Application Note, V1.1, Jan. 2004

AP16048

XC166

Flash-on-the-Fly

A concept to flash via CAN

Microcontrollers

Never stop thinking.

Revision History:	2004-01	**V1.1**
Previous Version:	-	
Page	Subjects (major changes since last revision)	
6	Table moved to appendix, as only valid for AC material	

Edition 2004-01

Published by
Infineon Technologies AG
81726 München, Germany

Table of Contents Page

1 Introduction

1.1 Foreword

Manufacturers of automotive Electronic Control Units (ECUs) have the requirement to program microcontrollers at the end of the manufacturing process rather than programming the microcontroller before soldering. In some instances, the vehicle manufacturers need to perform a final programming sequence once the ECU is fitted into the vehicle.

In this paper, the possibility of such a concept is presented.

1.2 Short Overview

End-of-line programming: The end-of-line programming is supported by the on-chip CAN-bootloader. Using this bootloader, the programming device can send a routine to the microcontroller’s internal Program RAM (PRAM). This routine itself can program the Flash. The basic ideas of such a program are given in this application note.

Flash-on-the-Fly: A concept to reprogram the Flash without storing additional information in PRAM.

2 The On-chip CAN Bootloader

There are two ways to start the on-chip CAN bootloader, depending on the state of the /EA input. If /EA is high (internal start) then the configuration should be as follows:

(/EA = 1), /RD = 0 and ALE = 1

This configuration optimizes the configuration for internal memory and does not consider the PORT0 configuration. However, if /EA is low (external start), then the on-chip bootloader is started by a PORT0 configuration as follows:

(/EA = 0), SMOD = $(1001)_B$ (note: SMOD pins are P0L.5..2)

The bootloader waits for a message to be sent by the programmer. As soon as this message is sent, the bootloader tries to synchronize to the bus baud rate. This message should have its sample point set to 80%.

The 11-bit identifier should be chosen as 0x555, as this identifier changes the bus level every bit time. The format of the message shall be as follows:

Data Length Code = 5

Data Byte 0 = BTR Low

Data byte 1 = BTR High

Data byte 2 = ACK ID Low

Data byte 3 = ACK ID High

Data byte 4 = the number of messages to receive.

Where:

BTR is the value of the Node A bit timing register needed to set the baud rate to the value that should be used after the first contact. This mechanism allows the TwinCAN to switch to a higher baud rate following synchronization.

ACK ID (acknowledge ID) is the identifier of the handshake message that will be sent from the microcontroller after synchronization

The last data byte (number of messages) gives the number of messages that will be stored in PRAM before the boot loader automatically runs the downloaded code from PRAM.

As soon as the device to be programmed receives the first message, it calculates the baud rate by measuring bit times. When the baud rate is detected, the device acknowledges the message sent by the programmer (drives an active bit in the acknowledge bit slot). The device then sends a handshake message to the programmer at the baud rate defined by the BTR values received in the initial message from the programmer. The device is then ready to receive the specified number of messages. The messages have to be pure code, no addressing, as the program is downloaded one message after another to PRAM. After receiving the defined number of messages a Watchdog reset takes places and the code in PRAM starts running.

It is recommended that the first instruction downloaded into the PRAM is a DISWDT instruction to prevent watchdog timeouts resetting the device.

The data bytes contained within the next CAN frames sent (number defined in the first CAN message as described above) are stored into the internal PRAM, starting at address 0xE0’0000. For maximum efficiency, the data should be sent in groups of 8 bytes as per the maximum allowed per CAN frame.

3 Flash-on-the-Fly as a Concept

Flash-on-the-Fly itself shall include the program in PRAM with erase routines, program routines, initialization routines for the CAN module and two small interrupts. Those interrupts shall serve the CAN messages of the master program. Given a short overview of the concept:

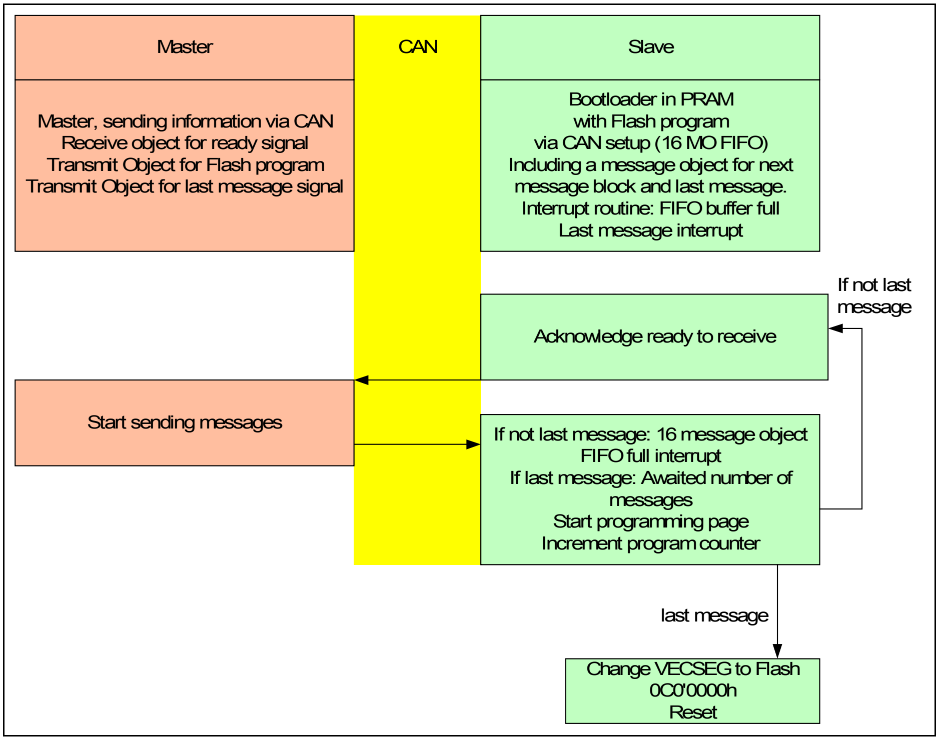
Master
CAN
Slave
Master, sending information via CAN
Receive object for ready signal
Transmit Object for Flash program
Transmit Object for last message signal
Bootloader in PRAM
with Flash program
via CAN setup (16 MO FIFO)
Including a message object for next
message block and last message.
Interrupt routine: FIFO buffer full
Last message interrupt
If not last
message
Acknowledge ready to receive
Start sending messages
If not last message: 16 message object
FIFO full interrupt
If last message: Awaited number of
messages
Start programming page
Increment program counter
last message
Change VECSEG to Flash
0C0'0000h
Reset

Figure 1 Basic concept of Flash-on-the-Fly

4 Functional Specification

The functional specification is divided into 3 parts:

1) Flash routines
2) CAN routines and interrupts
3) General e.g. stacks

4.1 Flash Routines

To get a working environment, the following routines have to be inside this code:

1) read sequence error
2) erase sector
3) program page buffer (AC step and higher routines)
4) read Busy flag

Read sequence error, gives the possibility to detect an error during erasing or programming. Erase sector, erases the sector up from the start address. Program page buffer, programs a page. Read Busy flag polls the busy as long as the current sequence is active.

4.1.1 Read Sequence Error

Declaration:	int Read_sequence_error(void)
Input:	nothing
Output:	Error (1) or no error (0).
Usage:	Tests if sequence error occurred or not.

4.1.2 Erase Sector

Declaration:	int Erase_sector(unsigned long int sector)
Input:	unsigned long int sector: Sector start address
Output:	Error (1) or no error (0).
Usage:	Erase a sector.

4.1.3 Program Page Buffer

Declaration:	void Program_page_buffer(void)
Input:	none
Output:	none
Usage:	Program page

4.1.4 Read Busy flag

Declaration:	void Read_Busy_flag(void)
Input:	none
Output:	none
Usage:	Poll the busy flag until it is reset

4.2 CAN Routines

To get the CAN functionality into this program, the following functions are needed

1) Init CAN
2) FIFO buffer full interrupt
3) Last MO interrupt

Init CAN gives the basic CAN initialization for a working protocol environment. The FIFO buffer full interrupt shall get the data out of the message objects, program this data into the page buffer and program the page. The last message interrupt, does actually the same as FIFO full interrupt, but it reacts without waiting on the buffer full event and changes the VECSEG to 0xC0 as well as it performs a software reset.

4.2.1 Init CAN

Declaration:	void Init_CAN(void)
Input:	none
Output:	none
Usage:	Initialize the CAN as well as the corresponding interrupts.

4.2.2 FIFO buffer full interrupt

Declaration:	void interrupt (CANSRC_0_IntNo) FIFO_full(void)
Input:	none
Output:	none
Level:	Level 5 Group 0
Usage:	Get data to Flash. Send receive ready signal

4.2.3 Last message interrupt

Declaration:	void interrupt (CANSRC_1_IntNo) Last_message(void)
Input:	none
Output:	none
Level:	Level 6 Group 0
Usage:	Get data to Flash and reset.

4.3 General

To have a complete system the following functions are required:

Cstart.asm:	Configure to internal start, define the stacks.
Main:	Configure CAN and enable interrupts.

4.3.1 Cstart

Declaration:	own file
Input:	none
Output:	none
Usage:	Configure for internal start and define user and program stack

4.3.2 Main Function

Declaration:	void main(void)
Input:	none
Output:	none
Usage:	Configure CAN and enable interrupts

5 Appendix

5.1 XC16x-16FF AC step

Table 1 CAN Data Byte Mapping

Data Byte	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Offset	2	3	0	1	6	7	4	5

For other devices afterwards, no data exchange is needed.

5.2 Baudrate Detection

In case a mechanism for the baudrate detection is needed, the following algorithm can be used:

1. Use timer to calculate the baudrate. (The length of one bittime corresponds to the baudrate)
2. Baudrate detected? Yes, goto 3 else goto 1
3. Switch on Analyzer mode
4. If two times LEC = 3, then switch off Analyzer Mode and start operation, else if LEC ≠ 3 in n loops, then go back to 1.

5.3 Source Code

```
//***********************************************************************
//                   Flash-on-the-Fly
//
//        A sample program to program the Flash via CAN.
//  To get the complete loader, pogram Master-CAN has to be used, as it
//  gives the possibility to use the CAN bootloader.
//  Runs at 16 Mhz (Otherwise change baudrate)
//  Baudrate: 100kBaud
//  Node chosen: A
//  Code tested on 128k Device
//  Runs up from XC16x-16FF AC
//
//  Originator: AI MC MA TM
//  Version:    1.0
//  March 2003
//
//***********************************************************************

#include "xc164cs_CAN.h"
```

```
#include "regxc164cs.h"

#define FALSE 1
#define OK    0
#define BTR 0x168F //100kBaud bei 16Mhz
#define CANSRC_0_IntNo 0x54
#define CANSRC_1_IntNo 0x55
#define value_CAN_0IC 0x54
#define value_CAN_1IC 0x58
unsigned long int Page=0;
//************************************************************************
// @Function      void read_FSR_Busy(void)
//------------------------------------------------------------------------
// @Description   This function read FSR Register
//------------------------------------------------------------------------
// @Returnvalue   none
//************************************************************************
void Read_FSR_Busy(void)
{
  unsigned int far *Flash_Register = (unsigned int far *) 0xFFF000;
  unsigned int Register;

  Register = *Flash_Register;
  while(Register & 0x0001)
  {
    Register = *Flash_Register;
  }
}

//************************************************************************
// @Function      int read_sequence_error(void)
//------------------------------------------------------------------------
// @Description   This function detects sequence errors
//------------------------------------------------------------------------
// @Returnvalue   Error yes/no
//************************************************************************
int Read_sequence_error(void)
{
  unsigned int far *Flash_Register = (unsigned int far *) 0xFFF000;
  unsigned int Register;

  Register = *Flash_Register;

  if(Register & 0x0040)
  {
    return(FALSE);
  }
  else
  return(OK);
}

//*********************************************************************
// @Function      int Erase_sector (unsigned long int sector)
//---------------------------------------------------------------------
// @Description   This function erases one sector
//---------------------------------------------------------------------
// @Parameter     Sector start address
//---------------------------------------------------------------------
// @Returnvalue   Success
//---------------------------------------------------------------------
// @Parameters    any constant or register value
//*********************************************************************
int Erase_sector (unsigned long int sector)
{
  unsigned int far *Flash_Command_1 = (unsigned int far *) 0x0c000AA;
  unsigned int far *Flash_Command_2 = (unsigned int far *) 0x0c00054;
  unsigned int far *Flash_Command_3 = (unsigned int far *) sector;

  //command sequence
  *Flash_Command_1 = 0x0080;
  *Flash_Command_2 = 0x00AA;
  *Flash_Command_3 = 0x0033;

  Read_FSR_Busy();
  return (Read_sequence_error());
}

//*********************************************************************
// @Function      void Program_page_buffer (void)
//---------------------------------------------------------------------
// @Description   This function programs one page using
//                the contents of the 16 message object
//                FIFO, starting at MO0
//---------------------------------------------------------------------
// @Returnvalue   none
//---------------------------------------------------------------------
// @Parameters    none
//---------------------------------------------------------------------
// @Date          ...
//*********************************************************************
void Program_page_buffer (void)
{
  unsigned int MO_no=0;
  unsigned int value;
  unsigned int far *Flash = (unsigned int far *) 0x0c000AA;
  unsigned int far *Load_Page = (unsigned int far *) 0x0c000F2;
  unsigned int far *Write_Page = (unsigned int far *) 0x0c0005A;
  unsigned int far *Page_destination = (unsigned int far *) Page;
  unsigned int far *CAN_MO = (unsigned int far *) 0x200300;

  //enter page mode
  *Flash = 0x0050;
  *Page_destination = 0x00AA;

  // load buffer...
  while (MO_no <= 0x100)
  {
     for (value=0; value<4;value++)
     {
       *Load_Page = CAN_MO[MO_no+value];
     }
     MO_no+=0x10;
  }
  Page += (sizeof(char) * 128);

  // write page
  *Flash = 0x00A0;     // PA
  *Write_Page = 0x00AA;

  Read_FSR_Busy();
}

//**********************************************************************
// @Function      int CAN_vInit(unsigned long int ID_receive,
//                              unsigned long int ID_ACK,
//                              unsigned long int ID_last)
//----------------------------------------------------------------------
// @Description   This function initilizes the CAN according
//                 to the proposed protocol.
//----------------------------------------------------------------------
// @Returnvalue   Success
//----------------------------------------------------------------------
// @Parameters    Identifiers
//**********************************************************************
int CAN_vInit(unsigned long int ID_receive,
              unsigned long int ID_ACK,
              unsigned long int ID_last)
{
  unsigned int between;

  PISEL=0x80;
```

```
ACR=0x41;                  // Basic settings
ABTRL=BTR;
AIMRH0=0x1;
AIMRL0=0x8000;
AIMR4=0x2;

DP4=0x40;
ALTSEL0P4=0x40;

between=((unsigned int)ID_receive)<<2;   // Convert ID to register

//    Write to arbitration register value between.
#pragma asm (@w1=between,@w2,@w3,@w4)  // MO 0 - 15
     MOV     @w2,#00h    ; @w2 = counter
     MOV     @w3,#030ah  ; @w3: base address
     MOV     @w4,#080h   ; @w4 DPP
  label1:
     EXTP    @w4,#01h
     MOV     [@w3],@w1
     ADD     @w3,#20h
     CMPI1   @w2,#0fh
     JMPR    cc_SLT,label1
#pragma endasm

MSGARH16=ID_last<<2;
MSGARH17=ID_ACK<<2;

//  For MO 0 to 17 CAN_MO_CTR[i]=0x5595;
#pragma asm(@w1,@w2,@w3,@w4,@w5)
    MOV     @w1,#05595h  ; value
    MOV     @w2,#00h    ; counter
    MOV     @w3,#310h    ; base address
    MOV     @w4,#80h
  label2:
    EXTP    @w4,#01h
    MOV     [@w3],@w1
    ADD     @w3,#20h
    CMPI1   @w2,#11h
    JMPR    cc_SLT,label2
#pragma endasm

MSGCTRL15=0x5599;
MSGCTRL16=0x5599;

MSGFGCRL0=0xf;
MSGFGCRH0=0x200;
```

```
  //  For MO 1 to 15 CAN_MO_MSGFGCR[i]=(unsigned long int)0x0300000f;
  #pragma asm (@w1,@w2,@w3,@w4,@w5)
      MOV @w1,#338h ; base address
      MOV @w2,#0    ; counter
      MOV @w3,#300h ; High word
      MOV @w4,#0fh  ; Low word
      MOV @w5,#80h  ; DPP
    label3:
      EXTP    @w5,#02h
      MOV     [@w1+#02H],@w3
      MOV     [@w1],@w4
      ADD     @w1,#020h
      CMPI1   @w2,#0fh
      JMPR    cc_SLT,label3
  #pragma endasm

  MSGCFGL17=0x88;
  MSGCFGH16=0x1;
  // all others 0x0

  ACR=0;  // Stop init

  MSGDRL170=0x5555;
  MSGDRH170=0x5555;
  MSGDRL174=0x5555;
  MSGDRH174=0x5555;

  CAN_0IC=value_CAN_0IC;
  CAN_1IC=value_CAN_1IC;

  return(1);
}

void interrupt (CANSRC_0_IntNo) FIFO_full(void)
{

  //  for (i=0; i<0xF0;i+=0x10)  // Shut of MO
  //  {
  //    CAN_MO_CTR[i]=0x5555;
  //  }
  #pragma asm(@w1,@w2,@w3,@w4,@w5)
     MOV     @w1,#05555h  ; value
     MOV     @w2,#00h     ; counter
     MOV     @w3,#310h    ; base address
     MOV     @w4,#80h
   label7:
     EXTP     @w4,#01h
     MOV      [@w3],@w1
     ADD      @w3,#20h
     CMPI1    @w2,#0fh
     JMPR     cc_SLT,label7
  #pragma endasm

  Program_page_buffer(); // Program page to page pointer
  //  for (i=0; i<0xF0;i+=0x10)
  //  {
  //    CAN_MO_Data0[i]=0x00000000;
  //    CAN_MO_Data4[i]=0x00000000;
  //  }
  #pragma asm(@w1,@w2,@w3,@w4,@w5) // Delete Data
      MOV      @w1,#00h   ; value
      MOV      @w2,#00h    ; counter
      MOV      @w3,#300h    ; base address
      MOV      @w4,#80h
    label8:
      EXTP     @w4,#04h
      MOV      [@w3],@w1
      MOV      [@w3+#02h],@w1
      MOV      [@w3+#04h],@w1
      MOV      [@w3+#06h],@w1
      ADD      @w3,#20h
      CMPI1    @w2,#10h
      JMPR     cc_SLT,label8
  #pragma endasm

  //  for (i=0; i<0xF0;i+=0x10)
  //  {
  //    CAN_MO_CTR[i]=0x5599;
  //  }
  //    For MO 0 to 16 CAN_MO_CTR[i]=0x5595;
  #pragma asm(@w1,@w2,@w3,@w4,@w5) // Switch on MO
     MOV      @w1,#05595h  ; value
     MOV      @w2,#00h     ; counter
     MOV      @w3,#310h    ; base address
     MOV      @w4,#80h
   label9:
     EXTP     @w4,#01h
     MOV      [@w3],@w1
     ADD      @w3,#20h
     CMPI1    @w2,#0fh
     JMPR     cc_SLT,label9
  #pragma endasm
  MSGCTRL15=0x5599;
```

```
  MSGCTRL16=0x5599;
  MSGCTRL17=0xe7ff;
}

void interrupt (CANSRC_1_IntNo) Last_message(void)
{
  int i;
  unsigned long int far
    *CAN_MO_CTR = (unsigned long int far *) 0x200310;

  for (i=0; i<0x110;i+=0x10) // Switch off
  {
    CAN_MO_CTR[i]=0x5555;
  }
  Program_page_buffer(); // Program

  #pragma asm // Change start address to Flash and reset
    MOV VECSEG,0c0h
    SRST
  #pragma endasm
}

void main(void)
{
  DP9_4=0x1; // Give alive sign
  SDA2=0x1;
  while(Erase_sector (0xC00000)); // Erase 128k of Flash
  while(Erase_sector (0xC02000));
  while(Erase_sector (0xC04000));
  while(Erase_sector (0xC06000));
  while(Erase_sector (0xC08000));
  while(Erase_sector (0xC10000));
  Page=0xc00000; // Set start address to beginning of Flash
  CAN_vInit(0x123,0x234,0x567);  // Init CAN
  IEN=1; // Enable interrupts
  MSGCTRL17=0xe7ff; // Give ready message
  SDA2=0x0; // I am running ...
  while(1);
}
```

This software has been written in Tasking. If you use Keil you need the keyword volatile for the Flash_Register, otherwise the FSR register will be read only once.