GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2017

Lab #7: Controller Area Network

Contents

1 Background Material **1**
- 1.1 Introductory Comments . . . 1
- 1.2 Basic Concepts of CAN . . . 2
- 1.3 Debugging Two Targets Simultaneously . . . 4
- 1.4 Relevant Microcontroller Documentation . . . 5
- 1.5 Target Hardware Schematic Diagrams and Data Sheets . . . 6

2 CAN Module: Step-by-Step Guidelines **6**
- 2.1 Initialize the CAN Registers . . . 7
  - 2.1.1 Set Pin Multiplexer . . . 7
  - 2.1.2 Enable Module Clock . . . 7
  - 2.1.3 Enable Pin Functions . . . 7
  - 2.1.4 Set Timing Parameters . . . 8
  - 2.1.5 Activate Normal Operation . . . 8
  - 2.1.6 Configure Mailboxes . . . 8
- 2.2 Utilize the CAN Pins . . . 9
  - 2.2.1 Transmit Data . . . 9
  - 2.2.2 Receive Data . . . 9

3 Lab Assignment **10**
- 3.1 Pre-Lab Preparation . . . 10
- 3.2 Specification of the Assigned Tasks . . . 10
  - 3.2.1 Communication Between Two CAN Nodes . . . 10
  - 3.2.2 Motor Control with CAN Command Signal . . . 11

1 Background Material

1.1 Introductory Comments

We have seen in Lab 6 how a single microcontroller can serve multiple purposes in a control system implementation: one on-chip peripheral circuit may serve as an interface to the plant's actuator; another on-chip peripheral circuit may serve as an interface to the plant's sensor; yet another on-chip peripheral circuit may serve as a timer to activate time-periodic actuator writes and sensor reads; and finally the on-chip computational engine may make control decisions based on real-time solutions of the differential equations that define the control algorithm. Position control of a single motion axis is an application that typically requires just one microcontroller.

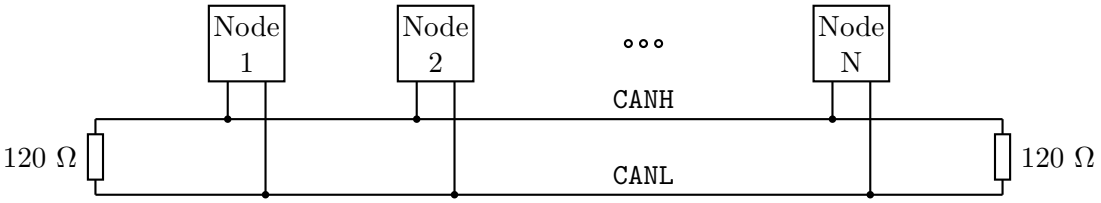
Complex systems often require more than one microcontroller to implement multiple operating modes and to control various subsystems. For example, the powertrain of a hybrid electric vehicle can supply power to the wheels from two sources, individually or simultaneously; one power source is the gasoline engine subsystem, whereas the other power source is the electric drive subsystem

(consisting of a battery pack, a power converter and an electric machine). During operation, the hybrid supervisory controller must send torque requests to the two power sources and to the friction brake system, based on the positions of the accelerator and brake pedals and the shift lever (assigned by the human driver) and the battery pack state-of-charge (which cannot be allowed to become too large or too small). In order to do its job, the hybrid supervisory controller requires information from two pedals and one lever located in the vehicle cockpit and the battery pack located in the rear of the vehicle, and it must supply information to two power sources located in the front of the vehicle and to the friction brakes located in the front and rear of the vehicle. One way to provide the required communication would be to install dedicated wiring between the hybrid supervisory controller and each of the relevant subsystems, which in this case would amount to eight separate long wire bundles and many connection pins. Such an approach adds unnecessary cost and mass and creates a reliability concern, so an alternative communication method is desired.

The preferred communication method for complex and physically large control systems is based on the Controller Area Network (CAN) standard, which involves serial communication over a single two-wire twisted-pair cable called the CAN bus. Originally developed at Bosch in the mid 1980s for automotive applications, CAN has since been widely used in vehicles of all types, as well as for industrial automation and medical equipment. The objective of this lab is to learn how to use CAN bus communication in multi-node microcontroller-based control systems. In particular, we will establish CAN bus communication between the Peripheral Explorer Motherboard and the Motor Control Motherboard, so that the former may be used to command the latter.

1.2 Basic Concepts of CAN

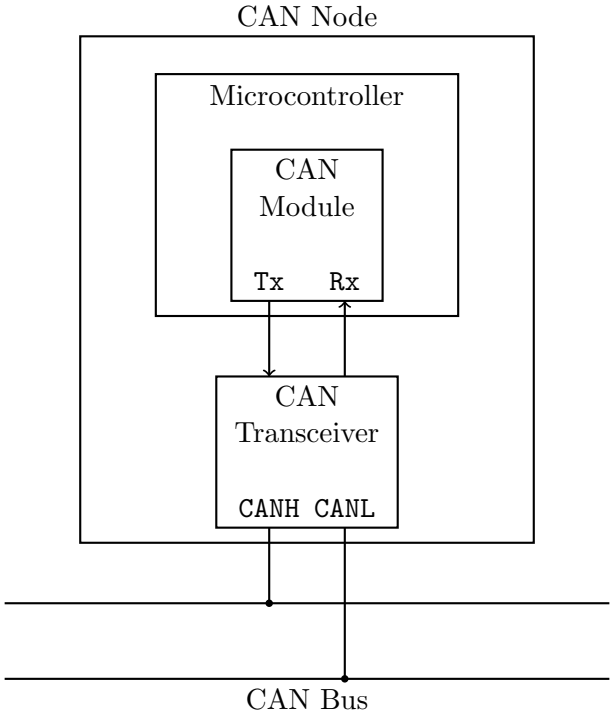
A controller area network, depicted in the figure below, allows serial communication between multiple nodes using a protocol that grants equal rights to all nodes. Any of the nodes may transmit one message at a time, whenever the bus is idle. In addition to a raw data value, each message also contains an identifier which reflects the nature and priority of that particular message (e.g. commanded engine torque or measured vehicle speed). If more than one node attempts to transmit simultaneously, the competing nodes are forced to transmit their messages in sequence according to their message identifiers. Every node on the bus receives every transmitted message, and each node decides to use or discard any given message based on its message identifier. With such a network, all the information needed to monitor and control a physical system may be efficiently and reliably distributed throughout the system in real time.



In typical applications, a shielded four-conductor cable is used to distribute the two bus wires (labeled `CANH` and `CANL` above), as well as power and ground wires (not shown above), throughout the physical system; the ground wire of this cable is used to establish a common ground for all nodes on the bus. The length of the bus can be quite large, depending on the bit rate used; at the maximum bit rate of 1 Mbps, the maximum bus length is about 50 m. The two ends of the bus must be terminated by 120 Ω resistors, as shown above, to minimize signal reflection.

As shown below, each node on the bus incorporates a CAN transceiver (a dedicated chip) and a CAN controller (a dedicated chip or a peripheral module inside a microcontroller); these

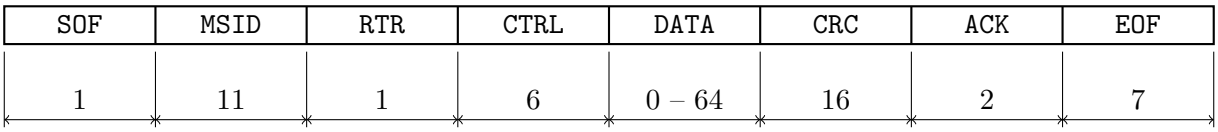
two elements implement (in silicon) the physical layer and the data-link layer of the network, respectively. In the past, when the transceiver and controller elements were both dedicated chips, 5 V logic was commonly employed. Now that the controller element is often a microcontroller peripheral module, 3.3 V logic offers some advantages. Since the transceiver element operates on the basis of differential voltage, it is possible to combine 5 V and 3.3 V transceivers on the same bus. In fact, in this lab we will be using the TI SN65HVD230 3.3 V transceiver on the Peripheral Explorer Motherboard and the TI ISO1050 5 V isolated transceiver on the Motor Control Motherboard.



As mentioned above, the data-link layer of the network is implemented in silicon, so programmers don't need to know all the details of the protocol to write code. However, a few features of the protocol are worth considering since they reveal why CAN is superior to other protocols for real-time control applications. Each message, or frame, is constructed as shown below. The first field `SOF` is a single start-of-frame bit. The second field `MSID` is an 11-bit message identifier; its important role has been previously described above. The third field `RTR` is a single remote-frame-request bit; it is used to distinguish standard messages that broadcast data from messages that request data. The fourth field `CTRL` contains 6 bits and conveys (among other things) the length of the data packet, in units of bytes. The fifth field `DATA` is perhaps the core of the frame; it consists of up to 8 bytes (i.e. 64 bits) of data, and protocol overhead is reduced by packing numerous distinct data values into different bytes (e.g. 8 different 8-bit values, 4 different 16-bit values, or some mix of different 8-bit and 16-bit values). The sixth field `CRC` has 16 bits and contains a cyclic redundancy code used to detect errors that may occur during transmission. The seventh field `ACK` has 2 bits and serves to acknowledge the presence or absence of transmission errors. The eighth field `EOF` has 7 bits and marks the end of the frame or message.

As discussed above, each message is a sequence of bits, so transmission and reception of messages requires time synchronization. Each node on the bus must be configured for operation at a particular bit rate. The maximum possible bit rate is 1 Mbps, or 1×10^6 bits per second. Once a bit rate has been established, what will the resulting data-transfer bandwidth end up being?

CAN Frame Architecture
(number of bits in each field is displayed)



According to the discussion above, each message containing a full 8-byte data packet consists of 108 bits. Some time must elapse between messages, and this is accounted for by adding an inter-frame gap equivalent to 3 additional bits, resulting in an effective message length of 111 bits. The protocol permits only 5 consecutive 0-bits or 1-bits, so a process called bit stuffing occurs in the background to insert (transmission) and extract (reception) extra non-informative bits, as necessary. The worst-case bit stuffing adds 24 bits for messages with an 8-byte data packet, resulting in an effective message length of 135 bits. Based on this analysis, data will be exchanged at a maximum rate of 7404 messages per second. Since the data packet is large enough to contain numerous signal values, we may conclude that implementation of a typical feedback control loop could be performed at an update rate exceeding 7 kHz, using the bus to shuttle information between sensors, actuators and decision makers. If the amount of information to be shuttled is reduced, the achievable controller update rate is increased. Considering the fact that the controller update rate used in Lab 6 for position control was merely 1 kHz, it follows that typical motion control systems may be readily implemented over a CAN bus, provided that the sensors and actuators distributed throughout the system have the required CAN node hardware embedded within them.

1.3 Debugging Two Targets Simultaneously

Code development for two targets begins with the creation of two separate CCS projects in the same CCS workspace. These two projects are created in the normal way, and the two codes are compiled in the normal way as well.

In the past, your host computer running CCS has been connected to just one device during a debug session. Since this lab considers a network of two devices, it is essential to learn the new skill of debugging two devices simultaneously. CCS will use the serial numbers associated with your two debug probes to distinguish between them during debug sessions, and TI has provided a utility that we may use to extract the serial number of any connected debug probe. The required file, `xds100serial.exe`, has been uploaded to tsquare, and its use is straightforward; type `cmd` in the Windows start tool to bring up the command prompt, then navigate to the appropriate location and type `xds100serial` to execute the utility. Make a note of the serial numbers for your two debug probes, as you will need them for the next step.

Every debug session has a target configuration file associated with it. For example, if you open any of your previous CCS projects, you will find the folder `targetConfigs` and the default target configuration file `TMS320F28069.ccxml`. What we require now is a user-defined target configuration file that identifies multiple connection types, multiple device types, and multiple debug probe serial numbers. The procedure for creating such a file is given below.

1. Select `View` and `Target Configurations`, then click the `New Target Configuration File` icon, provide your desired `File Name` and select `Finish`.

2. Under `General Setup`, set the `Connection` to be `Texas Instruments XDS100v2 USB Debug Probe`, set the `Board or Device` to be `TMS320F28069` by clicking the appropriate check box, and select `Save`.

3. Under Advanced Setup, click Target Configuration and New, set Connections to be Texas Instruments XDS100v2 USB Debug Probe, select Finish, then click Add, set Devices to be TMS320F28069 and select Finish.

4. Select Texas Instruments XDS100v2 USB Debug Probe_0. At Connection Properties, set Debug Probe Selection to Select by serial number, then type in the serial number of the desired debug probe at Enter the serial number.

5. Select Texas Instruments XDS100v2 USB Debug Probe_1. At Connection Properties, set Debug Probe Selection to Select by serial number, then type in the serial number of the desired debug probe at Enter the serial number.

6. Select Save, and your target configuration file will be available in the Target Configurations window inside the User Defined folder. Right click your file and select Launch Selected Configuration to launch a first debug session.

At this point, you have successfully saved the new target configuration you will need for this lab (and launched it once already), so you can now close the Target Configurations window. Use of this new target configuration for a two-device debug session involves the following steps.

1. Launch subsequent debug sessions by left clicking the arrow beside the Debug icon, and then select the new target configuration file you previously saved.

2. Select Texas Instruments XDS100v2 USB Debug Probe_0/C28xx

   (a) Left click the Connect Target icon
   (b) Left click the arrow beside the Load icon, and select Load Program
   (c) Select the desired .out file from those listed, and select OK
   (d) Left click the Resume icon to begin executing your code

3. Select Texas Instruments XDS100v2 USB Debug Probe_1/C28xx

   (a) Left click the Connect Target icon
   (b) Left click the arrow beside the Load icon, and select Load Program
   (c) Select the desired .out file from those listed, and select OK
   (d) Left click the Resume icon to begin executing your code

4. Use the Expressions window to display global variables from either device; switch your selection from one device to the other in the Debug window as necessary.

1.4 Relevant Microcontroller Documentation

The overall objective of these lab projects is to teach you how to do embedded design with microcontrollers in a general sense, not just how to approach one specific application using one specific microcontroller. Therefore, the guidance provided herein focuses more on general thought processes and programming recommendations; step-by-step instructions of an extremely specific nature have been intentionally omitted. Use fundamental documentation as your primary source of information as you work through details of implementation. Being able to read and understand such documentation is an important skill to develop, as similar documentation would need to

be consulted in order to use other microcontrollers or other application hardware. By making the effort to extract required details from fundamental documentation yourself, you will have developed transferable skills that will serve you well in your engineering career. For this lab, review:

- Technical Reference Manual
  - §16.1–16.5, for a description of the CAN module.
  - §16.6, for the status and control register field descriptions.
  - §16.8, for the mailbox register field descriptions.
  - §16.10.1–16.10.2 and Figure 16-41, for bit rate parameter selection.
    Note that the bit rate is governed by

$$f_{\text{bus}} = \frac{f_{\text{clk}}/2}{(\text{bit rate prescaler}) \times (\text{time quanta per bit})}$$

$$\text{bit rate prescaler} = \texttt{BRPREG} + 1$$

$$\text{time quanta per bit} = 1 + (\texttt{TSEG1REG} + 1) + (\texttt{TSEG2REG} + 1).$$

    The values `BRPREG`, `TSEG1REG` and `TSEG2REG` are the integer values assigned to the corresponding fields of the Bit Timing Configuration register `CANBTC`. To accommodate the bit timing specifications defined in the tasks, a good option would be to use 15 time quanta per bit, and to set the sample point to be at 13 time quanta.

1.5 Target Hardware Schematic Diagrams and Data Sheets

- Peripheral Explorer Motherboard
  - transceiver chip (SN65HVD230, a 3.3 V transceiver)
  - bus connection header (J4 or J5, double header allows bus continuation)
  - ground connection header (J19, J20 or J21, connect to other motherboard)
  - bus termination resistor (R8, introduced by placing a jumper on J24)
- Motor Control Motherboard
  - transceiver chip (ISO1050, a 5 V transceiver with isolation)
  - bus connection header (J7, pin 1 is indicated by arrow on silkscreen)
  - ground connection header (J7, pin 3, connect to other motherboard)
  - bus termination resistor (R101, introduced by placing a jumper on JP4)

2 CAN Module: Step-by-Step Guidelines

We will need to interact with two types of CAN module registers: status and control registers; and mailbox registers. As indicated in §16.3.2.1 of Technical Reference Manual, all reads and writes of status and control registers (those with HAL structure variables having prefix `ECanaRegs`) must always be 32-bits wide, so the `.bit` option must be avoided when reading from or writing to the HAL structure variables. However, the `.bit` option is a very convenient way of interacting with various fields buried within a large register. Therefore, we will declare duplicates of the HAL structure variables by placing the following declaration at the top of the source file:

```
struct ECAN_REGS ECanaCopy;
```

As shown below in the step-by-step guidelines of the following sections, we will use multi-step processes whenever we need to interact with status and control registers.

1. To guarantee 32-bit writes (see Example 16-1 in Technical Reference Manual)
   (a) Copy the entire HAL structure variable (prefix `ECanaRegs`) into a duplicate structure variable (prefix `ECanaCopy`) using the `.all` suffix.
   (b) Write to the individual fields of the duplicate structure variable (prefix `ECanaCopy`) as needed using the `.bit` option for convenience.
   (c) Copy the entire duplicate structure variable (prefix `ECanaCopy`) into the HAL structure variable (prefix `ECanaRegs`) using the `.all` suffix.
2. To guarantee 32-bit reads (see Example 16-2 in Technical Reference Manual)
   (a) Copy the entire HAL structure variable (prefix `ECanaRegs`) into a duplicate structure variable (prefix `ECanaCopy`) using the `.all` suffix.
   (b) Read from the individual fields of the duplicate structure variable (prefix `ECanaCopy`) as needed using the `.bit` option for convenience.

In the sections that follow, the essential code lines for each step are provided, with the symbol `?` representing missing information to be supplied. Study the register diagrams and field description tables to help determine appropriate choices for the missing information.

2.1 Initialize the CAN Registers

2.1.1 Set Pin Multiplexer

The module may be electrically connected (multiplexed) to pins of the microcontroller package using the `GPIO?` fields of the `GP?MUX?` registers.

```
GpioCtrlRegs.GP?MUX?.bit.GPIO? = ?;
```

2.1.2 Enable Module Clock

The module clock is enabled using the `ECANAENCLK` field of the `PCLKCR0` register. A short time delay (two `NOP` commands) should occur prior to interacting with any module registers.

```
SysCtrlRegs.PCLKCR0.bit.ECANAENCLK = ?;
```

2.1.3 Enable Pin Functions

The controller element cannot begin to interact with the transceiver element until the functions of the `Tx` and `Rx` pins have been enabled. These pin functions are enabled using the `TXFUNC` field of the Transmit Input-Output Control register `CANTIOC` and the `RXFUNC` field of the Receive Input-Output Control register `CANRIOC`.

```
ECanaCopy.CANTIOC.all = ECanaRegs.CANTIOC.all;
ECanaCopy.CANTIOC.bit.TXFUNC = ?;
ECanaRegs.CANTIOC.all = ECanaCopy.CANTIOC.all;
ECanaCopy.CANRIOC.all = ECanaRegs.CANRIOC.all;
ECanaCopy.CANRIOC.bit.RXFUNC = ?;
ECanaRegs.CANRIOC.all = ECanaCopy.CANRIOC.all;
```

2.1.4 Set Timing Parameters

The TSEG1REG, TSEG2REG and BRPREG fields of the Bit Timing Configuration register CANBTC are defined and discussed in §16.10.1 and §16.10.2 of Technical Reference Manual, and a summary of their role has been provided in a previous section of this document. These parameters determine the bit rate of the bus, f_{bus}, in relation to the system clock frequency, f_{clk}.

```
ECanaCopy.CANBTC.all = ECanaRegs.CANBTC.all;
ECanaCopy.CANBTC.bit.TSEG1REG = ?;
ECanaCopy.CANBTC.bit.TSEG2REG = ?;
ECanaCopy.CANBTC.bit.BRPREG = ?;
ECanaRegs.CANBTC.all = ECanaCopy.CANBTC.all;
```

2.1.5 Activate Normal Operation

The Change Configuration Request field CCR of the Master Control register CANMC must be assigned appropriately to transition from initialization mode to normal mode.

```
ECanaCopy.CANMC.all = ECanaRegs.CANMC.all;
ECanaCopy.CANMC.bit.CCR = ?;
ECanaRegs.CANMC.all = ECanaCopy.CANMC.all;
```

According to Figure 16-40 of Technical Reference Manual, a requested transition from initialization mode to normal mode will not occur until a bus-idle condition is detected. The Change Configuration Enable field CCE of the Error and Status register CANES will automatically change state when the requested mode transition has actually been achieved.

```
do ECanaCopy.CANES.all = ECanaRegs.CANES.all;
while (ECanaCopy.CANES.bit.CCE != ?);
```

2.1.6 Configure Mailboxes

Configuration of a given mailbox requires that the length of the data field, in units of bytes, be assigned to the DLC field of the Message Control register MSGCTRL. According to §16.8.3 of Technical Reference Manual, all fields of the MSGCTRL register should be set to zero in a first step before assigning the DLC field in a second step. The message identifier for a given mailbox is located in the ID field of the Message Identifier register MSGID, but since the standard mode of operation (11-bit identifier) requires all other fields of this register to be zero, it is appropriate to assign the message identifier using the .all option of the corresponding HAL variable. As indicated in Table 16-30, the message identifier must be located in bits 18–28 and bits 0–17 have no meaning (so left shifting the numerical value is required prior to assignment).

```
ECanaMboxes.MBOX?.MSGCTRL.all = 0;
ECanaMboxes.MBOX?.MSGCTRL.bit.DLC = ?;
ECanaMboxes.MBOX?.MSGID.all = ?;
```

The mailbox-dependent fields MD? of the Mailbox Direction register CANMD are used to identify any given mailbox as a transmitter of messages or as a receiver of messages.

```
ECanaCopy.CANMD.all = ECanaRegs.CANMD.all;
ECanaCopy.CANMD.bit.MD? = ?;
ECanaRegs.CANMD.all = ECanaCopy.CANMD.all;
```


The mailbox-dependent fields `ME?` of the Mailbox Enable register `CANME` are used to enable any given mailbox; all other mailbox configuration steps must be completed prior to this final step.

```
ECanaCopy.CANME.all = ECanaRegs.CANME.all;
ECanaCopy.CANME.bit.ME? = ?;
ECanaRegs.CANME.all = ECanaCopy.CANME.all;
```

2.2 Utilize the CAN Pins

2.2.1 Transmit Data

Several options exist for how one might pack data into the data field of the message associated with a given mailbox; this data field is populated according to two 32-bit Message Data Registers, `CANMDL` and `CANMDH`. You can use 8-bit segments (`.byte` option), 16-bit segments (`.word` option) or 32-bit segments (`.all` option); these choices involve HAL variables of the forms shown below.

```
ECanaMboxes.MBOX?.MD?.byte.BYTE? = ?;
ECanaMboxes.MBOX?.MD?.word.?_WORD = ?;
ECanaMboxes.MBOX?.MD?.all = ?;
```

Once the data has been packed, message transmission is initiated by manipulating the mailbox-dependent field `TRS?` of the Transmit Request Set register `CANTRS`.

```
ECanaCopy.CANTRS.all = 0;
ECanaCopy.CANTRS.bit.TRS? = ?;
ECanaRegs.CANTRS.all = ECanaCopy.CANTRS.all;
```

These transmit code lines may be located in a timer ISR.

If a sensor node needs to communicate with a processor node, it may seem tempting to impose SI units (i.e. `float32` data types) within a CAN data packet. However, since we are using HAL variables—which have unsigned integer data types—to pack data at transmitting nodes and unpack data at receiving nodes, explicit use of `float32` data types is not possible. Instead, transmitting nodes must encode `float32` data according to the forward mapping

$$\texttt{Uint32} = m \times \texttt{float32} + b$$

and receiving nodes must decode `float32` data according to the inverse mapping

$$\texttt{float32} = (\texttt{Uint32} - b) \div m$$

where m scales and b shifts. The coefficients m and b must be selected such that all expected `float32` values fit into the range of `Uint32` values; full use of the available range results in higher resolution. If less resolution is needed, then `Uint16` encoding and decoding would be preferable, since the smaller data packets would reduce the loading on the CAN bus.

2.2.2 Receive Data

The only way for a node to successfully receive a message is to know in advance the message's identifier and data packing arrangement. If you are integrating a commercial sensor into your network, you would need to determine the required information from its data sheet in order to program your receiving node. In this lab, you have programmed the transmitting node yourself, so you will just use the inverse of its data packing structure when programming the receiving node.

```
? = ECanaMboxes.MBOX?.MD?.byte.BYTE?;
? = ECanaMboxes.MBOX?.MD?.word.?_WORD;
? = ECanaMboxes.MBOX?.MD?.all;
```

In many applications, it is preferable to read a receiving node mailbox only when it contains a new data packet. This selective approach is made possible by following a two-step process that involves an appropriate bit within the Received-Message-Pending register: 1) reading the appropriate bit prior to reading the mailbox data; 2) writing to the appropriate bit to indicate that the mailbox data has been read.

```
ECanaCopy.CANRMP.all = ECanaRegs.CANRMP.all;
if (ECanaCopy.CANRMP.bit.RMP? == ?) {
    ? = ECanaMboxes.MBOX?.MD?.all;
    ECanaCopy.CANRMP.bit.RMP? = ?;
    ECanaRegs.CANRMP.all = ECanaCopy.CANRMP.all;
}
```

These receive code lines may be located in a timer ISR.

3 Lab Assignment

3.1 Pre-Lab Preparation

Each individual student must work through the pre-lab activity and prepare a pre-lab deliverable to be submitted *by the beginning of the lab session*. The pre-lab deliverable consists of a brief typed statement, no longer than one page, in response to the following pre-lab activity specification:

1. Read through this entire document, and describe the overall purpose of this week's project.
2. Describe in specific terms how the relevant registers will be used to complete the tasks assigned in §3.2. If you will write to a HAL variable, state the numerical value to be written; if you will read from a HAL variable, state how the numerical value read will be interpreted. You will need to consult circuit diagrams in order to complete this part of your pre-lab preparation.

Please note that it is not essential to write application code prior to the lab session; the point of the pre-lab preparation is for you to arrive at the lab session with firm ideas regarding register usage and other relevant issues in relation to the tasks assigned in §3.2.

3.2 Specification of the Assigned Tasks

3.2.1 Communication Between Two CAN Nodes

Develop separate codes for 1) a first F28069 microcontroller that enables the Peripheral Explorer Motherboard to function as a transmitting CAN node, and 2) a second F28069 microcontroller that enables the Motor Control Motherboard to function as a receiving CAN node. The transmitting node should transmit just one type of message using identifier 100, and it should carry the following 3 bytes of data: one byte is for the status of pushbutton PB1 (equal to 0 or 1); one byte is for the status of pushbutton PB2 (equal to 0 or 1); one byte is for the status of the hex encoder (equal to values between 0 and 15). The receiving node should receive just one type of message using identifier 100 and, after unpacking the 3 bytes of data, the expressions window should be used to reveal the status of the pushbuttons and hex encoder. Set timing parameters as specified below.

CPU clock frequency	(both nodes)	f_{clk}	90 MHz
CAN bus frequency	(both nodes)	f_{bus}	500 kbps
Timer interrupt frequency	(transmitting node)	f_{tmr}	1 kHz
Timer interrupt frequency	(receiving node)	f_{tmr}	1 kHz

Instructor Verification (separate page)

3.2.2 Motor Control with CAN Command Signal

Extend both codes developed in the previous task. The microcontroller on the Peripheral Explorer Motherboard must compute and transmit a time-periodic position command signal; use reference command shaping with parameters $r_{1/2} = 0$ rad, $r_{2/1} = 10\pi$ rad, $s_c = \pm 125$ rad/s and $a_c = \pm 3000$ rad/s^2, and with motion strokes recurring every 0.5 s (see Problem Set 7). The microcontroller on the Motor Control Motherboard must receive the time-periodic position command signal and use it to perform the position control task (as in Lab 6). There will still be only one CAN message involved, having identifier 100, but now it will contain 8 bytes of data. The lower 32 bits can carry the 3 bytes needed for the pushbuttons and hex encoder, and the upper 32 bits can be used to carry the position command values. Log reference position r (in rad), actual position y (in rad) and voltage u (in V) for two complete motion cycles (i.e. 2 s duration). Start the receive node first and the transmit node second; use a flag to detect when data begins flowing on the CAN bus, and begin logging r, y and u on the receive node at that point in time. Generate response plots in Matlab using exported data, and summarize your findings.

Instructor Verification (separate page)

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Fall 2017

Lab #7: Controller Area Network

Instructor Verification Page

Lab Section	Begin Date	End Date
L01, L02	October 31	November 7
L03, L04	November 2	November 9

To be eligible for full credit, do the following:

1. Submissions required by each student (one per student)
   (a) Upload your pre-lab deliverable to tsquare before lab session begins on begin date.
   (b) Upload your `main.c` file for §3.2.2 to tsquare before lab session ends on end date.
2. Submissions required by each group (one per group)
   (a) Submit a hard-copy of this verification page before lab session ends on end date.
   (b) Attach to this page the hard-copy plots requested in §3.2.2.

Name #1: _____

Name #2: _____

Checkpoint: Verify completion of the task assigned in §3.2.1.

Verified: _____ **Date/Time:** _____

Checkpoint: Verify completion of the task assigned in §3.2.2.

Verified: _____ **Date/Time:** _____