AN10835

LPC2000 secondary bootloader for code update using IAP

Rev. 01 — 26 May 2009 **Application note**

Document information

Info	Content
Keywords	LPC2000, Secondary bootloader, IAP, Code update
Abstract	This application note describes the design and implementation of a secondary bootloader which can update the user application code in on-chip flash via UART with 1K XMODEM protocol, SD/MMC with file system, EEPROM with I2C interface and CAN interface using IAP (In-Application Programming).

founded by Philips

Revision history

Rev	Date	Description
01	20090526	Initial version

Contact information

For additional information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

In most LPC2000 devices, the primary bootloader is the firmware which resides in the Boot Block and is executed every time the part is powered on or reset. The secondary bootloader in this document refers to a user-defined application that provides the user with an option to update the code or execute the previously programmed code.

In Application Programming (IAP executes erase and write operations on the on-chip flash memory, as directed by the end-user application code.

Code update is a typical application of IAP. This document describes four secondary bootloaders using different interfaces (UART using the 1K XMODEM protocol, SD/MMC using a FAT file system, I2C interfaced to EEPROM and CAN).

2. LPC2000 flash programming

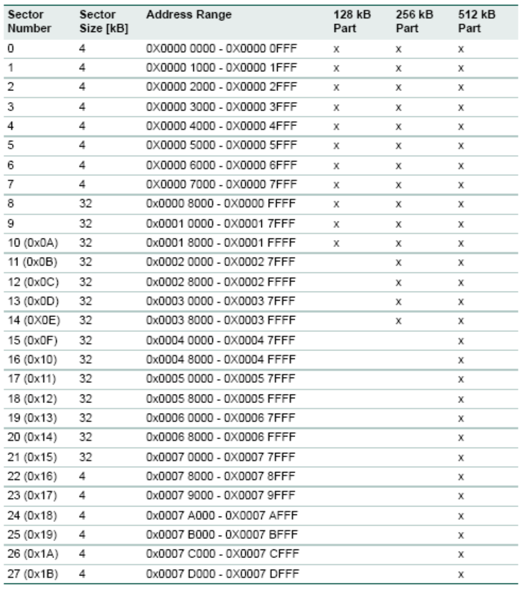
2.1 Sector description

The IAP commands operate on a sector-by-sector basis. This means that in order to make any modifications (even if it is just one byte) in a particular sector, the entire sector must be erased.

The user application and the secondary bootloader share the same on-chip flash space. This means that the user's application code should not be placed inside any of the sectors on which the bootloader resides. Therefore, the application should be erased and programmed in sectors – separate from the bootloader.

IAP, ISP, and RealMonitor routines are located in the primary bootloader (Boot block). The boot block is present at addresses 0x0007 E000 to 0x0007 FFFF (8 kB) in all devices. Depending on the device, not all of the flash is available to the user. This is the case if the part contains 512 kB of flash. Rather than having 512 kB available, the user will only have 504 kB available for the application. In devices that have less than 512 kB flash available, please refer to the respective user manuals.

Fig 1 indicates the correspondence between sector numbers and memory addresses for LPC23xx devices containing 128, 256 and 512 kB of flash respectively. For other LPC2000 devices please refer to the user manual.



Sector Number	Sector Size [kB]	Address Range	128 kB Part	256 kB Part	512 kB Part
0	4	0X0000 0000 - 0X0000 0FFF	x	x	x
1	4	0X0000 1000 - 0X0000 1FFF	x	x	x
2	4	0X0000 2000 - 0X0000 2FFF	x	x	x
3	4	0X0000 3000 - 0X0000 3FFF	x	x	x
4	4	0X0000 4000 - 0X0000 4FFF	x	x	x
5	4	0X0000 5000 - 0X0000 5FFF	x	x	x
6	4	0X0000 6000 - 0X0000 6FFF	x	x	x
7	4	0X0000 7000 - 0X0000 7FFF	x	x	x
8	32	0x0000 8000 - 0X0000 FFFF	x	x	x
9	32	0x0001 0000 - 0X0001 7FFF	x	x	x
10 (0x0A)	32	0x0001 8000 - 0X0001 FFFF	x	x	x
11 (0x0B)	32	0x0002 0000 - 0X0002 7FFF		x	x
12 (0x0C)	32	0x0002 8000 - 0X0002 FFFF		x	x
13 (0x0D)	32	0x0003 0000 - 0X0003 7FFF		x	x
14 (0X0E)	32	0x0003 8000 - 0X0003 FFFF		x	x
15 (0x0F)	32	0x0004 0000 - 0X0004 7FFF			x
16 (0x10)	32	0x0004 8000 - 0X0004 FFFF			x
17 (0x11)	32	0x0005 0000 - 0X0005 7FFF			x
18 (0x12)	32	0x0005 8000 - 0X0005 FFFF			x
19 (0x13)	32	0x0006 0000 - 0X0006 7FFF			x
20 (0x14)	32	0x0006 8000 - 0X0006 FFFF			x
21 (0x15)	32	0x0007 0000 - 0X0007 7FFF			x
22 (0x16)	4	0x0007 8000 - 0X0007 8FFF			x
23 (0x17)	4	0x0007 9000 - 0X0007 9FFF			x
24 (0x18)	4	0x0007 A000 - 0X0007 AFFF			x
25 (0x19)	4	0x0007 B000 - 0X0007 BFFF			x
26 (0x1A)	4	0x0007 C000 - 0X0007 CFFF			x
27 (0x1B)	4	0x0007 D000 - 0X0007 DFFF			x

Fig 1. Sectors in LPC2300 devices

2.2 In Application Programming (IAP)

2.2.1 IAP introduction

A boot code 8 kB in size is programmed into the on-chip flash after factory. The boot code controls initial operation after reset and also provides the means to accomplish programming of the flash memory. This could be initial programming of a blank device, erasure and re-programming of a previously programmed device, or programming of the Flash memory by the application program in a running system.

In Application Programming (IAP) executes erase and write operations on the on-chip flash memory, as directed by the end-user application code.

2.2.2 IAP application

Using IAP, users can update the application code by various communication interfaces such as UART, USB or Ethernet. Flash sectors that aren't used for the secondary bootloader or the user application may be used as non-volatile data storage.

While the application is running, the user can update some portion of the code using IAP commands which we call "online code updates". It is not necessary to power off or even to remove the chip from the board to have it serviced by some commercial programming tools.

Having the device work as data storage, PCB costs and sizes can be reduced. Caution is however advised in handling data storage sectors. Since sectors functioning as data storage can be erased, no application code should be contained in these sectors.

Frequent erasing and programming will reduce the flash's lifecycle. The LPC2000 provides a minimum of 10,000 write/erase cycles and 10 years of data retention.

2.2.3 IAP commands

For in application programming, the IAP routine should be called using a word sized pointer that has been loaded into register r0, which is pointing to memory (on-chip RAM) containing the command code and its parameters. Register r1 contains the results of the IAP command returned by a pointer (pointing to a table). The user can reuse the command table for the results by passing the same pointers into registers r0 and r1. Ensure that the results table is large enough to store all the results coming from the IAP command issued. Refer to Fig 2.

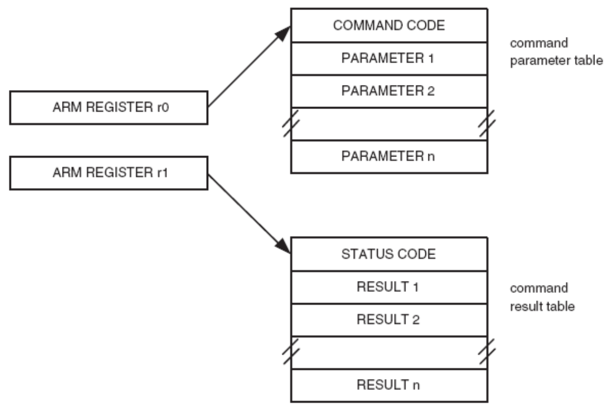
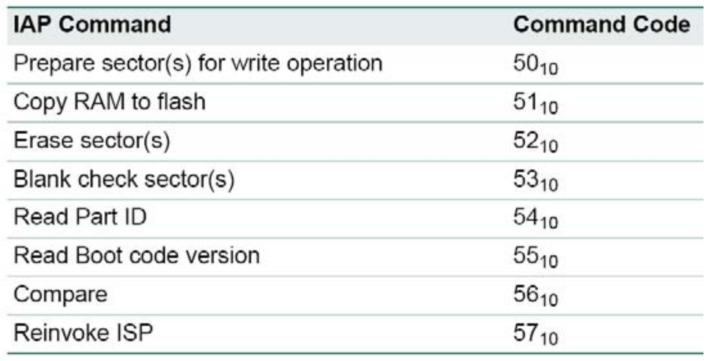


Fig 2. IAP parameter passing

The IAP commands and codes are listed in Fig 3.



IAP Command	Command Code
Prepare sector(s) for write operation	50_{10}
Copy RAM to flash	51_{10}
Erase sector(s)	52_{10}
Blank check sector(s)	53_{10}
Read Part ID	54_{10}
Read Boot code version	55_{10}
Compare	56_{10}
Reinvoke ISP	57_{10}

Fig 3. IAP command summary

For detailed information, please refer to the LPC2000 user manual.

2.2.4 Using IAP

Fig 4 shows the necessary steps to perform the flash programming using IAP.

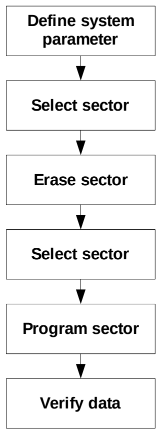


Fig 4. Flash programming steps

2.2.4.1 Define system parameters

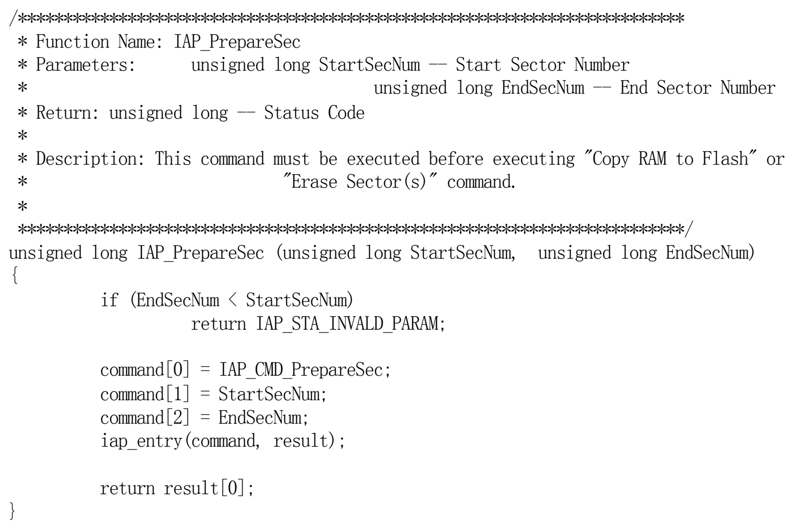
Some constants such as system clock, IAP entry point, input and output buffers should be defined before an IAP call.

```
#define IAP_CLK Fcclk

#define IAP_LOCATION 0x7FFFFFF1
#define iap_entry(a, b)     ((void (*)())(IAP_LOCATION))(a, b)

unsigned long command[5] = {0,0,0,0,0};
unsigned long result[3]= {0,0,0};
```

Fig 5. Define system parameter

2.2.4.2 Select sector

The sectors have to be selected before any erase or programming operation. More than one sector can be selected.

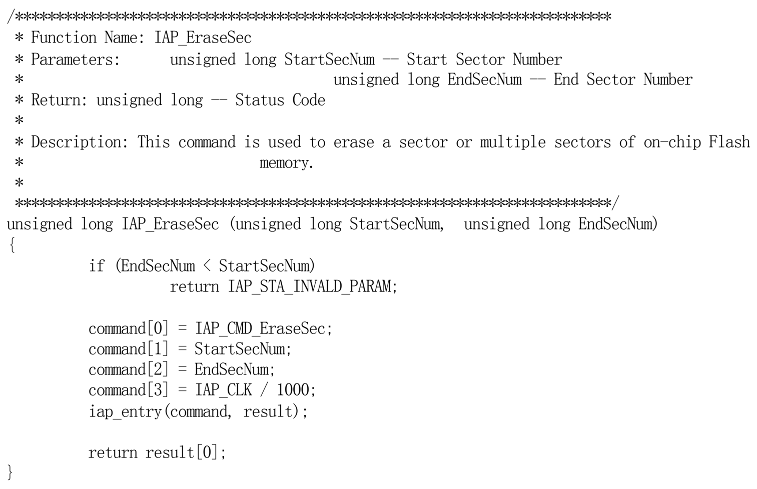


```
/*****************************************************************************
 * Function Name: IAP_PrepareSec
 * Parameters:      unsigned long StartSecNum -- Start Sector Number
 *                                  unsigned long EndSecNum -- End Sector Number
 * Return: unsigned long -- Status Code
 *
 * Description: This command must be executed before executing "Copy RAM to Flash" or
 *                          "Erase Sector(s)" command.
 *
 *****************************************************************************/
unsigned long IAP_PrepareSec (unsigned long StartSecNum,  unsigned long EndSecNum)
{
         if (EndSecNum < StartSecNum)
                  return IAP_STA_INVALD_PARAM;

         command[0] = IAP_CMD_PrepareSec;
         command[1] = StartSecNum;
         command[2] = EndSecNum;
         iap_entry(command, result);

         return result[0];
}
```

Fig 6. Select sector

2.2.4.3 Erase sector

Like other Flash implementations, the LPC2000 on-chip should be erased before programming. However, if the target sector is already erased, it is not necessary to erase the sector again. More than one sector can be erased at a time.



```
/*****************************************************************************
 * Function Name: IAP_EraseSec
 * Parameters:      unsigned long StartSecNum -- Start Sector Number
 *                                  unsigned long EndSecNum -- End Sector Number
 * Return: unsigned long -- Status Code
 *
 * Description: This command is used to erase a sector or multiple sectors of on-chip Flash
 *                          memory.
 *
 *****************************************************************************/
unsigned long IAP_EraseSec (unsigned long StartSecNum,  unsigned long EndSecNum)
{
         if (EndSecNum < StartSecNum)
                  return IAP_STA_INVALD_PARAM;

         command[0] = IAP_CMD_EraseSec;
         command[1] = StartSecNum;
         command[2] = EndSecNum;
         command[3] = IAP_CLK / 1000;
         iap_entry(command, result);

         return result[0];
}
```

Fig 7. Erase sector

2.2.4.4 Program sector

During this stage, the data will be programmed from on-chip RAM to Flash.

Note:

1. **Data can only be programmed from on-chip SRAM to on-chip Flash.**
2. **The address in on-chip Flash should be on a 256 byte boundary.**

3. **On-chip RAM should be located on the local bus which means neither USB SRAM nor Ethernet SRAM can be used.**
4. **Number of programmed bytes each time should be 256, 512, 1024 or 4096.**

```
/******************************************************************************
 * Function Name: IAP_CopyRAMToFlash
 * Parameters:      unsigned long dst -- Destination Flash address, should be a 256 byte boundary.
 *                                    unsigned long src -- Source RAM address, should be a word boundary
 *                                    unsigned long number -- 256 | 512 |1024 |4096
 * Return: unsigned long -- Status Code
 *
 * Description: This command is used to program the flash memory.
 *
 ******************************************************************************/
unsigned long IAP_CopyRAMToFlash (unsigned long dst,  unsigned long src,
          unsigned long number)
{
          command[0] = IAP_CMD_CopyRAMToFlash;
          command[1] = dst;
          command[2] = src;
          command[3] = number;
          command[4] = IAP_CLK / 1000;  // Fcclk in KHz
          iap_entry(command, result);

          return result[0];
}
```

Fig 8. Program sector

2.2.4.5 Verify data

With this function, user does not have to write their own code to compare the data in RAM and Flash.

Note: Source address, destination address and number of bytes should be a word boundary or a multiple of 4.

```
/******************************************************************************
 * Function Name: IAP_Compare
 * Parameters:      unsigned long dst -- Destination Flash address
 *                                    unsigned long src -- Source RAM address
 *                                    unsigned long number -- Should be in mutilple of 4
 * Return: unsigned long -- Status Code
 *
 * Description: This command is used to compary the memory contents at two locations.
 *
 * NOTE:  Compary result may not be correct when source or destination address contains
 *                  any of the first 64 bytes starting from address zero. First 64 bytes can be re-mapped
 *                  to RAM.
 *
 ******************************************************************************/
unsigned long IAP_Compare (unsigned long dst,  unsigned long src,
          unsigned long number, unsigned long *offset)
{
          command[0] = IAP_CMD_Compare;
          command[1] = dst;
          command[2] = src;
          command[3] = number;
          iap_entry(command, result);

          if (result[0] == IAP_STA_COMPARE_ERROR)
                    *offset = result[1];

          return result[0];
}
```

Fig 9. Verify data

2.2.4.6 Interrupts during IAP

The on-chip flash memory is not accessible during erase/write operations. When the user application code starts executing, the interrupt vectors from the user flash area are active.

The user should either disable interrupts, or ensure that the user's interrupt vectors reside and are active in RAM, before making a flash erase/write IAP call. The IAP code does not use or disable interrupts.

```
__swi(0x00) void SwiHandle1(int Handle);

#define IRQDisable() SwiHandle1(0)
#define IRQEnable() SwiHandle1(1)

/*
* prepare and erase the sectors with index from "start" to "end"
* if successful, return TRUE, elsewise FALSE
*/
BYTE IAP_PrepareErase_Sector(DWORD start, DWORD end)
{
          DWORD IAP_status;
          BYTE result = FALSE;

          IRQDisable();
          IAP_status = IAP_PrepareSec(start, end);
          if (IAP_status == IAP_STA_CMD_SUCCESS)
          {
                    IAP_status = IAP_EraseSec(start, end);
                    if (IAP_status == IAP_STA_CMD_SUCCESS)
                    {
                              result = TRUE;
                    }
          }
          IRQEnable();

          return result;
}
```

Fig 10. Interrupt handle during IAP

2.2.4.7 RAM used by IAP command handler

Flash programming commands use the top 32 bytes of on-chip RAM. The maximum stack usage in the user allocated stack space is 128 bytes (for LPC2300) and grows downwards.

2.2.4.8 Running the user application

After loading the application, the user can execute the application by modifying the PC register by pointing it to the starting address of the application code.

```
#define AP_ADDR 0x8000                    // where the user app located
typedef void (*FP)(void);

/*
* run the user application from pre-specified address
*/
void Run_Application()
{
   FP fp;

   fp = (FP)AP_ADDR;
   (*fp)();
}
```

Fig 11. run the user application

3. Secondary bootloader and user application

3.1 Secondary bootloader

After power on or reset, the secondary bootloader will always run and display a menu on a PC terminal window (e.g., Teraterm). With this menu, the user can erase or program the internal flash. Depending on the bootloader selected the application code can be programmed using one of the four different interfaces (UART, SD Card, I^{2}C, or CAN).

The secondary bootloader resides in sectors starting at 0 and must not overlap the user application code.

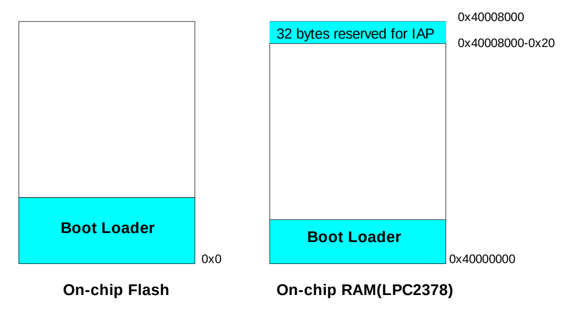


Fig 12. Bootloader memory map

3.1.1 Bootloader Features and Constraints

The UART, SD/MMC, I^{2}C, and CAN bootloaders function as a reference to allow the user to customize it to their particular needs. The bootloaders at their current state have the following design constraints and features:

- The bootloader's default user application code starting address is 0x10000.
- The bootloader will always wait for user input before continuing on into the application code.
- The bootloader has no knowledge whether user code is present or where it is located at in flash.
- The SD/MMC bootloader requires a SD Card to be inserted for proper operation.
- The SD/MMC bootloader allows the user to select which binary file should be flashed onto the microcontroller.
- If the application code utilizes interrupts, it needs to maintain its own interrupt vector table in SRAM, see section 3.2.2.
- The user application's starting address can be changed at compile time; however, it must match the address at which the code is to be programmed using the bootloader.

3.2 User application

3.2.1 Memory map

The user application is located in different sectors along with the secondary bootloader and consumes the on-chip RAM starting from 0x40000000.

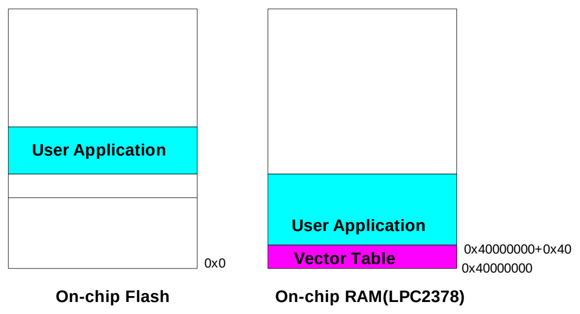
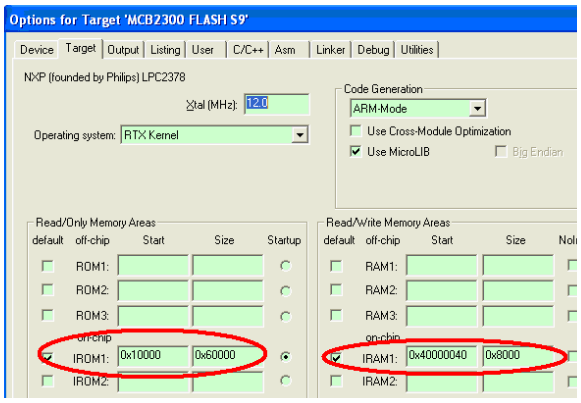


Fig 13. User application memory map

If the user application utilizes interrupts, the first 64 bytes (0x40) at the bottom of the on-chip RAM should be used as the interrupt vector table. Refer to section 3.2.2.

Fig 14 shows the memory configuration for the user application in Keil's uVision 3 IDE. The on-chip RAM is from 0x40000040 instead of 0x40000000.


Options for Target 'MCB2300 FLASH S9'

Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities

NXP (founded by Philips) LPC2378

Xtal (MHz): 12.0

Operating system: RTX Kernel

Code Generation: ARM-Mode

[ ] Use Cross-Module Optimization
[✓] Use MicroLIB
[ ] Big Endian

Read/Only Memory Areas

default		Start	Size	Startup
[ ]	ROM1:			○
[ ]	ROM2:			○
[ ]	ROM3:			○
on-chip				
[✓]	IROM1:	0x10000	0x60000	●
[ ]	IROM2:			○

Read/Write Memory Areas

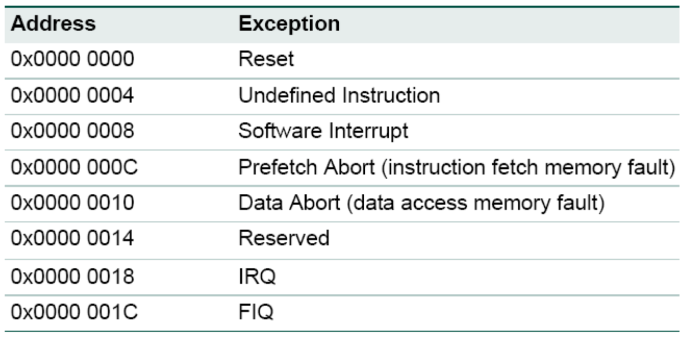
default		Start	Size	NoI
[ ]	RAM1:			[ ]
[ ]	RAM2:			[ ]
[ ]	RAM3:			[ ]
on-chip				
[✓]	IRAM1:	0x40000040	0x8000	[ ]
[ ]	IRAM2:			[ ]

Fig 14. Memory configuration for User application in Keil uVision 3

By changing the "IROM1 Start address" in uVision, the user can change the starting location at which the application will be stored. Note that it is important not to use a starting address that is located in a sector that contains the secondary bootloader.

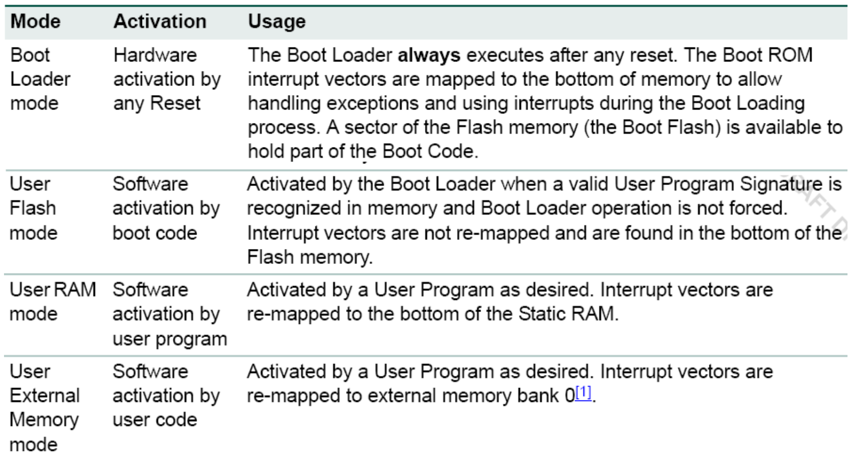
3.2.2 Interrupt vector table re-mapping

Because of the location of the interrupt vectors on the ARM7 processor (at addresses 0x0000 0000 through 0x0000 001C, as shown in Fig 15 below), a small portion of the Boot ROM and SRAM spaces need to be re-mapped in order to allow alternative uses of interrupts in the different operating modes described in Fig 16 below.



Address	Exception
0x0000 0000	Reset
0x0000 0004	Undefined Instruction
0x0000 0008	Software Interrupt
0x0000 000C	Prefetch Abort (instruction fetch memory fault)
0x0000 0010	Data Abort (data access memory fault)
0x0000 0014	Reserved
0x0000 0018	IRQ
0x0000 001C	FIQ

Fig 15. ARM exception vector locations



Mode	Activation	Usage
Boot Loader mode	Hardware activation by any Reset	The Boot Loader **always** executes after any reset. The Boot ROM interrupt vectors are mapped to the bottom of memory to allow handling exceptions and using interrupts during the Boot Loading process. A sector of the Flash memory (the Boot Flash) is available to hold part of the Boot Code.
User Flash mode	Software activation by boot code	Activated by the Boot Loader when a valid User Program Signature is recognized in memory and Boot Loader operation is not forced. Interrupt vectors are not re-mapped and are found in the bottom of the Flash memory.
User RAM mode	Software activation by user program	Activated by a User Program as desired. Interrupt vectors are re-mapped to the bottom of the Static RAM.
User External Memory mode	Software activation by user code	Activated by a User Program as desired. Interrupt vectors are re-mapped to external memory bank 0[1].

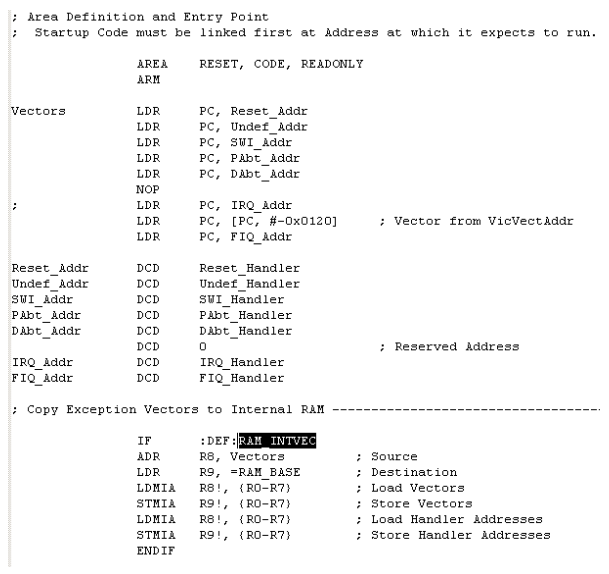
Fig 16. LPC2300 Memory mapping modes

The portion of memory that is re-mapped allows for interrupt processing in different modes and includes the interrupt vector area (32 bytes) as well as an additional 32 bytes (a total of 64 bytes) to facilitate branching to interrupt handlers at distant physical addresses. The remapped code locations overlay addresses 0x0000 0000 through 0x0000 003F.

The interrupt vectors of the secondary bootloader occupies the physical address from 0x0000 0000 through 0x0000 003F in on-chip flash, thus the interrupt vectors for the user application has to be re-mapped to the bottom of the on-chip SRAM using "User RAM Mode".

Before entering the main application, the user should copy the 64 bytes (32 bytes interrupt vector and 32 bytes additional bytes) into the bottom of the on-chip RAM and set the mapping mode to "User RAM Mode".

Fig 17 shows how to copy the 64 bytes of interrupt vectors from on-chip flash to the bottom of on-chip RAM.



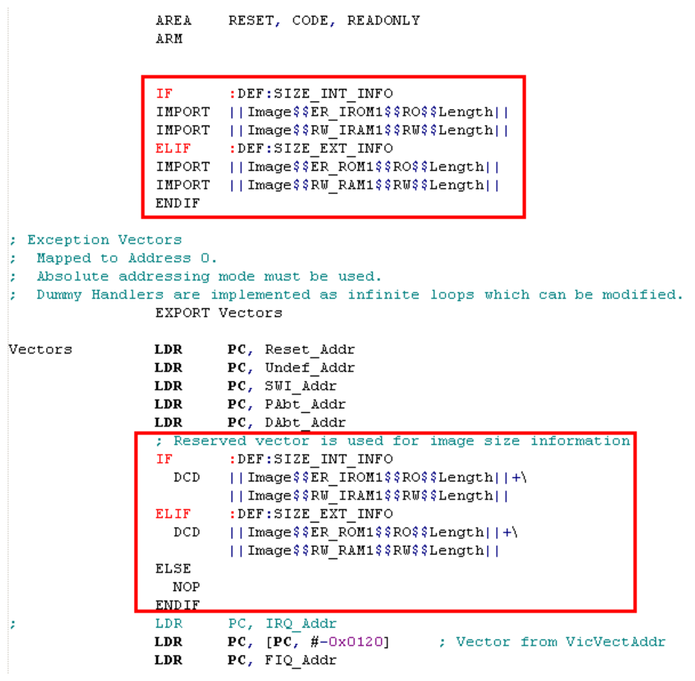
```
; Area Definition and Entry Point
;  Startup Code must be linked first at Address at which it expects to run.

                AREA    RESET, CODE, READONLY
                ARM

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0120]     ; Vector from VicVectAddr
                LDR     PC, FIQ_Addr

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0                      ; Reserved Address
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler

; Copy Exception Vectors to Internal RAM ----------------------------------

                IF      :DEF:RAM_INTVEC
                ADR     R8, Vectors         ; Source
                LDR     R9, =RAM_BASE       ; Destination
                LDMIA   R8!, {R0-R7}        ; Load Vectors
                STMIA   R9!, {R0-R7}        ; Store Vectors
                LDMIA   R8!, {R0-R7}        ; Load Handler Addresses
                STMIA   R9!, {R0-R7}        ; Store Handler Addresses
                ENDIF
```

Fig 17. Vector table re-mapping in user application (file lpc2300.s)

3.2.3 User application size

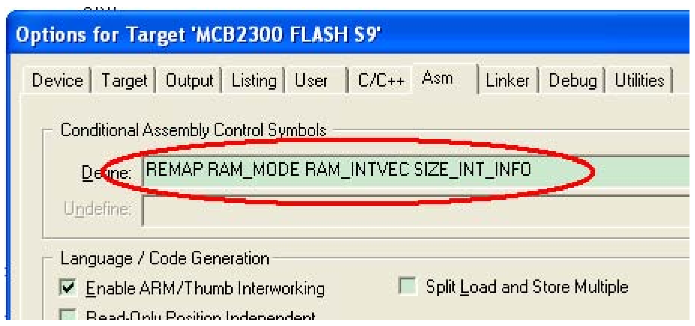
One key parameter needed to update the user application is the size of the application itself. When loading the user application from the SD/MMC or UART interface, the end and the size of the binary file can be easily determined.

CAN and I^{2}C only handles the physical data transfers from the interface onto on-chip memory. Without an additional protocol, loading the application via CAN or I^{2}C requires prior knowledge of the application size before it can be programmed into flash. The following figures show how to record the image size.



```
                AREA    RESET, CODE, READONLY
                ARM

                IF      :DEF:SIZE_INT_INFO
                IMPORT  ||Image$$ER_IROM1$$RO$$Length||
                IMPORT  ||Image$$RW_IRAM1$$RW$$Length||
                ELIF    :DEF:SIZE_EXT_INFO
                IMPORT  ||Image$$ER_ROM1$$RO$$Length||
                IMPORT  ||Image$$RW_RAM1$$RW$$Length||
                ENDIF

; Exception Vectors
;  Mapped to Address 0.
;  Absolute addressing mode must be used.
;  Dummy Handlers are implemented as infinite loops which can be modified.
                EXPORT Vectors

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                ; Reserved vector is used for image size information
                IF      :DEF:SIZE_INT_INFO
                  DCD   ||Image$$ER_IROM1$$RO$$Length||+\
                        ||Image$$RW_IRAM1$$RW$$Length||
                ELIF    :DEF:SIZE_EXT_INFO
                  DCD   ||Image$$ER_ROM1$$RO$$Length||+\
                        ||Image$$RW_RAM1$$RW$$Length||
                ELSE
                  NOP
                ENDIF
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0120]     ; Vector from VicVectAddr
                LDR     PC, FIQ_Addr
```

Fig 18. Record user application image size in lpc2300.s

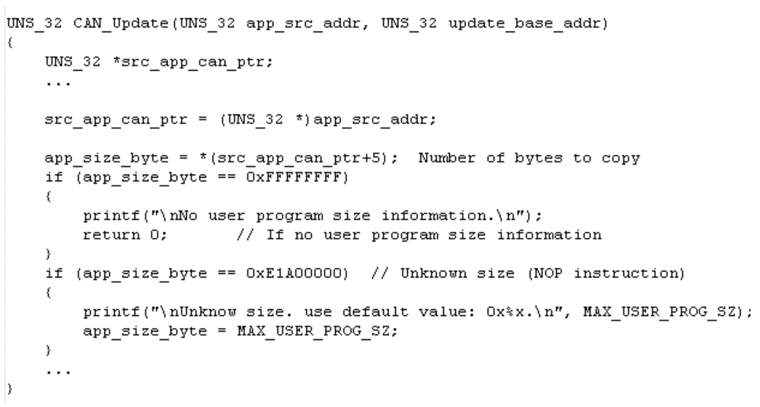

Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Conditional Assembly Control Symbols
Undefine:
Language / Code Generation
Split Load and Store Multiple
Read-Only Position Independent

Options for Target 'MCB2300 FLASH S9'

Define: REMAP RAM_MODE RAM_INTVEC SIZE_INT_INFO

[✓] Enable ARM/Thumb Interworking

Fig 19. Control symbols definition for lpc2300.s in user application



```
UNS_32 CAN_Update(UNS_32 app_src_addr, UNS_32 update_base_addr)
{
    UNS_32 *src_app_can_ptr;
    ...

    src_app_can_ptr = (UNS_32 *)app_src_addr;

    app_size_byte = *(src_app_can_ptr+5);  Number of bytes to copy
    if (app_size_byte == 0xFFFFFFFF)
    {
        printf("\nNo user program size information.\n");
        return 0;       // If no user program size information
    }
    if (app_size_byte == 0xE1A00000)  // Unknown size (NOP instruction)
    {
        printf("\nUnknow size. use default value: 0x%x.\n", MAX_USER_PROG_SZ);
        app_size_byte = MAX_USER_PROG_SZ;
    }
    ...
}
```

Fig 20. Retrieve user application size in secondary bootloader

4. Demo description

4.1 Demo setup

4.1.1 Hardware setup

This demo is tested on the KEIL MCB2300 evaluation board (version 4.7). For more information about MCB2300, please refer to: http://www.nxp.com/redirect/keil.com/mcb2300/.

No other special hardware is needed except a RS-232 cable, Standard USB (A-B) cable (for power), and an EEPROM with I^2C interface (if using the I^2C bootloader). The RS-232 cable is for the connection between the PC and MCB2300 on COM1. The ULINK-ME JTAG module can be used to program the bootloader onto the board. Optionally, the FlashMagic tool can be used to program the bootloader using the RS-232 cable connected onto UART0 (COM0).

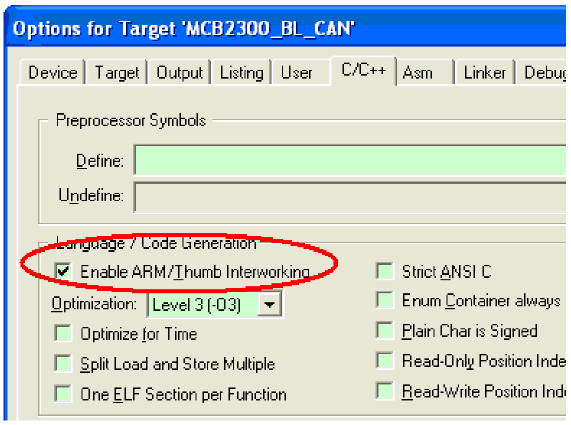
UART1 (COM1) Setting: 115200 baud, 8N1

The FlashMagic tool is available for free at: http://www.nxp.com/redirect/flashmagictool.com/

4.1.2 Software setup

Keil uVision3 and Flashmagic are used to edit, compile, link, debug and download the code. Tera term (or other tools) is used for serial communication between PC terminal and MCB2300 and configured at 115200 baud, 8-bits, no parity, 1 stop bit, XON/XOFF.

Because the IAP routines are run in thumb code, the bootloaders need to have ARM/Thumb interworking enabled. Fig 21 shows the configuration in Keil uVision3 IDE.



[✓] Enable ARM/Thumb Interworking
Optimization: Level 3 (-O3)
[ ] Optimize for Time
[ ] Split Load and Store Multiple
[ ] One ELF Section per Function
[ ] Strict ANSI C
[ ] Enum Container always
[ ] Plain Char is Signed
[ ] Read-Only Position Inde
[ ] Read-Write Position Ind

Fig 21. Enable ARM/Thumb interworking in KEIL uVsion3 IDE

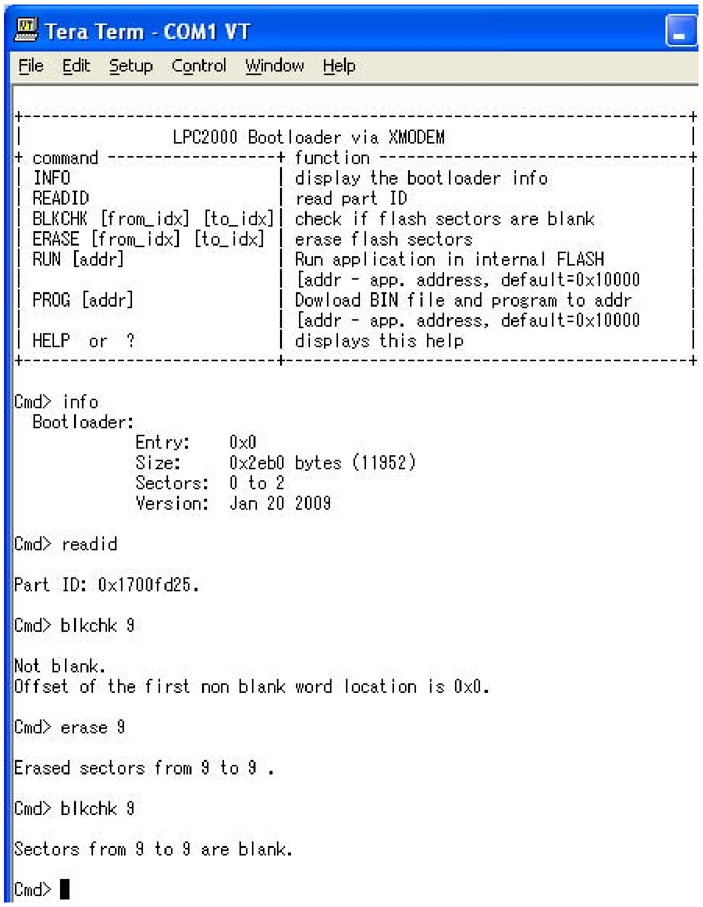
4.2 Start the demo

The reason for having four different bootloaders instead of just one bootloader is to minimize the size of the secondary bootloader itself. The smaller the bootloader, the more flash is available for the user application.

At power on or reset, a menu will be displayed in the Tera Term if properly configured. Most of the functions listed in menu are same. With these functions, the user can erase or program the flash with the application code using a specific interface.

Throughout this section “<starting address>” refers to the actual address which should be entered into the serial terminal.

For example, if you want to use address 0x10000 and the command is shown as “prog <starting address>”, then the user should type: “prog 0x10000”.



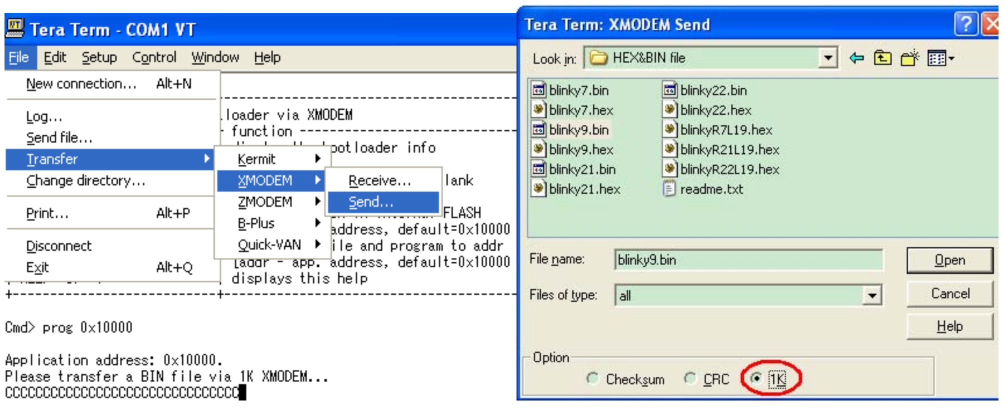
```
+-------------------------------------------------------------------------+
|                  LPC2000 Bootloader via XMODEM                          |
+ command ------------------+ function ----------------------------------+
| INFO                      | display the bootloader info                 |
| READID                    | read part ID                                |
| BLKCHK [from_idx] [to_idx]| check if flash sectors are blank            |
| ERASE [from_idx] [to_idx] | erase flash sectors                         |
| RUN [addr]                | Run application in internal FLASH           |
|                           | [addr - app. address, default=0x10000       |
| PROG [addr]               | Dowload BIN file and program to addr        |
|                           | [addr - app. address, default=0x10000       |
| HELP  or  ?               | displays this help                          |
+---------------------------+---------------------------------------------+

Cmd> info
  Bootloader:
              Entry:    0x0
              Size:     0x2eb0 bytes (11952)
              Sectors:  0 to 2
              Version:  Jan 20 2009

Cmd> readid

Part ID: 0x1700fd25.

Cmd> blkchk 9

Not blank.
Offset of the first non blank word location is 0x0.

Cmd> erase 9

Erased sectors from 9 to 9 .

Cmd> blkchk 9

Sectors from 9 to 9 are blank.

Cmd>
```

Fig 22. Secondary bootloader menu

4.2.1 Code update via UART

Type command “prog <starting address>” and then send a binary file using 1K XMODEM protocol. See Fig 23.
Then input “run <starting address>”, which should jump to the application.

Note: The “address” should be the same with the entry point of the binary file.



```
loader via XMODEM
- function ------------------------------
                        ootloader info
                                          lank
                                          FLASH
                        address, default=0x10000
                        ile and program to addr
 [addr - app. address, default=0x10000
 displays this help
+--------------------------+-----------------------

Cmd> prog 0x10000

Application address: 0x10000.
Please transfer a BIN file via 1K XMODEM...
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Look in: HEX&BIN file

blinky7.bin, blinky7.hex, blinky9.bin, blinky9.hex, blinky21.bin, blinky21.hex, blinky22.bin, blinky22.hex, blinkyR7L19.hex, blinkyR21L19.hex, blinkyR22L19.hex, readme.txt

File name: blinky9.bin

Files of type: all

Option: Checksum CRC [✓ 1K]

Fig 23. Update via 1K XMODEM

4.2.2 Code update via CAN

To simplify the test of the CAN interface and protocol, CAN1 will act as transmitter and CAN2 as receiver on the MCB2300 board. Since CAN transceivers are already integrated in MCB2300, the user only needs to connect pins 2 and 7 directly together to both CANs as shown in Fig 24.

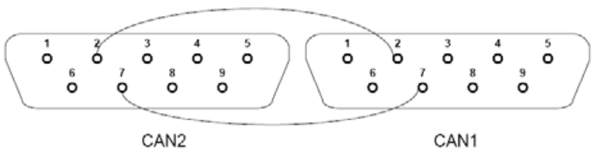
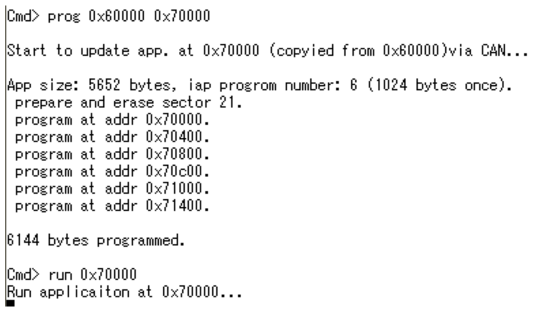


Fig 24. CAN port connection

The user application is linked at address of 0x70000 but is programmed at address 0x60000. Afterwards, type “prog 0x60000 0x70000”. This causes the user application code to be copied from address 0x60000 onto address 0x70000 via the CAN interface.



```
Cmd> prog 0x60000 0x70000

Start to update app. at 0x70000 (copyied from 0x60000)via CAN...

App size: 5652 bytes, iap progrom number: 6 (1024 bytes once).
 prepare and erase sector 21.
 program at addr 0x70000.
 program at addr 0x70400.
 program at addr 0x70800.
 program at addr 0x70c00.
 program at addr 0x71000.
 program at addr 0x71400.

6144 bytes programmed.

Cmd> run 0x70000
Run applicaiton at 0x70000...
```

Fig 25. Code update via CAN

4.2.3 Code update via I2C

An additional EEPROM (24LC64) can be used on the MCB2300 board to test the I2C interface and protocol.

As shown in Fig 26, the 24LC64 works as an I2C slave with address of 0XA0.

Note: The SDA and SCL bus requires a pull-up resistor to V_{cc} (typical 10 kΩ for 100 kHz, 2 kΩ for 400 kHz).

The application code needs to be already programmed into EEPROM.

By typing command “prog <starting address>”, the user application code will be copied from EEPROM via I2C interface onto internal flash at the address specified in the “prog” command.

Then input “run <starting address>”, which should jump to the application.

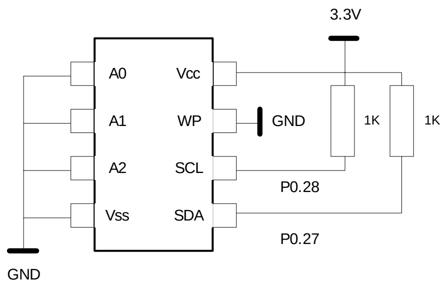
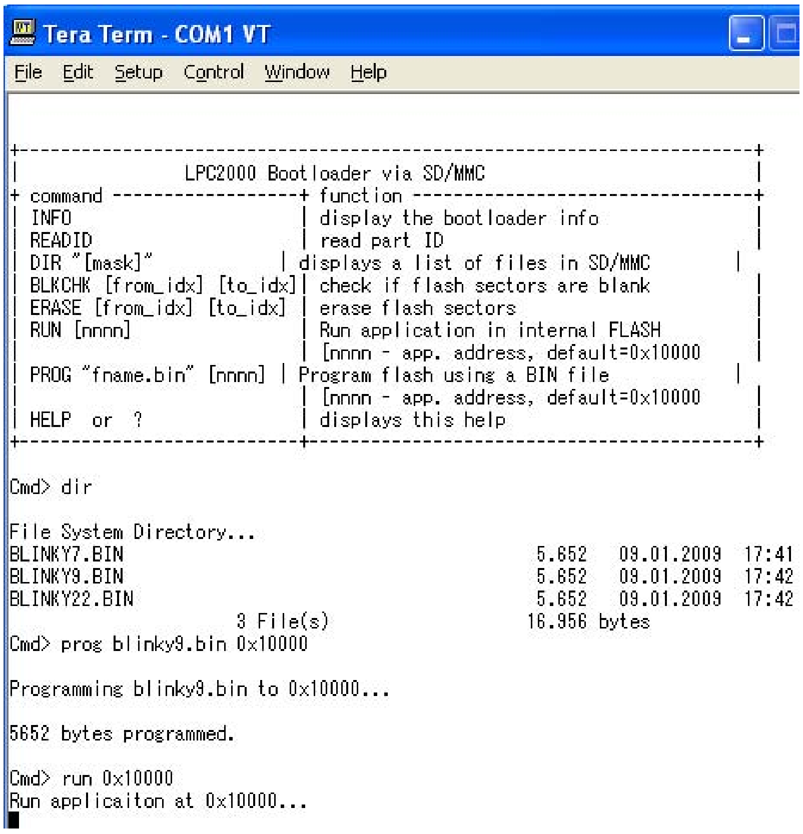


Fig 26. 24LC64 circuit

4.2.4 Code update via SD/MMC

The SD/MMC bootloader utilizes a FAT file system so that the user can just copy the binary application code onto a SD Card from a workstation and then flash the application from the SD/MMC interface.

Using this bootloader, the user can list separate files located on the SD card and flash the microcontroller with the name specified. See Fig 27.



```
+-----------------------------------------------------------------------+
|                LPC2000 Bootloader via SD/MMC                          |
+ command ------------------+ function -------------------------------+
| INFO                      | display the bootloader info             |
| READID                    | read part ID                            |
| DIR "[mask]"             | displays a list of files in SD/MMC      |
| BLKCHK [from_idx] [to_idx]| check if flash sectors are blank        |
| ERASE [from_idx] [to_idx] | erase flash sectors                     |
| RUN [nnnn]                | Run application in internal FLASH       |
|                           | [nnnn - app. address, default=0x10000   |
| PROG "fname.bin" [nnnn] | Program flash using a BIN file          |
|                           | [nnnn - app. address, default=0x10000   |
| HELP  or  ?               | displays this help                      |
+---------------------------+-------------------------------------------+

Cmd> dir

File System Directory...
BLINKY7.BIN                                       5.652   09.01.2009  17:41
BLINKY9.BIN                                       5.652   09.01.2009  17:42
BLINKY22.BIN                                      5.652   09.01.2009  17:42
                      3 File(s)                  16.956 bytes
Cmd> prog blinky9.bin 0x10000

Programming blinky9.bin to 0x10000...

5652 bytes programmed.

Cmd> run 0x10000
Run applicaiton at 0x10000...
```

Fig 27. Code update via SD/MMC

5. Legal information

5.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

5.2 Disclaimers

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of a NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is for the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from national authorities.

5.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

6. Contents

Date of release: 26 May 2009

Document identifier: AN10835_1

founded by PHILIPS