
RUNTIME UPDATING OF FLASH MEMORY IN ROMMABLE APPLICATIONS

INTRODUCTION

Updating the contents of Flash memory without an external programmer, requires some precaution when designing code that writes/erases the same Flash memory in which it is located. Currently, no flash memories exist that allow read cycles during a write/erase cyclus. Therefore, the code that performs the actual write/erase of the flash memory, **must** reside outside the flash memory.

The best way to overcome this problem is to copy the write/erase routines to RAM and execute them from there. This allows reprogramming the Flash memory from inside the system.

This application note describes how you can compile and locate code that updates Flash as part of your application. The code that updates the Flash can be allocated in Flash itself without the danger of overwriting: Special locator features will be used to take care of an automatic Flash-to-RAM copy phase before executing the code.

The following code will be used to demonstrate the principles of this process, using the TASKING R3000 toolset. Note that this code may require several changes/extensions to make it suitable for your particular situation and Flash memory type. This code is written to demonstrate the basic techniques, simulating Flash memory areas in RAM, so that this program can be tested and examined at common evaluation environments without real Flash memory.

prflash.c:

```
#define FLM_BNKSIZE      0x10          /* redefine to real Flash memory bank size */
unsigned char  flm_rambuf[ FLM_BNKSIZE];

/*
** read Flash memory
**
** flbp : byte address in Flash memory
**
** stub function for true Flash memory read code
** (if required at all)
*/
__inline unsigned char flash_read( unsigned char *flbp )
{
    return( *flbp );
}
```

```
/*
** write Flash memory
**
** flbp : byte address in Flash memory
** val  : new memory value
**
** stub function for true Flash memory write code
*/
void flash_write( unsigned char *flbp, unsigned char val )
{
    *flbp = val;
}

/*
** erase Flash memory
**
** flbp : byte address in Flash memory
**
** stub function for true Flash memory write code
*/
void flash_erase(unsigned char *flbp )
{
    *flbp = 0;
}

/*
** copy Flash memory to ram buffer
**
** rambuf : buffer address
** flash  : Flash memory address
** size   : number of bytes to copy from Flash to ram buffer
*/
flash2ram( unsigned char *rambuf, unsigned char *flash, int size )
{
    while ( size-- )
        *rambuf++ = flash_read( flash++ );
}

/*
** update ram buffer with new memory values
**
** rambuf   : buffer address
** buf_start: offset in ram buffer where update must start
** bufp     : pointer to buffer with update values
** len      : number of bytes to update
** bufsize  : rambuf size
*/
update_ram( unsigned char *rambuf, int buf_start, unsigned char *bufp, int *len, int bufsize )
{
    int i = 0;
    for ( i=0; *len && i + buf_start < bufsize; i++ )
    {
        *(rambuf+i+buf_start) = *bufp++;
        *len -=1;
    }
}

/*
** clear Flash memory
**
** flmp : Flash memory address

```

```

** size : number of bytes to clear
*/
clear_flash( unsigned char *flmp, int size )
{
    while ( size-- )
        flash_erase( flmp++ );
}

/*
** copy ram buffer to Flash
**
** rambuf: ram buffer address
** flmp : Flash memory address
** size : number of bytes to write from buffer to Flash
*/
ram2flash( unsigned char *rambuf, unsigned char *flmp, int size )
{
    while ( size-- )
    {
        flash_write( flmp, *rambuf );
        flmp++;
        rambuf++;
    }
}

/*
** program Flash memory
**
** flmp : Flash memory address
** bufp : buffer with new Flash memory values
** length : number of bytes to update
**
** This function updates Flash memory bankwise. The memory area
** in Flash that has to be updated does not have to be aligned at
** bank addresses.
**
*/
void prflash( unsigned char *flmp, unsigned char *bufp, int length )
{
    unsigned char *align_flmp = flmp;
    int block_len = length;
    align_flmp = (unsigned char*)((int)align_flmp & ~FLM_BNKSIZE);
    while ( length )
    {
        /* copy Flash to RAM */
        flash2ram( flm_rambuf, align_flmp, FLM_BNKSIZE );
        /* update RAM buffer */
        update_ram( flm_rambuf, flmp - align_flmp, bufp, &length, FLM_BNKSIZE );
        /* clear the Flash */
        clear_flash( align_flmp, FLM_BNKSIZE );
        /* copy buffer to Flash */
        ram2flash( flm_rambuf, align_flmp, FLM_BNKSIZE );
        if ( flmp != align_flmp )
            bufp += FLM_BNKSIZE - (flmp - align_flmp);
        else
            bufp += FLM_BNKSIZE;
        flmp = align_flmp + FLM_BNKSIZE;
        align_flmp = flmp;
    }
}

```

This code will be added to the following (very small) application:

main.c:

```

/*
** use a local memory buffer to simulate Flash memory updates
** for this example
*/
unsigned char flash_mem[80];
unsigned char flash_update[40];

extern void prflash( unsigned char *, unsigned char *, int );

void main( int argc, char *argv[] )
{
    int i;
    for ( i=0; i<40; i++ )
        flash_update[i] = flash_mem[i] = i;
    for ( ; i<80; i++ )
        flash_mem[i] = i;
    prflash( flash_mem +20, flash_update, 40 );
    for ( i=0; i<80; i++ )
    {
        printf( "%2d ", flash_mem[i] );
        if ( ( i + 1 ) % 10 == 0 )
            putchar( '\n' );
    }
    putchar( '\n' );
}

```

This program, which is for the sake of this example is considered ‘the normal program code’, will be compiled using the default section names:

c3 -O2 -g -v -c main.c

The compiler generates code in section **.text**, which the locator finally will map to the Flash memory area.

Although the Flash programming routines will also be allocated in Flash initially, different section names are required to enable the copy phase to RAM.

The command:

c3 -v -g -O2 -R.text=.ftext prflash.c

tells the compiler to rename the default section name for code to **.ftext**

After linking the object modules with the startup code and the libraries with the command:

c3 main.o prflash.o

the relocatable object file a.out is generated.

Note that it is important that you use the startup code as supplied in the library. The startup code uses a locator generated table (**Copy Table**) for copying ROM data sections from ROM to RAM (This is why a rommable application can still use initialized data: by copying the ROM image of the data section to the runtime RAM memory locations, the program finds its initialized data in RAM.) This feature will also be used to copy the code that updates the Flash from ROM to RAM.

Before the program can be located you need to make a modification in your locator description file. For this example we use the file **proto.dsc** as supplied with the TASKING R3000 C compiler. (Copy proto.dsc to pflash.dsc in your current directory)

Since we have added the section `.ftext` to the application, a locating entry for this section is required:

```
section .ftext {
    p_block          = general_data;
    p_c_block        = code;
    v_space          = kseg0;
    attribute        = s;
}
```

This entry informs the locator that we want this section to be located with runtime addresses in RAM (*general_data*) and a ROM image of this code in the memory block *code*. The locator detects that for this section the load addresses (in ROM) differ from the runtime addresses and therefore it reserves an entry in the **Copy Table**. The locator adds for each section that requires ROM-to-RAM copying one entry in the copy table with the start address in ROM the start address in RAM and the section length. The default startup code automatically picks up the entries in this tables and takes care of the copy phase before the program starts.

With the entry above, pflash.dsc looks like:

```
cpu          c:/r3000/etc/r3000.cpu
memory       c:/r3000/etc/sb3000.mem
software {
    defaults c:/r3000/etc/sb3000.dfl

    load_mod * {
        window = kernel; /* default window is 'user' */
        section .ftext {
            p_block          = general_data;
            p_c_block        = code;
            v_space          = kseg0;
            attribute        = s;
        }
    }
}
```

Note that you can find the locating rules for the default compiler generated sections in the file sb3000.dfl.

After the command:

```
lc3 -Mf1 -g a.out pflash.dsc
```

the locator generated two files: a.abs and a.map.

A.abs is the object file ready for debugging with CrossView and a.map shows the mapping of the sections in memory.

A.map shows:

MAP per P_BLOCK

=====

p_block code, general_data, stack_data:

Pid #	Section	Physical	Virtual	Size	Vect/Segm Space
0001	table	0x00040000	0x80040000	0x00000070	- kseg0
0001	[.data]	0x00040070	0x80040070	0x00000190	- kseg0
0001	[.sdata]	0x00040200	0x80040200	0x00000038	- kseg0
0001	[.double]	0x00040238	0x80040238	0x00000010	- kseg0
0001	.text	0x00040248	0x80040248	0x00002948	- kseg0
0001	[.ftext]	0x00042b90	0x80042b90	0x00000288	- kseg0
0001	heap	0x00080000	0x80080000	0x00002800	- kseg0
0001	stack	0x00082800	0x80082800	0x00001000	- kseg0
0001	.data	0x00083800	0x80083800	0x00000190	- kseg0
0001	.sdata	0x00083990	0x80083990	0x00000038	- kseg0
0001	.sbss	0x000839c8	0x800839c8	0x00000020	- kseg0
0001	.double	0x000839e8	0x800839e8	0x00000010	- kseg0
0001	.string	0x000839f8	0x800839f8	0x00000028	- kseg0
0001	.bss	0x00083a20	0x80083a20	0x00000170	- kseg0
0001	.ftext	0x00083b90	0x80083b90	0x00000288	- kseg0

As you can see there are four sections that have two entries in this table: one with and one without []. The entry within [] is the ROM copy of a section. Notice that in the 'ROM' area 0x80040000-0x80050000 the section .text is the only section which loads directly at its execution address. The other sections in this area are copied to their runtime locations at program startup.

Conclusion

This application note showed how you can create program code in ROM that executes from RAM, using the locator capability to generate images at load addresses different from their execution addresses. This feature saves you time when writing rommable code that reprograms all Flash memory in a system at runtime.

You can get an electronic copy of this application note and the program text at the TASKING web site: <http://www.tasking.com/support/R3000>

Information furnished is believed to be accurate and reliable. However, Tasking assumes no responsibility for the consequences of use of such information nor for any infringement of patents or rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Tasking. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. Tasking products are not authorized for use as critical components in life support devices or systems without the express written approval of Tasking.

© 1996 Tasking Software BV - All Rights Reserved

Boston Systems Office, BSO, Boston Systems Office/TASKING, BSO/TASKING and EDE are registered trademarks of Tasking Inc and Tasking Software BV.