

Program with Cross C Compiler **(for S12)**

Beibei Shao

“The C Programming Language” (1988)

- **Developed by Dennis Ritchie in Bell Lab for Unix on PDP-11**
- **“The C Programming Language 2nd Ed” (1988)**
Kernighan & Ritchie
- **The ANSI C Standard**
 - **Defined the Syntax and Semantics Standard**
 - **Defined the Standard C library**

大学计算机教育丛书（影印版）

SECOND EDITION

THE



PROGRAMMING
LANGUAGE

C程序设计语言

（第二版）

Brian W. Kernighan
Dennis M. Ritchie

清华大学出版社 • PRENTICE HALL

“Hello World!”

C and Unix

- C is originally from Unix (Linux gcc)
- Unix (Linux) is written with C

What Unix (Linux) Does?

- **Process management** (Kernel, but not Real Time)
- **Memory Management** (MMU)
- **Files Management**
- **I/O Management**

How is the I/O in Unix ?

- Everything in Unix is to be seen as a file
- Any I/O is seen as a file:
- If you want to use an I/O, you have to:
 - Mount the device (data flow or data block)
 - Open the file (Open for Read or Write)
 - Memory buffer
 - Access the file (R/W)
 - Close the file
 - Release the buffer

I/O in Cross C for Embedded System

- I/O is not the part of ANSI C
- I/O can be part of OS
- For an application, only write Hardware Independent Part with Cross C.
- Because normally no file sys in an Embedded System, I/Os can be treated as tasks

What Special? (1)

Cross C Compiler for Embedded?

- RAM & ROM are **limited resource**

- need memory management & code/data optimization

- memory map & system resources **various for different MCUs**

- unique **memory map header file** & **emulator personality file** for every MCU

- **variables** are stored in RAM, while **codes & constant tables** are stored in ROM

What Special? (2)

Cross C Compiler for Embedded?

- ❑ **system resources** (variables, I/Os & functional modules) **should be initialized** during **startup**
- ❑ mostly include **real time** applications
- ❑ executable codes are **NOT re-locatable**
- ❑ usually, functions cannot be **re-entrant**

What Special? (3)

Initialed and Non-Initialed Variables

- **Example of Initialed Variable:**

```
int a = 5;
```

- **Initial Non-Initialed Variable in your Program**

```
int a ;
```

```
a = 5;
```

- **Do not use Initialed Variable directly in your program!**

Useless Functions in ANSI C

- **Command Line parameters:**

```
main (argc,argv)
```

```
int *argc;
```

```
char *argv[];
```

```
{
```

```
    if (argc = 2)
```

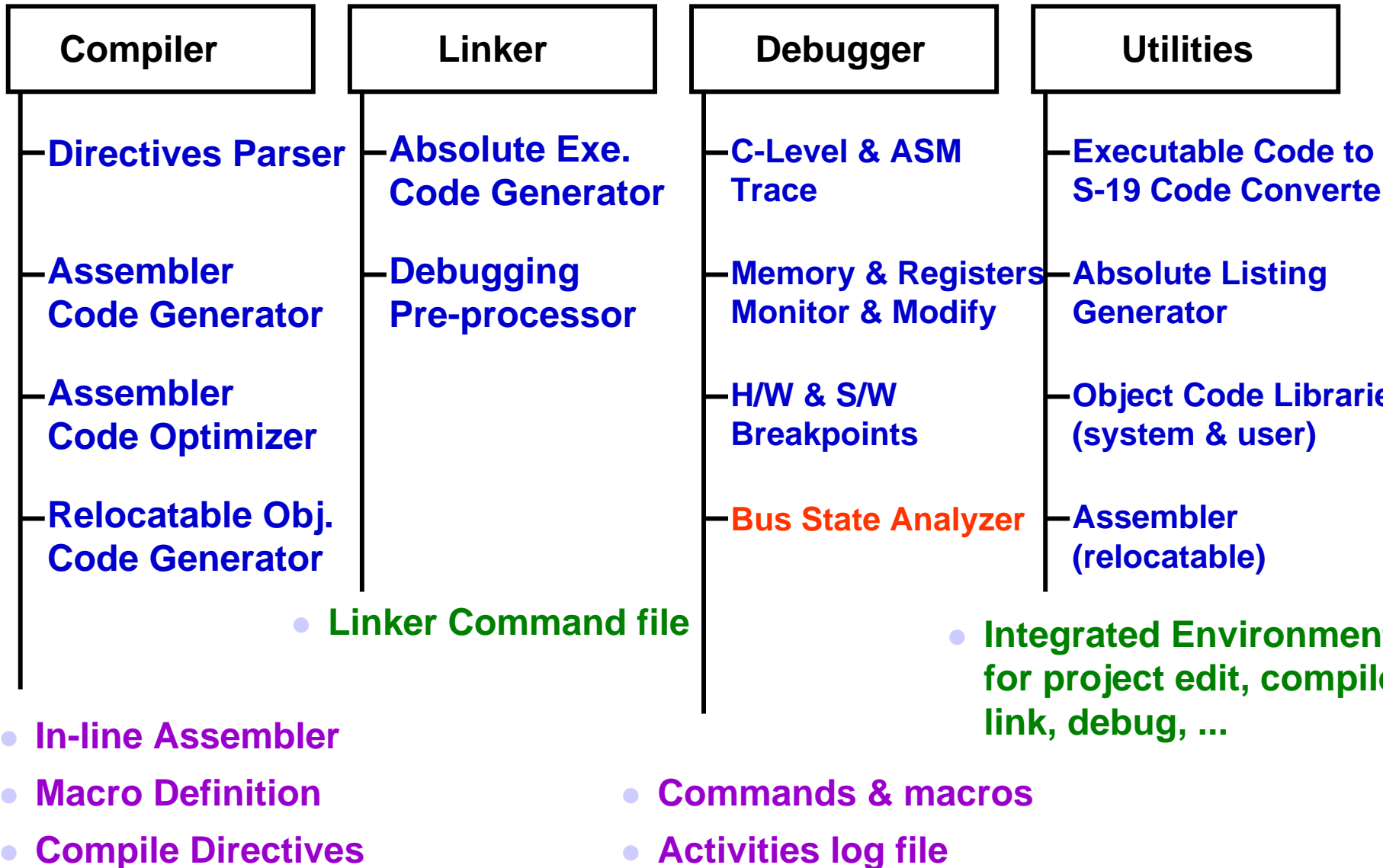
```
        .....
```

```
}
```

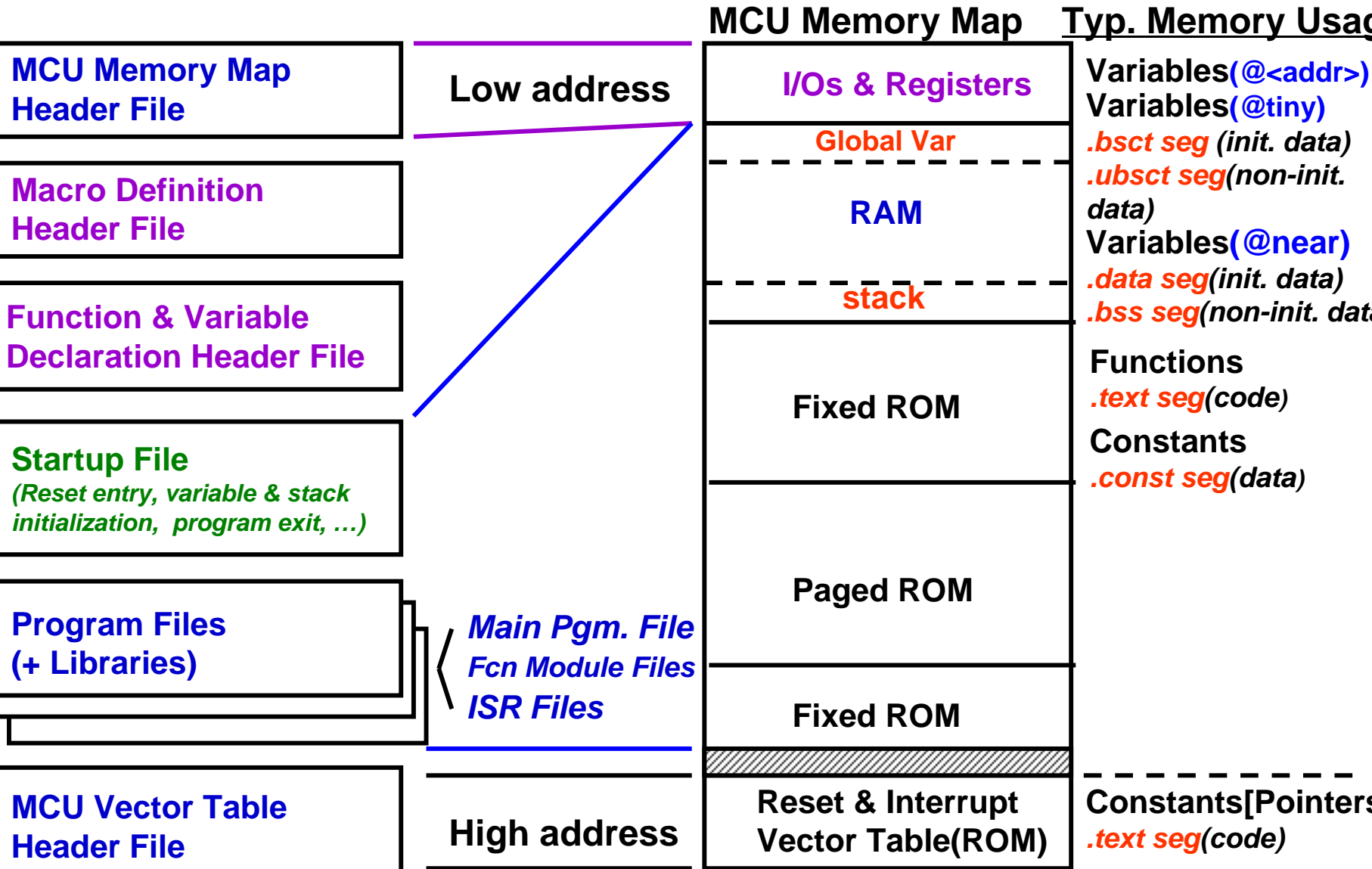
- **Input Output Redirection :**

```
Copy <file1 >file2
```

C-compiler Architecture



Source Files for HCS12 C Program



General Cross C compiler

If main() is a hardware independent Code, it can run at any computer with following hardware dependent code:

- **Startup file:**

- Initial Stack Pointer ;Where is RAM?
- Initial System ; Hardware dependent
- Call main() ; Hardware independent code
- Call Exit() ;return to Monitor

- **Much simple than ANSI C**

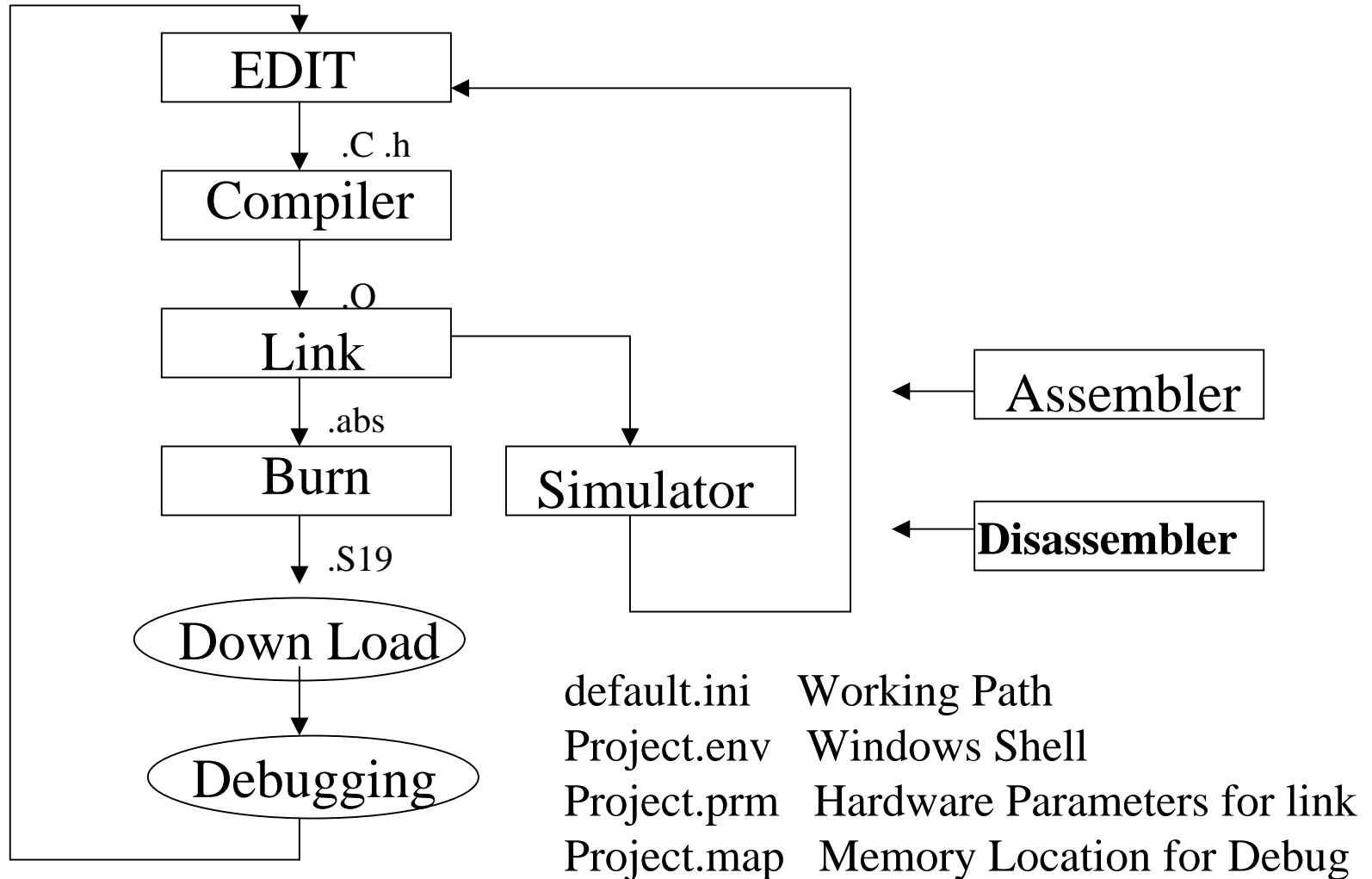
- No Command line parameters,
- No input output redirection
- No initial variable

Startup file

Set hardware environment to run C code

LDS	#StackTop	//initial SP
JSR	HardwareInit	//call Hardware Init function
JSR	main	//call main
JMP	exit	// return to debug

Compiler Processes



How to Start a Cross C ?

Start from simplest Program (1)

Because C is hardware independent language, without OS, `printf("Hello World")` may not work. Let us start from:

```
Function1()
```

```
{  
};
```

You got .o file, in which only one machine code of:

```
RTS ;
```

Start from Simplest Program (2)

Main()

```
{  
}
```

You got more code than other functions

Man()

```
{ printf (“\n Hello World!”)  
}          /* in most case, it does not work!*/
```

How to Insert ASM in C

```
#define HALT          {asm CLRA; asm SWI }  
#define EnableInterrupts {asm CLI;}  
#define DisableInterrupts {asm SEI;}
```

How to define hardware address:

In your .h file:

```
#define Input_port_A    (*(unsigned char *) 0x1800)
```

In your .c program:

```
Main()
```

```
{
```

```
Input_port_A = 5;
```

```
}
```

The flexibility makes your program less portability!

Mixing ASM and C Example

```
#pragma NO_ENTRY
```

```
void strcpy (char *from, char *to)
```

```
/* 'to' is passed in D, 'from' is passed on the stack SP:2 */
```

```
{  asm {  
    TFR      D,X  
    LDY      2,SP  
loop:  
    LDAA     1,Y+  
    STAA     1,X+  
    BNE      loop  
    }  
}
```

The flexibility makes your program less portability!

Better to Separate ASM & C

Write Hardware Dependent in ASM, Hardware Independent in C

/* Calling function & Called function */

application() **/* calling function*/**

```
{  
c = add(3,5);  
}
```

int add(int a, int b) **/* called function*/**

```
{  
return(a+b);  
}
```

It would be much flexible, if C can call ASM, ASM can call C

How is ASM Code Generated in C?

Global Var.

C:

```
extern int a;
```

```
Global()
```

```
{  
    a = 5;  
    return(0);  
}
```

How is ASM Code Generated in C?

Global Var.

C:

```
extern int a;
```

```
Global()
```

```
{  
    a = 5;  
    return(0);  
}
```

ASM:

```
a      XREF    a  
      RMB     2 ; somewhere in RAM
```

```
XDEF    Global  
  
Global LDD     #5  
      STD     a  
      CLRB  
      CLRA  
      RTS
```

How is ASM Code Generated in C Compiler?

Local Var.

Local()	Local:	PSHX	;int a in the stack
{		LDD #5	
int a;		TSX	; Let X points a
a=5;		STD 0,x	; assign a at D
return (a);		PULX	; release MEM
}		RTS	

Relationship between C & ASM

- Parameters Transfer in C and ASM
 - Data transferred by Stack
 - Where is the first parameter?
 - Where is the returned value?
 - What is the order of Parameters in the stack?

Parameters Transfer in Functions (CodeWarrior)

Define the function in C:

```
int Add(int a, int b, int c)
{
    return(a+b+c);
};
```

Call the Function in C:

```
main()
{
    d = Add(1,3,5);
}
```

Parameters Transfer in Functions (CodeWarrior)

Define the function in C: Compiler generate:

```
int Add(int a, int b, int c)
{
    return(a+b+c);
};
```

```
PSHD          ; c
LDD           6,SP    ; a
ADDD          4,SP    ; +b
ADDD          0,SP    ; +c
PULX          ; Release
RTS
```

Call the Function in
C:

```
main()
{
    d = Add(1,3,5);
}
```

```
LDAB          #1      ;a
CLRA
PSHD
LDAB          #3      ;b
PSHD
LDAB          #5      ;c
BSR           Add     ; Call Add()
LEAS          4,SP    ;Release:SP+=4
```

Parameters Transferred in Functions

(CodeWarrior)

- **Return Value:**
 - Char in B; int or *p in D; long in X:D;
- **Called function()**
 - Believe Par's already in the stack, last par. in D, first par is far from SP
- **Calling Function()**
 - Par's pushed from Left to Right, last one in D
- **Unfixed number of Par does the opposite**
 - Par's pushed from Right to Left, first one in D

Notice: CodeWarrior is Diff. from GCC

- **CodeWarrior:**
 - For fixed number of Parameters for Calling Function():
 - Par's pushed from **Left to Right**, last one in D
- **GCC does the opposite:**
 - For fixed number of Parameters for Calling Function():
 - Par's pushed from **Right to Left**, first one in D
- **You can see, the diff. between Compilers**
 - That is why porting RTOS very depends the Compiler tools you used

Be Careful on Optimization

- **Modern Cross C compiler may optimize your code:**
 - by execution speed;
 - by code size;
 -
- **Do disable the optimization for hardware related code!**

Link File default .prm

Let Compiler knows your Hardware

/* This is a generic Prm File

**If it does not fit your needs, you may adapt it or choose an another one
in the linker preference panel */**

NAMES END

SECTIONS

MY_RAM = READ_WRITE 0x2000 TO 0x3FFF;

MY_ROM = READ_ONLY 0x4000 TO 0xFEFF;

PLACEMENT

DEFAULT_ROM INTO MY_ROM;

DEFAULT_RAM INTO MY_RAM;

END

STACKSIZE 0x600

VECTOR 0 _Startup /* set reset vector on _Startup */

Generate Motorola S Code

S00D0000746F776572732E616273EA

S123F0009BCCF4190000F30E0000000011570001F01A0000F026F01EF02010000058000068

S109F020FFFF00000000E8

S105FFFEF419F0

S123F028A7FECE1054898ACE1055650000931F9EEF028B869EE7019F8B8848594859898A63

S123F04897D6100BC71055D6100AC71054654F8395E601FEA702818789A7FA9E6F049E6F2B

.....

.....

.....

S123F3A8F014F7205395F687EE018AE6019EE706F69EE705AD509EEE02878AE6019EE7044A

S123F3C8F69EE703AD4095F795F687EE018AF68795E60387EE048A86F7956C0326026C028A

S123F3E86C0126017C6D0526026A046A05E60487EE058A65000092D095F687EE018AE60153

S123F408FA26A2A70A819FAB029EE7048B86A90081C6F005A502260DC6F00CB7FEC6F00D0

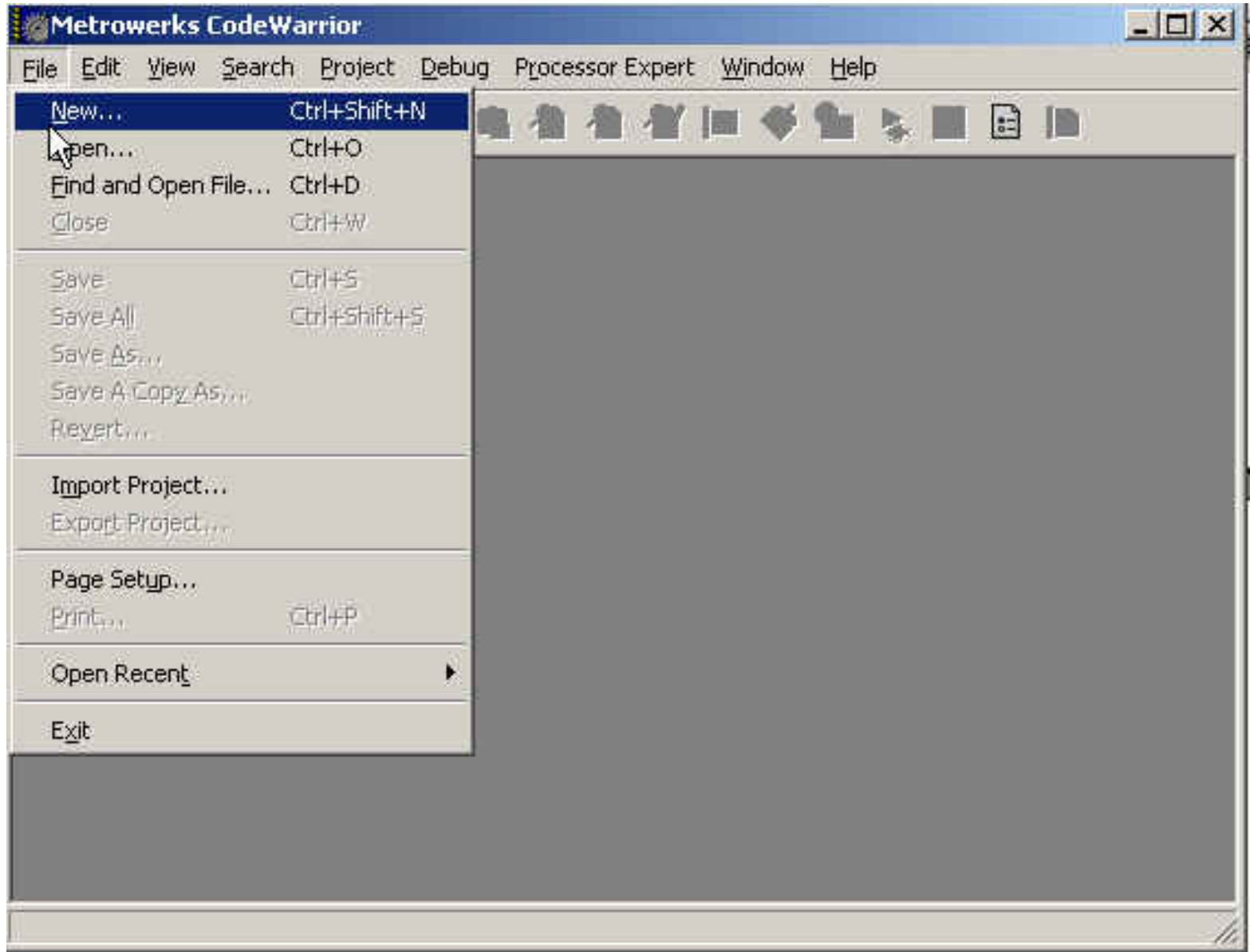
S116F428B7FF55FE94CDF331CEF006898ACEF007FD20DEA8

S105F0260000E4

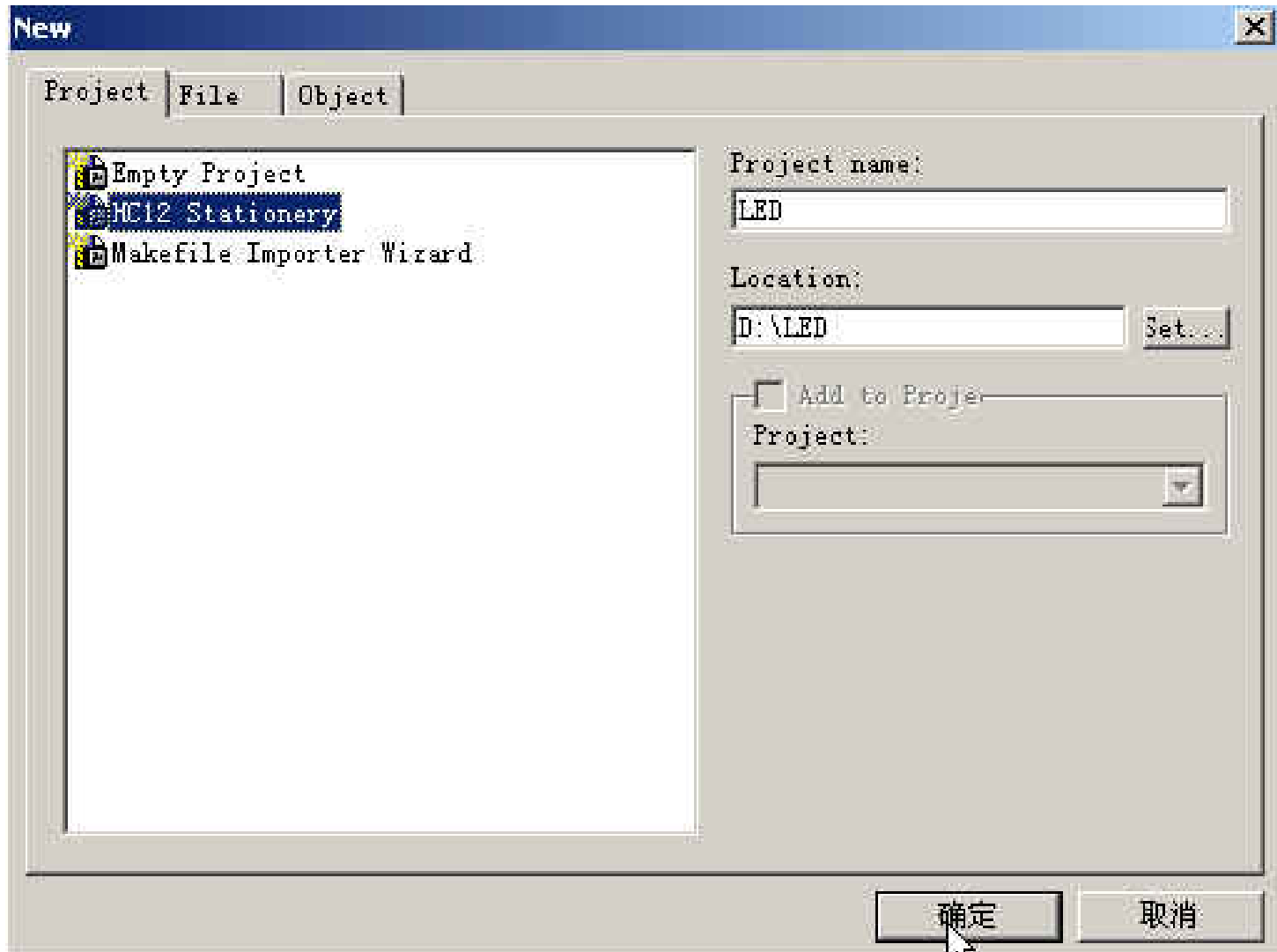
S903F0000C

CodeWarrior for S12 C Compile

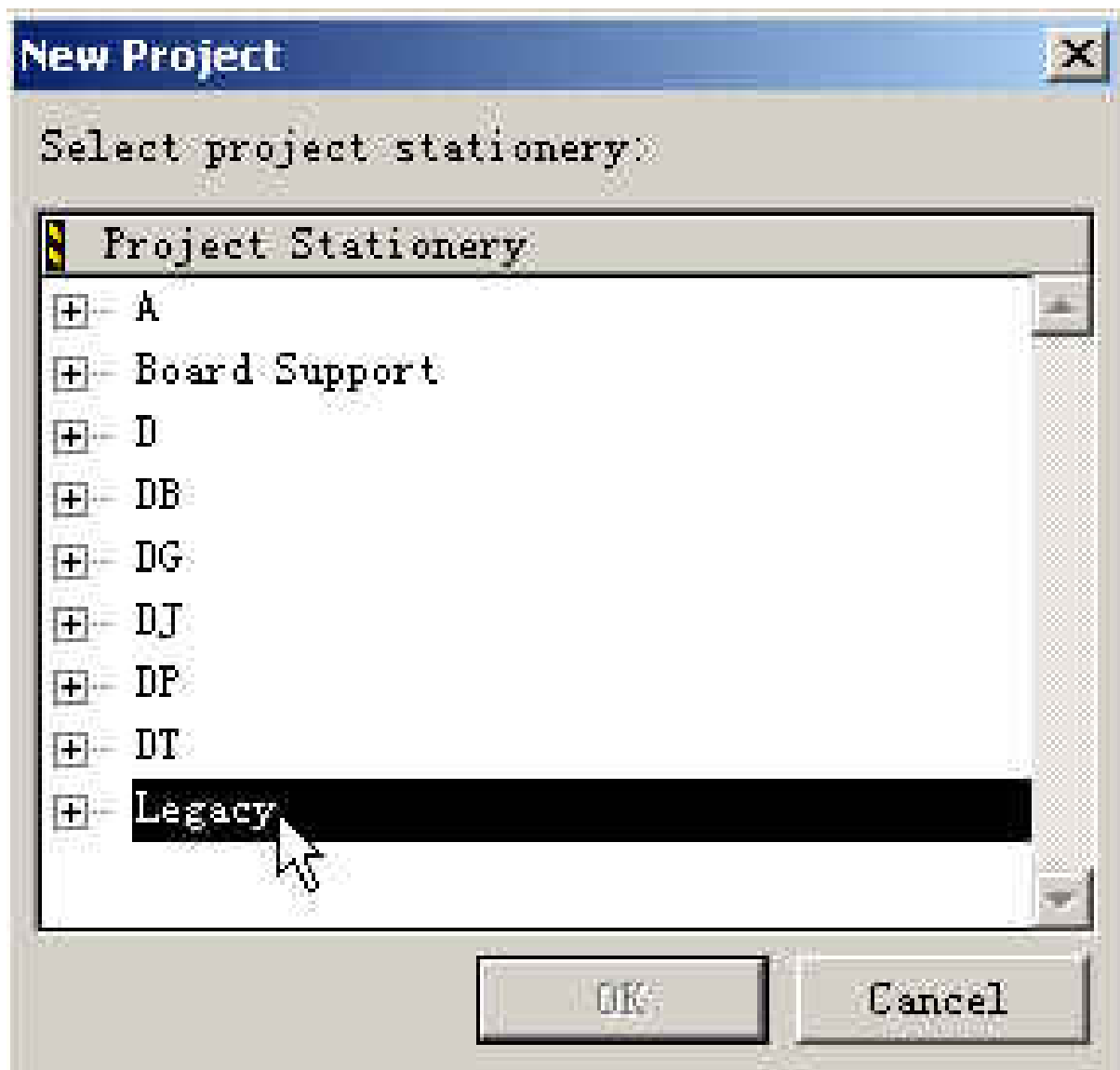
Open a New Project



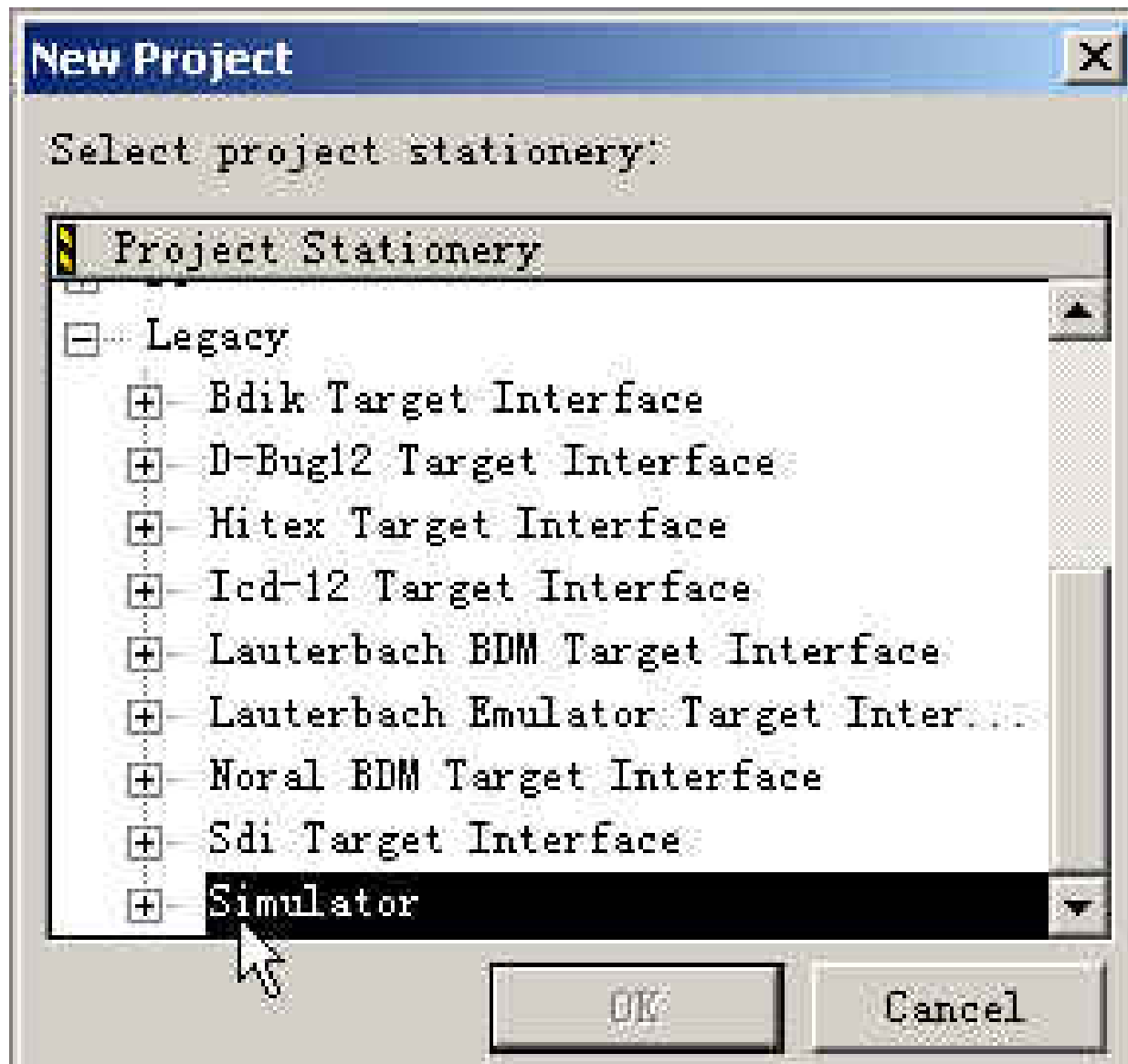
Assign Project Name



Use “Legacy” for General S12



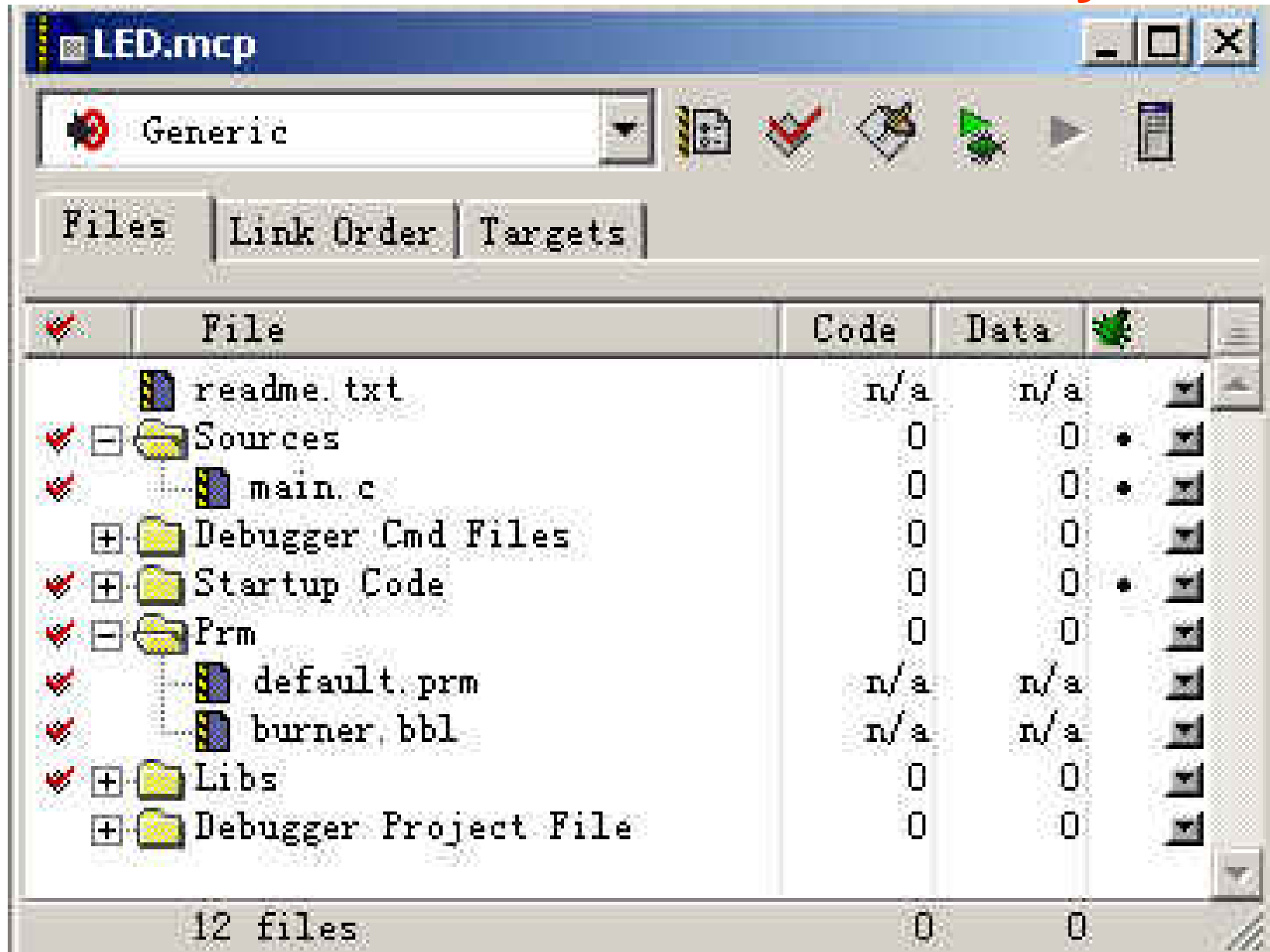
Do Not Need Any Hardware Tool



Select ASM or C



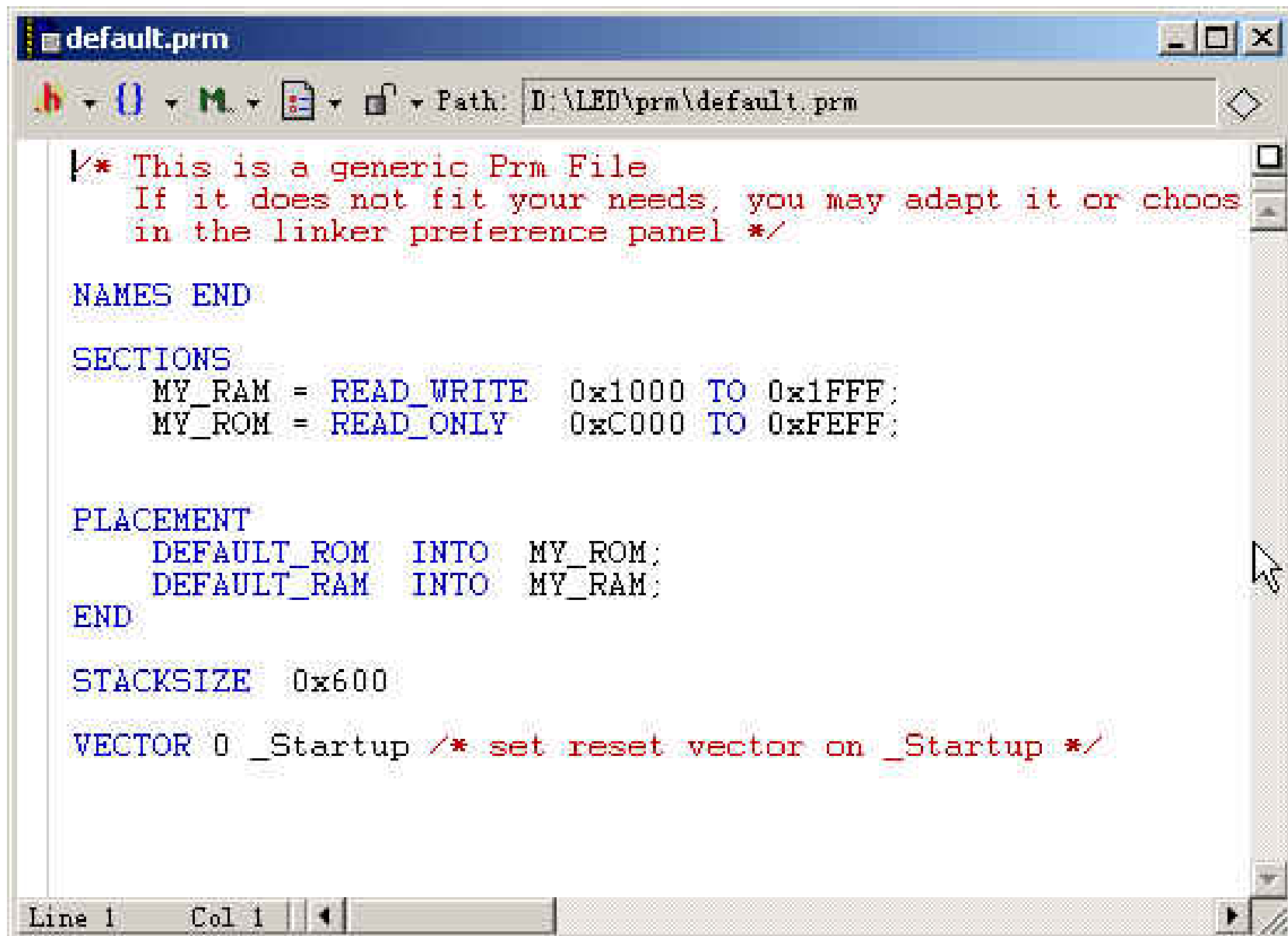
Files Generated Automatically



The screenshot shows the LED.mcp IDE window. The title bar is "LED.mcp". Below the title bar is a toolbar with icons for a generic target, a document, a checkmark, a folder, a green bug, a play button, and a list. Below the toolbar are three tabs: "Files", "Link Order", and "Targets". The "Files" tab is selected. The main area displays a list of files and folders in a table-like structure. The columns are "File", "Code", "Data", and a column with icons. The files listed are: readme.txt, Sources (folder), main.c, Debugger Cmd Files (folder), Startup Code (folder), Prm (folder), default.prm, burner.bbl, Libs (folder), and Debugger Project File (folder). The "Code" and "Data" columns show values for each file. The "Sources" folder is expanded, showing its contents. The "Libs" folder is also expanded. The "Debugger Project File" folder is expanded. The "Files" tab is selected. The status bar at the bottom shows "12 files", "0", and "0".

File	Code	Data	Icon
readme.txt	n/a	n/a	
Sources	0	0	*
main.c	0	0	*
Debugger Cmd Files	0	0	
Startup Code	0	0	*
Prm	0	0	
default.prm	n/a	n/a	
burner.bbl	n/a	n/a	
Libs	0	0	
Debugger Project File	0	0	
12 files	0	0	

Link file default.prm



The screenshot shows a text editor window with a blue title bar labeled 'default.prm'. The toolbar includes icons for file operations and a path field showing 'D:\LED\prm\default.prm'. The text area contains a linker script with red comments and blue code. The script defines memory sections (MY_RAM, MY_ROM) and their placement into default sections, sets a stack size, and defines a reset vector.

```
/* This is a generic Prm File
   If it does not fit your needs, you may adapt it or choose
   in the linker preference panel */

NAMES END

SECTIONS
    MY_RAM = READ_WRITE    0x1000 TO 0x1FFF;
    MY_ROM = READ_ONLY     0xC000 TO 0xFEFF;

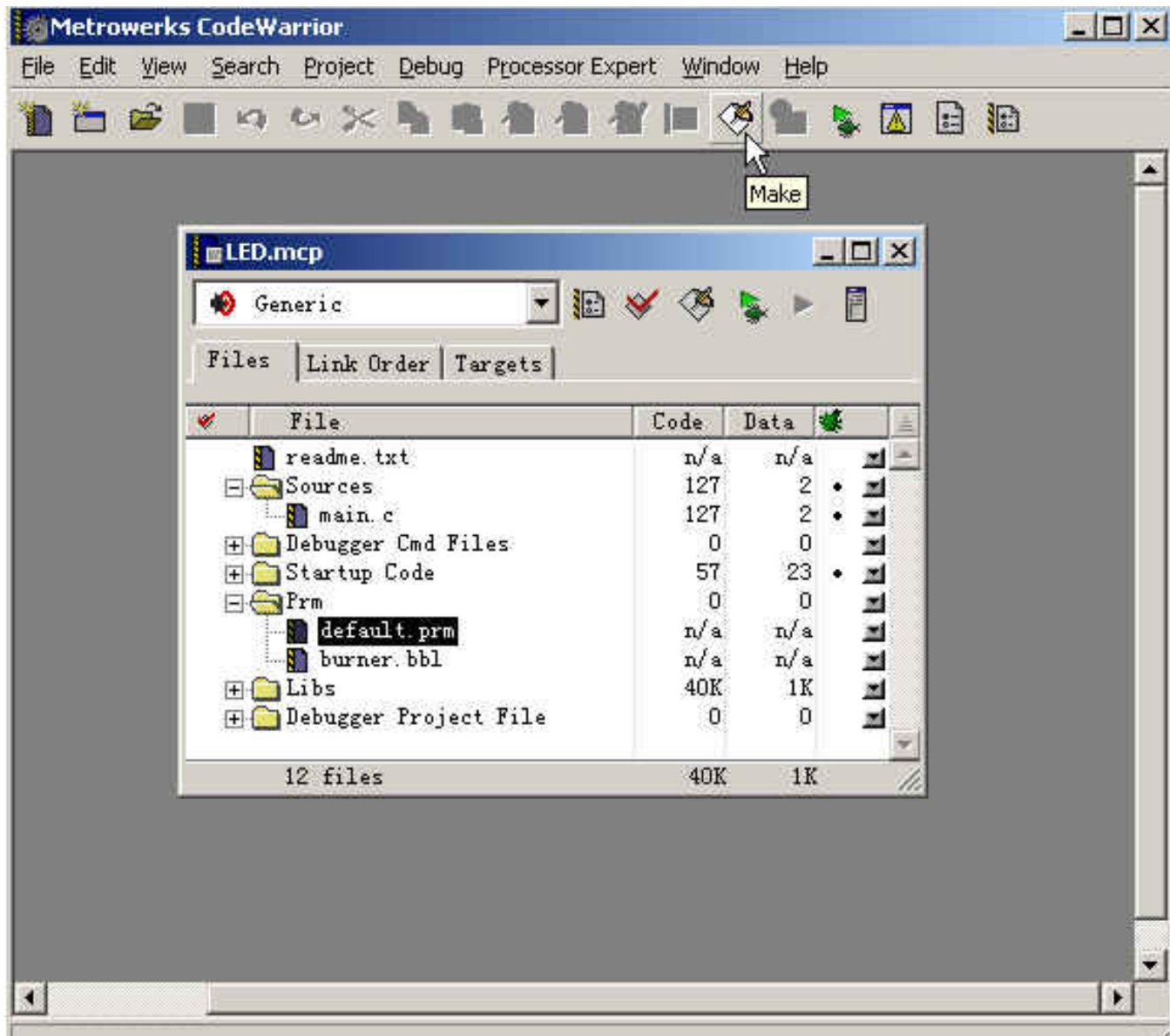
PLACEMENT
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
END

STACKSIZE 0x600

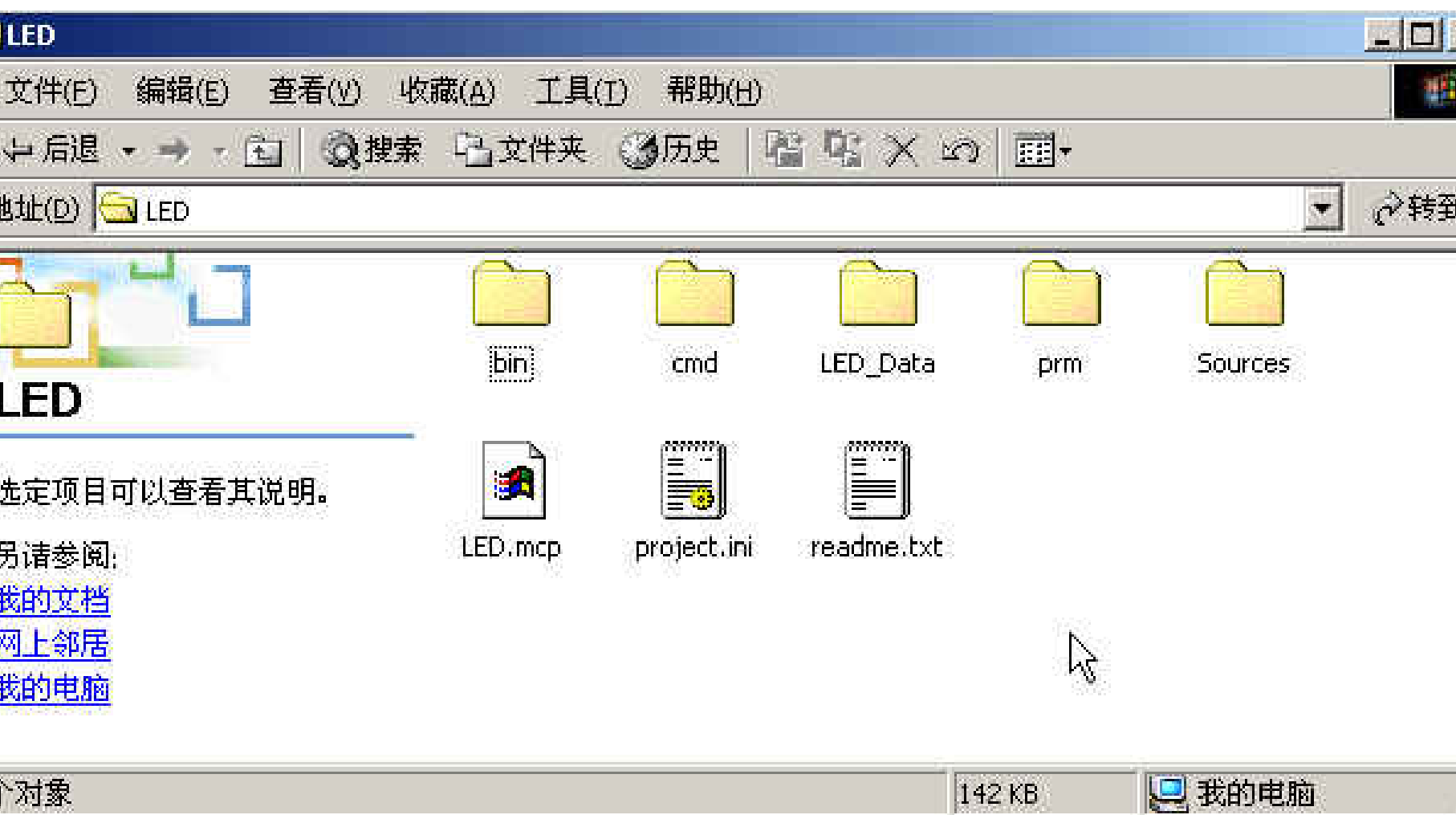
VECTOR 0 _Startup /* set reset vector on _Startup */
```

Line 1 Col 1

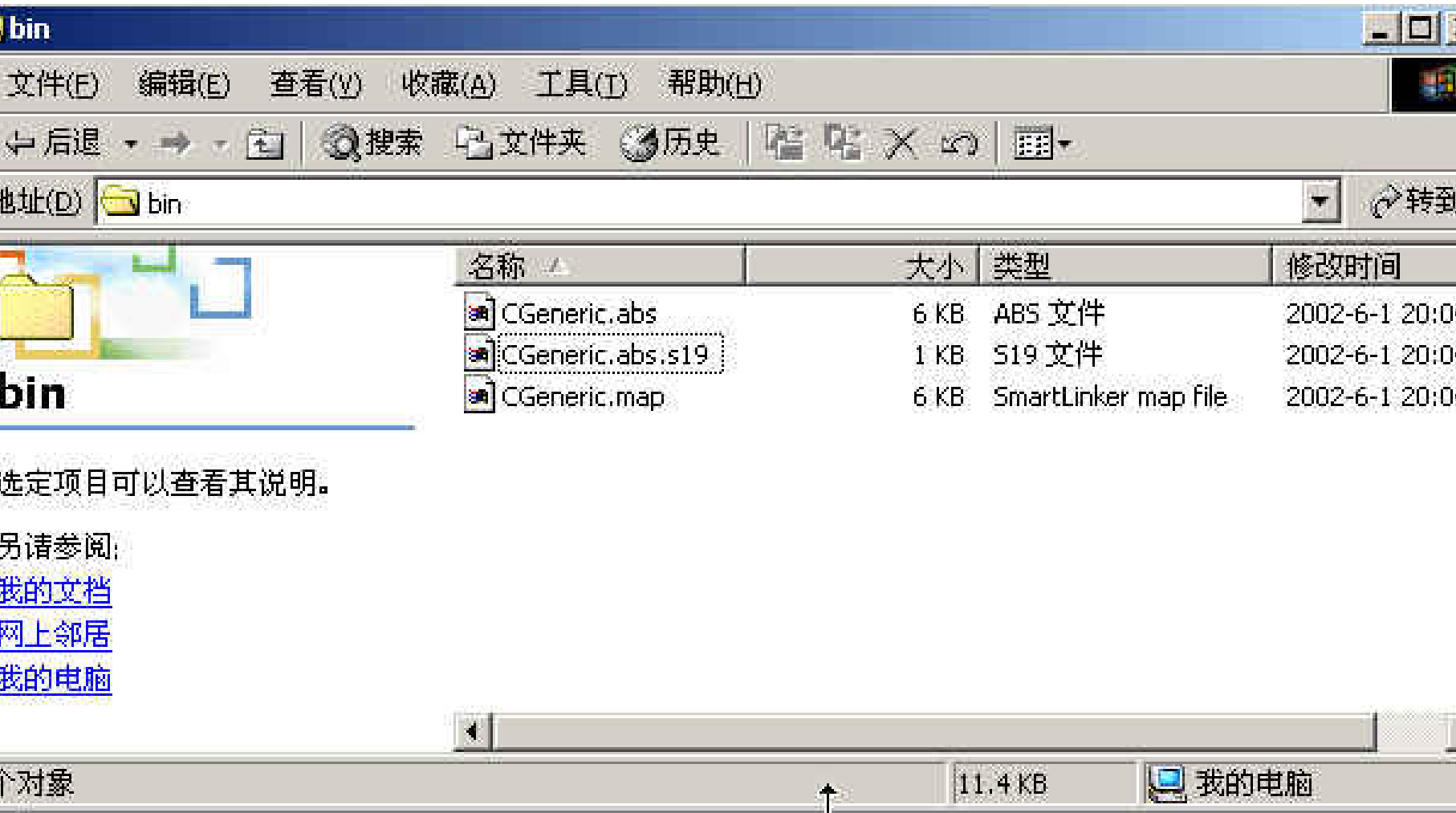
Use “Make” to do the compile & Link



Project File System



.S19 Code: CGeneric.abs.s19



Use Our Startup.c to Replace

```
#include "includes.h"
```

```
#include "main.h"
```

```
void _Startup(void) {
```

```
asm{
```

```
    LDS    #StackTop        //initial SP
```

```
    JSR    HardwareInit     //call HardwareInit function
```

```
    JSR    main              //call main
```

```
    JMP    MonitorWarmStart  //jump back to debug
```

```
}
```

```
}
```

About Printf() & Sprintf()

- **Printf() output formatted string to stdio**
- **In CodeWarrior:**
 - default stdio is BDM but SCI;
 - Printf() is powerful and in large code size;
 - We can replace with our printp()
- **Simplest way is to use hardware independent function sprintf()**
 - It outputs formatted string to memory buffer;
 - Write your hardware driver copy the buffered string to SCI