

FRONTGRADE

APPLICATION NOTE

UT32M0R500

Arm Cortex M0+ CAN Unit

8/14/2019

Version #: 1.0.0

Table 1: Cross Reference of Applicable Products

Product Name	Manufacturer Part Number	SMD #	Device Type	Internal Pic Number
Arm Cortex M0+	UT32M0R500	5962-17212	CAN Unit	QS30

1.0 Overview

The UT32M0R500 contains two Control Area Network (CAN) controllers. CAN is a serial data communications bus with error detection and data rates up to 1 Mbits per second. Originally developed by Bosch Corporation for automobiles, CAN has expanded to automation and control applications. The CAN protocol is part of the ISO 11989 standard. The CAN system consists of the bus with CANH and CANL wires terminated with 120 Ohm resistors, the UT64CAN333x transceiver (recommended), and the UT32M0R500 CAN controller. Figure 1 shows the CAN communication system.

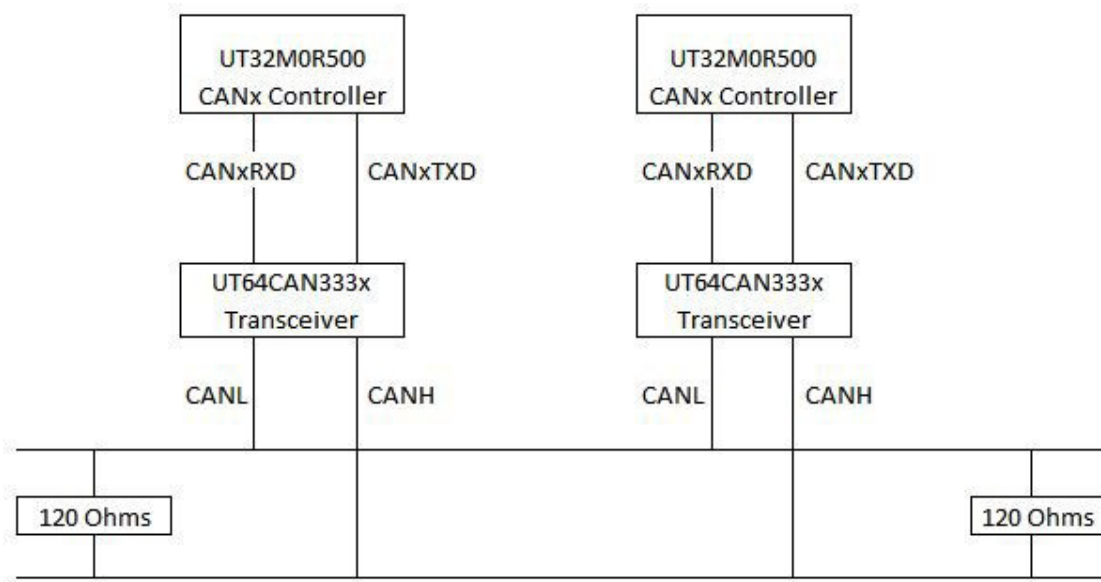


Figure 1: CAN communication system

CAN provides data frame transmission and remote transmission request communication. Data frame transmission is for sending a message; remote transmission request is for requesting a message. Error signaling and retransmission is done internal to the hardware.

CAN is based on broadcasting messages rather than address-based messages. CAN sends each message on the bus with a unique identifier; the unique identifier defines both the contents and priority of the message. The priority of the message is for bus arbitration when several nodes compete for bus access.

For BasiCAN or Standard CAN, the frame can have up to 135 bits, which includes Inter Frame Space (IFS), 24 stuff bits and the 11 bit identifier can have up to 2^{11} possible addressable messages, see Figure 2.

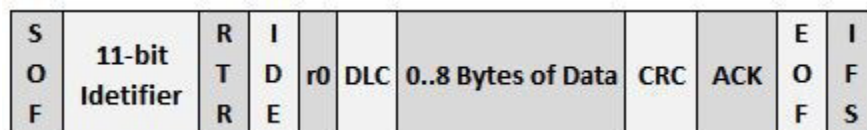


Figure 2: Basic (Standard) CAN Frame

For Extended CAN or PelICAN, the frame can have up to 160 bits, which includes Inter Frame Space (IFS) and 24 stuff bits and the 29 bit identifier can have up to 2^{29} possible addressable messages, see Figure 3.

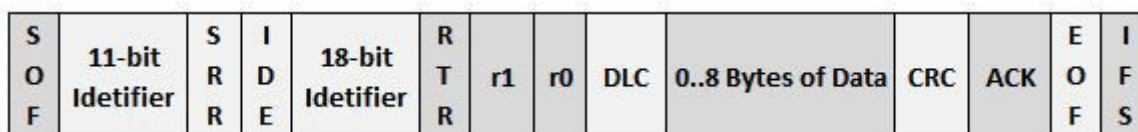


Figure 3: Extended (PelICAN) CAN Frame

2.0 Application Note Layout

This application note (AN) provides a description of the CAN unit's hardware, configuration and programming.

3.0 CAN Bus

The CAN bus contains two wires in differential mode for one logic bit; the wires are CANH and CANL. The state of the transmitter is either dominant or recessive, see Figure 4.

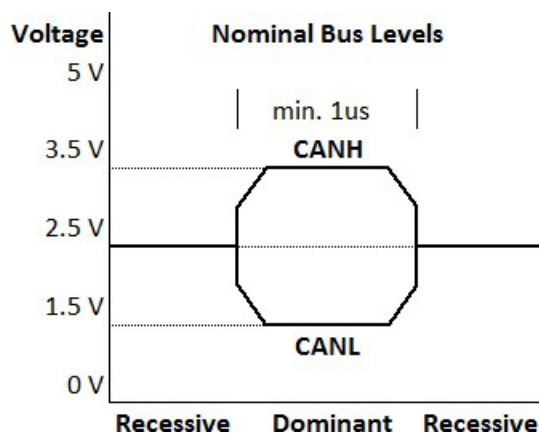


Figure 4: CAN Nominal Bus Levels

When two or more nodes are competing, their output follows a wired-AND mechanism with the **dominant** state overriding the **recessive** one, see Figure 5 for CAN arbitration.

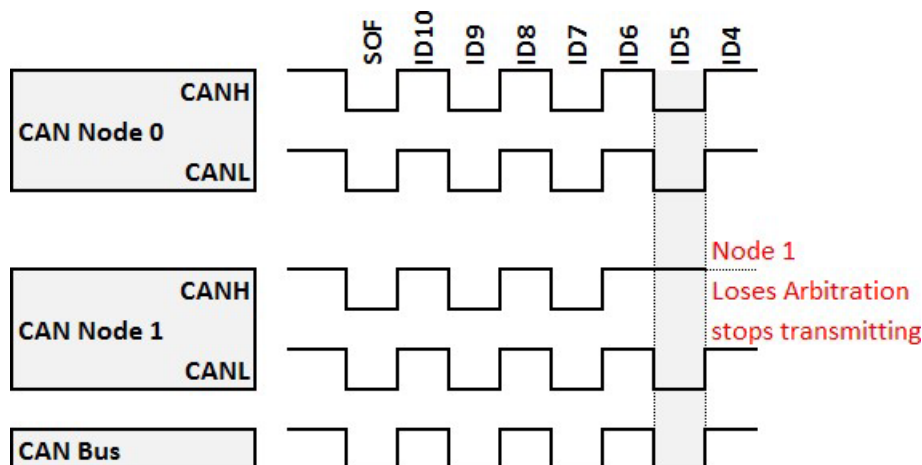


Figure 5: CAN Arbitration

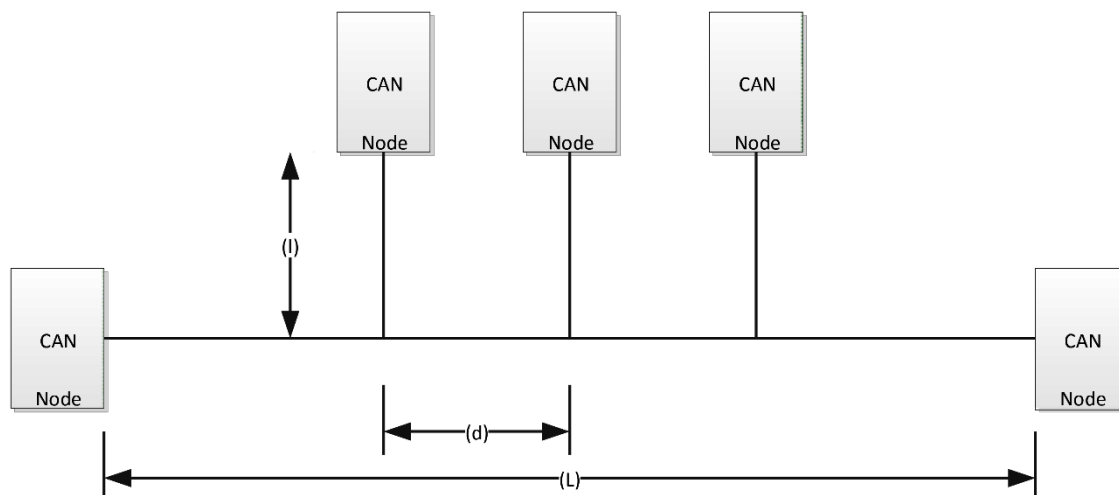
3.1 CAN BUS Connection

The CAN network is a party-line connection that allows many CAN nodes to be connected. The logical number of nodes on the bus is limited to the size of the CAN ID. In CAN 2.0B extended mode, more nodes can be connected. In order for the CAN bus to function properly, termination resistors are used to impede reflections on the bus. To determine the correct value of the termination resistors, check the impedance of the cable and match the resistor to it. For a CAN bus cable with a 120-ohm line impedance, a 120-ohm resistor is used. In high-speed bus communications, termination resistors at both ends of the bus will be required.

Usually, the Master is placed at one end of the bus, but it's possible to have it connected in the middle with termination of nodes at both ends of the bus line. Figure 6 shows the CAN nodes connection paradigm and the relationship between nodes and their respective maximum distance. Table 2 shows the distance between nodes based on end nodes, sub-nodes, and bus speed.

Table 2: CAN Bus Speed and Cable Length Legend:

Bus Speed	Bus Length (L)	Cable Stub Length (l)	Node Distance (d)
1 Mbit/s	40 meters/131 feet	0.3 meters/1 foot	40 meters/131 feet
500Kbits/s	100 meters/328 feet	0.3 meters/1 foot	100 meters/328 feet
100Kbits/s	500 meters/1640 feet	0.3 meters/1 foot	500 meters/1640 feet
50 Kbits/s	1000 meters/3280 feet	0.3 meters/1 foot	1000 meters/3280 feet



Legend:

- L: Maximum Bus Length
- l: Maximum Cable Stub Length
- d: Maximum Node Distance

Figure 6: CAN Nodes Connection Paradigm

4.0 CAN Transceiver

4.1 UT64CAN333x Transceiver (recommended) Connection

The UT32M0R500 CAN controllers transmit and receive data up to 1 Mbps/s; each CAN controller provides a simple connection to the CAN UT64CAN333x transceiver (recommended) via CANx_TXD and CANx_RXD pins as shown in Table 3 and Figure 7 for CAN0 connection.

Table 3: CAN Module TX/RX Signal Pin

UT32M0R500 CAN Controller 0	UT32M0R500 CAN Controller 1	Description
CAN0_TXD	CAN1_TXD	Transmit Signal Pin
CAN0_RXD	CAN1_RXD	Receive Signal Pin

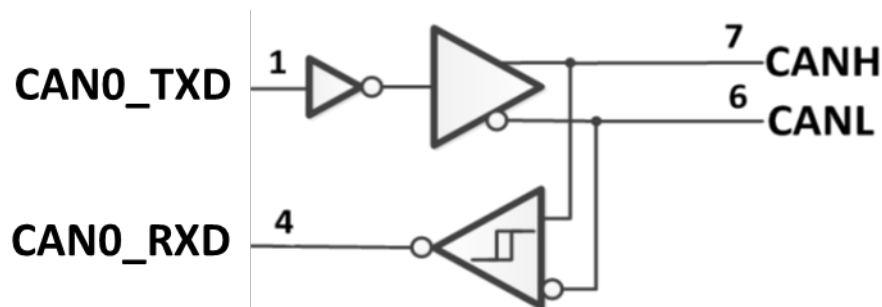


Figure 7: CAN0 Connection

4.2 UT64CAN333x Transceiver (recommended) Interface Connectors

There are several types of connectors used for CAN interface such as the 9-pin male D-sub, 10-pin header, RJ-Style, 7-pin open style, etc. Among these connectors, the 9-pin male D-sub connector is most widely used and Table 4 shows the pinout. The UT64CAN333x eval board (recommended) provided by Frontgrade uses the 9-pin male D-sub connector.

Table 4: 9-Pin Male DSUB Connector Pinout for CAN Bus

Pin #	Signal names	Signal Description
1	Reserved	Upgrade Path
2	CAN_L	Dominant Low
3	CAN_GND	Ground
4	Reserved	Upgrade Path
5	CAN_SHLD	Shield, Optional
6	GND	Ground, Optional
7	CAN_H	Dominant High
8	Reserved	Upgrade Path
9	CAN_V+	Power, Optional

5.0 CAN Controller

In this section, we will explain how the CAN's priority and bitwise arbitration work, how to calculate CAN baud rate and how acceptance filters work.

5.1 CAN Priority and Bitwise Arbitration

CAN protocol does not have a priority field. It uses the message ID and non-destructive bitwise arbitration to determine which message has the highest priority. When the CAN ID bits are cleared, dominant bits, it has the highest priority and when the bits are set, recessive bits, it has the lowest priority.

If all challenging message ID signals on the bus are synchronized and connected in a wired-AND circuit then the low ID's bit signal, the dominant bit, overwrites the high ID's bit signal, the recessive bit. The high priority message ID overwrites the low priority message ID causing all low priority nodes to terminate its transmission. This is how nondestructive bitwise arbitration works with the cooperation of every node on the bus, see [Figure 5](#).

5.2 CAN Baud Rate

The BUS_TIMING_0 and BUS_TIMING_1 are the bus timing control registers, see Table 5-Table 6.

Table 5: Bus Timing Register 0 (BTR0), Offset 6

Bit #	7	6	5	4	3	2	1	0
R	SJW		BRP					
W								
Reset	00		[00...0]					

Table 6: Bus Timing Register 1 (BTR1), Offset 7

Bit #	7	6	5	4	3	2	1	0
R	SAM	TSEG2			TSEG1			
W								
Reset	0	000			[00...0]			

BUS_TIMING_0: SJW[7:6] are the Synchronization Jump Width bits; 0 for high-speed communication; BRP[5:0] are the Baud Rate Prescaler bits.

BUS_TIMING_1: SAMP[7] bit; 1 sample three times, 0 sample once; TSEG2[6:4] Time Segment 2; TSEG1[3:0] Time Segment 1.

The clock period for individual bits is $2 * fsck * (BRP[5:0] + 1)$, where $fsck$ is the system clock frequency.

The input bit is sampled at the time in between Segment 1 and Segment 2, see Figure 8.

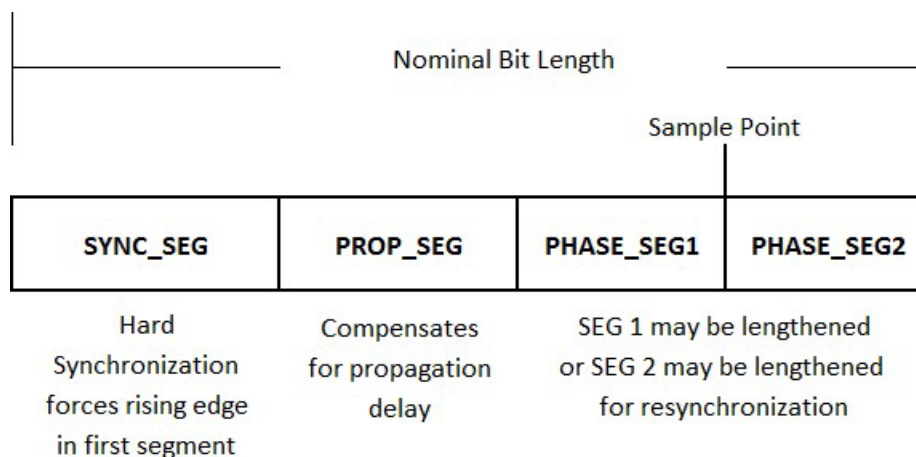


Figure 8: CAN Input Bit Sample Point

There are three time segments: Segment 0, Segment 1 and Segment 2.

Segment 0 is one clock period: $2 * fsck * (BRP[5:0] + 1)$.

Segment 2 is TSEG2[6:4]; the length = $fsck * (TSEG2[6:4] + 1)$.

Segment 1 is TSEG1[3:0]; the length = $fsck * (TSEG1[3:0] + 2)$.

Then, to find the time for each bit, include all the segments:

$CAN_TS2_3tq = 2, /*!< 3 \text{ time quanta} */$

$CAN_TS2_4tq = 3, /*!< 4 \text{ time quanta} */$

Bit Time = $2 * fsck * (BRP[5:0] + 1) * (TSEG2[6:4] + 1 + TSEG1[3:0] + 2)$

= $2 * (20ns) * (24+1) * (2 + 1 + 3 + 2)$

= 8usec

Or Bit Time = $1/8usec = 125Khz$

5.3 CAN Acceptance Filters

The UT32M0R500 Can controller has the ability to filter incoming messages. It has Acceptance Code and Acceptance Mask registers. The acceptance filter mode (AFM) bit chooses between 1: single acceptance filter or 0: dual acceptance filter. The acceptance and mask registers are compared bit-by-bit with the incoming message and if the message and registers match, then the microcontroller will read the message; otherwise, the message is discarded.

Masked bits set to 0 will be filtered and bits set to 1 will be accepted regardless of the ID bit value. For extended CAN filtering information and CAN filtering examples, refer to App-Note-UT32M0R500-CAN-Filtering at

www.frontgrade.com/HiRel.

6.0 CAN Controller Programming

The following sections show programming examples by making use of Frontgrade APIs for the UT32RM0R500.

6.1 BasiCAN Initialization

Code 1 initializes the CAN0 controller for BasiCAN and for specifics on the APIs, refer to www.frontgrade.com/HiRel.

```
// init the CAN
CAN_StructInit (&CAN_InitStruct);

// CAN BAUD 1 MHz
// 50 MHz / 10 prescaler = 5 MHz
// (Tsync = 1) + 2 + 2 = 5 time quanta
// 5 MHz / 5 quanta = 1 MHz

CAN_InitStruct.CAN_Prescaler = 4;    // (4 + 1) * 2 = 10
CAN_InitStruct.CAN_TS2 = CAN_TS2_2tq; // 1 + 1 = 2
CAN_InitStruct.CAN_TS1 = CAN_TS1_2tq; // 1 + 1 = 2

CAN_Init (CAN0, &CAN_InitStruct);

CAN_FilterInitStruct.CAN_FilterMode = CAN_FILTER_MODE_SINGLE;
CAN_FilterInitStruct.CAN_FilterID_Value1 = 0x0555;
CAN_FilterInitStruct.CAN_FilterID_Value1_Mask = 0;

CAN_FilterInit (CAN0, &CAN_FilterInitStruct);
```

Code 1: BasiCAN0 Initialization

6.2 Extended CAN Initialization

Code 2 initializes the CAN0 controller for Extended CAN and for specifics on the APIs, refer to www.frontgrade.com/HiRel.

```
// init the CAN
CAN_StructInit (&CAN_InitStruct);

// CAN BAUD 1 MHz
// 50 MHz / 10 prescaler = 5 MHz
// (Tsync = 1) + 2 + 2 = 5 time quanta
// 5 MHz / 5 quanta = 1 MHz

CAN_InitStruct.CAN_Prescaler = 4;    // (4 + 1) * 2 = 10
CAN_InitStruct.CAN_TS2 = CAN_TS2_2tq; // 1 + 1 = 2
CAN_InitStruct.CAN_TS1 = CAN_TS1_2tq; // 1 + 1 = 2

CAN_InitStruct.CAN_Mode = GEN_CAN_ACCESS_MODE_PELICAN;

CAN_Init (CAN0, &CAN_InitStruct);

CAN_FilterInitStruct.CAN_FilterMode = CAN_FILTER_MODE_DUAL;
CAN_FilterInitStruct.CAN_FilterID_Value1 = 0x0555;
CAN_FilterInitStruct.CAN_FilterID_Value1_Mask = 0;

CAN_FilterInit (CAN0, &CAN_FilterInitStruct);
```

Code 2: Extended CAN0 Initialization

For **Keil MDK**, under **Preprocessor Symbols**, define the preprocessor directive `CAN_ENABLE_PELICAN_SUPPORT`, see Figure 9.

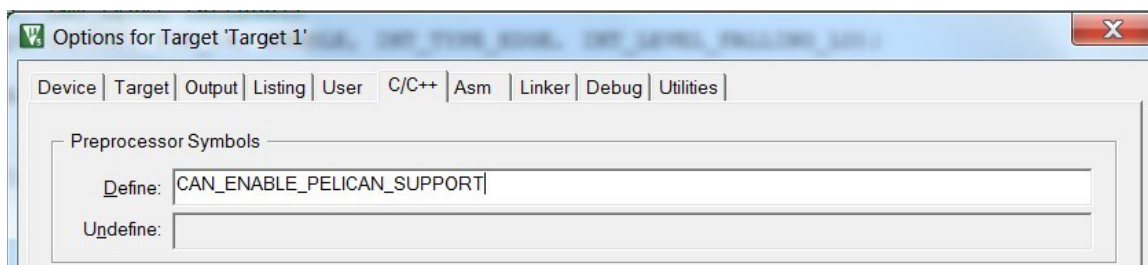


Figure 9: Extended CAN Support Option

6.2.1 CAN0 Interrupt Initialization

CAN0 interrupt (IRQ) is mapped to number 16 in the Interrupt Vector Table. The address of interrupt 16 in the Interrupt Vector Table is mapped to the CAN0_IRQHandler, which is the interrupt service routine (ISR). In the ISR, software must check for which interrupt happened, see Code 3.

```
CAN_IntConfig (CAN0, CAN_INT_RECEIVE, ENABLE); // Enable CAN RX interrupt

//Enables a device specific interrupt in the NVIC interrupt controller.
NVIC_EnableIRQ (CAN0_IRQn);
```

Code 3: CAN0 Receive Interrupt settings

6.3 Standard CAN Send Message (Data Frame Transmission)

Code 4 shows how to transmit a message. It begins by setting each of the fields in the message frame. The FrameFormat sets the ID extended bit (IDE) to 0 for standard format; the 11-bit identifier (ID) set 0x111; remote transmit request (RTR) bit set to 0 for data frame; data length code (DLC) bits set to 8 for eight bytes of data; eight bytes are copied into the data field of the message; finally, by setting the TR bit to 1 in the command register, the message will start being transmitted out on CANxTD line.

```
// create the TX message
TxMessage.FrameFormat = CAN_FRAME_FORMAT_STANDARD;
TxMessage.ID = 0x111;
TxMessage.RTR = 0; // Remote transmit request: 0 data frame, 1 initiate request
TxMessage.DLC = 8; // Data length code
for (i = 0; i < 8; i++)
{
    TxMessage.Data[i] = i;
}
// for this example code, Timer 0 is used to send a message about every 1.3ms
if(TimerTickExpired){
    TimerTickExpired=0;
    // Send a CAN message
    ErrorCode = CAN_Transmit (CAN0, &TxMessage);
}
```

Code 4: Standard Can Data Frame Transmission

Figure 10 shows the timing diagram for transmitting a CAN data frame message and the node acknowledging it.

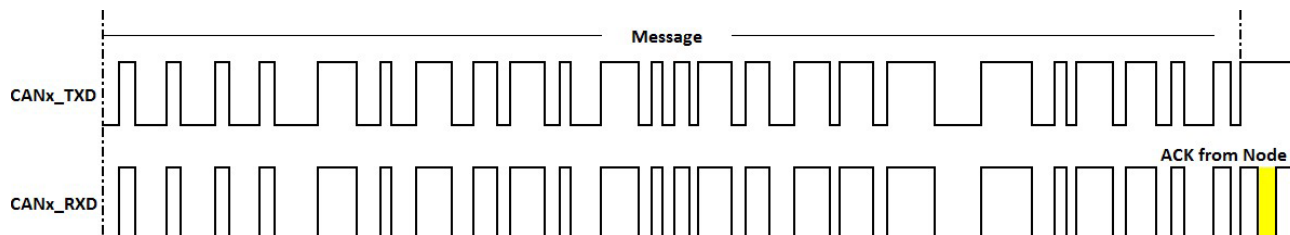


Figure 10: BasiCAN TX Message

6.4 PeliCAN Send Message (Data Frame Transmission)

Code 5 shows how to transmit a message. It begins by setting each of the fields in the message frame. The FrameFormat sets the ID extended bit (IDE) to 1 for extended format; the 11-bit identifier (ID) set 0x111; remote transmit request (RTR) bit set to 0 for data frame; data length code (DLC) bits set to 8 for eight bytes of data; eight bytes are copied into the data field of the message; finally, by setting the TR bit to 1 in the command register, the message will start being transmitted out on CANxTD line.

```
// create the TX message
TxMessage.FrameFormat = CAN_FRAME_FORMAT_EXTENDED;
TxMessage.ID = 0x111;
TxMessage.RTR = 0; // Remote transmit request: 0 data frame, 1 initiate request
TxMessage.DLC = 8; // Data length code
for (i = 0; i < 8; i++)
{
    TxMessage.Data[i] = i;
}
// for this example code, Timer 0 is used to send a message about every 1.3ms
if(TimerTickExpired){
    TimerTickExpired=0;
    // Send a CAN message
    ErrorCode = CAN_Transmit (CAN0, &TxMessage);
}
```

Code 5: PeliCAN Data Frame Transmission

Figure 11 shows the timing diagram for transmitting a CAN data frame message and the node acknowledging it.

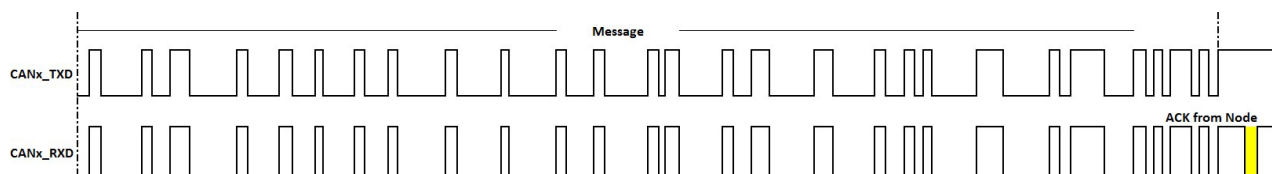


Figure 11: PeliCAN TX Message

6.4.1 CAN Receiving a Message

Code 6 shows how to receive a message. When the message has been received, the CAN_INT_RECEIVE flag is set and an interrupt is requested. Inside the interrupt handler (ISR), the interrupt flag is cleared. The API provides a function for copying 13 bytes from the CAN receive buffer into the RxMessage structure. The message includes ID, data length and data from the specific slave device, see Figure 12-Figure 13. Once the information has been copied, the receive buffer is released by writing 1 to RRB bit in the command register.

```
if(ReceiveComplete){
    ReceiveComplete=0;

    // Read CAN byte(s)
    ErrorCode = CAN_Receive (CAN0, &RxMessage);
}
```

Code 6: CAN read byte(s)

Figure 12 shows the timing diagram for receiving a BasiCAN message.

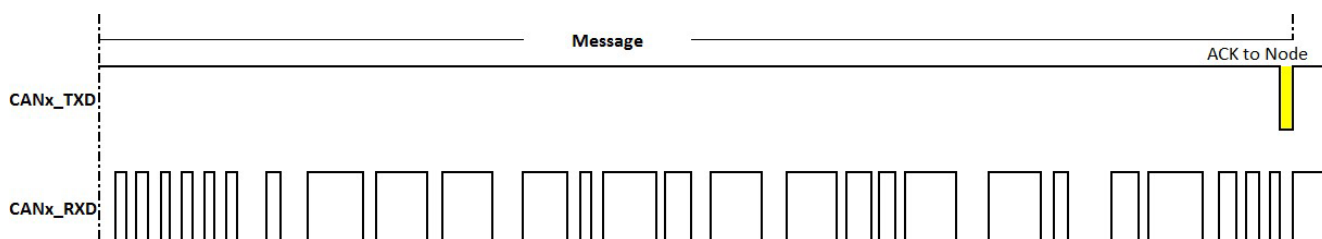


Figure 12: BasiCAN receiving message from a Node

Figure 13 shows the timing diagram for receiving a PeliCAN message.



Figure 13: PeliCAN receiving message from a Node

Putting it all together, Code 7 shows the main subroutine for sending and receiving CAN0 frame message inside an endless loop, and it shows The CAN0_IRQHandler, which is the interrupt service routine for handling the particular CAN interrupt.

```
int main (void){
    // Initialization and settings from previous sections go here.
    for (;;) {
        if (TimerTickExpired) {
            TimerTickExpired = 0;
            ErrorCode = CAN_Transmit (CAN0, &TxMessage);
        }
        if (ReceiveComplete) {
            ReceiveComplete = 0;
            ErrorCode = CAN_Receive (CAN0, &RxMessage);
        }
    }
}

void CAN0_IRQHandler (void) {
    // Clear the flag
    NVIC_ClearPendingIRQ (CAN0_IRQn);
    ReceiveComplete = TRUE;
}
```

Code 7: BasiCAN Example program

7.0 Summary and Conclusion

With excellent internal hardware for error signaling and retransmission, CAN is a high-integrity serial data communication bus.

For more information about our UT32M0R500 microcontroller and other products, please visit our website:
www.frontgrade.com/HiRel.

Revision History

Date	Revision #	Author	Change Description	Page #
8/14/2019	1.0.0	JA	Initial Release	

Datasheet Definitions

	Definition
Advanced Datasheet	Frontgrade reserves the right to make changes to any products and services described herein at any time without notice. The product is still in the development stage and the datasheet is subject to change . Specifications can be TBD and the part package and pinout are not final .
Preliminary Datasheet	Frontgrade reserves the right to make changes to any products and services described herein at any time without notice. The product is in the characterization stage and prototypes are available.
Datasheet	Product is in production and any changes to the product and services described herein will follow a formal customer notification process for form, fit or function changes.

Frontgrade Technologies Proprietary Information Frontgrade Technologies (Frontgrade or Company) reserves the right to make changes to any products and services described herein at any time without notice. Consult a Frontgrade sales representative to verify that the information contained herein is current before using the product described herein. Frontgrade does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by the Company; nor does the purchase, lease, or use of a product or service convey a license to any patents, rights, copyrights, trademark rights, or any other intellectual property rights of the Company or any third party.