CAN Peripheral Module

Introduction

This module covers the CAN peripheral, library and examples provided with the LM3S8962 evaluation kit.

Learning Objectives

- CAN Basics
- Ethernet vs. CAN
- CAN Drivers
- CAN Stacks
- Lab
Module Topics

CAN Peripheral Module ......................................................................................................................................................... 1

Module Topics ........................................................................................................................................................................ 2

CAN Basics ........................................................................................................................................................................... 3
  CAN Bus ............................................................................................................................................................................. 3
  CAN Node .......................................................................................................................................................................... 4
  CAN Principles ................................................................................................................................................................... 4
  Message Format .................................................................................................................................................................. 5
  Arbitration .......................................................................................................................................................................... 5
  Message Types .................................................................................................................................................................. 6
  Message Objects ................................................................................................................................................................. 6
  FIFO Mode .......................................................................................................................................................................... 7
  Ethernet vs. CAN ................................................................................................................................................................. 7
  Stellaris CAN ..................................................................................................................................................................... 8

Kits and Software .................................................................................................................................................................... 9
  Drivers .................................................................................................................................................................................. 10
  APIs .................................................................................................................................................................................... 10
  Code Example .................................................................................................................................................................... 11
  Stacks .................................................................................................................................................................................. 14

CAN Lab ................................................................................................................................................................................ 15
  Description: ......................................................................................................................................................................... 15
  Hardware list: ..................................................................................................................................................................... 16
  Software list: ..................................................................................................................................................................... 16
  Procedure: .......................................................................................................................................................................... 17
Controller Area Network (CAN)  
A Multi-Master Serial Bus System

- Bosch CAN 2.0 A/B Standard
- High speed (up to 1 Mbps)
- Add a node without disturbing the bus (number of nodes not limited by protocol)
- Fewer wires (lower cost, less maintenance, and more reliable)
- Redundant error checking (high reliability)
- No node addressing (message identifiers)
- Broadcast based signaling

CAN Bus

- Two wire differential bus (usually twisted pair)
- Max. bus length depend on transmission rate
  - 40 meters @ 1 Mbps
CAN Node

**Principles of Operation**

- Data messages transmitted are identifier based, not address based
- Content of message is labeled by an identifier that is unique throughout the network
  - (e.g. rpm, temperature, position, pressure, etc.)
- All nodes on network receive the message and each performs an acceptance test on the identifier
- If message is relevant, it is processed (received); otherwise it is ignored
- Unique identifier also determines the priority of the message
  - (lower the numerical value of the identifier, the higher the priority)
- When two or more nodes attempt to transmit at the same time, a non-destructive arbitration technique guarantees messages are sent in order of priority and no messages are lost
## Message Format

### CAN Message Format

- **Data is transmitted and received using Message Frames**
- **8 byte data payload per message**
- **Standard and Extended identifier formats**
  - **Standard Frame: 11-bit Identifier (CAN v2.0A)**
  - **Extended Frame: 29-bit Identifier (CAN v2.0B)**

### Arbitration

#### Non-Destructive Bitwise Arbitration

- Bus arbitration resolved via arbitration with wired-AND bus connections
  - Dominate state (logic 0, bus is high)
  - Recessive state (logic 1, bus is low)
Message Types

**CAN Message Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Frame</td>
<td>“Hello everyone, here’s DATA ALPHA, hope you like it!”</td>
</tr>
<tr>
<td>Remote Frame</td>
<td>“Hello everyone, can somebody please produce DATA ALPHA?”</td>
</tr>
<tr>
<td>Error Frame</td>
<td>“[Everyone out loud] “Uh-oh! Whoever sent that, let’s try again.””</td>
</tr>
<tr>
<td>Overload Frame</td>
<td>“I heard you, but I’m bigger than you – and I am busy. Can you please wait?”</td>
</tr>
</tbody>
</table>

CAN networks hear everything… when a node sends something, everyone listens and they either REACT or IGNORE.

- There are two types of transmissions; event-triggered and time-triggered.
  - In event-triggered, events (such as a temperature threshold) cause transmission.
  - In time-triggered, each CAN-entity is only allowed to transmit during a given time allotment.
  - Stellaris features the ability to disable auto-retransmission for the benefit of TTCAN (so our device doesn’t speak during someone else’s turn)

**Message Objects**

**Message Objects (Mailboxes)**

- **Message main parts**: message identifier and data
  - Identifier “names” the content of the message (11 bits or 29 bits)
  - Data can be from 0 to 8 bytes (0=remote frame)
- **Messages are**:
  - Received by protocol controller
  - Passed to message handler
  - Loaded into appropriate message object
  - Messages can be filtered (ignored) using a mask
- **Message object RAM**
  - 32 identical blocks
  - Ordered by priority (1 is highest, 32 lowest)
  - Application reads/writes Message RAM via the message object registers, not accessible directly
FIFO Mode

You can “concatenate” multiple message objects (saves interrupt overhead for messages larger than 8 bytes)

Set the identifier and mask to be the same value

Data is filled starting with the highest priority message object (1 to 32)

EOB bit (End of Buffer) set to 1 in last object (0 indicates this object belongs to a_FIFO)

Ethernet vs. CAN

<table>
<thead>
<tr>
<th></th>
<th>PRO</th>
<th>CON</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN</td>
<td>Robust, deterministic, real-time</td>
<td>Slower, not as widely adopted on HMI*-side of industrial control</td>
</tr>
<tr>
<td>Ethernet</td>
<td>Standardization in traditional office equipment, web-interface capability, high throughput</td>
<td>Non-real-time, non-deterministic</td>
</tr>
</tbody>
</table>

Both CAN and Ethernet (together and independently) have viable benefits in industrial applications

Gateways and floor terminals tend to be the most common places for CAN+Ethernet solutions

An ARM Cortex-based Ethernet hub means code compatibility throughout the entire system

* Human-Machine Interface
Stellaris CAN

Stellaris Integrated Controller Area Network (CAN)

Main Features
- Up to 3 Bosch-licensed CAN controllers
- Each supports CAN protocol version 2.0 part A/B
- Bit rates up to 1Mb/s
- 32 message objects, each with its own identifier mask
- Maskable interrupt
- Disable automatic retransmission mode for Time Triggered CAN (TTCAN)
- Programmable loop-back mode for self test operation

TI also provides
- Over 50 CAN-enabled Stellaris® ARM® Cortex™-M3 microcontrollers
- The EK-LM3S2965 CAN-network-in-a-CAN evaluation kit
- The EK-LM3S8962 CAN-network-plus-Ethernet evaluation kit
- Access to CAN quickstart applications and software examples from renowned CAN stack providers.
Kits and Software

**Ethernet+CAN Connected MCUs**

<table>
<thead>
<tr>
<th>Memory and Speed</th>
<th>Core</th>
<th>CAN Connected Interfaces</th>
<th>USB Interface</th>
<th>Power</th>
<th>I/O</th>
<th>Serial Interface</th>
<th>Analog</th>
<th>Digital</th>
<th>Package/Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM3S8962x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LM3S2110x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Stellaris LM3S8962 MCU with fully-integrated CAN module
- OLED graphics display with 128 x 64 pixel resolution
- User LED, navigation switches, and select pushbuttons
- Magnetic speaker
- LM3S8962 I/O available on labeled break-out pads
- Standard ARM® 20-pin JTAG debug connector with input and output modes

**Stellaris LM3S8962 Evaluation Kit**

$89

- First MCUs featuring fully integrated 10/100 Ethernet MAC+PHY and up to 3 Bosch CAN 2.0 A/B MACs
- IEEE 1588 Precision Time Protocol hardware assist

**CAN Connected MCUs**

<table>
<thead>
<tr>
<th>Memory and Speed</th>
<th>Core</th>
<th>CAN Connected Interfaces</th>
<th>USB Interface</th>
<th>Power</th>
<th>I/O</th>
<th>Serial Interface</th>
<th>Analog</th>
<th>Digital</th>
<th>Package/Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM3S2965x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LM3S2110x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Featuring up to 3 Bosch CAN 2.0 A/B CAN MACs
- Independent CAN buffer allows simultaneous CAN usage with all other peripherals

**Stellaris LM3S2965 Evaluation Kit**

$79

- OLED graphics display with 128 x 64 pixel resolution
- User LED, navigation switches, and select pushbuttons
- Magnetic speaker
- LM3S2965 I/O available on labeled break-out pads
- Standard ARM® 20-pin JTAG debug connector with input and output modes

- Standalone CAN device board using Stellaris LM3S2110 microcontroller
- CAN ribbon cable, USB and JTAG cables
Drivers

StellarisWare CAN Drivers

- StellarisWare has a full set of drivers for the CAN modules
- Bit timing calculations are often complex and difficult to understand. StellarisWare has an easy to use API called `CANBitTimingSet` that automatically configures the CAN bit timing based on a passed set of parameters
  - There are also a set of pre-configured bit timing settings for each speed that you can use
- The clocking on the Fury devices is different than the Dust Devil and Tempest devices
  - Fury uses an 8MHz dedicated clock when the PLL is enabled, and the crystal clock otherwise
  - Dust Devil and Tempest use the system clock, whether that is the main oscillator, internal oscillator, or PLL

APIs

Stellaris CAN APIs: Initialization and Interrupts

- **Initialization**
  - `CANInit()`
  - `CANBitTimingSet()`, `CANBitTimingGet()`
  - `CANBitRateSet()`
  - `CANEnable()`, `CANDisable()`
  - `CANRetrySet()`, `CANRetryGet()`

- **Interrupts**
  - `CANIntRegister()`, `CANIntUnregister()`
  - `CANIntEnable()`, `CANIntDisable()`
  - `CANIntStatus()`
  - `CANIntClear()`
Stellaris CAN APIs: Message Objects and Status

- **Message objects**
  - CANMessageSet()
  - CANMessageClear()
  - CANMessageGet()

- **Status**
  - CANStatusGet()
  - CANErrCntrGet()

Code Example – CAN Initialization

```c
// Configure CAN pins
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinTypeCAN(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

// Enable the CAN controllers
SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
CANInit(CAN0_BASE);

// Configure the clock rate 8MHz, bit rate set to 250K bits/sec
CANBitRateSet(CAN0_BASE, 8000000, 250000);
CANEnable(CAN0_BASE);

// Enable interrupts from CAN controller
CANIntEnable(CAN0_BASE, CAN_INT_MASTER | CAN_INT_ERROR);

// Set up the message objects that will receive all messages
CANConfigureNetwork();

// Enable interrupts for the CAN in the NVIC
IntEnable(INT_CAN0);
```

Code Example...
Message Objects Used in QS Application

- Four CAN message objects are used in qs_ek-lm3s8962 app (can_net.c file)
  - `g_MsgObjectButton` - receives button messages (Rx interrupt enabled) note: there are two buttons, and two events (press & release)
  - `g_MsgObjectLED` - sends LED brightness (Tx interrupt enabled)
  - `g_MsgObjectTx` - sends commands to 2110 (Tx interrupt enabled)
  - `g_MsgObjectRx` - receives commands from 2110 (Rx interrupt enabled)
- The “Message Object #” selects one of 32 locations in message object memory, also determines priority (lower # is higher)
- The “Message ID” is the 11-bit or 29-bit message identifier, in this application, no masking is needed or enabled

<table>
<thead>
<tr>
<th>Message Object#</th>
<th>symbol</th>
<th>Message ID</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MSGOBJ_NUM_BUTTON</td>
<td>0x10</td>
<td>MSGOBJ_ID_BUTTON</td>
</tr>
<tr>
<td>2</td>
<td>MSGOBJ_NUM_LED</td>
<td>0x12</td>
<td>MSGOBJ_ID_LED</td>
</tr>
<tr>
<td>3</td>
<td>MSGOBJ_DATA_TX</td>
<td>0x20</td>
<td>MSGOBJ_ID_DATA_TX</td>
</tr>
<tr>
<td>4</td>
<td>MSGOBJ_DATA_RX</td>
<td>0x21</td>
<td>MSGOBJ_ID_DATA_RX</td>
</tr>
</tbody>
</table>

Code Example – Configure Receive Object

- LM3S8962 board receives “button” messages from LM3S2110 CAN device board
- `g_MsgObjectButton` structure is initialized in `can_net.c` as follows:

```c
// Set the identifier and mask for the button object
    g_MsgObjectButton.ulMsgID = MSGOBJ_ID_BUTTON;
    g_MsgObjectButton.ulMsgIDMask = 0;
// This enables interrupts for received messages
    g_MsgObjectButton.ulFlags = MSGOBJ_RX_INT_ENABLE;
// Set the size of the message and the data buffer used
    g_MsgObjectButton.ulMsgLen = 2;
    g_MsgObjectButton.pucMsgData = g_pucButtonMsg;
// Configure the Button receive message object
    CANMessageSet(CAN0_BASE, MSGOBJ_NUM_BUTTON,
                   &g_MsgObjectButton, MSGOBJ_TYPE_RX);
```
Code Example – Configure Transmit Object

- **LM3S2110** board sends “button” messages to the **LM3S8962** board
- **g_MsgObjectButton** structure is initialized in *can_device_qs.c* as follows:

```c
// This is the message object used to send button updates. This
// message object will not be “set” using CANMessageSet right now
// as that would trigger a transmission
  g_MsgObjectButton.ulMsgID = MSGOBJ_ID_BUTTON;
  g_MsgObjectButton.ulMsgIDMask = 0;
// This enables interrupts for transmitted messages.
  g_MsgObjectButton.ulFlags = MSG_OBJ_TX_INT_ENABLE;
// Set message length, which should only be two bytes and the
// data is always whatever is in g_pucButtonMsg
  g_MsgObjectButton.ulMsgLen = 2;
  g_MsgObjectButton.pucMsgData = g_pucButtonMsg;
```

Code Example – Transmit Button Object

- **LM3S2110** board transmits a button message to the **8962** using
  **g_MsgObjectButton**
- **g_MsgObjectButton** has already been initialized except for the
  two bytes of data containing the event (press or release) and specific
  button (up or down)
- Once data is set, **CANMessageSet()** is called to transmit the
  message

```c
// ucEvent can be EVENT_BUTTON_PRESS (0x10)
// or EVENT_BUTTON_RELEASED (0x11)
  g_MsgObjectButton.pucMsgData[0] = ucEvent;
// ucButton can be TARGET_BUTTON_DN (1) or TARGET_BUTTON_UP (0)
  g_MsgObjectButton.pucMsgData[1] = ucButton;
// send the message
  CANMessageSet(CAN0_BASE, MSGOBJ_NUM_BUTTON, &g_MsgObjectButton,
               MSG_OBJ_TYPE_TX);
```
Kits and Software

Code Example – Receive Object

> 8962 board receives a button message from the 2110 using `g_MsgObjectButton` in the interrupt handler `CANHandler()``

```
CANHandler(void)
{
    unsigned long ulStatus;
    // Find the cause of the interrupt, if it is a status interrupt then just
    // acknowledge the interrupt by reading the status register.
    ulStatus = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE);
    switch (ulStatus)
    {
        case MSGOBJ_NUM_BUTTON:
        {
            // Read the Button Message.
            CANMessageGet(CAN0_BASE, MSGOBJ_NUM_BUTTON, &g_MsgObjectButton, 1);
            // . . . Rest of the code to handle the button message
            break;
        }
        // all other cases: MSGOBJ_NUM_LED, MSGOBJ_NUM_DATA_RX, MSGOBJ_NUM_DATA_TX
    }
    CANIntClear(CAN0_BASE, ulStatus);
}
```

Stacks

CAN Stacks

> The main two protocols used in CAN-based designs are CANopen and DeviceNet

- Other protocols include POWERLINK, EtherCAT, Profinet, EtherNetIP and also TCP/IP

http://www.micrium.com
http://www.rtautomation.com/
http://www.port.de
CAN Lab

Description:

We’ll use the LM3S8962 eval board and the LM3S2110 CAN device board to investigate CAN communication.

- LM3S8962 EVM
- LM3S2110 CAN Device board
- 10-pin CAN cable (not the 20-pin JTAG cable)
- USB cable
- Buttons on device board send CAN volume messages
- Select button on LM3S8962 board lights device LED
Hardware list:

- LM3S8962 Evaluation Board
- LM3S2110 CAN device board
- 10-pin CAN ribbon cable
- 3 meter, 4 pin, USB Type A-M to mini-USB Type B-M

Software list:

- Code Composer Studio 4.1
- LM3S8962 software examples
Procedure

1. **Open the IDE**
   Open Code Composer Studio and create a workspace_8962CAN workspace (we’re using a new workspace for each lab to prevent issues and learn the steps by repetition). Close the Welcome screen if it appears. Click the C/C++ Perspective button to make sure that you’re in the editing perspective.

2. **Import driverlib**
   From the menu bar, click on Project, and then select Import Existing CCS/CCE Eclipse Project. When the Import dialog appears, browse to the root directory of the driver library (C:\StellarisWare\driverlib). Click OK. Be sure that the checkbox next to driverlib in the Project pane is checked and that Copy projects into workspace is unchecked. Click Finish.

3. **Import the CAN project**
   From the menu bar, click on Project, and then select Import Existing CCS/CCE Eclipse Project. When the Import dialog appears, browse to the root directory of the quickstart project (C:\StellarisWare\boards\ek-lm3s8962\qs_ek-lm3s8962). Click OK. Be sure that the checkbox next to qs_ek-lm3s8962 in the Project pane is checked and that Copy projects into workspace is unchecked. Click Finish.

4. **Open can_net.c for editing**
   When the project has finished importing, close any open editor windows. Maximize the IDE window if you haven’t already and expand the qs_ek-lm3s8962 folder in the Projects pane.

   Double-click on can_net.c to open it for editing.

   can_net.c is made up of several modules:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANMain()</td>
<td>Handles connection to the other device and incoming messages</td>
</tr>
<tr>
<td>CANGetTargetVersion()</td>
<td>Retrieves the target board’s firmware version</td>
</tr>
<tr>
<td>CANUpdateTargetLED()</td>
<td>Sends message to set brightness of the LED on the target board</td>
</tr>
<tr>
<td>CANConfigureNetwork()</td>
<td>Configures message objects used</td>
</tr>
<tr>
<td>CANHandler()</td>
<td>CAN interrupt handler</td>
</tr>
<tr>
<td>CANConfigure()</td>
<td>Configures the CAN h/w and interrupts</td>
</tr>
</tbody>
</table>
As long as you’re poking around in the CAN code, here are some of the important objects to look for:

### Message Object Structure

```c
typedef struct {
    unsigned long ulMsgID;
    unsigned long ulMsgIDMask;
    unsigned long ulFlags;
    unsigned long ulMsgLen;
    unsigned char *pucMsgData;
} tCANMsgObject
```

**Members:**
- `ulMsgID` The CAN message identifier used for 11 or 29 bit identifiers.
- `ulMsgIDMask` The message identifier mask used when identifier filtering is enabled.
- `ulFlags` This value holds various status flags and settings specified by `tCANObjFlags`.
- `ulMsgLen` This value is the number of bytes of data in the message object.
- `pucMsgData` This is a pointer to the message object’s data.

**Description:**
The structure used for encapsulating all the items associated with a CAN message object in the CAN controller.

tCANObjFlags – these are the flag bits in the CAN message object

- **MSG_OBJ_TX_INT_ENABLE** This indicates that transmit interrupts should be enabled, or are enabled.
- **MSG_OBJ_RX_INT_ENABLE** This indicates that receive interrupts should be enabled, or are enabled.
- **MSG_OBJ_EXTENDED_ID** This indicates that a message object will use or is using an extended identifier.
- **MSG_OBJ_USE_ID_FILTER** This indicates that a message object will use or is using filtering based on the object’s message identifier.
- **MSG_OBJ_NEW_DATA** This indicates that new data was available in the message object.
- **MSG_OBJ_DATA_LOST** This indicates that data was lost since this message object was last read.
- **MSG_OBJ_USE_DIR_FILTER** This indicates that a message object will use or is using filtering based on the direction of the transfer. If the direction filtering is used, then ID filtering must also be enabled.
- **MSG_OBJ_USE_EXT_FILTER** This indicates that a message object will use or is using message identifier filtering based on the extended identifier. If the extended identifier filtering is used, then ID filtering must also be enabled.
- **MSG_OBJ_REMOTE_FRAME** This indicates that a message object is a remote frame.
- **MSG_OBJ_NO_FLAGS** This indicates that a message object has no flags set.
5. **Test the current software**
   Before we change anything, let’s make sure that the project works as it is. Connect your 8962 board to your laptop. From the menu bar, click **Project** and then select **Build Active Project**. Sometimes it’s handy to build the project and check for errors without attempting to load it to Flash memory.

6. **Debug and Run**
   When the build completes, we need to load the program to the board and start the debugger. Click **Target** on the menu bar, then click **Debug Active Project** or you can simply click the **Debug** button.

   The program will load to the 8962 Flash memory and the perspective will switch to **Debug**. Click the **Run** button to start program execution. The maze game on the OLED display should appear and the speaker should emit the game sounds.

7. **Back to the Editor**
   Click the **C/C++ Perspective** button (in the upper right) to return to the editor. Page down and find the **CANHandler()** module in **can_net.c**. This module handles the interrupt caused by received CAN messages. When you have more time, you can look at where those messages were sent by exploring the **can_device_qs** project that runs on the LM3S2110.

8. **Look at CANUpdateTargetLED()**
   We’d like to add some code that will turn the LM3S2110 board status LED on when the up button is released and off when the down button is released.

   If you haven’t looked closely at the **CANUpdateTargetLED()** module, now would be a good time to do so. Note that there are two passed parameters:

   - **ucLevel**
     - 0 is off and 1 is on
   - **bFlash**
     - true flashes the LED, false does not

   Calling **CANUpdateTargetLED()** with the correct parameters sends a CAN message from the LM3S8962 board to the LM3S2110 board.
9. **Add some code**

Find the `CANHandler()` module and note how `AudioVolume()` is called to alter the game volume. Add the calls to `CANUpdateTargetLED()` as shown below:

```c
if(g_MsgObjectButton.pucMsgData[1] == TARGET_BUTTON_UP)
{
    // Adjust the volume up by 10.
    AudioVolumeUp(10);
    CANUpdateTargetLED(1, false);  //LED ON
}

// Check if the down button was released.
//
if(g_MsgObjectButton.pucMsgData[1] == TARGET_BUTTON_DN)
{
    // Adjust the volume down by 10.
    AudioVolumeDown(10);
    CANUpdateTargetLED(0, false);  //LED OFF
}
}
break;
```
10. **Change Preferences**
   It would be nice not to have to Build and Debug in two separate steps. We can change a selection so that clicking on Debug will rebuild the project if needed.

   On the menu bar, click **Window**, then **Preferences**. Click on the + next to CCS, and then click on **Debug**. Check the **Rebuild the program (if required) before loading** checkbox. Click OK.

11. **Build/Load/Run**
   Click the ![Debug](debug_icon.png) **Debug** button. Save the changes to **can_net.c** and reload the **.out** file when prompted. The project will automatically build and load to the 8962 Flash memory.

   Change the perspective to **Debug**. Click the ![Run](run_icon.png) **Run** button to start program execution. The maze game on the OLED display should appear and the speaker should emit the game sounds.

12. **Testing our changes**
   After you click the **resume** button, don’t touch the **up/down** buttons until **Press Select To Play** appears on the OLED. At that point, pressing the **up** button on the LM3S2110 sends a **VolumeUp CAN** message to the **LM3S8962 board**, which then responds by sending a CAN message to turn on the status LED on the **LM3S2110 board**. The down button works in a similar fashion to turn the LED off.

   If you wait too long and the screen saver appears, press the **Select** button on the **LM3S8962 board** to exit the screen saver mode.

13. **Remove changes**
   Go back to the **C/C++ perspective** and edit the **can_net.c** file. Comment out the two lines that you added to the code. **Build/Load/Run** again to make sure you’ve removed the changes.

   Remember that the QuickStart application has been replaced by this application. You can reprogram it at any time with either Code Composer Studio or the Flash Programmer.

   ![STOP](stop_icon.png)

   You’re done
*** A mind is the only thing that can really be blank ***