

Microcontrollers

ApNote

AP1629

: Additional file
AP162904.EXE available

In-System Programming of C163-16F Flash Devices

Siemens C163-16F 16-bit microcontrollers provide 128 KBytes Flash memory on-chip. This application note gives hints and examples for in-system programming of C163-16F Flash devices. In-system programming is supported by the Windows-based OTP/ Flash memory programming tool "Memtool".

Author : Peter Kliegelhöfer / HL DC AT Microcontroller Application Support

Contents	Page
1 Flash Memory Overview	3
2 Flash Memory Configuration	5
3 Flash Command Register	6
3.1 Handling of Flash addresses	7
4 Programming Examples	8
4.1 Example 1 „Read Flash Status“	8
4.2 Example 2 „Burst Write“	9
4.3 Example 3 „Sector Erase“	11
5 Flash operation control by using the Flash Status Register FSR	12
6 Memtool - The OTP/ Flash Memory Programming Tool	13
Appendix	15
A flash.h	15
B flash.c	17

AP1629 ApNote - Revision History		
Actual Revision : 07.98		Previous Revision : 04.98
Page of actual Rev.	Page of prev.Rel.	Subject changes since last release
7		Chapter "Handling of Flash Addresses" added
13-14	13-14	MEMTOOL.EXE and Flash/ OTP programming drivers updated

1. Flash Memory Overview

The C163-16F Flash devices provide 128 KBytes of electrically erasable and reprogrammable non-volatile flash EPROM on-chip for both instruction code and constant data.

The C163-16F flash module uses the standard 5 Volt power supply for all read and write / erase functions.

In standard read mode (the normal operating mode) the flash memory appears like an on-chip ROM with the same timing and functionality. Instruction fetches and data operand reads are performed with all addressing modes of the C16x instruction set.

All other operations besides normal read operations are initiated and controlled by protected but simple command sequences written to the flash address/ command register. In case of the command write addresses care must be taken because not all addressing modes of the C16x instruction set are allowed. Due to an integrated state machine flash memory programming and erase are directly controlled by commands. Therefore special algorithms for programming or erase and verify operations respectively are not required.

The flash status register (FSR) reflects the overall status of the flash module after reset and after reception of the different commands. Sector specific states are also indicated in the FSR. Note that the FSR is no real register (SFR or GPR) but is rather virtually mapped into the active address space of the flash memory.

The entire flash memory is divided into four sectors with the same size (32/ 32/ 32/ 32 KByte). This allows to erase each block separately, when only parts of the flash memory need to be reprogrammed.

The flash module provides a burst mechanism for data write operations which allows to collect 32 words in an assembly burst register before being written to flash in one programming cycle.

The flash module offers an access time of 60 ns, allowing operation of CPU with 25 MHz and without wait states. Programming typically takes 1 ms per burst (effectively 32 μ s per word), sector erasing typically takes 10 ms. The flash memory features a typical endurance of more than 1000 erase/ programming cycles. Erased flash memory cells contain all '0's.

The flash memory can be programmed both in an appropriate programming board (not available yet) and in the target system which provides a lot of flexibility. As long as the C163-16F Flash has no on-chip (or in the flash memory pre-programmed) bootstrap loader may it is necessary to load and start the programming code from external memory or to program the flash before soldering¹). Any code that programs or erases flash memory locations must be executed from memory outside the on-chip flash memory itself (on-chip RAM or external memory).

The lower 32 KBytes of the on-chip flash memory of the C163-16F Flash can be mapped to either segment 0 (00'0000_H to 00'7FFF_H) or segment 1 (01'0000_H to 01'7FFF_H) during the initialization phase to allow external memory to be used for additional system flexibility. The upper 96 KBytes of the on-chip flash memory are assigned to locations 01'8000_H to 02'FFFF_H.

1 please refer to ApNote AP1638 "Bootstrap Loader on C163 Flash" and to the actual status sheet

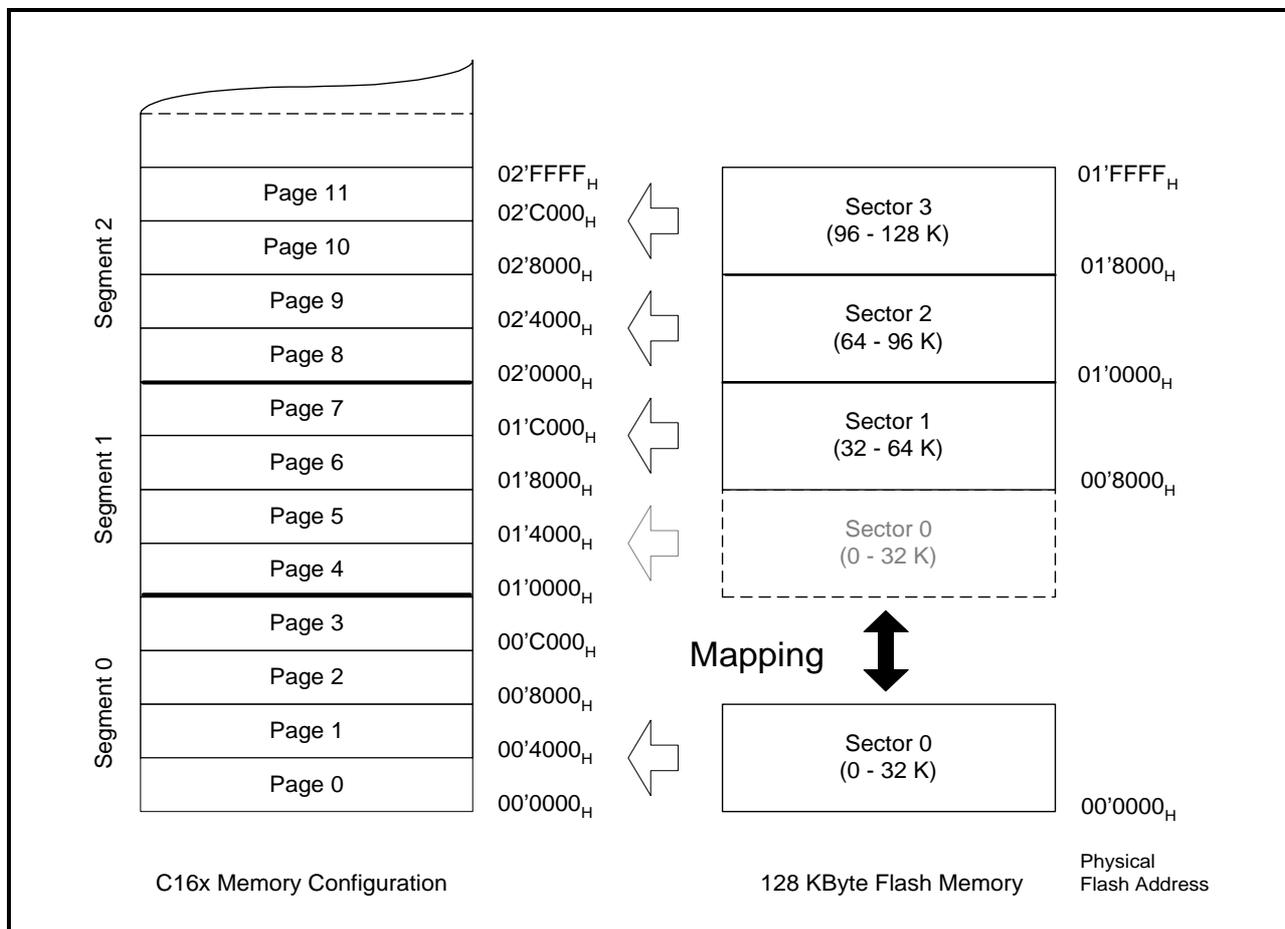


Figure 1
Flash Memory Overview

2 Flash Memory Configuration

Upon reset the default memory configuration of the C163-16F Flash is determined by the state of its \overline{EA} pin. When \overline{EA} is low the internal flash memory is disabled and the startup code is fetched from external memory.

In order to access the on-chip flash memory after booting from external memory the internal flash memory must be enabled via software by setting bit ROMEN in register SYSCON. The lower 32 KBytes of the flash memory can be mapped to segment 0 or segment 1, controlled by bit ROMS1 in register SYSCON. Mapping to segment 1 preserves the external memory containing the startup code, while mapping to segment 0 replaces the lower 32 KBytes of the external memory with on-chip flash memory. In this case a valid vector table must be provided.

As the on-chip flash memory covers more than segment 0, segmentation should be enabled (by clearing bit SGTDIS in register SYSCON) in order to map the whole internal flash into the address space.

Whenever the internal memory configuration of the C163-16F Flash is changed (mapping, enabling, disabling) the following procedure must be used to ensure correct operation:

- Configure the internal flash memory as required
- Execute an inter-segment branch (JMPS, CALLS, RETS)
- Reload all four DPP registers

Note:

Instructions that configure the internal flash memory can only be executed from internal RAM or from external memory, not from the flash itself.

Register SYSCON can only be modified **before** the execution of the EINIT instruction.

Note:

For detailed informations concerning the handling of internal non-volatile memory please refer to the users manual, chapter „System Programming“ (Handling the Internal ROM/ Pits, Traps and Mines).

3 Flash Address/ Command Register

Flash operations are selected by writing specific address and data sequences to the address/command registers. Writing incorrect address and data values or writing them in the improper sequence will reset the module and set the command sequence error flag SQER in the status register. The valid command sequences are shown in figure 2.

Command Sequence	1. Cycle		2. Cycle		3. Cycle		4. Cycle		5. Cycle		6. Cycle	
	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Reset to Read	AAAA	xxF0	(RA)	(RD)	(RA)	(RD)	(RA)	(RD)	(RA)	(RD)	(RA)	(RD)
Enter Burst Load	AAAA	xx50	WA	1. WD	No RD							
Load Burst Data	A0F2	WD	No RD									
Store Burst	AAAA	xxAA	5554	xx55	AAAA	xxA0	WA	32. WD				
Erase Sector	AAAA	xxAA	5554	xx55	AAAA	xx80	5554	xxAA	AAAA	xx55	SA	xx30
Read Flash Status	AAAA	xxFA	SA	Status								
Clear Status	AAAA	xxF5										

Default: Addresses and data values which belong to a write command cycle

RA Read address; memory address of read data

RD Data read from location RA during a read operation

WA Write address; address of memory location to be programmed

WD Data to be loaded into burst assembly register before being stored (written) to flash memory

SA Address to the sector to be erased or to be sampled for its status

x Irrelevant

 Read Access

Figure 2
Command Sequence Table

3.1 Handling of Flash Addresses

All flash command, sector and data register addresses have to be located within the active flash memory space. The active space is that address range to which the physical flash addresses are mapped as defined by the user.

When using data page pointers (DPPs) for command or sector addresses one has to make sure that both MSBs of the command or sector address are reflected in both LSBs of the selected DPP.

Only *register-indirect* addressing like “mov [Rx], Ry”, “mov Rx, [Ry]” can be used for command, sector or write-data accesses. Direct addressing is **not allowed**. Make sure that the C-compiler doesn't use direct addressing.

Example:

By using data page pointers (DPPs):

```
MOV  DPP1, 09h      ; adjust data page pointers according to the addresses
MOV  DPP2, 0Ah
MOV  Rwm, #ADDRESS ; ADDRESS could be a dedicated unlock sequence address
                          ; (AAAAh, 5554h...) or the flash write address
MOV  Rwn, #DATA    ; DATA could be a dedicated unlock sequence data
                          ; (xxF0h, xx50h...) or data to be loaded into the assembly register
MOV  [Rwm], Rwn   ; indirect addressing
```

By using the extended segment (EXTS) instruction:

```
MOV  Rwm, #ADDRESS ; ADDRESS could be a dedicated unlock sequence address
                          ; (AAAAh, 5554h...) or the Flash write address
MOV  Rwo, #DATA    ; DATA could be a dedicated unlock sequence data
                          ; (xxF0h, xx50h...) or data to be loaded into the assembly register
MOV  Rwn, #SEGMENT ; the value of SEGMENT represents the sector number and could be
                          ; in our case 0, 1 or 2 (depending on sector mapping)
EXTS Rwn, #LENGTH ; the value of Rwn determines the 8-bit segment valid for the
                          ; corresponding data access for any long or indirect address in
                          ; the EXTS instruction. LENGTH defines the length of the effected
                          ; instruction sequence and has to be a value between 1...4
                          ; (see instruction manual)
MOV  [Rwm], Rwo   ; indirect addressing
```

When performing *read* cycles within flash commands (e.g. “read Flash Status Register”) it is **not allowed** to use indexed addressing mode, which is only available with instructions “mov Rx, [Ry+#data16]” and “movb Rx, [Ry+#data16]”. Make sure that the C-compiler doesn't use these instructions.

Note:

The EXTS instruction must be used carefully. Please refer to the instruction set manual.

4 Programming Examples

The following examples are written in „assembler“ - why ?

Most of the microcontroller programs are written in „C“ language where the data page pointers are automatically set by the compiler. However it has to be taken into account that the C compiler may use not allowed direct addressing for flash address/ command register operations.

In this case it is necessary to perform the flash address/ command register accesses (command sequences) with assembler in-line routines which use indirect addressing.

Note:

In the appendix you find runnable flash routines, written in „C“ and assembler in-line (Tasking).

These flash routines are only *examples*, they are no subject of the OTP/ Flash memory programming tool „Memtool“.

4.1 Example 1: Performing the command „Read Flash Status“

We assume that in the initialization phase the lowest 32K of flash memory (sector 0) have been mapped to segment 1.

According to the usual way of C16x data addressing with data page pointers, address bits A15 and A14 of a 16-bit command write address select the data page pointer (DPP) which contains the upper 10 bits for building the 24-bit physical data address. Address bits A13...A0 represent the address offset. In our case the command write address $AAAA_h$ ($1010\ 1010\ 1010\ 1010_b$) selects data page pointer 2 (DPP2).

We have to make sure, that

- DPP2 points to active flash memory space,
- address bits A15 and A14 are reflected in the both lower bits of DPP2 (as mentioned above).

To be independent of mapping of sector 0 we choose for **all** DPPs which are selected by the both MSBs of the command write addresses values which point to segment 2.

For this reason we load DPP2 with value 0Ah ($00\ 000\ 1010_b$).

The flash status register „FSR“ reflects the overall status of the flash module after Reset and after reception of the different commands. A sector specific state (“sector erased”) is also indicated in the FSR. Therefore we have to decide from which flash sector we like to get specific state information. In our example the FSR indicates the specific state of sector 0.

```

mov    r2, #0AAAAh      ; load auxiliary register r2 with command address (used in cycle 1)
mov    dpp0, #04h       ; data page pointer 0 (used in cycle 2) determines the sector
                          ; 04h = sector 0, 06h = sector 1, 08h = sector 2, 0Ah = sector 3
mov    dpp2, #0ah       ; pointer to flash / segment 2 (used in cycle 1)
mov    r4, #0FAh        ; load auxiliary register r4 with command data (used in cycle 1)
mov    [r2], r4         ; write command data to dedicated command address (cycle 1)

```

```

mov   r4, #00h           ; load auxiliary register r4 with address to the sector to be sampled
                           ; for its status (used in cycle 2). Address 0000h selects DPP0.
                           ; Thus DPP0 determines the sector of which FSR indicates specific
                           ; states.
mov   r12, [r4]          ; read FSR and write the result to register r12 (cycle 2)

```

In the example above the 16-bit registers r2 and r4 are used as auxiliary registers for indirect addressing.

Note:

For detailed informations concerning command sequences, sector addressing and register addresses please refer to the Data Sheet „C163-16F“.

4.2 Example 2: Performing a „burst write“ to the flash memory

We assume that in the initialization phase the lowest 32K of flash memory (sector 0) have been mapped to segment 1. In our example we assume further that a 32-word buffer „BUFFER“ (located somewhere in internal or external memory) contains the data which should be programmed to flash memory. Finally the data will be written to address 01'0000_H (flash sector 0, segment 1).

Writing data to flash memory is basically performed in three steps:

- Executing the „Enter Burst Load“ command and loading of first data word into the assembly register.
- Executing the „Load Burst Data“ command and consecutive loading of the next 30 data words into the assembly register
- Executing the „Store Burst“ command and loading of the last data word into the assembly register

After the last data word (32nd) is written to the assembly register this complete register is automatically programmed to flash memory.

```

mov   dpp0,#04h          ; segment 0, Page 4, flash sector 0
mov   dpp1,#09h          ; pointer to flash/ segment 2
mov   dpp2,#0Ah          ; pointer to flash/ segment 2
mov   r0,#0AAAAh        ; registers r0, r1, r3 are loaded with dedicated command write
                           ; addresses

mov   r1,#0A0F2h
mov   r3,#05554h
mov   r6,#0h             ; assembly register start/ flash store address
                           ; IMPORTANT: The assembly register address is defined only by
                           ; address bits A5...A0. Upper bits are not evaluated.
                           ; A5...A0 have to be zero values !!!
                           ; The flash store address is defined only by bits A16...A6. Lower
                           ; bits are not evaluated. Since both addresses use different

```

```

; address bits only one register can be used. In this case take
; care that the flash store address is either zero, 40h or a multiple
; of 40h (according to a burst width) to avoid trouble with the
; assembly register address.

mov    r9,#pof buffer    ; page offset of BUFFER (location of data to be programmed)
mov    r10,#pag buffer   ; page number of BUFFER (location of data to be programmed)
mov    r4,#50h
mov    [r0],r4           ; ENTER BURST LOAD (1st cycle)
extp   r10,#01h
mov    r2,[r9]           ; get first data word from BUFFER
add    r9,#02h           ; increment pointer by 2
mov    [r6],r2           ; write first data word into the assembly register
mov    r11,#29           ; load counter value for the next 30 data words
wrt_loop:
extp   r10,#01h
mov    r2,[r9]           ; get next data words (2....31) from BUFFER
add    r9,#02h           ; increment pointer by 2
mov    [r1],r2           ; LOAD BURST DATA. Write data words 2...31 into the assembly
; register. Always write to the same address: A0F2. For every
; sequential word load this address is internally incremented by 2

cmpd1  r11,#0
jmp    cc_ugt,wrt_loop

mov    r4,#0aah          ; STORE BURST
mov    [r0],r4           ; cycle 1
mov    r4,#055h
mov    [r3],r4           ; cycle 2
mov    r4,#0a0h
mov    [r0],r4           ; cycle 3
extp   r10,#01h
mov    r2,[r9]           ; get last data word (32nd) from BUFFER
add    r9,#02h
mov    [r6],r2           ; cycle 4: write last data word into the assembly register.
; Afterwards the complete assembly register is programmed
; automatically to flash memory.

...
;

wait_wrt:
mov    r4, #0fah         ; read flash status register FSR (see example 1) and check
; BUSY bit
mov    [r0], r4         ; cycle 1
mov    r4, #00h
mov    r12, [r4]        ; Read FSR (cycle 2)
mov    r4, #01h
and    r4, r12          ; check BUSY bit
jmp    cc_nz,wait_wrt

```

Note:

Multiple writes to the same flash location before erase are not allowed.

4.3 Example 3: Performing the command „Erase Sector“

We assume that in the initialization phase the lowest 32K of flash memory (sector 0) have been mapped to segment 1. In our example sector 1 will be erased.

```

mov    r0, #0AAAAh      ; load auxiliary registers r0 and r1 with dedicated command write
                          ; addresses

mov    r1, #05554h
mov    r8, #0h           ; flash sector erase address. Address 0h selects DPP0.
mov    dpp0, #06h       ; data page pointer 0 determines the sector to be erased
                          ; 04h = sector 0, 06h = sector 1, 08h = sector 2, 0Ah = sector 3

mov    dpp1, #09h       ; pointer to flash / segment 2
mov    dpp2, #0ah       ; pointer to flash / segment 2
mov    r4, #0aah        ; execute command „Erase Sector“
mov    [r0], r4         ; cycle 1
mov    r4, #055h
mov    [r1], r4         ; cycle 2
mov    r4, #080h
mov    [r0], r4         ; cycle 3
mov    r4, #0aah
mov    [r1], r4         ; cycle 4
mov    r4, #055h
mov    [r0], r4         ; cycle 5
mov    r4, #030h
mov    [r8], r4         ; cycle 6
mov    r6, #0h          ; wait while busy, but max. 1'0000h loops
wait:
sub    r6, #01h
jmp    cc_z, error      ; timeout error: enter specific failure routine
mov    r4, #0FAh        ; read FSR and poll BUSY flag
mov    [r0], r4        ; cycle 1
mov    r4, #00h
mov    r5, [r4]         ; cycle 2
jb     R5.0, wait
mov    r4, #0FAh        ; Read FSR and check error flags
mov    [r0], r4        ; cycle 1
mov    r4, #00h
mov    r5, [r4]         ; cycle 2
and    r5, #0EFh        ; mask low byte
cmp    r5, #0h
jmp    cc_nz, failed    ; error: enter specific failure routine

```

Note:

For detailed informations concerning the flash status register FSR and operation control and error handling please refer to the Data Sheet „C163-16F“

5 Flash operation control by using the Flash Status Register FSR

Following a recommendation for a flash operation control process:

- Write command sequence to Flash.
- Check for correct command sequence by sampling the SQER bit in status register FSR.
Note: This is mainly recommended in case of program evaluation.
- Check if the command is (still) in operation by sampling the BUSY bit in status register FSR. Because BUSY summarizes the states of ERASE and PROG an additional evaluation of these bits is not necessary.
Note: Separate evaluation of ERASE and PROG might could be helpful in case of program evaluation.
Note: If BUSY should be permanently set after a flash operation (which means longer than 100 ms) only a system (CPU) reset (do not mix up with the Reset to Read *command*!) can reset the BUSY flag. As long as BUSY is set, no other flash command can be executed.
- Check the error flags BUER and VPER when a flash operation is finished.
In most of all cases these flags can (and in one case *must*) be ignored, because
 - OPER (Operation Error) is not reliable and **must not** be evaluated,
 - BUER (Burst Error) is mainly helpful for program evaluation and
 - VPER (Voltage Error) indicates voltage break (< 2,3V and > 100ns).Note: A very simple error check after a flash operation could be to mask out the FSR low byte (ANDing with “00EFh”) and to compare the result with zero.
- In case of an indicated fault condition: clear the error flag with a Clear Status or a Reset to Read command and start a specific SW reaction.
Note: If you start a retry operation after a faulty programming attempt, be aware that multiple writes to the same flash location before erase are not allowed.

6 Memtool - The OTP/ Flash Memory Programming Tool

In-system programming of on-chip OTP- and Flash memory is supported by the Windows-based programming tool „Memtool“ which is freely available. Siemens provides driver updates on the Internet; please ask for the current status. Memtool is an application example as well as a programming tool for on-chip OTP- and Flash memory, supporting C161CI-32F, C167CR-16F, C167CS-32F and C163-16F Flash devices and C164CI-8E OTP devices.

Thanks to the modular structure Memtool is easily expandable for future C16x devices. For an interested user all driver sources are also available (figure 2). Along with the drivers come readme.txt files which contain informations about the hardware requirements and latest informations about Memtool and each driver.

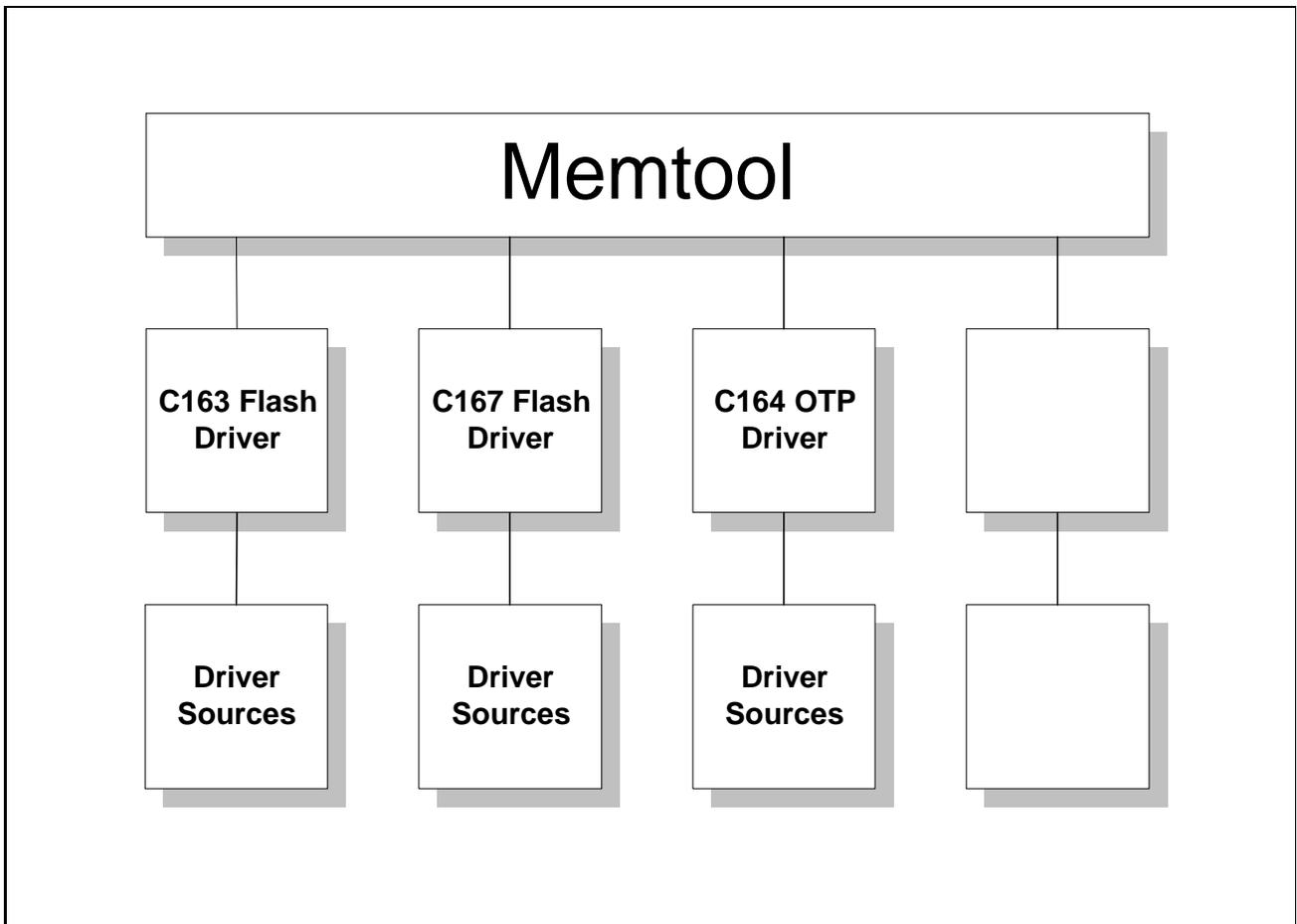


Figure 3
The structure of Memtool

Memtool allows programming and erasing the C163-16F flash memory employing an Ertec EVA163 evaluation board or a Phytex C163 starter kit.

The process of programming is based on the bootstrap loader (BSL) which resides on external EPROM/ Flash memory of the Ertec EVA163 evaluation board/ C163 Phytex starter kit. Be aware that the C163-16F Flash has no internal test ROM and therefore no internal bootstrap loader. In the future C163-16F Flash devices will be available with a pre-programmed BSL routine in the *internal flash memory*. Please refer to application note AP1638 "Bootstrap Loader on C163 Flash" and to the actual status sheet.

All software including the programming data is downloaded from a host PC into the internal RAM of the microcontroller. The application requires only 1 KBytes of internal RAM, external RAM is not required. Since the C163-16F Flash is a 5 Volt-only device, no additional voltage for flash programming or erase is required.

In order to download the application from a PC, a serial link has to be established. The C163-16F already provides an asynchronous serial interface that only has to be connected to COM1 or COM2 of your PC. Supposing you are using a Phytex C163 starter kit, you can directly connect its serial connector with the COM1/ COM2 interface of your PC.

Appendix A

```

/*****
/* SIEMENS HL MI E EM (c) Copyright SIEMENS AG 1997 */
/* */
/* FILE: FLASH.H */
/* C163 Flash CPU mode write routines */
/* */
/* These Tasking C small model routines expect the internal flash */
/* mapped to 1'0000H - 2'FFFFH and are designed to be executed from */
/* an external program memory. */
/* Define Macro -DINT_DISEAB in command line if interrupts should be */
/* disabled during write/ erase. Use BUSY command options with care. */
/* */
/* V1.0 R. Ullmann 17.2.97 */
*****/

#ifndef _FLASH
#include <reg163.h>

/*----- General Definitions -----*/

typedef unsigned char BYTE;
typedef unsigned int WORD;
typedef unsigned long DWORD;

#define ICPUFL_SECT_SIZE 0x08000L
#define ICPUFL_NR_OF_SECT 0x04
#define ICPUFL_BURST_SIZE0x40
#define ICPUFL_SIZE ICPUFL_NR_OF_SECT*ICPUFL_SECT_SIZE)
/* useful flash constant definitions */

extern WORD huge wptr_icpufl [ICPUFL_SIZE/sizeof(WORD)];
/* word array declared at flash location (see FLASH.C) */

#define HPTR_ICPUFL_SECT(sect)(((BYTE huge *)wptr_icpufl)+(sect)*ICPUFL_SECT_SIZE)
/* macro return pointer to the beginning of a flash sector 0 ... 3 */

#define B_WAIT 0
#define B_BUSY 0xFF
/* constant data to be passed at busy flash command arguments */

#define C163FL_STAT_BUSY 0x0001
#define C163FL_STAT_PRG 0x0002
#define C163FL_STAT_ERASE 0x0004
#define C163FL_STAT_BRST 0x0008
#define C163FL_STAT_OPER 0x0010
#define C163FL_STAT_VPER 0x0020
#define C163FL_STAT_SQER 0x0040
#define C163FL_STAT_BUER 0x0080
#define C163FL_STAT_ERASED 0x8000
/* flash status register (FSR) bits */

#define ICPUFL_FSR 0x00C0
/* FSR byte offset address within sector */

#define E_OK 0x0000 /* no error */
#define E_SQER 0x0020 /* C163 Flash Sequence Error */
#define E_OPER 0x0030 /* C163 Flash Operation Error */
#define E_VPER 0x0040 /* C163 Flash Voltage Error */
#define E_BUER 0x0050 /* C163 Flash Burst Error */
#define E_TMO 0x0060 /* TimeOut Error, Operation suspended */
#define E_ARG 0x0070 /* Argument Error */
/* error codes, returned by flash commands' return WORD */

```

```
/*----- Flash Commands -----*/  
  
extern WORD fl_init (void);  
/* initialization for flash usage, must be called before _einit() */  
  
extern WORD fl_read_status (WORD far * fptr_val_dest, void far * fptr_fl_sect);  
/* read flash status register of the sector, to which fptr_fl_sect points to  
and store it to fptr_val_dest. (the busy flag in FSR is sector specific!) */  
  
extern WORD fl_clear_status (void);  
/* perform clear status command */  
  
extern WORD fl_reset2read (void);  
/* perform reset to read command */  
  
extern WORD fl_write_burst (WORD far*fptr_fl_dest,WORD far*fptr_source,BYTE busy);  
/* Take 64 Byte burst data beginning at specified  
source address and program it to fl_dest. */  
/* Source data must lie within one page,  
destination address must be a multiple of 40H. */  
/* If BUSY != 0 function polls status until ready or functional timeout.  
If BUSY == 0 function returns and checks no status. */  
/* Written for SMALL memory modell */  
  
extern WORD fl_erase_sector (void far * fptr_fl_sect, BYTE busy);  
/* erase the flash sector (32 kB), into which fptr_fl_sect points  
if BUSY != 0 function polls status until ready or functional timeout  
if BUSY == 0 function returns and checks no status */  
  
extern WORD fl_wait_busy_tmo (void far * fptr_fl_sect);  
/* polls status until ready or functional timeout */  
  
#define _FLASH  
#endif
```

Appendix B

```

/*****
/* SIEMENS HL MI E EM (c) Copyright SIEMENS AG 1997 */
/* */
/* FILE: FLASH.C */
/* C163 Flash CPU mode FLASH write routines */
/* */
/* These Tasking C small model routines expect the internal flash */
/* mapped to 1'0000H - 2'FFFFH and are designed to be executed from */
/* an external program memory. */
/* Define Macro -DINT_DISEAB in command line if interrupts should be */
/* disabled during write/erase. Use BUSY command options with care. */
/* */
/* V1.0 R. Ullmann 17.2.97 */
/*****

#include „flash.h“

#define FL_TMO_VAL 800000L/5 /* Timeout repeat value max. 0.8s */

/*----- declarations -----*/

#pragma save_attributes

#pragma combine hb=A0x10000
/* C163 CPU Flash is a huge byte segment and starts in segment 1, page 4 */

#pragma class hb=MFLASH
/* declare dedicated flash memory class */

WORD huge wptr_icpufl [ICPUFL_SIZE/sizeof(WORD)];
/* declare flash memory as huge word array */

#pragma restore_attributes

/*----- initialization -----*/

WORD fl_init (void)
{
    SYSCON = 0x1584;
    /* map and enable internal RAM, function must be ececuted before _einit() */
    return E_OK;
}

/*----- read status -----*/

WORD fl_read_status (WORD far * fptr_val_dest, void far * fptr_fl_sect)
{
    /* read flash status register of the sector, to which fptr_fl_sect points to
    and store it to fptr_val_dest. (the busy flag in FSR is sector specific!) */

    WORD register seg, bseg, fsrsof, stat;

    seg = _seg(fptr_fl_sect);/* determine segment number -> SA */

    fsrsof =ICPUFL_SECT_SIZE * (_sof(fptr_val_dest)/ICPUFL_SECT_SIZE) + ICPUFL_FSR;
    /* determine flash status register align to start of sector */

    bseg = _seg(wptr_icpufl); /* base segment of flash, usually 1 */
    /* seg, fsrsof and bseg are passsed in arbitrary registers @1. @2, @3 to inline
    assembler routine, stat is returned in an arbitrary register @4 */

    #pragma asm (@1=seg, @2=fsrsof, @3=bseg, @4)
        MOV R7,#0FAH
        MOV R8,#0AAAAH

```

```

    EXTS @3,    #01H
    MOV [R8], R7      ; write to area command register via EXTS-access
                      ; directed to flash base segment

    EXTS @1,    #01H
    MOV @4, [@2]     ; read FSR (register-indirect!) at fsrofs via EXTS-access
                      ; within specified sector
#pragma endasm (stat=@4)

    * fptr_val_dest = stat; /* write FSR value to destination */
    return E_OK;
}

/*----- clear status -----*/

WORD fl_clear_status (void)
{
    /* perform clear status command */
    WORD register seg;

    seg = _seg(wptr_icpufl); /* base segment of flash, usually 1 */

    #pragma asm (@1=seg)
        MOV R7,#0F5H
        MOV R8,#0AAAAH
        EXTS @1,#01H
        MOV [R8], R7      ; write to area command register via EXTS-access
                          ; directed to flash base segment
    #pragma endasm

    return E_OK;
}

/*----- reset to read -----*/

WORD fl_reset2read (void)
{
    /* perform reset to read command */
    WORD register seg;

    seg = _seg(wptr_icpufl); /* base segment of flash, usually 1 */

    #pragma asm (@1=seg)
        MOV R7,#0F0H
        MOV R8,#0AAAAH
        EXTS @1,    #01H      ; write to area command register via EXTS-access
                          ; directed to flash base segment

        MOV [R8], R7
    #pragma endasm

    return E_OK;
}

/*----- write burst -----*/

WORD fl_write_burst (WORD far * fptr_fl_burst,WORD far * fptr_source, BYTE busy)
{
    /* take 64 Byte burst data beginning at specified
       source address and program it to fl_dest */
    /* source data must lie within one page, destination
       address is a multiple of 40H. */
    /* written for SMALL memory modell */
    /* if BUSY != 0 function polls status until ready or functional timeout */
    /* if BUSY == 0 function returns and checks no status */

#ifdef INTDISEAB
    /* if interrupt vector table is located to flash during programming, interrupts

```

```

must be disabled ! */
bit int_en;
#endif

WORD register seg, bpag, bpofs, spag, spofs;

seg = _seg(wptra_icpufl); /* base segment of flash, usually 1 */
bpag = _pag(fptra_fl_burst); /* page of flash destination address */

bpofs = ICPUFL_BURST_SIZE * (_pof(fptra_fl_burst)/ICPUFL_BURST_SIZE);
/* page offset of flash destination address, automatically adjusted
to next allowable burst start address. As DPP0 will be used to address
destination bits 14 and 15 remain zero */

spag = _pag(fptra_source); /* source data start page */

spofs = _pof(fptra_source) | 0x8000;
/* source data start page offset, as source data will be addressed via DPP2,
bit 15 is set to select DPP2 */

#ifdef INTDISEAB
/* disable interrupts */
int_en=IEN;
IEN=0;
#endif

#pragma asm (@1=seg, @2=bpofs, @3=bpag, @4=spofs, @5=spag)
MOV R7,#050H
MOV R8,#0AAAAH
EXTS @1, #01H
MOV [R8], R7 ; write first enter burst-command (register-indirect) using
; EXTS-access to flash area command register AAAAH in flash
; base segment

MOV DPP2, @5 ; set DPP2 to write data source page
MOV DPP0, @3 ; set DPP0 to flash destination page
; assure no pipeline problems by this command sequence !
MOV R7, [@4+] ; fetch first source data word relative to DPP2, increment source
; pointer
MOV [@2], R7 ; write (register-indirect)to flash write address (WA)
; relative to DPP0

MOV R8,#0A0F2H
MOV R9,#30 ; load assembly buffer with 30H words (loop) from source
burst1:
MOV R7, [@4+] ; fetch source data relative to DPP2, increment pointer
EXTS @1, #01H
MOV [R8], R7 ; write to flash assembly register using EXTS-access
SUB R9, #01H
JMP CC_NZ, burst1 ; end loop „enter burst data“

MOV R8,#0AAAAH ; store burst sequence, EXTS accesses
MOV R9,#05554H

MOV R7, #0AAH
EXTS @1, #03H
MOV [R8],R7
MOV R7, #55H
MOV [R9],R7
MOV R7, #0A0H
EXTS @1, #01H
MOV [R8],R7

MOV R7,@4+ ; fetch last data word via DPP2
MOV [@2],R7 ; write last data word via DPP0

MOV DPP2, #PAG ?BASE_DPP2 ; reload DPPs

```

```

MOV DPP0, #PAG ?BASE_DPP0 ; memorymodell-dependent!
#pragma endasm

if (busy)
    seg = E_OK;
    /* do not wait for flash return to read mode */
    /* may be dangerous if interrupt vector table points into flash */
    /* time might be used for some jobs but flash is busy and cannot */
    /* be read/ executed from! */
else
    seg = fl_wait_busy_tmo (fptr_fl_burst);
    /* wait for flash returns to read mode */

#ifdef INTDISEAB
    /* reenable interrupts */
    IEN = int_en;
#endif

return seg;
}

WORD fl_erase_sector (void far * fptr_fl_sect, BYTE busy)
{
    /* erase the flash sector (32 kB), into which fptr_fl_sect points */
    /* if BUSY != 0 function polls status until ready or functional timeout */
    /* if BUSY == 0 function returns and checks no status */

#ifdef INTDISEAB
    /* if interrupt vector table is located to flash, during programming,
    interrupts must be disabled ! */
    bit int_en;
#endif

WORD register fseg, fsof, bseg;

fseg = _seg(fptr_fl_sect);
/* segment of the flash sector to be erased, usually 1 or 2 */
fsof = ICPUFL_SECT_SIZE*(_sof(fptr_fl_sect)/ICPUFL_SECT_SIZE);
/* sector segment offset, aligned to sector size, 0H or 8000H */
bseg = _seg(wptr_icpufl); /* flash base segment, usually 1 */

#ifdef INTDISEAB
    /* disable interrupts */

int_en=IEN;
IEN=0;
#endif

#pragma asm (@1=fseg, @2=fsof, @3=bseg)
MOV R7,#0AAH
MOV R8,#0AAAAH
MOV R9,#05554H
EXTS @3, #01H
MOV [R8], R7
MOV R7,#055H
EXTS @3, #01H
MOV [R9], R7
MOV R7,#080H
EXTS @3, #01H
MOV [R8], R7
MOV R7,#0AAH
EXTS @3, #01H
MOV [R9], R7
MOV R7,#055H
EXTS @3, #01H
MOV [R8], R7

```

```

; erase sector sequence written to flash area command registers,
; using register-indirect (!) EXTS-accesses

    MOV R7, #030H
    EXTS @1, #01H
    MOV [@2], R7 ; write to sector address (SA) using register-indirect EXTS-access

#pragma endasm
    if (busy)
        fseg = E_OK;
        /* do not wait for flash return to read mode */
        /* may be dangerous if interrupt vector table points into flash */
        /* time might be used for some jobs but flash is busy and cannot
           be read/ executed from! */
    else
        fseg = fl_wait_busy_tmo (fptr_fl_sect);
        /* wait for flash returns to read mode */

#ifdef INTDISEAB
    /* reenables interrupts */

    IEN = int_en;
#endif
    return (fseg);
}

/*----- poll flash status while busy -----*/

WORD fl_wait_busy_tmo (void far * fptr_fl_sect)
{
    /* help routine, used to poll FSR until busy-state disappears and for
       error diagnosis */
    /* Pointer fptr_fl_sect must point into the flash sector which is expected to be
       busy or might report a hardware problem */

    WORD status;
    DWORD icpu_tmo;

    icpu_tmo=0;

    do
    {
        fl_read_status (&status, fptr_fl_sect); /* read FSR of sector */
    }
    while((status & C163FL_STAT_BUSY) && (++icpu_tmo<FL_TMO_VAL));
    /* loop while busy and not timeout */
    if (status&0x00FF) /* clear FSR if a hardware error is reported */
        fl_clear_status ();
    else
        return E_OK;
        /* generate user error codes */
    if (status & C163FL_STAT_BUSY)
        return E_TMO;
    else if (status & C163FL_STAT_SQER)
        return E_SQER;
    else if (status & C163FL_STAT_VPER)
        return E_VPER;
    else if (status & C163FL_STAT_OPER)
        return E_OPER;
    else if (status & C163FL_STAT_BUER)
        return E_BUER;

    return E_OK;
}

```