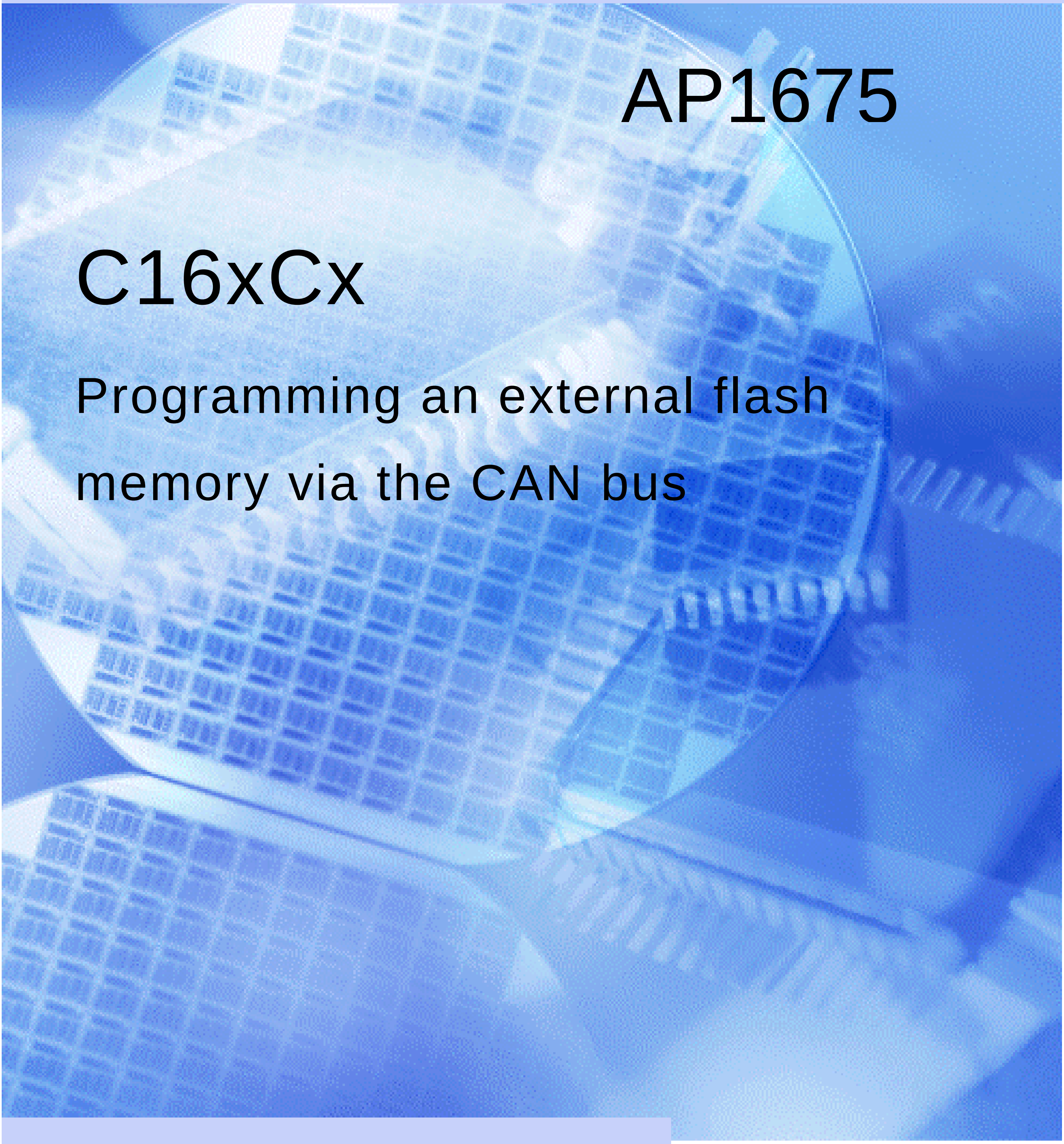
Application Note, V1.0, July 2001

AP1675

C16xCx

Programming an external flash memory via the CAN bus



Microcontrollers

Never stop thinking.

Revision History:	**2001-07**	V 1.0
Previous Version:	-	

Page	Subjects (major changes since last revision)

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

Edition 2006-07

Published by
Infineon Technologies AG
81726 München, Germany

© Infineon Technologies AG 2006.
All Rights Reserved.

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

1 Overview

Embedded systems based on microcontrollers and CAN (Controller Area Network) Bus are very widespread in Automotive and Industrial applications. To improve those embedded systems which control such devices as engine management systems, active suspension, ABS, gear control, A/C, airbag …it would be necessary and useful to be able to upgrade some embedded software via the CAN Bus.

This Application note describes how to implement in-system programming of an external flash device via the CAN bus. An example is provided using DAvE and the Keil tools. Certain features of the Keil tools make coding and copying the programming algorithms in C very fast and easy.

2 CAN Protocol Presentation

CAN is an asynchronous serial bus system with one logical bus line. A CAN bus consists of two or more nodes. The number of nodes on the bus may be changed dynamically without disturbing the communication of other nodes. This allows easy connection and disconnection of bus nodes (e.g. node for software upgrade, bus monitoring…).

The bus logic corresponds to a “wired-AND” mechanism, recessive bits are overwritten by dominants bits. As long as no bus node is sending a dominant bit, the bus line is in the recessive state, but a dominant bit from any node generates the dominant Bus State.

The maximum bus speed is 1Mbaud, which can be achieved with a bus length of up to 40m. At least 30 nodes may be connected without additional equipment.

Structure of a Standard data Frame generated by a CAN node:

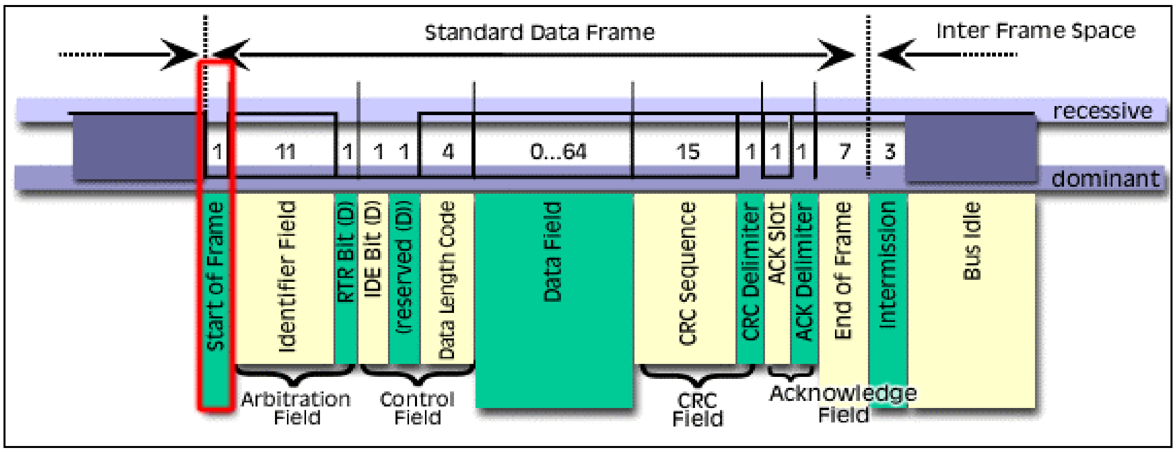


Figure 1 Standard CAN Data Frame

In this kind of frame, the identifier field has a length up to 11 bits giving 2048 message identifiers. But “extended CAN” specifications allow a 29 bits identifier field giving 536 Million messages identifiers. In this application note, only “standard CAN” will be used but C167CR and CS also support “extended CAN” specifications.

For bus arbitration, Carrier Sense Multiple Access / Collision Detection with Non-Destructive Arbitration is used. It means that if a node wants to transmit a message, it checks first that the bus is in the idle state (“Carrier Sense”). If this is the case the node becomes the bus master and sends its message. If many nodes start their transmission at the same time (“Multiple Access”) collision is avoided by bitwise arbitration (Collision Detection with Non-Destructive Arbitration). Each node sends the bits of its message identifier and monitors the bus level. A node which sends a recessive identifier bit but reads back a dominant one loses bus arbitration and switches to receive mode.

The CAN protocol allows a high data integrity with several means for error checking:

- Cyclic Redundancy Check Error.
- Acknowledge Error.
- Form Error.
- Bit Error.
- Stuff Error.

3 Hardware used in this Application Note

- Two Infineon Microcontroller Starter Kits C167CR or C167CS with Phytec Kitcon-167 equipped with 256K external Flash.
- CAN Analyzer, to see and send Can Bus Messages.
- CAN bus cable. If you haven't got one, you can easily make one compatible for the Infineon Microcontroller Starter Kits as shown in Figure 2:

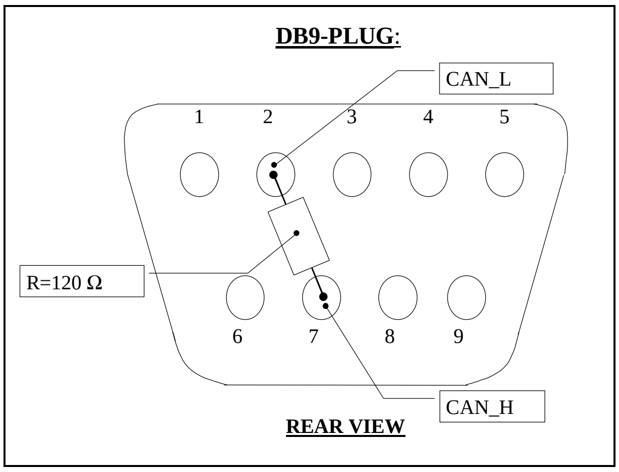


Figure 2 CAN Sub-D Connector

- One or two PCs with Win95/98/NT. Two PC is more comfortable, one for each board.
- A serial cable to connect the PC to C167Cx boards.

4 General Description of the System

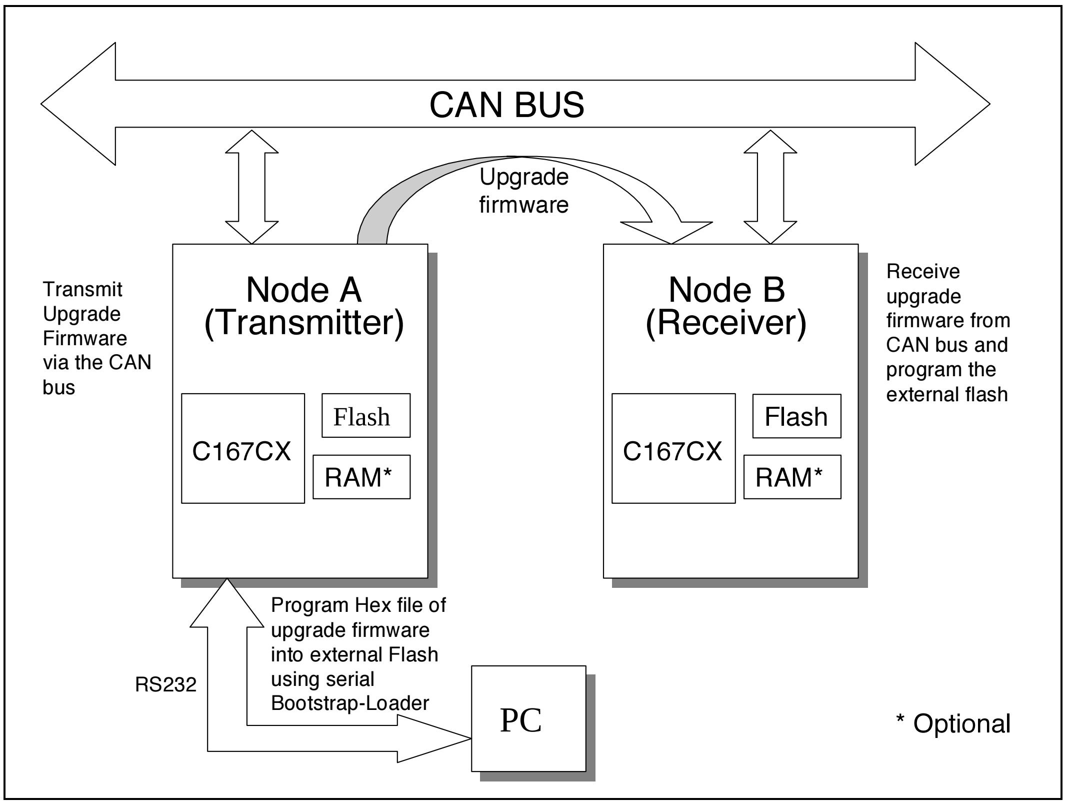

CAN BUS
Upgrade firmware
Transmit Upgrade Firmware via the CAN bus
Node A (Transmitter)
C167CX
Flash
RAM*
Node B (Receiver)
C167CX
Flash
RAM*
Receive upgrade firmware from CAN bus and program the external flash
RS232
Program Hex file of upgrade firmware into external Flash using serial Bootstrap-Loader
PC
* Optional

Figure 3 System Overview

To program a device via the CAN bus, at least two nodes must be present. The first node (node A) actually transmits the upgrade (application) firmware on the CAN bus.

The second node (node B) is the device to be upgraded. Node B receives the upgrade (application) firmware from the CAN bus and programs the external flash.

This application note contains software for both the transmitter and receiver. Node A may be a fixed node on the bus, or it might only be inserted when it is desired to upgrade the firmware on another node. This ApNote is organized in such a way that the hardware for the transmitter and the receiver boards can be identical. This is useful since it means that no special transmitter board needs to be developed. An extra receiver board can easily be reprogrammed to act as a transmitter. Note that neither the Transmitter or Receiver code require any external RAM, but the application program may use external RAM if required by the application.

The flash memory for the transmitter and receiver nodes is organized very similarly. The first sectors of flash in each node are reserved for CAN transmitting (transmitter node) or CAN reception and programming algorithms (receiver node). The rest of the flash is available for the application code. Note that this example reserves the first 64k of the flash. Actually less code is required. However since different Infineon microcontrollers have different memory maps, the entire 64k is reserved so that this ApNote can be easily ported to different controllers.

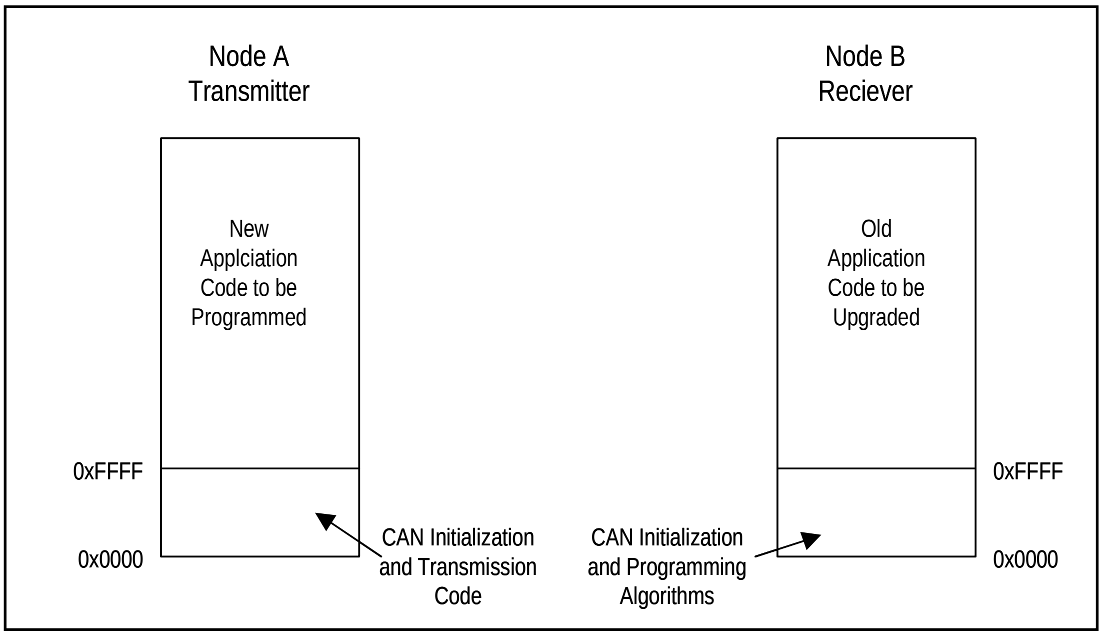

Node A
Transmitter
Node B
Reciever
New Applciation Code to be Programmed
Old Application Code to be Upgraded
0xFFFF
0x0000
0xFFFF
0x0000
CAN Initialization and Transmission Code
CAN Initialization and Programming Algorithms

Figure 4 Memory Map

The first 2 sectors of the receiver firmware need to be programmed once either in system with the serial bootstrap loader, or prior to assembly via a flash programmer. Once this firmware is programmed into these sectors, the application program can be programmed and updated via the CAN bus.

There are really three different pieces of code, or three projects associated with this Application Note. There is the CAN initialization and transmission code for the transmitter, the CAN initialization and programming code for the receiver, and the application code. First the CAN initialization and programming code must be programmed into the reserved sections of the receiver node. Then the CAN initialization and transmission code must be programmed into the reserved sections of the transmitter node. Once this is done, the application program is programmed into the unprotected sectors of transmitter node (either via a programmer or serial bootstrap loader). This code is then transferred via the CAN bus into the receiver node.

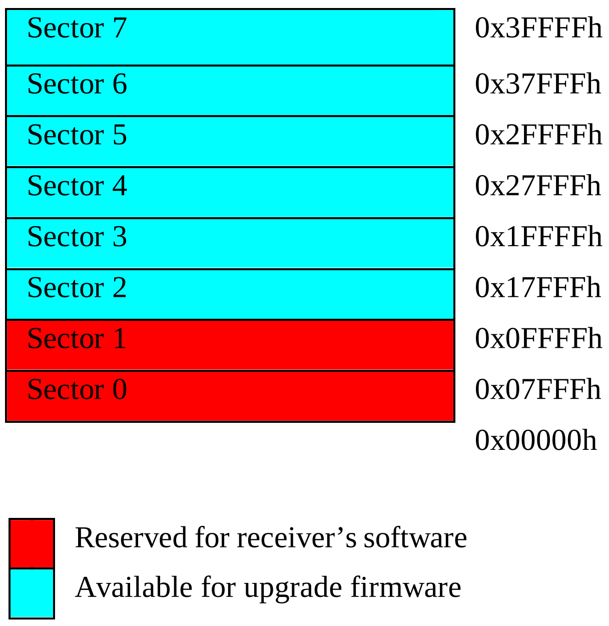
5 Flash Programming

The flash memory used on the Infineon microcontroller starter kits is the AMD29F010 or AMD29F010A or AMD29F010B. Each chip has a size of 128Kbytes. Infineon's starter kits based on C167CR or CS are equipped with two chips for a total of 256Kbytes. Each version of the flash memory should be fully compatible but some differences remain that will be described further in the "how to program the flash memory" section.

5.1 Structure of the Memory

The structure of the AMD29F010x is very simple; the memory is divided into 8 sectors of 16Kbytes. Since the flash memory on the starter kits are composed of two chips, the first one connected to lower bits of the data bus (D0-D7) and the second one connected to upper bits of the data bus (D8-D15), the onboard flash memory is divided into 8 sectors of 32Kbytes.

Onboard flash memory structure 256Kbytes



Note: In this Application Note, the first sectors of flash are used to store the programming algorithms. This means that the actual user application must reserve this *area. The user application should also “pretend” that the reset vector and all interrupt vectors are mapped above address 0xFFFF. The programming algorithm in the receiver firmware provided with this Application Note will not allow programming below address 0xFFFF.*

6 How to Program the Flash Memory

To perform some operations on the flash memory, except for reading operations, we have to write some command sequences. The following table gives all the possible operations and their command definitions:

Command Sequence		Cycles	Bus Cycles											
			First		Second		Third		Fourth		Fifth		Sixth	
			Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read		1	RA	RD										
Reset		3	5555	AA	2AAA	55	5555	F0						
Autoselect	Manufacturer ID	4	5555	AA	2AAA	55	5555	90	XX00	01				
	Device ID	4	5555	AA	2AAA	55	5555	90	XX01	20				
	Sector Protect Verify	4	5555	AA	2AAA	55	5555	90	(SA) X02	00				
										01				
Program		4	5555	AA	2AAA	55	5555	A0	PA	PD				
Chip Erase		6	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	5555	10
Sector Erase		6	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	SA	30

In this Application Note, it is assumed that the flash is organized as shown in the following figure. Therefore we must adjust the addresses of the commands definition because the address bus of the flash memory doesn't match with the address bus of the microcontroller. A0 is not connected to the flash.

Here is a schematic of how the flash memory is connected to the microcontroller on Infineon's starter kits using a 16-bit non-multiplexed bus connected to two 8-bit memories:

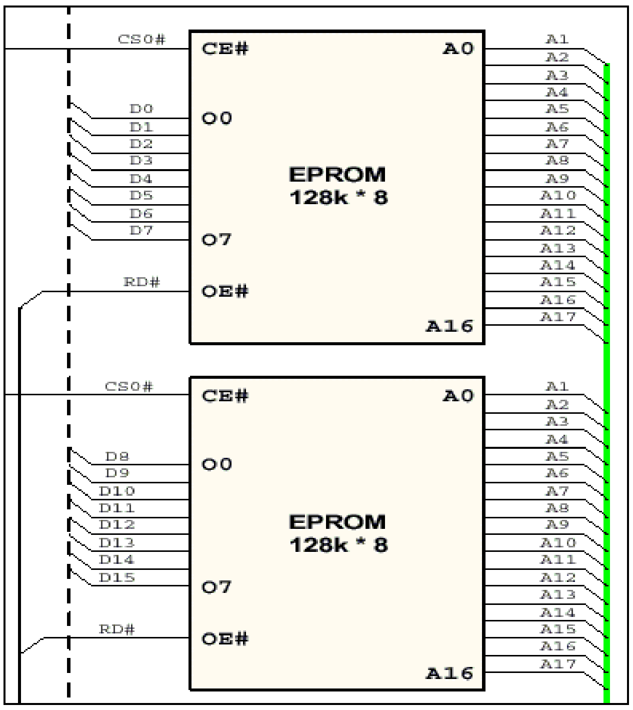


Figure 5 Memory Connections

We can easily see that the microcontroller‘s address bus is one bit left-shifted and a command address for the microcontroller is not the same command address for the flash memories.

So, to write the proper command sequence, you must replace the address values as follows:

-Change 0x5555 to: 0xAAAA.

-Change 0x2AAA to: 0x5554.

Note: -if your flash memory is an Amd29F010B, the addresses in the command sequences are different. 0x555 and 0x2AA replaces addresses 0x5555 and 0x2AAA. Of course, you have to calculate the right addresses to write with the microcontroller.

-If, as in this application note, you use a 16 bit non-multiplexed bus, you must write the data command in both the upper and lower bits of the data bus in order to make the same operation on the two chips of the flash memory at the same time.

6.1 Routines for Programming the Flash Memory

The Amd_Flash.c file includes all of the programming algorithms for the flash memory. If you are using another type of flash, you can simply modify this file to fit your needs. This Application Note provides algorithms for programming as well as sector and chip erasing. The constants associated with the programming algorithm are all contained in the file Amd_Flash.h.

To program or erase its flash memory, AMD uses an embedded algorithm. We must monitor the status of this embedded algorithm before attempting a new operation on the flash. AMD provides an algorithm, a data-polling Algorithm, which the system must respect to control the status of each operation.

Below is the data polling algorithm and the code associated:

```
uword data_polling(ulong flash_address, uword value)
{
uword data;
read:
data=*((uword far*) flash_address);
// read the data which has been written
if((data & 0x8080)==(value & 0x8080))
       return 1; //success
else
{
        if((data & 0x1010)==0x1010)
// if the fifth bit is one , read the data
// again otherwise timeout and failure.
        {
       data=*((uword far*) flash_address);
       if((data & 0x8080)==(value &
0x8080))
            return 1; //success
        else
             return 0; //failure

        }
        else
               goto read;
}

}
```

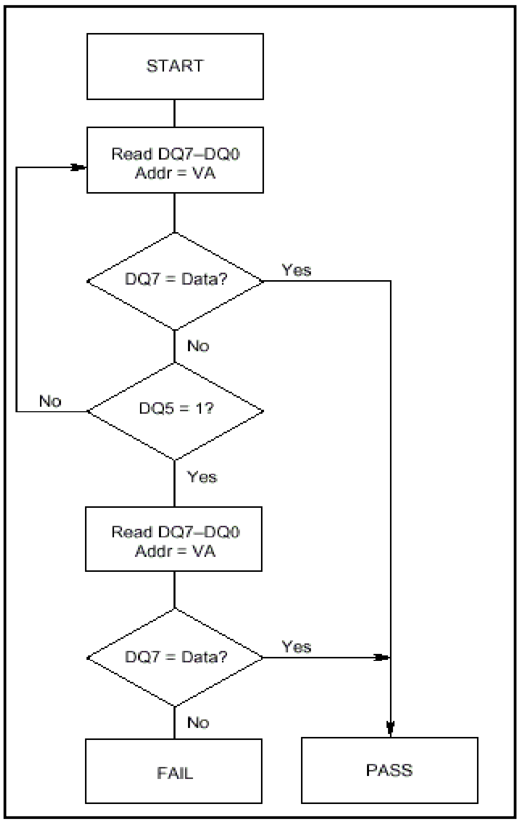


Figure 6 Data Polling Algorithm

7 Receiver Program

In the Node B receiver, there are two different hex files, the receiver program (containing the CAN initialization and flash programming algorithms) and the application program. The receiver program hex file is downloaded first (via a serial bootstrap loader or flash programmer) and its role is to program the application program (received via the CAN bus) into the external flash.

The receiver firmware is composed of two main parts: the checksum part and the programming part. The checksum part makes sure that valid code is in the flash memory and then initiates the programming mode if necessary. The programming part is divided in two sections: a section, which initializes the CAN peripheral, and a section which contains all the flash programming routines to program the flash memory.

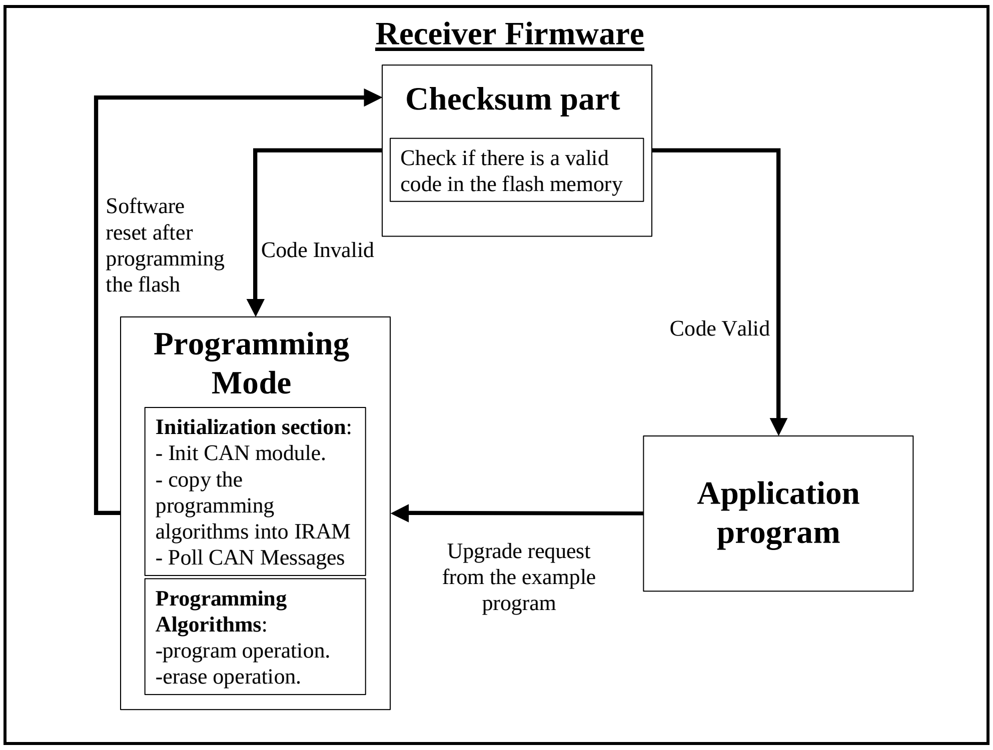


Figure 7 Receiver Firmware Structure

After a hard or soft reset, the receiver firmware executes a checksum. According to the result of this checksum, the receiver firmware jumps to the application program (address 0x10000) or into the programming mode.

Once the programming mode is entered, the CAN module is initialized. The programming algorithms are then copied into IRAM. The CAN module is then polled while the software waits for the application code to be transmitted.

After programming the flash memory, a software reset is generated and the checksum is executed again.

The example program can make, at anytime, a request to upgrade itself by simply jumping into the programming mode.

Programming the external flash presents two problems:

- We can’t program the flash memory with code located in the same flash memory. This means that the flash programming algorithms must be copied to and executed in the internal RAM.
- The application code must know where the programming mode code is located so it can upgrade itself.

For these reasons we will define two special sections in the receiver firmware. This allows us to know exactly where the code is located. We will also be able to move the programming algorithms section to IRAM, and jump exactly where we need to jump in the initialization section. This part will be detailed further.

7.1 Microcontroller Setup

The uVision2 project was setup using the C167CR, however the C167CS, C161CS or C164CI could also be used. Special care must be taken if the C164CI is used since it supports only a multiplexed external bus.

This ApNote was implemented on the evaluation board that comes with the C167CR starter kit. This board is called the KitCON167CR and was designed and manufactured by Phytec. This board contains 2 external 8-bit flash memories connected to Chip Select 0. The board also has 2 external 8-bit RAM memories connected to Chip Select 1, however external RAM is not required since the receiver code is so small.

The #BHE pin was configured to act a #WRH (for the flash memory connected to the upper half of the data bus) and the #WR pin acted as #WRL (for the flash memory connected to the lower half of the data bus). The bus was set up as a 16-bit de-multiplexed bus.

The CAN controller was setup with a baudrate of 1 Mbaud, using only two messages objects, and without using any interrupts. None of the receiver code uses interrupts since all of the interrupt vectors have been redirected so they can be used by the application code.

CAN message object 1 is setup as receive message with an 11-bit identifier that is set to 0x010. Of course this value can be modified to fit your system. The message 2 object is a transmit message with an identifier of 0x020 (this can also be changed). This message object is set up to transmit 6 data bytes.

7.2 Main.c

As previously mentioned in the overview, this project is divided in two parts, the checksum and the loader. The Checksum code is always located and executed from the external flash memory. After a hardware or software reset the checksum is executed. If the checksum matches the value that is programmed into a reserved location (the last 2 bytes of the flash memory), the application code is jumped to. Since the application code is located at address 0x10000, this can be accomplished easily by inserting an assembly instruction into the main() function. This will ensure that there is no garbage left on the stack. The following statement from the main() function shows how this is done.

```
////////// Jump to valid code (address 0x10000) //////////
        #pragma asm
                JMPS #1h,#0h
        #pragma endasm
```

In order to use this in-line assembly statement, the Keil compiler must generate a .SRC file. This is enabled easily through uVision2 by right clicking on the main.c file in the Project Window. Then make sure that the options “Generate Assembler SRC File”, “Assemble SRC File” and “Include in Target Build” are checked (with a dark checkmark).

If the checksum does not match with a value that is programmed into the reserved memory location (the last 2 bytes of the flash memory), then the programming mode is initiated.

When calculating the checksum, the value of each byte is incremented by 1. This means that blank flash bytes, which contain 0xFF, will not effect the value of the checksum. This is good because it means that the checksum program does not need to know the size of the Application code. It can just perform a checksum over the entire flash contents (except the reserved sectors which contain the receiver firmware).

7.3 Program_Mode.c

The programming mode code is more complex. First it initializes the CAN module. Then it must copy the program, erase and data-polling code into the IRAM since it cannot execute these routines out of the flash. The Keil toolchain has special linker options and macros which make this easy. The Keil Application Note 138 describes these features in detail.

The programming algorithm code is all located in one file called AMD_flash.c. A pragma statement near the top of this file assigns a class name to all of the code in this file:

#pragma RENAMECLASS (NCODE=PROG_ALGO)

The Keil Linker allows you to store a section in one location, but execute the code from another location. This is great since it makes it easy to store the programming algorithms in the flash, but specify that they will be executed out of IRAM. In the Keil uVision2 EDE, you can specify this with the following linker option for User Sections:

```
?PR?AMD_FLASH%PROG_ALGO(0xF600)[]
```

This specifies that the programming algorithms will be compiled as if they would be executed from addresses starting at 0xF600 (inside the IRAM). But they will actually be stored elsewhere in the flash. The empty brackets ([]) specify that the code will be stored anywhere in the SROM class memory. The SROM class is a special class created solely for these types of applications. We must specify the location of the SROM class with another linker option for User Classes:

```
SROM(0x0-0x7FFF)
```

This means that the programming algorithms can be stored anywhere in the first 32k of flash.

Copying the programming algorithms from the flash to the IRAM can be done quite easily in C using several macros and a library function that are provided with the Keil compiler. The macros are located in the header file SROM.h, and the library function is declared in the header file STRING.h. Both of these header files are located in the /keil/c166/inc directory of your Keil installation.

In the init_program_mode() function a macro is used to declare some variables which will contain the source location, destination location, and size of the programming algorithms. The hmemcopy library function is then used to actually copy the contents of the PROG_ALGO section (the Amd_Flash.c file) into IRAM.

Now, when any of the routines that are defined in the Amd_Flash.c file are called, the CPU will make calls to appropriate location in the IRAM.

If the application program wants to upgrade itself, it must jump to the init_program_mode() function. To do this, the application code must know where the function is located. There are many ways to do this, but the most simple way is to simply look in the linker map file and see where the linker placed this function. The following is an excerpt from a linker map file which shows that the function init_program_mode() is located at address 0x0644:

```
VALUE      PUBLIC SYMBOL NAME  REP   TGR  CLASS   SECTION
=====================================================================
00069CH    hmemcpy             LABEL  ---  NCODE   ?PR?HMEMCPY
000644H    init_program_mode   LABEL  ---  NCODE   ?PR?PROGRAM_MODE
0004E6H    main                LABEL  ---  NCODE   ?PR?MAIN
0004F8H    process_command     LABEL  ---  NCODE   ?PR?PROGRAM_MODE
```

One more important part about this file must be considered. What would happen if the Application code has a different location for its stacks and registers banks? What if the register banks of the application code are in the memory location that we want to copy the programming algorithms into? This would cause some pretty significant problems. To get around this problem, the function init_program_mode() re-initializes the DPPs, System and User Stacks and Context Pointer. This will allow the programming mode to continue even if the Application program located above 0xFFFF has different values for these pointers.

Re-initializing all of the pointers is easily done in Assembly. We use in-line assembly so that the rest of the programming functions can be done in C. The pointers are actually initialized in the assembly startup file (Start167.a66). Most of the pointers are initialized by the linker, so we simply import the pointer values using EXTRN statements. In order to use the in-line assembly statements, the Keil compiler must generate a .SRC file. This is enabled easily through uVision2 by right clicking on the Program_Mode.c file in the Project Window. Then make sure that the options "Generate Assembler SRC File", "Assemble SRC File" and "Include in Target Build" are checked (with a dark checkmark).

The Application code can simply jump to this address or use a function pointer when it wants to upgrade itself.

7.4 Start167.a66

The file Start167.a66 is used to initialize the microcontroller. A generic version of this file is provided by Keil. The user simply set the appropriate constants in this file. There are only three significant changes in this file that should be noted.

The symbol for the top of the User Stack (?C_USERSTKTOP) had to be made a public variable so that it could be exported to the init_program_mode() function. This allows the init_program_mode() function to re-initialize the user stack incase the Application program that is above address 0xFFFF places the user stack someplace else.

Another significant change in this file is that the definition of the registerbank is done differently. The normal way in which the Keil start-up file initializes the registerbank does not allow the location of the bank to be exported. To get around this, we created a section in IRAM and copied its location into the context pointer. We can then export this location to the init_program_mode() function.

The last significant change in the Kiel assembly start-up file is that the EINIT instruction is commented out. This instruction locks the contents of the SYSCON register. If we execute this instruction, then the Application program will not be able to modify the contents of SYSCON. Commenting this instruction allows the assembly start-up file for the Application program to use whatever SYSCON settings are desired.

7.5 Int_vector.a66

It is highly likely that the Application program which will reside at addresses above 0xFFFF will want to use interrupts. However the interrupt vectors are all hardwired to the lower addresses which are reserved for the receiver firmware. This presents a slight problem, however this is easily overcome. At each of the interrupt vector locations, instead of placing a jump to an interrupt service routine, we place a jump to the address of the vector plus 0x10000. The file Int_vector.a66 is an assembly file which places the jump instructions at the proper addresses.

The Application firmware must then tell the Keil tools that the interrupt vectors have been moved to address 0x10000. This is described in the section about the Application code.

8 Application Program

The Application program will have whatever functionality that you desire. A sample application program is provided in this Application Note so that you can see what types of things must be considered in setting up the project. The sample application is quite simple. It sets up the Timer 3 of the GPT1 block to overflow periodically. An ISR is activated every time the timer overflows. The ISR simply lights up a different LED on the evaluation board. The code also sets up the CAN peripheral so that it can receive messages. If the application receives the proper message, it will jump to the initialization code of the program mode which is located in the receiver firmware. This allows the application code to upgrade itself.

8.1 Project Setup

The project was setup in uVision2 just like any normal project, but with the following exceptions:

1. The external ROM memory was set to start at address 0x10000 and extend to address 0x3FFFE. The last two bytes of flash are reserved to hold the checksum.
2. The Interrupt Vector Table Address was set to address 0x10000 in the miscellaneous linker options. This matches with jump address programmed by the receiver application into the real vector table that starts at address 0x0000. This will allow the application code to use all interrupts.

8.2 CAN.c

This file is used to setup the CAN peripheral and to handle CAN interrupts. The example code uses CAN messages to signal when it is time to upgrade itself. Your application may use something else (eg. it may poll a port pin). Or you application may build the “self upgrade feature” into the protected receiver code for safety.

The only thing special about this file is that it jumps to the init_program_mode() function when it receives the appropriate CAN message. The location of this function was determined by examining the linker map file from the Receiver project as described earlier. A function pointer could be used instead of an in-line assembly jump. Since we use in-line assembly the options for the CAN.c file were set so that a .SRC file was generated and assembled.

9 Transmitter Program

The Transmitter program is the interface between development environment (usually a PC) and the CAN bus. The user must program the transmitter code into the first 2 sections of the flash of the transmitter node. This is easily done with an external programmer or a PC application (like the Phytec FlashTools) that uses the serial bootstrap loader of the microcontroller. The user must then program the Application hex file into the upper addresses of the flash on the transmitter node. Once this is done, a dumb terminal such as HyperTerminal can be used to interface with the transmitter board via the serial port. From the terminal the user can send commands that instruct the transmitter node to send CAN messages to the receiver. Details about this process will be described later. Figure 8 shows this process.

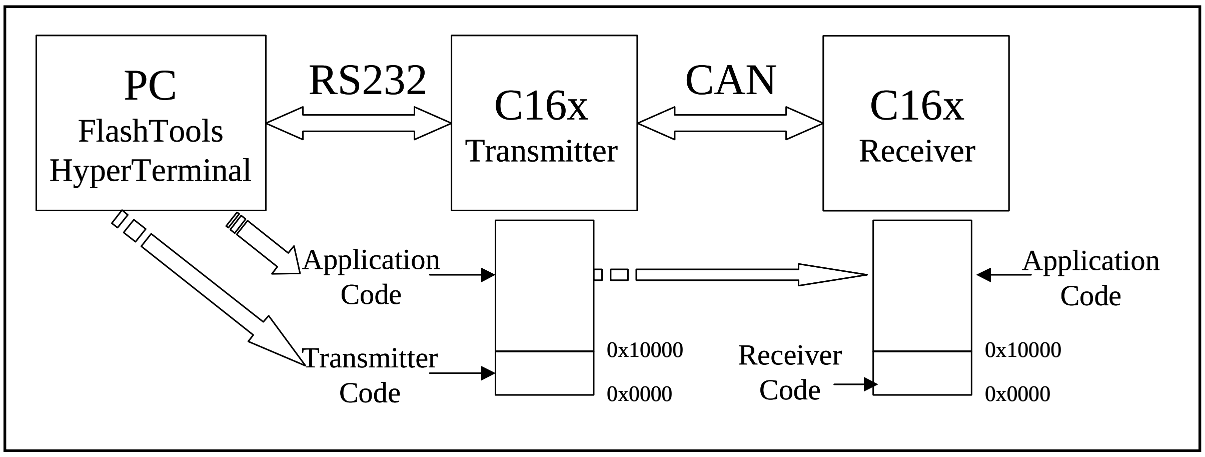


Figure 8 System Communication Flow

The Transmitter program uses only two peripherals of the microcontroller. The USART is used to communicate serially to HyperTerminal on the PC. The CAN peripheral is used to communicate to the Receiver board. The only interrupt that is used is the USART receive interrupt.

The Transmitter node and the Receiver node use a simple protocol when communicating on the CAN bus. The transmitter sends a command to the receiver (e.g. program one word, erase a sector, perform a software reset). After the receiver actually completes the command (except for the software reset command) it sends a message back to the transmitter with the same command. This tells the transmitter that it is OK to send another command.

The file ASC.c contains most of the useful functions. The function ASC_viISR() is the serial receive interrupt service routine. When the user types a command into the HyperTerminal, this interrupt occurs. The function interprets the command and then sends the appropriate CAN messages which instruct the receiver node.

The only thing that is tricky about the setup of the Transmitter code is that the linker settings must be setup so that all of the code is located below address 0x10000.

To save time and CAN bus load, before sending a command to the receiver node to program a 16-bit value into the flash, the transmitter first checks the value. If the value is 0xFFFF, then the transmitter will not send the message since this is the default state of an erased flash.

10 Testing this Application Note

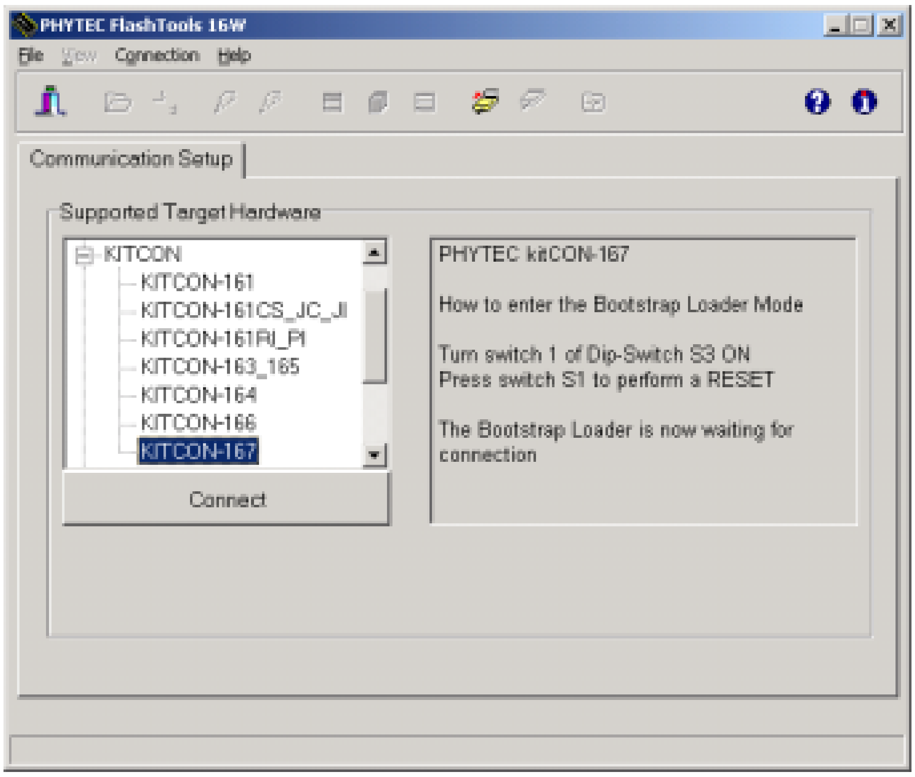
To test this Application Note, you will require 2 kitCON167 evaluation boards (with C167CR devices). This type of board is included in every C167CR Starter Kit. You will also need a CAN cable as described earlier. The only software required is the Phytec FlashTools (provided on the starter kit CDROM) and HyperTerminal (provided by MicroSoft with its operating systems).

10.1 Configuring the Receiver Node

To configure the receiver you must use the Phytec FlashTools. There is a DOS and a Windows version of the FlashTools program. The pictures below show the Windows interface. If you don't have the Windows version you can get it for free from Phytec, or you can use the DOS version.

First you must connect the receiver board to the PC via a serial cable. Switch #1 on the S3 bank of DIP switches on the kitCON board must be in the ON position to activate the bootstrap loader of the C167CR.

After starting the FlashTools program you must establish a connection to the kitCON board as shown below. Note that the DOS version of FlashTools automatically connects to the board when executed.



PHYTEC kitCON-167

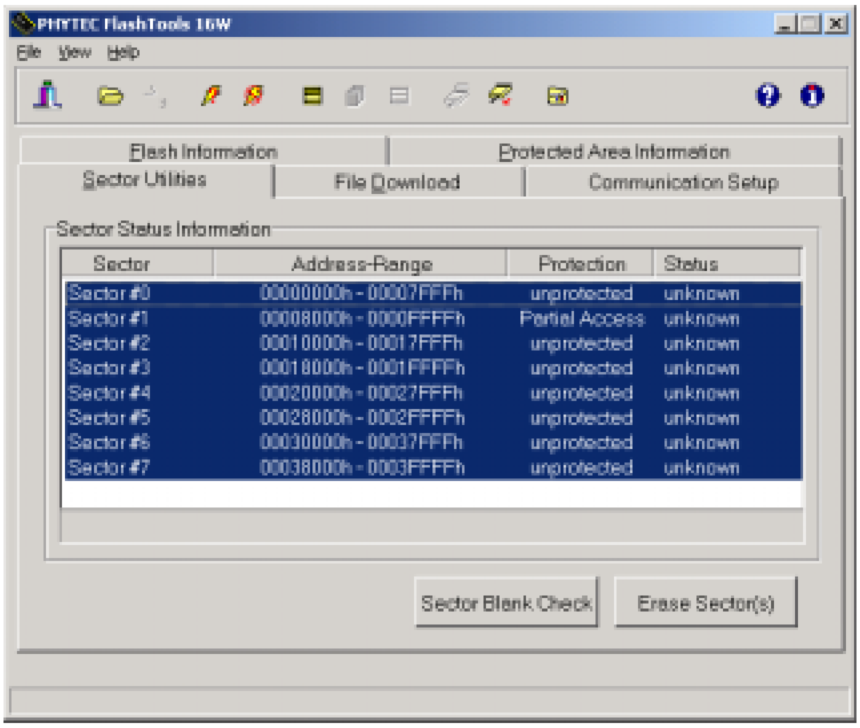
How to enter the Bootstrap Loader Mode

Turn switch 1 of Dip-Switch S3 ON
Press switch S1 to perform a RESET

The Bootstrap Loader is now waiting for connection

Figure 9 Phytec Flash Tools

After connecting to the board, you must erase the flash. In the Windows version of FlashTools, you can do this by simply highlighting all of the flash sectors and clicking the “Erase Sector(s)” button as shown.



Sector	Address-Range	Protection	Status
Sector #0	00000000h - 00007FFFh	unprotected	unknown
Sector #1	00008000h - 0000FFFFh	Partial Access	unknown
Sector #2	00010000h - 00017FFFh	unprotected	unknown
Sector #3	00018000h - 0001FFFFh	unprotected	unknown
Sector #4	00020000h - 00027FFFh	unprotected	unknown
Sector #5	00028000h - 0002FFFFh	unprotected	unknown
Sector #6	00030000h - 00037FFFh	unprotected	unknown
Sector #7	00038000h - 0003FFFFh	unprotected	unknown

Figure 10 Sector Utilities

After erasing the flash, the receiver firmware must be downloaded into the first 2 sectors of the flash. This is done from the “File Download” tab. You must download the file “Receiver.h86”.

After this is complete, you must disconnect FlashTools and put the DIP switch #1 back into the OFF position. You should then hit the RESET button on the evaluation board. The receiver node is now ready to go!

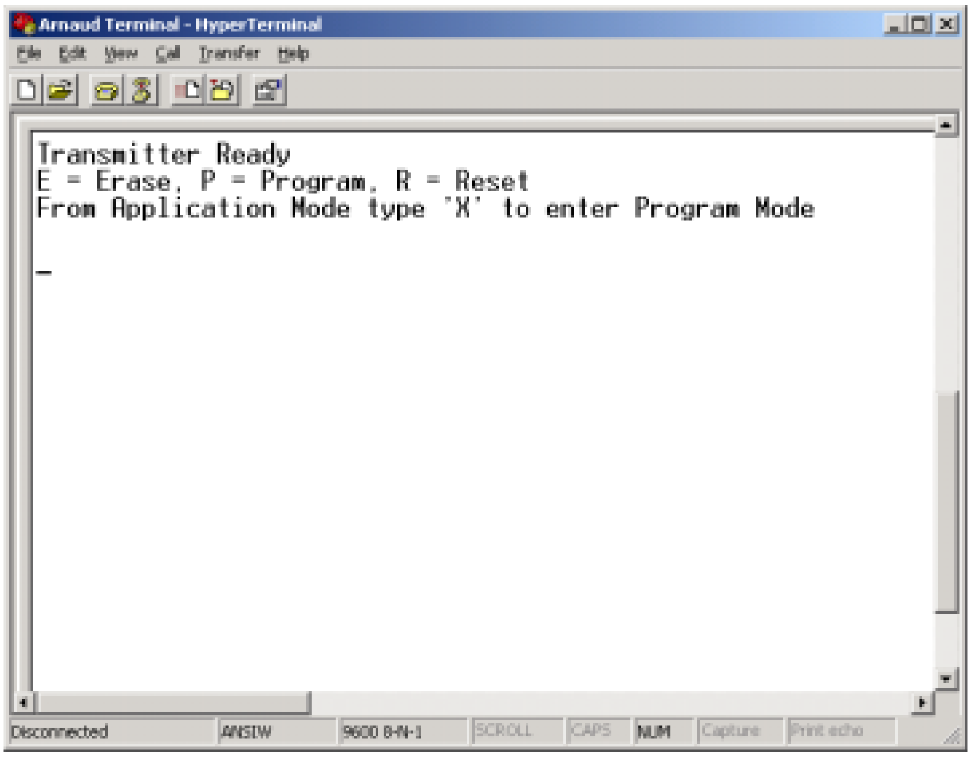
10.2 Configuring the Transmit Node

Configuring the Transmit node is similar to the Receive node. You must use the FlashTools to erase the flash on the evaluation board. You must then program 2 hex files into the flash. You should program the files “Transmitter.h86” and then “Application.h86” into the flash.

Once this is done, set the DIP switch #1 back to the OFF position.

Now you should exit the FlashTools software and start a HyperTerminal session. Configure the HyperTerminal so that is uses the COM port on your PC with a 9600 baud rate, 1 stop bit, 8 data bits, no parity and no flow control.

Once the HyperTerminal is connected, press the RESET button on the evaluation board, and you should see the message shown.


Arnaud Terminal - HyperTerminal
File Edit View Call Transfer Help
Disconnected
ANSIW
9600 8-N-1
SCROLL
CAPS
NUM
Capture
Print echo

```
Transmitter Ready
E = Erase, P = Program, R = Reset
From Application Mode type 'X' to enter Program Mode


_
```

Figure 11 HyperTerminal

The transmitter program allows you to perform 4 operations:

- Erase the unprotected sectors on the flash of the Reciever node
- Program the flash on the Receiver node with the contents of the flash on the Transmitter node (above address 0xFFFF).
- Make the Receiver node perform a software reset
- If the Reciever node is running the application code (if the LEDs on the evaluation board are blinking, then the application code is running) then it can jump into the programming mode

Make sure the transmitter and receiver are connected via the CAN cable. To erase the flash on the receiver, press 'E'. Once this operation is completed, you can press 'P' to program the Application code into the flash of the receiver. To execute the application code press 'R'. This causes the receiver to perform a software reset. If the checksum of the application code is correct, then it will automatically execute. You should see the LED's of the receiver node evaluation board start to blink.

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.

Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

h t t p : / / w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG