Application Note, V 1.1, January 2009

AP16092

XC16x

CAN Bootstrap Loader

Microcontrollers

Never stop thinking

XC16x

Revision History: 2009-01 V 1.1

Previous Version:
V1.0, 2006-03

Page	Subjects (major changes since last revision)
5	update for XC164CM

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents Page

1 Overview

In the Infineon XC166 Family, a built-in CAN bootstrap loader (CAN BSL) is implemented (see **Table 1**). With the CAN bootstrap loader it is possible to load code/data into the internal PSRAM of the XC16x via the TwinCAN interface.

This application note consists of two parts. It begins with detailed information about the CAN BSL implementation (as extension to the User's Manual). Then an example is attached and shows a download process of a simple user code from an external host to the XC16x with the CAN BSL.

Table 1 XC16x Derivative with TwinCAN Module

Derivative	Package
XC161CJ/CS	P-TQFP-144-19
XC164CS	P-TQFP-100-16
XC164D	P-TQFP-100-16
XC167CI	P-TQFP-144-19
XC164CM	PG-TQFP-64-8

2 CAN Bootstrap Loader

2.1 Introduction

The built-in CAN bootstrap loader of the XC166 Family provides a mechanism to load program code/data via the TwinCAN module into the PSRAM, and start executing the loaded code from address E0'0000$_H$ (address of the first transmitted byte).

The bootstrap loader is an integrated mechanism that can be selected via a port configuration during a system start after a HW reset.

The bootstrap loader will configure the TwinCAN module to the baudrate of the host. Once communication has been established, the bootstrap loader receives a host defined variable number of messages for downloading the code/data. The received code/data is sequentially written to the PSRAM.

After the download has been completed, the BSL begins executing the program code that has been loaded and the bootstrap loader is terminated.

During the data transmission phase between the XC16x and an external host, each data frame always transmits eight code/data bytes. The complete load sequence is based on the following three CAN standard frames:

- Initialization frame - sent by the external host to the XC16x.
- Acknowledge frame - sent by the XC16x to the external host.

- Data frame(s) - sent by the external host to the XC16x.

The initialization frame is used in the XC16x for baudrate detection. After a successful baudrate detection is reported to the external host by the acknowledge frame, data is transmitted by data frames. **Table 2** shows the parameters and settings of the three CAN standard frames used for CAN BSL.

Table 2 Structure of CAN Bootstrap Loader Frames

Frame Type	Parameter	Description
Initialization Frame	Identifier	$0x555_H$ 11-bit standard data frame
	DLC = 8 or 6	length code, 8 or 6 bytes transmitted within CAN frame
	Data byte 0	bit timing register BTR low byte for XC16x
	Data byte 1	bit timing register BTR high byte for XC16x
	Data byte 2	acknowledge message identifier ACK_ID, low byte
	Data byte 3	acknowledge message identifier ACK_ID, high byte
	Data byte 4	data message count MSG_CNT, low byte (max. 255)
	Data byte 5	data message count MSG_CNT, high byte = 00_H
	Data byte 6	don't care
	Data byte 7	don't care
Acknowledge Frame	Identifier	ACK_ID as received by data bytes [3:2] of the initialization frame (bits [11:0])
	DLC = 4	data length code, 4 bytes transmitted within CAN frame
	Data bytes 0 to 3	Identical to initialization frame
Data Frame	Identifier	user-defined, 11-bit standard data frame
	DLC = 8	data length code, 8 bytes transmitted within CAN frame
	Data bytes 0 to 7	data bytes, assigned to increasing destination (PSRAM) addresses

Note: In XC164CM device (64 pins package) the CAN bit-timing register will not be reconfigured with byte 0-1 of the initialization frame

2.2 Entering the CAN Bootstrap Loader

The XC164CM enters CAN BSL mode triggered by external configuration during a hardware reset:

- When pins $\overline{\text{TRST}}$, P9.5 and PH1.4 are sampled high and pins P9.4 low at the end of a hardware reset.

The other derivatives enter CAN BSL mode triggered by external configuration during a hardware reset:

- When pins P0.4 and P0.3 are sampled low and pins P0.5 and P0.2 high at the end of an external reset ($\overline{\text{EA}}$=0).
- When pin $\overline{\text{RD}}$ is sampled low and pin ALE is sampled high at the end of an internal reset ($\overline{\text{EA}}$=1).

In this case the built-in bootstrap loader is activated independent of the selected bus mode.

2.3 Initialization Phase

As in the ASC boot mode, the CAN BSL as the first task has to determine the CAN baudrate at which the external host is communicating. This task requires the external host to send initialization frames (see **Table 2**) continuously to the XC16x.

The identifier of the initialization frame has the baudrate detection pattern with value 555_H. Data bytes 0 and 1 contain the information for the XC16x Bit Timing Register. This provides the host the ability to change these protocol parameters to suit the individual user application. The user must ensure that the correct value is sent, otherwise, the communication could be halted. The XC16x will re-initialize its TwinCAN module to these parameters and transmit the acknowledge frame with the identifier (data bytes 2 and 3) sent from the host. Data bytes 4 and 5 tell the XC16x the number of messages to receive. Since bytes 4 and 5 are defined by the host, the host has the ability to decide how large a program to load.

Note: The bootstrap loader assumes all message data is valid and stores the received data in PSRAM sequentially. The host should send its code/data sequentially in multiples of 8 code/data bytes.

The CAN BSL starts measuring signal pulses at the TwinCAN receiver input pin RxDCA (capturing the time between any edge). After pulse measurements the CAN BSL will analyze the data to determine the smallest pulses. This time period is assumed to be one CAN bit time and is used as the measured value for the initial setting for detecting the correct baud rate.

Once the bit time has been determined, a software algorithm starts to determine the host baudrate. The baud rate is determined by performing an iterative loop using the parameters in **Table 5**. A detailed description for the calculation of the CAN bit time and parameter search operations is provided in **Chapter 2.7**. If an exact match between the calculated timer value and the measured timer value is found, then the loop will exit with the current parameters. However, if an exact match is not found, then a search is made to select the closest calculated value to the measured value.

The process of the message detection is subsequently performed. To recognize the initialization frames the XC16x initializes the TwinCAN module into the analyzer mode. In this operation mode, CAN frames are monitored by XC16x without an active participation in any CAN transfer. Once the XC16x can detect the initialization frame without errors, the XC16x will enable the TwinCAN module in its normal operation mode. The TwinCAN module will now acknowledge this frame by generating a dominant bit in its ACK slot. This signals the external host that communication has been established with the XC16x. If the message cannot be detected within all possible calculated timer value, the process will be restarted by recapturing dominant bits and repeating these steps until a message can be found.

2.4 Acknowledge Phase

In the acknowledge phase, the bootstrap loader will reconfigure the BTR register with the value in the initialization frame (bytes 0 and 1) and initialize the TwinCAN module. Afterwards, the bootstrap loader transmits an acknowledge frame back to the external host indicating that it is now ready to receive data frames with the baudrate defined by the external host. The acknowledge frame uses the message identifier ACK_ID that has been received with the initialization frame. The data bytes of the acknowledge frame are copies of the data bytes of the recognized initialization frame.

The CAN BSL initializes the TwinCAN in the following way:

- CAN node A of the TwinCAN module is used.
- Message object 0 is configured as receive object to receive the initialization frame and data frame.
- Message object 1 is configured as transmit object to transmit the acknowledge frame.
- XC164CM uses CAN node A input/output pins: P9.2=RxDCA and P9.3=TxDCA; the other derivatives use CAN node A input/output pins: P4.5=RxDCA, P4.6=TxDCA.

Table 3 lists all registers used in the CAN BSL (when the XC16x has entered BSL mode, this configuration is automatically set):

Table 3 The CAN Bootstrap loader used Registers

Register/Bitfield	Reset Value	Comments
Watchdog Timer	Disabled	
ADDRSEL7	2000h	
FCONCS7	0027h	
TCONCS7	0000h	
GPT12E_T6CON	0800h	
GPT12E_T6	xxxxh	
ABTRL	xxxxh	
DPP0	0380h	

Table 3 The CAN Bootstrap loader used Registers

Register/Bitfield	Reset Value	Comments
P4.6	‘1’	XC164CM: P9.3=‘1’
DP4.6	‘1’	XC164CM: DP9.3=‘1’
ALTSEL0P4.6	‘1’	XC164CM: ALTSEL0P9.3=‘1’
ACR.INT	‘0’	
All registers for message object 0	xxxxh	Defined as transmit message object
All registers for message object 1	xxxxh	Defined as receive message object

Other than after a normal reset the watchdog timer is disabled here, so the bootstrap loading sequence is not time limited.

2.5 Data Transmission Phase

In the data transmission phase, data frames are sent by the external host and received by the XC16x. The data frame uses the 11-bit identifiers. Eight data bytes are transmitted with each data frame. The first data byte is stored in PSRAM at E0’0000$_H$. Consecutive data bytes are stored at incrementing addresses.

Both communication partners evaluate the data message count MSG_CNT until the requested number of CAN data frames has been transmitted. After the reception of the last CAN data frame, the bootstrap loader sequence is finished and executes a jump to address E0’0000$_H$.

2.6 Exiting CAN Bootstrap Mode

After the bootstrap loader has been activated, the watchdog timer and the debug system are disabled. The debug system is released automatically when the BSL terminates after having received a host defined variable number of messages from the host.

The CAN BSL is also aborted after a hardware reset with the non-BSL port configuration

Note: Most probably the initially loaded routine will load additional code/data, some application is likely to require a second receive loop. This second loop may directly use the pre-initialized TwinCAN module to receive data and store it to user-defined locations. Since the watchdog timer is disabled only in the BSL, in this case it is recommended that the first instruction downloaded into the PSRAM is a DISWDT instruction to prevent a undesired watchdog timer reset.

2.7 Choosing the Baudrate for the Bootstrap Loader

According to ISO standard, a CAN bit time is subdivided into the three non-overlapping segments SYNC, TSEG1 and TSEG2, with the corresponding time durations t_{SYNC}, t_{TSEG1} and t_{TSEG2}. Each of these segments consists of multiples of a time quantum t_q (see **Figure 1**). A minimum of $3 \times t_q$ for t_{TSEG1}, $2 \times t_q$ for t_{TSEG2} and $8 \times t_q$ for one CAN bit time calculated as the sum of t_{SYNC}, t_{TSEG1} and t_{TSEG2} is requested by the ISO standard.

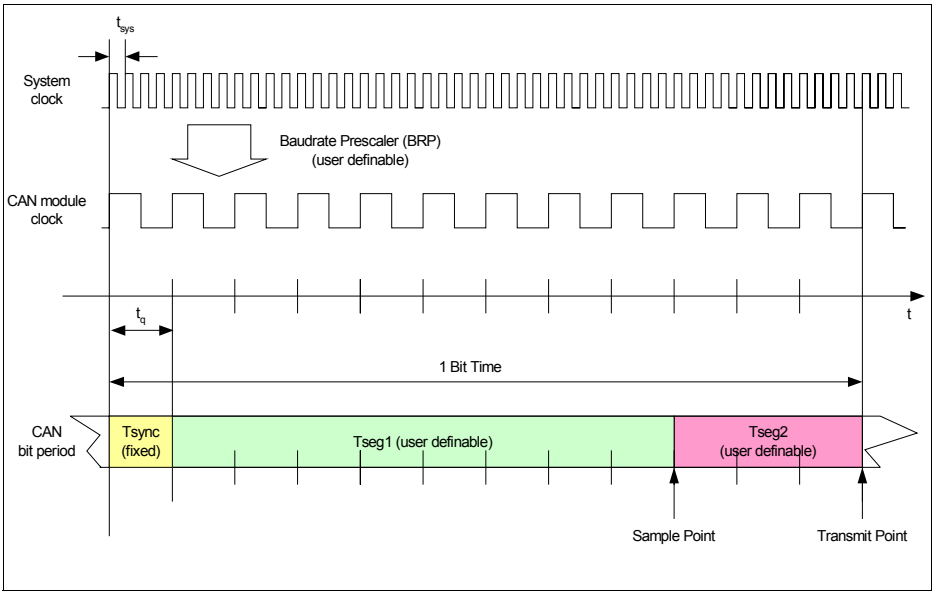

Figure 1 Principle of deriving the bit period as implemented in XC16x

A feature of the CAN protocol is that the bit rate, the sample point, and number of samples taken in a bit period (re-synchronization jump width) are user programmable. The control register BTR is used for setting up these bit timing parameters. Obviously, only integer values can be used for the programming of these parameters. **Table 4** lists all bit definition of register BTR in the TwinCAN module.

Table 4 Bit Timing Register

Field	Description	Bits	Range
BPR	baudrate prescaler	0...6	0...63
SJW	re-synchronization jump width	7,8	0...3

Field	Description	Bits	Range
TSEG1	time segment before sample point	8...11	0...15
TSEG2	time segment after sample point	12...14	0...7
DIV8X	division of module clock by 8	15	0 (the baudrate prescaler is directly driven by f_{CAN} in BSL)

From the ISO standard and the TwinCAN specification, the following assumptions about the bit timing parameters can be made:

$$8 \le N_{tq} \le 25$$

$$3 \le N_{Tseg1} \le 16$$

$$2 \le N_{Tseg2} \le 8$$

$$1 \le N_{Tsjw} \le 4,\ N_{Tsjw} \le N_{Tseg2}$$

Like the bit rate, the sample point is an important parameter. For example, choosing a later sample point in the bit period results in more tolerance with respect to propagation delay and therefore greater bus length. Conversely, choosing a sample point closer to the midpoint of the bit period will allow a greater oscillator tolerance for each node in the system. Obviously, a large allowable oscillator tolerance and a long bus length are conflicting goals, which can only be accomplished through optimization of the bit timing parameters. A good general rule is to set the sample point to about 80% of the bit timing.

From requirements described above **Table 5** summarizes all possible bit timing parameters.

Table 5 Time Quanta Parameters for 80% sample point

Item	N_{tq}	N_{Tseg1}	N_{Tseg2}	Sample Point
1	20 (19 ... 16)	15 (14 ... 11)	4	80.0% (...)
2	15 (14 ... 11)	11 (10 ... 7)	3	80.0% (...)
3	10 (11 ... 8)	7 (6, 5)	2	80.0% (...)

The XC16x uses timer GPT12E_T6 to measure the dominate bit on the CAN bus (polling the CAN receive pin to determine the amount of time that is required for the dominate bit to be transmitted). The dependency between the system clock and one CAN bit time is calculated with the following equations:

$$T6 = t_{CANbit} \bullet \frac{f_{sys}}{2}$$

$$BRP = \left(\frac{2 \bullet T6}{N_{tq}}\right) - 1$$

$$CalTimerValue = (BPR + 1) \bullet N_{tq}$$

$$N_{tq} = Sync + Tseg1 + Tseg2$$

Where:

BRP	Baudrate prescaler value
T6	Timer T6 value
N_{tq}	Number of time quanta per bit period
t_{CANbit}	One CAN bit time
N_{Tseg1}	Number of time quanta before the sample point
N_{Tseg2}	Number of time quanta after the sample point
Sync	Synchronization segment (always equal to 1)
f_{SYS}	CPU operating frequency

When choosing a baudrate, the host must determine what CAN baudrate is possible for the XC16x to detect. The major consideration in this determination is the operating frequency of the XC16x. In general, it is recommended to select the slowest possible baudrate for the initialization frame with a sample point (SP) of about 80%. The value for SJW is recommended to be three. Once communication has been established, the baudrate can be changed to a higher rate.

The baudrate used for the CAN BSL must fulfill the following prerequisites:

- The baudrate is within the specified operation range for TwinCAN module.
- The external host is able to use this baudrate and the XC16x is capable to detect it.
- The computed deviation error is below the GPT12E_T6 quantization error limit.

Table 6 lists the system clock frequency for some typical baudrates from practical measurements.

Table 6 System Clock Frequencies for CAN Baudrate Ranges

Baudrate	fsys
1000 kbit/s	fsys ⩾ 32 MHz
500 kbit/s	fsys ⩾ 16 MHz
250 kbit/s	fsys ⩾ 8 MHz
100 kbit/s	fsys ⩾ 4 MHz

The TwinCAN module can use the clock configuration $f_{CAN}=2\times f_{sys}$ by setting bit CPSYS in register SYSCON1. But in the CAN BSL the initial clock generation mode is defined by the reset value of register SYSCON1 (CPSYS=0) and PLLCON.

Note: The XC164CM has a 1:1 direct drive clock mode in the bootstrap loader; the other derivatives have 2:1 prescaler clock mode in Single Chip Mode ($\overline{EA}$=1) or have the clock configuration depends on the values latched from PORT0 in Bus Mode ($\overline{EA}$=0).

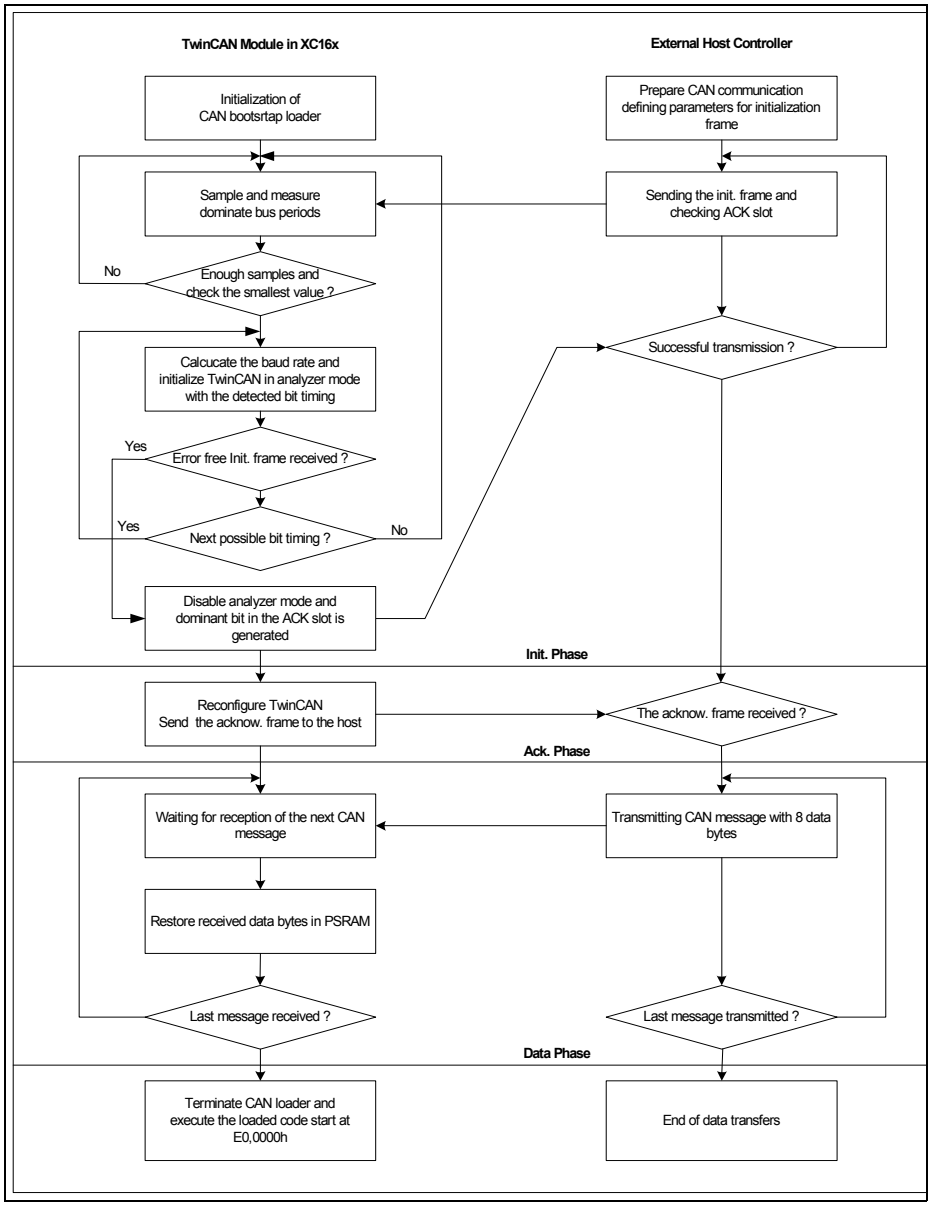

Figure 2 Flow Diagram of the CAN Bootstrap Loader

3 Downloading User Code with CAN Bootstrap Loader

This chapter shows how to load user software from an external host to the XC16x and run it. For a simple realization two XC16x boards are used here. The external host is marked as ‘host board’.

3.1 Hardware used in this Application Note

- Two Infineon XC16x boards (equipped with external RAM and used as ‘host board’, e.g. ‘XC16x board’). As a ‘slave board’ with the active CAN BSL, ‘XC16x board’, ‘XC164CS series Easy kit board’ and ‘XC164CM series Easy kit board’ can be used. For detailed information about the XC16x board, please refer to **www.infineon.com**.
- A serial cable to connect the PC and ‘host board’.
- Terminal program, e.g. ‘MiniMon’ tool, for downloading hex files into the internal flash or the external memory, and starting the code execution.
- CAN bus cable, ‘CAN Analyzer’ tool (optional).

3.2 General Description of the System

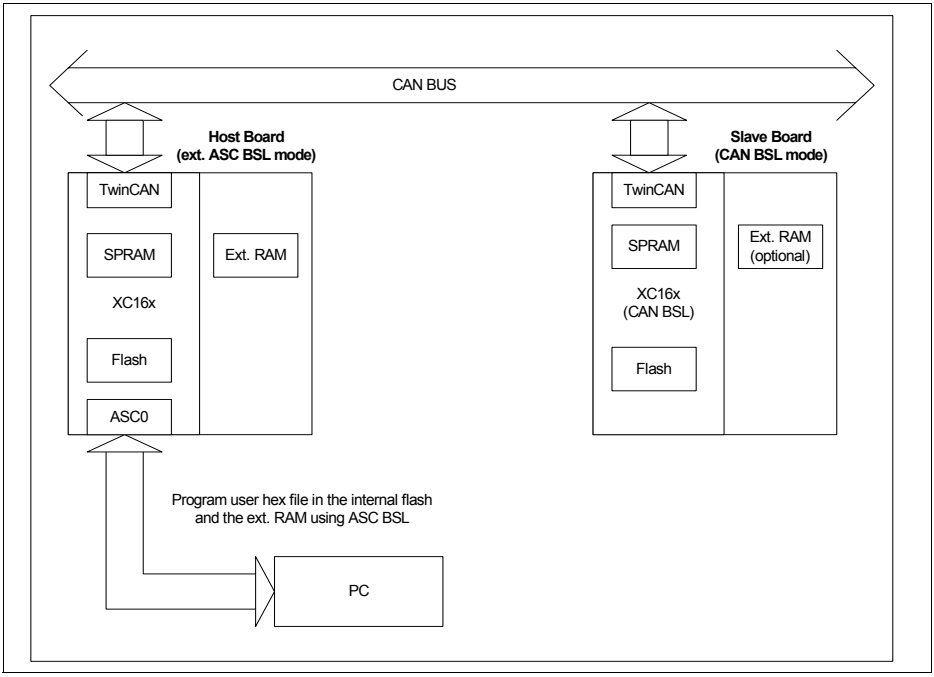

Figure 3 System Overview

Simple user code for pin toggle (p9_4_toggle.hex or p1L_0_toggle.hex) with address starting C0'0000$_H$ is included in this application note. It should be downloaded into the on-chip flash on the host board first. (For the flash programming the freeware tool 'MiniMon' can be used and downloaded from **www.infineon.com**.

The application program is a software routine for the host board. It contains the CAN module initialization code (including TwinCAN transmit pins and configuring of the initialization frame), data transmit process and a signal for starting and terminating the download sequence. It is located in external memory. Its role is to transfer the user code from the internal flash on the host board into the slave board via CAN bus.

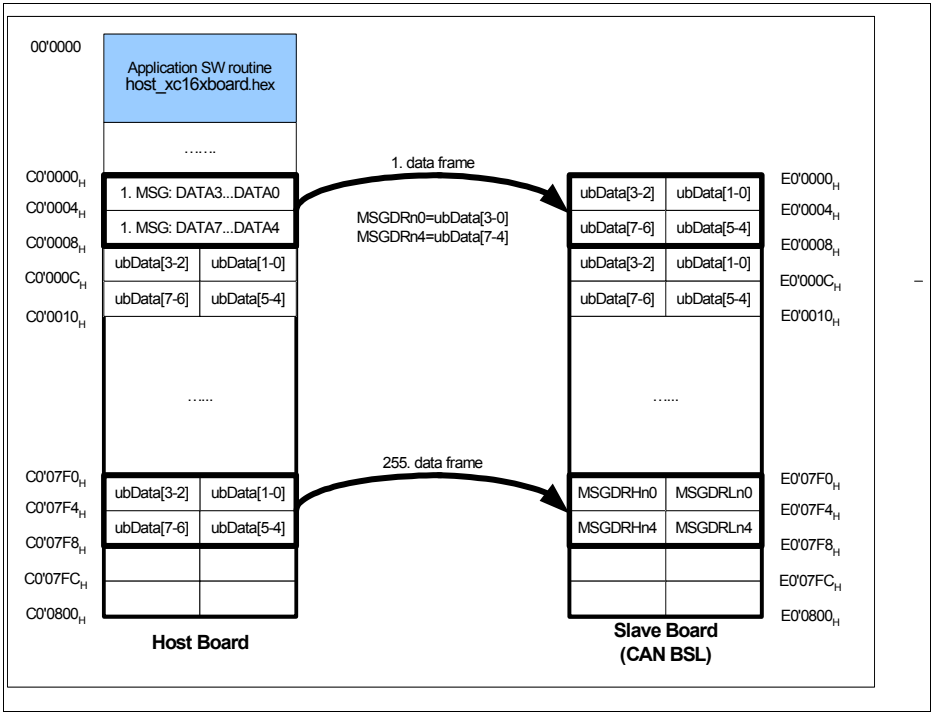

Figure 4 Memory Map

The source file (host_XC16xboard.c) includes many valid values for the bit timing register with a wide CAN baudrate ranges (50K...1000K baudrate) to be set in the initialization frame. You can simply modify this file to fit your needs. The compiled hex file has a fixed bit timing for 500K baudrate. It can be used in the host board with the external ASC BSL in prescaler mode (f_{OSC}=16 MHz) directly, when the slave board is in the CAN BSL with f_{SYS}=16 MHz.

3.3 Testing this Application Note

All hardware below should be configured first:

- Set the slave board in the active CAN BSL mode.
- Set the host board in the active ASC BSL with $\overline{EA}$=0 (use the external RAM on the host board).
- Connect the CAN nodes (node A of the TwinCAN module) between two boards.
- Connect the ASC0 interface on the host board to your PC and start the terminal program ‘MiniMon’.

Note: The DIP switch S401#1 on the two boards must be in ON position to active CAn node 0 transceiver.

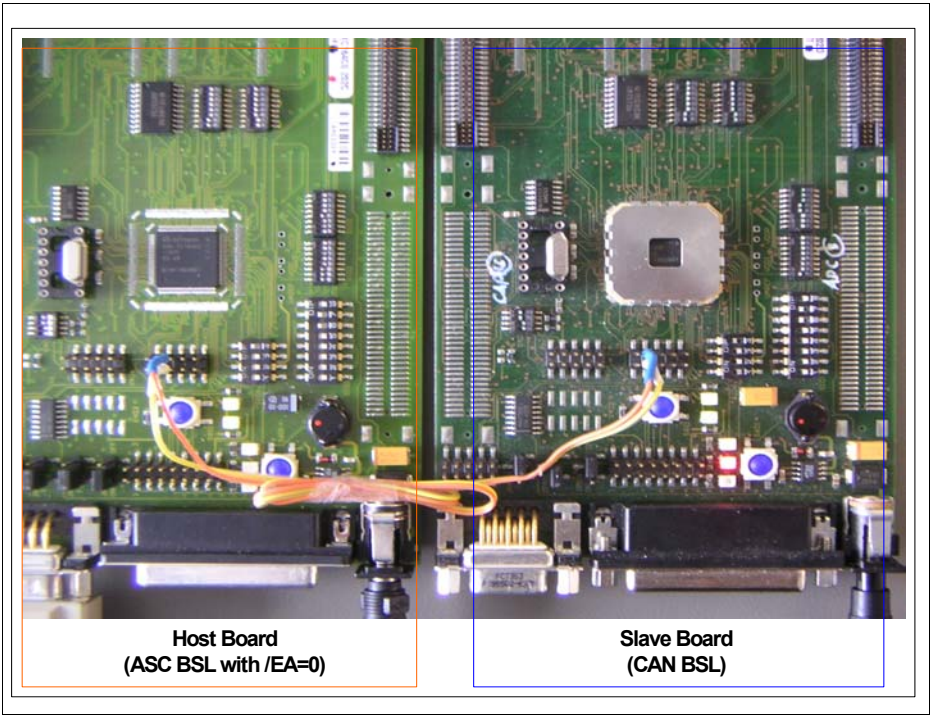

Figure 5 Boards Connection

After starting the ‘MiniMon’ you can program the user code (p9_4_toggle.hex) into the internal flash and load the application code (host_XC16xboard.hex) in the external memory. When the slave board enters the CAN BSL mode after a HW reset, the software routine in the host board can be started with tool instruction ’_jmp 00’ directly.

After a successful download sequence, pin 9.4 on the host board will be ON and the CAN BSL is terminated and the downloaded code (located at E0'0000$_H$) is now executing. The pin 9.4 (or pin1L.0) on the slave board blinks.

Note: You can also use other tools (e.g. CANalyzer) with CAN interface to download a hex file in XC16x with the CAN BSL directly.

Note: On the XC164CS/XC164CM Easy kit board port 9 is not available.

In XC16x device, only 2K PRAM (E0'0000$_H$-E0'07FF$_H$) is available and a max. MSG number of 255 (MSG_CNT= 0..0xFF) is allowed by CAN BSL. The received message will be restored from E0'0000$_H$ to E0'07F7$_H$.

The code restored in PSRAM may be the final application code. It may also contain a code sequence to change the system configuration and enable the bus interface to store the received data into external memory. For example, the flash driver may be loaded and used for the on-chip flash programming via CAN BSL. For details please refer to the application note "**XC166 Flash-on-the-Fly**".

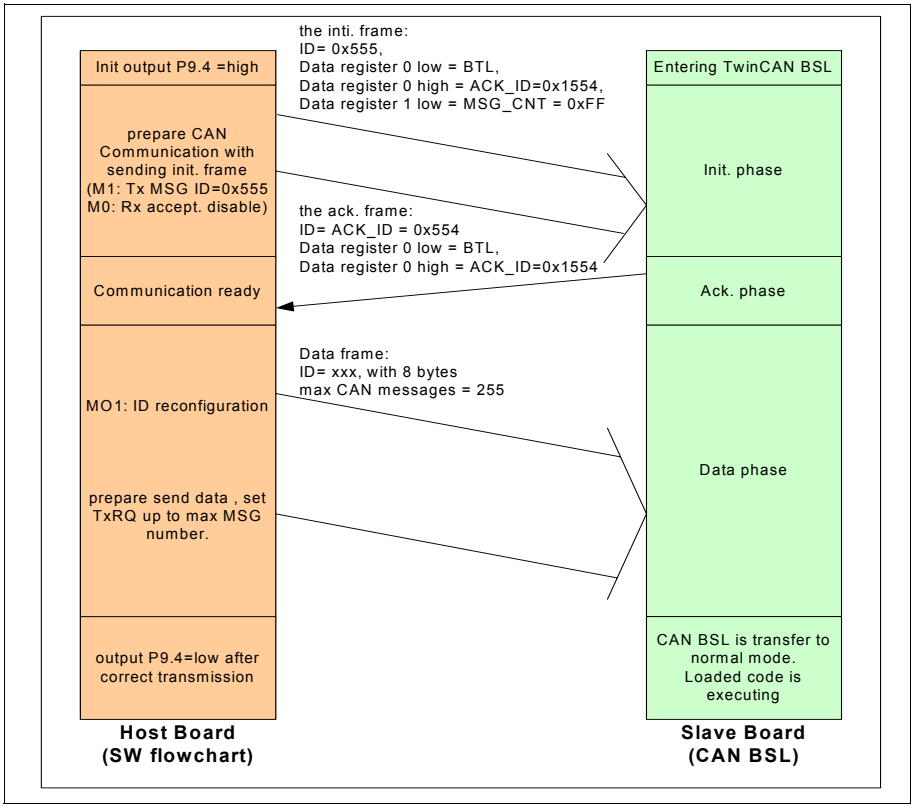

Figure 6 Date Communication on the CAN Bus

4 Appendix: Source Code

```
//*****************************************************************************
// @Function      void main(void)
//-----------------------------------------------------------------------------
// @Description   If the data trasmission is correct, P9_4 = 0 and an yellow
//                LED on the host board is then ON
//-----------------------------------------------------------------------------
// @Date          12.01.2006
//*****************************************************************************
void main(void)
{
  uword j, i=0;
  //--------------------------- init. phase ---------------------------------
  // - Port 9 is avalable in XC16x 100/144 pins device.
  // - TwinCAN Initialization on the host XC16x board
  // - Transmit the init. frame from MSG1, until it is acknowledged by the host
  P9_P4=1;
  DP9_P4=1;
  CAN_vInit();
  CAN_HWOBJ[1].uwMSGCTR = 0xe7ff;

  //--------------------------- ack. phase ----------------------------------
  // - Message object 0 is defined as receive object.
  // - Wait until the acknowledge frame is arrived(from TwinCAN_BSL(slave board))

  while(!((CAN_HWOBJ[0].uwMSGCTR & 0x0300) == 0x0200));

  //--------------------------- data phase ----------------------------------
  // - Use message object 1 to transmit the user code/data
  // - Transmit the data from address C0,0000-C0,07FF  (2K SPRAM)
  // - After a correct transition the LED is ON

  CAN_HWOBJ[1].uwMSGCTR = 0xfb7f;
  CAN_HWOBJ[1].ulCANAR  = 0x15540000;
  CAN_HWOBJ[1].ulCANAMR = 0xFFFFFFFF;
  CAN_HWOBJ[1].uwMSGCFG = 0x0088;
  CAN_HWOBJ[1].uwMSGCTR = 0xf7bf;

  for (j=1;j<((FLASH_2K)/4);j+=2)
  {
    while (!((CAN_HWOBJ[1].uwMSGCTR & 0x3000) == 0x1000));

    CAN_HWOBJ[1].uwMSGCTR = 0xfbff;
    CAN_HWOBJ[1].CMSGDRLn4.stData.ulDataLow = FlashData[j-1];
    CAN_HWOBJ[1].CMSGDRLn4.stData.ulDataHigh = FlashData[j];
    CAN_HWOBJ[1].uwMSGCTR  = 0xf6bf;

    CAN_HWOBJ[1].uwMSGCTR = 0xe7ff;
  }
  P9_P4=0;
  while (1);                        // for debug using
} //  End of function main

//*****************************************************************************
// @Function      void CAN_vInit(void)
//-----------------------------------------------------------------------------
// @Description   This is the initialization function of the CAN function
```

```
//                library. It is assumed that the SFRs used by this library
//                are in its reset state.
//                MO0 is defined as Rx object, MO1 is defined as Tx object
//----------------------------------------------------------------------------
// @Date          12.01.2006
//****************************************************************************
void CAN_vInit(void)
{
  //  ------------------------------------------------------------------------
  //  Configuration of CAN Node A:
  //  ------------------------------------------------------------------------
  CAN_ACR        =  0x0041;
  CAN_AGINP      =  0x0000;
  CAN_AECNTH     =  0x0060;
  ALTSEL0P4     |=  0x0040;
  DP4  = (DP4  & ~(uword)0x0040) | 0x0040;
  CAN_AFCRL      =  0x0000;
  CAN_AFCRH      =  0x0000;
  CAN_PISEL      =  0x0000;

  //  ------------------------------------------------------------------------
  //  Configuration of Message Object 0:
  //  standard 11-bit identifier, RX, CAN node A, acceptance mask dissabled
  //  ------------------------------------------------------------------------
  CAN_MSGCFGL0   =  0x0080;
  CAN_MSGCFGH0   =  0x0000;
  CAN_MSGCTRL0   =  0x5595;

  //  ------------------------------------------------------------------------
  //  Configuration of Message Object 1:
  //  standard 11-bit identifier =0x555, TX, 8 valid data bytes, CAN node A
  //  acceptance mask 0x7FF
  //  CAN_MSGDRL10= Bit Timing for TwinCAN_BSL in SlaveCAN
  //  CAN_MSGDRH10= ID of the ack. frame that CAN_BSL sends back to the host
  //  CAN_MSGDRL14= the number of messages to receive
  //  ------------------------------------------------------------------------
  CAN_MSGCFGL1   =  0x0088;
  CAN_MSGCFGH1   =  0x0000;
  CAN_MSGAMRL1   =  0xFFFF;
  CAN_MSGAMRH1   =  0xFFFF;
  CAN_MSGARL1    =  0x0000;
  CAN_MSGARH1    =  0x1554;

  // one MSG = 2 Long = 4 Word = 8 Bytes
  // the user is limited to sending a Max MSG = 255 (MSG Number 1 to 0xFF)
  // 2K SPRAM in XC16x (start address E0,0000- E0,07F7)
  CAN_MSGDRL14   =  (FLASH_2K)/8-1;
  CAN_MSGDRH14   =  0x0000;

  CAN_MSGFGCRL1  =  0x0000;
  CAN_MSGFGCRH1  =  0x0001;
  CAN_MSGCTRL1   =  0x5595;
  CAN_MSGCTRH1   =  0x0000;


  //  BTR and DataBytes[3-0]Configuration---------------------------------
  //  CAN_ABTRL, CAN_ABTRH: bit timing register
  //  CAN_MSGDRL10: DataBytes[1-0] => BTR for XC16x CAN_BSL
  //  CAN_MSGDRH10: DataBytes[3-2] => MSG ID for the CAN_BSL
```

```
  //  ------------------------------------------------------------------------

......

//##############################################################################
// 500 KBaud: CAN_BSL fcan > 16 MHz
//##############################################################################
// host XC16x board = 8 Mhz; TwinCAN-BSL = 16 Mhz
#if XC16X==ON
  CAN_ABTRL       =  0x2bc0;
  CAN_ABTRH       =  0x0000;
  CAN_MSGDRL10    =  0x2bc1;  // SP=81,25%, TSG2=2, TSG1=11, BRP=1, SWJ=3
  CAN_MSGDRH10    =  0x1554;
#endif

......

//##############################################################################
// 200 KBaud
//##############################################################################
// host XC16x board = 8 Mhz; TwinCAN-BSL = 8 Mhz
#if XC164CM==ON
  CAN_ABTRL       =  0x3EC1;
  CAN_ABTRH       =  0x0000;
  CAN_MSGDRL10    =  0x3EC1;  // SP=80%, TSG2=3, TSG1=14, BRP=1, SWJ=3
  CAN_MSGDRH10    =  0x1554;
#endif

......

  //  ------------------------------------------------------------------------
  //  Start the CAN Nodes:
  //  ------------------------------------------------------------------------
  CAN_PISEL      =  0x0000;      // load port input select register
  CAN_ACR       &= ~(uword)0x0041; // reset INIT and CCE
} //  End of function CAN_vInit
```

http://www.infineon.com

Published by Infineon Technologies AG