the descriptor and four or seven bytes of data to write to the configuration.

```
DES[7:6] = 01       (sub_instruction)
0001 < DES[3:0] < 1100 (packet_number)
```

Config_end: advises the MCU that the PC is finished sending configuration data. MCU should check to see that it has received a full configuration.

```
DES[7:6] = 10   (sub_instruction)
DES[3:0] = 0000 (packet_number)
```

Program: instructs the MCU to load the updated configuration settings into the TDC.

```
DES[7:6] = 11   (sub_instruction)
DES[3:0] = 0000 (packet_number)
```

The program command initiates a full reset of both TDCs. It disables the clock inputs, reprograms the configuration bits, then performs a full startup routine. If the user desires to configure both TDCs, it is possible to execute a config_TDC sequence for both TDCs from start through config_end, and then send a program command. This way, both TDCs will be updated simultaneously.

Reapply Configuration:

To reapply the current configuration to the TDCs, send a START command immediately followed by a PROGRAM command. This will cause the MCU to reset and configure both TDCs with the configuration bits stored in MCU memory (previously applied configuration, or default configuration if no previous change has been made). The sequence START-PROGRAM may be transmitted at any time to reapply the configuration.

MCU to PC packets (Responses):

The MCU has a response to each packet sent by the PC, and the PC should wait for the appropriate response to a packet before sending the next packet. The MCU responds to PC packets by changing the Message Sub-ID and echoing back the payload, unchanged.

For messages sent by the PC to the MCU:

```
MsgID[10:6] = 01000
MsgID[5:0]  = Echo
```

For responses by the MCU to the PC:

```
MsgID[10:6]  = 00110
MsgID[5:0]   = Echo
```

Each response by the MCU indicates that a particular action, dependent on the sub_instruction, has been taken by the MCU. The actions taken by the MCU, along with the meaning of the response, are as follows:

Start: Response indicates that the MCU has received the start command and is prepared to begin writing configuration information to MCU data memory.

Data: Response indicates that the MCU has received the configuration information, written it to data memory, and is ready to receive following information.

Config_end: Response indicates that the MCU has received a full 647 bit configuration string and is prepared to write the string to the TDC.

Program: Response indicates that the MCU has written the new configuration string to the TDC. Indicates that the old configuration has been reapplied if the program command immediately follows a start command.

If the MCU detects an error according to the conditions described above, a message is sent with the MCU-to-PC message ID, the error bit set, and the payload otherwise unchanged. None of the above-described actions are performed, and the PC should restart the configuration sequence.

Example

The following is an example of a correct CONFIGURE_TDC sequence (showing only arbitration and data fields) for configuring board 5, TDC3. The first 11 bits are the MsgID. Everything following is the payload.

```
New config data = 5C 00 16 00 00 00 25 C0 20 08 0B FF FE FB  ...
       ... 00 00 11 24 F0 F0 F0 00 00 00 00 01 D5 30 00 03 FF A2   (Note: Trailing 0)

‘01000 10 0 101 00000000’b
; start CONFIGURE_TDC for board 5 (101), TDC3 (10)

                                                                       ‘00110 10 0 101 00000000’b
                                                                ; acknowledge start CONFIGURE_TDC

‘0100 0000111 01000001’b ‘5C 00 16 00 00 00 25’h
; packet 1 with 7 bytes

                                                 ‘0011 0000111 01000001’b ‘5C 00 16 00 00 00 25’h
                                                                           ; acknowledge packet 1

‘0100 0000111 01000010’b ‘C0 20 08 0B FF FE FB’h
; packet 2 with 7 bytes

                                                 ‘0011 0000111 01000001’b ‘C0 20 08 0B FF FE FB’h
                                                                           ; acknowledge packet 2

...
; packets 3 - 9

                                                                                              ...
                                                                      ; acknowledge packets 3 – 9

‘0100 0000111 01001010’b ’00 00 11 24 F0 F0 F0’h
; packet 10 with 7 bytes

                                                 ‘0011 0000111 01001010’b ’00 00 11 24 F0 F0 F0’h
                                                                          ; acknowledge packet 10

‘0100 0000111 01001011’b ’00 00 00 00 01 D5 30’h
; packet 11 with 7 bytes

                                                 ‘0011 0000111 01001011’b ’00 00 00 00 01 D5 30’h
                                                                          ; acknowledge packet 11

‘0100 0000111 01001100’b ’00 03 FF A2’h
; packet 12 with 4 bytes

                                                          ‘0011 0000111 01001100’b ’00 03 FF A2’h
                                                                           ;acknowledge packet 12

‘0100 0000111 10000000’b
; config_end

                                                                         ‘0011 0000111 10000000’b
                                                                         ; acknowledge config_end

‘0100 0000111 11000000’b
; program TDC

                                                                         ‘0011 0000111 11000000’b
                                               ; acknowledge program TDC (Finished with sequence)
```

GET_STATUS

This function also has a command and response structure, though it is far simpler than CONFIGURE_TDC. The code sequence comprises a single command-response pair. The command tells the MCU to return the status of one of the four TDCs or the TDIG board itself (information from temperature sensors, MCU status, etc). The format of TDIG status is TBD.

For the PC Command:

```
MsgID[10:6] = 01100
MsgID[5:4]  = TDC Number
MsgID[3]    = TDIG status bit
MsgID[2:0]  = Board number
```

For the MCU response:

```
MsgID[10:6] = 01010
MsgID[5:4]  = TDC Number
MsgID[3]    = TDIG status bit
MsgID[2:0]  = Board number
```

TDIG status bit

If this bit is set to 1, then the request is for status of TDIG itself, not for a TDC. If this bit is set to zero, then the board number field functions as normal, and determines which TDC's status to return.

Payload

PC to MCU (Command):

There is no payload for a status request command. It is a zero-length message. The Message ID determines whether the request is for TDC1, TDC2, TDC3, TDC4, or for TDIG itself.

MCU to PC (Response):

The response payload for TDC status requests is 64 bits including two leading zeros and 62 bits of status information. Each TDC reports 62 bits of status. All 62 available bits are reserved for TDIG status. The response payload will be formatted as follows:

```
00 ssssss ssssssss ssssssss ssssssss ssssssss ssssssss ssssssss ssssssss

s = TDC status[61:0]
```

The response payload for TDIG status requests is 64 bits and includes one 10-bit temperature value for each of four temperature monitors. Two monitors are on TDIG, and two are on TAMP.

```
000000aa aaaaaaaa 000000bb bbbbbbbb 000000cc cccccccc 000000dd dddddddd

a = TDIG temperature readout under PLD
b = TDIG temperature readout at TDC2
c = TAMP temperature readout at board edge (inside tray)
d = TAMP temperature readout at board center (inside tray)
```

The 10-bit temperature word is the output of a 10-bit A/D converter inside the MCU of TDIG. The temperature may be determined by the formula:

$$T(°C) = \frac{\frac{A(decimal)+1}{1024} * 3300mV - 500mV}{10\frac{mV}{°C}}$$

T = Temperature in degrees celcius

A = ADC word, decimal (Ranges from 0 to 1023)

THRESHOLD

This function is used to set the discriminator threshold. TDIG contains an onboard DAC that converts a 12-bit word from the MCU into an analog output voltage that feeds the threshold offset inputs of the discriminator circuits.

For the PC Command:

```
MsgID[10:6] = 11000
MsgID[5:4]  = xx (Threshold affects all TDC’s)
MsgID[3]    = x
MsgID[2:0]  = Board number
```

For the MCU response:

```
MsgID[10:6] = 10110
MsgID[5:4]  = xx
MsgID[3]    = x
MsgID[2:0]  = Board number
```

Payload

PC to MCU (Command):

The payload consists of the 12-bit DAC word that is directly transcribed into the DAC. Four leading zeros will fill out the two byte payload.

```
0000dddd dddddddd

d = DAC word [11:0]
```

The DAC word is related to the threshold voltage by the following equation:

$$V = -0.05 * D + 101.51$$

$$D = -20.003 * V(mV) + 2030.5$$

The value of D returned by this equation is decimal. It must be converted to hexadecimal before being transmitted as the DAC word.

The maximum threshold setting is approximately 100 mV, DAC word = 0x000. The minimum threshold is approximately -100mV, DAC word = 0xFFF. The corresponding DAC word to threshold voltage conversion will also change at that time. Standard threshold level will be approximately 30mV, DAC word = 0x596.

MCU to PC (Response):

The response payload consists of the 10-bit ADC word that is read directly from the ADC. Six leading zeros will fill out the two byte payload.

```
000000aa aaaaaaaa

a = ADC word [9:0]
```

The ADC word can be used to check the value to which the threshold voltage gets set. It is related to the DAC and to the threshold voltage by the following equations:

A = 0.1606*D + 1.5168

V = -0.3113*A + 101.97

A = ADC code word

D = DAC code word

V = Threshold voltage (mV)

The value of A returned by this equation is decimal.

DAC word	DAC Decimal value	Resulting Threshold (mV)	ADC word	ADC Decimal Value
0x000	0	101.4	0x003	3
0x04F	79	97.6	0x00F	15
0x0FF	255	88.8	0x02B	43
0x1FF	511	76	0x053	83
0x2FF	767	63.2	0x07C	124
0x4FF	1279	37.5	0x0CD	205
0x5FF	1535	24.7	0x0F6	246
0x6FF	1791	11.9	0x11F	287
0x7FF	2047	-0.8	0x148	328
0x8FF	2303	-13.5	0x178	376
0x9FF	2559	-26.4	0x19D	413
0xAFF	2815	-39.2	0x1C9	457
0xBFF	3071	-52	0x1EF	495
0xCFF	3327	-64.8	0x218	536
0xDFF	3583	-77.6	0x240	576
0xFFF	4095	-103.3	0x292	658

DEBUG

Instruction space is reserved for a further command/response pair for debugging. The commands will consist of requests for various data and/or calls to various diagnostic functions. These functions might include restarting the TDCs, or simply requesting trigger or timing information. These functions will be used to allow the CAN bus to detect and/or correct system errors. Further definition of these functions will be added later.

For commands issued by the PC to the MCU:

```
MsgID[10:6] = 10100
MsgID[5:4]  = TDC number (where applicable)
MsgID[3]    = Unused (don’t care)
MsgID[2:0]  = Board number
```

For responses by the MCU to the PC:

```
MsgID[10:6] = 10010
MsgID[5:4]  = TDC number
MsgID[3]    = Unused (don’t care)
MsgID[2:0]  = Board number
```

MCU Restart

The DEBUG command can be used to restart the MCU. This process first shuts off the TDC clocks, then starts executing the MCU’s initialization code. This is as close to a power-off reset as one can get without actually turning off the power. *The TDCs will be restored to their default configuration.*

Header

Bits [5:3] of the header are ignored and may be set to any value. They will be echoed in the response.

Payload

The payload is an arbitrary 8 digit code to prevent accidental restarts:

```
0100 0101 0110 1001 0011 0011 0001 0100

HEX: 0x45 69 33 14
```

PLD Reset

This command is used to reset the PLD’s internal state machines and flush data FIFOs.

Header

Bits [5:3] of the header are ignored and may be set to any value. They will be echoed in the response.

Payload

Like MCU reset, the payload consists of an arbitrary four byte code:

```
0110 1001 1001 0110 1010 0101 0101 1010

HEX: 0x69 96 A5 5A
```

MCU Data Readout Mode

The MCU can read out TDC data in three different modes:

PLD (serial) readout:

Data is read serially from the TDC's by the PLD. The MCU receives this data from the PLD and transmits is up the CANbus. This data will include any words inserted by the PLD, including data separators, geographical words, etc. Note that setting the MCU to this mode will cause any data in the PLD FIFOs to immediately be sent.

JTAG readout:

This is a special function mode used only for major debugging. In this mode, data is read directly from the TDC's by the MCU via JTAG. Note that *the TDC's must be manually configured to allow JTAG readout.*

Silent mode (no readout):

In this mode, no data will be read out via CANbus. This is the default mode selected upon MCU startup.

Header

For serial readout and silent modes, bits [5:3] are ignored. For JTAG readout, bits [5:4] select the TDC desired for readout. Only one TDC may be selected at a time for JTAG readout. Sending a new JTAG readout mode command will allow the user to select a new TDC for JTAG readout.

Payload

The payload consists of two bytes. The first byte is 0xAA to indicate that this is a readout mode select message. The second byte selects one of the three modes.

```
10101010 000000mm

m = mode select bits:

00 -> Silent mode (default)
01 -> Serial PLD readout
10 -> JTAG readout
```

ERROR

When the MCU receives a CAN message that it does not understand, it replies with an error packet. Error conditions include: unrecognized message ID and malformed payload. For example, if a request for status is received and the first data byte is not F0, F3, or FB, an error message is issued. Note that a configure_TDC command received in error will result in a configure_TDC response packet with its error bit set.

```
MsgID[10:6] = 11110
MsgID[5:0]  = Echo
```

CHANGE_MCU_PROGRAM

This command sequence is used to change the program code running on the MCU or PLD. The CHANGE_MCU_PROGRAM sequence is similar to the CONFIGURE_TDC sequence, but with added safety features. The format used for the MsgID is slightly altered to make the transaction more efficient.

For commands issued by the PC to the MCU:

```
MsgID[10:6] = 10000
MsgID[5:3]  = xx
MsgID[2:0]  = Board number
```

For responses by the MCU to the PC:

```
MsgID[10:6] = 01110
MsgID[5:3]  = xx
MsgID[2:0]  = Board number
```

Overview:

The MCU (PIC18LF8720) has 128 kilobytes of internal program memory space, which is easily more than double the amount of program code used in TDIG (currently less than 8 kilobytes). As initially programmed, the program code will reside in the lower half of program memory, beginning at address 0x00000. When updated firmware is applied, the new code will be written to the upper half of program memory (starting at address 0x10000), and once written, execution will jump to that space. Upon reboot, a CAN message will need to be sent to jump execution back to the reprogrammed code.

Data will be programmed in groups of 64 bytes. Every 64 bytes the data will be checksummed, and a Program_64 command will be issued. Once the checksum is verified by the MCU, CAN communication with the MCU will pause while the MCU writes the 64 bytes to flash memory. After writing and verifying 64 bytes, the MCU will confirm that they have been written. This process will continue until all program changes are complete. At that time, a final checksum will be performed for the whole program code. Once verified, the PC will issue a JUMP_PC command. The MCU will acknowledge the JUMP_PC command and jump its program counter to the start of the new code. It is advised to include code at the beginning of the new code that will send a CAN message confirming execution of the new code.

To change the code a second time, the MCU will need to be rebooted to the original code, and the previous updated code will be overwritten. This way, the original code will never be overwritten. It is impossible for this document to promise any features in the newly programmed code. However, **it is strongly advised that the new code not be capable of reprogramming *any* flash memory.**

Payload:

There are several different types of packets involved in MCU reprogramming. Each type has a particular function and format. The type is identified by the sub-instruction field, stored in the descriptor that forms the first byte of all reprogramming packet payloads.

Descriptor:

The first byte of all reprogramming payloads is a descriptor field that gives information about the command type and any errors that have occurred. The descriptor field is laid out as follows:

```
DES[7]   = error
DES[6]   = error (where used)
DES[6:4] = not used (don’t care)
DES[3:0] = sub_instruction
```

The sub_instruction field is used to determine which of the reprogramming packet types commands the packet represents.

Error bits:

An error bit is set by the MCU to indicate that an error has occurred. If the MCU detects an error, it will **perform no further action**, and will send back the message payload with the appropriate error bit set. The meaning of the error bits is different for each sub_instruction, and is described below. If the PC receives a packet with the error bit set, the safe response is to restart the current 64 byte block by sending a new start sub_instruction and re-sending all data.

MCU to PC packets (Responses):

The MCU has a response to each packet sent by the PC, and the PC should wait for the appropriate response to a packet before sending the next packet. The MCU responds to PC packets by changing the Message ID. That is,

```
MsgID[10:6]  = 01110
MsgID[5:0]   = Echo (same as sent by PC)
```

Each error-free response by the MCU indicates that a particular action has been taken, dependent on the sub_instruction. The actions taken by the MCU are described below.

Start Packet:

```
DES[3:0] = 0000
```

Format:

The start command’s payload contains only the descriptor.

```
e000 0000

e = error bit
```

Effect:

The start command instructs the MCU to ready itself for the transfer of 64 bytes of new program data. Data will follow in later packets. Sending the start sub_instruction will always re-start the 64 byte transfer. Error-free response indicates that the MCU has received the start command and is prepared to begin receiving new program data.

A start command may be issued at any time, and will always any current reprogramming sequence.

Errors:

An error bit will be set if the MCU is not ready to accept new program data.

Data Packet:

Format:

```
DES[3:0] = 0001 (data)
```

Actual program data is sent in data packets, each of which contains four bytes of new program data. In addition to the descriptor and program data, each data packet contains a 16-bit start address. The start address is the address that the first data word is written to. The next data word is written to start address + 1. The first data packet in a 64 byte block must have a start address of a 64 byte block. Therefore the 6 LSB's will be zero (*e.g.*, 0x000, 0x040, 0x080, *etc*.).

```
e000 0001 aaaaaaaa aaaaaaaa 1ddddddd 2ddddddd 3ddddddd 4ddddddd
```

```
e = error bit
a = start address (MSB first)
1d = first data word
2d = second data word
```

Effect:

When the MCU receives a data packet, it will write the data words to data memory. Error-free response indicates that the MCU has received the new program data, written it to data memory, and is ready to receive following information.

The PIC18's 128 kilobyte address space is addressed with 17 bits. Only 16 bits of address are sent, and a leading 1 is appended by the MCU before writing data. This assures that only the top half of memory can be written with new code. Sending data with a start address of 0x0000 will result in that data being written to program memory at 0x10000. Of course, safety mechanisms like this **cannot** be guaranteed in the new code.

Errors:

A data packet must be preceded by a start instruction or another data packet. The error bit is set under either of two conditions:

1. The preceding packet was a start instruction, but the data packet address is not the start of a 64-byte sector. A first data packet will always have a start address that is the beginning of a 64 byte sector (that is, the 6 LSB's will be zero).
2. A non-first data packet has a start address not equal to the previous data packet start address plus four.

Checksum_64 Packet:

Format:

```
DES[3:0] = 0010 (chksum_64)
```

The message payload contains the descriptor, the start address of the just-transferred 64-byte block, and the 8-bit checksum of that 64-byte block.

```
e000 0010 aaaaaaaa aaaaaaaa cccccccc

e = error bit
a = 16-bit address for START of new program data
c = checksum of previous 64 program bytes
```

Effect:

The Checksum_64 command is used to confirm the successful transmission of one 64-byte block. It checks the program code that is temporarily stored in data memory, *before* the code has been written to flash program memory. Therefore **this message is only valid following a completed 64-byte data transaction**. It is also ok to send this command following a Program_64 programming error.

When the MCU receives this command, it calculates a checksum from the 64-byte temporary program array in data memory. It then compares the calculated value to that transmitted in the CAN message. If and only if there is a match, the MCU will allow a following Program_64 command (see below) to execute.

Errors:

Three conditions can cause an error in response to a Checksum_64 packet.

1. The preceding packet was not a Data packet or a Checksum_64 packet
2. The start address provided does not match the expected start address.
3. A checksum mismatch occurs.

In all cases, the error bit will be set. In the case of a checksum mismatch, the calculated checksum will be appended to the returned message, like so:

```
1000 0010 aaaaaaaa aaaaaaaa cccccccc xxxxxxxx

e = error bit (will always be 1 in this case)
a = 16-bit address for START of new program data
c = checksum sent by PC
x = data memory checksum calculated by MCU (mismatch)
```

A packet error (cases 1 and 2) response will not include the calculated checksum. Therefore, it is possible to determine which type of error occurred by checking the length of the returned error packet.

Program_64 Packet:

Format:

```
DES[3:0] = 0011
```

The payload contains the descriptor and a start address.

```
ex00 0010 aaaaaaaa aaaaaaaa

e = address error bit
x = programming error bit
a = 16-bit address for START of new program data
```

Effect:

The Program_64 packet is the final transaction for writing a 64 byte block to program memory. This packet is only accepted if the previous transaction was a successful Checksum_64. When the MCU receives a Program_64 packet following a successful checksum, the 64 bytes of new program code are written to FLASH program memory beginning at the start address included in the Program_64 packet and continuing through this address + 63. After the data has been written to program memory, it will be checked against the copy stored in data memory. An error-free response to this packet indicates that the previous 64 bytes have been successfully written to program memory and verified.

Errors:

The Program_64 response has two error bits. The address error bit is set if the preceding packet was not a Checksum_64, or if the start address provided does not match the expected start address. The start address must match that of the preceding Checksum_64 packet.

The programming error bit is set if there is a mismatch between the program data temporarily stored in data memory (and previously checksum-verified) and the data programmed into Flash program memory. If a programming error is detected, it is ok to resend the corresponding Checksum_64 packet and re-try the Program_64.

Final_chksum Packet:

Format:

```
DES[3:0] = 0100
```

The payload of a Final_chksum packet includes the descriptor, a final address, and an 8-bit checksum that is the 8-bit sum of all program memory data from 0x0000 to the final address. This is a checksum for all of the new program code.

```
e000 0011 aaaaaaaa aaaaaaaa cccccccc

e = packet error bit
x = checksum error bit
a = program code end address. Checksum is calculated from 0x0000 to this
     address (but not including this address!)
c = 8-bit checksum of new program code
```

Note that the address sent with the Final_chksum packet is *outside* the address space used by the new program code. It is equal to the highest address of the final 64-byte sector *plus one*. Therefore this address will be a 64-byte aligned address (6 LSB’s = 0).

Effect:

This packet is a final checksum for the entire new program code stored in FLASH program memory. If this checksum fails, there is little that can be done to easily correct the problem. The entire programming sequence must likely be restarted. However, it is a necessary final check to ensure that the new program code has made it uncorrupted into program memory. An error-free response to this packet indicates that the entire program code has successfully made it into program memory and is ready for execution.

Errors:

Three conditions can cause an error in response to a Final_chksum packet:

1. The Final_chksum packet was not preceded by a Start or Program_64 packet.
2. The Final_chksum packet follows a Program_64 packet, but the program code end address does not match the expected end address (calculated by adding 0x40 to the Program_64 address).
3. A checksum mismatch occurs.

In all cases, the error bit will be set. In the case of a checksum mismatch, the calculated checksum will be appended to the returned message, like so:

```
1000 0010 aaaaaaaa aaaaaaaa cccccccc xxxxxxxx

e = error bit (will always be 1 in this case)
a = 16-bit address for START of new program data
c = checksum sent by PC
x = data memory checksum calculated by MCU (mismatch)
```

An address or packet mismatch (cases 1 and 2) response will not include the calculated checksum. Therefore, it is possible to determine which type of error occurred by checking the length of the returned error packet.

Jump_PC Packet

Format:

```
DES[3:0] = 0101
```

The payload of a Jump_PC packet includes only the descriptor.

```
e000 0100

e = error bit
```

Effect:

The Jump_PC packet instructs the MCU to jump execution to address 0x10000, the start location of updated program code. *This packet must be preceded by a successful Final_Chksum packet.* There is no guaranteed response to this message, as it causes the program counter to begin executing new code. It is recommended that the new code be programmed to send a confirmation upon startup.

Errors:

The error bit is set if a Jump_PC packet is preceded by anything other than a successful Final_Chksum packet.

Summary

The key to understanding the reprogramming sequence is to understand what each command does, and the circumstances under which each command is allowed to execute. Some commands have more than one allowable preceding command. This chart will help:

Command	Required Preceding Command 1	Required Current Address 1	Required Preceding Command 2	Required Current Address 2
Start	Any	Any		
Data	Start	‘xxxxxxxx xx000000'	Data	Previous + 0x4
Checksum_64	Data	Previous + 0x3C	Checksum_64	Previous
Program_64	Checksum_64	Previous		
Final_chksum	Start	Any	Program_64	Previous + 0x40
Jump_PC	Final_chksum	N/A		

Important notes:

Following an MCU restart, the MCU will be executing old code, not the reprogrammed code. In order to switch to the new code, send a start command, followed by a Final_chksum (yes this means the checksum *must* be known!), followed by a Jump_PC command. One side effect of allowing this sequence is that a Final_chksum packet is not required to follow a Program_64 packet. However, it is **strongly advised** that when new program data is initially written, you follow the final Program_64 packet with a Final_chksum packet. This helps to ensure that the entire memory space that was supposed to be written has actually been written.

MCU Startup Message

In order to keep the user of a potentially reprogrammed system informed, a message is sent at the beginning of the MCU code. This message uses the ID of a reprogramming packet, and a message of this type should be used in newly reprogrammed code to indicate successful execution of upper memory code.

Format:

The payload of a startup packet includes an upper byte of all 1’s followed by three bytes of all zeros, representing the memory location of the startup code (location 0x00). Newly reprogrammed code should indicate its memory location by inserting 0x10000, which is the startup location of the new code.

```
11111111 00000000 00000000 00000000

Hex: 0xFF 00 00 00
```

```
NEW FUNCTIONS TO IMPLEMENT (OR OFFICIALIZE):

RESET TDCs
RESET TDIG
```