

Software Developer's Guide

iMOTION™ motor control IC with additional MCU

About this document

Scope and purpose

The IRMCx300 series motor control ICs are mixed signal devices optimized for permanent magnet motor control. They combine the iMOTION™ motion control engine (MCE) with an additional 8 Bit microcontroller (MCU) to improve application flexibility.

The MCE can perform sensor less FOC over the full speed range of the motor, including providing torque at zero speed and stable control at deep field weakening speeds. The IRMCx300 series motor control ICs can be combined with a μ IPM™ or a discrete power stage to implement a complete inverter for the control of a PMSM. Infineon is offering its ready-to-use and field proven FOC control algorithm along with the IRMCx300 products making it extremely easy for customers to get started.

In addition to controlling the motor the IRMCx300 series can also perform the power factor correction (PFC).

The integrated standard 8051 MCU can be programmed by the customer to interact with the MCE and to implement additional functionality running almost independently from the motion control algorithm on the MCE. A number of peripheral devices and special functions have been added to customize the operation for motor control applications.

The 8051 and MCE interact through a shared RAM, accessible by both processors.

The purpose of this guide is to describe the implementation of 8051 microprocessor control for use in the IRMCx300 series of motion control ICs. This document covers required initializations, settings and functions for 8051 control of the IC, as well as programming of the OTP memory using Infineon's demo boards or by in-circuit programming. Some examples are presented. This guide assumes that the user has experience with embedded software development and is also familiar with the application developer's guide and one of Infineon's reference design kits.

Code development should be performed with the RAM version of the ICs (IRMCF3xx) with the final production ready code than written to the OTP (one-time programmable) of the IRMCK3xx series ICs.

Alternatively an external EEPROM can be used for the IRMCF3xx series.

An additional document, the Reference Manual, also referred to frequently in this document, has detailed information on many topics covered here, as well as full descriptions of the MCE & MCU hardware registers and programming methods.

Intended audience

This software developer's guide is intended for customers implementing an inverterized drive.

User Guide #0610 V1.2

IRMCx300 Software Developer's Guide

By International Rectifier's iMotion Team

Table of Contents

	Page
1 Introduction	2
1.1 Purpose.....	2
1.2 Requirements.....	2
1.3 Overview	5
1.4 Boot Process.....	5
1.5 Memory Map	7
2 MCEDesigner Agent	11
2.1 Sequencer.....	11
2.2 MceInfo Structure.....	12
3 Setting Up the 8051 Development Tools	13
3.1 Software Setup.....	13
3.2 Hardware Setup	15
3.3 Keil uVision Project Options.....	16
4 Sample Code	22
4.1 Sample Code Structure.....	22
4.2 Running the Motor.....	26
4.3 Extending Functionality	29
4.4 Troubleshooting	30
5 Programming the Control IC	31
5.1 Programming the EEPROM with MCEDesigner	31
5.2 Using MCEProgrammer	32
6 Migrating from the F-version to the K-version	34
6.1 System Differences	34
6.2 Checking for MOVX Instruction Sequences.....	37
6.3 Using MCEProgrammer2 to Program the OTP	43
6.4 Creating Custom Programming Methods.....	44

Paragraph annotation of the contents of this User Guide.

1 Introduction

The 8051 microprocessor, included in the IRMCx300 series of motion control ICs, can be used to implement a large variety of control and protection functions for motor control applications. The instruction set and basic operation of the IRMCx300 Series 8051 microprocessor is consistent with the standard Intel 8051 processor. A number of peripheral devices and special functions have been added to customize the operation for motor control applications.

The IRMCx300 series ICs contain two processors: an 8051 processor and the Motion Control Engine (MCE). The 8051 and MCE interact through a shared RAM, accessible by both processors. The MCE is designed specifically to implement motor control loops, process feedback signals, and calculate PWM switching signals. The 8051 mediates between external control signals (such as the front panel of a washing machine) and the MCE, which ultimately produces the signals that operate the motor.

The 8051 software application controls and monitors the operation of the MCE through the read/write register interface of the shared RAM. The 8051 code (MCEDesigner Agent) used with the MCEDesigner tool can do this in two ways: in a simple lock-step manner, where MCEDesigner specifies each register to be read or written individually and the 8051 software performs only those operations as they are requested; or, for a limited number of functions, the 8051 will perform a sequence of operations, as described in Section 2.1. An 8051 user application, on the other hand, would typically perform entire sequences of operations automatically or in response to simple input commands such as “start” and “stop.”

1.1 Purpose

The purpose of this guide is to describe the implementation of 8051 microprocessor control for use in the IRMCF300 and IRMCK300 series of motion control ICs. This document covers required initializations, settings and functions for 8051 control of the IC. Some examples and the sample code, IRSamples, are presented. This application note assumes that the user has experience with embedded software programming and is also familiar with the Application Developer's Guide and one of IR's Reference Design Kits.

The sample code and examples given here are intended to allow the designer to create a control interface to replace MCEDesigner once the application development has been completed. One of the main tasks is to recreate MCEDesigner functions in the 8051 code. Code development should be performed with the RAM version of the IC (IRMCF300), for easy revision and downloading. After code development and testing with the F-version of the IC, the embedded 8051 code is intended to be written to the OTP (one-time programmable) ROM of the IRMCK300 series ICs. The last section of this guide details the process of migrating the application code from the F-version to the K-version of the IC. This guide will also describe the process of creating and downloading of EEPROM code and the related boot load process for the IRMCF300 IC and, analogously, the process of creating and writing the OTP image.

1.2 Requirements

The following software and hardware is required for 8051 application code development:

1. FS2 ISA-M8051EW Debugger with Keil uVision driver (FS2 debug pod), available at: <http://www.mips.com/products/software-tools/legacy/8051/>
2. Keil PK51 Professional Developers Kit (Keil uVision2 or uVision3)

Using Keil uVision, the developer can write the control program in the C programming language and then compile it into machine code for download to the IRMCF300 Series IC for testing. uVision provides a simulation mode so that the portions of the program which are not hardware-dependent can be tested without downloading to the actual IC. The sample source code, IRSamples, provided with the Reference Design Kit was developed with Keil uVision.

In addition to the requirements above, the circuit board containing the IRMCF300 IC should contain the appropriate connectors, drivers, and isolators to interface with the FS2 hardware. If IR's Reference Design Boards are used, the appropriate circuits are built in, with proper isolation for the FS2 hardware connection. If any other hardware is used, follow the instructions in the warning below.



Warning!

When connecting the FS2 debug pod to the circuit board, the FS2 hardware can be damaged by the high voltage on the board if appropriate isolation is not used. The problem arises because the DC bus minus (GND) is not at the same potential as Earth (or wall) ground. For this reason, if proper isolation is not used, it is recommended that the board be powered by a DC power supply with isolated ground when using the FS2 hardware.

1.2.1 FS2 USB-based Debugger

Recently, a USB based JTAG debugger was released to replace the parallel port based model. The USB version has a 10-pin header and bidirectional reset pin, in lieu of the 20-pin header and dual reset lines. *An adapter cable is available from the vendor, along with a schematic of the required changes if an in-house cable needs to be created.* The options to change the USB FS2 to a one-way reset are shown below. All of the Reference Design Kits have been updated to interface with the 10-pin cable, but some IRMCS3041 kits in inventory may still have the old 20-pin JTAG interface header. The IRMCS3012 and IRMCS3043 kits are configured to use the 20-pin interface by default; to change to the 10-pin header, follow the instructions “Board Conversion” below.

Cable Conversion

A schematic including the connector part numbers is shown in Figure 1 below. This can either be implemented as a daughter card with connectors or as a single ribbon cable. Notice that the ‘Reset’ pin is now disconnected. This is the reset signal sent from the target to the FS2. This signal is not required for debugging so long as the ‘reset button’ located on IR development boards is not pressed. Pressing this button will cause the device to reset, but the debugger will not be aware of it.

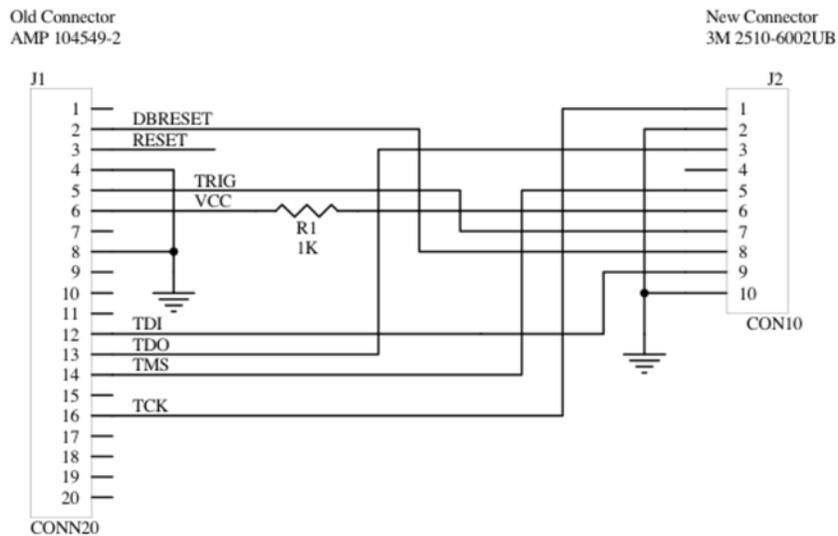


Figure 1—FS2 Cable Conversion Schematic

Board Conversion

The IRMCS3012 and IRMCS3043 boards can be changed from the old header (20-pin) to the new header with a few component changes. The boards are configured to the 20-pin header by default. To convert:

- 1) Remove Q7, U11 and R173
- 2) Install 0Ω resistors into R169 and R172
- 3) Setup FS2 Reset polarity as described below.

FS2 Configuration

The FS2 configuration will be different depending on which header is used. If the 20-pin header is used, the following modifications must be made in the FS2 ‘configuration’ dialog shown in Figure 2. These settings ensure that the reset signal is now one way to the target and that the proper reset logic levels are used.

- 1) ResetNegated = ‘low’
- 2) ResetAsserted = ‘high’
- 3) Tck Rate = 500000 (500Khz)

If the 10-pin header is used, setup the FS2 as below to correctly configure for a bidirectional reset.

- 1) ResetNegated = ‘high’
- 2) ResetAsserted = ‘low’
- 3) Tck Rate = 500000 (500Khz)

For either header, all other options should *not* be modified, as their default values are correct for this application. Be careful not to reduce the Tck rate far below or above 500Khz as unpredictable behavior may result.

When using Keil be sure to use the new driver included on the USB FS2 CD and not the older parallel port one. The ‘FS2 Getting Started Guide’ covers exactly what to do. Be sure to install Keil first, then FS2, so that the FS2 installation configures Keil to correctly control the FS2 hardware.



Figure 2—FS2 Configuration Dialog

1.3 Overview

There are two main parts of the code required for proper operation of the IRMCF300 Series IC, the 8051 program and the MCE program, corresponding to the two processors in the IC. The MCE program provides the PWM synchronous operations for calculation of the proper inverter gating. A standard MCE program is supplied by IR in the form of "IRMCF341_appl_layers.bin," for the IRMCF341, for example. This program may be modified by the user, but it will have the same file extension, ".bin". The .bin file is the primary output of MCE Compiler. The 8051 provides application level control and asynchronous interrupts to modify settings in the MCE program. This is generally developed by the user, though IR provides sample code called "IRSamples.hex." The .hex file is the output of the Keil uVision compiler. In this document, the MCE and 8051 code images may be referred to as the .bin file and .hex file, respectively.

The next section describes the boot process of the F-version of the IC. During the boot process, the on-chip RAM of the IC is loaded from the EEPROM. This boot process relies on the EEPROM data to have the proper format; this format is specified in detail in the Reference Manual, though the IR provided utility, IRProgrammer, can generate the EEPROM image for the developer. At power up, the 8051 and MCE processor programs are loaded from the EEPROM into the shared RAM of the IC, a map of which is provided in Section 1.5. The two processors have different data addressing and byte ordering schemes, which are described in Section 1.5.2.

Section 5 of this guide covers the IR supplied programming tools that can be used to automatically create a formatted EEPROM image. These tools, in combination with an FS2 Debugger, can be used for direct EEPROM programming without the need for dedicated 8051 code.

The .hex file, whose format is not described, is standard Intel hex format. Information on the Intel hex record format can be found here:

http://www.keil.com/support/man/docs/oh166/oh166_ih_record.htm

1.4 Boot Process

The reset and boot processes are closely tied together. The boot process is automatically accomplished following a proper reset sequence. Reset of the IC is triggered by any of the following events:

- 1) Power up
- 2) Under voltage lockout (UVCC) detects low voltage on 1.8V.
- 3) The watchdog timer times out.
- 4) External reset, achieved by holding RESET pin low for a minimum of 10usec.

A user application cannot intervene during the reset and boot process. The main task of the boot process is to copy the user application program stored in an external serial EEPROM to program RAM, initialize the program counter and transfer control to the 8051 CPU.

Once the reset is recognized in the system, the reset module counts up to 2048 clocks at the crystal frequency (i.e. 4 MHz, 512 μ sec) to insure that the internal PLL becomes stable for generation of the internal system clock. When this waiting period is complete, the boot module begins copying the user program from external EEPROM to program RAM via the I2C port. The time to complete the copy process depends on the size of the user program. The following example calculates the time required to copy a 16K-byte program (total of both .hex and .bin files):

Total user program to be copied: **16 * 1024 bytes**
Number of bit periods to transfer one byte: **9 bits**

I2C clock speed: **333 kHz**, for a transfer time of **3 µsec per bit**
Total transfer time: **16 * 1024 * 9 * 3 µsec = 442 msec**

The time to complete the copy process varies depending on the actual size of the application program and the I2C clock speed, which can be modified to accommodate various types of I2C EEPROM devices. Immediately after the copy process completes, the 8051 application program begins execution. For more detail on the boot process and setting the I2C clock speed, please refer to the Reference Manual.

A 16-bit checksum follows the last byte of boot data, low-order byte first. The checksum is calculated on all preceding bytes in EEPROM by summing all the bytes (one byte at a time, ignoring any overflow beyond 16 bits), then negating (two's complement) the resulting value. When the IRMCF300 IC validates the checksum by adding all bytes, the 16-bit result should be zero.

If it is not zero, the system halts with a checksum error and does not transfer control to the 8051 application program. There is no external indication of the error and no further operations are performed until the device is reset. To diagnose such an issue, check the I2C bus during boot (right after a reset). The SCL pin should show clock pulses, as data is transferred from the EEPROM to the control IC, which should cease after the transfer. This indicates that the IC is not damaged and is working properly. Next, use the JTAG interface to reprogram the EEPROM with a known good image. If this image operates correctly, then there is strong likelihood that there is a checksum error with the original image.

1.5 Memory Map

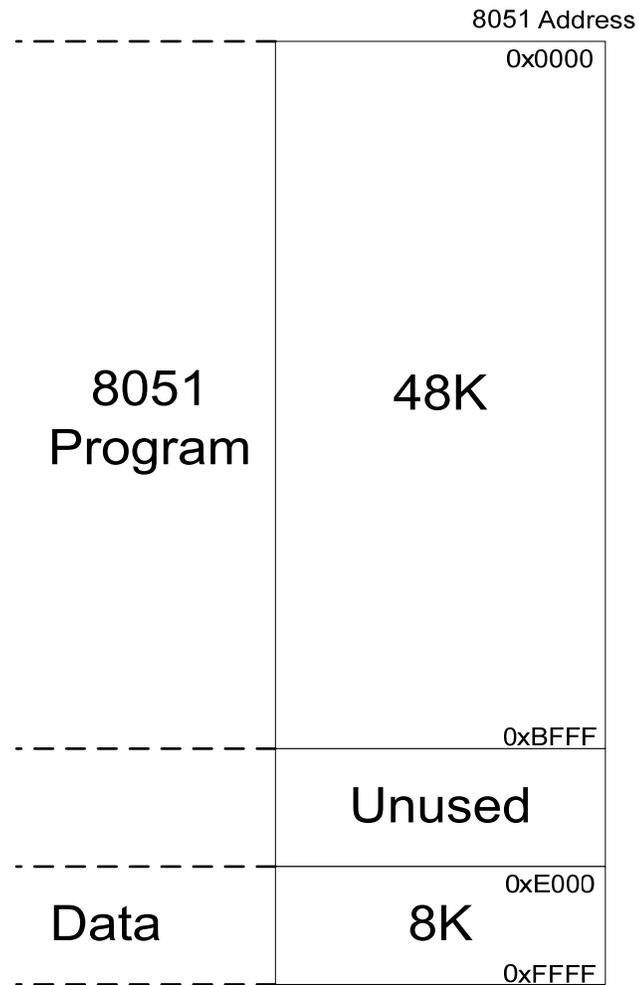


Figure 3—Memory Map of the IRMCF300 Series RAM

Figure 3 shows the default memory map of the IRMCF300 Series, which contains 48k bytes of program and 8k bytes of data RAM. The lower 48k bytes of the RAM are allocated for the 8051 program (8051 addresses: 0x0000 – 0xBFFF). The top (highest memory address) 2k bytes of the RAM are allocated as 8051 data (8051 RAM addresses: 0xF800 – 0xFFFF), which is shown in Figure 4. The program and data RAM should be defined in the Keil compiler settings. For example, for the default memory map, set the following fields, found at the **BL51 Locate** tab of **Options for Target**:

Code Range: 0x0000-0xBFFF

Xdata Range: 0xF800-0xFFFF (2K)

The memory addresses shown in Figure 3 are from the point of view of the 8051 processor. Memory addressing for the MCE is described below.

1.5.1 The Dual-Port RAM

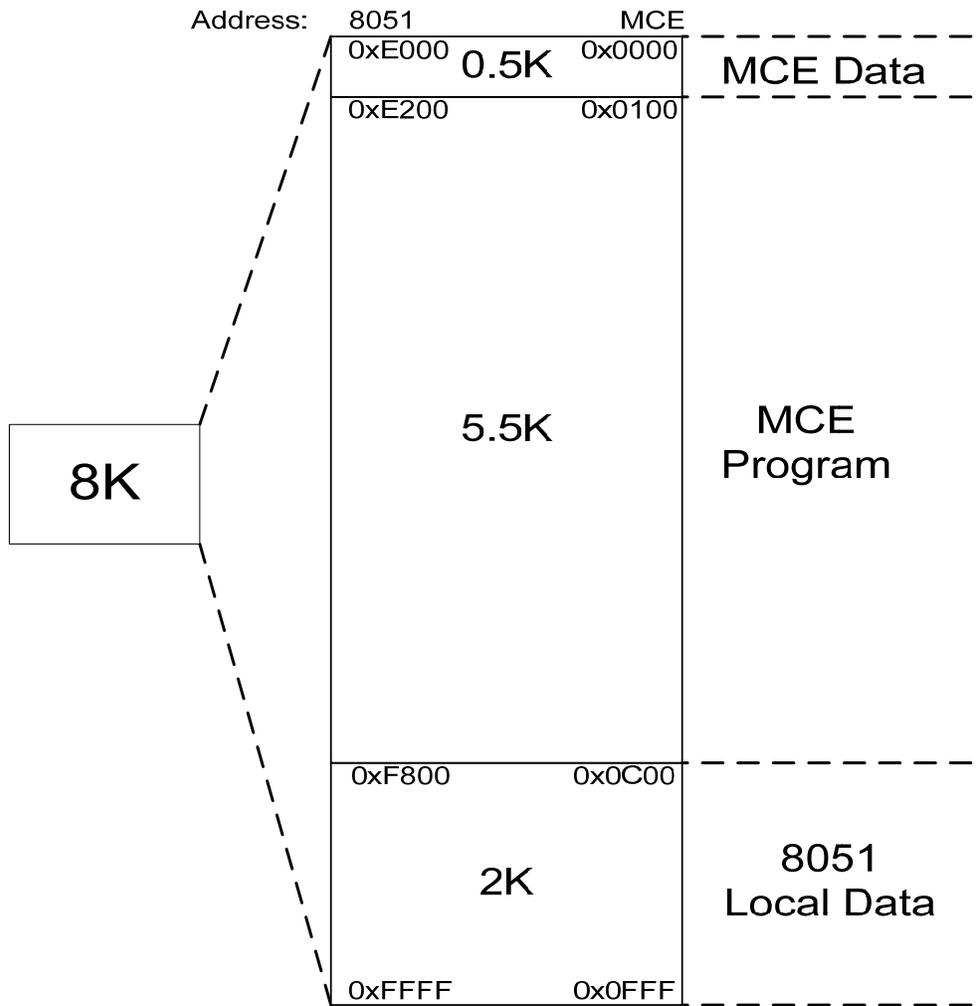


Figure 4—Shared RAM area of IRMCF300 Series ICs

There are 8K bytes of data RAM that are accessible to both the 8051 and MCE processors. The shared RAM (8051 external addresses: 0xE000 – 0xF7FF) comprises three sections, the locations of which are given in Table 2 below.

Typically 512 bytes are allocated for MCE data beginning at address 0xE000. These locations are used for MCE private storage and for information passed between the MCE and the 8051. Both the 8051 and the MCE access this area of RAM for reading and writing. Up to an additional 5.5k bytes are allocated for MCE instruction (program) space. The hardware loads the MCE program into this area of RAM at power up (as described in Section 1.4). The 8051 should not read or write this area of RAM during normal operation. The upper (highest memory address) 2k bytes of RAM are available for 8051 data storage. The MCE does not access this area.

The boundaries between the three sections of RAM are dynamic and determined by the MCE compiler at compilation time. The compiler always reserves 512 bytes for MCE data at address 0xE000, beginning the MCE program at address 0xE200. Depending on the size of the MCE application program, the compiler may allocate less than the allotted 5.5k bytes for MCE program, in which case more memory could be used for 8051 data RAM. For example, if the MCE program is smaller than 3.5k bytes, 8051 data could begin at address 0xF000 instead of 0xF800.

Remember that the 8051 data range is defined in the Keil compiler options, as noted above. Also, note that the MCE web compiler displays the total MCE program size in words (16 bit units). Therefore, the displayed size must be multiplied by two to determine the program size in bytes.

For reference, Table 1 shows the size of the MCE reference designs. The size of the MCE program code does not include the header, so it is slightly smaller than the .bin file.

IRMCF3xx IC	Size (bytes)
371, 341	1588
343	2516
311, 312	3398

Table 1— MCE program file sizes

1.5.2 8051 vs. MCE Addressing

The only memory space which the MCE has access to is the shared RAM. Therefore, the MCE address of 0x0000 corresponds to the address 0xE000 on the 8051 memory bus, as depicted in Figure 4, above.

RAM Section	Size	8051 Address Range	MCE Address Range
MCE Data RAM	0.5K bytes	0xE000 – 0xE1FF	0x0000 – 0x00FF
MCE Program RAM	5.5K bytes	0xE200 – 0xF7FF	0x0100 – 0x0BFF
8051 Data RAM	2K bytes	0xF800 – 0xFFFF	0x0C00 – 0x0FFF

Table 2—Memory addressing in the shared RAM

Table 2, above, specifies one address for 8051 use and another for MCE use. The MCE only has access to the shared RAM, and the 8051 address of 0xE000 corresponds to MCE address of 0x0000. The 8051 addresses the RAM 8 bits (1 byte) at a time, while the MCE addresses the RAM 16 bits at a time, requiring half the memory addresses to access the same data space. This is reflected in the memory addresses listed in the table and Figure 4.

1.5.3 Byte Ordering

The Keil compiler used for 8051 software development generates code that uses big endian byte ordering to store 16-bit and 32-bit values in memory. The MCE is a 16-bit processor and uses little endian byte ordering for data storage. *Functions to correctly read and write the shared RAM are included in the sample code, IRSamples. These functions correctly swap bytes when necessary, and lock out bus accesses to prevent data being read by one processor while it is still being written by the other.*

Byte ordering refers to the convention used to store 16-bit and 32-bit values in memory using a processor, such as the 8051, that has a native addressing mode of 8 bits. The two standard byte

ordering conventions are “big endian” or “Motorola” byte ordering and “little endian” or “Intel” byte ordering.

In big endian byte ordering, the “big end” of a value is stored first. That is, the high order byte is stored at the lowest memory address and the low-order byte is stored at the highest memory address. In little endian byte ordering the “little end” is stored first, with the low-order byte at the lowest memory address.

For example, suppose the 16-bit value 0x2345 is to be stored in memory at address 0x1000. Using big endian byte ordering, 0x23 is stored at address 0x1000 and 0x45 is stored at address 0x1001. Using little endian byte ordering, 0x45 is stored at address 0x1000 and 0x23 is stored at address 0x1001. Table 3 below shows how the value 0x456789AB would be stored at address 0x1000 using each of the byte ordering conventions.

Address	Big Endian	Little Endian
0x1000	0x45	0xAB
0x1001	0x67	0x89
0x1002	0x89	0x67
0x1003	0xAB	0x45

Table 3—Big and little endian byte conventions

The Keil compiler used for 8051 software development generates code that uses big endian byte ordering to store 16-bit and 32-bit values in memory.

The MCE is a 16-bit processor and uses little endian byte ordering for data storage. The smallest unit of data storage on the MCE processor is 16 bits (it cannot access a single byte in memory). The shared RAM used to exchange information between the 8051 and MCE processors is 8-bit addressable to the 8051, but 16-bit addressable to the MCE.

The MCE expects all data shared between the 8051 and MCE processors to be in little endian byte ordering. This means that the 8051 must swap bytes before writing to shared RAM and swap bytes after reading from shared RAM. Table 4 below shows how the value 0x456789AB would be correctly stored by the 8051 for sharing with the MCE. Note that the 8051 reads and writes a byte at a time, but the MCE always accesses the memory a word (16 bits) at a time.

8051 Address	8051 Bytes	MCE Address	MCE Words
0xE200	0xAB	0x0100	0x89AB
0xE201	0x89		
0xE202	0x67	0x0101	0x4567
0xE203	0x45		

Table 4—Byte ordering for sharing of data between 8051 and MCE

1.5.4 Synchronizing 8051 Register Access with the MCE

In user applications, the 8051 may be required to monitor or modify MCE registers during motor operation. There are some risks of errors when reading or writing to the dual-port RAM. To guard against problems, the designer can use the SYNC interrupt to synchronize 8051 access with the PWM cycles which dictate the updating of the inverter gating signal duty cycles.

The SYNC interrupt is generated from the MCE to signal the 8051 that a SYNC pulse has occurred. The SYNC interrupt is a periodic event signal generated by the MCE. Its timing is illustrated in Figure 5. This is the most important signal used for synchronization between the 8051 (CPU side) and the MCE (motion control side). An 8051 application software task that needs to pass commands to the MCE and/or receive updated data from the MCE may require specific synchronization with the MCE. This is due to the fact that MCE computation is initiated

and triggered by the SYNC pulse at every PWM carrier frequency period. It is also true that six PWM outputs to the power device gate drive will occur at exactly one clock moment of the system clock at the beginning of the SYNC event. If synchronization is not implemented and the 8051 application software writes multiple data items to the MCE via the shared RAM, it is possible that some of the data are written in the previous MCE scan period while the rest of data are written in the current MCE scan period. Therefore, 8051 application software should use the SYNC signal for synchronization to insure that multiple data items are updated or read coherently within a particular scan period.

The SYNC signal is also generated in an execution overrun fault condition, which occurs if the MCE does not complete its processing (indicated by the bar labeled “MCE computation” in Figure 5) before the end of the PWM period (*i.e.*, before the next SYNC pulse).

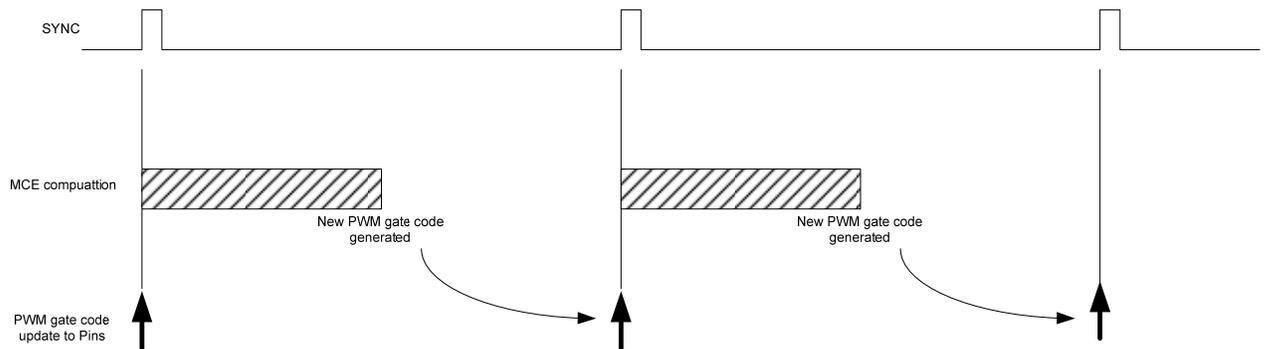


Figure 5. Timing of Sync and MCE Computation

A SYNC interrupt is generated for each component, motor 1 (M1), motor 2 (M2), and the PFC, regardless if they are enabled in the IC. If some component is not used (for example the IRMCx341 does not use M2 or PFC), set its PWM frequency to the same as that of another component and also set the PwmSyncEnb bit to 1. This will ensure that the correct number of SYNC interrupts is generated, with the correct timing. To distinguish between the SYNC interrupts of the components, read the SYNCs register described in the Reference Manual.

2 MCEDesigner Agent

Up to this point, the main interface to the motor control IC has been MCEDesigner. Provided with the Reference Design Kit is 8051 code designed to communicate with MCEDesigner over the UART interface; this code is referred to as “MCEDesigner Agent.” The Agent’s main purpose is to receive commands from the 8051 and, as a result, write to the MCE registers to perform the high level motion control, such as start, stop, and change speeds.

2.1 Sequencer

The MCEDesigner Agent makes use of a sequencer structure to perform certain functions, which are detailed below. In the case of a sequencer function, a number is written to an 8051 register, SeqCmd, which specifies which function to run. The MCEDesigner Agent then executes a sequence of register writes and delays to perform the function. The sequencer simplifies the basic operations of motor control for the designer. To see the detailed list of actions that sequencer functions execute, check the implementation of the functions in IRSamples. The functions which the sequencer executes are:

- Start

- Stop
- Fault Clear
- Catch-Spin

In contrast, the Agent can be directed to write to specific MCE registers as specified in a designer-created MCEDesigner function. This method gives the designer full control over the specific instructions and settings of the MCE.

2.2 MceInfo Structure

When MCEDesigner reads an MCE program file (.bin) and formats it for storage in EEPROM, it strips the header and creates the MceInfo structure, which contains the same information as the header in a slightly different format. MCEDesigner formats the EEPROM to include the 8051 program, the MCE program and the MceInfo structure. The MceInfo structure is loaded to a fixed location in 8051 program RAM during the hardware boot process.

The MceInfo structure is used by MCEDesigner to verify that the Register Map ID of the .irc file matches the Version ID of the MCE program. If they do not match, then MCEDesigner give an error. More information on this error can be found in the Application Developer's Guide.

The MceInfo structure is not required by the hardware boot process, but is a standard component of the 8051 MCEDesigner agent software as well as the 8051 sample source code. An 8051 application developed by the user is not required to define or make use of the MceInfo structure, although the structure will be included in the EEPROM image if MCEDesigner is used to format and program EEPROM.

The MceInfo structure is defined (in "C" language format) as follows:

```
typedef struct
{
    unsigned char validation [ IDV_VALID_LENGTH ];
    unsigned char numIdvDesignIdBytes;
    char idvDesignId [ MAX_IDV_DESIGN_ID_LENGTH + 1 ];
    unsigned char numIdvVersionBytes;
    char idvVersion [ MAX_IDV_VERSION_LENGTH + 1 ];
    unsigned char xdata * pLoadAddr;
    unsigned short loadSize;
    unsigned char xdata * pExecAddr;
    unsigned short xdata * pTraceAddr;
    unsigned char sysTracePage [ 16 ];
    unsigned short PageBase [ 8 ];
} MCE_INFO;
```

The relationships between fields of the MceInfo structure and fields of the binary file header are shown in the table below.

MceInfo Field	Binary Header Field	Note
validation	n/a	This field is added by MceDesigner and has no correspondence in the binary file header. MCEDesigner sets this field to the ASCII string "iMOTION" and the 8051 software uses it as an indication that a valid MceInfo structure has been copied from EEPROM.
numIdvDesignIdBytes	ILEN	The length of the design ID portion of the IDV string (preceding the newline). The length does

		not include the NULL terminator on the idvDesignId string.
idvDesignId	IDV	The design ID portion of the IDV string (preceding the newline character in IDV). The idvDesignId string is NULL terminated.
numIdvVersionBytes	ILEN	The length of the version portion of the IDV string (following the newline). The length does not include the NULL terminator on the idvVersion string.
idvVersion	IDV	The version portion of the IDV string (following the newline character in IDV). The idvVersion string is NULL terminated.
pLoadAddr	LOAD	
loadSize	PGMLEN	
pExecAddr	EXEC	
pTraceAddr	TRCBASE	
sysTracePage	RTMAP	The TracePage table (second 16 bytes of RTMAP)
PageBase	RTMAP	The PageBase table (first 16 bytes of RTMAP)

Below, the user will find more information about how the IDV strings and RTMAP are used by MCE Compiler:

IDV: The string has a variable length, specified by the value of the ILEN field. There is no NULL terminator or pad character following the last byte of the string. The RTLEN field of the header immediately follows the last byte of the IDV string. The IDV string identifies the Simulink design that was compiled to create the MCE binary file. It is made up of the name of the model file (minus the “.mdl” extension) and the version number of the model file (which is created automatically by Simulink and updated whenever the model file is saved). Within the IDV string, the design name and version number are separated by a newline character (0x0A). MCEDesigner uses the IDV string to verify that its current database (loaded from a configuration “.irc” file) is consistent with the MCE program loaded to memory on the target platform.

RTMAP: This field contains the PageBase table (8 16-bit words) followed by the TracePage table (16 8-bit words). The PageBase table provides the base addresses of the read and write registers defined in the Simulink design. The registers are divided into eight sections, with a base address for each section. (Depending on the design and the device type, not all sections are used.) The optional header file output by the MCE Compiler includes the definition and initialization of a PageBase table containing the same information that is stored in the binary file header. The TracePage table contains the page number (0 – 3) for each of the sixteen trace data items allowed in the Simulink design. The TracePage table is associated with MCEDesigner’s data monitoring feature, and serves no purpose if MCEDesigner is not being used.

3 Setting Up the 8051 Development Tools

This section explains how to get started with 8051 application software development using the FS2 debug pod, Keil uVision tools, and the IRMCF300 Series IC. Though this guide applies to a general hardware configuration, often the IRMCS3041 Reference Design Board is referenced as a specific case. The IRSamples program as a whole is designed to work with the IRMCS3041 Reference Design Board, if another platform is intended be sure to read about the modifications required in Section 4.3.

3.1 Software Setup

Keil uVision:

1. Start Keil uVision. Choose Project→Open Project and open the file IRSamples.Uv2, which is included with the sample code. In the “Project Workspace” window, click on the topmost folder, IRSamples.
2. Choose Project→Options for Target ‘IRSamples’. Select the Debug tab.
3. Click the radio button “Use:” and set the field to its right to *Fs2/Keil ISA-M8051EW Driver*, as shown in Figure 6 below. If this option does not appear in the list, the FS2 software has not been installed properly.
4. Click Setting and check that the Settings are: TckRate: 62500 and Tvcc Threshold: 2500. All the other settings under “Options for Target ‘IRSamples’” should be configured automatically. The Appendix to this application note lists all the options that should be configured.
5. Press “OK” in the Settings window and then “OK” in Options window.

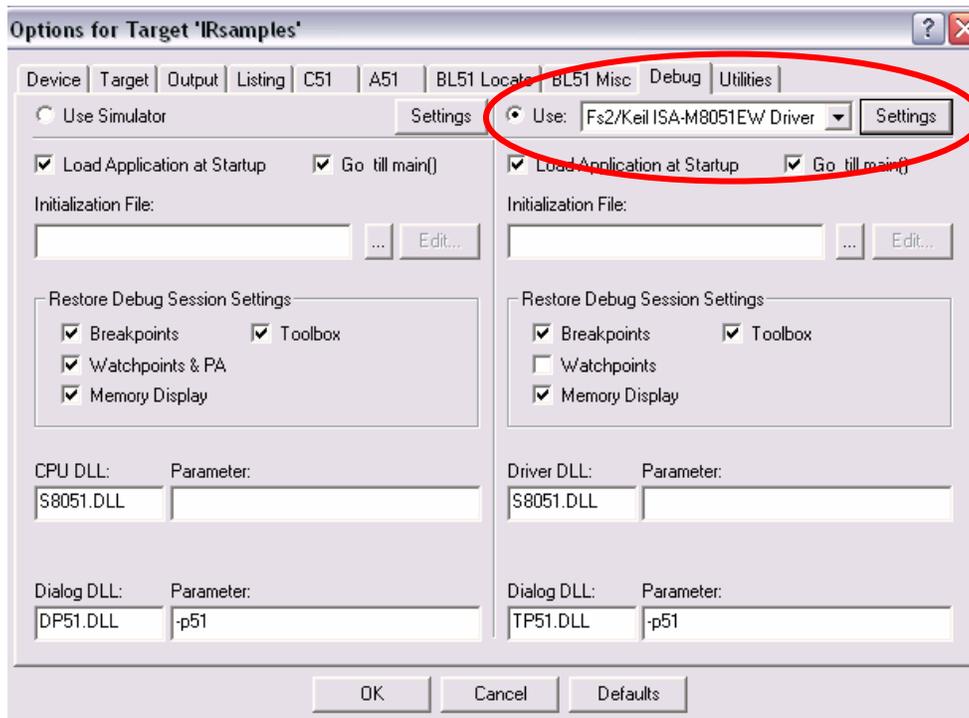


Figure 6—Debug Options window for a uVision project

HyperTerminal:

1. Open HyperTerminal (from Windows, choose Start→Programs→Accessories→Communications→HyperTerminal).
2. If the New Connection window does not automatically appear, select File→New Connection. Choose a name and icon for your connection and press “OK.”
3. Under “Connect Using,” select the appropriate COM port. This will likely be the same port that MCEDesigner uses to communicate with the control board. Press “OK” and the Properties window will open. Set the “Bits per Second” field to 57600, “Flow control” to None and verify the rest of the settings as shown below in Figure 7.

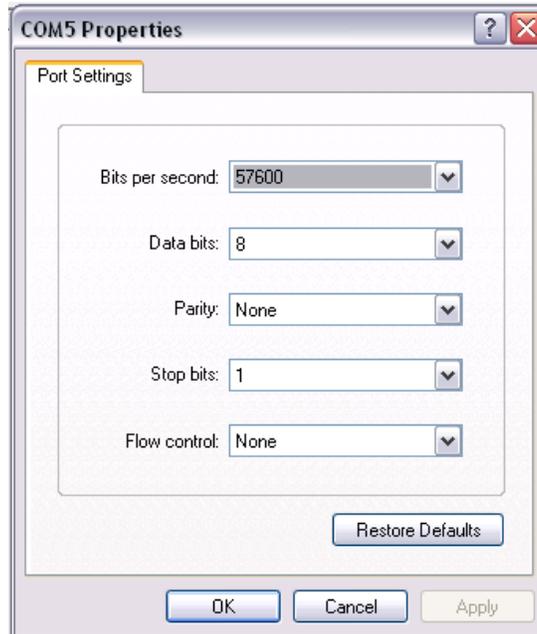


Figure 7—HyperTerminal connection Properties window

3.2 Hardware Setup

Connect the UART interface to the computer where you have installed Keil uVision and MCEDesigner. On the IRMCS3041, this is achieved using an RS-232 (serial) cable from the control board to the PC. Also, connect the FS2 debug pod to the control board. On the IRMCS3041, the FS2 Pod interfaces to the control IC through the connector J11. If the IRMCS3041 Reference Board is not used, follow the warning below.



Warning!

When connecting the FS2 debug pod to the circuit board, the FS2 hardware can be damaged by the high voltage on the board if appropriate isolation is not used. The problem arises because the DC bus minus (GND) is not at the same potential as earth (or wall) ground. For this reason, if proper isolation is not used, it is recommended that the board be powered by a DC power supply with isolated ground when using the FS2 hardware.

Hardware and Software Start-up

To properly start up the board and software, follow these steps:

1. Apply power to the controller board, and then turn on the FS2 pod.
2. Start Keil uVision. Choose Project→Open Project and open the file IRSamples.Uv2.
3. Choose Debug→Start/Stop Debug Session. The FS2 Console should come up briefly and a status bar in the lower left corner will display the progress in loading the 8051 code to RAM, shown in Figure 8 below.

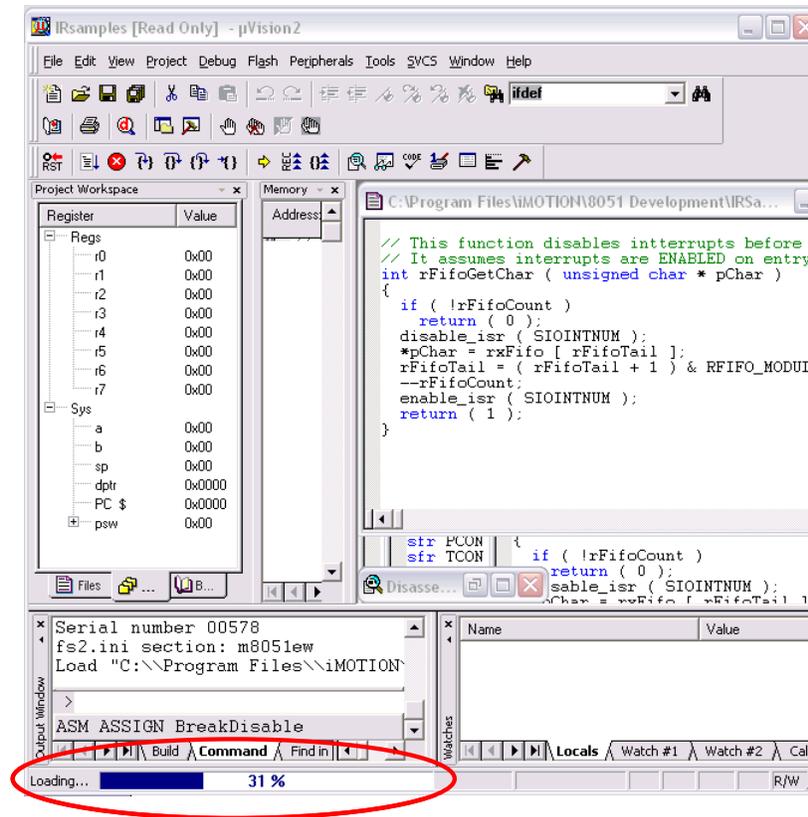


Figure 8—uVision window during 8051 program load.

4. Choose Debug→Go.
5. Start HyperTerminal and open the connection set up above. The motor can be controlled by commands sent through the HyperTerminal. (See section **Motor Functions** for commands.)
6. To exit from Keil after testing, choose Debug→Stop. Then Debug→ Start/Stop Debug Session. The program may be terminated at this time.

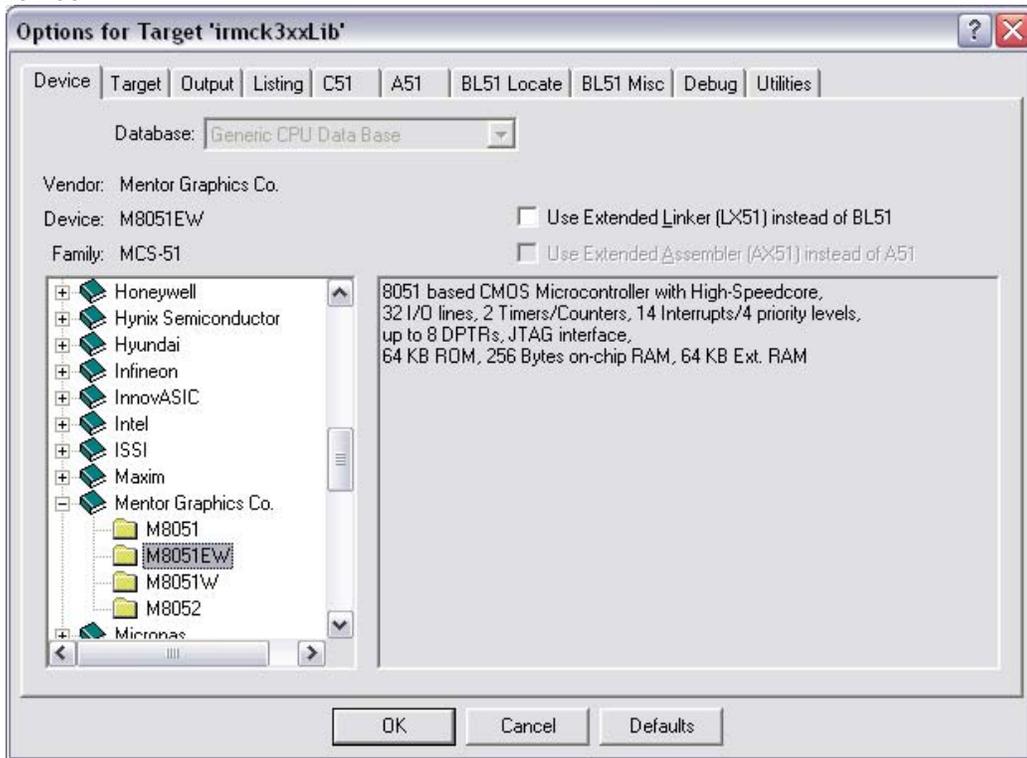
Making changes to the sample code

1. To make changes to the sample code, copy all the files of the sample code into a new directory.
2. Select all the files and right-click to select “Properties.” De-select “Read-only” and click “OK.”
3. Open the new project and modify files as desired. Save files by choosing Files→Save All.
4. To rebuild the machine code, choose Project→Rebuild all target files. The compiler will generate a new .hex file.
5. Follow the instructions in **Hardware and Software Start-up** to test the modified program in hardware. If the hardware is still powered on after testing a previous revision of your program, you can start at Step 3. It isn’t necessary to power the hardware off and back on before reuploading.

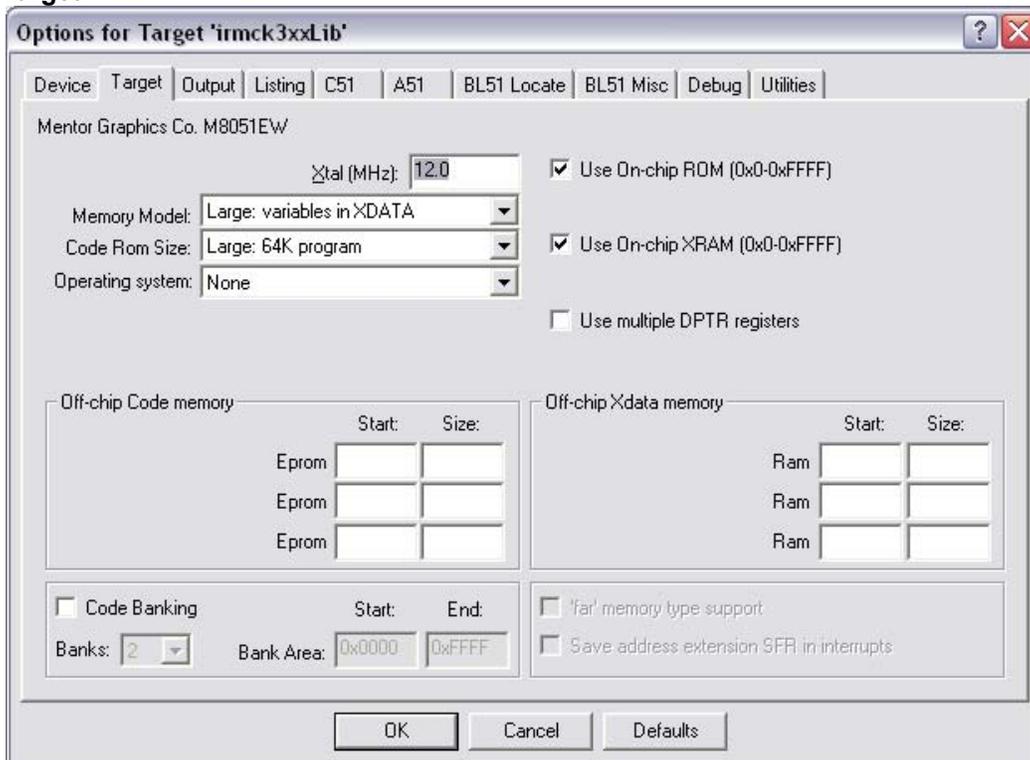
3.3 Keil uVision Project Options

These are the options that should be set under Project→Options for Target ‘IRSamples’. The uVision project file (IRsamples.Uv2) is shipped with all options set as shown below.

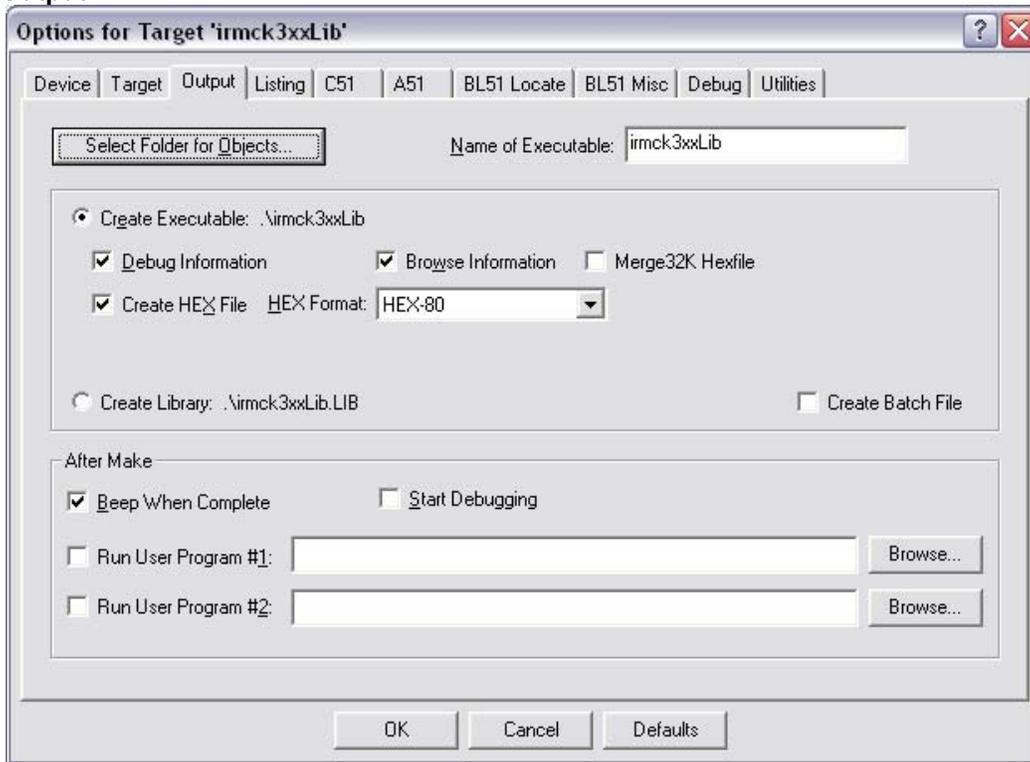
Device



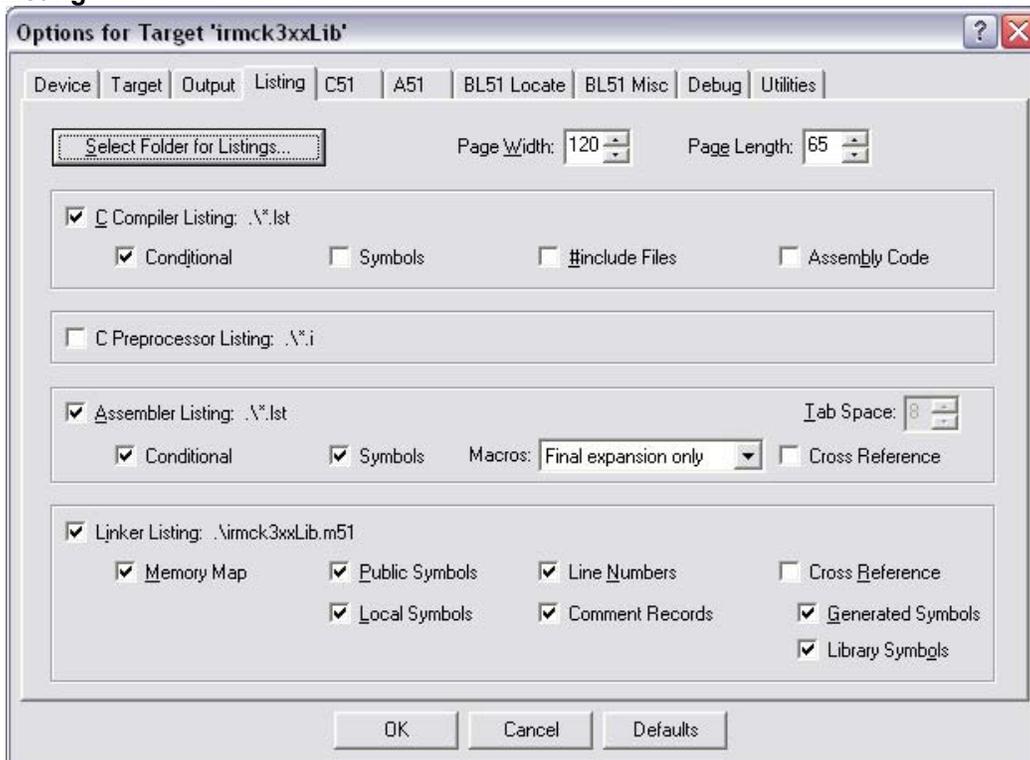
Target



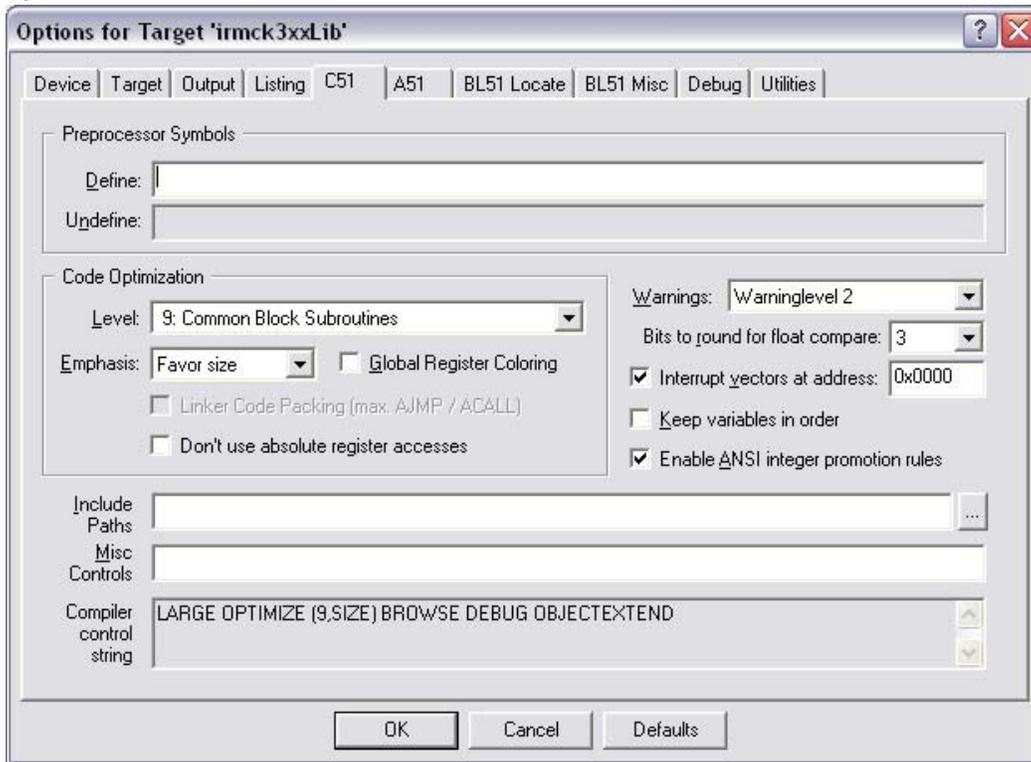
Output



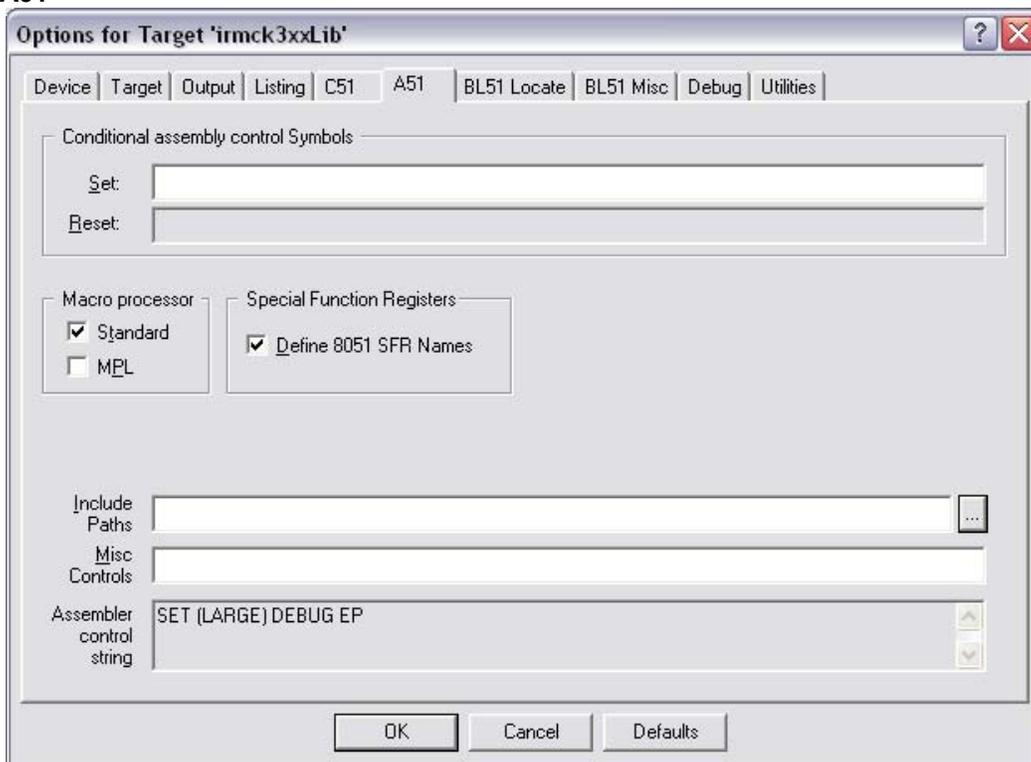
Listing



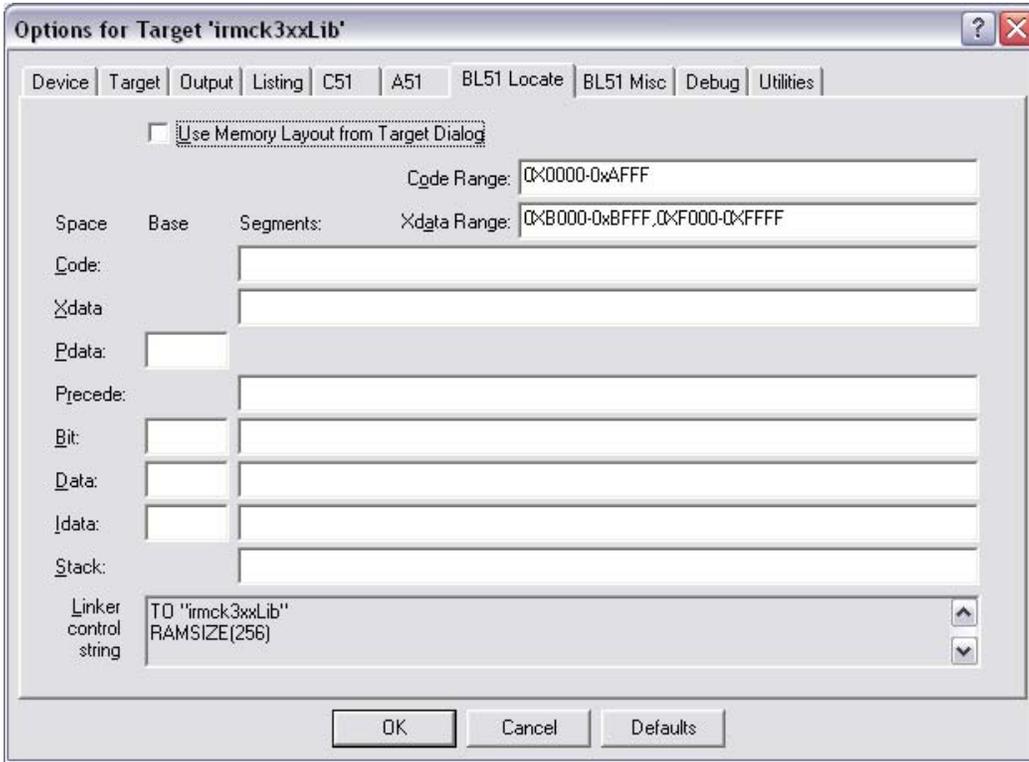
C51



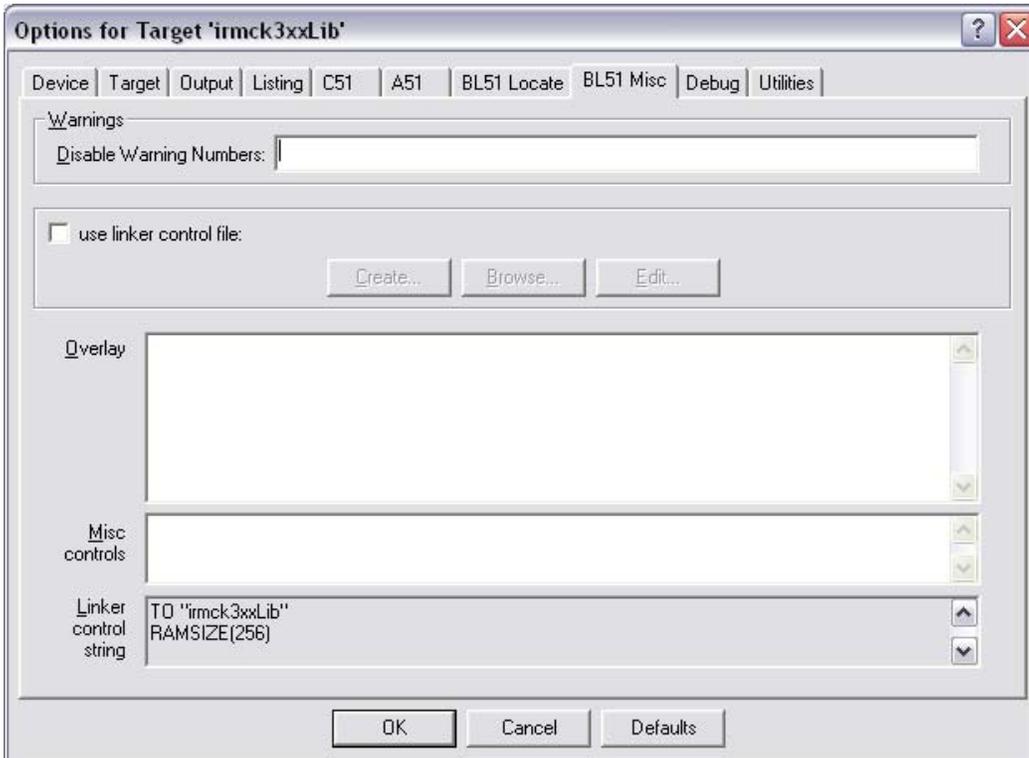
A51



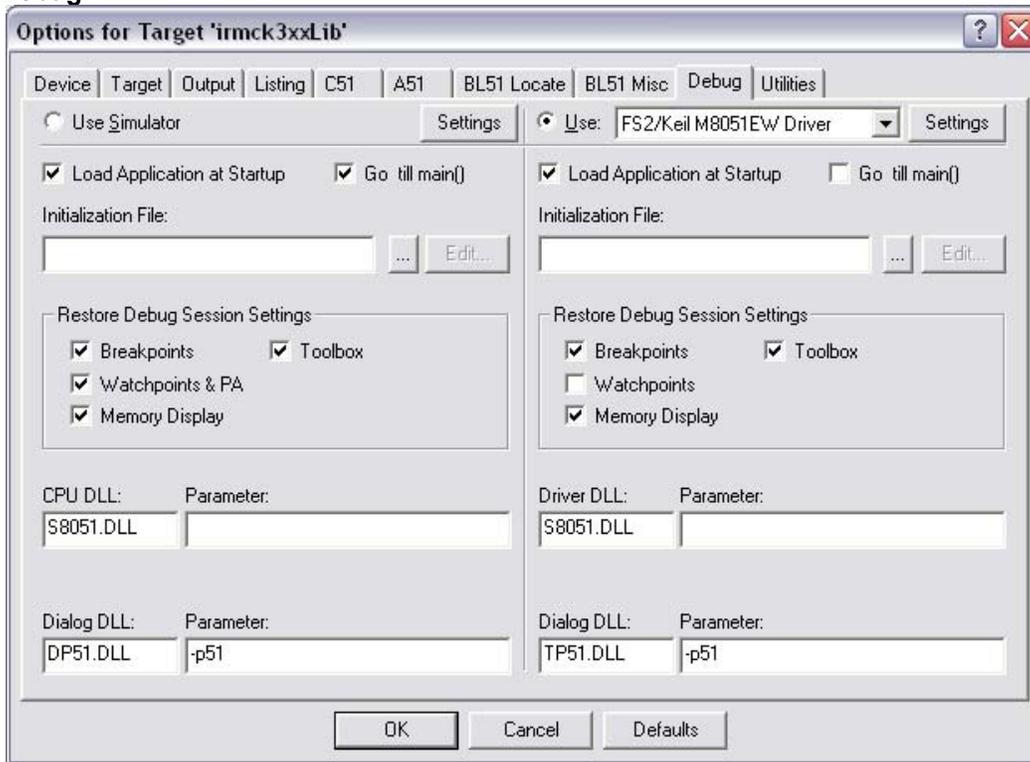
BL51 Locate



BL51 Misc

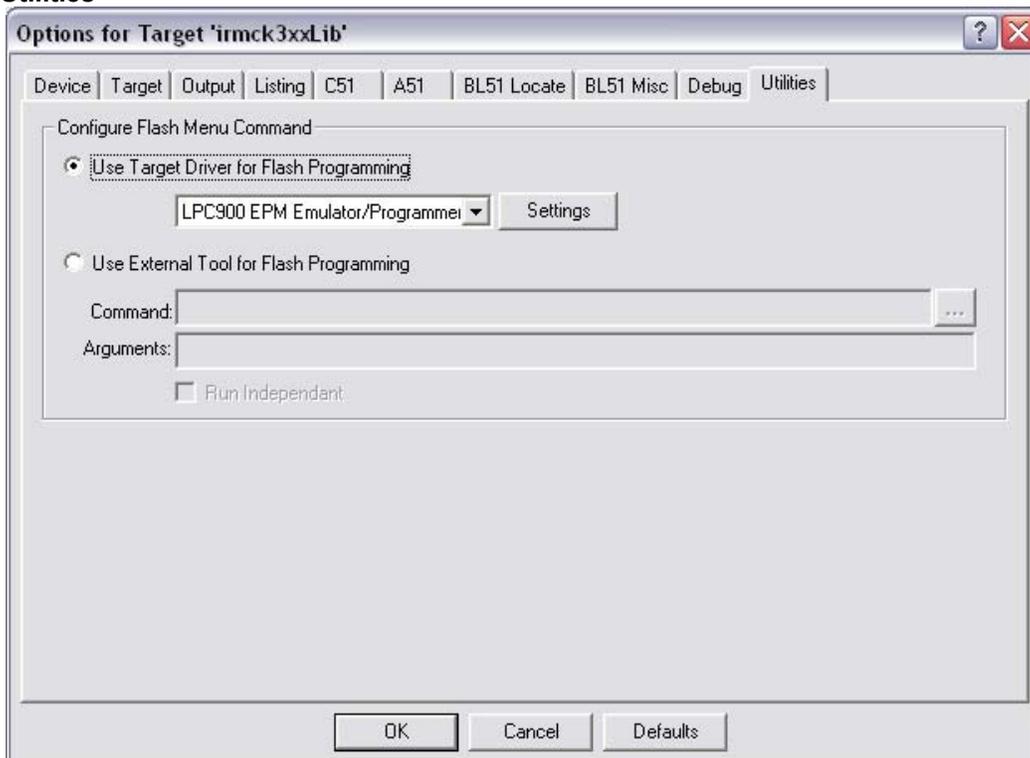


Debug



Note: The selection in the “Use:” box in the upper right of the Debug tab may not match this image’s text exactly. Be sure that it says “Fs2” and “Driver.” Also, be sure to uncheck “Go till main().”

Utilities



4 Sample Code

The sample code included with the Reference Design Kit, IRSamples, is intended to be a simple, easy-to-understand example. This program is not an efficient implementation with respect to function timing and optimization of processor usage during wait times. Also, IRSamples, as implemented, may not shut down the motor drive quickly enough in certain fault situations, depending on the application.

The 8051 code can be used to implement more complex motor control functions than those described in Section 4.2.3 below. For example, a washing machine “wash cycle” requires that the motor accelerate rapidly in one direction, stop, and accelerate rapidly in the other direction. This could be implemented in such a way that the number of rotations is dependant on the whether the soil setting is low, medium, or high. Other operations that can be implemented include and auto-rebalance or PFC sequencing.

Note that a command interface other than the UART may be used. For example, to receive commands from the digital I/O pins, modify *MotorCtrl* so that it monitors the state of the SFRs corresponding to the appropriate pins and then calls *MotorSeq* as needed.

Once the designer has set certain variables, such as the clock rate, these variables could be stored in a configuration area of the EEPROM.

4.1 Sample Code Structure

The main functions that an embedded 8051 application performs are to configure the MCE with the proper drive parameters and to start, stop, and regulate the speed of the motor. These are the same functions performed by MCEDesigner; the difference is that the “intelligence” is transferred from the host application (MCEDesigner) to the embedded 8051 application. Any MCEDesigner function (a pre-defined series of register operations), including timing delays, can be implemented directly in the 8051 application so that it can control the motor independently or with simple external commands.

The 8051 controls and monitors the MCE by reading and writing interface registers. The registers are described in the next section. The general steps that the embedded 8051 application must perform are:

1. Initialize the hardware—set clock frequency, initialize counters and timers, etc.
2. Start the MCE—verify that a valid MCE program has been loaded and initialize the MCE program counter
3. Configure the drive—write drive parameters to MCE registers for the desired motor
4. Start, stop, change direction, change speed—keep track of current state in order to correctly implement command (e.g. don't change direction if motor is running).
5. Monitor for faults—periodic interrupts handle external commands, reset the Watchdog timer, check for faults/errors and shut down drive if necessary.

4.1.1 Clock Frequency

The MCE code must set-up the phase-locked loop to generate the SYSCLK frequency by writing to the SFRs PLLF0 and PLLF1. The sample code already contains instructions to select between 32, 50, 64 and 128 MHz for the clock frequency. Simply uncomment the appropriate #define statement in *timer.h*.

Based on clock frequency the code should set the proper baud rate, timer initialization and reset values. These also are set up correctly when the clock frequency is selected in *timer.h*. However, some drive parameters (such as PwmPeriod, which sets the PWM frequency) are also configured based on the clock frequency. The user should regenerate the drive parameters from the “MCEWizard” using the new clock rate. See Section 4.2.1 below.

4.1.2 Registers

There are several types of registers, listed and described below.

1. Special Function Registers (SFRs)—SFRs can only be accessed by the 8051 microprocessor. Only a subset of the SFRs is described in this Guide. A complete list and description of the SFRs can be found in the Reference Manual. The SFRs can be used to:
 - Initialize and modify processor registers
 - Configure and read I/O ports
 - Set the clock frequency
 - Configure, initialize and reset timers
 - Configure the UART
 - Enable analog features such as op-amps
 - Enter and exit low-power modes
 - Configure and enable interrupts
 - Read faults and status
 - Configure and use I²C/SPI serial interface
 - Read and write the fixed MCE registers
 - Read and write the user-defined MCE registers

Since SFRs can only be accessed by the 8051 microprocessor, writing to these registers is relatively simple. They are defined in *irmcx3xx.h* using a special “sfr” keyword. Each SFR is assigned a name that corresponds to its memory address. In the sample code, the convention is that SFR names are in all capitals. They can be written and read using the name, as with any other variable. Below is an example of the SFR assignments that must be made to set the clock frequency to 50MHz.

```
PLLFO = 0x62; // set clock speed to 50 MHz
PLLFI = 0xC0;
PLLFI = 1; // switch to PLL clock
PLLFI = 0;
```

2. Fixed MCE Registers (FREGs)—The FREGs are accessible by both the MCE and the 8051 microprocessor. The MCE accesses the FREGs through the Motion Peripheral Blocks, and may modify a subset of them every PWM cycle. The 8051 microprocessor accesses the FREGs through a set of dedicated SFRs. Most FREGs only need to be configured once before running the motor, which can be done by the 8051 application. Additionally, the 8051 application will write to FREGs to start and stop the motor, or to adjust parameters due to varying operating conditions.

Each FREG is defined in *reglf.h* with an alias that begins with “FREG_” and corresponds to its memory offset. The functions *DoRegRd* and *DoRegWr* in the sample code are provided to read and write these registers. These functions determine the type of register, and then access the register through the dedicated SFRs. It is recommended that these functions be used without modification as the specific sequence of operations is critical for correct operation. Below is an example of writing the number “0” to Fixed MCE Register *pwmctrl_1*.

```
DoRegWr ( FREG_pwmctrl_1, 0 );
```

3. User-defined MCE Registers (RAM_REGS)—These registers are also accessible by both processors. However, in contrast to the FREGs, the RAM_REGS are not fixed in memory. The RAM_REGS are defined in the MCE Simulink design, and are assigned RAM addresses by the MCE Compiler. The compiler outputs a header file that should be incorporated into the 8051 code (detailed in Section 4.2.1). The name assigned to the register has the format:

```
<Simulink Sub-system>_<register name in Simulink>
```

Although RAM_REGS are directly addressable to the 8051 in the shared RAM, another dedicated set of SFRs are used to access them in a controlled fashion that prevents data corruption. (This is necessary because the MCE accesses the registers as a single 16-bit operation while the 8051 requires two 8-bit operations.) The functions *DoRegWr* and *DoRegRd* are provided in the sample code to read and write RAM_REGS through the dedicated SFRs. Below is an example of setting the DC Bus Over-voltage Level (of the Motor1 sub-system) to 172.

```
DoRegWr ( Motor1_DcBusOvLevel, 172 );
```

Note: In MCEDesigner, the FREGs can be distinguished from the RAM_REGS by looking at the “Type” column in the right side of the Motor1 window. To see this column, click on “Register Structure Definitions” on the left side of the Motor1 window. The “Type” column will show one of the labels listed below:

Fixed	Fixed MCE Register (FREG)
MCE	User-defined MCE Register (RAM_REG)
OBS	Obsolete Register
8051	Local 8051 register (created for MCEDesigner access to 8051 variables)

4.1.3 Files and Functions of Sample Code

The sample code is composed of several C source files, which divide the functions into groups according to their use. The .c files are:

1. *main.c* — Execution begins here. The *main* function calls each of the samples. The last sample function, *MotorCtrl*, does not return.
2. *regif.c* — This file contains functions to read and write 16-bit registers in shared RAM with guaranteed coherency using 8051 SFRs. See *RtlRegs.SRC* for the low-level implementation of the FREG interface and *Coherent.SRC* for an implementation of the RAM_REG interface. Examples of calls to the register interface functions can be found in *MotorCtrl.c*.
3. *EepromI2C.c* — This file contains sample code to read and write the EEPROM using the I2C interface.
4. *Timer.c* — This file contains a function that initializes timer 1 to generate interrupts at 2 msec intervals. A global variable "systicks" is incremented on each interrupt and the FREG_FaultFlags register is checked for a fault condition. The interrupt service routine also resets the Watchdog timer. The Watchdog timer must be reset periodically; otherwise the IC as a whole will reset. Timer setup varies based on the clock rate, which is set in *timer.h*.
5. *MceBoot.c* — This file contains functions to initialize the MCE using code that has been programmed to EEPROM by the MCEDesigner tool. It assumes that the automatic boot process has copied the MCE code from EEPROM to shared RAM and an "MCE Info" structure from EEPROM to a fixed location in 8051 program RAM. See Section 2.2 for the description and function of the MCE Info structure.

The function *StartMce* first copies the "MCE Info" structure from 8051 program RAM to a location in data RAM and verifies the validation field in the structure. If the validation field is incorrect, the entire structure is assumed to be invalid and the MCE is not initialized. Otherwise, the MCE Info structure provides the starting load address in RAM and the MCE execution address. The *StartMce* function uses this information to zero the MCE data area

preceding the start of the MCE program. The function *doMceBoot* is called to initialize the MCE special registers and begin MCE execution.

6. *asyncDriver.c* — This file contains functions to set up the UART and read and write data using FIFO (first-in-first-out) buffers. For IRMCx300 versions that support two UARTs, the code can be compiled for UART1 by commenting out line 20, which defines `USE_UART0`. The following functions are included in the file:

siolsr - This is the UART interrupt service routine, which handles transmit and receive interrupts. Received characters are placed in the receive FIFO. Characters to be transmitted are taken from the transmit FIFO.

siolnit - This function initializes the transmit and receive data structures and the SFRs that control the UART.

flushTx - Initializes the transmit FIFO.

flushRx - Initializes the receive FIFO.

setBaudRate - Initializes the baud rate SFR for 57,600 bps, based on the default clock rate of 64 MHz.

putChar_ - This function is called from a higher level (such as the **MotorCtrl** function) to transmit a character. If the transmitter is currently busy, it adds the character to the transmit FIFO. If no transmission is already in progress, it writes the character directly to the UART transmit buffer. The function returns 0 if the transmit FIFO is full (character cannot be accepted for transmission); or 1 if successful.

getChar_ - This function is called from a higher level to read a received character from the receive FIFO. It returns 0 if the receive FIFO is empty (no character available) or 1 if successful.

xFifoRoom - Called from *putChar_* to check the status of the transmit FIFO. Returns 0 if the transmit FIFO is full; 1 otherwise.

xFifoPutChar - Called from *putChar_* to add a character to the transmit FIFO. Returns 0 if the transmit FIFO is full; 1 if the character was successfully added to the FIFO.

xFifoGetChar - Called from *siolsr* to get the oldest character from the transmit FIFO. Returns 0 if the transmit FIFO is empty; 1 if a character is removed from the FIFO.

rFifoRoom - Called from *siolsr* to check the status of the receive FIFO. Returns 0 if the FIFO is full; 1 if the received character was successfully added to the FIFO.

rFifoPutChar - Called from *siolsr* to add a character to the receive FIFO. Returns 0 if the receive FIFO is full; 1 if the character was successfully added to the FIFO.

rFifoGetChar - Called from *getChar_* to get the oldest character from the receive FIFO. Returns 0 if the receive FIFO is empty; 1 if a character is removed from the FIFO.

IMPORTANT NOTE: The transmit and receive FIFOs are manipulated from both the interrupt level and the "task" (non-interrupt) level. For this reason, it is very important to ensure that UART interrupts are disabled while characters are added to and removed from the FIFOs at the task level.

7. *MotorCtrl.c* — This file contains a simple example of motor drive configuration and control. It reads character commands from the serial port using the functions provided by the UART driver. You can use a HyperTerminal (or equivalent) connection to send commands and read responses. A list of supported commands and their descriptions can be found in Section 4.2.3.

The function *MotorCtrl* checks that the MCE versions defined in *regif.c* and loaded from the EEPROM match, before allowing motor control operations. For more information, see section 4.2.2.

8. *RtlRegs.SRC* — Assembly-language functions to read and write RTL registers through the SFR interface. These functions are called by *DoRegWr* and *DoRegRd*.
9. *Coherent.SRC* — Assembly-language functions to read and write shared RAM registers using SFR registers for coherent data transfer. These functions are called by *DoRegWr* and *DoRegRd*.
10. *utils.c* — Utility functions to enable and disable a particular interrupt, identified by the interrupt number, as defined at the beginning of the file.

4.2 Running the Motor

4.2.1 Drive Configuration

After power-up, the motor will not run properly until the MCE has been configured with the correct parameter settings. These drive parameter values are generated using the “Parameter Configurator” (Excel spreadsheet). The second tab of the spreadsheet contains the correctly scaled values for the MCE registers. These values should replace the sample values defined in *parameters.h*. This process can be somewhat automated by exporting the sheet in text format and then adding “#define” at the start of each line.

4.2.2 MCE Header File

The MCE code is generated when the Simulink model file is compiled. The compiler also produces a header file (.h) which contains 1) definitions of user-defined MCE registers, 2) register map structures for addressing of the user-defined MCE registers, and 3) product and version identification. This code should replace the samples at the top of *reglf.c* and *reglf.h*. In *reglf.h*, replace the section titled “COMPILER GENERATED DEFINITIONS” with the corresponding section in the MCE Compiler header file. Specifically, replace the following sections in *reglf.h*:

```

/* Product ID, Design ID and Version strings */
char ir_productID = 61;
char ir_designID [] = "IRMCS3041_Release_2_0";
char ir_vers [] = "1.301";

RegMapType RegMap [] = {
    { 2, 6, 0, 16 }, /* 0 */
    { 2, 4, 0, 16 }, /* 1 */
    ...
    ...
    ...
    { 3, 2, 0, 16 }, /* 25 */
    { 3, 0, 0, 16 }, /* 26 */
};

unsigned short PageBase [] = {
    0xE000,

```

```

...
...
...
0x0000,
};

```

Similarly, in *reglf.c*, replace the section, reproduced below, titled “COMPILER GENERATED INITIALIZATIONS” with the corresponding code in the header file.

```

/* Register map array indexes */
#define Motor1_CriticalOV_Fault    0
#define Motor1_LV_Fault           1
...
...
...
#define Motor1_TargetSpeed        28
#define Motor1_VhzEnable          29

/* Definitions for Page field in RegMap and indexes into PageBase */
#define PAGE0_RD                   0
...
...
...
#define PAGE3_WR                   7

```

Note that the MCE design ID and version number of the header file must match that of the MCE code loaded from the EEPROM for correct operation. If the Simulink model is changed, then a new header file must be created during compilation and added to the code as described in the paragraph above.

The sample code is configured to correctly write to the RAM_REGS of the IRMCS3041 Reference Design Kit and has the proper drive parameters to run the Golden Age GK6040-6AC31 motor.

4.2.3 Motor Functions

The sample code treats the motor as a state machine, with three states: DRIVE_IDLE, DRIVE_RUN and DRIVE_FAULT. The function *MotorCtrl* takes input commands from the serial port and passes valid ones to *MotorSeq*. Based on the current motor state, *MotorSeq* calls appropriate functions to implement the command or returns an error indicating that the command was invalid. If an invalid command is entered, 'Invalid Command' is returned to the HyperTerminal display. Listed below are the commands supported from the function *MotorCtrl*, with explanations of their operation.

C or c

Configure motor drive and clear faults. 'Configured' will be echoed back on the UART if successful. If the motor is running, the command is ignored and 'Invalid Command' is returned instead. See section 4.2.1 above.

+

Set forward direction. 'Forward' is echoed when the operation is complete. If the motor is running or in a fault condition, the command is ignored and 'Invalid Command' is sent instead.

-

Set reverse direction. 'Reverse' is echoed when the operation is complete. If the motor is running or in a fault condition, the command is ignored and 'Invalid Command' is sent instead.

F or f

Clear fault condition. 'Fault Clear' is echoed when the operation is complete. If the drive is not in a fault condition, the command is ignored and 'Invalid Command' is sent instead.

G or g

Run motor. The motor is placed in run state and turns in the configured direction at a low speed. 'Started' is echoed when the operation is complete. If the motor is already running or in a fault condition, the command is ignored and 'Invalid Command' is sent instead.

S or s

Stop motor. The motor is stopped and 'Stopped' is echoed when the operation is complete. If the motor is already stopped or in a fault condition, the command is ignored and 'Invalid Command' is sent instead.

R or r

Set motor speed. This is a multi-character command. The command character must be followed by exactly four decimal digits (0 - 9) defining the target speed in rotor RPM. If the motor is not running the command is ignored and 'Invalid Command' is echoed. If the requested speed is out of range for the motor (according to the value of "#define Mtr_Max_Speed") then the Mtr_Max_Speed value will be used. Otherwise, the operation is performed after all four digits have been received, at which point 'Speed Set' is echoed. If a character other than a digit is received, an 'X' is echoed and the command is aborted.

?

(1) Get motor speed. This command returns the motor speed when the drive is running. The current speed is output in motor RPM. This RPM calculation relies on the parameters generated by the parameter configurator.

(2) Read FaultFlags. When the drive is in a fault state, this returns the value of the FaultFlags register. The register value is displayed in hexadecimal format.

H or h

Catch-Spin Start. This begins the catch-spin startup sequence for the motor. The system will monitor the speed and direction of the motor to determine if the motor should be stopped and reversed, or if the motor is already going in the correct direction and catch it. This startup mode is suitable for an instance where the motor may already be in motion due to outside forces (such as wind blowing a fan). This command is allowable only in the idle state, otherwise 'Invalid Command' is echoed. At the end of the sequence, 'CatchSpin Complete' is echoed.

T or t

Ramp Stop. This function will slowly ramp the motor down to zero speed. This is opposed to simply stopping the motor by halting the PWM. Upon successful stopping of the motor 'Ramp Stop Complete' will be echoed. The rate is determined by the *rampTime* variable, which is the time in seconds to ramp to zero.

Z or z

Zero Vector Brake. This function will turn on the zero vector brake command for 20 seconds, then halt the PWM and turn off zero vector brake. In the case of a fault, the function will break out of the 20 second wait time and halt the PWM.

> Write to Register.

< Read from Register.

When one of these commands is given, the controller will prompt the user for register number, which is a three digit number which correspond to the register's address. If it is an RTL register, then the register number = (Address found in the Reference Manual + 256). For an MCE register, the register number can be found in the .h or .map file output from the MCECompiler. Alternatively, the register numbers for both types of registers can be found in *regif.h*. If the write

command is given, then the controller will prompt for the value to be written, which is a five digit number. This command is valid regardless of the sequencer state.

4.3 Extending Functionality

IRSamples only allows control of a single motor, without PFC, and is configured to work only with development kit release MCE program. It is likely that new register names are defined in a new MCE implementation. If this is the case then modifications to the base IRSamples is required.

```
<Simulink Sub-model>_<register name in Simulink>
```

Most likely the Sub-model and possibly register names will change between MCE designs. These names are hard-coded into the uVision project and will need updating. The Fixed-Registers are also hard-coded into the uVision project, but they are constant across the 300 series product family.

Before making changes to the IRSamples code for a new MCE, import the new motor parameters and the new header (*.h) file from the MCE Web Compiler.

4.3.1 Modifications Required For New MCE Register Names

These MCE register names, such as 'Motor1_MotorSpeed' may be modified to 'Compressor_MotorSpeed' in the .mdl MCE design file. If such a modification occurs, it is necessary to not only update the *regif.h* and *regif.c* files as outlined previously, but also to modify all references to the old MCE register name in IRSamples. Most of these modifications will be located in *MotorCtrl.c's* *MotorSeq* function or *Timer.c's* *Timer1* function . An example of this would be...

```
DoRegWr ( Motor1_TargetDir, 0 );
```

Which would need to be changed, in the case of IRMCx312, to...

```
DoRegWr ( Compressor_TargetDir, 0 );
```

After properly converting the old names to the new MCE register names, the system should successfully compile. If compilation errors arise, be sure to verify that some old register names were not missed and left in the system. This step can be quickly and accurately performed with the Find-Replace function in uVision.

4.3.2 Considerations for IRMCx341/371

The 341/371 products do not have any special considerations at this time.

4.3.3 Considerations for IRMCx343

1. Although IRMCx343 has PFC capability, IRSamples does not include any PFC sequencing. If PFC functionality is to be added it is necessary to enable two additional op-amps; PFC current, and VAC. This is accomplished by setting the 'HWCFG' SFR to '0xC1'. It is best to set the HWCFG register in the *main.c* file, before calling the *MotorCtrl* function.

```
'HWCFG = 0xD5';
```

2. The STOPS SFR should be set to choose the source of the PFC GATEKILL.
3. If PFC functionality is added, the *PwmSyncEnb* register must be enabled (1) so that proper PFC to Motor 1 synchronization is maintained.

4. The *FREG_syscfg* register must be set to '5' if PFC sequencing is to be implemented.
5. Fault Masking in the *Timer.c's Timer1* function must be modified to not mask PFC GateKill if PFC sequencing is to be implemented.

4.3.4 Considerations for IRMCx311/312

1. Although IRMCx311 has PFC capability, IRSamples does not include any PFC sequencing. If PFC functionality is to be added it is necessary to enable three additional op-amps; PFC current, VAC, and Motor 2 current. This is accomplished by setting the 'HWCFG' SFR to '0xD7'. It is best to set the HWCFG register in the *main.c* file, before calling the *MotorCtrl* function.

```
'HWCFG = 0xD7';
```

2. The STOPS SFR should be set to choose the source of the PFC GATEKILL.
3. If PFC functionality is added, the *PwmSyncEnb* register must be enabled (1) so that proper PFC to Motor 1 synchronization is maintained.
4. Fault masking in the *Timer.c's Timer1* function must be modified to not mask PFC GateKill if PFC is implemented. Fault masking will also need to be modified if Motor 2 will be used instead of Motor 1.
5. The FREG_syscfg register must be set to '5' if PFC sequencing is implemented or should be set to '3' if Motor 2 is used. FREG_syscfg should be set to '1' if both PFC and Motor 2 are used.
6. With the IRMCx311/312 device it is now possible to use Motor 1 instead of Motor 2 as IRSamples does. To make this transition nearly all of the MCE defined registers must be modified to their Motor 1 (referred to as 'fan' in dev. kits) counterparts. This will also require that any motor specific FREG registers (FREG_Rotation_2) be changed to their Motor 1 version (FREG_Rotation_1). Most of these registers will be located in the *MotorCtrl.c* or *Timer.c* file and will always end with a _1 or _2 if they are motor specific.
7. If changing to Motor 2 the following registers will need to be commented out in the 'configure' function. These registers are not used in the Motor 2 configuration, but are in Motor 1.
 - ModLim
 - VqLimFilBw
 - FwkSpd
 - VqLim (MCE Register)

4.4 Troubleshooting

When a debug Session is started, the message "****error122: AGDI: memory read failed" appears in the Output Window.

- Check that the FS2 debug pod is connected to the parallel port and is turned on.
- Under Project→Options for Target 'IRSamples', click on the Debug tab and then the Settings button and verify that the Comm Port setting is Lpt1 (or the correct one for your PC.)

No characters echoed in HyperTerminal:

- Check that the FS2 debug pod is properly connected to hardware and turned on.
- Under Project→Options for Target 'IRSamples', click on the Debug tab and verify that "Use:" is set to *Fs2/Keil ISA-M8051EW Driver*. Check that the Settings are: TckRate: 62500 and Tvcc Threshold : 2500.

- Check that the computer running HyperTerminal is connected (by serial cable) to the hardware. Also, verify that HyperTerminal is using the correct port with the correct communication options as described earlier in this document (**Software Setup**). The COM port should be the same one that MCEDesigner uses.

LED does not change from red to blinking green after the drive is configured (c or C in HyperTerminal) (for IRMCS3041)

- Check that DC bus voltage is within range and not causing a fault.

LED changes from red to blinking green (for IRMCS3041) after the drive is configured, but the motor does not turn when commanded.

- Check that the correct drive parameters are entered into MotorControl.h
- Check that the MCE is properly started—Set a break point in *MCEBoot.c* to see whether **doMceBoot** is called. If not, the MceInfo structure may be incorrect, or the memory address, RomMceInfo may be incorrect.

5 Programming the Control IC

5.1 Programming the EEPROM with MCEDesigner

5.1.1 Downloading 8051 Code to EEPROM with MCEDesigner

Once the 8051 application code has been fully tested using the FS2 Pod and Keil uVision, the code may be downloaded to the EEPROM for stand-alone testing. To do so, follow the steps below:

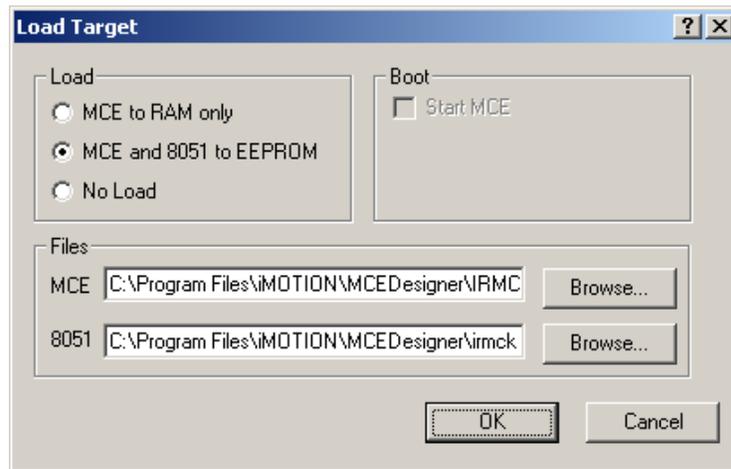


Figure 9—Load Target in MCEDesigner

1. Power down the controller board, then the FS2 debug Pod. Disconnect the FS2 Pod from the controller board.
2. Power up the board and start MCEDesigner. Open an .irc file for your controller board.
3. Click on the System window and then select Tools→Load Target. The Load Target window is shown in Figure 5 above.
4. Select “MCE and 8051 to EEPROM.” Choose the appropriate MCE program and, for the 8051 program, the .hex file created by uVision.
5. When the download is complete (about two minutes), power down the controller board and wait for the COM to go Down.
6. Power up the board, and the COM should stay Down. Close MCEDesigner to release the COM port.

7. Start HyperTerminal (or other UART communication program) and verify that motor control functions operate correctly.

5.1.2 Restoring the Original 8051 Code

After stand-alone testing, the user may want to restore the original 8051 code (MCEDesigner Agent) to the EEPROM so that application development can be continued with MCEDesigner. To do so, follow the steps below:

1. Power down all equipment. Connect the computer and FS2 Pod to the controller board.
2. Power up the FS2 Pod, then the controller board.
3. In uVision, open the project corresponding to the MCEDesigner 8051 code (e.g. IRMCx341Lib.Uv2). Choose Debug→Start/Stop Debug Session. Wait for the program to load. Then select Debug→Go.
4. Open MCEDesigner and COM should come Up (green).
5. Click on the System window and then select Tools→Load Target.
6. Select “MCE and 8051 to EEPROM.” Choose the appropriate MCE program and, for the 8051 program, the .hex file corresponding to the original code (e.g., IRMCx341Lib.hex).
7. When the download is complete (about two minutes), power down the controller board and wait for the COM to go Inactive. Stop the debugger and end the debug session. Turn off and disconnect the FS2 Pod.
8. Power up the board, and COM should come Up once again. The system is now ready to take commands from MCEDesigner.

5.2 Using MCEProgrammer

MCEProgrammer is an IR supplied utility to generate and program EEPROM images or OTP images. This section will only describe EEPROM image generation and programming with MCEProgrammer. Using the utility for OTP image generation and programming is covered in Section 6.

5.2.1 Generating an EEPROM Image

MCEProgrammer is a command-line script which takes an MCE '.bin' and 8051 '.hex' file and then creates an EEPROM image. There are multiple output formats shown below.

1. FS2 TCL Script. This is the default output format, and can be input to the FS2 Console application to program the EEPROM directly through the JTAG interface.
2. Binary file. This is a raw data file containing a binary image of the data to be programmed to EEPROM and can be used with an external device such as a gang programmer.
3. Initialized C array. This is a '.c' file which can be added to an 8051 uVision project to program the EEPROM.

The MCE Info structure can also be automatically included if MCE Designer or IRSamples are being programmed to EEPROM. There are three possible configuration options for the MCE Info structure.

1. MCE Info for MCEDesigner. MCEDesigner expects to find the MCE Info structure in the 8051 program RAM at address 0x5FA0 for the IRMCF34x and IRMCF371 devices and at address 0xBFA0 for the IRMCF31x devices. If you are programming the MCEDesigner Agent software to EEPROM, you must include the MCE Info structure at the proper address or the Agent will not initialize the MCE processor.
2. MCE Info for IRSamples. IRSamples expects to find the MCE Info structure at address 0x5FA0. By defining “LARGE_MEMORY” in the sample code, you can change the expected address to 0xBFA0. If you are programming IRSamples to EEPROM, you must include the MCE Info structure at the proper address or the MCE processor will not be

initialized. You can, of course, modify the sample code so that the MCE Info structure has a different address or is not used at all.

3. If you are programming custom software to EEPROM, use of the MCE Info structure is optional and can be excluded.

5.2.2 Programming EEPROM with the FS2

The FS2 TCL script option is used as an input file to the FS2 debugger. By executing this script in the FS2 debugger environment the EEPROM can be directly programmed by using the SFR registers. To use this programming method, please follow the instructions below.

1. Start the FS2 System Navigator. This should bring up the following window.

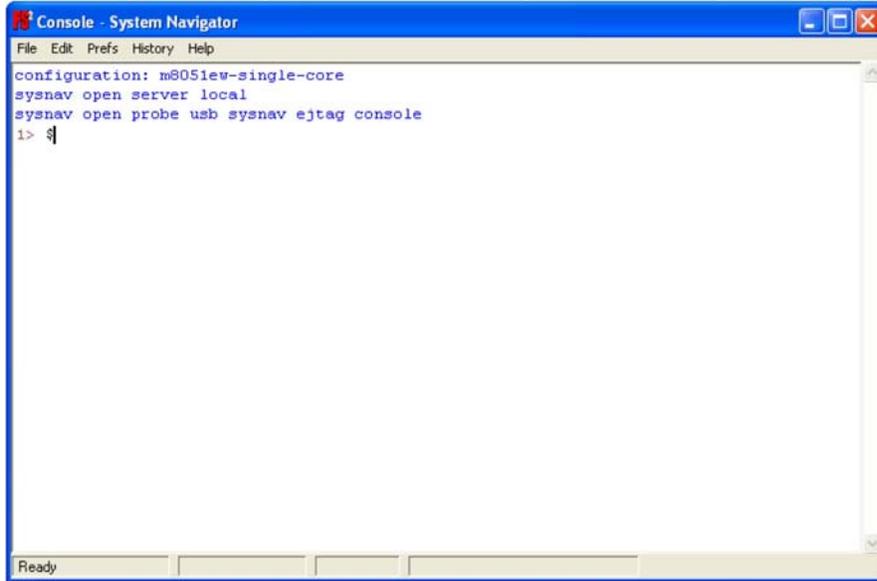


Figure 10—FS2 Debugger System Navigator Window

2. Next go to the 'File Menu' and select 'Source...'

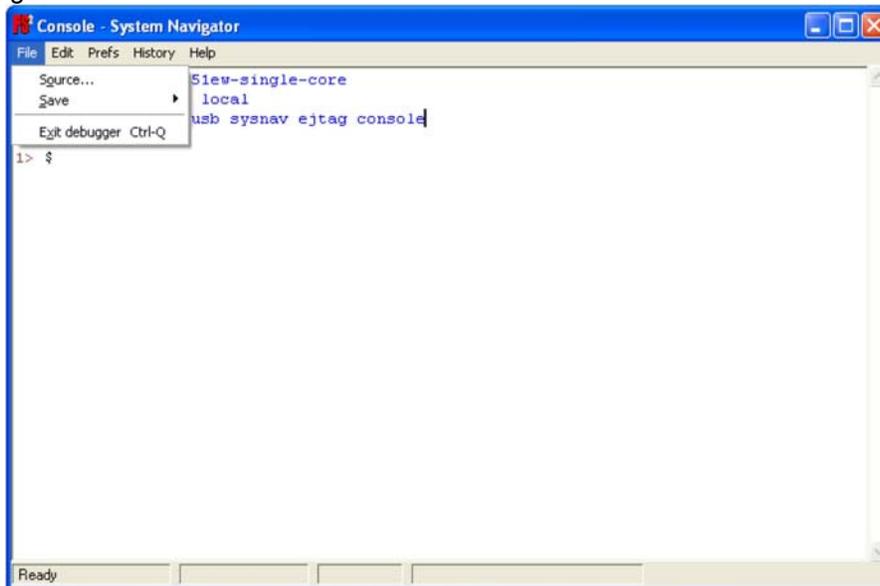


Figure 11—FS2 Debugger File Menu

3. Select the FS2 TCL output file with extension '.tcl' and click 'OK'. The script will execute immediately.

When the script begins execution, it displays the message:

Begin EEPROM Programming.

in the FS2 Console window. The script displays a period (".") for every 256 bytes that are programmed to EEPROM. This process is slower than other EEPROM programming methods because of the overhead of JTAG communication and SFR access, so please be patient. When programming is complete the script displays the message:

EEPROM Programming complete. Verifying.

and then displays a period for every 256 bytes that are read from EEPROM for comparison to the programmed data. If an error is detected the script displays the message:

```
xxx != yyy at address zzz
```

where xxx is the value read from EEPROM, yyy is the expected value and zzz is the EEPROM address at which the error was detected. The script terminates after detecting a single error.

If no errors are detected, the script displays the message:

```
EEPROM verified; programming successful.
```

If there is a problem with the programming, try decreasing the Tck rate (using the FS2 Connect Dialog) from 500k to 100k or 50k.

6 Migrating from the F-version to the K-version

6.1 System Differences

6.1.1 Program and Data Memory Space

The OTP (One-Time Programmable) memory in the IRMCK3xx contains 64Kbytes of memory space that is split between the 8051 microprocessor and the MCE. 56Kbytes of that memory is reserved for 8051 program space, while the remaining allocation is used to load the 8K of RAM available in the system. This 8K is split into a 0.5Kbyte zone for the MCE Data (Dual Port Shared) RAM, 5.5Kbyte zone for MCE Program RAM, and 2Kbytes zone for 8051 local data RAM (Figure 12).

In the F-version of the IC, MCEDesigner Agent stores trace data in the 8051 program space temporarily before it is transferred to the PC running MCEDesigner. The K-version can be run with MCEDesigner, but the trace function is limited to 512 points instead of 1024, as the trace data must be stored in the 8051 Local Data section of the RAM.

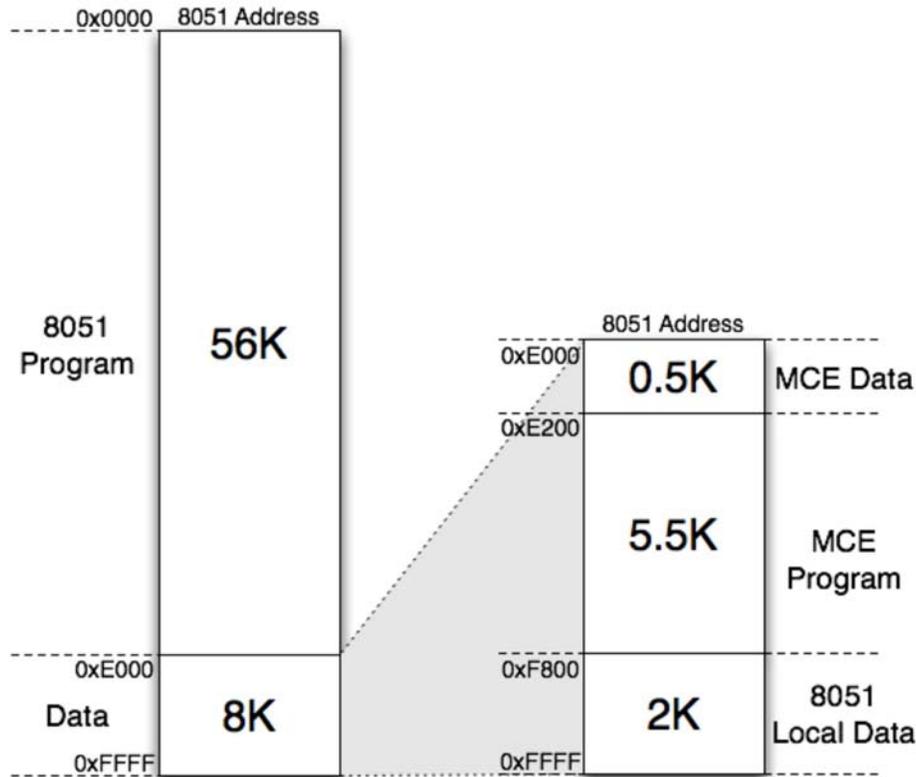


Figure 12—Memory Map of OTP

6.1.2 Bootload Sequence

When the controller is turned on, without an 8051 debugger connected, it will load the last 8Kbytes of OTP memory into the 8Kbyte RAM. This will effectively load the MCE program along with any shared initial data. If a debugger is connected the bootload process will not occur to prevent third parties from having access to the MCE program. The 8051 program space does not get transferred to RAM, as the OTP memory is directly read by the 8051 microprocessor.

It is important to understand that the MCE Data and 8051 Local Data are also transferred during the boot load sequence. This includes the MCE Control registers, which are mapped to the MCE Data area at addresses 0xE000 – 0xE01F. An error can occur if the MCE Control registers are initialized incorrectly during bootloading. The MCE can begin executing while bootload is still occurring, resulting in corruption of the MCE program. To prevent this error, the MCE Control register section of the OTP should be loaded with 0x00. MCE Programmer automatically fills the correct values into these memory locations. For more information about the MCE Control registers, see Section 5.2 of the Reference Manual.

6.1.3 8051 and MCE Internal Clocks

Because the 8051 will execute its program directly from the OTP memory it must be constrained to the maximum operating frequency of the OTP. In the IRMCK3xx the maximum OTP and, therefore, the maximum 8051 frequency is 33Mhz. Because the MCE program is accessed via RAM it is possible to run the MCE significantly faster so long as the 8051's clock is divided.

The IRMCK3xx supports running the MCE clock at multiples of 1, 2, 3, and 4 times the 8051 clock. This means the maximum frequency of 128Mhz could be maintained for the MCE, while the 8051 could run at 32Mhz. This would allow the OTP to operate below its maximum 33Mhz limitation, while having the maximum amount of computational power available in the MCE. This

clock division is accomplished through two bits located at PLLF2[6:5]. These two bits will cause the clock divider to work in the following manner (Table 5). It is important to note that all of the UARTS, timers and all other clocked components except the MCE are on the slower clock.

PLLF2[6:5]	MCE Clock Frequency	8051 Clock Frequency
0b00	MstrClk	MstrClk/1
0b01	MstrClk	MstrClk/2
0b10	MstrClk	MstrClk/3
0b11	MstrClk	MstrClk/4

Table 5—PPLF2 Register Options For Clock Dividing

The designer should ensure that timers and counters are correctly initialized if the 8051 frequency changes when moving from the F to the K-version. A specific problem to watch for, which can arise due to the different clock frequencies between the 8051 and MCE, is the servicing of the SYNC interrupt. If the 8051 clock is too slow, then it may not be able to complete the interrupt service routine (ISR) quickly enough. This problem can be solved by shortening the ISR or reducing one or more pwm frequencies in the MCE.

Included in the iMotion software, in a directory called “*MCEDesigner Agent_K*”, are .hex files for each part specifically to allow the designer to use the K-version with MCEDesigner. Note that in these files, the MCE Clock is set to 128MHz and the 8051 Clock is 128MHz/4 = 32MHz.

6.1.4 UART Baud Rate

The UART in the F version had a single 8bit SFR to set the baud rate. The K version has added an additional upper 8bit SFR so that a 16bit value is used for the baud rate. This allows a minimum available baud rate of 36BPS. The new registers are U0BRH (SFR 0x93) and U1BRH (SFR 0x9b).

6.1.5 8051 and MCE Memory Protection

To prevent third parties from connecting to the IRMCK3xx and downloading proprietary motor control algorithms, it is not possible (without specifically enabling) to read the OTP contents from external sources such as a debugger. When reading the OTP from a debugger, the memory will be scrambled in a destructive manner. There is no way to recover the scrambled data that can be read from a debugger.

This scrambling is, by default, always enabled and must be explicitly disabled during OTP programming. The very last byte of OTP (address 0xFFFF) determines if scrambling is enabled. If this byte is all 1s (0xFF) then scrambling is disabled. If this byte is any other value then scrambling is enabled. An additional step is required before JTAG can properly read the OTP (if the last byte is 0xFF); the JTAG interface must read this last byte. Upon reading this byte the controller will determine if the OTP will be allowed unscrambled read access. Scrambling is not disabled until 0xFF is read from the 0xFFFF address.

6.1.6 8051 Debugging

This feature of the IC should be kept in mind when attempting to debug the 8051. The debugger (Keil) must first read the very last byte of memory so that scrambling is disabled. It is also possible to have the 8051's program read this last byte of memory so that memory protection is disabled automatically.

When debugging, it is still possible to enter a breakpoint into the 8051 program even though the memory is OTP. A single breakpoint can be entered by way of the FS2 debugger that will be trapped when set. This breakpoint must then be cleared so that a new breakpoint can be defined.

6.2 Checking for MOVX Instruction Sequences

6.2.1 Problem Description

The IRMCK3xx IC can have a memory read or write error when the 8051 reads or writes to the IC RAM immediately after a previous instruction has read or written to RAM. The error can occur from any sequential combination of external read or write instructions, called MOVX (with hex codes 0xE0, 0xE2, 0xE3, 0xF0, 0xF2 or 0xF3). The result is that the second read or write is not correctly executed so that either an incorrect value is read from memory or the write is not executed at all.

The root cause of this problem is the logic used to synchronize the MCE and 8051 clocks when the 8051 accesses the shared RAM. This synchronization is necessary due to the separate clock domains of the MCE and 8051. During clock synchronization the 8051 execution is stalled, but the stall does not occur correctly if there are sequential MOVX commands.

This occurs in the K-version of the IC only. A more detailed description of this problem can be found in the "IRMCK300 Errata Sheet."

The output assembly file should be checked for the presence of consecutive MOVX instructions after the code is built; IR provides a tool to automate this process, described in the next section. If a MOVX issue is identified then the following remedial actions can eliminate consecutive MOVX instructions:

- a) Change the optimization level.
- b) Modify the C source code.
- c) Modify the assembly code to eliminate consecutive MOVX instructions.

The output assembly file must be checked after modification to confirm that the MOVX problem has been eliminated.

6.2.2 MOVX Check Utility Description

The MOVX instruction sequence checker utility is a Perl script built into executable form (so Perl does not need to be installed to run it). The utility can be easily installed into the Keil uVision IDE to run it from the uVision Tools menu. Messages from the utility are displayed in the uVision Build window.

The utility, called movxcheck, searches the 8051 object code stored in the Intel hex file which is output from uVision when the project is built. It finds any potential instruction sequences which could trigger the memory access problem, described above, when the 8051 program contains two contiguous (back-to-back) MOVX instructions.

The 8051 instruction set defines opcodes with values between 00h and FFh. All values except A5h represent valid instructions. Instructions can be 1, 2 or 3 bytes in length. That is, some instructions have only an opcode, some have an opcode followed by an 8-bit operand, and some have an opcode followed by a 16-bit operand. The movxcheck utility looks for any sequence of two contiguous MOVX instructions, represented by the opcodes: 0xE0, 0xE2, 0xE3, 0xF0, 0xF2 and 0xF3. All of these are 1-byte instructions. The utility looks for pairs of these six values in any combination or order. (For example, F0-F0, E2-F0, or F0-E2.)

Since the program contains both opcodes and operands (and possibly constant data defined in the code section), a pair of these values is not necessarily two opcodes in sequence, but could be a combination of opcode, operand and data. This is difficult to determine, but the movxcheck

utility evaluates the sequence of bytes immediately preceding the potential opcode pair and eliminates some “false hits” in cases where it can determine that one or both potential problem opcodes are actually operands of a preceding instruction. When an opcode pair is determined to be valid or its validity can’t easily be determined, the script displays a message providing the sequence of machine code values and the address of the occurrence. The user can perform additional analysis manually, as described in this document, to determine whether each potential opcode pair is a legitimate problem sequence or a “false hit” (contains operand or data rather than two opcodes).

6.2.3 Using MOVX Check

Installing the Utility

The installation instructions below use the IRsamples project as an example.

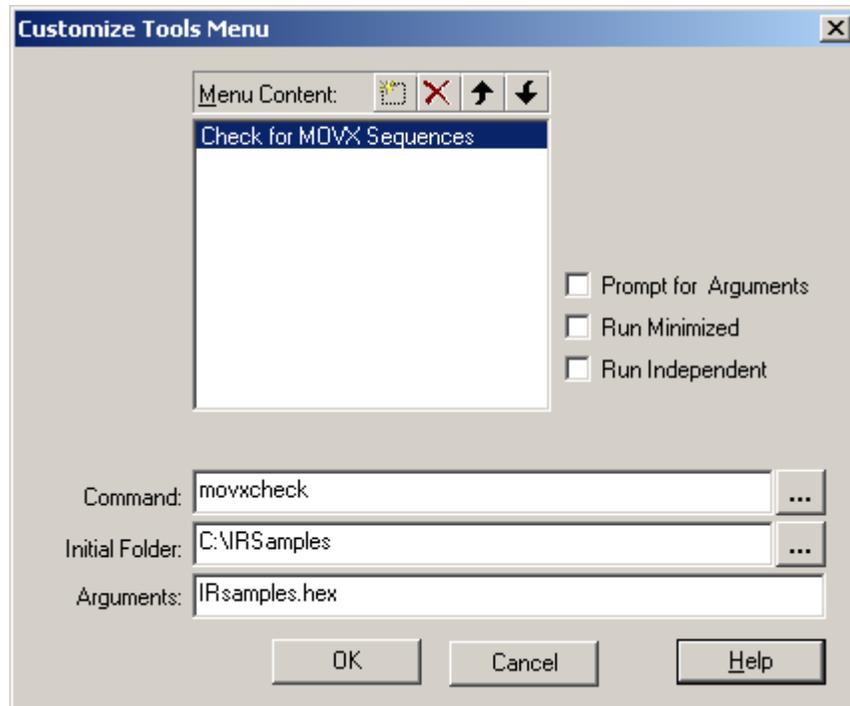
Step 1.

Copy the executable file movxcheck.exe into your uVision project folder (with your 8051 source files). This file can be found in the IRSamples project folder.

Step 2.

Open your project in uVision. From the Tools menu, select Customize Tools Menu. The Customize Tools Menu window allows you to define a new entry for the Tools menu, which you can use to run the movxcheck utility. Set up the new entry as follows:

- In the Menu Content box, enter the text for the Tools menu. In the example below, the text is “Check for MOVX Sequences”.
- In the Command box, enter the name of the utility, “movxcheck”.
- In the Initial Folder box, enter (or browse for) the pathname of your uVision project directory.
- In the Arguments directory, enter the name of the Intel .hex file produced when you build your project. In this example, the file name is IRsamples.hex.



Click OK when you have entered all the information.

Step 3.

Click on the Tools menu and verify that a new option is shown with the text entered in the Menu Content box.

Step 4.

Build your project and then select the new option from the Tools menu to run movxcheck. The output of the utility is displayed in uVision's Build window.

Using the Utility

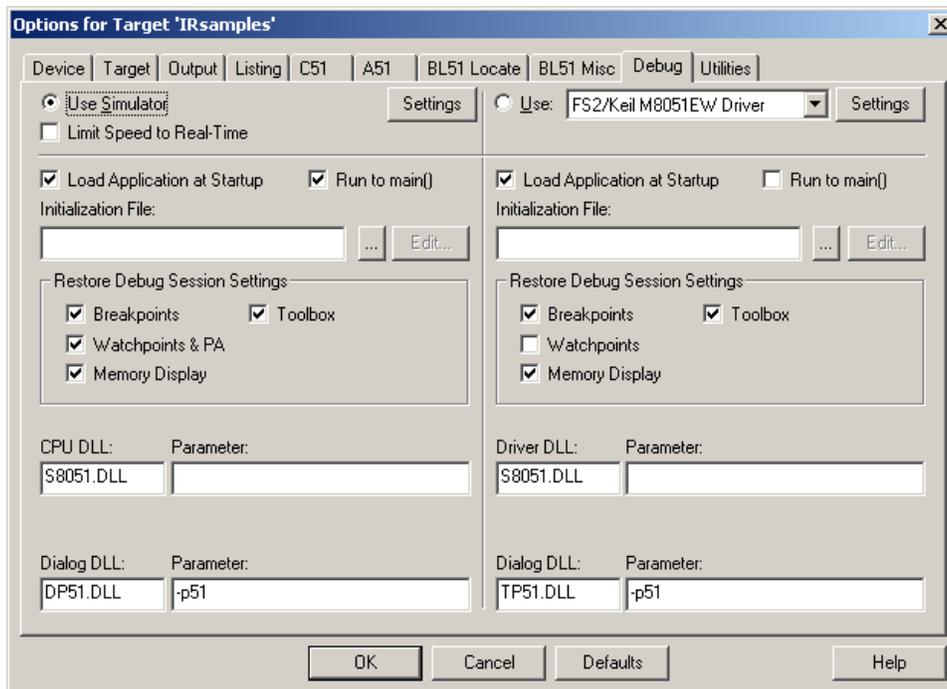
This section describes how to run the movxcheck utility and how to understand and evaluate its output. The steps can be summarized as follows:

1. Modify the uVision project settings to use the simulator for debugging.
2. Build the project to produce a .hex output file.
3. Start a debug session in order to view the program in assembly source code format in uVision's Disassembly window.
4. Run the movxcheck utility. Perform steps 5 – 7 for each potential problem sequence it finds.
5. In the Disassembly window, search for the instruction address closest to the problem address given by movxcheck and match the machine code sequence with the values displayed by movxcheck.
6. Check the assembly source instructions corresponding to the machine code to see whether there are two back-to-back movx instructions.
7. If back-to-back MOVX instructions are found, modify the C code or generate assembly source code and modify it directly to change the instruction sequence. Rebuild and then run movxcheck again to verify that the problem sequence is no longer detected.

Each step is described in detail below. The examples use the IRsamples project.

Step 1.

Select Options for Target from the uVision Project menu and click the Debug tab. Click the Use Simulator radio button in the top left corner of the debug settings window. This selects debugging using the uVision simulator instead of the normal connection to the target device through FS2. Click OK in the debug settings window.



Step 2.

Build the project to produce a .hex output file.

Step 3.

Start a debug session by selecting Start/Stop Debug Session from the Debug menu. A Disassembly window is displayed showing the entire program in assembly source format. (If the Disassembly window isn't displayed, select Disassembly Window from the View menu.)

Step 4.

Run the movxcheck utility by selecting it from the Tools menu. If no problem sequences are found, the utility displays the following message:

```

"movxcheck" IRsamples.hex
IRMCK3xx Problem Instruction CHecker version 1.00.
Copyright 2009 International Rectifier Corp. All rights reserved.
Opening input file IRsamples.hex
No potential problem instruction sequences found in program.
  
```

If movxcheck finds potential problem sequences, it displays a message for each sequence like the one shown below:

```

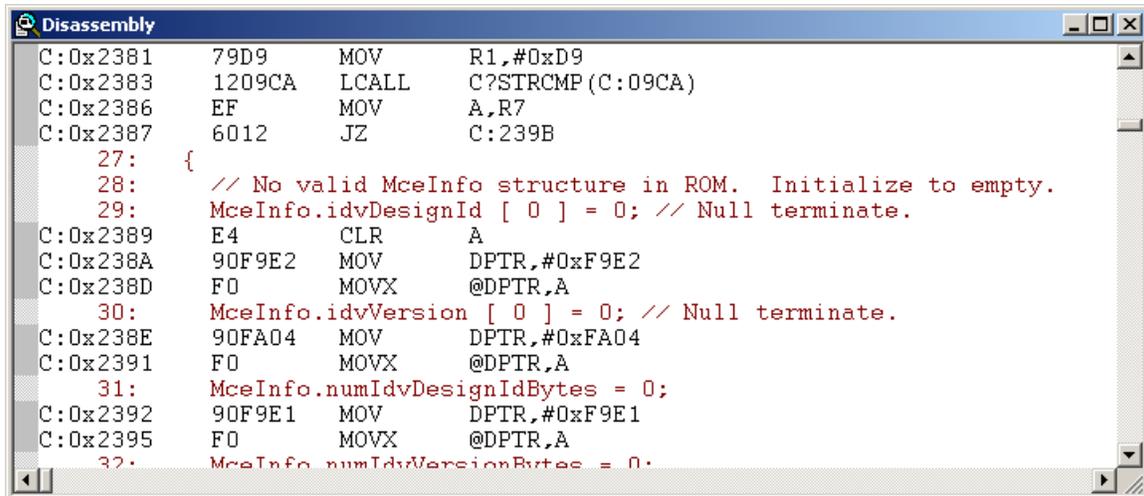
"movxcheck" IRsamples.hex
IRMCK3xx Problem Instruction CHecker version 1.00.
Copyright 2009 International Rectifier Corp. All rights reserved.
Opening input file IRsamples.hex
Potential problem instruction sequence found at address 0x238C.
12 E4 90 F9 E2 F0 90 FA
      ^ ^  ^ ^
  
```

The address shown (0x238C in this example) is the address of the first of the two MOVX opcode values. In the byte sequence on the next line, arrows are shown underneath the two MOVX opcodes. In this example, the two opcode values are 0xE2, 0xF0. Several bytes preceding and following each sequence are displayed to help find the machine code sequence in the Disassembly window (Step 5).

Perform the remaining steps for each potential problem sequence displayed in the Build window.

Step 5.

The Disassembly window shows instruction addresses in the left column, instruction machine code in the second column and assembly source code in the remaining columns. In the Disassembly window, search for the instruction address closest to the problem address given by movxcheck. and match the machine code sequence at that address with the values displayed by movxcheck.



```
Disassembly
C:0x2381 79D9 MOV R1,#0xD9
C:0x2383 1209CA LCALL C?STRCMP(C:09CA)
C:0x2386 EF MOV A,R7
C:0x2387 6012 JZ C:239B
27: {
28: // No valid MceInfo structure in ROM. Initialize to empty.
29: MceInfo.idvDesignId [ 0 ] = 0; // Null terminate.
C:0x2389 E4 CLR A
C:0x238A 90F9E2 MOV DPTR,#0xF9E2
C:0x238D F0 MOVX @DPTR,A
30: MceInfo.idvVersion [ 0 ] = 0; // Null terminate.
C:0x238E 90FA04 MOV DPTR,#0xFA04
C:0x2391 F0 MOVX @DPTR,A
31: MceInfo.numIdvDesignIdBytes = 0;
C:0x2392 90F9E1 MOV DPTR,#0xF9E1
C:0x2395 F0 MOVX @DPTR,A
32: MceInfo.numIdvVersionBytes = 0;
```

Step 6.

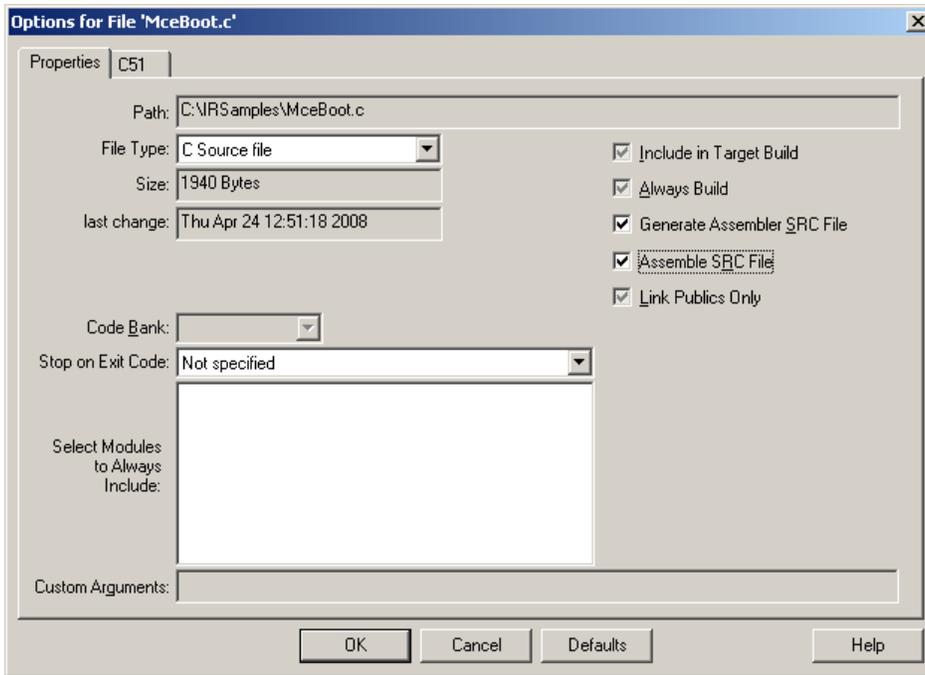
Check the assembly source instructions corresponding to the machine code to see whether there are two back-to-back movx instructions. In this example, it's clear that the 0xE2 0xF0 sequence detected by movxcheck at address 0x238C is not a valid problem sequence. The E2 value is part of the operand of the MOV instruction at address 0x238A, followed by a single MOVX instruction (0xF0) at address 0x238D. Therefore, this instruction sequence will not trigger the MOVX memory access problem and no modification to the software is necessary.

Step 7.

If back-to-back MOVX instructions are found, modify the instruction flow to separate the two MOVX instructions or replace them with an alternate set of instructions. This may be done by modifying the C source code, but sometimes it's difficult to determine what change at the C source level will give the desired result at the machine code level. In that case, you may need to modify the assembly source code directly. The remaining steps describe how to generate and build an assembly source file.

Step 8.

In the uVision Project Workspace window, right click on the C source file that needs to be modified at the assembly source level and select Options for File. (MceBoot.c is used in this example.) Click the Generate Assembler Source File and Assemble SRC File options twice so they're checked with black (not gray) checkmarks as shown in the example below. Click OK to close the Options window.



Step 9.

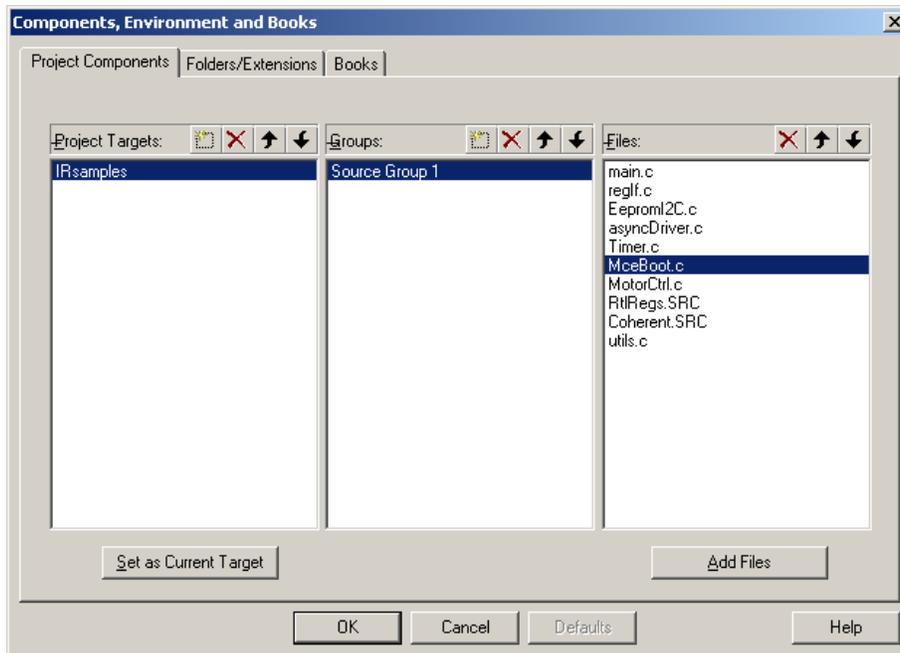
Rebuild the project to translate the C source file into an assembly source file (.SRC). In our example, the new file MceBoot.SRC is created.

Step 10.

Modify the assembly source file (.SRC) as needed.

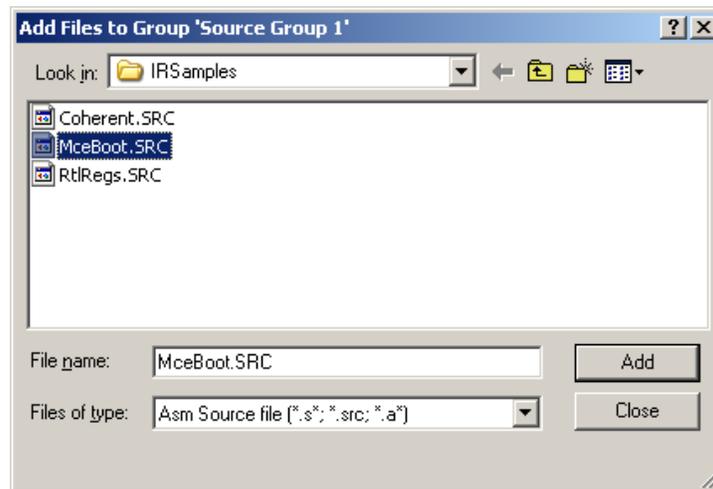
Step 11.

In the uVision Project Workspace window, right click on the project name and select Manage Components.



In the center pane of the Project Components tab, click on the source group that contains your C source file. In the right-hand pane, click on the C source file and then click the Delete button (red X) at the top of the right-hand pane. (This removes the C source file from the project, but doesn't delete the file.)

Now, click the Add Files button at the bottom of the Project Components tab and browse to the new assembly source (.SRC) file. Click Add to add it to the project. Click OK to close the Components window.



Step 12.

Rebuild the project to incorporate the changes you made to the assembly source file. The C source file is no longer compiled because it has been removed from the project; the modified assembly source file, which you added to the project, is assembled instead.

Step 13.

Run movxcheck again to verify that the problem instruction sequence has been resolved.

6.3 Using MCEProgrammer2 to Program the OTP

MCEProgrammer2 is a software utility used to create EEPROM and OTP images, and can perform direct OTP programming by communicating with the FS2 JTAG debugger or a USB-based Corelis (USB-1149.1/E or USB-1149.1/1E) JTAG device. This utility can be used to import a design's MCE program (.bin file) and 8051 program (.hex file) and then program it to OTP.

To program the OTP, the user can either design their own hardware to communicate with the IRMCK3xx through the JTAG pins, or use the IRMCS3xxPROG Programming Board and MCEProgrammer2 utility provided by IR. The JTAG header on the Programming Board is designed to connect directly to the Corelis JTAG device. To use the FS2 debugger with the Programming Board, some of the pins must be rewired. Programming the OTP is considerably faster using Corelis than with FS2.

The Programming Board has an IRMCK3xx socket on it and comes with a 12V power supply. An on board power supply provides 3.3V, 1.8V and 6.75V to the OTP part. Connect the JTAG pins to the JTAG controller (Corelis USB-1149.1/E or USB-1149.1/1E JTAG device), which connects to the PC via a USB port. There is an OTP Programming board for each of the versions, IRMCK341, IRMCK343, IRMCK371, IRMCK311 and IRMCK312.

If the FS2 debug pod is used for programming the OTP, then the user must setup the location of the FS2 Console. From Tools, select “FS2 Setup” and enter the location of the FS2 executable. If the FS2 program is installed in the default location, then this location will be:

```
C:/Program Files/Fs2/m8051ew/Bin/clisysnav.exe
```

The MCEProgrammer2 software utility provided by IR performs the OTP programming. At the main window, select the “Product” name from a list (i.e. IRMCx341), and select the “Operation”. If OTP programming is desired, then choose “Program OTP via Corelis (or FS2) JTAG” and provide the locations of the 8051 .hex file and MCE .bin file. To protect the contents of the OTP from being read back, check the “Protect OTP” box. By default, there is no protection and the contents of the OTP can be read back properly. “MCE Info Structure Address” allows the user to change the location of the MCE Info structure, which is only required in order to work with MCEDesigner. There is a default location for each product. After all the options are set, click “Execute” and a progress bar window will pop up to show the progress. If the FS2 debugger is used, then the FS2 console window will come up. After the programming is done, MCE Programmer will automatically read back the OTP content and verify if the programming is correct. If the verification fails, an error window will appear giving the address of the first byte with incorrect data.

6.4 Creating Custom Programming Methods

IR provides a programming kit to aid in programming the OTP. If the user chooses to design a custom programming method, detailed specifications are given in this section.

6.4.1 OTP Image Generation

MCEProgrammer is a utility that can be used to create IRMCx300 OTP memory images. These .bin files can then be integrated in any custom OTP programmer. The binary file created is the full address range of OTP (64K) and goes from address 0x0000 to 0xFFFF.

6.4.2 JTAG Test Mode

The IRMCK3xx has a single JTAG port that can be used to either debug the embedded 8051 microprocessor or to program the OTP memory. To program the memory the controller must be put into ‘Test Mode’. Once in this mode the OTP memory will be available over the JTAG interface.

6.4.3 Programming Pins

The OTP programming is performed over the standard JTAG interface available on the IRMCK3xx. In addition to this standard interface, a supply voltage of 6.75V must be supplied to the OTP during programming. This voltage is supplied to the dual-purpose pin SCL/VPP. When 6.75V is supplied to this pin it will act as the supply rail for the internal OTP memory. If performing programming in-circuit it is important to either verify that external components can withstand 6.75V or isolate the SCL/VPP pin during programming. An easy way to isolate these devices is to have a jumper present on the SCL line; after the OTP is programmed the jumper can be put in place.

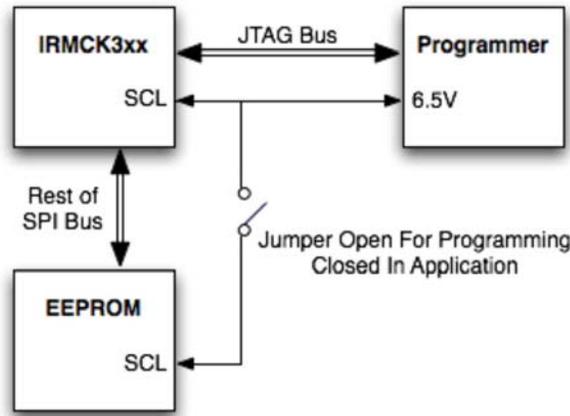
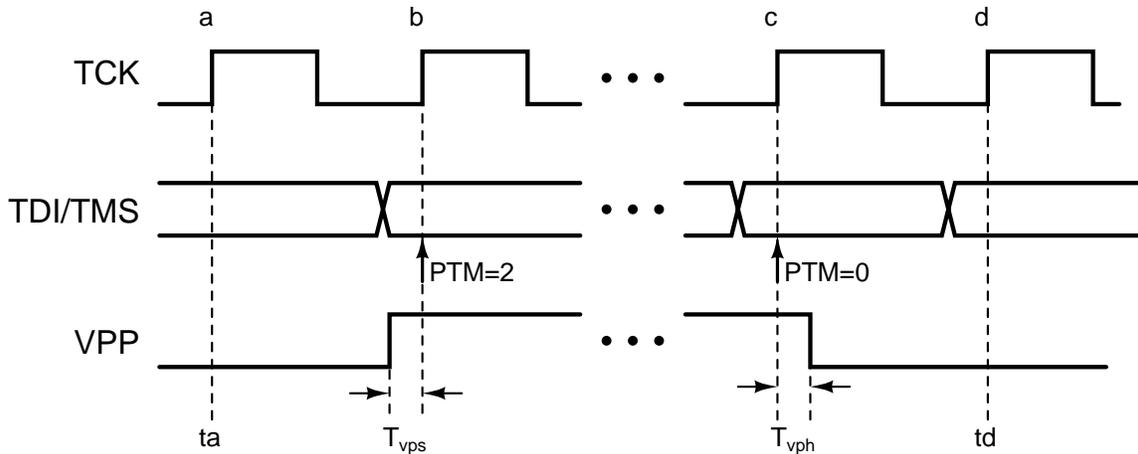


Figure 13—EEPROM Isolation Example

During OTP read mode, the VPP pin can be at either VDD or VSS, or floating. When OTP is configured into program mode, the VPP pin has to be high at least 10ns before the rising edge of TCK (JTAG clock input pin). When OTP is configured back to read mode, the VPP pin has to be high at least 15ns after the rising edge of TCK, as shown in the timing diagram below. VPP high requires a typical 6.75V supply voltage, with 6.5V minimum and 7.0V maximum.



Note: PTM=2 means OTP is configured into program mode; PTM=0 means OTP is configured back to read mode. T_{vps} =10ns, T_{vph} =15ns.

6.4.4 JTAG Overview

The JTAG interface in the IRMCK3xx is the standard four pin configuration. Data is shifted into TDI and shifted out of TDO. The state machine is controlled via the TMS line and TCK is the clock for communication. (Table 6)

Pin Name	TCK	TMS	TDI	TDO
Function	Clock	State Machine	Serial Data In	Serial Data Out
Direction	Input	Input	Input	Output

Table 6—JTAG Pins

6.4.5 TCK Cycle

Over the course of the JTAG programming cycle, data should be loaded into TMS and TDI on the negative edge of TCK. TDO will change output states on the negative edge of TCK. Data will be sampled from the TMS and TDI lines on the positive edges of TCK.

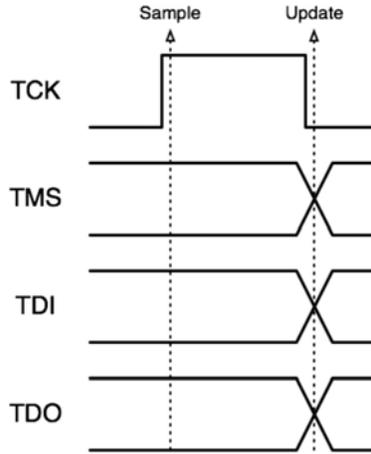


Figure 14—Basic JTAG Cycle

6.4.6 JTAG Registers

Two registers are present in JTAG implementations, IR (Instruction Register) and DR (Data Register). The JTAG interface will shift in and out the IR and DR registers, respectively, as it performs functions. To perform a command an instruction code is loaded into IR, and then the data associated with that command is loaded into DR. The order of IR and DR loads is dependent on the type of command being performed. There are additional registers defined for performing burn and verify operations listed below (Table 7). These registers can be written to and read from with specific IR values. It should be noted that the DR register is not a physical register, but can be treated as one with respect to how it behaves in the system. More information can be found in IEEE Std 1149.1.

Register	Width	Description
IR	8 bits	Instruction Register
DR	16 bits	Data Register
otp_setup	8 bits	The OTP programming configuration register
otp_wr_timer	8 bits	Number of TCK clocks X 64 to write one byte of data
otp_jtag_address	16 bits	The current address of OTP that data will be burned to
otp_data	8 bits	The data byte that will be burned to OTP
test_modes	16 bits	Defines what mode the test interface is in (OTP Burn-in)

Table 7—The OTP Programming Registers

OTP_Setup[8:0]

The OTP_Setup register contains configuration information for accessing the OTP. It is defined as follows.

Address Advance			SKIP	POEB	PTM		
7	6	5	4	3	2	1	0
OTP_Setup.7	-	AdAdv[2:0]					
OTP_Setup.5							
OTP_Setup.4			SKIP				
OTP_Setup.3			POEB				
			Number of address to advance in auto-increment modes (# to advance = 2 [^] OTP_Setup[7:5])				
			Skip OTP_Addr[5:4]:0000 in auto-increment mode				
			OTP Output Enable Bit 1: Enables OTP memory read access				

		0: Must be zero for programming
OTP_Setup.2 – OTP_Setup.0	PTM[2:0]	Program Test Mode 000 = Read OTP 010 = 1x Normal Programming 011 = 4x Accelerated Programming

Table 8—OTP_Setup Register Definition

OTP_Wr_Timer[7:0]

This register adjust how long a OTP write operation should last. The write time is OTP_Wr_Timer[7:0] X 64 TCK cycles. This value then is dependent on the chosen TCK rate and minimum OTP write time.

OTP_JTAG_Address[15:0]

This register is the address that the next OTP data write command will go to. This is where the value written to DR will be written to OTP_JTAG_Address[15:0]. This register will auto-increment the OTP address after each write or read when OTP_Setup.4 is enabled.

OTP_Data[7:0]

This register is the data that the next OTP data write command will use to write to OTP_JTAG_Address[15:0]. This register is normally not accessed directly.

Test_Modes[15:0]

The Test_Modes[15:0] register defines what specific functions are to be performed by the test interface of the IRMCK3xx controller. Only one mode is needed for OTP programming, which is entered by writing 0x0002 into this register. This allows the TCK clock input to become the main system clock and, which allows synchronization between the control interface and the OTP memory.

6.4.7 IR Write Commands

IR Command	Command Description
0x00	Read JTAG TAP controller ID
0xF5	Enter the 'test mode' for OTP memory access
0xF6	Exit the 'test mode' and return to standard 8051 JTAG interface
0x70	Write contents of DR Test_Modes[15:0]
0x50	Write contents of DR to OTP_Setup[7:0]
0x51	Write contents of DR to OTP_Address[15:0]
0x52	Write contents of DR to OTP_Data[7:0]
0x54	Write contents of DR to OTP_Timer[7:0]
0x71	Enable auto-increment burn mode for OTP programming (see below)

Table 9—IR Write Command List

In order to perform these write commands the following actions should be taken.

- Step 1:
Load IR
- Step 2:
Load DR
- Step 3:
System will auto-execute instruction with new DR

The 0x71 auto-increment command is slightly different in its operation. It exists to reduce the number of commands during programming. When 0x71 is set in IR every following DR load will cause that data to be written to the OTP and then automatically increment the

OTP_Address[15:0] register. Therefore when programming the IR is set to 0x71 and then DR loads are repeated until the memory is fully programmed.

6.4.8 IR Read Commands

IR Command	Command Description
0x60	Read OTP_Setup[7:0] to DR
0x61	Read OTP_JTAG_Address[15:0] to DR
0x62	Read OTP_Data[7:0] to DR
0x63	Read OTP_Wr_Timer[7:0] to DR
0x64	Read OTP_Wr_Counter[15:0] to DR
0x72	Enable auto-increment read mode for OTP verification

Table 10—IR Read Command List

When the IR register is loaded with a read command it will return the specified register value to the DR register. This value can then be returned by reading DR and shifting it out over the TDO line.

6.4.9 TMS State Machine Diagram

This diagram describes is the standard JTAG state machine. Through use of only the TMS signal line either IR and DR values can be loaded into the system. For more information about JTAG refer to IEEE Std 1149.1.

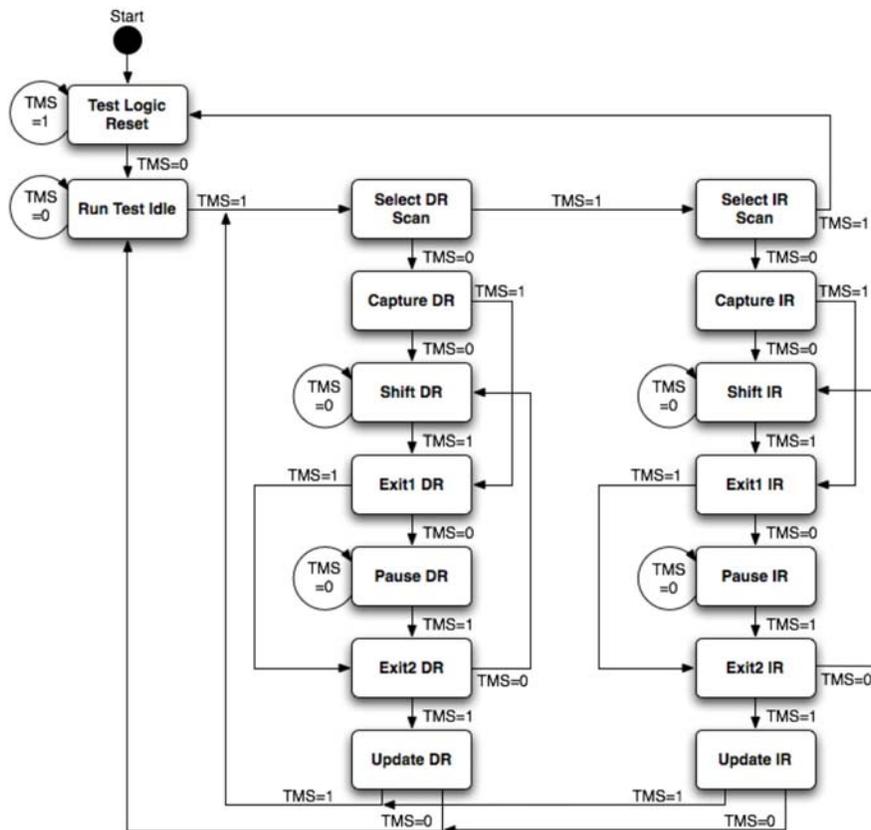


Figure 15—JTAG TMS State Diagram

6.4.10 OTP Programming Example

- Write 0xF5 to IR - Enter test mode
- Write 0x70 to IR then 0x0002 to DR - Set TCK to main system clock
- Write 0x54 to IR then 0x07 to DR - Set OTP_Wr_Timer to 0x07
- Write 0x50 to IR then 0x0A to DR - Set OTP_Setup to 0x0A

Write 0x51 to IR then 0x0205 to DR
 Write 0x71 to IR
 Write 0xA2 to DR
 Toggle TCK
 Write 0xA3 to DR
 Toggle TCK
 ...
 Write 0xF6 to IR

- Set OTP_Address to 0x0205
- Enable auto-increment OTP write
- Write 0xA2 to OTP address 0x0205
- # of pulses according to Wr_Timer
- Write 0xA3 to OTP address 0x0206
- # of pulses according to Wr_Timer
- Exit test mode

6.4.11 OTP Verification Example

Write 0xF5 to IR
 Write 0x70 to IR then 0x0002 to DR
 Write 0x50 to IR then 0x00 to DR
 Write 0x51 to IR then 0xFFFF to DR
 Write 0x72 to IR
 Read DR
 Write 0x51 to IR then 0x0205 to DR
 Write 0x72 to IR
 Read DR
 Read DR
 Read DR
 ...
 Write 0xF6 to IR

- Enter test mode
- Set TCK to main system clock
- Set OTP_Setup to 0x00
- Set OTP_Address to 0xFFFF
- *Italic steps disable OTP read protection*
- *Dummy read*
- Set OTP_Address to 0x0205
- Enable auto-increment OTP read
- Dummy read
- Read OTP address 0x0205
- Read OTP address 0x0206
- Exit test mode

6.4.12 OTP IR Load Example

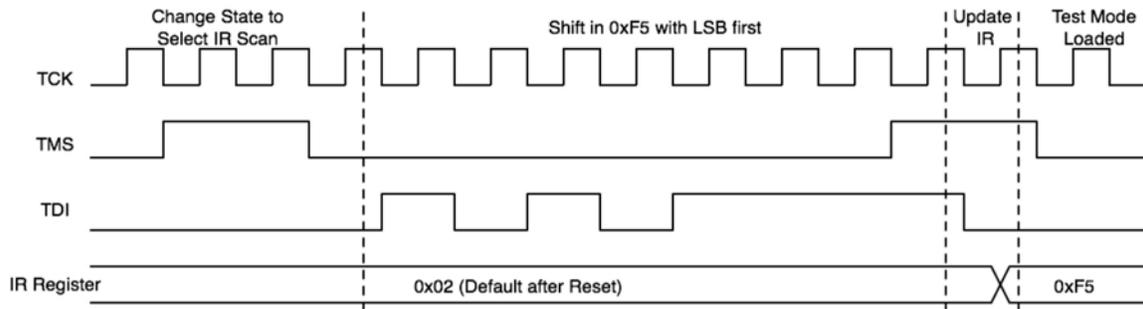


Figure 16—JTAG Load 0xF5 to IR (Enter Test Mode)

6.4.13 OTP Timing Information

There are three important parameters required when programming the OTP through the JTAG interface. This information is listed below.

Parameter	Symbol	Min	Max	Unit
Clock Cycle Time	T_{cyc}	25	-	ns
Program Pulse Width	T_{pw}	90	110	μ s
Program Pulse Interval	T_{pwi}	5	-	μ s

Table 11—Critical OTP Programming Timing Information

Trademarks of Infineon Technologies AG

μ HVIC™, μ IPM™, μ PFC™, AU-ConvertIR™, AURIX™, C166™, CanPAK™, CIPOS™, CIPURSE™, CoolDP™, CoolGaN™, COOLiR™, CoolMOS™, CoolSET™, CoolSiC™, DAVE™, DI-POL™, DirectFET™, DrBlade™, EasyPIM™, EconoBRIDGE™, EconoDUAL™, EconoPACK™, EconoPIM™, EiceDRIVER™, eupec™, FCOS™, GaNpowIR™, HEXFET™, HITFET™, HybridPACK™, iMOTION™, IRAM™, ISOFACE™, IsoPACK™, LEDriviR™, LITIX™, MIPAQ™, ModSTACK™, my-d™, NovalithIC™, OPTIGA™, OptiMOS™, ORIGA™, PowIRaudio™, PowIRstage™, PrimePACK™, PrimeSTACK™, PROFET™, PRO-SIL™, RASIC™, REAL3™, SmartLEWIS™, SOLID FLASH™, SPOC™, StrongIRFET™, SupIRBuck™, TEMPFET™, TRENCHSTOP™, TriCore™, UHVIC™, XHP™, XMC™

Trademarks updated November 2015

Other Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2010-04-05

Published by

Infineon Technologies AG
81726 Munich, Germany

© 2016 Infineon Technologies AG.
All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

IRMCx300_SWDevGuide

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.