AP08058

XC800

Loading Code to XRAM

8bit

Microcontrollers

Never stop thinking

Edition 2007-02

Published by
Infineon Technologies AG
81726 München, Germany

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.

XC800

Revision History: V1.0, 2007-02

Previous Version(s):
none

Page	Subjects (major changes since last revision)
V1.0	Initial Release

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

Table Of Contence

1 Introduction

XC800 provides the ability to execute the code from the XRAM.

This document will show the ways to load the code to XRAM and execute it from there.

In this document, all of the examples are provided for the KEIL C51 Compiler. The examples for other compiler will be supported in the future.

There are different ways to do it and this document will provide 3 ways to copy the code to XRAM and execute it from there.

Also it shows, how to setup the simulator (KEIL) for test purposes

In this document, user can see how the following is done:

1. Copy the code “by hand” and execute it via an LCALL using a function pointer and the XBYTE macro (absacc.h)
2. Using the SROM.H and memcpy(), provided by KEIL
   once with a program segment.
   once with a user class
3. Adjust the Linker / Locator LX51
4. Setup and using of the simulator

2 Example Code

Assuming the following small function code:

```
unsigned char counter;
for (counter=0; counter < 0xFF; counter++);
P3_DATA++
```

Resulting in the following hexadecimal code

```
0xE4,           // CLR     A
0xF5,0x08,      // MOV    counter,A
// BACKWARD:
0xE5,0x08,      // MOV    A,counter
0xC3,           // CLR     C
0x94,0xFF,      // SUBB   A,#0FFH
0x50,0x04,      // JNC     FORWARD
0x05,0x08,      // INC     counter
0x80,0xF5,      // SJMP   BACKWARD
// FORWARD:
0x05,0xB0,      // INC     P3_DATA
0x22,           // RET
```

This will be executed from P-Flash, D-Flash and XRAM

3 Simple Method

The easiest way is, to load the hardcoded function into XRAM manually.
For example by using the XBYTE[] macro coming with absacc.h file.

```
void CopySmallFuncHardCoded2XRAM (void)
//loads into XRAM starting from fix address 0xF100
{
     unsigned char cnt=0;
     while(SmallFuncHardCoded[cnt] != 0x00)
     {
          XBYTE[0xF100+cnt]= SmallFuncHardCoded[cnt];
          cnt++;
     }
}
```

Than a function pointer like ...
((void (code *) (void)) 0xF100) ();
... can be used to generate a “LCALL” to the XRAM for the execution.

This method is ugly and inflexibly, but possible for short sequences / tests of functions.

4 Advanced Method

A more flexible and elegant way is to use the SROM.h, which provides the possibility to copy a certain function / code part without the knowledge of hexadecimal code and / or length.

Be aware that there is still a routine needed, that copies to XRAM,
before it can be executed from there.

4.1 User Class

An own user class should be created in a separate file.

```
#include “main.h”
#pragma USERCLASS (CODE = XRAM) // XRAM is a freely choosable name

void STDFLASH2XRAM_Function (void)// The function name
{
      FUNCBODY();      // A macro that contains the loop and
                       // increment of P3 with P3_DATA++;
}
```

Inside the copy function, use

```
SROM_MC (CODE_XRAM)
```

to create the external xdata types and the access macros.

The copy routine (using memcpy) looks like this:

```
void CopyUserClassTo0xF000 (void)
{
      // copy flash function from flash to RAM
      memcpy (SROM_MC_TRG(CODE_XRAM), SROM_MC_SRC(CODE_XRAM),
            SROM_MC_LEN(CODE_XRAM));
}
```

where the SROM_MC_XXC macros are created by SROM_MC(CODE_XRAM)

*Note: The argument CODE_XRAM comes from our USERCLASS directive (CODE = XRAM), that is expanded to **CODE_XRAM** (look into MAP-file).*

```
#include “main.h”
#pragma USERCLASS (CODE = XRAM)// XRAM is a freely choosable name

void STDFLASH2XRAM_Function (void)// The function name
{
     FUNCBODY();     // A macro that contains the loop and P3_DATA
                     // increment P3_DATA++;
}

void CopyUserClassTo0xF000 (void)    // copy flash function from
{                                    // flash to RAM
     memcpy (SROM_MC_TRG(CODE_XRAM),
           SROM_MC_SRC(CODE_XRAM),
           SROM_MC_LEN(CODE_XRAM));
```

That is all (and very easy) for user classes.

4.2 Program Segment

The scheme is the same for the program segment, but more care must be taken for the naming convention.

Assuming there is a file called stpflash2xram.c with a function inside:

```
#include “main.h”
void STPFLASH2XRAM_Function (void)
{
     FUNCBODY();
}
```

then the copy routine will looks like:

```
SROM_PS(STPFLASH2XRAM_Function_STPFLASH2XRAM)
void CopyProgramSegmentTo0xF000 (void) //Copy flash function from
{                                      // FALSH to XRAM
     memcpy(SROM_PS_TRG(STPFLASH2XRAM_Function_STPFLASH2XRAM),
          (SROM_PS_SRC(STPFLASH2XRAM_Function_STPFLASH2XRAM),
          (SROM_PS_LEN(STPFLASH2XRAM_Function_STPFLASH2XRAM));
}
```

That would be expanded to ?PR?STPFLASH2XRAM_FUNCTION?STPFLASH2XRAM

Keep that in mind, to avoid “unresolved external symbol ... ” linker error messages.

Conclusion for program segments:

```
#include “main.h”                  // in the file stpflash2xram.c
void STPFLASH2XRAM_Function (void)
{
      FUNCBODY();
}
```

is expanded to ?PR?STPFLASH2XRAM_FUNCTION?STPFLASH2XRAM

therefore, use :

SROM_PS(STPFLASH2XRAM_Function_STPFLASH2XRAM)

and access in the copy routine with:

```
void CopyProgramSegmentTo0xF000 (void) //Copy Flash function from
{                                      // FLASH to XRAM
      memcpy(SROM_PS_TRG(STPFLASH2XRAM_Function_STPFLASH2XRAM),
            (SROM_PS_SRC(STPFLASH2XRAM_Function_STPFLASH2XRAM),
            (SROM_PS_LEN(STPFLASH2XRAM_Function_STPFLASH2XRAM));
}
```

Remember: FUNCTIONNAME_MODULENAME

SROM_PS_xxx => **Program Segment**

SROM_MC_xxx => **Memory Class**

5 Linker / Locator Adjustment

Now the linker / locator must be adjusted.

Note that **LX51** must be used for this purpose, since the BL51 is not able to do.

Assuming the following storage and execution addresses are required:

	Storage	**Execution**
Hard coded block	anywhere	0xF100
PFLASH	0x2000	0x2000
DFLASH	0xAA00	0xAA00
PFLASH2XRAM	0x4000 ... 0x4xxx	0xF000
DFLASH2XRAM	0x4CB0 ... 0x4xxx	0xF000

That also means that user can re-use the XRAM from 0xF000 to run more than one executable code (not at the same time).

5.1 Select LX51

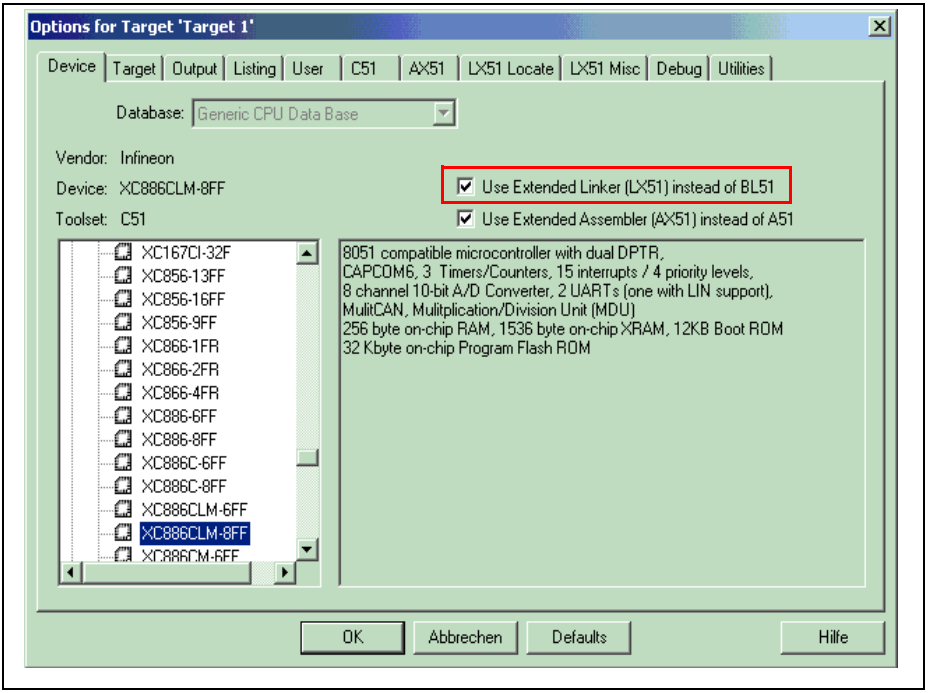


Figure 1 Target Option Window

5.2 Adjustment for the LX51 locator

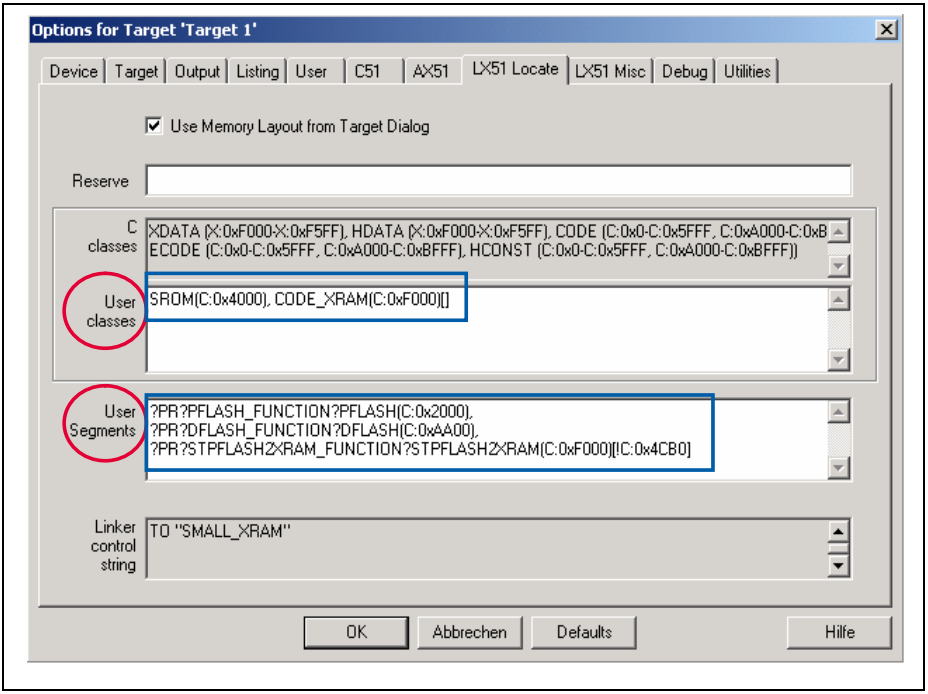
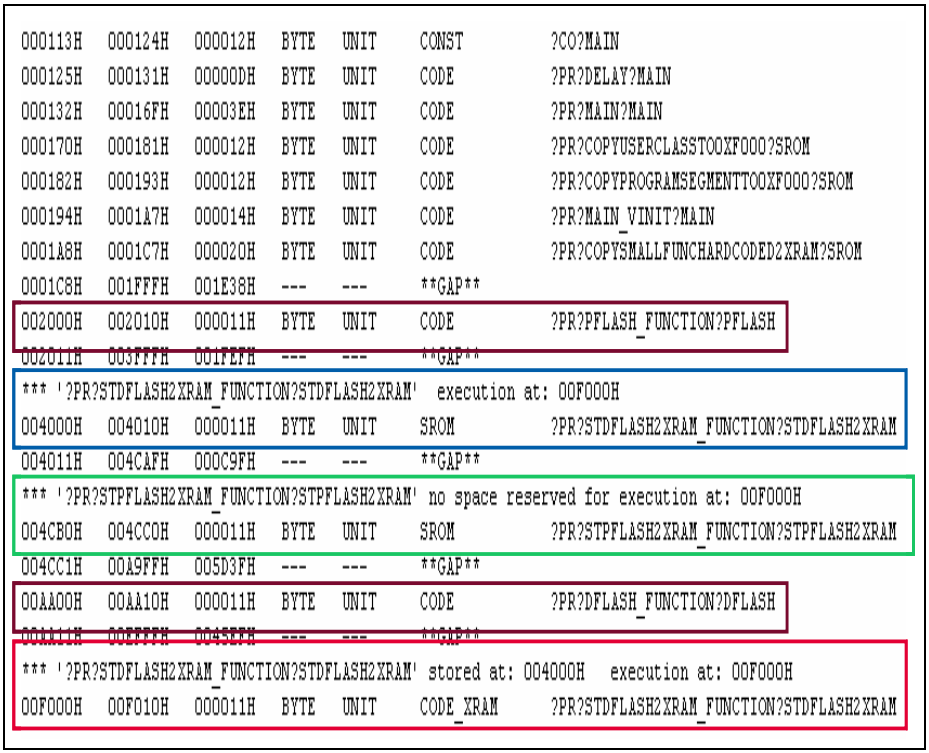


Figure 2 Linker / Locator LX51 - Settings



```
000113H   000124H   000012H   BYTE   UNIT     CONST          ?CO?MAIN
000125H   000131H   00000DH   BYTE   UNIT     CODE           ?PR?DELAY?MAIN
000132H   00016FH   00003EH   BYTE   UNIT     CODE           ?PR?MAIN?MAIN
000170H   000181H   000012H   BYTE   UNIT     CODE           ?PR?COPYUSERCLASSTOOXF000?SROM
000182H   000193H   000012H   BYTE   UNIT     CODE           ?PR?COPYPROGRAMSEGMENTTOOXF000?SROM
000194H   0001A7H   000014H   BYTE   UNIT     CODE           ?PR?MAIN_VINIT?MAIN
0001A8H   0001C7H   000020H   BYTE   UNIT     CODE           ?PR?COPYSMALLFUNCHARDCODED2XRAM?SROM
0001C8H   001FFFH   001E38H   ---    ---      **GAP**
002000H   002010H   000011H   BYTE   UNIT     CODE           ?PR?PFLASH_FUNCTION?PFLASH
002011H   003FFFH   001FEFH   ---    ---      **GAP**
*** '?PR?STDFLASH2XRAM_FUNCTION?STDFLASH2XRAM'  execution at: 00F000H
004000H   004010H   000011H   BYTE   UNIT     SROM           ?PR?STDFLASH2XRAM_FUNCTION?STDFLASH2XRAM
004011H   004CAFH   000C9FH   ---    ---      **GAP**
*** '?PR?STPFLASH2XRAM_FUNCTION?STPFLASH2XRAM' no space reserved for execution at: 00F000H
004CB0H   004CC0H   000011H   BYTE   UNIT     SROM           ?PR?STPFLASH2XRAM_FUNCTION?STPFLASH2XRAM
004CC1H   00A9FFH   005D3FH   ---    ---      **GAP**
00AA00H   00AA10H   000011H   BYTE   UNIT     CODE           ?PR?DFLASH_FUNCTION?DFLASH
00AA11H   00FFFFH   0045EFH   ---    ---      **GAP**
*** '?PR?STDFLASH2XRAM_FUNCTION?STDFLASH2XRAM' stored at: 004000H   execution at: 00F000H
00F000H   00F010H   000011H   BYTE   UNIT     CODE_XRAM      ?PR?STDFLASH2XRAM_FUNCTION?STDFLASH2XRAM
```

Figure 3 Map file

The Most Important:

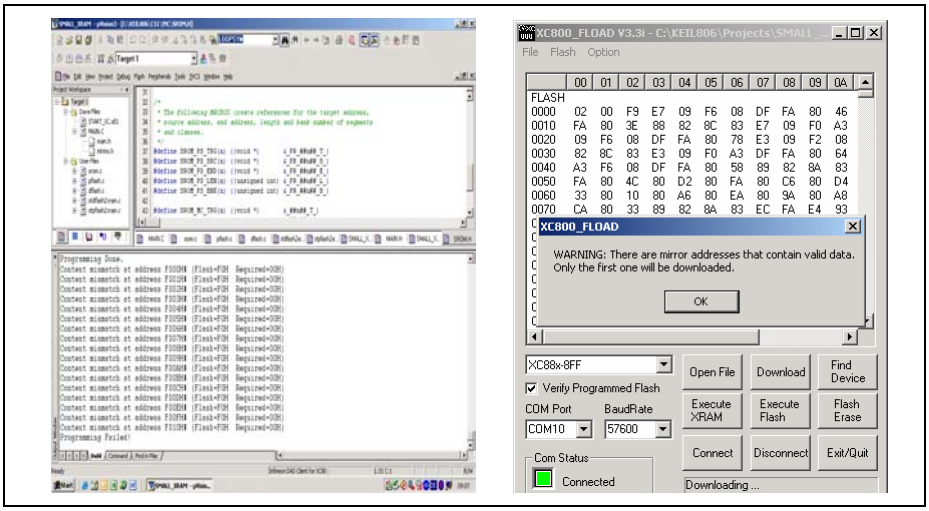


Figure 4 Loading hex file using U-Link/ Wiggler Box/ XC800 FLOAD

Just ignore the error messages and / or warnings.
That is because there is no algorithm on how to program into XRAM.
Simply remove debugger (or disconnect FLOAD) and reset your board.
Now the code is running.

Since these messages are not really nice, we can use a 3rd method to avoid.
Simply spoken the Linker is bypassed by an intermixing of the two methods shown before.

6 Simulator

```
//***************************************************************************
// @Filename      MAIN.C
//***************************************************************************
#include "MAIN.H"

#define LCALL(Address) (void (code *) (void)) Address) ();
#message " *** Ensure LX51 is selected ***"


extern unsigned char _PR_TEST_P3_S_;
extern unsigned char _PR_TEST_P3_L_;
extern void test (void);

void MAIN_vInit(void)
{
  SFR_PAGE(_su1, noSST);        // switch to page1
  CMCON         =  0x10;        // load Clock Control Register
  SFR_PAGE(_su0, noSST);        // switch to page0
  EA            =  1;
}

void xram_copy (void)
{
 unsigned int data lpcnt,clength = 0;
unsigned char code  *  data SrcPtr;
unsigned char xdata *  data DstPtr;
clength  = (unsigned int)& _PR_TEST_P3_L_;
  SrcPtr = (unsigned char code *)& _PR_TEST_P3_S_;
  DstPtr  = 0xF000;
for(lpcnt=0;lpcnt<clength;lpcnt++)
{
  *DstPtr++ = *SrcPtr++;
}
}

void main(void)
{
  MAIN_vInit();
  P3_DIR=255;
  P3_DATA=0;
  xram_copy();
  LCALL(0xF000)
  //test();

  while(1);
}
```

Simulator.ini - file

For debugging with simulator, the memory must mapped propperly.
The following mapping is done for a XC888 device

```
map X:0xF000,X:0xF5FF read write exec vnm
map C:0x0000,C:0x5FFF read exec
map C:0xA000,C:0xBFFF read exec
map C:0xF000,C:0xF5FF read exec

bk *    // Kill all previously breakpoints

E CHAR C:0xF000 = 0x00;  // simulate NOP instruction opcode
E CHAR C:0xF001 = 0x00;
E CHAR C:0xF002 = 0x00;
E CHAR C:0xF003 = 0x00;
E CHAR C:0xF004 = 0x00;  // only to ensure XRAM is filled with zeros
E CHAR C:0xF005 = 0x00;
E CHAR C:0xF006 = 0x00;
E CHAR C:0xF007 = 0x00;
E CHAR C:0xF008 = 0x00;  // at start up
E CHAR C:0xF009 = 0x00;
E CHAR C:0xF00A = 0x00;
E CHAR C:0xF00B = 0x00;

$ = 0xF000; // Set program counter to first XRAM address for execution
bs C:0xF000;  // Optional break point
```

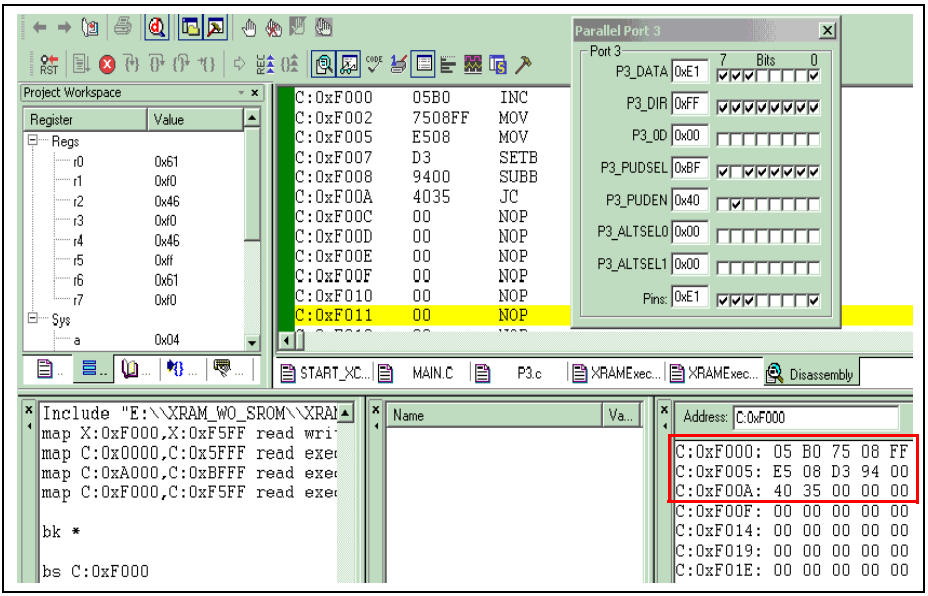


Figure 5 Simulator executing code from mapped XRAM

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG