**CAN IMPLEMENTATION**

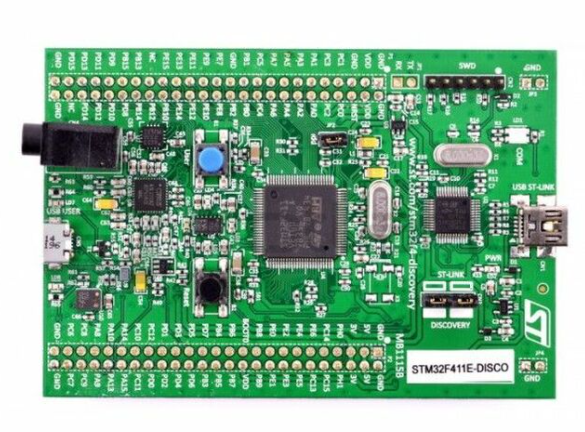
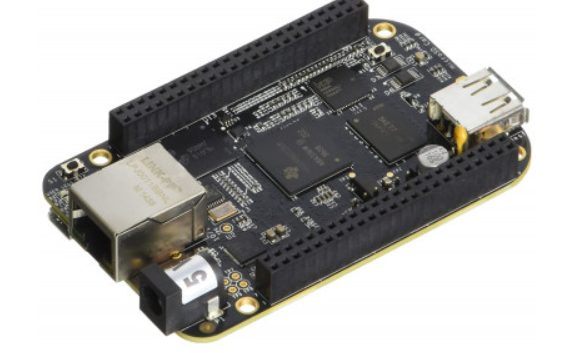
“Working with CAN is not a **piece of CAKE**”

Interfacing of CAN protocol on **STM32** :

CAN implementation broadly divided into two:

1. Care to be taken on **Hardware** end
2. Care to be taken on **Software** end
3. **Points to remember when dealing with hardware**

Hardware includes =>STM32 board, CAN transceivers, cables(End Point of Cables should be tight), Beagle Bone Black & Connections.

Connections:



STM32 → CAN TRX → CAN TRX → Beagle Bone

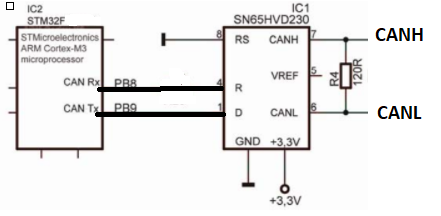
3.3V 3.3V 3.3V VCC/3.3V

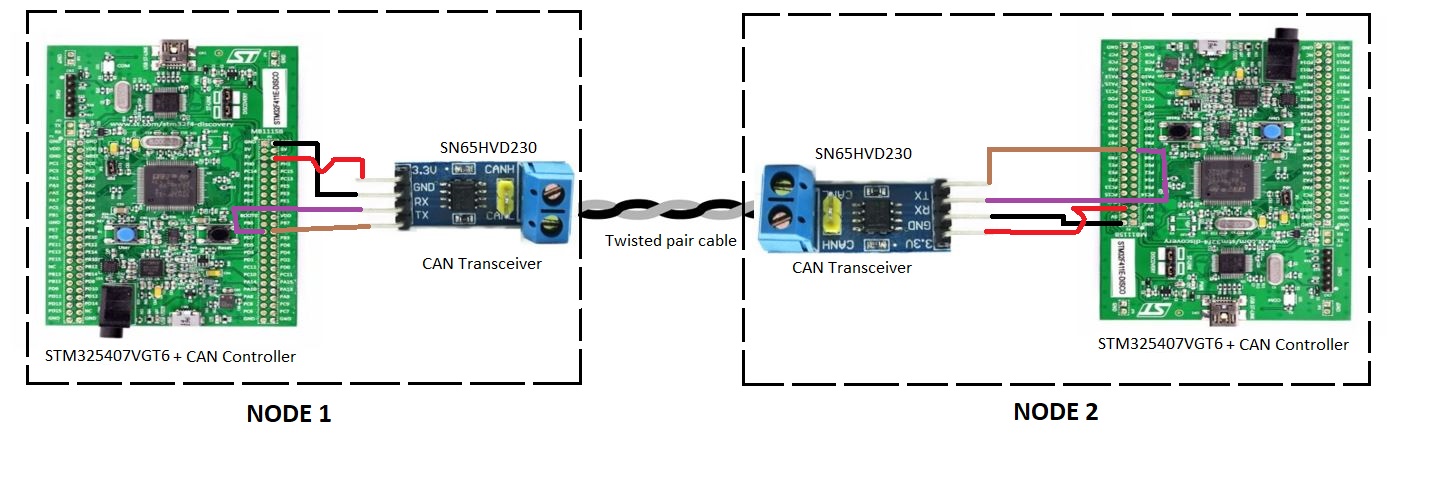
GND GND GND GND

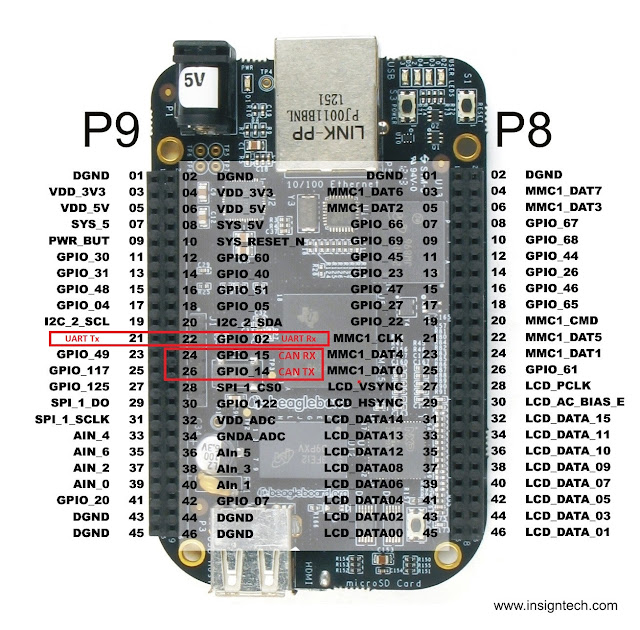
CAN\_Rx RX CANH CANH RX CAN\_Rx

CAN\_Tx TX CANL CANL TX CAN\_Tx

**\*Note: Also make sure to have a Common Ground for STM and BBB**







**Note:-Refer the above pins for connection between Can transceiver to Beaglebone**

● Working of CAN transceivers must be checked thoroughly.

Operating voltage (**3.3V[SN65VD230]** or 5V), terminating resistor between CANH & CANL

(120 ohms)

Purpose of 120 Ohm resistance is to avoid Reflections(due to Impedance Mismatching) in the Transmission Lines transmitting data at a high speed:

CANTx of STM -CANTx of Transceiver

CANRx of STM -CANRx of Transceiver

* Check if CANTx or CANRx of Transceiver is working fine.

In some cases CANTx of transceivers is not working but CANRx of transceivers is working.

Check the pins on IC or change the transceivers from another vendor and then check.

* **Connections must be tight while working with CAN** (minimum noise on CAN bus).
* CAN Transceivers must have proper soldering done (Check that No Joint should be dry soldered)
* Connections on breadboard must be tight (voltage properly applied).
* Directly connect the CANH & CANL using twisted pair cable & make sure the terminating resistors are present.
* Check the Silk Screen print on both the sides of IC is same .

1. **Points to remember when dealing with Software part**

Software includes=>STMCubeIDE

* SystemCore->RCC->HSE->Crystal/Ceramic Oscillator
* Clock Configuration(RCC external clock from oscillator) must be same at both Tx-Rx side

External clock on STM32F407VGTx HSE is on 8Mhz,

* Default Pins for CAN1 are PA11, PA12 as these Pins are Not Available as a Part of Headers Pins so Replace Them with PB8,PB9 (Beside Reset Button), to Replace PIN just do ***Ctrl + Click on the PA11,12 to show Alternate***
* CAN Configuration (**CAN1 is working on APB1 clock**, Prescaler, Time Quantas, Mode of operation) must be same, preferable clock on APB1 are **24Mhz(Preferred )**
* Preferable CAN protocol bit rate 62.5Kbps, 125Kbps[Speed upto 125kbps is Low speed CAN] & 500kbps[Speed upto 1Mbps is High Speed CAN].
* If you are using CAN Interrupt for receiving methods then don’t forget to use Receive Callback, If you are using CANRx Polling don’t forget to use RxFiFoFillLevel API.

**Getting Started with CAN initialization:**

**CAN Bus has two initialization modes:**

1. **Normal Mode:** for normal CAN Operation using at least two nodes
2. **Loopback Mode:** for Testing CAN BUS with single node

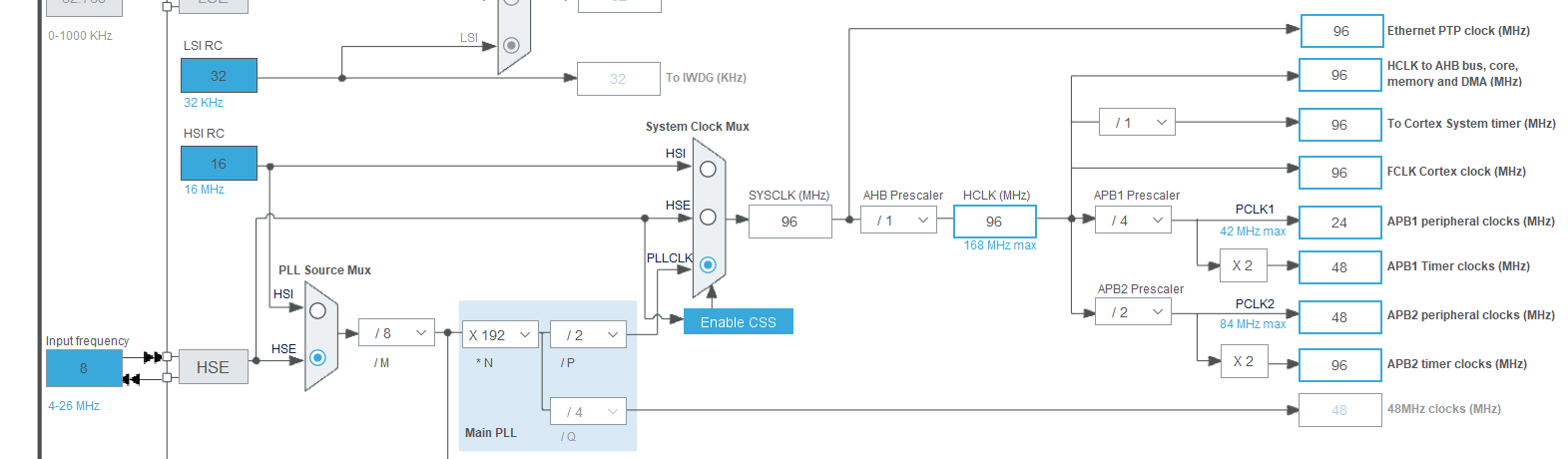
**Note: In any of the above mode Transceiver must be attached.**

Before interfacing CAN in Normal mode, check CAN in loopback mode i.e. transmit from CAN1 & receive on CAN1 internally in software.

1. Check the HAL\_API version & level in STMCubeMx before generating code.

->This is the Clock Configuration to achieve **24 Mhz** at APB1

->External Oscillator is on 8 MHz, APB1 for CAN operates on 24Mhz.



2. CAN Initialization (Prescaler, Time Qantas, Mode Selection)-

Basically the CAN bit period can be subdivided into four time segments. Each time segment consists of a number of Time Quanta (tq).

**The Time Quanta(Tq) is the smallest time unit for all configuration values.**

**#Configuring Pre Scalar for Tq**

**Example:**

**Preferred 1:**

**For Baud rate of 62.5Kbps if Pre-scalar = 24, Clock APB1=24Mhz -> 24Mhz/24 = 1Mhz=1us =1000ns**

**Preferred 2:**

**For Baud rate of 125Kbps if Pre-scalar = 12, Clock APB1=24Mhz -> 24Mhz/12 = 2Mhz=0.5us =500ns**

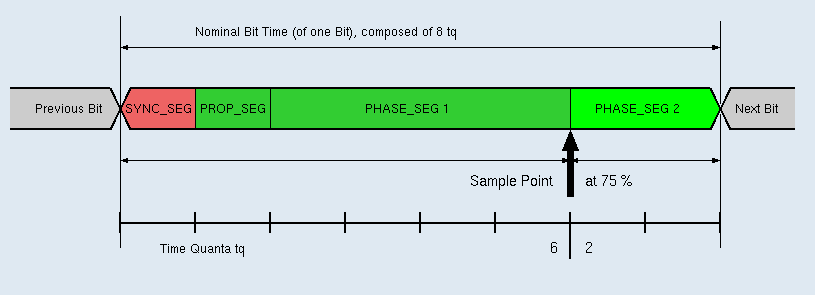
**Preferred 3:**

**For Baud rate of 500Kbps if Pre-Scalar = 3, Clock APB1=24Mhz -> 24Mhz/3 = 8Mhz=0.125us=125ns**

**What will be the Tq for baud rate 250Kbps, 1Mbps?**

* **SYNC\_SEG** is 1 Time Quantum long. It is used to synchronize the various bus nodes.
* **PROP\_SEG** is programmable to be 1, 2,... 8 Time Quanta long. It is used to compensate for signal delays across the network.
* **PHASE\_SEG1** is programmable to be 1,2, ... 8 Time Quanta long. It is used to compensate for edge phase errors and may be lengthened during resynchronization.
* **PHASE\_SEG2** (Seg 2) is the maximum of PHASE\_SEG1 and the Information Processing Time long. It is also used to compensate for edge phase errors and may be shortened during resynchronization. For this the minimum value of PHASE\_SEG2 is the value of SJW.

The picture shows you the time segments of a CAN-Bit as defined by ISO-11898.



|  |  |
| --- | --- |
| Sync\_Seg: | 1 tq |
| Prop\_Seg + Phase\_Seg1: | 1 .. 16 tq |
| Phase\_Seg2: | 1 .. 8 tq |
| (Table calculation uses Prop\_Seg = 0) | |

### Bit Rate Calculation Examples

**Q.** **Calculate Required Time Quanta to achieve different baud rate if the system clock is 24mhz and CAN Pre Scalar is 24 or 12 or 3.**

|  |  |  |  |
| --- | --- | --- | --- |
| Input Freq. at APB1 Bus | 24mhz | 24mhz | 24mhz |
| Prescaler | 24 | 12 | 3 |
| CAN Clock(fcan) | 24mhz/24=1mhz | 24/12=2mhz | 24/3=8mhz |
| Time Quanta(tq)=1/fcan | 1/1mhz=1000ns | 1/2mhz=500ns | 1/8mhz=125ns |
| **Bit rate** | **62.5kbps** | **125kbps** | **500kbps** |
| Bit time =1/Bit rate | 1/62.5kbps=16000ns | 1/125kbps=8000ns | 1/500kbps=2000ns |
| No of tq= Bit time/tq | 16000ns/1000ns=16tq | 8000ns/500ns=16tq | 2000ns/125ns=16tq |
| TqSEG1 | 13 | 13 | 13 |
| TqSEG2 | 2 | 2 | 2 |
| ReSync | 1 | 1 | 1 |

**Q. Configure TSEG1 and TSEG2 to set sampling at 80% of a bit time.**

(Tsync\_seg+TSEG1)/(Tsync\_seg+TSEG1+TSEG2)=80%

as we calculated above: bit time = Tsync\_seg+TSEG1+TSEG2=16tq

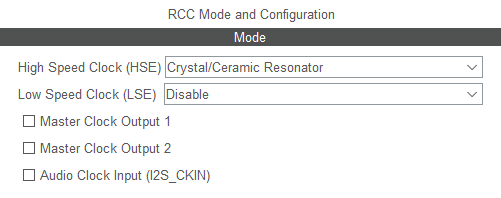
so , (1+TSEG1)/(16)=80%

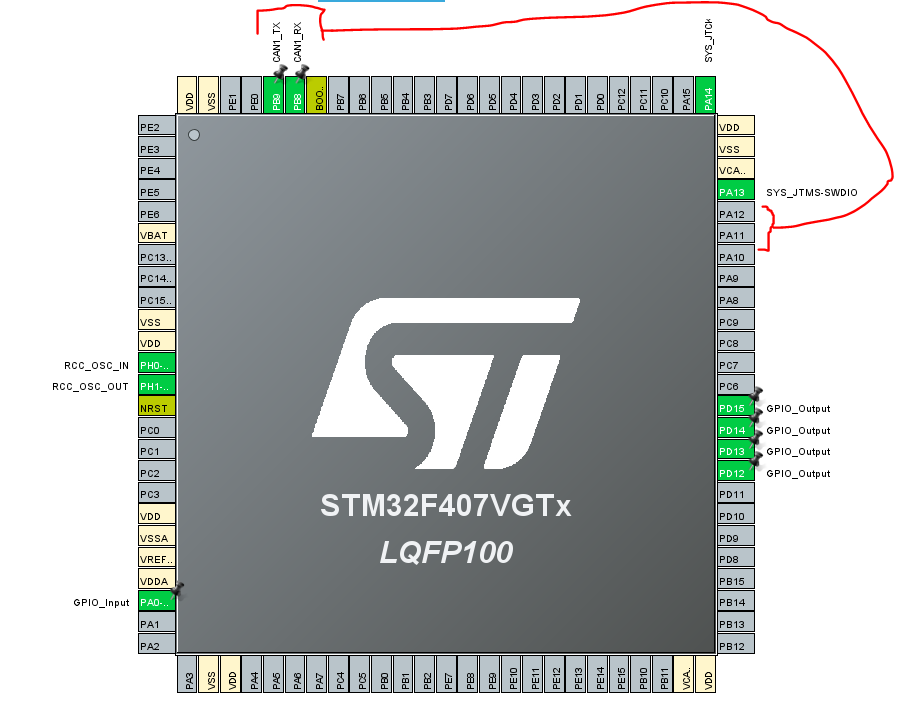
TSEG1=14-1=13

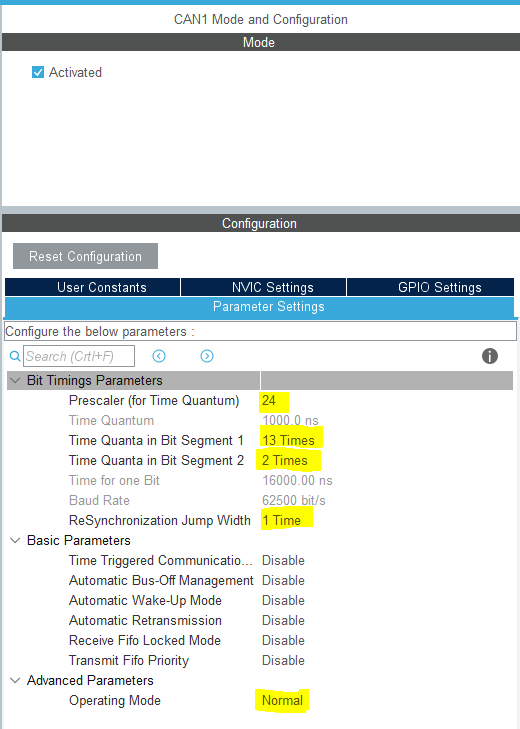
so TSEG2=20-1-15=2

ReSync=1 [Fixed]

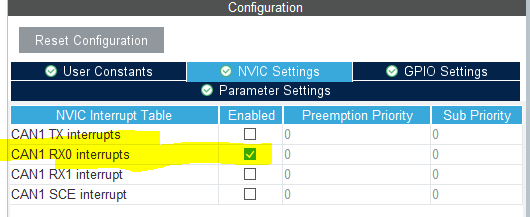
**STM32CUBE IDE Settings**

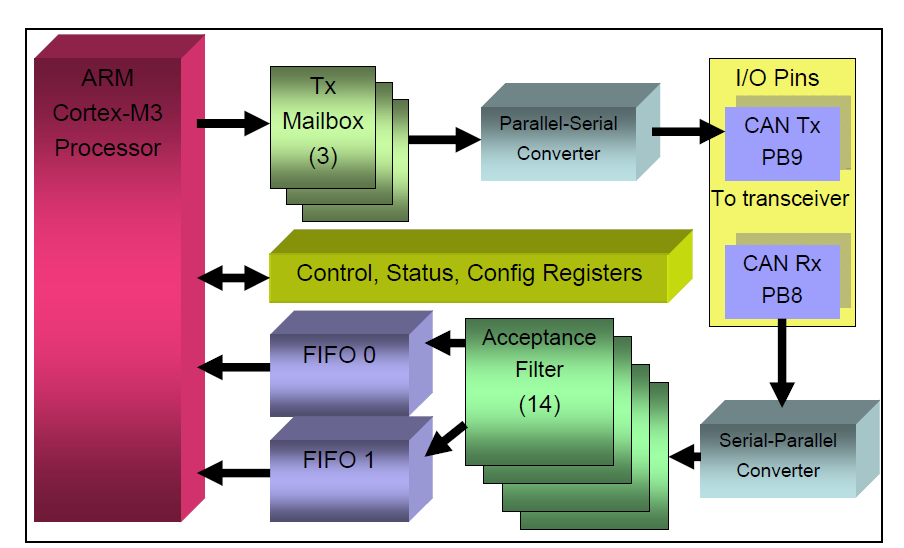






3. Also Enable CANRx Interrupt & set subpriority-





4. Generate the code & start with CAN configuration-

* **Private Variables and Functions**

|  |
| --- |
| /\* USER CODE BEGIN PV \*/ |
| uint8\_t ubKeyNumber = 0x0; //Counter Variable |
| CAN\_TxHeaderTypeDef TxHeader; // CAN Data Frame header fields like RTR Bit, DLC, ID |
| CAN\_RxHeaderTypeDef RxHeader; // CAN Data Frame header fields like RTR Bit, DLC, ID |
| uint8\_t TxData[8]; //Actual Payload |
| uint8\_t RxData[8]; //Actual Payload |
| uint32\_t TxMailbox; //Buffer for Tx Messages |
| /\* USER CODE END PV \*/ |
|  |
| /\* USER CODE BEGIN PFP \*/ |
| **void** **CAN\_filterConfig**(**void**); //Acceptance filter Configuration |
| **void** **LED\_Display**(uint8\_t LedStatus); |
| /\* USER CODE END PFP \*/ |

* **CAN Acceptance filter Configuration (filter bank, FilterIdHigh, FilterIdLow etc..)**
  + **This config will remain same for Polling and Interrupt**

|  |  |
| --- | --- |
| **//Can Filter config function to Accept All the message** | |
| **static void** **CAN\_filterConfig** (**void**) | //CAN Std ID = 11bits |
| { | //Filter Scale is 32bits which is divided into FilterIdHigh & FilterIdLow each 16bits |
| CAN\_FilterConfTypeDef sFilterConfig; |  |
| sFilterConfig.FilterNumber = 0; |  |
| sFilterConfig.FilterMode = CAN\_FILTERMODE\_IDMASK; |  |
| sFilterConfig.FilterScale = CAN\_FILTERSCALE\_32BIT; |  |
| sFilterConfig.FilterIdHigh = 0x000 <<5; | //we want to fit CAN Std ID(11bits) in FilterIdHigh(16bits) so <<5 is required |
| sFilterConfig.FilterIdLow = 0x0000; | //E.g. Std ID =0x123 ,if we shift it to <<5 it will fit MSB of Std ID in 16bits of FilterIdHigh |
| sFilterConfig.FilterMaskIdHigh = 0x0000; //0xFFE0 | //0xFFE0 are High 11 bits of 16bit MaskIdHigh Reg. |
| sFilterConfig.FilterMaskIdLow = 0x0000; | // if ((IdHigh & MaskIdHigh) == ID) then ACCEPT MSG; |
| sFilterConfig.FilterFIFOAssignment = 0; | //Eg if((0x123<<5) & (0xFFE0) == 0x123) then 0x123 is ACEEPTED |
| sFilterConfig.FilterActivation = *ENABLE*; | //Eg if((0x123<<5) & (0xFFE0) == 0x456) then 0x456 is REJECTED |
| sFilterConfig.BankNumber = 14; |  |
|  |  |
| **if**(HAL\_CAN\_ConfigFilter(&hcan1, &sFilterConfig) != *HAL\_OK*) | |
| { |  |
| /\* Filter configuration Error \*/ |  |
| Error\_Handler(); |  |
| } |  |
| } |  |

**CAN Polling**

* CAN Start using HAL\_APIs

|  |
| --- |
| \* USER CODE BEGIN 2 \*/ |
| /\*##Filter Configuration ###########################################\*/ |
| CAN\_filterConfig(); |
|  |
| /\*##Start the CAN peripheral ###########################################\*/ |
| **if** (HAL\_CAN\_Start(&hcan1) != *HAL\_OK*) |
| { |
| /\* Start Error \*/ |
| Error\_Handler(); |
| } |
|  |
| /\*## Transmit Data ###########################################\*/ |
| CAN\_TransmitData(); |
|  |
| /\*## Receive Data ###########################################\*/ |
| **if**(CAN\_ReceiveData() == 1) |
| { |
| /\* OK: Turn on LED1 \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12, *SET*);//green led |
| } |
| **else** |
| { |
| /\* KO: Turn on LED2 \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13, *SET*);//orange led |
| } |
| /\* USER CODE END 2 \*/ |

5. CAN Transfer & Receive:

* Use CAN Transmit & Receive APIs-

|  |
| --- |
| **void** **CAN\_TransmitData**(**void**) |
| { |
| /\*##- Start the Transmission process #####################################\*/ |
| TxHeader.StdId = 0x11; |
| TxHeader.RTR = CAN\_RTR\_DATA; |
| TxHeader.IDE = CAN\_ID\_STD; |
| TxHeader.DLC = 2; |
| TxHeader.TransmitGlobalTime = *DISABLE*; |
| TxData[0] = 0xCA; |
| TxData[1] = 0xFE; |
|  |
| /\* Request transmission \*/ |
| //API HAL\_CAN\_AddTxMessage to request a transmission of new Message |
| **if**(HAL\_CAN\_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox) != *HAL\_OK*) |
| { |
| /\* Transmission request Error \*/ |
| Error\_Handler(); |
| } |
|  |
| /\* Wait for transmission to complete \*/ |
| //HAL\_CAN\_GetTxMailboxesFreeLevel To get no of free Tx Mailboxes, wait until at least 3 Mailbox is free |
| **while**(HAL\_CAN\_GetTxMailboxesFreeLevel(&hcan1) != 3) {} |
| } |
|  |
| **int** **CAN\_ReceiveData**(**void**) |
| { |
| /\*##- Start the Reception process ########################################\*/ |
| //Monitor the Reception of the Message using HAL\_CAN\_GetRxFifoFillLevel until at least one Msg is Received |
| **if**(HAL\_CAN\_GetRxFifoFillLevel(&hcan1, CAN\_RX\_FIFO0) != 1) |
| { |
| /\* Reception Missing \*/ |
| Error\_Handler(); |
| } |
|  |
| //Get the Msg using HAL\_CAN\_GetRxMessage |
| **if**(HAL\_CAN\_GetRxMessage(&hcan1, CAN\_RX\_FIFO0, &RxHeader, RxData) != *HAL\_OK*) |
| { |
| /\* Reception Error \*/ |
| Error\_Handler(); |
| } |
|  |
| **if**((RxHeader.StdId != 0x11) || |
| (RxHeader.RTR != CAN\_RTR\_DATA) || |
| (RxHeader.IDE != CAN\_ID\_STD) || |
| (RxHeader.DLC != 2) || |
| ((RxData[0]<<8 | RxData[1]) != 0xCAFE)) |
| { |
| /\* Rx message Error \*/ |
| **return** 0; |
| } |
|  |
| **return** 1; |
|  |
| } |

6. Error Handler:

|  |
| --- |
| **void** **Error\_Handler**(**void**) |
| { |
| /\* USER CODE BEGIN Error\_Handler\_Debug \*/ |
| /\* User can add his own implementation to report the HAL error return state \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_14, *SET*); |
| /\* USER CODE END Error\_Handler\_Debug \*/ |
| } |

**CAN Interrupt**

**Note:** Sometimes **HAL\_CAN\_ActivateNotification(&hcan1, CAN\_IT\_RX\_FIFO0\_MSG\_PENDING)** won't allow the STM to be in the Debugging Session or Running the code, then you need to comment out this API and Run Code or Debug the code after that uncomment the API and Re-run the code or again Debug the Code then it works fine.[It's a Work Around]

|  |
| --- |
| /\* USER CODE BEGIN 2 \*/ |
|  |
| /\*##-Step1:Filter Configuration ###########################################\*/ |
| CAN\_filterConfig(); |
|  |
| /\*##-Step2:Start the CAN peripheral ###########################################\*/ |
| **if** (HAL\_CAN\_Start(&hcan1) != *HAL\_OK*) |
| { |
| /\* Start Error \*/ |
| Error\_Handler(); |
| } |
| /\*##-Step3:Activate CAN RX notification i.e. INTERRUPTS ########################\*/ |
| **if** (HAL\_CAN\_ActivateNotification(&hcan1, CAN\_IT\_RX\_FIFO0\_MSG\_PENDING) != *HAL\_OK*) |
| { |
| // Notification Error |
| Error\_Handler(); |
| } |
|  |
| /\*##-Step4:Configure Transmission process #####################################\*/ |
| TxHeader.StdId = 0x123; |
| TxHeader.RTR = CAN\_RTR\_DATA; |
| TxHeader.IDE = CAN\_ID\_STD; |
| TxHeader.DLC = 2; |
| TxHeader.TransmitGlobalTime = *DISABLE*; |
| /\* USER CODE END 2 \*/ |

**Things to keep in while(1)**

|  |
| --- |
| /\* USER CODE BEGIN WHILE \*/ |
| **while** (1) |
| { |
| **if** (HAL\_GPIO\_ReadPin(GPIOA, GPIO\_PIN\_0) == *GPIO\_PIN\_SET*) |
| { |
| HAL\_Delay(50);//Debouncing Delay |
| **if** (HAL\_GPIO\_ReadPin(GPIOA, GPIO\_PIN\_0) == *GPIO\_PIN\_SET*) |
| { |
| **if** (ubKeyNumber == 0x4) |
| { |
| ubKeyNumber = 0x00; |
| } |
| **else** |
| { |
| LED\_Display(++ubKeyNumber); |
|  |
| /\* Set the data to be transmitted \*/ |
| TxData[0] = ubKeyNumber; |
| TxData[1] = 0xAD; |
|  |
| /\* Start the Transmission process \*/ |
| **if** (HAL\_CAN\_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox) != *HAL\_OK*) |
| { |
| /\* Transmission request Error \*/ |
| Error\_Handler(); |
| } |
| HAL\_Delay(100);//Delay just for better Tuning |
| } |
| } |
| } |
| /\* USER CODE END WHILE \*/ |

**Callback for Interrupt**

|  |
| --- |
| **void** **HAL\_CAN\_RxFifo0MsgPendingCallback**(CAN\_HandleTypeDef \*hcan) |
| { |
| /\* Get RX message from Fifo0 as message is Pending in Fifo to be Read \*/ |
| **if** (HAL\_CAN\_GetRxMessage(hcan, CAN\_RX\_FIFO0, &RxHeader, RxData) != *HAL\_OK*) |
| { |
| /\* Reception Error \*/ |
| Error\_Handler(); |
| } |
|  |
| /\* Display LEDx \*/ |
| **if** ((RxHeader.StdId == 0x123) && (RxHeader.IDE == CAN\_ID\_STD) && (RxHeader.DLC == 2)) |
| { |
| LED\_Display(RxData[0]); |
| ubKeyNumber = RxData[0]; |
| } |
| } |
|  |
| /\*\* |
| \* @brief Turns ON/OFF the dedicated LED. |
| \* @param LedStatus: LED number from 0 to 3 |
| \* @retval None |
| \*/ |
| **void** **LED\_Display**(uint8\_t LedStatus) |
| { |
| /\* Turn OFF all LEDs \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12|GPIO\_PIN\_13|GPIO\_PIN\_14|GPIO\_PIN\_15, *GPIO\_PIN\_RESET*); |
|  |
| **switch**(LedStatus) |
| { |
| **case** (1): |
| /\* Turn ON LED1 \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12, *GPIO\_PIN\_SET*); |
| **break**; |
|  |
| **case** (2): |
| /\* Turn ON LED2 \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13, *GPIO\_PIN\_SET*); |
| **break**; |
|  |
| **case** (3): |
| /\* Turn ON LED3 \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_14, *GPIO\_PIN\_SET*); |
| **break**; |
|  |
| **case** (4): |
| /\* Turn ON LED4 \*/ |
| HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_15, *GPIO\_PIN\_SET*); |
| **break**; |
| **default**: |
| **break**; |
| } |
| } |
| /\* USER CODE END 4 \*/ |

**CAN in RTOS**

- Implement an **interrupt-driven** CAN BUS driver, as interrupt-based drivers are crucial when working with an RTOS.

- Create a task dedicated to handling CAN BUS messages, **including receiving,** decoding, and notifying other system components.

- This event-driven approach makes the system much more efficient.

- The interrupt signaling the reception of a new CAN frame can wake up a task to process the frame, eliminating the need for polling.

- Avoid polling for CAN messages, as it wastes CPU time and power when there's nothing to process.

- Organize the rest of your design into separate tasks, ideally by creating a task for each asynchronous input or related groups of inputs.

**CAN in BareMetal**

### Key Terms for CAN Initialization

1. **Registers**:
   * **MCR (Master Control Register)**: Used to configure the CAN controller.
   * **MSR (Master Status Register)**: Provides status information about the CAN controller.
2. **Bit Manipulation Functions**:
   * sbi(register, bit): Sets a bit in a register.
   * cbi(register, bit): Clears a bit in a register.
   * cb(register, bit): Reads the state of a bit in a register.

**void can\_enter\_init()**

**{**

**sbi(CAN1->MCR, 0); // Set bit 0 (INRQ) in MCR to enter initialization mode.**

**cbi(CAN1->MCR, 1); // Clear bit 1 (SLEEP) in MCR to exit sleep mode. SLEEP is auto-set on reset.**

**sbi(CAN1->MCR, 4); // Set bit 4 (No Retransmit) in MCR: messages are transmitted once.**

**// Wait until the initialization mode is confirmed**

**while (!(cb(CAN1->MSR, 0))); // Bit 0 (INAK) in MSR is 0 normally; waits until it is set (init mode).**

**// Ensure the controller is not in sleep mode**

**while (cb(CAN1->MSR, 1)); // Bit 1 (SLAK) in MSR indicates sleep; waits until cleared (awake).**

**}**

#### Steps:

1. **Initialization Mode**: By setting INRQ in MCR, the CAN controller requests to enter initialization mode.
2. **Exit Sleep Mode**: Clearing the SLEEP bit ensures the controller is awake.
3. **No Retransmission**: Setting bit 4 ensures that messages will not be retransmitted.
4. **Polling for Status**: The function waits until:
   * The INAK bit in MSR confirms that initialization mode is active.
   * The SLAK bit in MSR is cleared, confirming the controller is not in sleep mode.

This function below exits initialization mode and puts the CAN peripheral into normal mode.

**void can\_exit\_init()**

**{**

**cbi(CAN1->MCR, 0); // Clear bit 0 (INRQ) in MCR to enter normal mode.**

**cbi(CAN1->MCR, 1); // Clear bit 1 (SLEEP) in MCR to exit sleep mode.**

**// Wait until the CAN controller is synchronized and ready for normal mode**

**while ((cb(CAN1->MSR, 0)) && (cb(CAN1->MSR, 1)));**

**// This checks that both `INAK` (init mode) and `SLAK` (sleep mode) are cleared.**

**}**

#### Steps:

1. **Normal Mode**: Clearing the INRQ bit allows the CAN controller to transition to normal mode.
2. **Exit Sleep Mode**: Clearing the SLEEP bit ensures the controller is awake.
3. **Wait for Synchronization**: The controller waits for 11 consecutive recessive bits for synchronization before entering normal mode. This is done by monitoring:
   * INAK to confirm exit from initialization mode.
   * SLAK to confirm the controller is not in sleep mode.

### Key Points to Note

1. **Bit Settings**:
   * **INRQ** (Initialization Request): Controls entry/exit from initialization mode.
   * **SLEEP**: Puts the CAN controller into low-power sleep mode.
   * **INAK** (Initialization Acknowledge): Indicates if initialization mode is active.
   * **SLAK** (Sleep Acknowledge): Indicates if the controller is in sleep mode.
2. **Polling for Status**: Both functions use while loops to wait until specific bits in the MSR register reflect the desired state.
3. **No Retransmission**: The No Retransmit setting in MCR is application-specific and might be useful in scenarios where repeated messages could cause issues.

**CAN Baud Rate Setup at BareMetal Level**

**void can\_testmode()**

**{**

**unsigned int brp;**

**can\_setup(); // Call to initialize or setup the CAN peripheral (shown below).**

**can\_enter\_init(); // Put the CAN controller into initialization mode for configuration.**

**// Calculate the Baud Rate Prescaler (BRP)**

**brp = (42000000 / 7) / 500000;**

**/\* The formula ensures the CAN clock is divided to achieve a desired baud rate.**

**- CAN clock: 42 MHz (assumed)**

**- Time Quanta (TQ): 7 per bit**

**- Baud rate: 500 kbps \*/**

**/\* Set the Bit Timing Register (BTR) \*/**

**// Configure the CAN controller to achieve a sample point at ~71% of the bit time.**

**// TSEG1 = 4, TSEG2 = 2, SJW = 3.**

**// 1 CAN bit = 7 Time Quanta (TQ), sample point = (4+1)/(7) ≈ 71%.**

**// Clear all fields in BTR related to timing, SJW, and BRP**

**CAN1->BTR &= ~(((0x03) << 24) | ((0x07) << 20) | ((0x0F) << 16) | (0x3FF));**

**// Set BTR fields for:**

**// - SJW (Synchronization Jump Width): 3-1 = 2 (0-indexed)**

**// - TSEG2: 2-1 = 1**

**// - TSEG1: 4-1 = 3**

**// - BRP: brp-1**

**CAN1->BTR |= ((((3 - 1) & 0x03) << 24) | (((2 - 1) & 0x07) << 20) | (((4 - 1) & 0x0F) << 16) | ((brp - 1) & 0x3FF));**

**// Clear Loopback and Silent mode bits in BTR**

**CAN1->BTR &= ~((1 << 30) | (1 << 31));**

**// Uncomment the line below to enable Loopback and Silent modes for testing.**

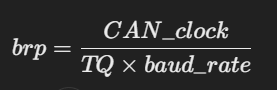
**// CAN1->BTR |= ((1 << 30) | (1 << 31));**

**// Exit initialization mode and start CAN communication**

**can\_exit\_init();**

**}**

### Step-by-Step Breakdown

1. **Initial Setup**:
   * can\_setup() and can\_enter\_init() are called to set up the CAN peripheral and put it into initialization mode, where configuration changes can be made.
2. **Baud Rate Calculation**:
   * The baud rate prescaler (brp) is calculated using:
   * 
     + CAN\_clock = 42 MHz
     + TQ = 7 (time quanta per CAN bit)
     + baud\_rate = 500 kbps
3. **Bit Timing Configuration**:
   * The BTR register (Bit Timing Register) controls the timing of CAN frames. It's cleared first to reset all relevant fields.
   * Fields are then set:
     + **SJW (Synchronization Jump Width)**: 3 (encoded as 3-1).
     + **TSEG2 (Time Segment 2)**: 2 (encoded as 2-1).
     + **TSEG1 (Time Segment 1)**: 4 (encoded as 4-1).
     + **BRP (Baud Rate Prescaler)**: Calculated brp-1.

These settings ensure a 7 TQ bit duration, with the sample point at approximately 71% of the bit time.

1. **Test Modes**:
   * The BTR register has two test mode bits:
     + **Loopback Mode** (bit 30): Internal self-test mode; frames are sent internally without transmitting on the bus.
     + **Silent Mode** (bit 31): The controller does not influence the CAN bus (for passive monitoring).
   * Both modes are disabled by default. Uncommenting the corresponding line enables them for testing purposes.
2. **Exit Initialization**:
   * can\_exit\_init() is called to put the CAN peripheral back into normal operating mode, ready for communication.

### Important Concepts

#### Bit Timing Parameters

* **SJW** (Synchronization Jump Width): Determines how much the bit time can be adjusted during synchronization.
* **TSEG1 and TSEG2**: Define the length of the propagation and phase segments of a CAN bit.
* **Sample Point**: The point during a CAN bit where the value is sampled. Adjusting TSEG1 and TSEG2 helps place this point optimally.

#### Test Modes

* **Loopback Mode**: Used to test the CAN controller without actual bus activity.
* **Silent Mode**: Allows passive monitoring of the CAN bus without affecting it.

### Summary

This function:

1. Configures the CAN peripheral for a 500 kbps baud rate.
2. Sets optimal bit timing parameters for reliable communication.
3. Optionally enables testing modes (loopback and silent mode).
4. Exits initialization mode to allow normal CAN communication.

The function can\_setup() initializes the CAN peripheral and its associated GPIO pins on the microcontroller. It prepares the CAN peripheral for communication by configuring clocks, GPIO pins, and other settings. Let's break it down:

**void can\_setup(void)**

**{**

**/\***

**CAN Rx and Tx are connected to PB8 and PB9.**

**Both pins use AF9 (Alternate Function 9) for CAN communication.**

**\*/**

**// Enable clock for CAN and GPIOB**

**RCC\_APB1ENR |= (1 << 25); // Enable CAN peripheral clock (bit 25 of RCC\_APB1ENR)**

**RCC\_AHBENR |= (1 << 18); // Enable GPIOB clock (bit 18 of RCC\_AHBENR)**

**// Configure PB8 (CAN Rx) and PB9 (CAN Tx) as Alternate Function mode**

**GPIOB\_MODER |= (1 << 17) | (1 << 19); // Set PB8 and PB9 to Alternate Function (MODER bits: 10b)**

**// Select Alternate Function 9 (AF9) for PB8 and PB9**

**GPIOB\_AFRH |= (1 << 0) | (1 << 3) | (1 << 4) | (1 << 7);**

**/\* AFRH configuration:**

**- PB8 (AF9) occupies AFRH[0:3].**

**- PB9 (AF9) occupies AFRH[4:7]. \*/**

**// Enter Initialization Mode for configuring the CAN peripheral**

**can\_enter\_init();**

**// Uncomment the following line to enable Hot Self-Test Mode**

**// CAN\_BTR |= (1 << 30) | (1 << 31); // Enable Loopback and Silent modes for testing**

**// Exit Initialization Mode to allow normal operation**

**can\_exit\_init();**

**}**

### Step-by-Step Breakdown

1. **Enable Peripheral Clocks**:
   * **CAN Clock**: The CAN peripheral's clock is enabled by setting bit 25 in the RCC\_APB1ENR register.
   * **GPIOB Clock**: The clock for GPIO port B is enabled by setting bit 18 in the RCC\_AHBENR register.

This step ensures the CAN peripheral and its associated GPIO pins have the required clock signals.

1. **Configure GPIO Pins for CAN**:
   * **PB8 (Rx) and PB9 (Tx)**:
     + Set to **Alternate Function Mode** by modifying the MODER register. The bits corresponding to PB8 and PB9 are set to 10 (Alternate Function mode).
   * **Select Alternate Function 9 (AF9)**:
     + Modify the AFRH (Alternate Function High Register) to select AF9 for PB8 and PB9. Each pin is configured for its alternate function slot in the AFRH register.
2. **Enter Initialization Mode**:
   * The can\_enter\_init() function is called to put the CAN peripheral into initialization mode, where configuration changes can be made safely.
3. **Optional Test Mode (Commented Out)**:
   * **Hot Self-Test Mode**: By setting bits 30 and 31 in the CAN\_BTR register, the CAN peripheral can be configured for Loopback and Silent modes. This allows internal testing without affecting the bus. However, the line is commented out, indicating this mode is optional.
4. **Exit Initialization Mode**:
   * The can\_exit\_init() function is called to transition the CAN peripheral from initialization mode to normal operational mode.

### Register Details

#### RCC Registers:

1. **RCC\_APB1ENR**: Enables peripherals on the APB1 bus.
   * Bit 25: CAN clock enable.
2. **RCC\_AHBENR**: Enables peripherals on the AHB bus.
   * Bit 18: GPIOB clock enable.

#### GPIO Registers:

1. **GPIOB\_MODER**: Configures the mode of GPIO pins.
   * PB8 and PB9 are set to **Alternate Function Mode** (10b).
2. **GPIOB\_AFRH**: Selects the alternate function for GPIO pins PB8 and PB9.
   * AF9 is used for CAN Rx and Tx.

### Key Features of the Code

1. **CAN Peripheral Setup**:
   * Ensures the CAN peripheral is initialized and ready for communication.
2. **GPIO Configuration**:
   * Configures the correct pins and alternate function for CAN communication.
3. **Optional Test Mode**:
   * Includes provisions for enabling Loopback and Silent mode for internal testing (commented out by default).

### Applications

This setup function is typically called during the initialization phase of a program that uses CAN communication. It ensures the CAN peripheral and its pins are properly configured before sending or receiving data.

**BeagleBone Black Enable Internet**

[**Link**](https://www.digikey.in/en/maker/blogs/how-to-connect-a-beaglebone-black-to-the-internet-using-usb)

**Configure the Windows firewall to allow pings**

[**Link**](https://kb.iu.edu/d/aopy)

**Enable Internet Sharing Option Disabled by network Administrator**

[**Link**](https://xvii.com.au/1242/windows/internet-connection-sharing-has-been-disabled-by-the-network-administrator-windows-8/)

**BeagleBone Black Testing Virtual CAN Driver**

modprobe vcan

ip link add dev vcan0 type vcan

ifconfig vcan0 up

ip -details -statistics link show vcan0

candump vcan0 &

cansend vcan0 442#DEADBABE

**BeagleBone Black Enable Real CAN Driver**

nano /boot/uEnv.txt

Edit :

###Overide capes with eeprom

uboot\_overlay\_addr0=/lib/firmware/BB-CAN1-00A0.dtbo

**BeagleBone Black CAN Commands**

modprobe can

modprobe can-raw

modprobe can-dev

ip link set can1 up type can bitrate 125000

candump can1 & // after running this connect stm32f4 disc board with transceiver & send data to BBB

//check weather data received and then proceed further , if not received data then debug connection/code here

cansend <interface\_name> msg\_id#data\_in\_hex\_without\_adding\_0x

cansend can1 123#02AD //this command will send 2 bytes of data ie 0x02 , 0xAD with id 0x123

ifconfig can1 down

ip -details -statistics link show can1

#### BeagleBone Black CAN Errors

A device may enter the “bus-off” state if too many errors occurred on the CAN bus. Then no more messages are received or sent. An automatic bus-off recovery can be enabled by setting the “restart-ms” to a non-zero value, e.g.:

sudo ip link set can1 type can restart-ms 100

Alternatively, the application may realize the “bus-off” condition by monitoring CAN error frames and do a restart when appropriate with the command:

sudo ip link set can1 type can restart

Note that a restart will also create a CAN error frame.

**Python CAN in BBB**

***receiveCAN.py***

#!/usr/bin/python

import can

def main():

#can\_interface = 'vcan0'

can\_interface = 'can1'

bus = can.interface.Bus(can\_interface, bustype='socketcan')

try:

while True:

msg = bus.recv(0.0)

if msg:

msg\_id=msg.arbitration\_id

print("ID",msg\_id)

data=msg.data

data\_format1=msg.data[1]

print("Data Format0",data)

print("Data Format1",data\_format1)

print("Data Format2",msg)

except KeyboardInterrupt:

bus.shutdown()

if \_\_name\_\_ == "\_\_main\_\_":

main()

***sendCAN.py***

import can

bustype = 'socketcan'

#channel = 'vcan0'

channel ='can1'

def main():

bus = can.interface.Bus(channel=channel, bustype=bustype)

for i in range(10):

print ("Send a message...")

#msg = can.Message(arbitration\_id=0x123, data=[i, i+1, 0, 1, 3, 1, 4, 1], is\_extended\_id=False)

#msg1 = can.Message(arbitration\_id=0x123,data=[0x64, 0x65, 0x61, 0x64, 0x62, 0x65, 0x65, 0x66], is\_extended\_id=False)

#msg2 = can.Message(arbitration\_id=0x123,data=b'deadbeef', is\_extended\_id=False)

msg3 = can.Message(arbitration\_id=0x123,data=b'de', is\_extended\_id=False)

bus.send(msg3)

time.sleep(1)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**BeagleBone Black Enable UART2**

nano /boot/uEnv.txt

Edit :

###Overide capes with eeprom

uboot\_overlay\_addr1=/lib/firmware/BB-UART2-00A0.dtbo

|  |  |  |  |
| --- | --- | --- | --- |
| **UART** | **RX** | **TX** | **Device** |
| **UART2** | **P9\_22** | **P9\_21** | **/dev/ttyO2** |

**UART2 Enabled Confirmation**

# cat /sys/kernel/debug/pinctrl/44e10800.pinmux-pinctrl-single/pinmux-pins | grep -i uart

pin 84 (PIN84): 48024000.serial (GPIO UNCLAIMED) function pinmux\_bb\_uart2\_pins group pinmux\_bb\_uart2\_pins

pin 85 (PIN85): 48024000.serial (GPIO UNCLAIMED) function pinmux\_bb\_uart2\_pins group pinmux\_bb\_uart2\_pins

**Install Minicom to use as UART Terminal**

sudo apt-get update

sudo apt-get install minicom

sudo minicom -b 9600 -D /dev/ttyO2

**Python UART Test Script**

import Adafruit\_BBIO.UART as UART

import serial

#UART.setup("UART2")

with serial.Serial(port = "/dev/ttyO2", baudrate=9600) as ser:

print("Serial is open!")

ser.write(b"Hello World123!")

**More about UART**

<https://learn.adafruit.com/setting-up-io-python-library-on-beaglebone-black/uart>

### AWS CAN Script Publish and Subscribe

[Link](https://drive.google.com/file/d/1UjU05OlLQ_tyo7GZyyXDxiGHxUoMhXqY/view?usp=sharing)

**Supported links:**

<https://www.youtube.com/watch?reload=9&v=ymD3F0h-ilE>

<https://www.youtube.com/watch?v=ar3I38lCLT4&list=PLfExI9i0v1sn_lQjCFJHrDSpvZ8F2CpkA&index=34>

**Time Quanta calculation**- refer to this link

STMicroelectronics uses bxCAN, select the APB1 frequency & check the Time Quantas from the table.

<http://www.bittiming.can-wiki.info/>

CAN Primer -<http://www.keil.com/download/files/canprimer_v2.pdf>

Link for CAN Code:

This is correct that HAL CAN driver has been redesigned with new API.

Please find below a short migration guide:

* Fields of CAN\_InitTypeDef structure are renamed: SJW to SyncJumpWidth, BS1 to TimeSeg1, BS2 to TimeSeg2, ABOM to AutoBusOff, AWUM to AutoWakeUp, NART to AutoRetransmission (inversed), RFLM to ReceiveFifoLocked and TXFP to TransmitFifoPriority
* HAL\_CAN\_Init() is split into both HAL\_CAN\_Init() and HAL\_CAN\_Start()
* HAL\_CAN\_Transmit() is replaced by HAL\_CAN\_AddTxMessage() to place Tx request, then HAL\_CAN\_GetTxMailboxesFreeLevel() for polling until completion
* HAL\_CAN\_Transmit\_IT() is replaced by HAL\_CAN\_ActivateNotification() to enable transmission with interrupt mode, then HAL\_CAN\_AddTxMessage() to place Tx request
* HAL\_CAN\_Receive() is replaced by HAL\_CAN\_GetRxFifoFillLevel() for polling until reception, then HAL\_CAN\_GetRxMessage() to get Rx message
* HAL\_CAN\_Receive\_IT() is replaced by HAL\_CAN\_ActivateNotification() to enable reception with interrupt mode, then HAL\_CAN\_GetRxMessage() in the receive callback to get Rx message
* HAL\_CAN\_Sleep() is renamed to HAL\_CAN\_RequestSleep()
* HAL\_CAN\_TxCpltCallback() is split into HAL\_CAN\_TxMailbox0CompleteCallback(), HAL\_CAN\_TxMailbox1CompleteCallback() and HAL\_CAN\_TxMailbox2CompleteCallback()
* HAL\_CAN\_RxCpltCallback() is split into HAL\_CAN\_RxFifo0MsgPendingCallback() and HAL\_CAN\_RxFifo1MsgPendingCallback()

More complete 'how to use the new driver' is detailed in the driver header section itself.

https://community.st.com/s/question/0D50X00009XkgP6/halcaninit-is-failing