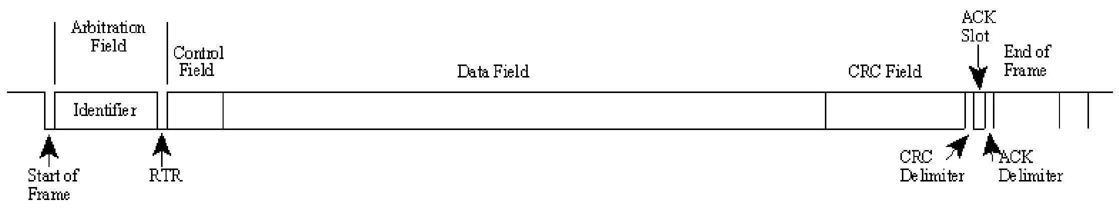
CAN Plan

Thursday, February 20, 2020 2:04 PM

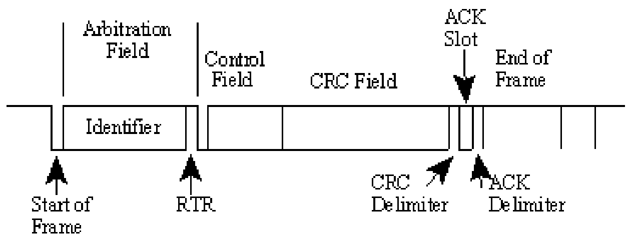
BASIC INFO:

CAN stands for controller area network. It's a collection of ECUs (electronic control units) or nodes that are able to communicate over two lines (CAN_High and CAN_Low). All nodes are able to send and receive messages to all other nodes. The bus decides which nodes transmit messages first based on the content of their messages. As long as none of the nodes send two messages containing the same ID, there will be no collisions on the bus and the message with the lowest ID will be sent first. ID's are not node addresses; any node is able to send a message with any ID. It is the programmers' responsibility to ensure that nodes do not send the same message ID. The TTUCAN software can prevent this as long as no two nodes are initialized with the same address. The ID protocol used for the software will be explained in the MESSAGE ID DESIGN section.

A CAN message is a string of binary numbers (all 1's and 0's, but can be represented in code using the Hexadecimal numbering system). A CAN message has a few important parts (called fields). The Arbitration Field is the message ID. It is used to decide which message is given access to the bus first if there are multiple nodes trying to transmit. ID values closer to 0 will take priority over ID values with higher numbers (These IDs should be reserved for very important messages). An ID can be 11bits long for standard CAN protocol and 29bits long for extended CAN protocol. The message ID can also be used to hold other information (more info about this in the MESSAGE ID DESIGN section). The next important part of a message is the RTR bit, it is used to indicate if a message is a Remote Frame or not (RTR bit is 0 if not remote frame, 1 if remote frame). The Data Field contains the data to be sent to the other nodes. This field will be explained in more detail in the DATA FIELD DESIGN section. The last field of interest is the ACK bit. This is set high by other controllers to let the sender know that the message was transmitted without error.



There are four message types (called frames); two are important to this design. A Data Frame is just a message that contains data. It is structured in the same way as the image above. A Remote Frame is used to ask for a corresponding Data Frame from another node. It uses the same ID as the message that should send the data, but the RTR bit is set high so that it doesn't break the rule that no two nodes should send messages with the same ID. Remote Frames don't have any Data Field because they are asking for data. Example of Remote Frame structure is below.



Links for more detailed info: http://www.keil.com/download/files/canprimer_v2.pdf
https://www.kvaser.com/can-protocol-tutorial/

Nodes can use masks and filters to ensure that they only accept messages that contain data that is important to them. Nodes will look at a message's ID and compare it to their filter to decide whether or not to accept and read the message. The mask tells the controller which bits in the ID of the received frame should be compared to the filter. If any of the bits specified by the mask do not match the bits in the filter, the message is rejected. Filters and masks can also be used to designate an ID that should always be accepted. Even if a message is filtered out, all nodes will still check its validity in order to send an ACK bit to the sender. Example:

Example 2. we wish to accept only frames with IDs of 00001560 thru to 0000156F

- set filter to 00001560
- set mask to 1FFFFFF0

More info on masks/filters: http://www.cse.dmu.ac.uk/~eg/tele/CanbusIDandMask.html

MESSAGE ID DESIGN:

This section will outline the protocol used for assigning message IDs. This outline should help to ensure that no two nodes send messages with the same IDs. There are two message ID types; Standard (11bit ID) and Extended (29bit ID). The decision to use either Standard or Extended will be used based on the amount of nodes on the bus. The bus will have a designated Home node. This node will use the least priority IDs and be used for displaying the messages sent on the bus to a Serial Monitor.

Standard IDs will be used when there a 16 or less nodes on the bus. A Standard ID contains 11 bits. We will divide these bit into three sections to convey three types of information. Each node will be given a 4 bit "address" (0000-1111 or 0-15, 16 nodes in total) to designate who is sending or who should be receiving data. The most significant 4 bits of an ID will be used to indicate the receiving node's address. The next 4 bits will be used to indicate the sending node's address. The last three (least significant) bit will be used to indicate the type of data being sent (000-111 or 0-7, 8 types of data in total).

```
| 4-bit destination id | 4-bit source id | 3-bit message id |
```

Using this protocol, a node can have 8 distinct message types to send to any node and it will use the last 3bits of its message ID to designate which type of data is being sent. These bits could also be used to designate data from certain sensors. Remote Frames can be sent that use these last three bits to request a certain type of data from a node. The messages with the most critical data should be given the lowest types (000 is lowest). Similarly, the nodes that receive and act upon the most critical information must be given the lowest addresses (0000 is the lowest). Nodes that send critical information must be given the next highest priority. It is up to designers/programmers to decide which nodes will have the most critical applications, and assign the ID's accordingly. It is also up to programmers to ensure IDs are used consistently and no two nodes use the same ID, this can be done by making sure that no two nodes are initialized with the same address. The Home node will be given the highest address value (lowest priority), as it will be used mainly for monitoring and debugging the bus.

Extended IDs will be used if the bus ever has more than 16 nodes; it cannot have more than 256 nodes using this design. An Extended ID contains 29bits. We will divide these into the same three sections. The most significant 8 bits will be used to indicate the receiving node (0000 0000-1111 1111 or 0-255, 256 nodes total). The next 8 bits will indicate the sending node. The last 13 bits will indicate the message type (allowing for each node to send 8,192 different message types). The addresses and message types should be set according to priorities, just like in the Standard ID configuration.

Standard ID layout:	0RRR RSSS SDDD
R	Receiver node bits (0000 – 1111 or node 0-15)

S	Sender node bits (0000 – 1111 or node 0-15)
D	Data descriptor bits (000 – 111 or descriptor 0-7)

Extended ID layout:	000R RRRR RRRS SSSS SSSD DDDD DDDD DDDD
R	Receiver node bits (0000 0000 – 1111 1111 or node 0-255)
S	Sender node bits (0000 0000 – 1111 1111 or node 0-255)
D	Data descriptor bits (0 0000 0000 0000 – 1 1111 1111 1111 or descriptor 0-8192)

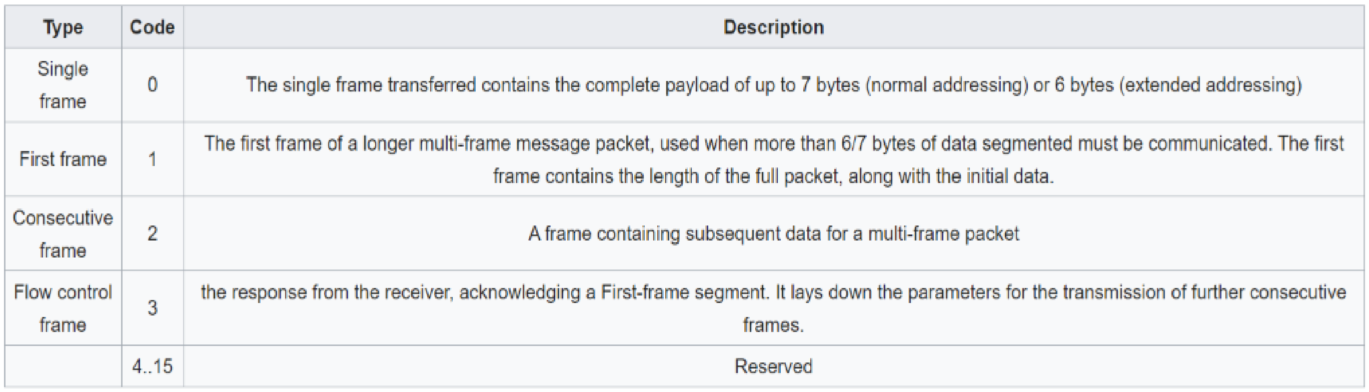
Using this design for IDs allows for nodes to communicate directly to each other while also simplifying the filtering process. It also allows nodes to filter out any messages that are not directly addressed to them, because the IDs in each message contain the receiving node's address. The MCP2515 (can controller that we are using) has 5 filter registers and 2 mask registers. Using this protocol, all nodes will have one filter and one mask dedicated to checking that their own address is in the receiving portion of the message ID. This will ensure that all nodes will only process data that is necessary to their functions (saving them time to do other things).

This protocol provides some outline and structure to the assignment of IDs, but it does create some limitations. It specifies that every message should have two associated addresses: the sender and the receiver. Because the MCP2515 only has 5 filter registers and two mask registers, each node can only accept a limited amount of messages that are not directly addressed to it. The first mask and the first two filters are dedicated to accepting the general network ping sent by the networkStatus() function and the direct ping sent by the pingNode() and checkNode() functions. The second mask is dedicated to filtering messages sent by other nodes. The third filter is used to accept messages sent directly to the node by other nodes. The last three filters are initialized to be the same as the third filter, but they can be changed to accept messages addressed to other nodes. Any node is limited to accepting messages from three other nodes. This can be problematic in the case where several nodes need the same data from another node as soon as possible. This issue can be mitigated in a few ways. If the data to be sent is very urgent to several nodes, it is best to address this data to the most critical node (lowest address) and give filters to the other interested nodes that allow them to accept data addressed to the lowest address. Another way to deal with this issue is to have the sending node send the same data multiple times to each receiving node. This solution will decrease the amount of filters needed, but it should be used for data that is less time critical. It will help to plan ahead by grouping important sensors/info on high priority sending nodes, and grouping important actuators on high priority receiving nodes to ensure that messages get to their destinations as quickly as possible. It is important to keep in mind that this grouping approach could increase the risk of system failure if one of the critical nodes failed. It is important to balance out these pros and cons while designing a system using this protocol.

DATA FRAME DESIGN:

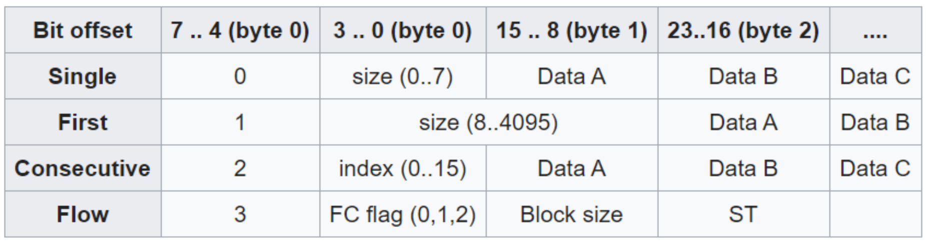
As previously stated, a Data Frame is a message sent on the bus that contains data. The part of this message that contains data is called the Data Field. CAN bus messages are limited to only carry 8 bytes (64 bits) of data in one message. We will make use of the ISO-TP protocol that allows for data to be sent over multiple messages. This will change our data limit from 8 bytes to 4095 bytes. (4095 bytes is a theoretical limit; the real maximum amount will be lower, it and depends on several factors.) We will implement this protocol by inheriting a pre-written library into our new TTUCAN class.

This protocol works by splitting the Data Field into parts that contain metadata (info about the message itself), and it works similarly to how the message ID design is used. Each Data Field begins with Protocol Control Information (PCI). The PCI can be 1,2, or 3 bytes long depending on the message being sent. The PCI contains the type code for the message being sent, the length (in bytes) of the data to be sent, and some other information depending on the time. It begins at the Least Significant Byte of the 8 byte Data Field. There are four different message types, and each of their PCI bytes have some minor differences.



Type	Code	Description
Single frame	0	The single frame transferred contains the complete payload of up to 7 bytes (normal addressing) or 6 bytes (extended addressing)
First frame	1	The first frame of a longer multi-frame message packet, used when more than 6/7 bytes of data segmented must be communicated. The first frame contains the length of the full packet, along with the initial data.
Consecutive frame	2	A frame containing subsequent data for a multi-frame packet
Flow control frame	3	the response from the receiver, acknowledging a First-frame segment. It lays down the parameters for the transmission of further consecutive frames.
	4..15	Reserved

CAN-TP Header



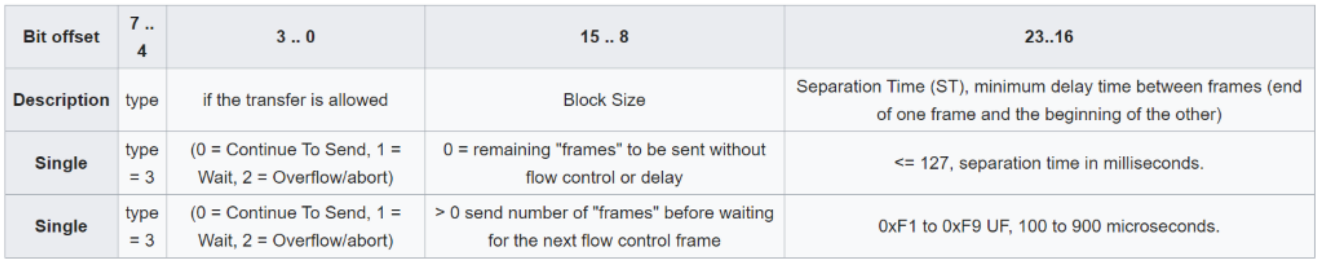
Bit offset	7 .. 4 (byte 0)	3 .. 0 (byte 0)	15 .. 8 (byte 1)	23..16 (byte 2)	....
Single	0	size (0..7)	Data A	Data B	Data C
First	1	size (8..4095)		Data A	Data B
Consecutive	2	index (0..15)	Data A	Data B	Data C
Flow	3	FC flag (0,1,2)	Block size	ST	

A Single Frame (SF) is a Data Frame in which all the data can fit in the Data Field (7 bytes of data or less). The least significant byte (byte 0) of the Data Field contains the PCI. The PCI contains the message type (0000 for a Single Frame), and the length of the data in bytes (0-7 bytes for a single frame). The other bytes will contain the data to be sent.

A First Frame (FF) is used when the data that needs to be sent will be more than 7 bytes. The PCI for a first frame takes up two bytes. It contains the message type (0001 for FF) and the length of the data that will be sent over multiple frames.

A Flow Control Frame (FCF) is sent by the receiving node after receiving a First Frame. This message is used to set the rules for the transmission of the following frames. The PCI is three bytes. It contains the type (0011 for FCF), the flow control flag, and the block size. The flow control flag tells the sending node whether to stop, wait, or continue sending data. The block size tells the sender how many frames to send before waiting for the next FCF. The rest of the Data Field is used to tell the sender the minimum allowed delay time between the messages being sent.

Flow Control



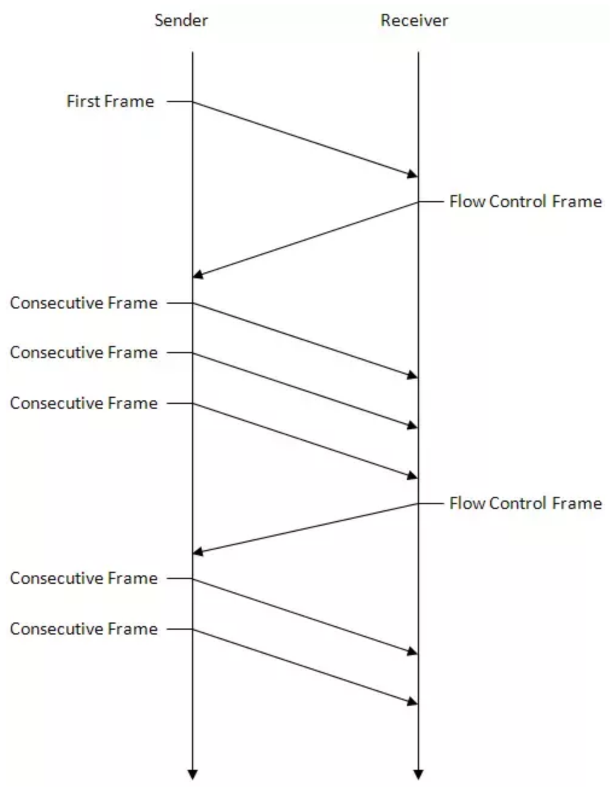
Bit offset	7 .. 4	3 .. 0	15 .. 8	23..16
Description	type	if the transfer is allowed	Block Size	Separation Time (ST), minimum delay time between frames (end of one frame and the beginning of the other)
Single	type = 3	(0 = Continue To Send, 1 = Wait, 2 = Overflow/abort)	0 = remaining "frames" to be sent without flow control or delay	<= 127, separation time in milliseconds.
Single	type = 3	(0 = Continue To Send, 1 = Wait, 2 = Overflow/abort)	> 0 send number of "frames" before waiting for the next flow control frame	0xF1 to 0xF9 UF, 100 to 900 microseconds.

A Consecutive Frame (CF) is sent after the sending node receives the Flow Control Frame. Multiple CFs will be sent until the either the data is complete or the receiving node sends a command to stop. The PCI for this message type is one byte. It contains the message type (0010 for CF), and the index of the data (0-15 in binary). The index tells the receiver what order the frames are being sent in, and helps it recognize if a message was missed. The frames count up from 0 to 15 and then restart from 0 again until all of the data has been sent. Up to 7 bytes of data can be sent in each Consecutive Frame.

The general order for sending data packets is this (these should be handled by the library):

- 1) Sender sends a First Frame with information about how long the data packet will be, as well as some actual data. Sender then waits for acknowledgement from the receiver.
- 2) Receiver send Flow Control Frame telling Sender if it is ok to transmit, and how much data to transmit before waiting for another FCF.
- 3) Sender sends Consecutive Frames according to the rules set by the Receiver node.
- 4) Receiver sends updated FCF to control flow of data transmission.
- 5) Repeat steps 3 and 4 until all data is transmitted.

Example Diagram:



More info on ISO-15765 (ISO-TP): https://en.wikipedia.org/wiki/ISO_15765-2
https://www.quora.com/How-is-more-than-eight-bytes-of-data-is-transmitted-on-the-CAN-bus-network-protocol
Original library that handles ISO-Tp: https://github.com/altelch/iso-tp
Library that we created by modifying the above: https://github.com/saidm5797/TTU_IsoTp

- This library is still under development and is not yet fully functional.

HOME NODE:

The HOME node will be used for debugging and monitoring activity on the bus. Users are able to input commands through the serial monitor to run different network status functions. Use the **Home_Node** example for setting up this node. The available functions are outlined here:

- display_activity(void) - this will allow the home node to receive and display all of the data being transmitted on the bus on the serial monitor until a user inputs a stop command.

- networkStatus(void) - function used to check if there is a functioning network (bus). This is done by sending a message and waiting for an ACK bit; if the message is acknowledged by any node, there is a functional network. The message sent by the home node will be the lowest priority message (0x7FF for Standard ID or 0x1FFFFFFF for Extended ID).

- checkNodes(void) - this function will be used to check which nodes are functioning correctly on the can bus. It is implemented using a for loop that sends remote requests to each node and waits for a response. If a node responds to the remote request it will be considered as functioning. This function will display on the serial monitor the status of each node as it checks them. The user can enter a stop command to end the function at any time.

- Remote ping IDs use the address of the target node in both the send/receive areas, the data descriptor will be 7 to maintain lowest priority (111 in binary).
    - Standard ID example - 0bTTT TTTT T111 (T stands for target address bit)
- If they are functioning, the targeted nodes will respond with a message addressed to the home node with data descriptor 7.
    - Standard ID example - 0b111 1TTT T111 (T stands for target address bit; home address is 0b1111)

- pingNode(address) - this function will send a remote request to a single node (node address is specified as argument to the function) to check its functionality and return true or false.
    - Remote ping IDs will use the address of the target node in both the send/receive areas, the data descriptor will be 7 to maintain lowest priority (111 in binary). 0bTTT TTTT T111 (T stands for target address bit)

GENERAL USE:

This section will lay out some general guidelines for using the software.

The finished library can be found here:

- TTUCAN: https://github.com/saidm5797/TTUCAN

The necessary supporting libraries are found here:

- MCP_CAN_lib: https://github.com/coryjfowler/MCP_CAN_lib
- TTU_IsoTp: https://github.com/saidm5797/TTU_IsoTp

To create a CAN object use the class constructor:

- TTUCAN(INT8U _CS, uint8_t mcp_int, INT32U _address, INT8U _ext)

The arguments are the chip select pin (recommended pin 10), the interrupt pin (recommended pin 2), the address of the node you are initializing (0-15 for standard; 0-255 for extended), and the extension status (0 for standard; 1 for extended). It is important to make sure that the node's address and extension status match. It is also important that all nodes use the same extension status or else they will filter out all messages.

Use the TTU_begin() function to initialize the node:

- TTU_begin(INT8U idmodeset = MCP_STDEXT, INT8U speedset = CAN_500KBPS, INT8U clockset = MCP_16MHZ)

This function works using default arguments, but other arguments can be used if desired. Look in the MCP_CAN_lib for options of other arguments. The begin function will set up all of the necessary filters automatically. To initialize other filters use the addFilter() function:

- addFilter(INT32U filterAddress, INT8U filterRegister)

The arguments are the node address whose messages to receive, and the filter register to initialize (each node can only change filters 3, 4, and 5)

It is up to the user to decide what message IDs to use for each type of information based on the guidelines mentioned above. It is necessary to know what messages a node will be sending and receiving before runtime so that the data being transmitted can be processed accordingly. Use the **display_IDs** example to see a list of all message IDs that a node can send and receive from other nodes. This will help in planning communication between nodes.

Other functions are available for getting message IDs.

- buildTransmitID(INT32U to_addr, INT32U descriptor)
- buildReceiveID(INT32U from_addr, INT32U descriptor)

These functions take the node to send a message to or receive a message from, and the descriptor of the ID. They return the message IDs to be passed to the send_Msg() function or to be used in the preProcess() and postProcess() functions.

After planning out the messages to be sent and received, the user should then modify the preProcess() and postProcess() functions.

- preProcess(byte *data, INT32U to_addr, INT32U descriptor)
- postProcess(byte *data, INT32U receiveID)

The preProcess() function takes a data array to be filled, the node address to send to, and the data descriptor to be used.
The postProcess() functions takes a data array to be filled, and the ID of the received message.
These functions look at the IDs of messages to be sent or that have been received and then construct variables into a data array or deconstruct the received data into variables. It is important to use global variables as these functions do not take any variables as arguments.
Use the **pre_post_process** and **three_node_comm** examples to see how these functions should be structured.

To send a message use the send_Msg() function:

- send_Msg(INT32U msgID, INT8U rtr, INT8U *data, INT32U len)

This function takes the message ID to use, the remote status (1 for remote message, 0 for non-remote message), the data array to send, and the length of the data in bytes (8 bytes in most cases).
Before using the send_Msg() function, use the buildTransmitID() function or the preProcess() functions. These functions return a message ID that can be passed to the send_Msg() function.
The preProcess() function will also fill the data array based on the node you plan to send to and the data descriptor used.

To receive a message use the receive_Msg() function:

- receive_Msg(INT32U *id, INT8U *len, INT8U *buf)

This function takes pointers to variables that hold the received message's ID and length, as well as a data array to fill with received data. This function should always be used within an if statement as shown here:

```
If(!digitalread(CAN0_int)){
        receive_Msg(&rxId, &len, rxBuffer)
}
```

This makes use of the interrupt pin (named CAN0_int in this case) to make sure the controller only looks for messages when it has received one. Look at the **Send_and_Receive** example to see how this is done.

After receiving a message, use the postProcess() function to update the variables in your program so that they can be used as normal.