Embedding Flash Programming into TMS320LF240x Applications

C2000 Applications Group

Texas Instruments Incorporated

ABSTRACT

Embedded Flash programming offers several opportunities to the system designer by allowing customization of the functionality of the device in system. This can enable field reprogramming for firmware updates or calibration data storage. To do this flash programming must be integrated into the end application. To accelerate this process TI has implemented an API based interface to the flash programming algorithms. This document describes these algorithms and how to integrate these into your TMS320LF240x Application.

Contents

Figures

1 Introduction

On the TMS320LF240x DSP Controllers, there is on-chip flash memory. This flash memory contains program code for the DSP controller. Reprogramming the flash memory in the system can offer several opportunities. TI supplies implementations of programming algorithms to program this flash memory. This document examines several aspects of working with the algorithms.

1.1 Why embed flash programming into the application ?

Incorporating the a DSP with an embedded flash offers several advantages. Some of these are:

1. It is possible to customize the functionality of the device in system. A single controller board can serve multiple servo-axis designs. The controller can be customized in the system to suit the system it is installed in. This can minimize inventory carried for the controller boards, offering significant flexibility
2. Firmware can be field up-dated for better performance or for added features. This can also be used to eliminate a factory recall for software upgrades.
3. Calibration data may be stored in the embedded flash during installation or factory testing to account for the variation in the system parameters. In several applications the controller can run calibration routines during installation, and then store the calibration data into the embedded flash memory.

To accelerate the integration of the flash programming algorithms into the system application, TI has implemented algorithms with an easy to use API.

1.2 Flash API

The Flash API is a simple-to-use interface to the flash programming algorithms. It consists of well documented function calls, that the client application calls to perform flash specific operations. In a layered view of the system, the flash API can be seen as a layer which abstracts complex functionality. The flash API provides a simple way to use the flash memory core, registers and algorithm functions

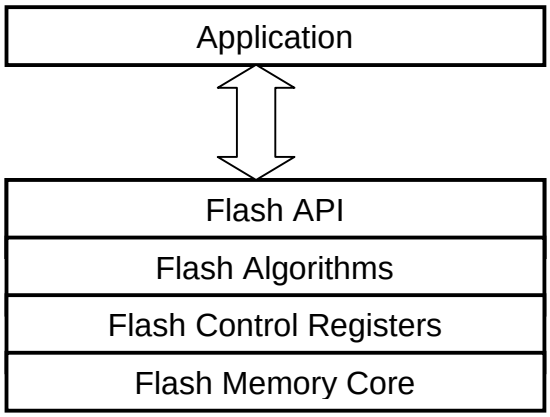


Figure 1. Flash API Organization

This helps abstraction of the flash memory details from the application, and leads to better organization of the system. TI provides flash algorithms for performing the operations such as Clear, Erase, and Program on the LF240x Controllers. Using TI supplied algorithms allows you to focus on your application, and frees up developer time to concentrate on your value proposition.

The flash API has been written to be easy to integrate into client applications. This means that you are freed up from managing complex timing requirements and verification levels, to focus on getting your application done faster.

2 Working with the Flash API

Integration of the Flash Programming into the system requires that the system designer implement operations so that several key requirements are satisfied. These are discussed in this section, together with implementation aspects.

2.1 Clock Rate Configuration

The algorithms must be configured for the CPU clock rate at which they will run. The flash programming algorithms contain several delay parameters, implemented as software delays. Timing is VITAL, and to ensure this the flash algorithms must be run at the correct speed.

To configure the flash programming algorithms locate the ‘var.h’ file in the ‘include’ sub-directory.

Step 1 is used only by the JTAG based front-end programs to set the PLL pre-scalar. This does not actually get used in the embedded algorithms. However the CPU clock speed combination, so that the final clock speed at which your system runs must be determined from the (known) input clock speed and the PLL multiplier put in by the system boot function.

Step 2: This file contains the ‘include’ statement used to choose which timing file is used to compile the flash algorithms. Uncomment the include line to include the timing file for the speed at which the target system is to be run.

Step 3: If the timing file for the actual speed the target system is to be run is not available, then such a file can be generated by using the excel spreadsheet in the ‘include’ sub-directory.

Step 4: This is completed when compiling your main project.

See Var.h for all the details.

2.2 Run Time Considerations

The On the LF240x DSP controllers, there is only one flash core. The flash core architecture imposes the restriction that the flash core can perform only one operation at a time. So when programming the flash, code cannot execute from the flash. Consequently all algorithm code must be run from RAM. If any communication routines run in between flash programming operations, they must be copied to run from RAM as well.

To this end the COPY_XXX functions are used to copy the corresponding algorithm into RAM at runtime. The runtime addresses are determined during linking and the linker command file sections are discussed in Section 2.4.

2.3 Programming Voltage Supply

During the flash programming process, pulses are applied to the flash array with several different voltages. These voltages are generated by an on-chip charge pump. This charge pump is powered from the Vccp pin, and 5V must be connected to this pin during the flash programming process. This supply should source at least the current as per the device datasheet. There should NOT be a resistor in this line.

2.4 Linker Command File Sections

The flash algorithms contain relocation sensitive code. This is placed by the algos into the CSPL_text, ESPL_text, and PSPL_text sections (for the clear, erase and program respectively).

The SPL_text section in the linker command file example shown in the Figure 2, shows how these three sections are concatenated into a single SPL_text section. This must be done exactly as shown, since all the three sections are address sensitive.

The SPL_text section has separately defined run and load addresses. These allow the COPY functions (discussed in section TBD) to copy the flash algos to RAM at runtime. (Recall that this is a constraint from the flash architecture – section 2.2. The rest of the code from the clear, erase and program algorithms is assembled into the CLR_text, ERA_text and PGM_text sections. This is concatenated into a single ALG_text section. This is run from RAM as well, similar to the SPL_text section.

The KER_text section is wrapped between two dummy sections (KER_strt and KER_end). This allows copying the KER_text section into RAM at runtime without requiring any labels to be defined within the KER_text section. Any and all code placed into the KER_text section by user functions will be copied to RAM at runtime.

The linker command file shown in Figure 2 does not overlay these algorithms in RAM. This allows the ENTIRE flash array to be cleared and erased, without requiring a sector to remain un-modified. To reduce the RAM usage, the algorithms can be overlaid in memory, and this is discussed in section 2.5. It may be noted that this RAM is used only during programming, that is, it is free for other usage at runtime, when the flash is not being programmed. However if it contains data, this data must be expendable, for it will be trashed with the flash algos are copied into RAM.

```
MEMORY
{
  PAGE 0 : VECS:     origin = 0x0000, length = 0x0040
  PAGE 0 : PROG:     origin = 0x0044, length = 0x7fbc
  PAGE 0 : ABSRAM:   origin = 0x8000, length = 0x0040
  PAGE 0 : SARAM:    origin = 0x8040, length = 0x07c0
  PAGE 1 : BLKB2:    origin = 0x0060, length = 0x0020
  PAGE 1 : BLKB0B1:  origin = 0x0200, length = 0x0200
}
SECTIONS
{
 VECTORS : {} > VECS PAGE 0
    .text   : {} > PROG PAGE 0
    .cinit  : {} > PROG PAGE 0

/* Allocate the relocation sensitive portions in SARAM at proper addresses. The
relocation sensitive section should always be sent to a RAM block starting on XXX0
   for the relocation mechanism to properly locate the code with relocation
sensitivities. */

    SPL_text: load = PROG PAGE 0, run= ABSRAM PAGE 0
    {
    . += 7 ;
    ClearAlgoRunSpl   = . ;
    *(CSPL_text)
    . += 6 ;
    ProgramAlgoRunSpl = . ;
    *(PSPL_text)
    . += 6 ;
    EraseAlgoRunSpl   = . ;
    *(ESPL_text)
    }

/* Next deal with the rest of the algorithm code */
    ALG_text : load = PROG PAGE 0, run = SARAM PAGE 0
    {
    ClearAlgoRunMain = .   ;
    *(CLR_text)
    EraseAlgoRunMain = .   ;
    *(ERA_text)
    ProgramAlgoRunMain = . ;
    *(PGM_text)
    }
  KER_comp : load = PROG PAGE 0, run = SARAM PAGE 0
    {
       KernelRun = .;
       *(KER_strt)
       *(KER_text)
       *(KER_end)
    }
    .flshvar  : {} > BLKB2  PAGE 1
}
```

Figure 2. Linker Command File Example

```
MEMORY
{
  PAGE 0 : VECS:     origin = 0x0000,    length = 0x0040
  PAGE 0 : PROG:     origin = 0x0040,    length = 0x7fc0
  PAGE 0 : ABSRAM:   origin = 0x8000,    length = 0x0040
  PAGE 0 : SARAM:    origin = 0x8040,    length = 0x07c0
  PAGE 1 : BLKB2:    origin = 0x0060,    length = 0x0020
  PAGE 1 : BLKB0B1:  origin = 0x0200,    length = 0x0200
}
SECTIONS
{       VECTORS : {} > VECS PAGE 0
        .text   : {} > PROG PAGE 0
        .cinit  : {} > PROG PAGE 0
/* Allocate the relocation sensitive portions in SARAM at proper addresses. The
relocation sensitive section should always be sent to a RAM block starting on XXX0
   for the relocation mechanism to properly locate the code with relocation
sensitivities. */
 UNION    run = ABSRAM PAGE 0
 {      CSPL_tex: load = PROG PAGE 0
        {
        .+= 7;
        ClearAlgoRunSpl = . ;
        *(CSPL_text)
        }
        PSPL_tex:  load = PROG PAGE 0
        {
        .+= 7;
        ProgramAlgoRunSpl = . ;
        *(PSPL_text)
        }
        ESPL_tex:  load = PROG PAGE 0
        {
        .+= 7;
        EraseAlgoRunSpl = . ;
        *(ESPL_text)
        }
 }
UNION   run = SARAM  PAGE 0
 {      CLR_text:    load = PROG PAGE 0
        {
        ClearAlgoRunMain = .   ;
        }
        ERA_text:    load = PROG PAGE 0
        {
        EraseAlgoRunMain = .   ;
        }
        PGM_text     load = PROG PAGE 0
        {
        ProgramAlgoRunMain = . ;
        }
    }
  KER_comp : load = PROG PAGE 0, run = SARAM PAGE 0
    {   KernelRun = .;
        *(KER_strt)
        *(KER_text)
        *(KER_end)
    }
    .flshvar  : {} > BLKB2  PAGE 1
}
```

Figure 3. Overlaying algorithms to conserve RAM

2.5 Overlaid linker command file allocation

In embedded systems with memory constraints it is desirable to reduce RAM consumption. To do this in certain cases the algorithms may be overlaid in memory. In some systems it may be possible to leave a sector unmodified, and this can be used to contain the algorithms. Or, the algos may be streamed in over the serial port, with the algos contained on the host. To overlay the algos, the UNION statement is used, and this is shown in Figure 3.

2.6 Device Type setting

The Flash API version 1.3 onwards supports the LF2401A DSP Controller as well as the other LF240x controllers. Since the LF2401 has only 8k words of flash, it needs separate handling. Accordingly a configuration option has been made available in the file VAR.H. This allows the API to be targeted to the LF2401, or the LF240x controllers.

To set this up edit the VAR.H option section:

1. Determine the device to be programmed.

Device number	Device type setting
'LF2401A	LF2401A
'LF2407	LF2407
'LF2407A	LF2407
'LF2406	LF2407
'LF2406A	LF2407
'LF2402	LF2407
'LF2402A	LF2407
'LF2403A	LF2407

2. Set the device type setting. (Change ONLY the DEV_TYPE setting line.)

```
DEV_TYPE     .set     LF2401A
```

3 API Reference

This section describes the API interface to the flash programming algorithms and utility functions.

3.1 List of Functions implemented by the API

Here is a list of the functions in the ASM interface:

Group A: Flash Operations

1. CLEAR_FLASH
2. ERASE_FLASH
3. PROGRAM_FLASH

Group B: Utility Functions

4. COPY_CLEAR_ALGO
5. COPY_ERASE_ALGO
6. COPY_PROGRAM_ALGO
7. COPY_KERNEL

Group A: Flash Operations

1. void clearFlash(FlashAlgoVars *);
2. void eraseFlash(FlashAlgoVars *);
3. void programFlash(FlashAlgoVars *);

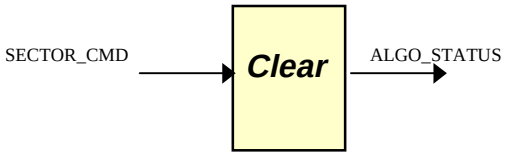
Group B: Utility Functions

4. void copyClearAlgo(void);
5. void copyEraseAlgo(void);
6. void copyProgramAlgo(void);
7. void copyKernel(void);

3.2 CLEAR

Description

The Clear functions pre-conditions the flash sector(s) in preparation for the erase.



Availability

This module is available in two interface formats:

The direct mode assembly-only interface (Direct ASM)
The C-callable interface version.

Module Properties

Type: Target Dependent, Application Dependent
Target Device/s: LF2402, LF2406, LF2407
File Name(s): clr_alg.asm, var.h, var.asm, rundefs.h, timings.xx.
C-callable Version File Name(s): Cflash.h

Item	Asm only	C callable ASM	Comments
Code size	265 words	265 words	234 words when configured as LF2401A (See section 2.6 for more information on how to configure)
Data RAM	24 words	24 words	
XDAIS ready	No	No	
XDAIS component	No	No	Hardware drivers.
Multiple Instances	No	No	

(Note: The Clear, Erase, Program and Copy share the same block of 24 words of data RAM – they use it in turns)

Direct ASM Format: Interface Description

Module Terminal Variables			
Name	**Description**	**Format**	**Range**
Inputs			
SECTOR_CMD	Describes which sectors are to be cleared. Bit 0 corresponds to sector 0, bit 1 to sector 1 etc. If the Bit in SECTOR_CMD is set then the Sector is CLEARED. if the bit is a 0 then the sector is left alone.	integer	0x0000 to 0x000F.
Outputs			
ALGO_STATUS	Returns an error code reflecting the operation status. 0x0000: Operation was run to completion. 0x0001: The algorithm failed to clear the flash despite applying maximum number of pulses allowable. 0x000a: The algo was called with a zero sector mask, i.e. the algo was asked to clear 'no sectors'.	integer	

Module Usage / Calling Convention

Variable Declaration:
In the system file include the following statements:

```
.include var.h
```

Memory map:
All variables are mapped to a named section '.flshvar'
Code is mapped into the CLR_text and CSPL_text sections.

Special Variables

```
flashAlgoVars    External structure instance.

.globl    flashAlgoVars

flshvar      .struct
ADDR         .int
PAD          .int
READ         .int
```

```
DATA        .int
PAD1        .int
PLS_CNT     .int
LASTVER     .int
FL_SECST    .int
FL_SECEND   .int
FL_CMD      .int
ERASESEC    .int
DATA_PTR    .int
FAIL_CMD    .int
SECTOR_KEY  .int
SECTOR_CMD  .int
ALGO_STATUS .int
flshvar_len .endstruct

flashAlgoVars   .tag    flshvar
```

The flashAlgoVars is assigned the characteristics of flshvar structure.

Example code:

```
LDP     #flashAlgoVars.SECTOR_CMD        ;Point DP to flash vars.
SPLK    #0fh,flashAlgoVars.SECTOR_CMD    ;Pass sector command.
CALL          CLEAR_FLASH                          ;Call the clear routine.
```

C / C-callable ASM Format: Interface Description

Module Terminal Variables

Module Terminal Variables			
Name	**Description**	**Format**	**Range**
Inputs			
SECTOR_CMD	Describes which sectors are to be cleared. Bit 0 corresponds to sector 0, bit 1 to sector 1 etc. If the Bit in SECTOR_CMD is set then the Sector is CLEARED. if the bit is a 0 then the sector is left alone.	integer	0x0000 to 0x000F.
Outputs			
ALGO_STATUS	Returns an error code reflecting the operation status. 0x0000: Operation was run to completion. 0x0001: The algorithm failed to clear the flash despite applying maximum number of pulses allowable. 0x000a: The algo was called with a zero sector mask, i.e. the algo was asked to clear 'no sectors'.	integer	

Special Constants and Datatypes

FlashAlgoVars **Data type for interfacing to the flash algorithm variables**

flashAlgoVars **External structure instance.**

The template for FlashAlgoVars is as below:

```
typedef  struct FlashAlgoVars {
int ADDR;
int PAD;
int READ;
int DATA;
```

```
int PAD1;
int PLS_CNT;
int LASTVER;
int FL_SECST;
int FL_SECEND;
int FL_CMD;
int ERASESEC;
int DATA_PTR;
int FAIL_CMD;
int SECTOR_KEY;
int SECTOR_CMD;
int ALGO_STATUS;
} FlashAlgoVars;
```

Module Functions: Clear

void extern clearFlash(FlashAlgoVars *);

Module Usage

Structure Reference

```
extern FlashAlgoVars flashAlgoVars;
```

Calling the Clear function

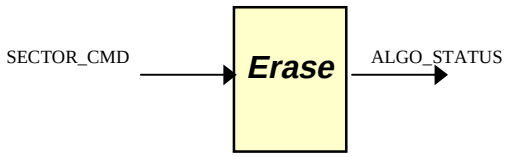
The following example calls the clear algo:

```
/* Setup the sector permissions */
 flashAlgoVars.SECTOR_CMD=(0xf);
/* Call the clear function */
 clearFlash(&flashAlgoVars);
if(0 != flashAlgoVars.ALGO_STATUS)
 {
/* Handle failure */
 }
```

3.3 ERASE

Description

Erases the flash sector(s).



Availability

This module is available in two interface formats:

The direct mode assembly-only interface (Direct ASM)
The C-callable interface version.

Module Properties

Type: Target Dependent, Application Dependent
Target Device/s: LF2402, LF2406, LF2407
File Name(s): era_alg.asm, var.h, var.asm, rundefs.h, timings.xx.
C-callable Version File Name(s): Cflash.h

Item	Asm only	C callable ASM	Comments
Code size	332 words	332 words	306 words when configured as LF2401A (See section 2.6 for more information on how to configure)
Data RAM	24 words	24 words	
XDAIS ready	No	No	
XDAIS component	No	No	Hardware drivers.
Multiple Instances	No	No	

(Note: The Clear, Erase, Program and Copy share the same block of 24 words of data RAM – they use it in turns)

Direct ASM Format: Interface Description

Module Terminal Variables			
Name	**Description**	**Format**	**Range**
Inputs			
SECTOR_CMD	Describes which sectors are to be erased. Bit 0 corresponds to sector 0, bit 1 to sector 1 etc. If the Bit in SECTOR_CMD is set then the Sector is ERASED. If the bit is a 0 then the sector is left alone.	integer	0x0000 to 0x000F.
Outputs			
ALGO_STATUS	Returns an error code reflecting the operation status. 0x0000: Operation was run to completion. 0x0002: The algorithm failed to erase the flash despite applying maximum number of pulses allowable. 0x000a: The algo was called with a zero sector mask, i.e. the algo was asked to erase 'no sectors'.	integer	

Module Usage / Calling Convention

Variable Declaration:
In the system file include the following statements:

```
.include var.h
```

Memory map:
All variables are mapped to a named section '.flshvar'
Code is mapped into the ERA_text and ESPL_text sections.

Special Variables

flashAlgoVars **External structure instance.**

```
.globl    flashAlgoVars

flshvar     .struct
ADDR        .int
PAD         .int
READ        .int
DATA        .int
PAD1        .int
PLS_CNT     .int
```

```
LASTVER     .int
FL_SECST    .int
FL_SECEND   .int
FL_CMD      .int
ERASESEC    .int
DATA_PTR    .int
FAIL_CMD    .int
SECTOR_KEY  .int
SECTOR_CMD  .int
ALGO_STATUS .int
flshvar_len .endstruct

flashAlgoVars   .tag    flshvar
```

The flashAlgoVars is assigned the characteristics of .flshvar structure.

Example code:

```
LDP     #flashAlgoVars.SECTOR_CMD         ;Point DP to flash vars.
SPLK    #0fh,flashAlgoVars.SECTOR_CMD     ;Pass sector command.
CALL    ERASE_FLASH                       ;Call the erase routine.
```

C / C-callable ASM Format: Interface Description

Module Terminal Variables

Module Terminal Variables			
Name	**Description**	**Format**	**Range**
Inputs			
SECTOR_CMD	Describes which sectors are to be erased. Bit 0 corresponds to sector 0, bit 1 to sector 1 etc. If the Bit in SECTOR_CMD is set then the Sector is ERASED. If the bit is a 0 then the sector is left alone.	integer	0x0000 to 0x000F.
Outputs			
ALGO_STATUS	Returns an error code reflecting the operation status. 0x0000: Operation was run to completion. 0x0002: The algorithm failed to erase the flash despite applying maximum number of pulses allowable. 0x000a: The algo was called with a zero sector mask, i.e. the algo was asked to erase 'no sectors'.	integer	

Special Constants and Datatypes

FlashAlgoVars **Data type for interfacing to the flash algorithm variables**

flashAlgoVars **External structure instance.**

The template for FlashAlgoVars is as below:

```
typedef  struct FlashAlgoVars {
int ADDR;
int PAD;
int READ;
int DATA;
int PAD1;
int PLS_CNT;
int LASTVER;
```

```
int FL_SECST;
int FL_SECEND;
int FL_CMD;
int ERASESEC;
int DATA_PTR;
int FAIL_CMD;
int SECTOR_KEY;
int SECTOR_CMD;
int ALGO_STATUS;
} FlashAlgoVars;
```

Module Functions: Erase

void extern eraseFlash(FlashAlgoVars *);

Module Usage

Structure Reference

```
extern FlashAlgoVars flashAlgoVars;
```

Calling the Erase function

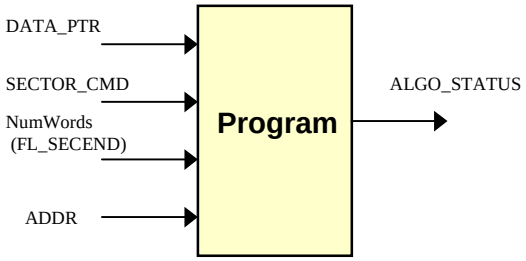
The following example calls the erase algo:

```
/* Setup the sector permissions */

 flashAlgoVars.SECTOR_CMD=(0xf);

/* Call the erase function */
 eraseFlash(&flashAlgoVars);

 if(0 != flashAlgoVars.ALGO_STATUS)
 {
/* Handle failure */
 }
```

3.4 PROGRAM

Description

Programs data into the flash sector(s).



Availability

This module is available with two interfaces:

The direct mode assembly-only interface (Direct ASM)
The C-callable interface version.

Module Properties

Type: Target Dependent, Application Dependent
Target Device/s: LF2402, LF2406, LF2407
File Name(s): pgm_alg.asm, var.h, var.asm, rundefs.h, timings.xx.
C-callable Version File Name(s): Cflash.h

Item	Asm only	C callable ASM	Comments
Code size	200 words	200 words	
Data RAM	24 words	24 words	
XDAIS ready	No	No	
XDAIS component	No	No	Hardware drivers.
Multiple Instances	No	No	

(Note: The Clear, Erase, Program and Copy share the same block of 24 words of data RAM – they use it in turns)

Direct ASM Format: Interface Description

Module Terminal Variables			
Name	**Description**	**Format**	**Range**
Inputs			
DATA_PTR	Address of the buffer which contains the data to be programmed into the flash memory.	integer	any valid data memory area.
FL_SECEND	Number of words to be programmed into the flash.	integer	0<N<BufferSize
ADDR	Address in Flash memory where the data is to be programmed.	integer	any valid flash memory area.
SECTOR_CMD	Describes which sectors contain the datum to be programmed into the flash. The locations which are to be programmed must already contain 1s (erased bits). Bit 0 corresponds to sector 0, bit 1 to sector 1 etc. If the Bit in SECTOR_CMD is set then the words can be programmed into the flash.	integer	0x0000 to 0x000F.
Outputs			
ALGO_STATUS	Returns an error code reflecting the operation status. 0x0000: Operation was run to completion. 0x0003: The algorithm failed to program the flash despite applying maximum number of pulses allowable. 0x0004: The algorithm was asked to program a '1' into one or more bits already at a '0'.	integer	

Module Usage / Calling Convention

Variable Declaration:
In the system file include the following statements:

```
.include var.h
```

Memory map:
All variables are mapped to a named section '.flshvar'
Code is mapped into the PGM_text and PSPL_text sections.

Special Variables

flashAlgoVars **External structure instance.**

```
.globl    flashAlgoVars
```

```
flshvar     .struct
ADDR        .int
PAD         .int
READ        .int
DATA        .int
PAD1        .int
PLS_CNT     .int
LASTVER     .int
FL_SECST    .int
FL_SECEND   .int
FL_CMD      .int
ERASESEC    .int
DATA_PTR    .int
FAIL_CMD    .int
SECTOR_KEY  .int
SECTOR_CMD  .int
ALGO_STATUS .int
flshvar_len .endstruct

flashAlgoVars   .tag    flshvar
```

Thus the flashAlgoVars is instanced and assigned the characteristics of flshvar structure.

Example code:

```
LDP    #flashAlgoVars.DATA_PTR          ;Point DP to flash vars.
SPLK   #300h, flashAlgoVars.DATA_PTR    ;Setup ptr to data buffer.
SPLK  #0eh,  flashAlgoVars.SECTOR_CMD   ;Pass sector command.
SPLK  #4,    flashAlgoVars.FL_SECEND    ;Number of words to pgm.
SPLK  #1023h,flashAlgoVars.ADDR         ;Pass sector command.
CALL  PROGRAM_FLASH                     ;Call the program routine.
```

C / C-callable ASM Format: Interface Description

Module Terminal Variables

Module Terminal Variables			
Name	**Description**	**Format**	**Range**
Inputs			
DATA_PTR	Address of the buffer which contains the data to be programmed into the flash memory.	integer	any valid data memory area.
FL_SECEND	Number of words to be programmed into the flash.	integer	0<N<BufferSize
ADDR	Address in Flash memory where the data is to be programmed.	integer	any valid flash memory area.
SECTOR_CMD	Describes which sectors contain the datum to be programmed into the flash. The locations which are to be programmed must already contain 1s (erased bits). Bit 0 corresponds to sector 0, bit 1 to sector 1 etc. If the Bit in SECTOR_CMD is set then the words can be programmed into the flash.	integer	0x0000 to 0x000F.
Outputs			
ALGO_STATUS	Returns an error code reflecting the operation status. 0x0000: Operation was run to completion. 0x0003: The algorithm failed to program the flash despite applying maximum number of pulses allowable. 0x0004: The algorithm was asked to program a '1' into one or more bits already at a '0'.	integer	

Special Constants and Datatypes

FlashAlgoVars **Data type for interfacing to the flash algorithm variables**

flashAlgoVars **External structure instance.**

The template for FlashAlgoVars is as below:

```
typedef  struct FlashAlgoVars {
int ADDR;
int PAD;
int READ;
```

```
int DATA;
int PAD1;
int PLS_CNT;
int LASTVER;
int FL_SECST;
int FL_SECEND;
int FL_CMD;
int ERASESEC;
int DATA_PTR;
int FAIL_CMD;
int SECTOR_KEY;
int SECTOR_CMD;
int ALGO_STATUS;
} FlashAlgoVars;
```

Module Functions: Erase

void extern programFlash(FlashAlgoVars *);

Module Usage

Structure Reference

```
extern FlashAlgoVars flashAlgoVars;
```

Calling the Program function

The following example calls the program algo:

```
 flashAlgoVars.SECTOR_CMD=(SECTOR1+SECTOR2+SECTOR3);

/* Setup ptr to data buffer. */
    flashAlgoVars.DATA_PTR=(int)(&bufferArray);

/* Number of words to program */

    flashAlgoVars.FL_SECEND=0x4;

/* Block address */

 flashAlgoVars.ADDR=0x1023;

/* Call the program routine */

 programFlash(&flashAlgoVars);
```

3.5 COPY

Description

Copies the flash algorithms into RAM at runtime from their load addresses.
This module is implemented as four separate function calls. Each of these calls copies the clear, erase, program and kernel routines into the RAM block specified by the 'run' addresses from the load address.

Availability

This module is available with two interfaces:

The direct mode assembly-only interface (Direct ASM)
The C-callable interface version.

Module Properties

Type: Target Dependent, Application Dependent
Target Device/s: LF2402, LF2406, LF2407
File Name(s): copy_alg.asm, var.h, var.asm, rundefs.h, link.cmd
C-callable Version File Name(s): Cflash.h

Item	Asm only	C callable ASM	Comments
Code size	159 words	159 words	(includes the 2 words for KER_strt and KER_end)
Data RAM	24 words	24 words	
XDAIS ready	No	No	
XDAIS component	No	No	
Multiple Instances	No	No	

(Note: The Clear, Erase, Program and Copy share the same block of 24 words of data RAM – they use it in turns)

Direct ASM Format: Interface Description

Module Usage / Calling Convention

Variable Declaration:
In the system file include the following statements:

```
.include var.h
```

Memory map:

All variables are mapped to a named section '.flshvar'
Code is mapped into the .text section.

Special Variables

flashAlgoVars **External structure instance.**

```
;-------------------------------------------------------
; Define a template to put the counter variables in the
; flashAlgoVars block. This is unused at the time the
; copy is invoked, so the flashAlgoVars are not
; subject to trashing (since they are not yet created).
;-------------------------------------------------------

counterStruct  .struct

source      .int
dest        .int
counter     .int
temp        .int
counter_len     .endstruct
;-------------------------------------------------------
; Assign template to the flashAlgoVars block.
;-------------------------------------------------------
flashAlgoVars   .tag counterStruct
```

Thus the flashAlgoVars is instanced and assigned the characteristics of flshvar structure.
(The copy algo actually assigns the flashAlgoVars structure a private template and uses these locations as variables to complete the copy)
Example code:

```
        CALL    COPY_CLEAR_ALGO
        CALL    COPY_ERASE_ALGO
        CALL    COPY_PROGRAM_ALGO
        CALL    COPY_KERNEL
```

C / C-callable ASM Format: Interface Description

Special Constants and Datatypes

(The copy algo actually assigns the flashAlgoVars structure a private template and uses these locations as variables to complete the copy)

FlashAlgoVars **Data type for interfacing to the flash algorithm variables**

flashAlgoVars **External structure instance.**

The COPY module doesn't define a C structure for the flashAlgoVars. It is entirely implemented in assembly and the caller does not need to pass the copy routines any parameters. See the section on the linker command file for details on how the copy routine finds the algos and run addresses.

Module Functions: Erase

void copyClearAlgo(void);
void copyEraseAlgo(void);
void copyProgramAlgo(void);
void copyKernel(void);

Module Usage

Calling the copy functions

The following example copies and calls the clear and erase algorithms:

```
copyClearAlgo();
flashAlgoVars.SECTOR_CMD=(SECTOR1+SECTOR2+SECTOR3);
clearFlash(&flashAlgoVars);

copyEraseAlgo();
flashAlgoVars.SECTOR_CMD=(SECTOR1+SECTOR2+SECTOR3);
eraseFlash(&flashAlgoVars);
```