CYGNAL Application Note

Cache Optimizations for C8051F12x

Relevant Devices

This application note applies to the following devices:
C8051F120, C8051F121, C8051F122, C8051F123, C8051F124, C8051F125, C8051F126, C8051F127

Introduction

The C8051F12x family of devices have a branch target cache and prefetch engine to provide optimal performance at operating frequencies up to 100 MHz.

This application note describes:

- How to preload branch targets into the cache to ensure minimal and deterministic latency. This is useful for applications requiring low latency interrupt service routines.
- How to cache sections of code to allow code execution during FLASH write and erase operations. This allows data logging applications to respond to interrupts during log updates.
- How to perform FLASH block writes to increase the effective FLASH write rate. This is useful for storing large amounts of data in non-volatile memory.

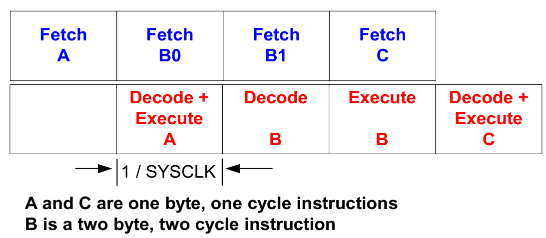
Key Points

- For most applications, the cache control registers should be left in their reset states.
- The branch target cache allows the CIP-51 core to execute instructions while FLASH write and erase operations are taking place.

Overview

The CIP-51 core implements a two stage pipeline for fetching and executing instructions. Figure 1 shows how the pipeline handles one and two cycle instructions.

Figure 1. CIP-51 Instruction Execution



The CIP-51 can fetch instructions from the cache, prefetch engine, or FLASH, as shown

CYGNAL Integrated Products, Inc.
4301 Westbank Drive
Suite B-100
Austin, TX 78746
www.cygnal.com

in Figure 2. Instructions found in the cache or the prefetch engine can be accessed in one instruction cycle. This is called a “cache hit”. “Cache misses” occur when instructions are not found in the cache or the prefetch engine. Each “cache miss” can cause the CIP-51 to stall for at least 40ns (4 clock cycles if FLASH read timing bits in the FLSCL register are set for 100 MHz operation, 2 clock cycles if FLASH read timing is set for 50 MHz, etc.) while the current instruction is read from FLASH.

Due to the operation of the Prefetch Engine, cache misses cannot occur when executing linear code. They can only occur when a branch, jump, or call is taken.

FLASH in the C8051F12x series is accessed in 4-byte segments aligned on multiples of 4 (i.e. the address of the first byte in the segment ends in 00b and the address of the last byte in the segment ends in 11b). The prefetch engine keeps the CIP-51 pipeline full when running linear code (code without any jumps or taken branches). After the CIP-51 receives a 4-byte segment to decode and execute, the prefetch engine immediately starts fetching the next 4-byte segment from FLASH. This allows the CIP-51 to execute linear code at full speed.

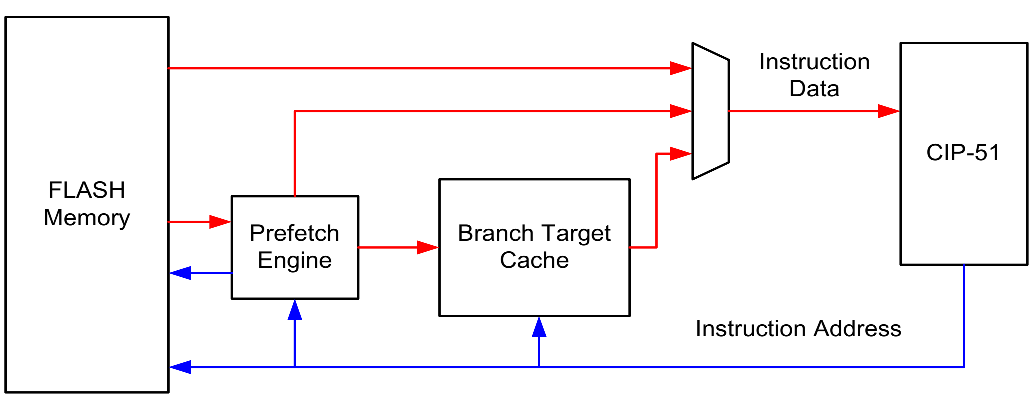
As data is read from FLASH by the prefetch engine and provide to the CPU, it is checked for the criteria necessary for instructions to be cached. Cached instructions remain in the cache until the cache is flushed or they are replaced with new data. The Cache Control (CCH0CN) and Tuning (CCH0TN) registers allow the user to specify cache criteria and select a cache replacement algorithm, respectively.

Preloading Branch Targets into the Cache

Most branch targets will be cached the first time they are encountered. However, to ensure a “cache hit” on select code branches, branch destinations may be preloaded and locked in the cache to minimize latency when the branch is taken. In ‘C’, code branches can occur on function calls, interrupts, *if*, *if-else*, *do*, *while*, *for*, *goto*, and *switch* statements.

Locking branch destinations in the cache is accomplished by setting the CHPUSH bit (CCH0LC.7) to a ‘1’ and issuing a MOVC instruction than targets the desired address in FLASH. This operation is called a “cache push”. When a FLASH address is pushed, the 4-byte segment at the pushed address is written to the next available unlocked slot, pointed to by the CHSLOT pointer (CCH0LC[5:0]). The

Figure 2. Branch Target Cache Data Flow



CHSLOT pointer is automatically decremented after each cache push. When CHSLOT reaches 000000b (all cache slots have been locked), the entire cache is unlocked.

It is important to note that "pushed" (locked) cache slots are never replaced by the cache engine. All "non-pushed" (unlocked) slots are used by the cache engine in its standard operation. If desired, pushed cache slots can be freed by issuing a "cache pop" operaton. A "cache pop" operation is accomplished by setting the CHPOP bit (CCH0LC.6) to a '1' and issuing a MOVC instruction. This operation increments the CHSLOT pointer and unlocks the most recently locked cache location. Note that a "cache pop" should not be initiated if CHSLOT is equal to 111110b.

If the branch destination being pushed is aligned on a 4-byte boundary, (the LSBs of the address of the first instruction are 00b), as shown in Figure 3, then only the 4-byte segment containing the first instruction needs to be locked in the cache in order to ensure no further miss penalties. If the target address is not aligned on a 4-byte boundary, then the first two 4-byte segments of the branch destination need to be locked in the cache to ensure that the CIP-51 will not stall until the next branch is encountered.

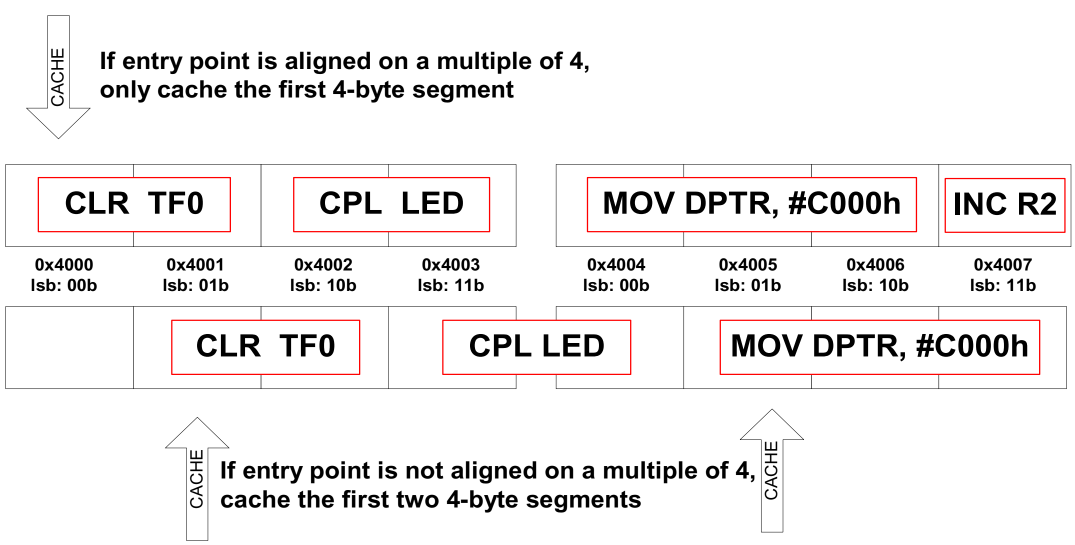
This optimization is especially important when trying to achieve deterministic response latency for an Interrupt Service Routine (ISR). Software Example 1 shows how to verify that branch targets have been cached properly by measuring the response latency of an ISR.

Example 1: Verifying that Branch Targets Have Been Cached

There are two approaches to verifying that branch targets have been cached. One method is to read the Cache Miss Accumulator (CCH0MA[6:0]) before and after executing the segment of code in question. If the Cache Miss Accumulator value increases, then some branch targets were not properly cached.

A more creative method that also demonstrates the effect of "cache misses" on ISR latency uses the on-chip debug capabilities and an

Figure 3. FLASH Segment Alignment



unused timer. The timer is configured to count SYSCLKs and service an interrupt on overflow. If a breakpoint is placed on the first instruction of the timer interrupt, the value of the timer when the breakpoint is reached is the number of SYSCLK cycles taken to enter the ISR after the overflow event has occurred. The software for this example is found at the end of this note.

The *main()* routine in this example locks Timer0_ISR's entry point in the cache then sets the CIP-51 in Idle mode waiting for the interrupt. The CPU wakes up when the Timer 0 overflow flag is set and vectors to the Timer0_ISR. This happens in 10 SYSCLK cycles (6 cycles to reach the interrupt vector and 4 cycles for the LJMP to the ISR entry point) if both the interrupt vector and the interrupt entry point have been locked in the cache properly.

If the interrupt vector and the ISR entry point have not been cached properly, it may take up to 16 SYSCLK cycles to reach the first instruction of the ISR, depending on the contents of the cache and the FLASH read timing register (FLSCL). Since the Timer0_ISR entry point is automatically cached when it is called, it is probable that the latency will only be non-minimal the first time the ISR is called. However, data that is cached by hardware may be replaced by the cache engine during normal operation.

Executing Code During FLASH Write and Erase Operations

C8051F12x devices have the capability of writing to FLASH memory from application code. During FLASH write and erase operations, the FLASH is taken off-line and is unavailable for read operations, which include instruction fetches and data reads. If the CPU attempts a FLASH read operation while the FLASH is off-line, the CPU is stalled until the FLASH operation has completed. Any interrupts which occur when the CPU is stalled are held pending and are serviced in priority order when the CPU resumes operation.

The cache on C8051F12x devices allows the CPU to execute code during FLASH write and erase operations. Because instruction and data fetches from the cache do not require access to FLASH memory, the CPU does not go off-line when a FLASH write or erase operation is initiated. However, all program code and data that is requested by the CPU during the FLASH operation must reside in the cache to keep it from stalling.

Many data logging applications require servicing interrupts while a FLASH log is updated. There are two requirements for servicing an interrupt during FLASH write and erase operations. First, all enabled interrupts should have their interrupt service routines pre-loaded in the cache to prevent the CPU from stalling during the FLASH write or erase operation. Second, the PC counter should be confined to a finite number of cached instructions for the duration of the FLASH operation. This prevents the CPU from going off-line during the FLASH operation.

A good way to contain the PC counter is by polling FLBUSY (FLSTAT.0) after the FLASH operation is initiated as shown in

Figure 4. FLBUSY will go low once the FLASH write or erase operation has completed. When using this technique, make sure that the code that polls FLBUSY after every FLASH operation is cached. The code in Figure 4 is placed inside a function and cached during device initialization, as described below in Example 2.

Example 2: Pre-loading an ISR in the Cache

Preloading an ISR in the cache involves caching all FLASH bytes that have the possibility of being requested by the CIP-51 in order to service an interrupt. This includes the interrupt vector, all opcode bytes in the ISR, and any data constants required by the ISR. This example shows how to preload and lock the Timer 0 interrupt service routine shown in Figure 5 into the cache. The software for this example is included at the end of this note.

Figure 5. Timer 0 Interrupt Service Routine

```
void Timer0_ISR (void) interrupt 1 {
   TF0 = 0;     // Clear Overflow Flag
   LED = ~LED; // Toggle LED
}
```

The three pieces of information needed to lock an ISR in the cache are the interrupt number, address of the ISR entry point, and the ISR length. In this example, the *Cache_ISR()* routine takes these three parameters as arguments and locks an ISR in the cache.

The interrupt number is used to determine the interrupt vector address to cache. The address of the interrupt vector is calculated using Equation 1. Most 'C' compilers insert an LJMP instruction at the interrupt vector instead of locating the ISR entry point at the interrupt vector address. Because all interrupt vectors start at the third byte of a 4-byte FLASH segment (the LSBs of the vector

Figure 4. Polling FLBUSY After a FLASH Write or Erase Operation

```
EA = 0;                             // Disable Global Interrupts
SFRPAGE = LEGACY_PAGE;
FLSCL |= 0x01;                      // Enable FLASH writes/erases
PSCTL  = 0x01;                      // MOVX writes FLASH byte

*pwrite++ = dat;                    // Initiate FLASH write (generate MOVX write)

SFRPAGE = LEGACY_PAGE;
PSCTL  = 0x00;                      // MOVX targets XRAM
FLSCL &= ~0x01;                     // Disable FLASH writes/erases
EA = 1;                             // Enable global interrupts

SFRPAGE = CONFIG_PAGE;
while(FLBUSY);                      // Any interrupts with preloaded ISRs may be
                                    // serviced while the CPU is in this loop
```

address are 10b), two cache pushes are necessary to lock the 3-byte LJMP instruction.

Equation 1. Calculating the Interrupt Vector Address from the Interrupt Number

$$\text{Interrupt Vector Address} = (\text{Interrupt Number} \times 8) + 3$$

When programming in C, the function name evaluates to the address of the first byte in the function. The first and second parameters of the *Cache_ISR()* routine are the starting address and length of the ISR. Figure 6 shows a *Cache_ISR()* function call that caches the Timer 0 ISR in this example. The interrupt service routine length may be obtained from the linker MAP (.M51) file for the project. Refer to the example (.M51) listing file in Linker/Locator chapter of the KEIL Assembler/Linker manual, available in PDF format from the *Help* menu of the Cygnal IDE, for more information on how to read MAP (.M51) files.

FLASH Block Writes

FLASH block writes allow writing up to four sequential FLASH bytes at once. FLASH block writes are enabled when CHBLKW (CCH0CN.0) is set to '1'.

FLASH blocks in code space consist of 4 adjacent FLASH bytes and are always aligned on multiples of 4 (i.e. the LSBs of the addresses must be 00b, 01b, 10b, and 11b). When a byte with address ending in 00b, 01b, or 10b is written, it is cached until a byte ending in 11b is written. The FLASH block write operation is initiated when the byte ending in 11b is written. In the Scratchpad area, FLASH blocks consist of 2 adjacent FLASH bytes and are aligned on multiples of two. The FLASH block write is performed when the byte with address ending in 1b is written.

FLASH Write Throughput

When storing a large array of data in FLASH, turning on FLASH block writes can decrease the time needed to copy the array by a factor of four. Table 1 compares the FLASH write

Figure 6. Calling the *Cache_ISR()* Routine

```
#define Timer0_ISR_LEN 5                          // Timer0_ISR is 5 bytes long
#define Timer0_ISR_NUM 1                          // Timer0 interrupt number is 1

// function prototype
void Cache_ISR(   unsigned int start_address, unsigned int interrupt_length,
                  unsigned int interrupt_number );

void main( void )
{
   ...
   // Pre-load Timer0_ISR in the cache
   Cache_ISR( (unsigned int) Timer0_ISR, Timer0_ISR_LEN, Timer0_ISR_NUM );
   ...
}
```


throughput between different methods of writing to FLASH.

Table 1. FLASH Write Throughput

FLASH Write Method	Throughput (KB/Sec)
Standard FLASH Byte Write to Scratchpad or Program Code	16.3
FLASH Block Write to Scratchpad	32.6
FLASH Block Write to Program Code	65.1

FLASH Block Write Considerations

FLASH block writes should only be used when writing **sequential** FLASH bytes. The reason for this is that the hardware only checks the 2 LSBs of the cached bytes. Also, the FLASH block write operation always targets the 4-byte FLASH segment containing the byte with address ending in 11b. Figure 7 shows an example of how data may be lost if FLASH block writes are enabled and FLASH writes do not target sequential memory locations.

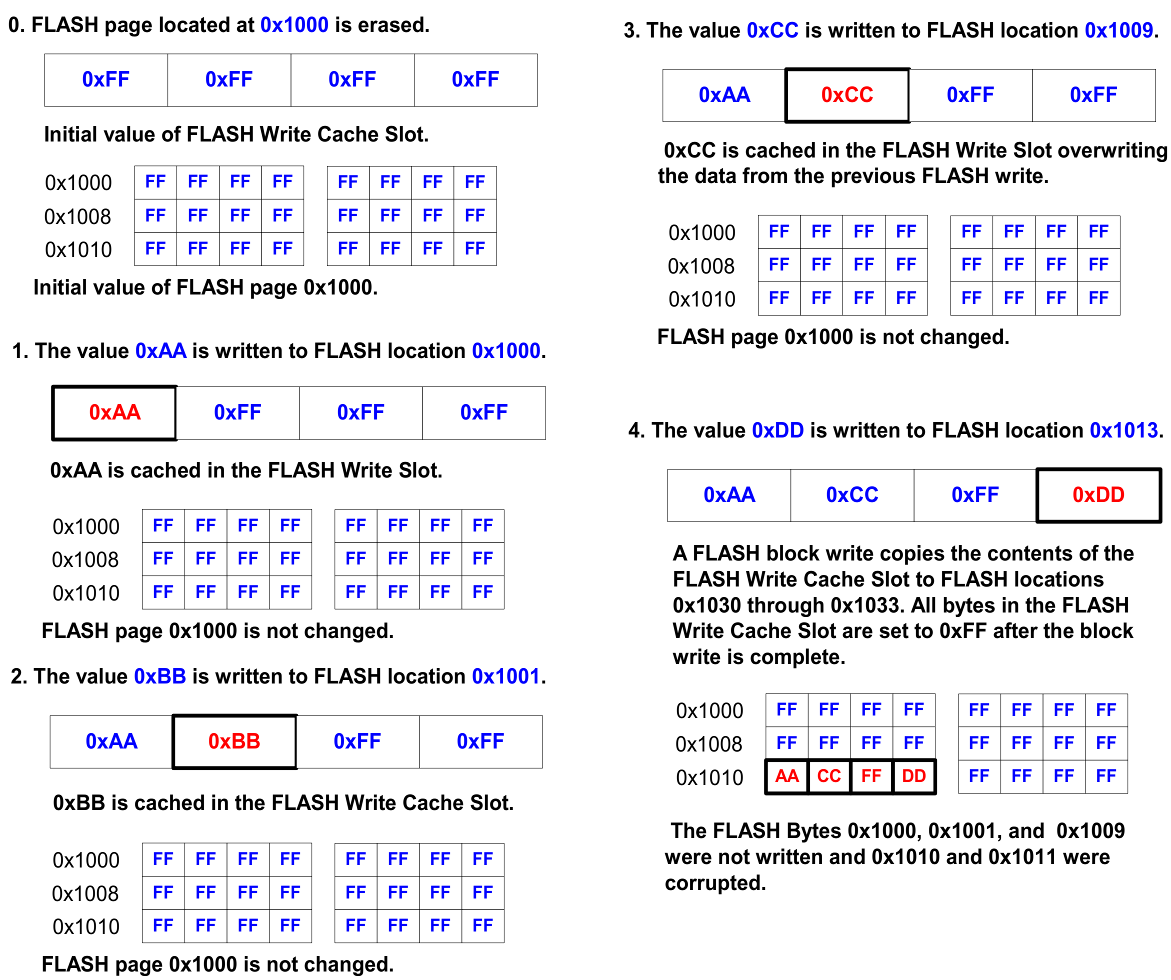
Example 3: Storing an XRAM Image in FLASH

This example demonstrates using FLASH block writes to copy an array of data from RAM to non-volatile memory. The routines provided in this example can be used to implement an application which captures an image of RAM on power down and restores the contents of RAM on power up. In this type of application, FLASH write time is crucial because the system only has a limited time to capture a RAM image after detecting a power failure.

In this example the *IMAGE_Capture()* routine writes the contents of XRAM to an array in FLASH. Since the FLASH array is written sequentially, FLASH block writes are used to decrease the time required to capture an image.

The *IMAGE_Capture()* routine uses a MOVX read to access data from XRAM and a MOVX write to store the data in FLASH. Since MOVX reads always target XDATA space and

Figure 7. Consequenses of Writing to Non-Sequential FLASH Locations when FLASH Block Writes are Enabled



not FLASH, the *IMAGE_Capture()* routine may be optimized by placing the code that enables and disables FLASH writes outside the *for* loop (see the Example 3 code at the end of this note). Caching the *IMAGE_Capture()* routine using techniques described earlier in this note will also decrease the time needed to capture an image of XRAM.

Example 4: Real-Time Non-Volatile Data Capture

This example combines several of the techniques discussed in this note to store 8-bit ADC samples in FLASH. The maximum ADC sampling rate in this example is limited by FLASH write time.

Determining the Maximum Sampling Rate

In this example SYSCLK is 49.76 MHz (we assume 50 MHz to make the math easier). A FLASH block write takes up to 60 µs to write four consecutive bytes to FLASH. The maximum byte write rate is (1/60 µs x 4 = 66.6 kHz). We will choose 65 kHz for simplicity. Because we are sampling 8-bit data, the ADC sampling rate is also set to 65 kHz.

Achieving the Maximum Sampling Rate

To achieve a 65 kHz FLASH write rate, the next 4 ADC samples should be ready to write to FLASH as soon as the current FLASH operation is complete. This is accomplished by executing code exclusively from the cache once the capture operation has started. This allows FLASH write operations to occur back to back.

Software Examples

Example 1:

```
//-----------------------------------------------------------------------------
// Cache_Example_1.c
//-----------------------------------------------------------------------------
// Copyright 2002 Cygnal Integrated Products, Inc.
//
// AUTH: FB
// DATE: 21 NOV 02
//
// This example shows how to cache an ISR entry point to achieve deterministic
// latency.  The Cache_ISR_Entry() function call may be commented out or compiled
// to compare the difference between cached ISR entry points verses ISRs which
// do not have their entry points cached.
//
// Timer0  is configured to count SYSCLKs, and service an interrupt on
// overflow. If a breakpoint is placed on the first instruction of Timer0_ISR,
// the value of the timer when the breakpoint is reached is the
// number of SYSCLK cycles taken to enter the ISR after the overflow event
// has occurred.
//
// This program uses the the 24.5 MHz internal oscillator multiplied by two
// for an effective SYSCLK of 49 MHz.
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03 / KEIL EVAL C51
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f120.h>                  // SFR declarations
#include <stdio.h>                      // printf() and getchar()

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F12x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                  // data pointer
sfr16 ADC0     = 0xbe;                  // ADC0 data
sfr16 ADC0GT   = 0xc4;                  // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                  // ADC0 less than window
sfr16 RCAP2    = 0xca;                  // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                  // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                  // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                  // Timer2
sfr16 TMR3     = 0xcc;                  // Timer3
sfr16 TMR4     = 0xcc;                  // Timer4
sfr16 DAC0     = 0xd2;                  // DAC0 data
sfr16 DAC1     = 0xd2;                  // DAC1 data
sfr16 PCA0CP5  = 0xe1;                  // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                  // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                  // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                  // PCA0 Module 4 capture
```



```
sfr16 PCA0     = 0xf9;                    // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                    // PCA0 Module 0 capture
sfr16 PCA0CP1  = 0xfd;                    // PCA0 Module 1 capture

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define TRUE         1
#define FALSE        0

#define INTCLK       24500000             // Internal oscillator frequency in Hz
#define SYSCLK       49000000             // Output of PLL derived from (INTCLK*2)

sbit LED = P1^6;                          // LED='1' means ON
sbit SW2 = P3^7;                          // SW2='0' means switch pressed

#define Timer0_ISR_NUM 1                  // Timer0 interrupt number

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void main(void);
void SYSCLK_Init(void);
void PORT_Init(void);
void Timer0_Init (void);
void Timer0_ISR (void);
void Cache_ISR_Entry( unsigned int start_address,
                      unsigned int interrupt_number );

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{

   WDTCN = 0xde;                          // disable watchdog timer
   WDTCN = 0xad;

   PORT_Init ();                          // initialize crossbar and GPIO
   SYSCLK_Init ();                        // initialize oscillator
   Timer0_Init ();                        // initialize Timer0;

   //Cache_ISR_Entry( (unsigned int) Timer0_ISR, Timer0_ISR_NUM);

   EA = 1;

   while(1){

      PCON |= 0x01;                       // Put the CPU in idle mode. The CPU
                                          // will only enter idle mode when
                                          // running at full speed and will return
                                          // to full operation when Timer0
                                          // overflows. This instruction has no
                                          // effect if the CIP-51 is in debug
                                          // mode.
   }
```

```
}


//-----------------------------------------------------------------------------
// Timer0_ISR
//-----------------------------------------------------------------------------
//
// This interrupt service routine toggles the LED when Timer0 overflows.
//
void Timer0_ISR (void) interrupt 1 {

   TF0 = 0;                            // Clear Timer0 overflow flag
   LED = ~LED;                         // Toggle LED
}

//-----------------------------------------------------------------------------
// Cache_ISR_Entry
//-----------------------------------------------------------------------------
//
// This routine pushes and locks an interrupt vector and the first one or two
// 4-byte FLASH segments of an ISR entry point into the cache.
//
void Cache_ISR_Entry (   unsigned int start_address,
                         unsigned int interrupt_number   )
{
   char SFRPAGE_SAVE = SFRPAGE;        // Preserve current SFR page
   char EA_SAVE = EA;                  // Preserve interrupt state

   unsigned char code* pread;          // Pointer used to generate MOVC
                                       // instructions to initiate a cache
                                       // push operation

   unsigned char temp_char;            // Temporary char.

   // Set the <pread> pointer to the address of the interrupt vector.
   pread = ((interrupt_number * 8) + 3 );


   SFRPAGE = CONFIG_PAGE;              // Set SFR page

   EA = 0;                             // Disable Interrupts

   CCH0LC &= ~0xC0;                    // Clear the CHPUSH and CHPOP bits
   CCH0LC |=  0x80;                    // Enable cache pushes, MOVC instructions
                                       // will push 4-byte FLASH segments into
                                       // the cache

   // Check if there is enough room to cache the interrupt vector and the first
   // two 4-byte segments of the ISR entry point.  When the two (or one) slot(s)
   // required to cache the ISR entry point and the two slots needed to cache the
   // interrupt vector (Total 4 slots) are subtracted from the CHSLOT pointer,
   // the result is the number of unlocked slots available.
   if( ((CCH0LC & 0x3F) - 4) <= 0 ){
      while(1);                        // Handle Error Condition.
                                       // Not enough room to cache ISR Entry Point
   }
```



```
   // Push the interrupt vector in the cache slot pointed to by CHSLOT.
   // An interrupt vector generated by the KEIL C51 compiler consists of a
   // 3-byte LJMP instruction. Since the LSBs of an interrupt vector address are
   // always 10b, the first two bytes of the LJMP instruction fall in the first
   // 4-byte FLASH segment and the remaining byte of the instruction falls in the
   // second 4-byte segment.  Two cache pushes are neccessary.

   temp_char = *pread;                 // Push the first 4-byte segment (MOVC)
   temp_char = *(pread + 4);           // Push the second 4-byte segment (MOVC)

   // If the first byte of the ISR is aligned on a multiple of 4 (i.e. the LSBs
   // of the first byte are 00b), then cache the first 4-byte segment. Otherwise,
   // cache the first two 4-byte segments.
   pread = (unsigned char code*) start_address;
                                       // Set <pread> to the address of the
                                       // interrupt service routine.

   temp_char = *pread;                 // Push the first 4-byte segment (MOVC)

   if(start_address & 0x03){
      temp_char = *(pread + 4);        // Push the second 4-byte segment (MOVC)
   }                                   // if the two LSBs of <start_address>
                                       // are not zero

   CCH0LC &= ~0x80;                    // Clear CHPUSH to disable cache pushes.
   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page.
   EA = EA_SAVE;                       // Restore interrupt state.

}

//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal oscillator
// at 24.5 MHz multiplied by two using the PLL.
//
void SYSCLK_Init (void)
{
   int i;                           // software timer

   char SFRPAGE_SAVE = SFRPAGE;     // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;           // set SFR page

   OSCICN = 0x83;                   // set internal oscillator to run
                                    // at its maximum frequency

   CLKSEL = 0x00;                   // Select the internal osc. as
                                    // the SYSCLK source

   //Turn on the PLL and increase the system clock by a factor of M/N = 2
```

```
   SFRPAGE = CONFIG_PAGE;

   PLL0CN  = 0x00;                  // Set internal osc. as PLL source
   SFRPAGE = LEGACY_PAGE;
   FLSCL   = 0x10;                  // Set FLASH read time for 50MHz clk
                                    // or less
   SFRPAGE = CONFIG_PAGE;
   PLL0CN |= 0x01;                  // Enable Power to PLL
   PLL0DIV = 0x01;                  // Set Pre-divide value to N (N = 1)
   PLL0FLT = 0x01;                  // Set the PLL filter register for
                                    // a reference clock from 19 - 30 MHz
                                    // and an output clock from 45 - 80 MHz
   PLL0MUL = 0x02;                  // Multiply SYSCLK by M (M = 2)

   for (i=0; i < 256; i++) ;        // Wait at least 5us
   PLL0CN  |= 0x02;                 // Enable the PLL
   while(!(PLL0CN & 0x10));         // Wait until PLL frequency is locked
   CLKSEL  = 0x02;                  // Select PLL as SYSCLK source

   SFRPAGE = SFRPAGE_SAVE;          // Restore SFR page
}


//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// This routine configures the crossbar and GPIO ports.
//
void PORT_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;     // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;           // set SFR page

   XBR0     = 0x00;
   XBR1     = 0x00;
   XBR2     = 0x40;                 // Enable crossbar and weak pull-up


   P1MDOUT |= 0x40;                 // Set P1.6(LED) to push-pull

   SFRPAGE = SFRPAGE_SAVE;          // Restore SFR page
}


//-----------------------------------------------------------------------------
// Timer0_Init
//-----------------------------------------------------------------------------
//
// Configure Timer0 to count SYSCLKs in Mode 1 (16-bit timer).
//
void Timer0_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;     // Save Current SFR page

   SFRPAGE = TIMER01_PAGE;
```



```
   TMOD   &= ~0x0F;
   TMOD   |=  0x01;                  // TMOD: Timer0, mode 1, 16-bit timer

   CKCON  |= 0x08;                   // Timer0 counts SYSCLKs

   ET0 = 1;                          // Enable Timer0 overflow interrupts

   TR0 = 1;                          // Start Timer0

   SFRPAGE = SFRPAGE_SAVE;           // Restore SFR page

}
```

Example 2:

```
//-----------------------------------------------------------------------------
// Cache_Example_2.c
//-----------------------------------------------------------------------------
// Copyright 2002 Cygnal Integrated Products, Inc.
//
// AUTH: FB
// DATE: 21 NOV 02
//
// This example shows how to service interrupts during FLASH write operations.
// During device initialization, the FLASH_ByteWrite_NonBlocking() function
// and the Timer0_ISR() are locked in the cache. This allows Timer0 interrupts
// to be serviced while a FLASH write is taking place.
//
// This program uses the the 24.5 MHz internal oscillator multiplied by two
// for an effective SYSCLK of 49 MHz. This program also initializes and uses
// UART1 at <BAUDRATE> bits per second.
//
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03 / KEIL EVAL C51
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f120.h>                 // SFR declarations
#include <stdio.h>                     // printf() and getchar()

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F12x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                 // data pointer
sfr16 ADC0     = 0xbe;                 // ADC0 data
sfr16 ADC0GT   = 0xc4;                 // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                 // ADC0 less than window
sfr16 RCAP2    = 0xca;                 // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                 // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                 // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                 // Timer2
sfr16 TMR3     = 0xcc;                 // Timer3
sfr16 TMR4     = 0xcc;                 // Timer4
sfr16 DAC0     = 0xd2;                 // DAC0 data
sfr16 DAC1     = 0xd2;                 // DAC1 data
sfr16 PCA0CP5  = 0xe1;                 // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                 // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                 // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                 // PCA0 Module 4 capture
sfr16 PCA0     = 0xf9;                 // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 capture
sfr16 PCA0CP1  = 0xfd;                 // PCA0 Module 1 capture

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
```

```
#define TRUE          1
#define FALSE         0

#define INTCLK        24500000          // Internal oscillator frequency in Hz
#define SYSCLK        49000000          // Output of PLL derived from (INTCLK*2)

sbit LED = P1^6;                        // LED='1' means ON
sbit SW2 = P3^7;                        // SW2='0' means switch pressed

#define Timer0_ISR_NUM 0x01             // Timer0 interrupt number
#define Timer0_ISR_LEN 0x05             // Length of Timer0_ISR in bytes
                                        // Check the MAP (*.M51) file for
                                        // this project.

#define FLASH_ByteWrite_NonBlocking_LEN 0x22
                                        // The number of bytes that will be
                                        // cached in the FLASH write routine.

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void main(void);
void SYSCLK_Init(void);
void PORT_Init(void);
void Timer0_Init (void);
void Timer0_ISR (void);
void Cache_ISR(   unsigned int start_address,
                  unsigned int interrupt_length,
                  unsigned int interrupt_number );
void Cache_Function( unsigned int start_address, unsigned int length );
void FLASH_ByteWrite_NonBlocking ( unsigned char xdata* pwrite, unsigned char dat);
void FLASH_PageErase(unsigned int address);


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{
   unsigned char xdata* data pwrite;    // FLASH write and erase pointer
    unsigned char temp_char = 0x00;      // Temporary char
   int i;                               // Software counter

   WDTCN = 0xde;                        // Disable watchdog timer
   WDTCN = 0xad;

   PORT_Init ();                        // Initialize crossbar and GPIO
   SYSCLK_Init ();                      // Initialize oscillator
   Timer0_Init ();                      // Initialize Timer0;

   Cache_Function( (unsigned int) FLASH_ByteWrite_NonBlocking,
                                  FLASH_ByteWrite_NonBlocking_LEN );


   Cache_ISR( (unsigned int) Timer0_ISR, Timer0_ISR_LEN, Timer0_ISR_NUM);
```

```
   FLASH_PageErase(0x1000);            // Erase the FLASH page at 0x1000
   pwrite = 0x1000;                    // Set FLASH write pointer to 0x1000

   EA = 1;

   for( i = 0; i < 100; i++ ){

      // Write a byte to FLASH without blocking any cached ISRs (eg. Timer0_ISR)
      FLASH_ByteWrite_NonBlocking( pwrite++, temp_char++);
   }

   while(1);

}


//-----------------------------------------------------------------------------
// Timer0_ISR
//-----------------------------------------------------------------------------
//
// This interrupt service routine toggles the LED when Timer0 overflows.

void Timer0_ISR (void) interrupt 1 {

   TF0 = 0;                            // Clear Timer0 overflow flag
   LED = ~LED;                         // Toggle LED


}

//-----------------------------------------------------------------------------
// Cache_ISR
//-----------------------------------------------------------------------------
//
// This routine pushes and locks an entire ISR and its interrupt vector into
// the cache. This function assumes that the interrupt vector and ISR is
// located in the common area.
//
void Cache_ISR(   unsigned int start_address,
                  unsigned int interrupt_length,
                  unsigned int interrupt_number    )
{
   char SFRPAGE_SAVE = SFRPAGE;        // Preserve current SFR page
   char EA_SAVE = EA;                  // Preserve interrupt state

   unsigned char code* pread;          // Pointer used to generate MOVC
                                       // instructions to initiate a cache
                                       // push operation

   unsigned char temp_char;            // Temporary char.

   SFRPAGE = CONFIG_PAGE;              // Set SFR page

   // Check if there is enough room to cache the ISR. When the number of
   // slots required by the ISR + interrupt vector are subtracted from
   // the CHSLOT pointer, the result is the number of unlocked slots available.
   if( ((CCH0LC & 0x3F) - (interrupt_length/4 + 2)) <= 0){
```



```
      while(1);                             // Handle Error Condition.
                                            // Not enough room to cache ISR
   }

   // Set the <pread> pointer to the address of the interrupt vector.
   pread = ((interrupt_number * 8) + 3 );

   EA = 0;                                  // Disable Interrupts

   CCH0LC &= ~0xC0;                         // Clear the CHPUSH and CHPOP bits
   CCH0LC |=  0x80;                         // Enable cache pushes, MOVC instructions
                                            // will push 4-byte FLASH segments into
                                            // the cache

   // Push the interrupt vector in the cache slot pointed to by CHSLOT.
   // An interrupt vector consists of a 3-byte LJMP instruction and the
   // LSBs of the vector address are always 10b. One byte of the LJMP
   // instruction fall in the first 4-byte FLASH segment and the remaining
   // two bytes of the instruction falls in the second 4-byte segment.
   // Two cache pushes are neccessary.

   temp_char = *pread;                      // Push the first 4-byte segment (MOVC)
   temp_char = *(pread + 4);                // Push the second 4-byte segment (MOVC)

   // Push the interrupt service routine in the cache. The interrupt starts
   // at <START_LABEL> and is <interrupt_length> bytes long.

   pread = (unsigned char code*) start_address;
                                            // Set <pread> to the address of the
                                            // interrupt service routine.

   while( pread < start_address + interrupt_length ){

      temp_char = *pread;                   // Push a 4-byte segment (MOVC)

      pread += 4;                           // Increment <pread> to the next
                                            // segment.
   }


   CCH0LC &= ~0x80;                         // Clear CHPUSH to disable cache pushes.
   SFRPAGE = SFRPAGE_SAVE;                  // Restore SFR page.
   EA = EA_SAVE;                            // Restore interrupt state.

}

//-----------------------------------------------------------------------------
// Cache_Function
//-----------------------------------------------------------------------------
//
// This routine pushes and locks a function into the cache. This routine
// assumes that the function is located in the common area.
//
//
void Cache_Function( unsigned int start_address, unsigned int length )
{
   char SFRPAGE_SAVE = SFRPAGE;             // Preserve current SFR page
```

```
   char EA_SAVE = EA;                 // Preserve interrupt state

   unsigned char code* pread;         // Pointer used to generate MOVC
                                      // instructions to initiate a cache
                                      // push operation

   unsigned char temp_char;           // Temporary char.

   SFRPAGE = CONFIG_PAGE;             // Set SFR page

   // Check if there is enough room to cache the function. When the number of
   // slots required to cache the function is subtracted from
   // the CHSLOT pointer, the result is the number of unlocked slots available.
   if( (CCH0LC & 0x3F) - (length/4) <= 0){
      while(1);                       // Handle Error Condition.
                                      // Not enough room to cache function
   }

   // Set <pread> to the address of the function
   pread = (unsigned char code*) start_address;



   EA = 0;                            // Disable Interrupts

   CCH0LC &= ~0xC0;                   // Clear the CHPUSH and CHPOP bits
   CCH0LC |=  0x80;                   // Enable cache pushes, MOVC instructions
                                      // will push 4-byte FLASH segments into
                                      // the cache

   // Push the function in the cache. The interrupt starts
   // at <START_LABEL> and is <length> bytes long.
   while( pread < start_address + length ){

      temp_char = *pread;             // Push a 4-byte segment (MOVC)

      pread += 4;                     // Increment <pread> to the next
                                      // segment.
   }


   CCH0LC &= ~0x80;                   // Clear CHPUSH to disable cache pushes.
   SFRPAGE = SFRPAGE_SAVE;            // Restore SFR page.
   EA = EA_SAVE;                      // Restore interrupt state.

}

//-----------------------------------------------------------------------------
// FLASH_ByteWrite_NonBlocking
//-----------------------------------------------------------------------------
//
// This routine writes one byte to FLASH and allows interrupts with cached ISRs
// to be serviced during the FLASH write operation.
//
void FLASH_ByteWrite_NonBlocking ( unsigned char xdata* pwrite, unsigned char dat )
{
```

```
   EA = 0;                            // Disable Global Interrupts
   SFRPAGE = LEGACY_PAGE;
   FLSCL |= 0x01;                     // Enable FLASH writes/erases
   PSCTL  = 0x01;                     // MOVX writes FLASH byte

   *pwrite++ = dat;                   // Initiate FLASH write (generate MOVX write)

   SFRPAGE = LEGACY_PAGE;
   PSCTL  = 0x00;                     // MOVX targets XRAM
   FLSCL &= ~0x01;                    // Disable FLASH writes/erases
   EA = 1;                            // Enable global interrupts

   SFRPAGE = CONFIG_PAGE;
   while(FLBUSY);                     // Any interrupts with preloaded ISRs may be
                                      // serviced while the CIP-51 is in this loop
}

//-----------------------------------------------------------------------------
// FLASH_PageErase
//-----------------------------------------------------------------------------
//
// This routine erases the FLASH page located at <address>;
//
void FLASH_PageErase(unsigned int address)
{
   char SFRPAGE_SAVE = SFRPAGE;         // Preserve current SFR page
   bit EA_SAVE;                         // Preserve interrupt state
   char xdata * idata pwrite;           // FLASH write/erase pointer

   // Set the FLASH write/erase pointer to the FLASH page at 0x1000
   pwrite = (char xdata *) address;

   SFRPAGE = LEGACY_PAGE;

   EA_SAVE = EA;
   EA = 0;                              // Disable interrupts
   FLSCL |= 0x01;                       // Enable FLASH writes/erases
   PSCTL = 0x03;                        // MOVX writes FLASH byte

   *pwrite = 0;                         // Initiate FLASH page erase

   FLSCL &= ~0x01;                      // Disable FLASH writes/erases
   PSCTL = 0x00;                        // MOVX targets XRAM
   EA = EA_SAVE;                        // Restore interrupt state

   SFRPAGE = SFRPAGE_SAVE;              // Restore SFR page

}

//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
```

```
// This routine initializes the system clock to use the internal oscillator
// at 24.5 MHz multiplied by two using the PLL.
//
void SYSCLK_Init (void)
{
   int i;                              // software timer

   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;              // set SFR page

   OSCICN = 0x83;                      // set internal oscillator to run
                                       // at its maximum frequency

   CLKSEL = 0x00;                      // Select the internal osc. as
                                       // the SYSCLK source

   //Turn on the PLL and increase the system clock by a factor of M/N = 2
   SFRPAGE = CONFIG_PAGE;

   PLL0CN  = 0x00;                     // Set internal osc. as PLL source
   SFRPAGE = LEGACY_PAGE;
   FLSCL   = 0x10;                     // Set FLASH read time for 50MHz clk
                                       // or less
   SFRPAGE = CONFIG_PAGE;
   PLL0CN |= 0x01;                     // Enable Power to PLL
   PLL0DIV = 0x01;                     // Set Pre-divide value to N (N = 1)
   PLL0FLT = 0x01;                     // Set the PLL filter register for
                                       // a reference clock from 19 - 30 MHz
                                       // and an output clock from 45 - 80 MHz
   PLL0MUL = 0x02;                     // Multiply SYSCLK by M (M = 2)

   for (i=0; i < 256; i++) ;           // Wait at least 5us
   PLL0CN  |= 0x02;                    // Enable the PLL
   while(!(PLL0CN & 0x10));            // Wait until PLL frequency is locked
   CLKSEL  = 0x02;                     // Select PLL as SYSCLK source

   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}


//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// This routine configures the crossbar and GPIO ports.
//
void PORT_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;              // set SFR page

   XBR0     = 0x00;
   XBR1     = 0x00;
   XBR2     = 0x40;                    // Enable crossbar and weak pull-up
```

```
   P1MDOUT |= 0x40;               // Set P1.6(LED) to push-pull

   SFRPAGE = SFRPAGE_SAVE;        // Restore SFR page
}


//-----------------------------------------------------------------------------
// Timer0_Init
//-----------------------------------------------------------------------------
//
// Configure Timer0 to count SYSCLKs in Mode1(16-bit counter timer).
//
void Timer0_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;   // Save Current SFR page

   SFRPAGE = TIMER01_PAGE;

   TMOD   &= ~0x0F;
   TMOD   |=  0x01;               // TMOD: Timer0, mode 1, 16-bit timer

   CKCON  |= 0x08;                // Timer0 counts SYSCLKs

   ET0 = 1;                       // Enable Timer0 overflow interrupts

   TR0 = 1;                       // Start Timer0

   SFRPAGE = SFRPAGE_SAVE;        // Restore SFR page

}
```

Example 3:

```
//-----------------------------------------------------------------------------
// Cache_Example_3.c
//-----------------------------------------------------------------------------
// Copyright 2002 Cygnal Integrated Products, Inc.
//
// AUTH: FB
// DATE: 14 OCT 02
//
// This example demonstrates using FLASH block writes to copy an array of data
// from RAM to non-volatile FLASH memory.
//
// This program uses the the 24.5 MHz internal oscillator multiplied by two
// for an effective SYSCLK of 49 MHz.
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03 / KEIL EVAL C51
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f120.h>                  // SFR declarations
#include <stdio.h>                      // printf() and getchar()

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F12x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                  // data pointer
sfr16 ADC0     = 0xbe;                  // ADC0 data
sfr16 ADC0GT   = 0xc4;                  // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                  // ADC0 less than window
sfr16 RCAP2    = 0xca;                  // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                  // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                  // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                  // Timer2
sfr16 TMR3     = 0xcc;                  // Timer3
sfr16 TMR4     = 0xcc;                  // Timer4
sfr16 DAC0     = 0xd2;                  // DAC0 data
sfr16 DAC1     = 0xd2;                  // DAC1 data
sfr16 PCA0CP5  = 0xe1;                  // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                  // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                  // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                  // PCA0 Module 4 capture
sfr16 PCA0     = 0xf9;                  // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                  // PCA0 Module 0 capture
sfr16 PCA0CP1  = 0xfd;                  // PCA0 Module 1 capture

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define TRUE         1
#define FALSE        0

#define INTCLK       24500000           // Internal oscillator frequency in Hz
```



```
#define SYSCLK       49000000          // Output of PLL derived from (INTCLK*2)

sbit LED = P1^6;                       // LED='1' means ON
sbit SW2 = P3^7;                       // SW2='0' means switch pressed

#define FLASH_PAGESIZE 1024            // The size of a FLASH page in bytes
#define IMAGE_BUFF_LEN 4096            // The size of <IMAGE_BUFF> in bytes
#define XRAM_BUFF_LEN 4096             // The size of <XRAM_BUFF> in bytes

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void main(void);
void SYSCLK_Init(void);
void PORT_Init(void);
void IMAGE_Erase(void);
void IMAGE_Capture(void);
void IMAGE_Restore(void);

//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------
char xdata XRAM_BUFF[XRAM_BUFF_LEN] _at_ 0x0000;
char code IMAGE_BUFF[IMAGE_BUFF_LEN] _at_ 0x1000;


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{
   int i;                              // software counter

   WDTCN = 0xde;                       // disable watchdog timer
   WDTCN = 0xad;

   PORT_Init ();                       // initialize crossbar and GPIO
   SYSCLK_Init ();                     // initialize oscillator

   // Fill XRAM with 0xAA
   for( i = 0; i < XRAM_BUFF_LEN; i++){
      XRAM_BUFF[i] = 0xAA;
   }

   // Erase the XRAM image buffer in FLASH and capture the current
   // XRAM contents
   IMAGE_Erase();
   IMAGE_Capture();

   // Fill XRAM with 0xBB
   for( i = 0; i < XRAM_BUFF_LEN; i++){
      XRAM_BUFF[i] = 0xBB;
   }

   // Restore the original XRAM image.
   IMAGE_Restore();
```

```
}

//-----------------------------------------------------------------------------
// Support Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// IMAGE_Erase
//-----------------------------------------------------------------------------
//
// This function erases the FLASH pages reserved for capturing an image of XRAM.
//
//
void IMAGE_Erase(void)
{
   char SFRPAGE_SAVE = SFRPAGE;        // Preserve current SFR page
   bit EA_SAVE;                        // Preserve interrupt state
   int i;                              // Software counter
   char xdata * idata pwrite;          // FLASH write/erase pointer

   // Set the FLASH write/erase pointer to the beginning of <IMAGE_BUFF>
   pwrite = IMAGE_BUFF;

   // Erase all FLASH pages in the XRAM image buffer
   for( i = 0; i < IMAGE_BUFF_LEN; i += FLASH_PAGESIZE ){

      SFRPAGE = LEGACY_PAGE;

      EA_SAVE = EA;
      EA = 0;                          // Disable interrupts
      FLSCL |= 0x01;                   // Enable FLASH writes/erases
      PSCTL = 0x03;                    // MOVX writes FLASH byte

      pwrite[i] = 0;                   // Initiate FLASH page erase

      FLSCL &= ~0x01;                  // Disable FLASH writes/erases
      PSCTL = 0x00;                    // MOVX targets XRAM
      EA = EA_SAVE;                    // Restore interrupt state
   }

   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}

//-----------------------------------------------------------------------------
// IMAGE_Capture
//-----------------------------------------------------------------------------
//
// This function captures an image of XRAM and stores it in FLASH.
//
//
void IMAGE_Capture(void)
{
   char SFRPAGE_SAVE = SFRPAGE;        // Preserve current SFR page
   bit EA_SAVE;                        // Preserve interrupt state
   int i;                              // Software counter
   char xdata * idata pwrite;          // FLASH write/erase pointer
```



```
   unsigned char dat;                   // Holds data to copy

   // Set the FLASH write/erase pointer to the beginning of <IMAGE_BUFF>
   pwrite = IMAGE_BUFF;

   // Enable FLASH block writes
   SFRPAGE = CONFIG_PAGE;
   CCH0CN |= 0x01;                      // Set the CHBLKW bit to enable
                                        // FLASH block writes

   // Copy an image of the entire XRAM to <IMAGE_BUFF> in FLASH
   for( i = 0; i < IMAGE_BUFF_LEN; i++ ){

      dat = XRAM_BUFF[i];               // Read one byte from XRAM

      SFRPAGE = LEGACY_PAGE;

      EA_SAVE = EA;
      EA = 0;                           // Disable interrupts
      FLSCL |= 0x01;                    // Enable FLASH writes/erases
      PSCTL = 0x01;                     // MOVX writes write FLASH byte
                                        // MOVX reads still target XRAM

      pwrite[i] = dat;                  // Write Data to FLASH

      FLSCL &= ~0x01;                   // Disable FLASH writes/erases
      PSCTL = 0x00;                     // MOVX targets XRAM
      EA = EA_SAVE;                     // Restore interrupt state
   }

   SFRPAGE = CONFIG_PAGE;
   CCH0CN &= ~0x01;                     // Clear the CHBLKW bit

   SFRPAGE = SFRPAGE_SAVE;              // Restore SFR page
}

//-----------------------------------------------------------------------------
// IMAGE_Restore
//-----------------------------------------------------------------------------
//
// This function restores the contents of XRAM from a memory image stored
// in FLASH.
//
void IMAGE_Restore(void)
{
   int i;                               // Software counter

   // Restore the image in <IMAGE_BUFF> in to XRAM
   for( i = 0; i < IMAGE_BUFF_LEN; i++ ){

      XRAM_BUFF[i] = IMAGE_BUFF[i];     // Restore one byte of XRAM
   }

}

//-----------------------------------------------------------------------------
// Initialization Routines
```

```
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal oscillator
// at 24.5 MHz multiplied by two using the PLL.
//
void SYSCLK_Init (void)
{
   int i;                              // software timer

   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;              // set SFR page

   OSCICN = 0x83;                      // set internal oscillator to run
                                       // at its maximum frequency

   CLKSEL = 0x00;                      // Select the internal osc. as
                                       // the SYSCLK source

   //Turn on the PLL and increase the system clock by a factor of M/N = 2
   SFRPAGE = CONFIG_PAGE;

   PLL0CN  = 0x00;                     // Set internal osc. as PLL source
   SFRPAGE = LEGACY_PAGE;
   FLSCL   = 0x10;                     // Set FLASH read time for 50MHz clk
                                       // or less
   SFRPAGE = CONFIG_PAGE;
   PLL0CN |= 0x01;                     // Enable Power to PLL
   PLL0DIV = 0x01;                     // Set Pre-divide value to N (N = 1)
   PLL0FLT = 0x01;                     // Set the PLL filter register for
                                       // a reference clock from 19 - 30 MHz
                                       // and an output clock from 45 - 80 MHz
   PLL0MUL = 0x02;                     // Multiply SYSCLK by M (M = 2)

   for (i=0; i < 256; i++) ;           // Wait at least 5us
   PLL0CN  |= 0x02;                    // Enable the PLL
   while(!(PLL0CN & 0x10));            // Wait until PLL frequency is locked
   CLKSEL  = 0x02;                     // Select PLL as SYSCLK source

   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}


//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// This routine configures the crossbar and GPIO ports.
//
void PORT_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page
```



```
   SFRPAGE = CONFIG_PAGE;              // set SFR page

   XBR0     = 0x00;
   XBR1     = 0x00;
   XBR2     = 0x40;                    // Enable crossbar and weak pull-ups


   P1MDOUT |= 0x40;                    // Set P1.6(LED) to push-pull

   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}
```

Example 4:

```
//-----------------------------------------------------------------------------
// Cache_Example_4.c
//-----------------------------------------------------------------------------
// Copyright 2002 Cygnal Integrated Products, Inc.
//
// AUTH: FB
// DATE: 21 NOV 02
//
// This example combines several of the techniques discussed in AN034 to
// store ADC samples in FLASH. The maximum sampling rate in this example
// is limited by FLASH write time.
//
// A FLASH block write takes less than 60 µs to write four consecutive bytes
// to FLASH. The maximum sampling rate is (1/60 µs x 4 = 66.6 kHz). We will
// choose 65 kHz for simplicity. To achieve a 65 kHz FLASH write rate, the next
// 4 ADC samples should be ready to write to FLASH as soon as the current FLASH
// operation is complete. This is accomplished by executing code exclusively
// from the cache once the capture operation has started. This will allow FLASH
// write operations to occur back to back.
//
// This program uses a 22.1184 Mhz crystal oscillator multiplied by (9/4)
// for an effective SYSCLK of 49.7664 Mhz. This program also initializes and
// uses UART0 at <BAUDRATE> bits per second.
//
// Target: C8051F12x
// Tool chain: KEIL C51 6.03 / KEIL EVAL C51
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f120.h>                  // SFR declarations
#include <stdio.h>                      // printf() and getchar()

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F12x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                  // data pointer
sfr16 ADC0     = 0xbe;                  // ADC0 data
sfr16 ADC0GT   = 0xc4;                  // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                  // ADC0 less than window
sfr16 RCAP2    = 0xca;                  // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                  // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                  // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                  // Timer2
sfr16 TMR3     = 0xcc;                  // Timer3
sfr16 TMR4     = 0xcc;                  // Timer4
sfr16 DAC0     = 0xd2;                  // DAC0 data
sfr16 DAC1     = 0xd2;                  // DAC1 data
sfr16 PCA0CP5  = 0xe1;                  // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                  // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                  // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                  // PCA0 Module 4 capture
sfr16 PCA0     = 0xf9;                  // PCA0 counter
```



```
sfr16 PCA0CP0   = 0xfb;                // PCA0 Module 0 capture
sfr16 PCA0CP1   = 0xfd;                // PCA0 Module 1 capture


//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define TRUE         1
#define FALSE        0

#define EXTCLK       22118400          // External oscillator frequency in Hz.
#define SYSCLK       49766400          // Output of PLL derived from
                                       // (EXTCLK*9/4).

#define BAUDRATE     115200            // Baud rate of UART in bps
                                       // Note: The minimum standard baud rate
                                       // supported by the UART0_Init routine
                                       // in this file is 19,200 bps when
                                       // SYSCLK = 49.76MHz.

#define SAMPLERATE   65000             // Approximate ADC0 Sampling Rate,
                                       // limited by FLASH write rates.


sbit LED = P1^6;                       // LED='1' means ON.
sbit SW2 = P3^7;                       // SW2='0' means switch pressed.


#define ADC0_ISR_LEN 0x63              // This macro defines the length (in
                                       // bytes) of the ADC0_ISR routine. It
                                       // is used to determine how many bytes
                                       // to push into the cache. This
                                       // information can be obtained from
                                       // the *.M51 file associated with this
                                       // project.

#define ADC0_ISR_NUM 15                // This macro defines the ADC0 End of
                                       // conversion interrupt number.

#define while_TR3_NonBlocking_LEN 0x04 // Length of the while_TR3_NonBlocking()
                                       // function in bytes.


#define LOG_START 0x1000               // Starting address of LOG.
#define LOG_END 0x2000                 // Ending address of LOG.
#define FLASH_PAGESIZE 1024            // Number of bytes in a FLASH page.

//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------

#define input_str_len 4                // Buffer to hold characters entered
char input_str[input_str_len];         // at the command prompt.

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
```



```
void main(void);

// Initialization Routines
void SYSCLK_Init(void);
void PORT_Init(void);
void UART0_Init (void);
void ADC0_Init (void);
void CACHE_Init (void);
void Timer3_Init(int counts);

// Support Routines
void print_menu(void);
void print_data(char print_raw_data);
void FLASH_Erase(void);
void Cache_Function( unsigned int start_address, unsigned int length );
void Cache_ISR(   unsigned int start_address,
                  unsigned int interrupt_length,
                  unsigned int interrupt_number );


// Cached Routines
void while_TR3_NonBlocking(void);
void ADC0_ISR (void);


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void)
{

   WDTCN = 0xde;                       // disable watchdog timer
   WDTCN = 0xad;

   PORT_Init ();                       // initialize crossbar and GPIO
   SYSCLK_Init ();                     // initialize oscillator
   UART0_Init ();                      // initialize UART0
   ADC0_Init ();                       // initialize ADC0
   CACHE_Init ();

   Timer3_Init(SYSCLK/SAMPLERATE);     // initialize Timer4 to overflow
                                       // and generate interrupts at
                                       // <SAMPLERATE> Hz

   // Cache the ADC0 End of Conversion ISR
   Cache_ISR( (unsigned int) ADC0_ISR, ADC0_ISR_LEN, ADC0_ISR_NUM );

   // Cache the while_TR3_NonBlocking() routine
   Cache_Function( (unsigned int) while_TR3_NonBlocking,
                                  while_TR3_NonBlocking_LEN);

   EA = 1;                             // Enable Global Interrupts

   print_menu();                       // print the command menu

   while (1){
```



```
      SFRPAGE = UART0_PAGE;
      printf("\nEnter a command > ");
      gets(input_str, input_str_len);

      switch ( input_str[0] ){

         // Capture new data
         case '1': FLASH_Erase();               // erase FLASH log

                   SFRPAGE = TMR3_PAGE;
                   TR3 = 1;                     // start Timer3
                   while_TR3_NonBlocking();     // execute: while(TR3);
                                                // Timer3 will be stopped
                                                // after data has been captured.

                   SFRPAGE = UART0_PAGE;
                   printf("\nA new data set has been captured.\n");
                   break;

         // Print Data (Press CTRL+C to stop)
         case '2': print_data(TRUE);
                   break;

         // Print Average ADC Reading
         case '3': print_data(FALSE);
                   break;

         // Print Command List
         case '?': print_menu();
                   break;

         default:  printf("\n*** Unknown Command.\n");
                   break;
      }

   } // end while

}

//-----------------------------------------------------------------------------
// Cached Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// while_TR3_NonBlocking
//-----------------------------------------------------------------------------
//
// This function polls the TR3 bit and returns when TR3 = 0. This
// function should be cached during device initialization to allow servicing
// interrupts during FLASH write operations.
//
void while_TR3_NonBlocking(void)
{
   while(TR3);
}
```

```
//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
//
// This ISR is called on the end of an ADC0 conversion.
//
void ADC0_ISR (void) interrupt 15
{
   // FLASH write pointer
   static unsigned char xdata * idata pwrite = LOG_START;

   AD0INT = 0;                       // clear ADC0 conversion complete
                                     // interrupt flag
   SFRPAGE = LEGACY_PAGE;
   FLSCL |= 0x01;                    // enable FLASH writes/erases
   PSCTL  = 0x01;                    // MOVX writes FLASH byte

   SFRPAGE = ADC0_PAGE;
   *pwrite = ADC0H;                  // store the first 8-bits of the
                                     // 12-bit left-justified ADC0
                                     // result to FLASH
   SFRPAGE = LEGACY_PAGE;
   FLSCL &= ~0x01;                   // disable FLASH writes/erases
   PSCTL  = 0x00;                    // MOVX writes target XRAM

   pwrite++;                         // increment FLASH write pointer

   // Check for the end of the log. If <LOG_END> has been reached
   // disable Timer3 and reset it to its reload value. Also set the
   // FLASH write pointer to <LOG_START>
   if( pwrite >= LOG_END ){

      SFRPAGE = TMR3_PAGE;
      TR3 = 0;                       // Disable Timer3
      TMR3 = RCAP3;                  // Reset Timer3 to its reload value.
      pwrite = LOG_START;            // Reset FLASH write pointer
      return;                        // Exit ISR

   }
}
//-----------------------------------------------------------------------------
// Support Routines
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
// print_menu
//-----------------------------------------------------------------------------
//
// This routine prints the command menu to the UART.
//
void print_menu(void)
{
   SFRPAGE = UART0_PAGE;
   printf("\n\nC8051F12x Non-Volatile Data Capture Example\n");
   printf("---------------------------------------------\n");
   printf("1. Capture new data\n");
   printf("2. Print Data (Press CTRL+C to stop)\n");
   printf("3. Print Average ADC Reading\n");
```



```
   printf("?. Print Command List\n");
}

//-----------------------------------------------------------------------------
// print_data
//-----------------------------------------------------------------------------
//
//
void print_data(char print_raw_data)
{
   long i;
   long result = 0;


   char code * idata pread = LOG_START;         // FLASH read pointer

   // cycle through LOG from start to end
   for( i = 0; i < (LOG_END - LOG_START); i++ ){

         // add up all the samples for an average
         result += pread[i];

         // print each sample to the UART if <print_raw_data> is set
         if(print_raw_data){
            printf("0x%02bX ", pread[i]);
         }
    }


   // print the average to the UART
   SFRPAGE = UART0_PAGE;
   printf("\n\nThe average ADC reading is 0x%02bX\n", (char)(result/i));


}

//-----------------------------------------------------------------------------
// FLASH_Erase
//-----------------------------------------------------------------------------
void FLASH_Erase(void)
{
   char SFRPAGE_SAVE = SFRPAGE;        // Preserve current SFR page
   bit EA_SAVE;                        // Preserve interrupt state
   int i;                              // Software counter
   char xdata * idata pwrite;          // FLASH write/erase pointer

   // Set the FLASH write/erase pointer to the beginning of the FLASH log
   pwrite = LOG_START;

   // Erase all FLASH pages in the FLASH log
   for( i = 0; i < (LOG_END - LOG_START); i += FLASH_PAGESIZE ){

      SFRPAGE = LEGACY_PAGE;

      EA_SAVE = EA;
      EA = 0;                          // Disable interrupts
      FLSCL |= 0x01;                   // Enable FLASH writes/erases
```

```
      PSCTL = 0x03;                    // MOVX writes FLASH byte

      pwrite[i] = 0;                   // Initiate FLASH page erase

      FLSCL &= ~0x01;                  // Disable FLASH writes/erases
      PSCTL = 0x00;                    // MOVX targets XRAM
      EA = EA_SAVE;                    // Restore interrupt state
   }

   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}

//-----------------------------------------------------------------------------
// Cache_ISR
//-----------------------------------------------------------------------------
//
// This routine pushes and locks an entire ISR and its interrupt vector into
// the cache. This function assumes that the interrupt vector and ISR is
// located in the common area.
//
void Cache_ISR(   unsigned int start_address,
                  unsigned int interrupt_length,
                  unsigned int interrupt_number   )
{
   char SFRPAGE_SAVE = SFRPAGE;        // Preserve current SFR page
   char EA_SAVE = EA;                  // Preserve interrupt state

   unsigned char code* pread;          // Pointer used to generate MOVC
                                       // instructions to initiate a cache
                                       // push operation

   unsigned char temp_char;            // Temporary char.

   SFRPAGE = CONFIG_PAGE;              // Set SFR page

   // Check if there is enough room to cache the ISR. When the number of
   // slots required by the ISR + interrupt vector are subtracted from
   // the CHSLOT pointer, the result is the number of unlocked slots available.
   if( ((CCH0LC & 0x3F) - (interrupt_length/4 + 2)) <= 0){
      while(1);                        // Handle Error Condition.
                                       // Not enough room to cache ISR
   }

   // Set the <pread> pointer to the address of the interrupt vector.
   pread = ((interrupt_number * 8) + 3 );

   EA = 0;                             // Disable Interrupts

   CCH0LC &= ~0xC0;                    // Clear the CHPUSH and CHPOP bits
   CCH0LC |=  0x80;                    // Enable cache pushes, MOVC instructions
                                       // will push 4-byte FLASH segments into
                                       // the cache

   // Push the interrupt vector in the cache slot pointed to by CHSLOT.
   // An interrupt vector consists of a 3-byte LJMP instruction and the
   // LSBs of the vector address are always 10b. One byte of the LJMP
   // instruction fall in the first 4-byte FLASH segment and the remaining
```



```
   // two bytes of the instruction falls in the second 4-byte segment.
   // Two cache pushes are neccessary.

   temp_char = *pread;                // Push the first 4-byte segment (MOVC)
   temp_char = *(pread + 4);          // Push the second 4-byte segment (MOVC)

   // Push the interrupt service routine in the cache. The interrupt starts
   // at <START_LABEL> and is <interrupt_length> bytes long.

   pread = (unsigned char code*) start_address;
                                      // Set <pread> to the address of the
                                      // interrupt service routine.

   while( pread < start_address + interrupt_length ){

      temp_char = *pread;             // Push a 4-byte segment (MOVC)

      pread += 4;                     // Increment <pread> to the next
                                      // segment.
   }


   CCH0LC &= ~0x80;                   // Clear CHPUSH to disable cache pushes.
   SFRPAGE = SFRPAGE_SAVE;            // Restore SFR page.
   EA = EA_SAVE;                      // Restore interrupt state.

}

//-----------------------------------------------------------------------------
// Cache_Function
//-----------------------------------------------------------------------------
//
// This routine pushes and locks a function into the cache. This routine
// assumes that the function is located in the common area.
//
//
void Cache_Function( unsigned int start_address, unsigned int length )
{
   char SFRPAGE_SAVE = SFRPAGE;       // Preserve current SFR page
   char EA_SAVE = EA;                 // Preserve interrupt state

   unsigned char code* pread;         // Pointer used to generate MOVC
                                      // instructions to initiate a cache
                                      // push operation

   unsigned char temp_char;           // Temporary char.

   SFRPAGE = CONFIG_PAGE;             // Set SFR page

   // Check if there is enough room to cache the function. When the number of
   // slots required by the function are subtracted from
   // the CHSLOT pointer, the result is number of unlocked slots available.
   if( (CCH0LC & 0x3F) - (length/4) <= 0){
      while(1);                       // Handle Error Condition.
                                      // Not enough room to cache function
   }
```

```
   // Set <pread> to the address of the function
   pread = (unsigned char code*) start_address;

   EA = 0;                              // Disable Interrupts

   CCH0LC &= ~0xC0;                     // Clear the CHPUSH and CHPOP bits
   CCH0LC |=  0x80;                     // Enable cache pushes, MOVC instructions
                                        // will push 4-byte FLASH segments into
                                        // the cache

   // Push the function in the cache. The interrupt starts
   // at <START_LABEL> and is <length> bytes long.
   while( pread < start_address + length ){

      temp_char = *pread;               // Push a 4-byte segment (MOVC)

      pread += 4;                       // Increment <pread> to the next
                                        // segment.
   }


   CCH0LC &= ~0x80;                     // Clear CHPUSH to disable cache pushes.
   SFRPAGE = SFRPAGE_SAVE;              // Restore SFR page.
   EA = EA_SAVE;                        // Restore interrupt state.

}

//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use an external 22.1184 MHz
// crystal oscillator multiplied by a factor of 9/4 using the PLL as its
// clock source. The resulting frequency is 22.1184 MHz * 9/4 = 49.7664 MHz
//
void SYSCLK_Init (void)
{
   int i;                            // delay counter

   char SFRPAGE_SAVE = SFRPAGE;      // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;            // set SFR page

   OSCXCN = 0x67;                    // start external oscillator with
                                     // 22.1184MHz crystal

   for (i=0; i < 256; i++) ;         // Wait for osc. to start up

   while (!(OSCXCN & 0x80)) ;        // Wait for crystal osc. to settle

   CLKSEL = 0x01;                    // Select the external osc. as
                                     // the SYSCLK source
```

```
   OSCICN = 0x00;                    // Disable the internal osc.

   //Turn on the PLL and increase the system clock by a factor of M/N = 9/4
   SFRPAGE = CONFIG_PAGE;

   PLL0CN  = 0x04;                   // Set external osc. as PLL source
   SFRPAGE = LEGACY_PAGE;
   FLSCL   = 0x10;                   // Set FLASH read time for 50MHz clk
                                     // or less
   SFRPAGE = CONFIG_PAGE;
   PLL0CN |= 0x01;                   // Enable Power to PLL
   PLL0DIV = 0x04;                   // Set Pre-divide value to N (N = 4)
   PLL0FLT = 0x01;                   // Set the PLL filter register for
                                     // a reference clock from 19 - 30 MHz
                                     // and an output clock from 45 - 80 MHz
   PLL0MUL = 0x09;                   // Multiply SYSCLK by M (M = 9)

   for (i=0; i < 256; i++) ;         // Wait at least 5us
   PLL0CN  |= 0x02;                  // Enable the PLL
   while(!(PLL0CN & 0x10));          // Wait until PLL frequency is locked
   CLKSEL  = 0x02;                   // Select PLL as SYSCLK source

   SFRPAGE = SFRPAGE_SAVE;           // Restore SFR page
}


//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// This routine configures the crossbar and GPIO ports.
//
void PORT_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;      // Save Current SFR page

   SFRPAGE = CONFIG_PAGE;            // set SFR page

   XBR0     = 0x04;                  // Enable UART0
   XBR1     = 0x00;
   XBR2     = 0x40;                  // Enable crossbar and weak pull-up


   P0MDOUT |= 0x01;                  // Set TX0 pin to push-pull
   P1MDOUT |= 0x40;                  // Set P1.6(LED) to push-pull

   SFRPAGE = SFRPAGE_SAVE;           // Restore SFR page

}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <baudrate> and 8-N-1. In order to
// increase the clocking flexibility of Timer0, Timer1 is configured to count
// SYSCLKs.
```

```
//
// To use this routine SYSCLK/BAUDRATE/16 must be less than 256. For example,
// if SYSCLK = 50 MHz, the lowest standard baud rate supported by this
// routine is 19,200 bps.
//
void UART0_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;      // Save Current SFR page

   SFRPAGE = UART0_PAGE;

   SCON0   = 0x50;                   // SCON0: mode 0, 8-bit UART, enable RX
   SSTA0   = 0x10;                   // Timer 1 generates UART0 baud rate and
                                     // UART0 baud rate divide by two disabled
   SFRPAGE = TIMER01_PAGE;
   TMOD    &= ~0xF0;
   TMOD    |=  0x20;                 // TMOD: timer 1, mode 2, 8-bit reload

   TH1 = -(SYSCLK/BAUDRATE/16);      // Set the Timer1 reload value
                                     // When using a low baud rate, this equation
                                     // should be checked to ensure that the
                                     // reload value will fit in 8-bits.

   CKCON |= 0x10;                    // T1M = 1; SCA1:0 = xx


   TL1 = TH1;                        // initialize Timer1
   TR1 = 1;                          // start Timer1

   SFRPAGE = UART0_PAGE;
   TI0 = 1;                          // Indicate TX0 ready

   SFRPAGE = SFRPAGE_SAVE;           // Restore SFR page
}

//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Configure ADC0 to use Timer 3 overflows as a conversion source and to
// use left-justified output mode.
//
void ADC0_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;      // Preserve current SFR page

   SFRPAGE = ADC0_PAGE;

   ADC0CN = 0x85;                    // ADC0 enabled; normal tracking
                                     // mode; ADC0 conversions are initiated
                                     // on Timer3 overflows; ADC0 data is
                                     // left-justified

   REF0CN = 0x07;                    // enable temp sensor, on-chip VREF,
                                     // and VREF output buffer
   AMX0SL = 0x0F;                    // Select TEMP sens as ADC mux output
```

```
   ADC0CF = ((SYSCLK/2500000) << 3);// ADC conversion clock = 2.5MHz

   ADC0CF |= 0x01;                  // PGA gain = 2

   EIE2 |= 0x02;                    // Enable ADC0 End-of-conversion
                                    // interrupts

   SFRPAGE = SFRPAGE_SAVE;          // Restore SFR page.
}


//-----------------------------------------------------------------------------
// CACHE_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the cache by enabling FLASH block writes
//
void CACHE_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;     // Preserve current SFR page

   SFRPAGE = CONFIG_PAGE;

   CCH0CN |= 0x01;                  // Enable Block Writes to FLASH

   SFRPAGE = SFRPAGE_SAVE;          // Restore SFR page.

}

//-----------------------------------------------------------------------------
// Timer3_Init
//-----------------------------------------------------------------------------
// This routine initializes Timer3 in auto-reload mode to generate interrupts
// at intervals specified in <counts>.
//
void Timer3_Init (int counts)
{
   char SFRPAGE_SAVE = SFRPAGE;     // Preserve current SFR page

   SFRPAGE = TMR3_PAGE;

   TMR3CN = 0;                      // STOP timer; set to auto-reload mode
   TMR3CF = 0x08   ;                // Timer3 counts SYSCLKs
   RCAP3 = -counts;                 // set reload value

   TMR3 = RCAP3;                    // Initialize Timer3 to its reload value

   SFRPAGE = SFRPAGE_SAVE;          // Restore SFR page.
}
```