TEXAS INSTRUMENTS

MSP430 Family Serial Programming Adapter Manual

User's Guide

August 2000

Mixed Signal Products

SLAU048

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

This document describes the MSP430-family hardware and software installation and setup. It explains operation and EPROM programming

How to Use This Manual

This document contains the following chapters:

- Chapter 1 – Installation and Setup
- Chapter 2 – Operation
- Chapter 3 – Hardware
- Chapter 4 – EPROM Programming
- Chapter 5 – Flash Memory
- Appendix A– Schematics

Notational Conventions

This document uses the following conventions.

- Program code and program examples are shown in a `special typeface` similar to a typewriter's.

  Here is an example of programming code:

  ```
  long int VerifyFile(char* lpszFileName, long int iFileType)
  ```

Trademarks

Microsoft Windows is a trademark of Microsoft Corporation.

Contents

Figures

Tables

Chapter 1

Installation and Setup

This chapter describes the process of installing and programming the hardware and software for the MSP430–PRGS430 programming adapter used with the MSP430 family of microcontrollers.

Topic **Page**

1.1 Installing the Software

To install the MSP-PRGS430 software, perform the following steps:

1) Insert the MSP-PRGS430 CD-ROM in the computer's CD drive. It should start automatically. A setup routine will check if you have an HTML browser installed on your computer. The MSP430 start page will then be displayed. (Alternatively, use a browser to open the file *index.htm* that is located in the root directory of the MSP430 CD-ROM. The MSP430 start page will then be displayed.)

2) Select *Software*

3) Select *Serial Programming Adapter*

4) Select *Save it to disk*. A *Save As* dialog will be displayed.

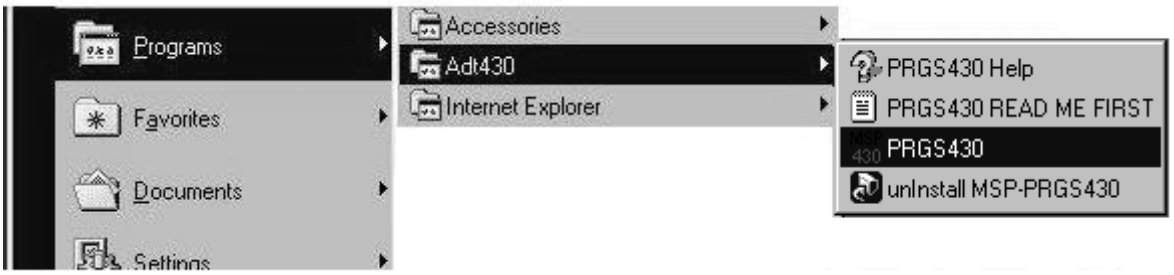
5) Use the Save As dialog to save *PRGS430_inst.exe* to the computer. Note the directory path to this saved file.

6) Navigate to this saved file (PRGS430_inst.exe), and execute it. A welcoming message will be displayed.

7) Follow the setup instructions on the screen. The setup program guides you through the installation process.

8) After you run setup, the MSP430 program icons are displayed. Double-click the Read me PRGS430 icon, shown in Figure 1–1, to obtain important information about the program device hardware and software.

Figure 1–1. ADT430 Program Icons



9) The appropriate program group and icons are added to the Windows program manager.

10) To start the programming adapter software, double-click the Program Device icon in the ADT430 program group.

1.2 Installing the Hardware

To install the programming adapter hardware, perform the following steps:

1) Using the 9-pin SUB-D connector, connect the programming adapter to the serial port (COM1–COM4) of the PC.

2) Connect an external power supply to the programming adapter. The voltage of the power supply must be between 14 V and 20 V dc and must provide a minimum of 200 mA of power. The center terminal of the supply connector at the programming adapter is the plus pole.

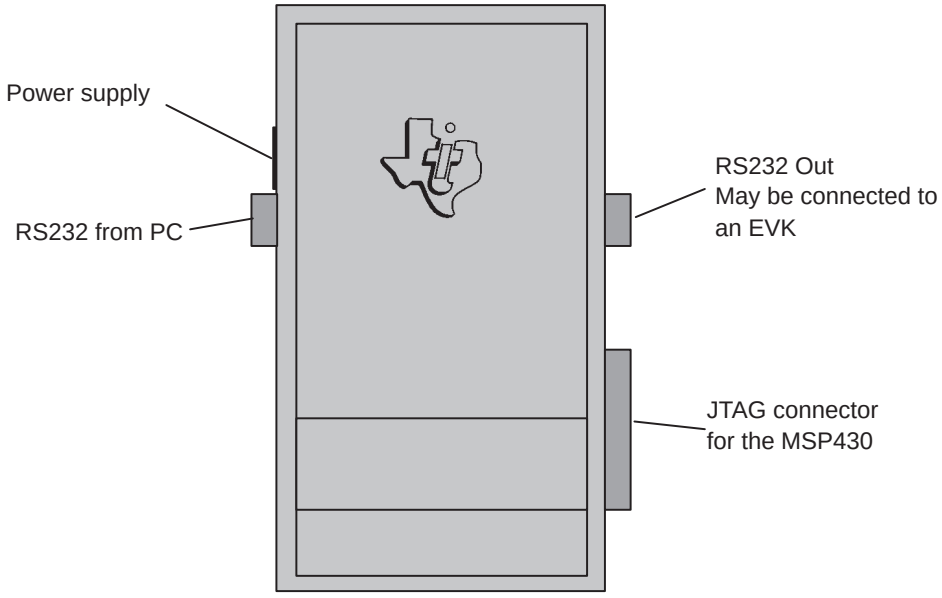
3) The red LED on the programming adapter lights if the power supply is properly connected. If the LED does not light and the power supply is properly connected, check the F1 fuse on the programming adapter printed wire board (PWB).

4) The MSP430 devices, in a socket or on a PWB, should be connected to the programming adapter through the 14-pin cable.

   The programming adapter provides the selected supply voltage V_{CC} at pin 14 of the 25-pin SUB-D connector, or at pin 2 of the 14-pin connector to supply the MSP430 device. The signal name is VCC_MSP.

   The voltage at MSP_VCC should be set to the same voltage level of the external V_{CC} if the device is supplied externally, for example, during in-circuit programming.

Figure 1–2. Program Adapter



Chapter 2

Operation

This section describes the programming procedure for MSP430 devices and the error messages you may encounter during the procedure.

Topic **Page**

2.1 Programming the MSP430 Devices

2.1.1 Basic Procedure

The following steps should be used to program the MSP430 devices:

1) Click on the Program Device icon in the ADT430 program group. The MSP430 program device dialog box appears.

   The status line at the bottom of the window shows the actual or the most recent activity (see Figure 2–1)

   The status line displays the message *Connecting to adapter...* until the programming adapter is detected and the baud rate is set.

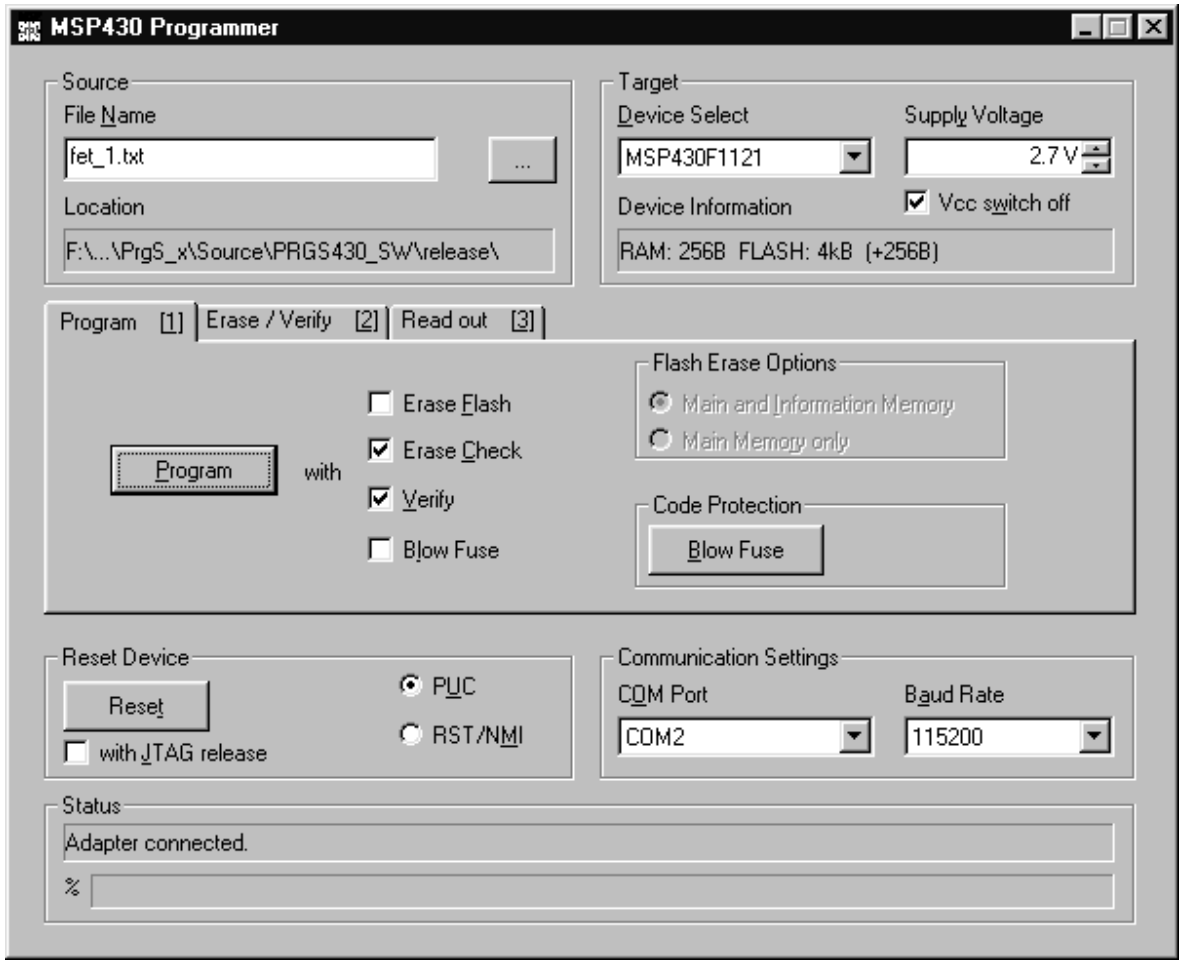
2) Select the correct device and supply voltage

3) Select the name of the object file

4) Select the additional options to program, if necessary (using Erase Flash/ Erase Check/Verify...)

5) Click on the *Program* button to start programming
   The status line at the bottom of the window shows the actual or most recent activity (see Figure 2–1)

Figure 2–1. MSP430 Programmer Dialog Box



2.1.2 Description of the MSP-PRGS430 GUI

An MSP430 device is commonly programmed as follows:

1) Select the file which contains the data to program from the MSP430 Programmer dialog box (see Figure 2–1).

2) Select the device. An error message appears on the screen if the device selected is different or not connected.

3) Set the required supply voltage, communication port COMx, and baud rate. The device configuration and memory type are selected automatically according to the selected device.

4) Use the program button to start the programming operation.

Table 2–1 describes the function of the buttons for different options and combinations for the MSP430 Programmer dialog box.

Table 2–1. MSP430 Function Buttons and Descriptions

Button Name	Sub-Functions	Description
File Name		Selects the name of the file to program (intel-hex or TI-txt format)
Device Select		Selects the MSP430 device type to program via-pull-down menu
Supply Voltage		Selects the supply voltage for the MSP430
	V_{CC} switch off	If selected (default), the supply voltage will be switched off after each MSP430 access; otherwise, the supply voltage remains connected.
Program		An object code is programmed to the on-chip memory using the select options.
	With Erase Flash	Memory will be erased before programming (only with flash devices). The following options are possible: – Main and Information Memory – Main Memory only
	With Erase Check	Erase check will be performed before programming operation is executed.
	With Verify	Each section is verified after it is programmed, or an error message is displayed if verification fails.
	With Blow Fuse	The code-protection fuse is blown after the entire object code, with verify, is programmed. This action is irreversible and disables future on-chip memory access (reading or programming). This step will not be performed if verify is disabled or verify fails. A warning is displayed.
Erase Flash		Erase operation can be done only with flash devices, according to the selected option.
	By file	Only the memory locations corresponding to the selected object file are erased. All other memory locations keep their old data (*smart erase*).
	By device	The entire flash memory of the device is erased.
	By range	An erase is performed depending on the values entered in the range fields.
Erase Check		Checks if memory locations are erased.
	By file	Checks only the memory locations used by the selected object file.
	By device	Checks the entire programmable memory of the device. (No RAM will be checked).
	By range	An erase check is performed according to the range of memory locations in the range for Erase Check/Readout field.

Table 2–1. MSP430 Function Buttons and Descriptions (Continued)

Button Name	Sub-Functions	Description
Verify		Verify the data in the MSP430 device according to the selected option
	By file	A verification of the memory locations vs. the selected object file is performed.
	By device	
	By range	Verify memory locations defined in the *range field* versus the data in the selected file. The defined range should not contain memory locations outside the data stored in the selected file, otherwise an error will be reported.
Blow Fuse		The on-chip security fuse is irreversibly disabled and any access such as reading or programming of the MSP430 is impossible.
Read Out		Read out data from MSP430 device. When this function is executed, a dialog box will appear; the file name for the data to store should be selected.
	By device	Read out the entire memory of the device and store the data into the file selected in the file name field.
	By range	Read out the memory locations selected by the *range field* and store the data in the file selected in the file name field.
Reset		The reset of a MSP430 can be performed in two ways. After reset, the MSP430 may remain under JTAG control or can be released to operate normally and execute the program.
	PUC	A software reset of the chip is generated.
	RST/NMI	Generates a hardware reset by applying a low pulse on RST/NMI pin.
	With JTAG-release	JTAG will be released after the execution of the reset (via JTAG or RST/NMI).
COM Port		Selects the Com port to which the programming adapter is connected
Baud Rate		Selects the baud rate for communication with the programming adapter hardware
Help		Help is available for programming MSP430 devices, command buttons, selectors, and the object file format used. The Help menu can be found in the system menu of the serial programming adapter software (right click on the symbol at the upper left corner of the program window) or with the F1 function key.

2.1.3 Error Messages

One of the following messages may show up if JTAG communication is not established correctly:

If the MSP430 device to program could not be found, the following message appears (Figure 2–2):

Figure 2–2. Communication Error Box

If the fuse is already blown, the error message shown in Figure 2–3 appears.

Figure 2–3. Communication Error Box for Blown Fuse

Additional message boxes appear for general error messages such as Erase Check (Figure 2–4).

Figure 2–4. Erase Check Error Message

When a read error is detected in the input file, such as a format error, the following message will be displayed (Figure 2–5):

Figure 2–5. Data Error

Table 2–2. Error Messages

Error Type	Error Message
Communication	Communication failed!
Communication	Adapter not connected!
Communication	Synchronization with adapter failed!
Communication	The present adapter is not a MSP-PRGS430!
Communication	Missing setting of V_{CC}!
MSP430	Target not connected!
MSP430	Wrong JTAG version!
MSP430	PUC failed!
MSP430	Wrong target!
MSP430	Target fuse is blown!
MSP430	Blown fuse failed!
MSP430	Supply voltage to low!
MSP430	Fuse not released for this device!
Setting	Unknown target!
Setting	No target selected!
Setting	Wrong V_{CC} selected!
Setting	Wrong baud rate!
Setting	Communication port error!
Setting	The selected range is invalid!
Setting	Wrong argument!
Setting	Error at target address (during erase check or verify)
Setting	Unknown command line option
Setting	Command line option out of valid range
System	DEVICE.CFG corrupted
System	General error!
System	File type could not be detected!
System	Unexpected end of file!
System	PROJECT.INI corrupted!
System	Filename mismatch
System	Error in DEVICE.CFG
Windows	Error during file I/O

2.2 Content of the PRGS430.ini File

The last settings of the PRGS430 graphical user interface (GUI) will be stored in the .ini file before exiting the program. This information is stored under the [Program Device System] section.

Additionally, the following parameters are in the [Options] section and may be modified:

[Options]

\BlowFuse = 1 → The blow fuse button in the GUI is disabled to prevent accidental blow of the irreversible fuse.

LastResult = 0 → If the program is called with command-line parameter, the error code which is returned to the system when exiting the program will also be stored here.

2.3 Use of a [Project].ini File

Some default options could be changed within a [Project].ini file. This file has to be in the same directory as the object code file. The following variables could be defined or redefined there.

The name of the file should have the same name as the object file with the extension .ini.

[ProgramDevice]
UserMemProtect = Start, Size
UserMemProtect2 = Start, Size
UserMemProtect3 = Start, Size
UserMemProtectn = Start, Size
DisableTIMemProtect = 0

Memory ranges defined in the UserMemProtect and UserMemProtect [n] option will be read out and reprogrammed after erase (Flash devise only). [n] could be a number greater or equal then 2 and have to be in ascending order.

If a memory protection is activated in the device definition file from Texas Instruments, it could be switched off with the DisableTIMemProtect = 1 option.

2.4 Command Line Options

2.4.1 General Definitions

0: Off 1: First selectable option
1: On 2: Second selectable option
3: Third selectable option

The PRGS430.ini file options are used if they are not specified in the command line. The command line option overwrites the ini file options.

The program will exit automatically if a command is passed via the command line and the command was executed. There will only be a small status window opened during the execution.

Only one command identifier (/CMD:) is allowed within the command line. Otherwise the execution will be canceled and an error will be returned.

If an error in the command line parameter is detected the program will exit with an error message.

filename may also contain a path. If special characters are used then the string has to be inside quotes (for example, \\server\adt430\PRG files\test.txt).
If an error is detected within the filename, the operation will be canceled and an error will be returned

Table 2–3. Command Line Options

Commands:	
/cmd:PRG	*Program* command
/cmd:VFY	*Verify* command
/cmd:ERS	*Erase* command
/cmd:CHK	*Erase check* command
/cmd:READ	*Read out* command
/cmd:RST	*Reset* command
/cmd:BLOW	*Blow fuse* command
Options:	
/COM:x	Specifies the serial port: /COM:1, /COM:2 /COM:3, or /COM:4
/BR:xxxxxx	Sets baud rate to be used: 9600/19200/38400/57600/115200, e.g. /BR:57600
/Dev:	Selects the device according to the name in the device.cfg file, e.g. /Dev:MSP430F1121
/SVolt:x.x	Selects supply voltage MSP_VCC of the programming adapter. The voltage is supplied between GND and MSP VCC, e.g., /Svolt: 3.0
/SVoff:{0,1}	Switches off supply voltage MSP_VCC after execution 0: Disable (do not switch off) 1: Enable (switch off)
filename	Specifies name of the object file to be programmed or verified
/FILE filename	(Second way to define the filename – space separated)

Table 2–3. Command Line Options (Continued)

Options for Program Command:	
/PE:{0,1,2}	Option program with erase (flash only) 0: Without erase 1: Main and Info memory 2: Main memory only
/PC:{0,1}	Option program with erase check 0: Disable 1: Enable
/PV:{0,1}	Option program with verify 0: Disable 1: Enable
/PB:{0,1}	Option program with blow fuse (only valid with verify successful) 0: Disable 1: Enable
Options for Erase/Erase Check and Verify Command	
/E:{1,2,..}	Option erase/erasecheck/verify by file/device/range 1: File 2: Device 3: Range
/ERange:0xXXX, 0xYYYY	Option erase/erasecheck/verify range (start: 0xXXXX, length: 0xYYYY)
Options for Read Out Command:	
/RO:{1,2}	Option read out by device/range 1: Device 2: Range
/RRange:0xXXXX, 0xYYYY	Option read out range (start: 0xXXXX, length: 0xYYYY)
/Rfile:file-name,{1,2}	Specifies read out file name 1: TI–TXT 2: Intel–Hex (Default directory should be the last object file directory)
Log Options:	
/Log:filename	Specifies Log file name (Default directory should be the PRGS430.exe directory)
/ALog:{0,1}	Option accumulative Log file 0: Disable 1: Enable

Example:

PRGS430.exe ”C:\adt430\test\test.txt” /Dev:MSP430F1121 /cmd:PRG /PE:1 /PC:0 /PV:1 /COM:2

This command programs the file test.txt, located in the directory C:adt430\test, into a MSP430F149 device. The device will be erased before programming. The erase check is disabled. The code will be verified after programming. The programming adapter is connected to ComPort 2. The baud rate is not passed with the command line, so the setting in the *PRGS430.ini* file will be used.

2.4.2 Return Values / Error Codes in the .ini File

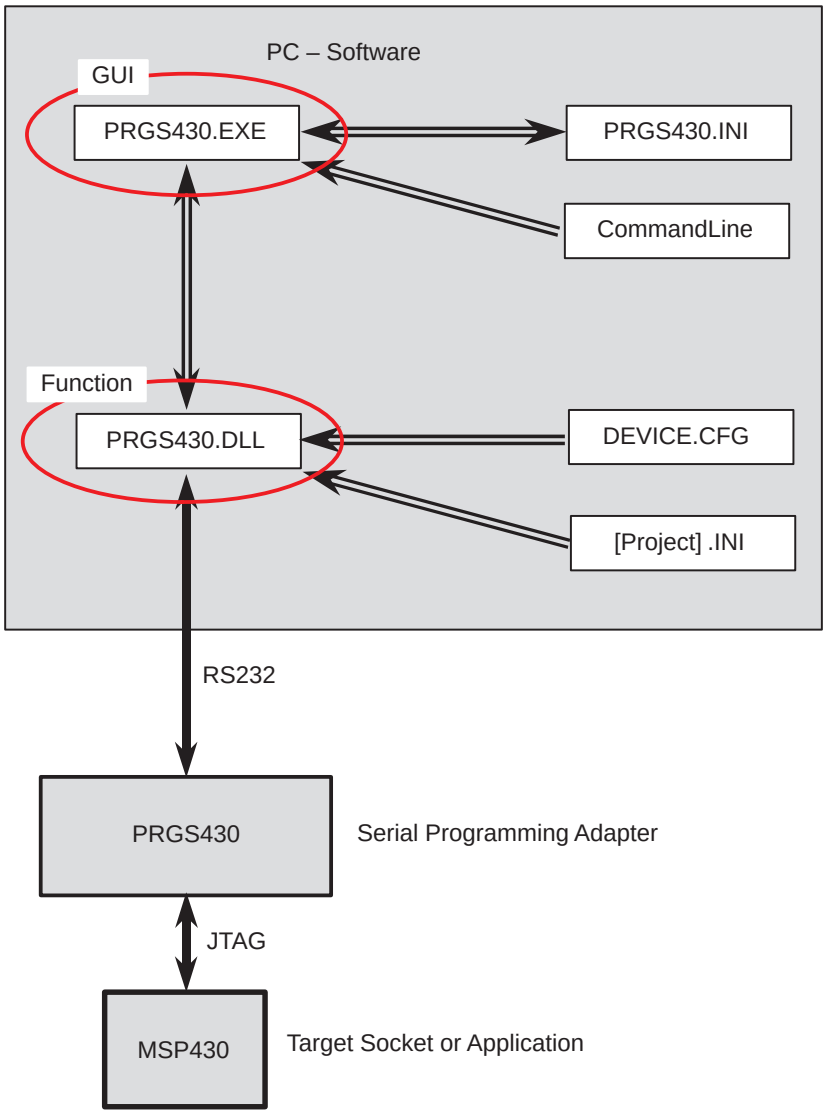
The error code will be returned to the PC operating system and is also stored in the 'PRGS430.ini'

File in the [Options] section:

LastResult=0

0	Ok
2	Communication failed!
3	Target not connected!
4	Adapter not connected!
5	Wrong JTAG version!
6	PUC failed!
7	Synchronization with adapter failed!
8	The present adapter is not a MSP–PRGS430!
9	Unknown target!
10	Wrong target!
11	No target selected!
12	Target fuse is blown!
13	Blow fuse failed!
14	Missing setting of Vcc!
15	Wrong Vcc selected!
16	Wrong baudrate!
17	Communication port error!
18	DEVICE.CFG corrupted!
19	General error!
20	The selected range is invalid!
21	Wrong argument!
22	Error during file I/O.
23	File type could not be detected!
24	Unexpected end of file!
25	PROJECT.INI corrupted!
26	Vcc Voltage to low for selected function!
27	Fuse not release for this device!
101	Error at target address (during erase check or verify)
102	Unknown command line option
103	Command line option out of valid range
104	Filename mismatch
105	Error in device.cfg

2.5 Software / Hardware Layers



2.6 PRGS430.DLL—Description

The PRGS430.dll is used to communicate with the MSP–PRGS430 hardware and the connected MSP430 device.

This dll could be used separately using the following conventions:

/FN0001/ InitCom

long int InitCom(char* lpszComPort, long int lBaudRate)

InitCom initializes (opens) the given communications port, establishes communication with the PRGS430 hardware, and sets the baud rate of the MSP–PRGS430. If successful, the MSP–PRGS430 is reset and Vcc is set to 0.0 V (the voltage should be set after the first user action to validate the correct value).

lBaudRate: valid baud rates are: 9600, 19200, 38400, 56800, and 115200 baud. The default baud rate after installation is 115200 baud.

lpszComPort: the name of the communication port—COM1, COM2, COM3, or COM4.

Example: `lFuncReturn = InitCom(”COM1” 115200)`

/FN0002/ ReleaseCom

long int ReleaseComm (void)

This new function is the counterpart to InitCom. It allows to close a communication with the MSP-PRGS430 hardware.

Vcc will be set to 0 and all outputs will be set to the HI-Z state.

Example: `lFuncReturn = ReleaseComm()`

/FN0003/ SetDeviceType

Example:
`lFuncReturn = SetDeviceType(char* lpszDeviceName)`

Selects the device type.

lpszDeviceName: name of the device in file device.cfg.

Example: `lFuncReturn = SetDeviceType(”MSP430F1121”)`

/FN0004/ InitTarget

long int InitTarget(char* lpszDeviceName)

Initializes the JTAG access to the target device, detects the device type, and reports when the detected device does not match the parameter DeviceName passed.

lpszDeviceName: name of the device in file device.cfg.

Example: `lFuncReturn = InitTarget (”MSP430F1121”)`

/FN0005/ ReleaseTarget

long int ReleaseTarget(void)

This function releases the JTAG access to the target device. All JTAG signals from the serial programming adapter will be switched to high impedance. The device will start program execution if it is still connected to Vcc.

Example: `lFuncReturn = ReleaseTarget()`

/FN0006/ Erase

long int Erase(long int wStart, long int wLength, long int Flags)

This function erases flash memory (if available). The protection of areas can be disabled by setting the DISABLE_TI_MEM_PROTECT–Bit in Flags.

wStart: start address of the area to be erased. Allowed values : 0x0000 to 0xFFFE (see memory map of the corresponding device)

wLength: length of the area. Allowed values : 0x0000 to 0xFFFE (see memory map of the corresponding device)

Flags

DISABLE_TI_MEM_PROTECT (0x01)

If this bit is set, the memory protection settings in device.cfg are ignored.

Example:

```
lFuncReturn = Erase(long:0xF000, long:0x1000, long:1)
```

/FN0007/ EraseFile

long int EraseFile(char* lpszFileName, long int Flags, char* lpszProjectIni)

EraseFile() erases all addresses used in the specified file.

iFlags:

FILETYPE_AUTO (0x00) – Autodetection of file type (intel–hex or ti–text)

FILETYPE_TI_TXT (0x01) – File type is TI txt

FILETYPE_INTEL_HEX (0x02) – File type is intel hex

lpszProjectIni: name of the {project}.ini file, if protection settings from this file shall be used. If there should be no protection, replace lpszProjectIni by NULL.

Example:

```
lFuncReturn = EraseFile(”text.txt”, long:0, NULL)
```

/FN0008/ EraseCheck

long int EraseCheck(long int wStart, long int wLength)

Performs an erase check of an area of the target’s memory.

wStart: Start address of the memory area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wLength: Size of the area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

The function EraseCheck() simply uses PatternCheck() with 0xFFFF as pattern.

EraseCheck(long int wStart, long int wLength)

{

return PatternCheck(wStart, wLength, 0xFFFF);

}

Example:
```
lFuncReturn = EraseCheck(long:0xF000, long:0x1000)
```

/FN0009/ EraseCheckFile

long int EraseCheckFile(char* lpszFileName, long int iFileType)

This function checks if all memory addresses, which are in the file, are erased.

lpszFilName: Name of the file

iFileType:

FILETYPE_AUTO	(0x00)	– autodetection of file type (intel–hex or ti–text)
FILETYPE_TI_TXT	(0x01)	– file type is TI txt
FILETYPE_INTEL_HEX	(0x02)	– file type is intel hex

Example:
```
lFuncReturn = EraseCheckFile(”test.txt”, long:0)
```

/FN00010/ PatternCheck

long int PatternCheck(long int wStart, long int wLength, long int wPattern)

Checks a memory range with word pattern passed.

wStart: Start address of the memory area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wLength: Size of the area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wPattern: Word pattern for check

Example: `lFuncReturn = PatternCheck(long:0xF000, long:0x1000, long:0xFFFF)`

/FN00011/ VerifyData

long int VerifyData(long int wStart, long int wLength, void* lpData)

This function verifies the content of the device with the data stored at passed pointer to data.

wStart: Start address of memory area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wLength: Length of the memory area to be checked. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

lpData: Pointer to buffer with data bytes in it

Example:
lFuncReturn = VerifyData(long:0xF000, long:0x1000, void* lpData)

/FN00012/ VerifyFile

long int VerifyFile(char* lpszFileName, long int iFileType)

This function checks if the memory contents of the target device are equal to the file contents.

lpszFileName: Name of the file

iFileType

FILETYPE_AUTO	(0x00)	– autodetection of file type (intel–hex or ti–text)
FILETYPE_TI_TXT	(0x01)	– file type is TI txt
FILETYPE_INTEL_HEX	(0x02)	– file type is intel hex

Example: `lFuncReturn = VerifyFile(”test.txt”, long:0)`

/FN00013/ VerifyFileRange

long int VerifyFileRange(char* lpszFileName, long int iFileType, long int wStart, long int wLength)

This function evaluates if the memory contents of the target device are equal to the file contents in a passed range.

lpszFileName: Name of the file

iFileType:

FILETYPE_AUTO	(0x00)	– autodetection of file type (intel–hex or ti–text)
FILETYPE_TI_TXT	(0x01)	– file type is TI txt
FILETYPE_INTEL_HEX	(0x02)	– file type is intel hex

wStart: Start address of memory area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wLength: Length of the memory area to be checked. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

Example: `lFuncReturn = VerifyFileRange(”test.txt”, long:0,` long:0xF000, long:0x1000)

/FN0014/ ProgramData

long int ProgramData(long int wStart, long int wLength, void* lpData, long int Flags)

This function writes data into an MSP430 device. Protection of ranges of memory locations defined in the DEVICE.CFG file can be disabled by setting the DISABLE_TI_MEM_PROTECT–Bit in Flags.

wStart: Start address of the range which is to be erased. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wLength: Length of the range

Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device)

lpData: Pointer to the Data to be programmed

Flags: The bits in Flags control the operation of ProgramData()..

iFlags:

DISABLE_TI_MEMPROTECT (0x01)

PGM_WITH_ERASE (0x02)

PGM_WITH_ERASECHECK (0x04)

Example: `lFuncReturn = ProgramData(long:0xF000, long:0x1000, void* lpData,` long:7)

/FN0015/ ProgramFile

long int ProgramFile(char* lpszFileName, long int iFileType,

long int iFlags, char* lpszProjectIni)

This function writes data from the file to the MSP430 device. The protection of ranges of memory locations defined in the DEVICE.CFG file can be disabled by setting the DISABLE_TI_MEM_PROTECT–Bit in Flags.

lpszFileName: Name of the file to be written into the target

iFileType:

FILETYPE_AUTO (0x00) – autodetection of file type (intel–hex or ti–text)

FILETYPE_TI_TXT (0x01) – file type is TI txt

FILETYPE_INTEL_HEX (0x02) – file type is intel hex

iFlags:

DISABLE_TI_MEMPROTECT (0x01)

PGM_WITH_ERASE (0x02)

PGM_WITH_ERASECHECK (0x04)

lpszProjectIni

Name of the {project}.ini file, if protection settings from this file shall be used. If no protection is required, replace lpszProjectIni by NULL.

The added features do not need to be used – for ProgramFile according to older specification just call ProgramFile(FileName, FileType, 0, NULL); if no {project}.ini file or erase check, should be used just call:

Example: `lFuncReturn = ProgramFile(FileName, 0, 0, NULL);` // with autodetect file type

Note: If an erase or erase-check function reports an error, the function ProgramFile() is aborted before programming is started.

/FN0016/ BlowFuse

long int BlowFuse(long int Flags)

This function blows the security fuse of the target device.

Flags:

Bitmap to control the operation of BlowFuse().

NO_INHIBIT 1

If this bit is set in Flags, BlowFuse() suppresses the inhibition from *.ini files. This flag should always be set to ensure execution.

Example: `lFuncReturn = BlowFuse(long:1)`

/FN0017/ SetVcc

long int SetVcc(long int iVoltage)

This function sets the MSP_Vcc voltage of the programming adapter to the given value.

iVoltage: Vcc in millivolts. (3000 –> 3V)

The correct MSP430 device should be selected before using this function.

The voltage range is limited to the voltage range allowed for the selected MSP430 device.

Example: `lFuncReturn = SetVcc(Long:3000)`

/FN0018/ ReadOutData

long int ReadOutData(long int wStart, long int wLength, void* lpBuffer)

Reads out data from the device and writes it to the buffer passed.

wStart: Start address of the area to be read out. Allowed values : 0x0000 – 0xFFFE (see memory map for the corresponding device).

wLength: Length of the area. Allowed values : 0x0000 – 0xFFFE (see memory map for the corresponding device).

lpBuffer: Pointer points to a buffer that receives the data. The buffer must be large enough to hold the entire data; otherwise, a fatal error of the operating system may occur!

Example: `lFuncReturn = ReadOutData(long:0xF000, long:0x1000, void* lpBuffer)`

/FN0019/ ReadOutFile

long int ReadOutFile(long int wStart, long int wLength, char* lpszFileName, long int iFileType)

Read out data from the device and writes it to a file.

wStart: Start address of the area to be read out. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

wLength: Length of the area. Allowed values : 0x0000 – 0xFFFE (see memory map of the corresponding device).

lpszFileName: Name of the file to receive data. If the file does not exist, it will be created; If the file already exists, it will be overwritten.

iFileType:

FILETYPE_TI_TXT	(0x01)	– file type is TI txt
FILETYPE_INTEL_HEX	(0x02)	– file type is intel hex

Example: `lFuncReturn = ReadOutFile(long:0xF000, long:0x1000, ”test.out”, long:1)`

/FN0020/ Reset

long int Reset(long int Flags)

This function provides the reset functionality for the target.

Flags: Flags is a bitmap and determines the type of reset.

PUC	0x01
RST_NMI	0x02
WITH_RELEASE	0x04

Reset | PUC means that the Jtag sends the command to the MSP430.

Reset | RST_NMI performs a reset via the RST/NMI pin of the MSP430. The JTAG will also be reset.

If the WITH_RELEASE option is selected, the device will be released from the JTAG access after the reset.

Example: `lFuncReturn = Reset(long:5)`

/FN0022/ GetProgressInfo

long int GetProgressInfo(long int &iPercent, char* lpszStatus,

long int iMaxLen)

This function can be polled (typically approximately 10 times per second or less) by the program software, and the progress bar can be updated while an operation is in progress.

iPercent: The state of the progress bar (0 to 100%) is written into this integer.

lpszStatus: Points to a buffer to receive the status string.

iMaxLen: The size of the buffer.

/FN0023/ GetDeviceCfgInfo

long int GetDeviceCfgInfo(long int InfoCmd, long int InfoIdx, void* lpBuf)

InfoType:

DEVICE_COUNT (0x01)
GetDeviceCfgInfo returns number of devices in Device.cfg; InfoIdx and lpBuf are ignored.

SELECT_DEVICE (0x02)
Selects the given device for further commands (device number in InfoIndex, first device is number 0; lpBuf is ignored).

DEVICE_NAME (0x03)
Fills the name of the selected device into lpBuf; InfoIdx is ignored.

DEVICE_ID (0x04)
Fills the DeviceID into lpBuf, InfoIdx is ignored.

DEVICE_DEFAULTOPTIONS (0x05)
Fills the default options into lpBuf, InfoIdx is ignored.

DEVICE_MEMDEF_COUNT (0x06)
GetDeviceCfgInfo() returns the number of memory definitions for selected device; lpBuf and InfoIdx are ignored.

DEVICE_MEMDEF (0x07)
Fills the definition of a memory definition (index passed by InfoIdx) into lpBuf.

DEVICE_MEMPROTECT_COUNT (0x08)
GetDeviceCfgInfo() returns the number of memory-protection definitions for the selected device; lpBuf and InfoIdx are ignored.

DEVICE_MEMPROTECT (0x09)
Fills the definition of a memory protection definition (index passed by InfoIdx) into lpBuf.

DEVICE_VCC (0x0A)
GetDeviceCfgInfo() returns the Vcc setting for selected device in millivolts; lpBuf and InfoIdx are ignored

DEVICE_VPP (0x0B)
GetDeviceCfgInfo() returns the Vpp setting for selected device in millivolts; lpBuf and InfoIdx are ignored.

DEVICE_VFUSE (0x0C)
GetDeviceCfgInfo() returns the blow-fuse setting for the selected device; lpBuf and InfoIdx are ignored.

2.6.1 Return Values / Error Codes From the PRGS430.DLL

Status	Return Value	Comment
OK	0	
SUCCESS	–1	Operation ok
ERR_COMMUNICATION	–2	Communication error (SSP)
ERR_TARGET_NOT_CONNECTED	–3	No target connected
ERR_SPA430_NOT_CONNECTED	–4	No SPA430 connected
ERR_WRONG_JTAG_VERSION	–5	JTAG version above 3
ERR_PUC_FAILED	–6	PUC did not succeed
ERR_SPA430_SYNC_FAILED	–7	Could not sync SPA430
ERR_NO_SPA430	–8	Adapter is not SPA430
ERR_UNKNOWN_TARGET	–9	Target type unknown
ERR_WRONG_TARGET	–10	Target type does not match
ERR_NO_TARGET_SELECTED	–11	No target selected (missing SetDeviceType() call)
ERR_TARGET_FUSE_BLOWN	–12	No target access because of blown fuse
ERR_BLOW_FUSE_FAILED	–13	Blown-fuse command failed
ERR_VCC_NOT_SET	–14	No Vcc selected (missing SetVolt() call)
ERR_WRONG_VCC	–15	Vcc out of allowed range
ERR_WRONG_BAUDRATE	–16	Invalid baud rate
ERR_COMPORT	–17	Error accessing the communications port
ERR_DEVICE_CFG	–18	Device.cfg corrupted
ERR_GENERAL	–19	General error (should not occur!)
ERR_RANGE	–20	Wrong range specified
ERR_ARGUMENT	–21	Wrong argument
ERR_FILE_IO	–22	Error during file I/O
ERR_FILE_DETECT	–23	File type could not be detected
ERR_FILE_END	–24	Unexpected end of file
ERR_PROJECT_INI	–25	Error reading {project}.ini
ERR_VCC_BELOW_VCCMINPROG	–26	Vcc to low for selected function
ERR_FUSE_NOT_RELEASED	–27	Fuse not release for this device
STATUS_CONNECTSPA	1	Connecting to SPA430
STATUS_CONNECTTARGET	3	Connecting to target
STATUS_RELEASETARGET	5	Releasing target
STATUS_RELEASESPA	7	Releasing SPA430
STATUS_RESETTARGET	9	Resetting target
STATUS_ERASE	11	Erasing target
STATUS_ERASECHECK	13	Erase checking target
STATUS_PATTERNCHECK	15	Pattern checking target
STATUS_VERIFY	17	Verifying target
STATUS_PROGRAM	19	Programming target
STATUS_READOUT	21	Reading out target
STATUS_BLOWFUSE	23	Blowing fuse

Chapter 3

Hardware

This chapter describes the hardware for the MSP430 family of microcontrollers, including specifications, components of the programming adapters, and connection of the programming adapter to the MSP430 device families.

Topic **Page**

3.1 Specifications

The specifications for the MSP430 hardware are shown in Table 3–1.

Table 3–1. MSP430 Hardware Specifications

Temperature range	10°C–45°C
Humidity	40%–70%
Power supply	14 V–20 V, 200 mA minimum
Dimensions	150 mm (W) × 30 mm (H) × 95 mm (D)

3.2 Basic Hints

These basic hints are useful for programming MSP430 devices or MSP430 devices on printed wire boards (PWB).

- ❑ All VCC pins of an MSP430 device are tied together and connected to the most positive terminal of the supply.
- ❑ All VSS pins of an MSP430 device are tied together and connected to the most negative terminal of the supply.
- ❑ The interface should supply the MSP430 with proper conditions according to the device data sheet in terms of current, voltage levels, and timing conditions.
- ❑ MSP430x3xx, MSP430x14x family: Six interconnections are needed as minimum:
  TMS, TCK, TDI/VPP, TDO/TDI, VSS, and XOUT.
- ❑ MSP430x11x family: Seven interconnections are needed as minimum:
  TMS, TCK, TDI, TDO/TDI, VSS, Test/VPP, and XOUT.
- ❑ Short cables to interconnect the interface to the MSP430 device or PWB; less than 20 cm is recommended.
- ❑ Ensure low-impedance interconnections: Especially for the path of the programming and fuse blow voltage – TDI/VPP (MSP430x3xx family) or Test/VPP (MSP430x11x family, or TDI (MSP430x13x/14x family).
- ❑ When a device with a transparent window (MSP430E3xx family) is programmed, the window should be already covered with an opaque label while the device is programmed. Since ambient light contains the correct wavelength for erasure, keep the transparent window covered after the device is programmed.

3.3 Programming Adapter Target Connector Signals

The target connector signals for the programming adapter ensure communication between programming adapter and MSP430 devices and supply low energy to systems without extra supply sources.

Figure 3–1 and Figure 3–2 show the target connector signals for the programming adapter.

Figure 3–1. 25-Pin Sub-D at the Programming Adapter

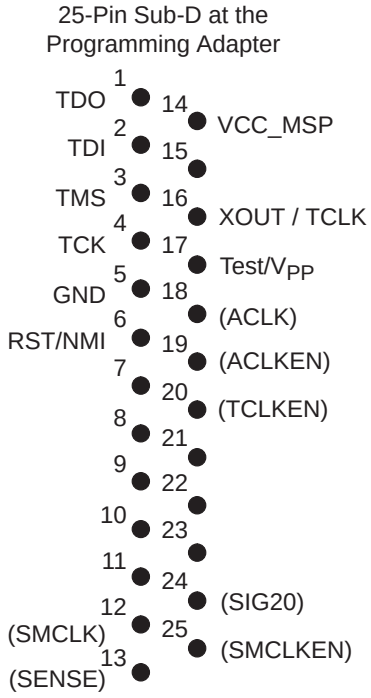


Figure 3–2. 14-Pin Connector at the End of the Interconnect Cable

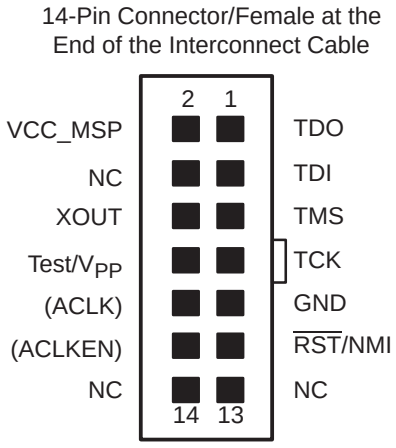


Table 3–2 lists the target connector signals and describes their requirement statuses and functions.

Table 3–2. Target Connector Signal Functions

Signal/Terminal Name	Required	Function/Comment
TMS	Mandatory	Test mode select functions according to IEE1149.1.
TCK	Mandatory	Test clock functions according to IEE1149.1.
TDI/VPP	Mandatory	Test data input functions according to IEE1149.1, but with additional programming voltage.
TDO/TDI	Mandatory	Test data output functions according to IEE1149.1, but additional data input is used when programming voltage is applied by TDI/VPP.
GND	Mandatory	GND is the most negative terminal.
VCC_MSP	Mandatory	Voltage source is used with MSP430 devices or PWBs. The voltage level is set through by software.
XOUT	Mandatory	Signal supplies the MSP430 system with clock signals.
$\overline{\text{RST}}$/NMI	Optional	If not connected, $\overline{\text{RST}}$/NMI pin must be held high
Test/VPP	Mandatory (depending on device)	Signal used with MSP430x11x devices to select pin or JTAG function or to apply VPP.

The output signal levels of the programming adapter are near GND or VCC_NSP.

- The $\overline{\text{RST}}$/NMI terminal of the device must be high; otherwise the access to the device via JTAG system may fail.
- The programming procedure (handling of the SW) is described in Chapters 1 and 2 of this manual.
- The connections from the MSP430 terminals must follow EMI rules; such as short lines and ground planes. If TMS line receives one negative pulse by EMI strike, the fuse current is activated (with fuse version 1.0). The fuse current flows from TDI(/VPP) pin to GND (or VSS).

Table 3–3. Programming Adapter Signal Levels

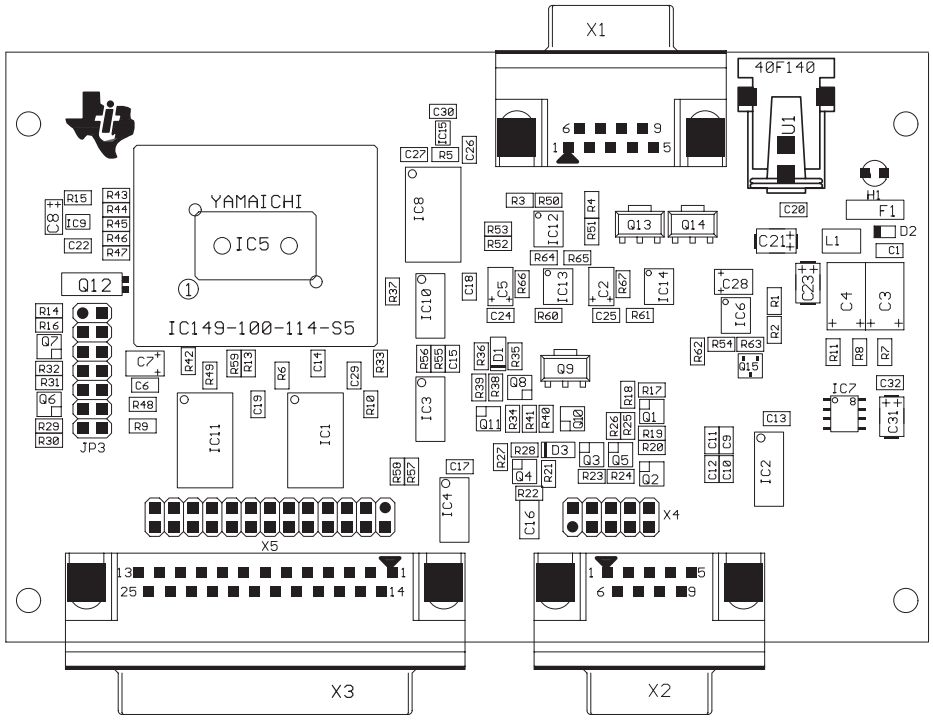
Signal/Pin	Signal/Pin Levels
TMS	VSS or VCC_MSP
TCK	VSS or VCC_MSP
TDI/VPP	VSS or VCC_MSP or VPP
TDO/TDI	VSS or VCC_MSP
XOUT	VSS or VCC_MSP
RST/NM	VSS or VCC_MSP
Test/VPP	VSS or VCC_MSP or VPP

3.4 MSP-PRGS430 Circuit Diagrams

The MSP-PRGS430 circuit diagrams are found in Appendix A.

3.5 Location of Components, MSP-PRGS430

Figure 3–3. MSP-PRGS430 Components

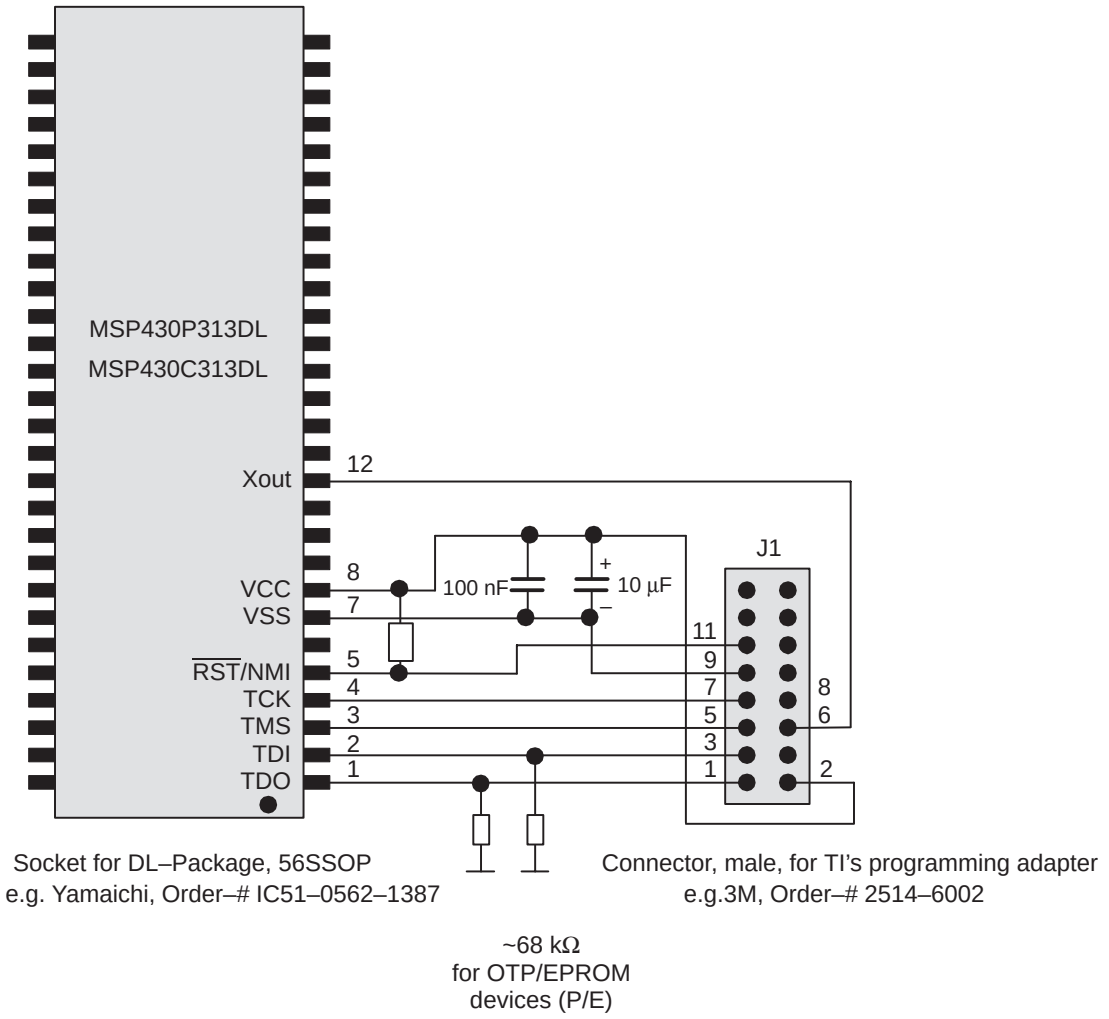


Note: Do not use J2 Pin 9 as RST/NMI pullup.

3.6 Interconnection of MSP-PRGS430 to MSP430C313DL/430P313SDL, MSP430C311SDL/P315SDL, or 'E313FZ

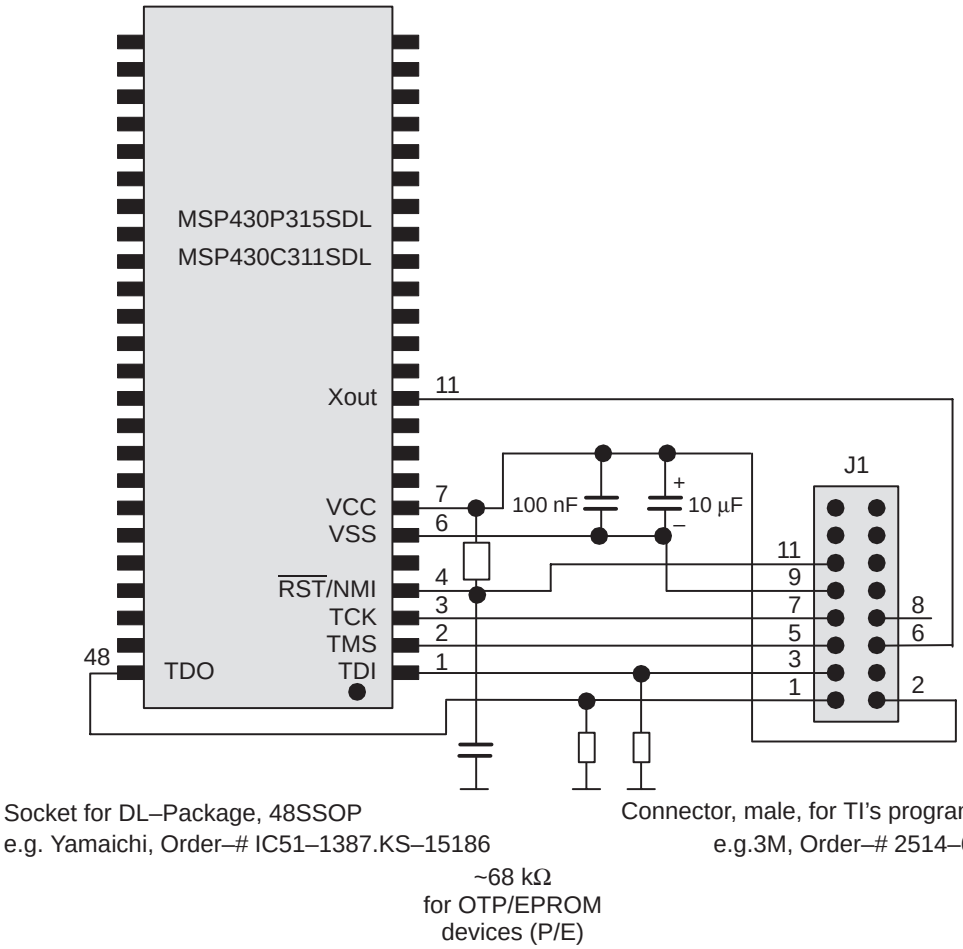
The circuit diagrams in Figure 3–5 show the connections required to program the MSP430P313DL device with programming adapter MSP-PRGS430 in a separate socket.

Figure 3–4. MSP-PRGS430 Used to Program the MSP430P313DL Device



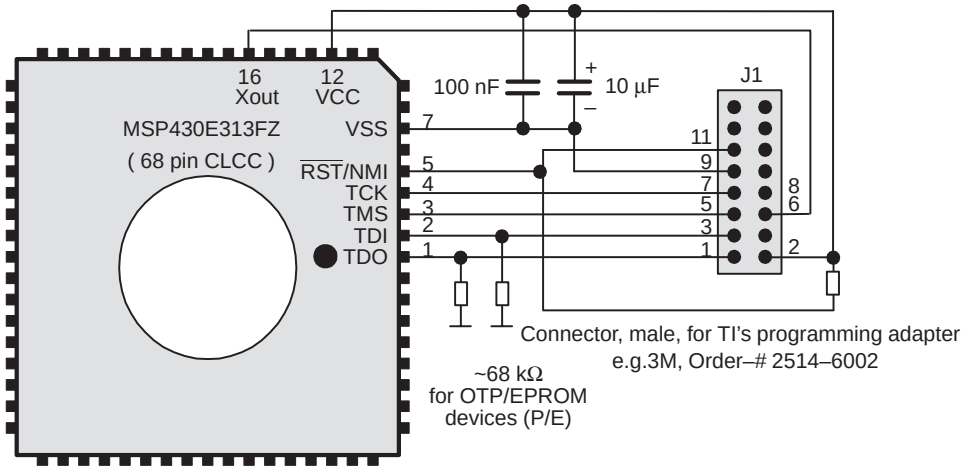
The $\overline{\text{RST}}$/NMI terminal on the MSP430 device has to be held high by an external resistor during access of the device through JTAG. In a noisy environment, consider using an additional capacitor from $\overline{\text{RST}}$/NMI to VSS.

Figure 3–5. MSP-PRGS430 Used to Program the MSP430P315SDL Device



The $\overline{\text{RST}}$/NMI terminal on the MSP430 device has to be held high by an external resistor during access of the device through JTAG. In a noisy environment, consider using an additional capacitor from $\overline{\text{RST}}$/NMI to VSS.

Figure 3–6. MSP-PRGS430 Used to Program the MSP430E313FZ Device



Socket for FZ–Package, 68CLCC
e.g. Yamaichi, Order–# IC51–0684–390–1

Note: The supply voltage is applied by TI's programming adapter. The MSP430 device is put into a socket without any additional application-specific components.

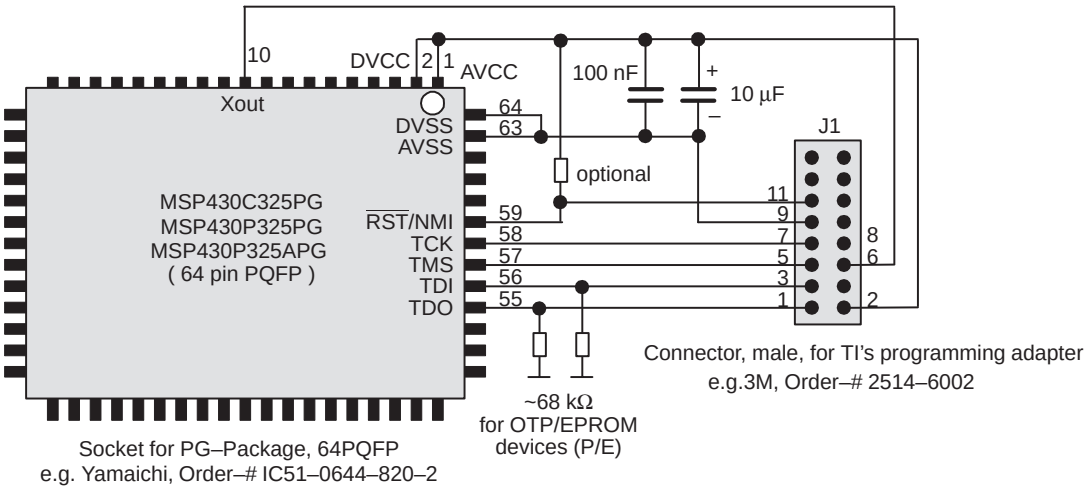
The $\overline{\text{RST}}$/NMI pin on the MSP430 device has to be held high by an external resistor during access of the device through JTAG. Any reset event disturbs the proper data sequences and produces unpredictable results. In a noisy environment, consider using an additional capacitor from $\overline{\text{RST}}$/NMI to VSS.

3.7 Interconnection of MSP-PRGS430 to MSP430C325PG, C325PM, MSP430P325PG, or ’P325PM

The circuit diagrams in Figure 3–19 show the connections required to program the MSP430C325PG, MSP430P325PG, or MSP430P325APG device with programming adapter PRGS430 in a separate socket.

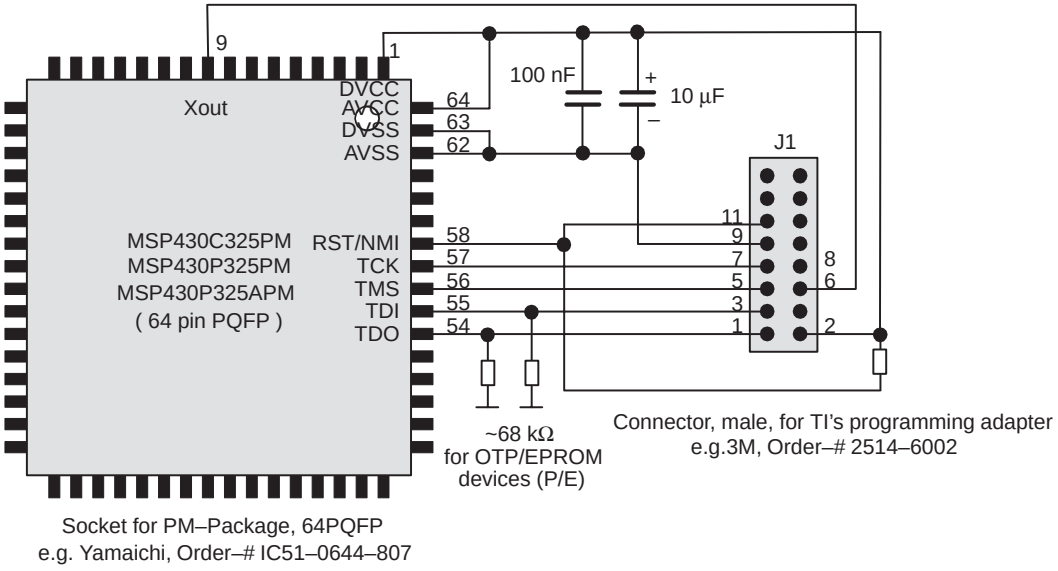
Ensure that both positive terminals AVCC and DVCC are connected. In addition, ensure that both negative terminals AVSS and DVSS are connected.

Figure 3–7. MSP-PRGS430 Used to Program the MSP430P325PG or MSP430P325APG Devices



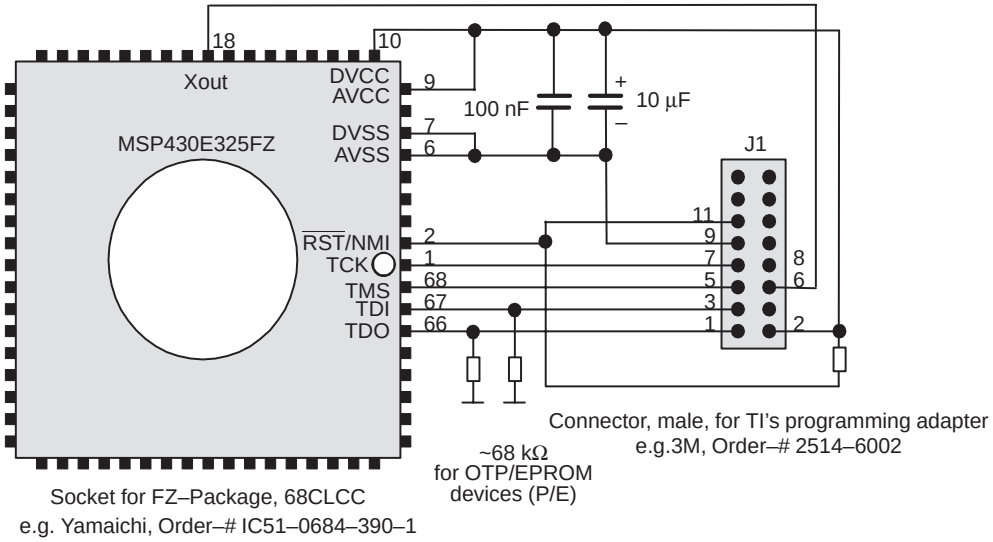
The $\overline{\text{RST}}$/NMI pin on the MSP430 device has to be held high by an external resistor during access of the device through JTAG. Any reset event disturbs the proper data sequences and produces unpredictable results. In a noisy environment, consider using an additional capacitor from $\overline{\text{RST}}$/NMI to VSS.

Figure 3–8. MSP-PRGS430 Used to Program the MSP430P325PM or MSP430P325APM Devices



The $\overline{\text{RST}}$/NMI pin on the MSP430 device has to be held high by an external resistor during access of the device through JTAG. Any reset event disturbs the proper data sequences and produces unpredictable results. In a noisy environment, consider using an additional capacitor from $\overline{\text{RST}}$/NMI to VSS.

Figure 3–9. MSP-PRGS430 Used to Program the MSP430E325FZ Device



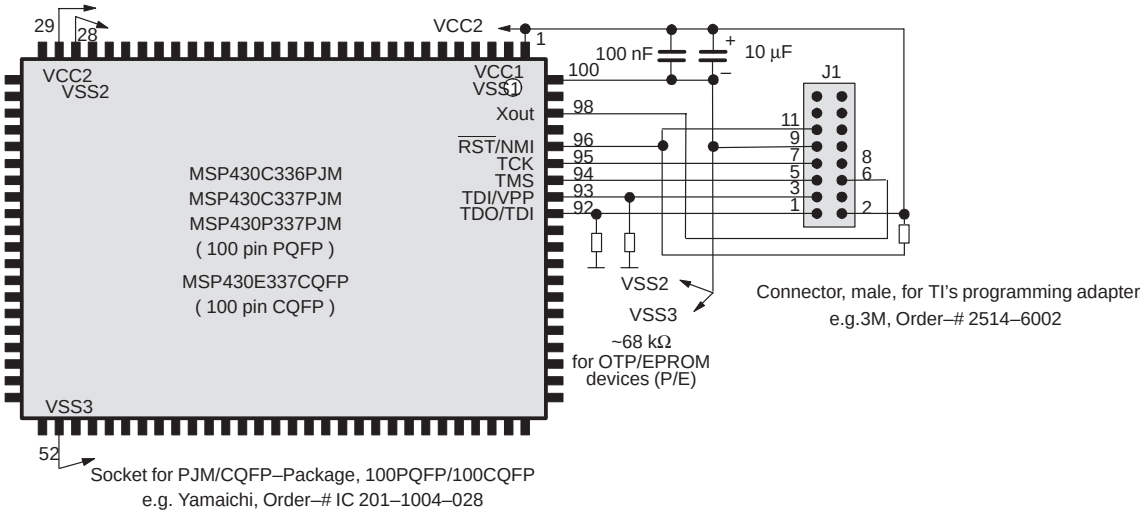
The $\overline{\text{RST}}$/NMI pin on the MSP430 device has to be held high by an external resistor during access of the device through JTAG. Any reset event disturbs the proper data sequences and produces unpredictable results. In a noisy environment, consider using an additional capacitor from $\overline{\text{RST}}$/NMI to VSS.

3.8 Interconnection of MSP-PRGS430 to MSP430C336PJM/337PJM or MSP430E337CQFP

The circuit diagram in Figure 3–10 show the connections required to program the MSP430C336PJM, MSP430P337PJM, or MSP430E337CQFP devices with programming adapter PRGS430 in a separate socket. Since the device is not connected to a power supply in this configuration, the necessary supply comes from the PRGS430.

Ensure that the two positive terminals, VCC1 and VCC2, as well as the three negative terminals, VSS1, VSS2, and VSS3, are connected.

Figure 3–10. MSP-PRGS430 Used to Program the MSP430x33xPJM or the MSP430E337CQFP Devices



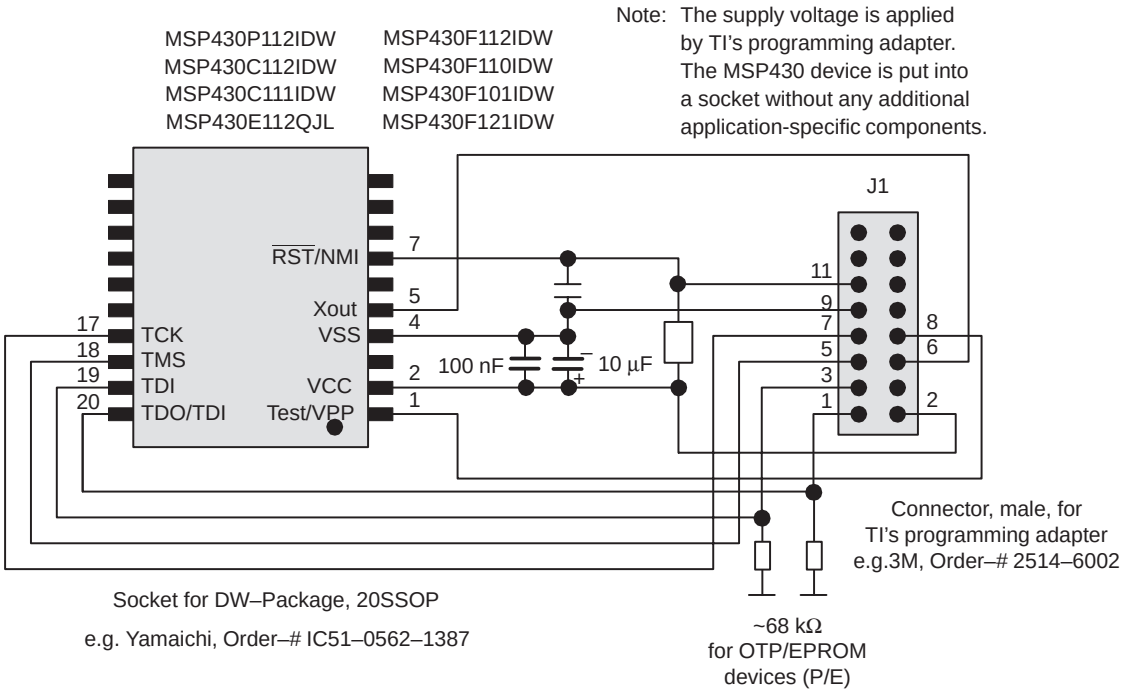
The $\overline{\text{RST}}$/NMI pin on the MSP430 device must be held high by an external resistor during access of the device through JTAG. Any reset event disturbs the proper data sequences and produces unpredictable results.

3.9 Interconnection of MSP-PRGS430 to MSP430C111DW, MSP430C112DW, MSP430P112DW, or MSP430E112JL

The circuit diagram in Figure 3–11 shows the connections required to program with the programming adapter PRGS430 in a separate socket. Special attention must be given to the design for the four JTAG pins, TDO/TDI, TDI, TMS, and TCK, since they are shared between the application's hardware and the programming adapter. The programming adapter should be able to drive the device correctly, but the application should continue working properly.

The $\overline{\text{RST}}$/NMI pin on the MSP430 device must be held high by an external resistor. In a noisy environment, consider using an additional capacitor from RST/NMI to VSS.

Figure 3–11. MSP-PRGS430 Used to Program the MSP430x11xIDW or the MSP430E112QJL Devices



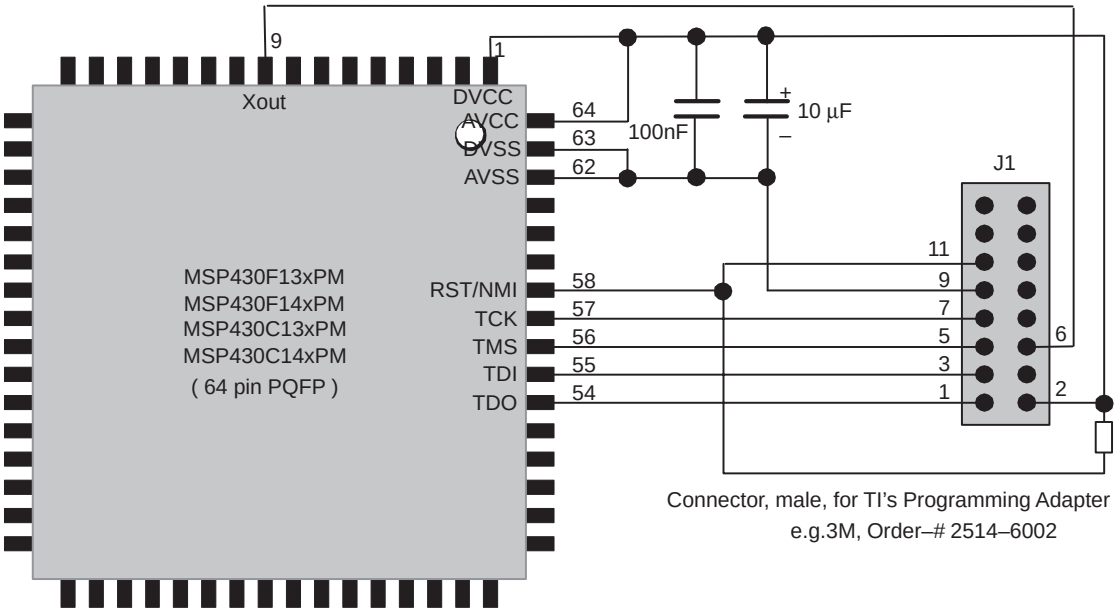
3.10 Interconnection of MSP-PRGS430 to the MSP430F13xPM, MSP430C13xPM, MSP430F14xPM, or MSP430C14xPM

The following circuit diagram shows the connections required to program the MSP430F13xPM, MSP430C13xPM, MSP430F14xPM or MSP430C14xPM devices with TI's Programming Adapter MSP-PRGS430 e.g. in a separate socket.

The $\overline{\text{RST}}$/NMI pin on the MSP430 device is held high by the resistor in the PRG430B or by an external resistor when MSP-PRGS430 is used. In a noisy environment an additional capacitor from RST/NMI to VSS should be considered. Ensure that both positive terminals AVCC and DVCC as well as both negative terminals AVSS and DVSS are connected together.

Figure 3–12. Interconnection of MSP-PRGS430 to MSP430x13xPM and MSP430x14xPM

Interconnection of MSP-PRGS430 to MSP430x13xPM, MSP430x14xPM



Note: The supply voltage is applied by TI's programming adapter.

Chapter 4

EPROM Programming

This chapter describes the MSP430 EPROM module. The EPROM module is erasable with ultraviolet light and electrically programmable. Devices with an EPROM module are offered in a windowed package for multiple programming and in an OTP package for one-time programmable devices.

Topic **Page**

4.1 EPROM Operation

The CPU acquires data and instructions from the EPROM. When the programming voltage is applied to the TDI/VPP terminal, the CPU can also write to the EPROM module. The process of reading the EPROM is identical to the process of reading from other internal peripheral modules. Both programming and reading can occur on byte or word boundaries.

4.1.1 Erasure

The entire EPROM may be erased before programming begins. Erase the EPROM module by exposing the transparent window to ultraviolet light.

Note: EPROM Exposed to Ambient Light (1)

Since normal ambient light contains the correct wavelength for erasure, cover the transparent window with an opaque label when programming a device. Do not remove the table until it has to be erased. Any useful data in the EPROM module must be reprogrammed after exposure to ultraviolet light.

The data in the EPROM module can be programmed serially through the integrated JTAG feature, or through software included as a part of the application software. The JTAG features an internal mechanism for security, provided by the implemented fuse. Once the security fuse is activated, the device cannot be accessed through the JTAG functions. The JTAG is permanently operating in the bypass mode.

Refer to the appropriate data sheet for more information on the fuse implementation.

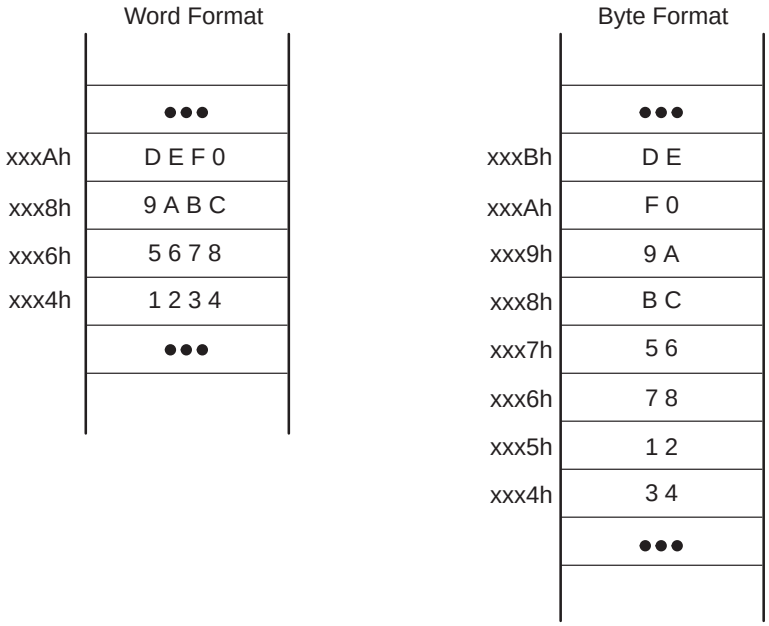
4.1.2 Programming Methods

The application must provide an external voltage supply to the TDI/VPP terminal to provide the necessary voltage and current for programming. The minimum programming time is noted in the electrical characteristics of the device data sheets.

The EPROM control register EPCTL controls the EPROM programming, once the external voltage is supplied. The erase state is a 1. When EPROM bits are programmed, they are read as 0.

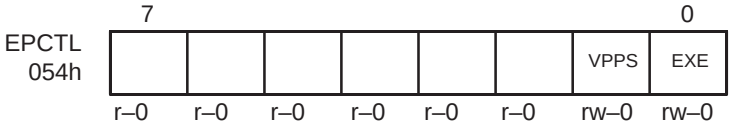
The programming of the EPROM module can be done for single bytes, words, blocks of individual length, or the entire module. All bits that have a final level of 0 must be erased before the EPROM module is programmed. The programming can be done on single devices or even in-system. The supply voltage should be in the range required by the device data sheet but at least the maximum supply voltage of the target application. The levels on the JTAG terminals are defined in the device data sheet, and are usually CMOS levels.

Example 4–1.MSP430 On-Chip Program Memory Format



4.1.3 EPROM Control Register EPCTL

Figure 4–1. EPROM Control Register EPCTL



For bit 0, the executable bit EXE initiates and ends the programming to the EPROM module. The external voltage must be supplied to the TDI/VPP or Test/VPP before the EXE bit is set. The timing conditions are noted in the data sheets.

For bit 1, when the VPPS bit is set, the external programming voltage is connected to the EPROM module. The VPPS bit must be set before the EXE bit is set. It can be reset together with the EXE bit. The VPPS bit must not be cleared between programming operations.

Note:

Ensure that no VPP is applied to the programming voltage pin (TDI/VPP or Test/VPP) when the software in the device is executed or when the JTAG is not fully controlled. Otherwise, an undesired write operation may occur.

4.1.4 EPROM Protect

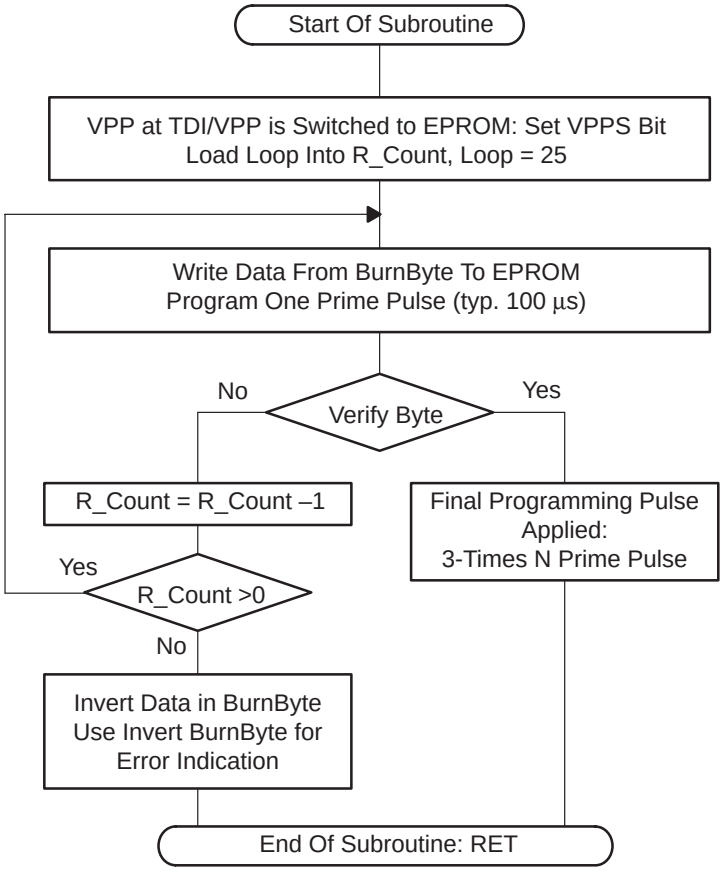
The EPROM access through the serial test and programming interface JTAG can be inhibited when the security fuse is activated. The security fuse is activated by serial instructions shifted into the JTAG. Activating the fuse is not reversible and any access to the internal system is disrupted. The by-pass function described in the standard IEEE 1149.1 is active.

4.2 FAST Programming Algorithm

The FAST programming cycle is normally used to program the data into the EPROM. A programmed logical 0 can be erased only by ultraviolet light.

Fast programming uses two types of pulses: prime and final. The length of the prime pulse is typically 100 μs (see the latest data sheet). After each prime pulse, the programmed data are verified. If the verification fails 25 times, the programming operation was false. If correct data are read, the final programming pulse is applied. The final programming pulse is 3 times the number of prime pulses applied.

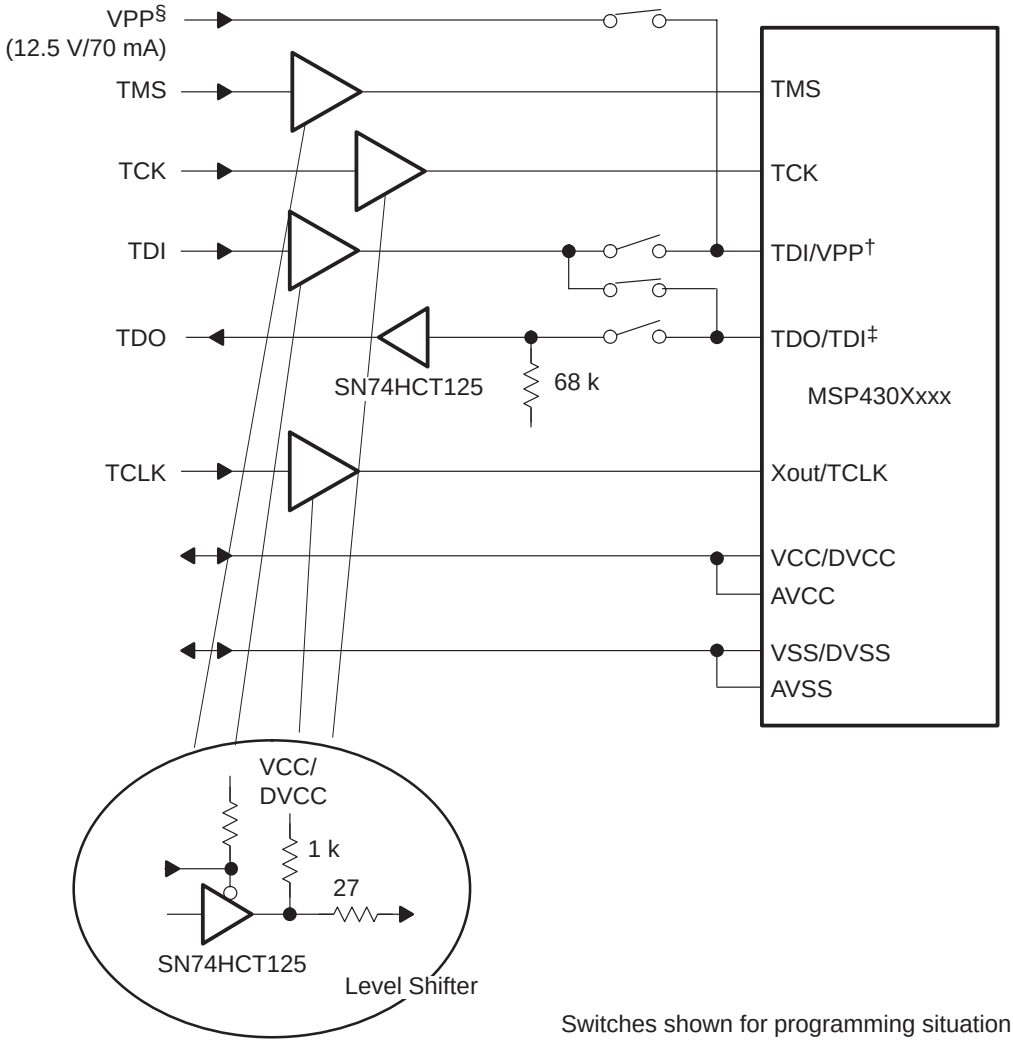
Example 4–2. Fast Programming Subroutine



4.3 Programming an EPROM Module Through a Serial Data Link Using the JTAG Feature

The hardware interconnection of the JTAG terminals is established through four separate terminals, plus the ground or VSS reference level. The JTAG terminals are TMS, TCK, TDI(/VPP), and TDO(/TDI).

Figure 4–2. EPROM Programming With Serial Data Link



† TDI in standard mode, VPP input during programming
‡ TDO in standard mode, data input TDI during programming
§ See electrical characteristics in the latest data sheet

4.4 Programming an EPROM Module With Controller's Software

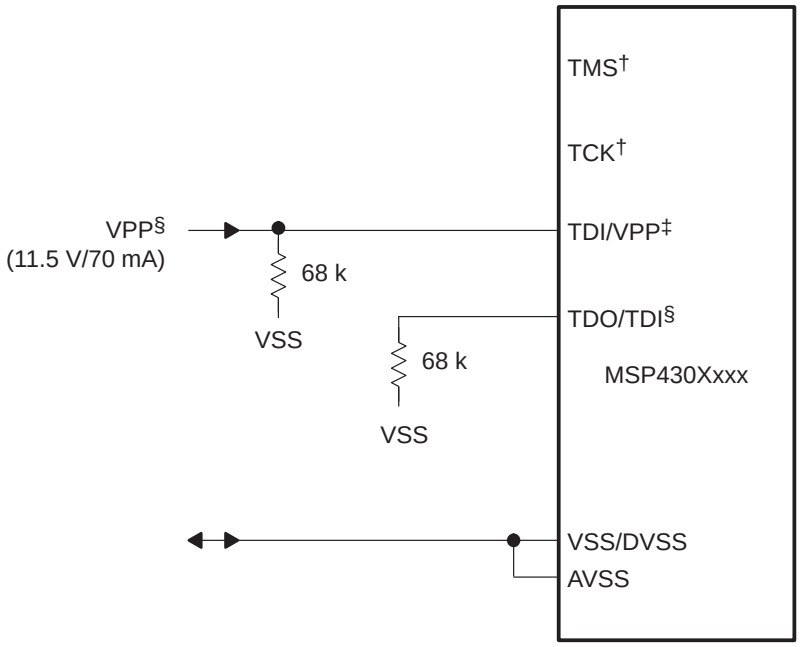
The procedure for programming an EPROM module is as follows:

1) Connect the required supply to the TDI/VPP terminal.

2) Run the proper software algorithm.

The software algorithm that controls the EPROM programming cycle cannot run in the same EPROM module to which the data are being written. It is impossible to read instructions from the EPROM and write data to it at the same time. The software needs to run from another memory such as a ROM module, a RAM module, or another EPROM module.

Figure 4–3. EPROM Programming With Controller's Software

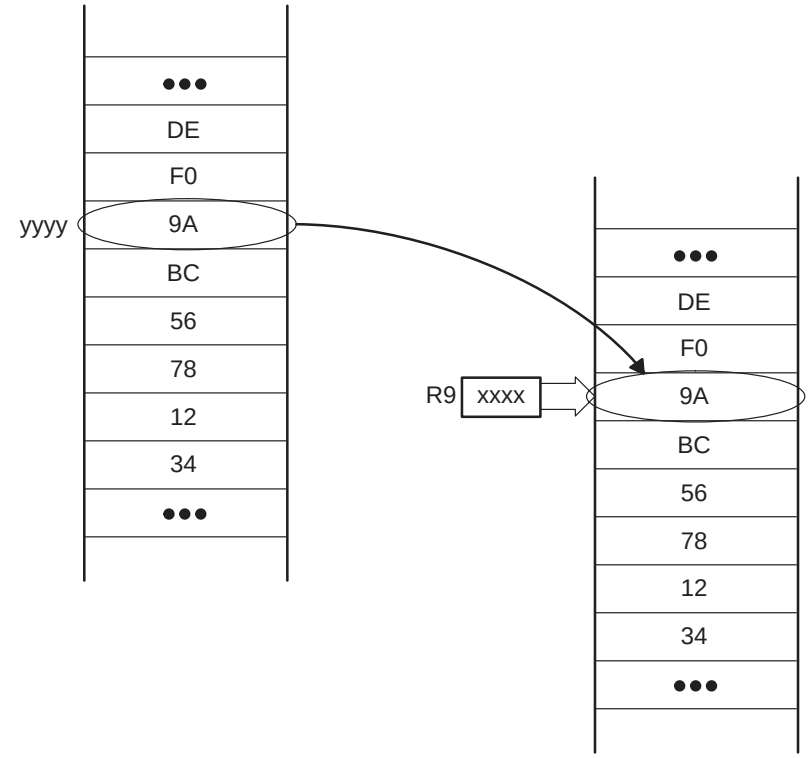


† Internally a pullup resistor is connected to TMS and TCK
‡ ROM devices of MSP430 have an internal pullup resistor at pin TDI/VPP. MSP430Pxxx or MSP430Exxx have no internal pullup resistor. They should be terminated according to the device data sheet.
§ The TDO/TDI pin should be terminated according to the device data sheet.

4.4.1 Example

The software example writes one byte into the EPROM with the fast programming algorithm. The code is written position-independent, and will have been loaded to the RAM before it is used. The programming algorithm runs during the programming sequence in the RAM, thus avoiding conflict when the EPROM is written. The data (byte) that should be written is located in the RAM address BurnByte. The target address of the EPROM module is held in the register pointer defined with the set directive. The timing is adjusted to a cycle time of 1μs. When another cycle time/processor frequency is selected, the software should be adjusted according to the operating conditions.

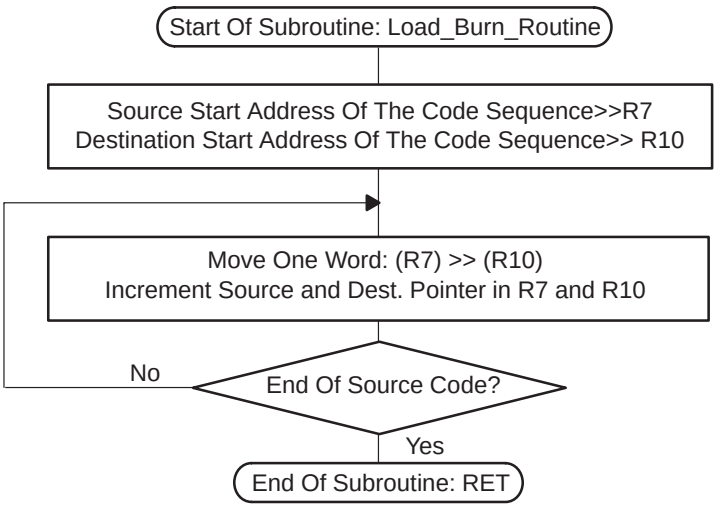
Example 4–3. Programming EPROM Module With Controller's Software



Example: Write data in yyyy into location xxxx
BumByte = (yyyy) = (9Ah)
R9 = xxxx

The target EPROM module cannot execute the programming code sequence while the data are being written into it. In the example, a subroutine moves the programming code sequence into another memory, for example, into the RAM.

Example 4–4. Subroutine



4.5 Code

```
;-------------------------------------------------------------
; Definitions used in Subroutine :
; Move programming code sequence into RAM (load_burn_routine)
; Burn a byte into the EPROM area         (Burn_EPROM)
;-------------------------------------------------------------

EPCTL     .set   054h  ; EPROM Control Register
VPPS      .set   2     ; Program Voltage bit
EXE       .set   1     ; Execution bit
BurnByte  .set   0220h ; address of data to be written
Burn_orig .set   0222h ; Start address of burn
                       ; program in the RAM
loops     .set   25
r_timer   .set   r8    ; 1us = 1 cycle
pointer   .set   r9    ; pointer to the EPROM address
                       ; r9 is saved in the main routine
                       ; before subroutine call is executed
r_count   .set   r10
lp        .set   3     ; dec r_timer   : 1 cycle  : loop_t100
                       ; jnz           : 2 cycles : loop_t100
ov        .set   2     ; mov #(100-ov)/lp,r_timer : 2 cycles
```

; Load EPROM programming sequence to another location e.g. RAM, Subroutine

```
;--- The address of Burn_EPROM (start of burn EPROM code) and
;--- the address of Burn_end (end of burn EPROM code) and
;--- the start address of the location of the destination
;--- code area (RAM_Burn_EPROM) are known at assembly/linking time

RAM_Burn_EPROM  .set    Burn_orig
load_burn_routine
   push     r9
   push     r10
   mov      #Burn_EPROM,R9        ; load pointer source
   mov      #RAM_Burn_EPROM,R10   ; load pointer dest.
load_burn1
   mov      @R9,0(R10)            ; move a word
   incd     R10                   ; dest.  pointer + 2
   incd     R9                    ; source pointer + 2
   cmp      #Burn_end,R9          ; compare to end_of_table
   jne      load_burn1
   pop      r9
   pop      r10
   ret
```

; Program one byte into EPROM, Subroutine

```
;--- Burn subroutine: position independent code is needed
;    since in this examples it is shifted to RAM  >> only
;--  relative addressing, relative jump instructions, is used!
;--- The timing is correct due to 1us per cycle

Burn_EPROM
      dint                          ; ensure correct burn timing
      mov.b #VPPS,&EPCTL            ; VPPS on
      push  r_timer                 ; save registers
      push  r_count                 ; programming subroutine
      mov   #loops,r_count          ; 2 cycles = 2 us
Repeat_Burn
      mov.b &BurnByte,0(pointer)    ; write to data to EPROM
```

```
                                ; 6 cycles = 6 us
      bis.b #EXE,&EPCTL         ; EXE on
                                ; 4 cycles = 4 us
                                ; total cycles VPPon to EXE
                                ; 12 cycles = 12 us (min.)
      mov   #(100-ov)/lp,r_timer  ;:programming pulse of 100us
wait_100                        ;:starts, actual time  102us
      dec   r_timer             ;:
      jnz   wait_100            ;:
      bic.b #EXE,&EPCTL         ;:EXE / prog. pulse off

      mov   #4,r_timer          ;:wait min. 10 us
wait_10                         ;:before verifying
      dec   r_timer             ;:programmed EPROM
      jnz   wait_10             ;:location, actual 13+ us

      cmp.b &BurnByte,0(pointer)  ; verify data = burned data
      jne   Burn_EPROM_bad        ; data ‡ burned data > jump

; Continue here when data correctly burned into EPROM location
      mov.b &BurnByte,0(pointer)  ; write to EPROM again
      bis.b #EXE,&EPCTL         ; EXE on
      add   #(0ffffh-loops+1),r_count
                                ; Number of loops for
                                ; successful programming
final_puls
      mov   #(300-ov)/lp,r_timer  ;:programming pulse of
wait_300                        ;:3*100us*N starts
      dec   r_timer             ;:
      jnz   wait_300            ;:
      inc   r_count             ;:
      jn    final_puls          ;:
      clr.b &EPCTL              ;:EXE off / VPPS off
      jmp   Burn_EPROM_end
Burn_EPROM_bad
      dec   r_count             ; not ok : decrement
                                ; loop counter
      jnz   Repeat_Burn         ; loop not ended : do
                                ; another trial
      inv.b &BurnByte           ; return the inverted data
                                ; to flag
                                ; failing the programming
                                ; attempt the EPROM address
                                ; is unchanged
                                ;
Burn_EPROM_end
         pop   r_count
         pop   r_timer
         eint
         ret
Burn_end
```

Chapter 5

Flash Memory

This chapter describes the MSP430 flash memory module. The flash memory module is electrically erasable and programmable. Devices with a flash memory module are multiple-time programmable devices (MTP). They can be erased and programmed off-board, or in a system via the MSP430's JTAG peripheral module or via the processor's resources.

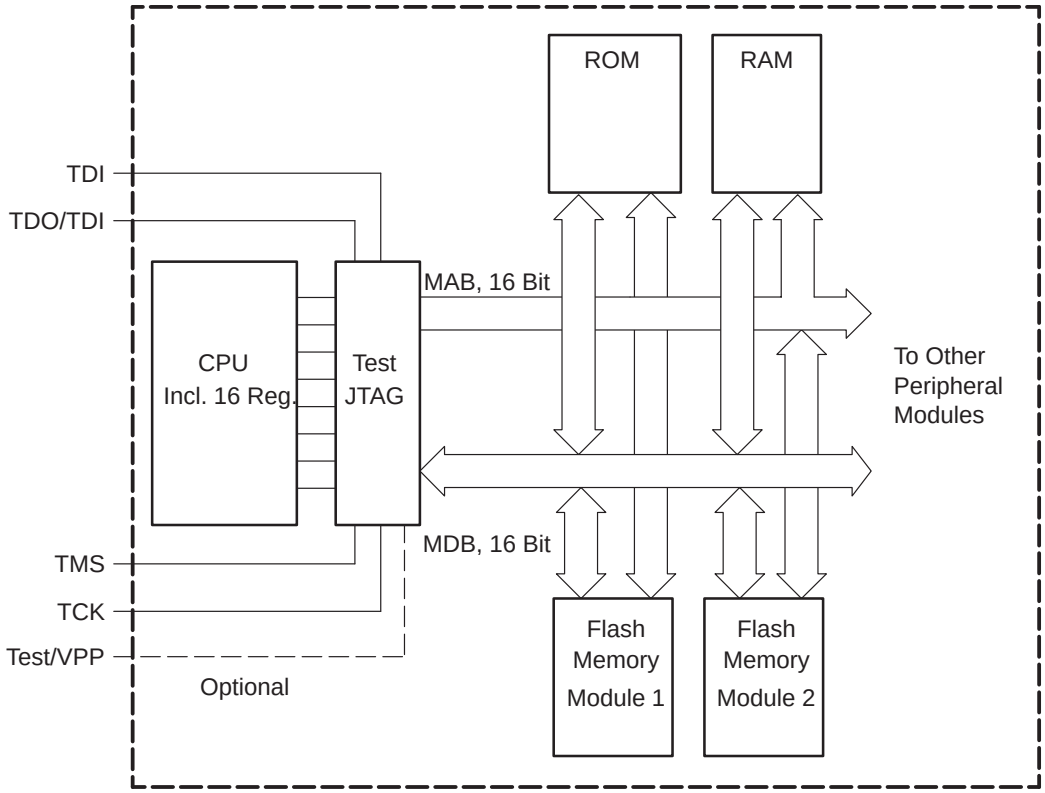
Software running on an MSP430 device can erase and program the flash memory module. This active software may run in RAM, in ROM, or in the flash memory. The flash memory may be a different memory module or the same memory module. The active software may not be in a memory location which is actively erased.

Topic **Page**

5.1 Flash Memory Organization

The flash memory may have one or more modules of different sizes as shown in Figure 5–1. A module is a physical memory unit that operates independent from other modules. In an MSP430 configuration with more than one flash memory module, all modules are located in one linear-address range.

Figure 5–1. Interconnection of Flash Memory Module(s)



Independent modules, such as Module1 and Module2, are intended to execute software code from one module while simultaneously programming or erasing another module.

Note: Flash Memory Module(s) in MSP430 Devices

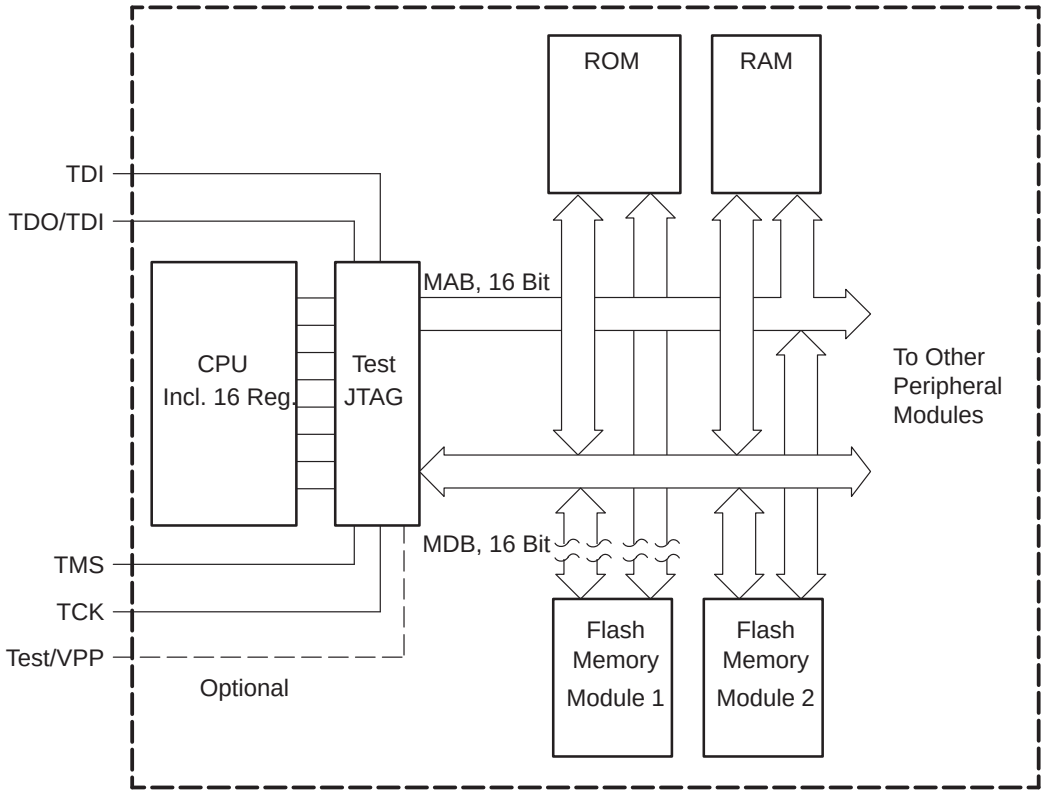
Different devices may have one or more flash memory modules.

If the active software and the target programming location are in the same flash memory module, the program execution is halted (flag BUSY=1) until the programming cycle is completed (flag BUSY=0). Then it proceeds with the next instruction. The active software may also erase segments of the flash memory module. The user should be careful not to erase memory locations that are necessary to execute the software correctly.

A flash memory module, being programmed or erased, can not be accessed.

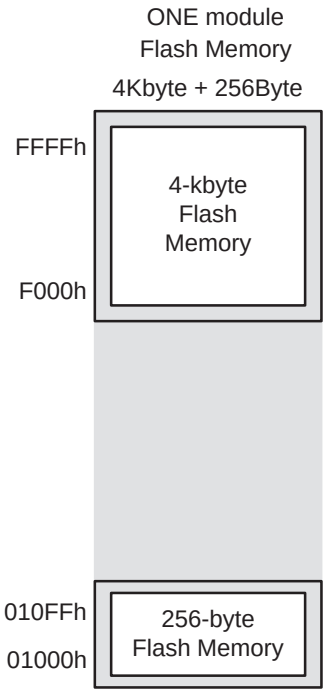
Figure 5–2 shows the flash memory Module1 in program or erase operation. During this operation the module is disconnected from the memory address bus and memory data bus. When a second module (here Module2) is implemented, program code in this module can be executed while Module1 is disconnected.

Figure 5–2. Flash Memory Module1 Disabled, Module2 Can Execute Code Simultaneously



One MSP430 flash memory module will have, in addition to its code segments, extra flash memory called *information memory.* The implementation is shown in Figure 17–3.

Figure 5–3. Flash Memory Module Example



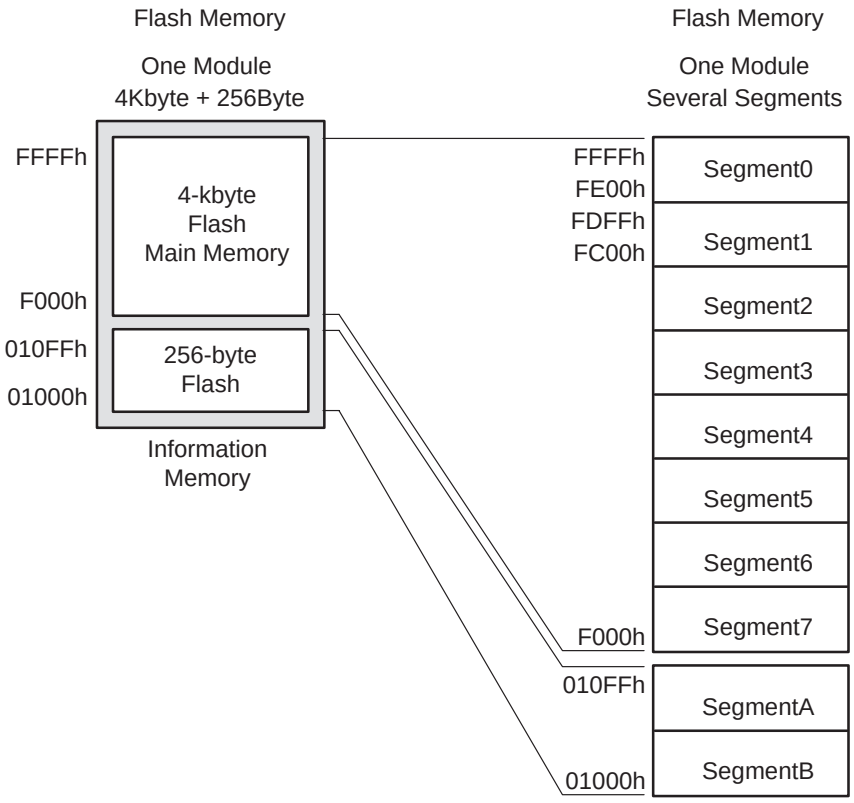
A module has several segments. The information memory has two segments of 128 bytes each. In the example in Figure 5–4, the 4-kB module has eight segments of 512 bytes (Segment0 to Segment7), and two 128-byte segments (SegmentA and SegmentB). Segment0 to Segment7 can be erased individually or as a group. SegmentA and SegmentB can be erased individually or as a group with segments 0 to 7.

The segment structure is described in the device's data sheet. The information memory can be located directly below the main memory's address, or at a different address but will be in the same module.

Note:

Flash memory modules may have different numbers of segments. Segment are numbered from 0 up to n, e.g., segment 0 to segment n.

Figure 5–4. Segments in Flash Memory Module, 4K-Byte Example



5.1.1 Why Is a Flash Memory Module Divided Into Several Segments?

Once a bit in flash memory has been programmed, it cannot be erased without erasing a whole segment. For this reason, the MSP430 flash memory modules have been heavily segmented to allow erasing and reprogramming of smaller memory segments.

5.2 Flash Memory Data Structure and Operation

The flash memory can be read and written (programmed) in bytes or words. Bits can be written as 0s once between erase cycles. The read access does not differ from access to masked ROM or RAM. Flash memory has restrictions in write operation:

- The default (erased) level for all bits is 1. Bits that are not programmed to 0s can be programmed to 0s at any time.
- The smallest memory portion to be erased is a segment. No single byte or word erase is possible.
- Access to a flash memory module is only possible when the module is not in a write or erase operation. For example, program code can not be executed in a module while it is processing a write or erase operation. The access limitation has no critical impact on program execution, but an access violation can be flagged in some situations (see flash memory register section in this chapter).

5.2.1 Flash Memory Basic Functions

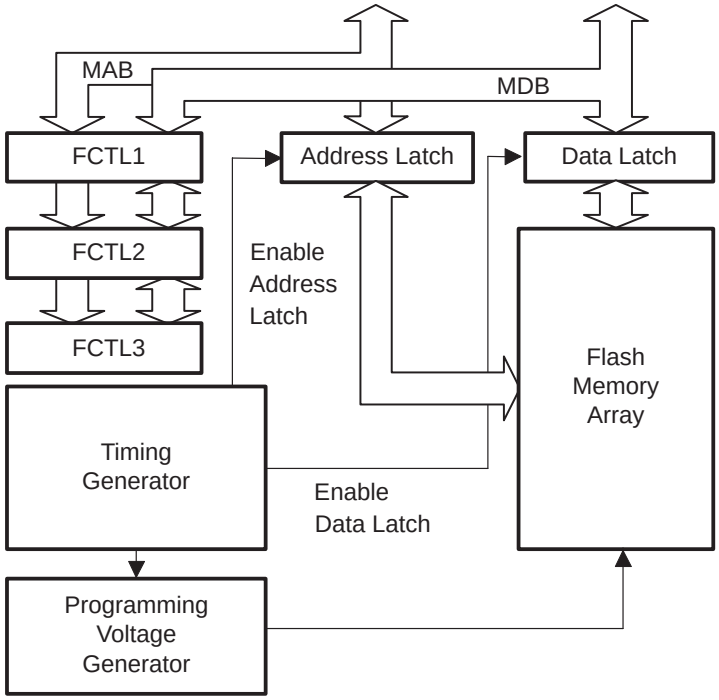
The basic functions of flash memory are to:

- Supply program code and data during program execution
- Erase, under software or JTAG control, parts of a module (one segment), multiple segments, or an entire module.
- Write data to a memory location under software or JTAG control. A double-speed programming sequence is implemented within a 64-byte section of the address range xx00h to xx3fh.

5.2.2 Flash Memory Block Diagram

The flash memory module has a minimum of three control registers, a timing generator, a voltage generator to supply program and erase voltages, and the flash memory itself. Data and address are latched when execution of a write (program) or erase operation is in progress.

Figure 5–5. Flash Memory Module Block Diagram



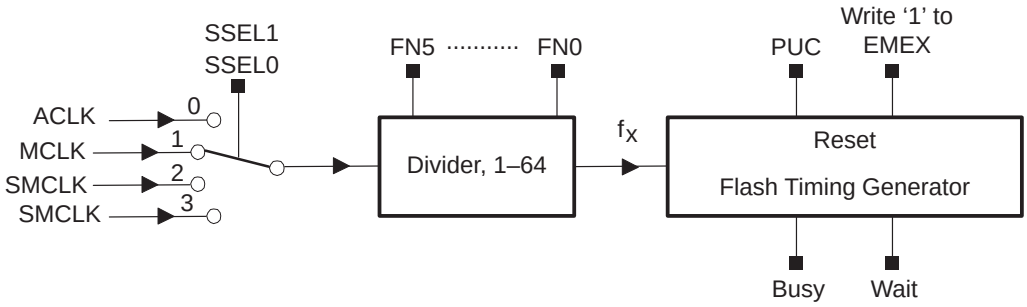
5.2.3 Flash Memory, Basic Operation

The flash memory module works in read mode most of the time, the address and data latch are transparent, and the timing generator and programming voltage generator are off. The flash memory module changes its mode of operation when data is written (programmed) to the module, or when the flash memory, or parts of it, are erased. In these situations, flash control registers FCTL1, FCTL2, and FCTL3 need to be set up properly to ensure correct write

or erase operation. Once these registers are set up and write or erase is started, the timing generator controls the entire operation and applies all signals internally. If the BUSY control signal is set, it indicates that the timing generator is active and a write or erase cycle is active. The segment write mode also uses a second control bit WAIT. There are three basic parts to a write or erase cycle: preparation of program/erase voltage, control timing for the program or erase operation, and the switch-off sequence of the program/erase voltage. Once a write or erase function is started, the software should not access the flash memory until the BUSY signal indicates, with *0*, that it can be accessed again. In critical situations where flash programming or erase should be immediately stopped, the *emergency exit* bit EMEX can be set. The current operation may be incomplete or the result may be incorrect.

Two different clock sources (ACLK, MCLK, or SMCLK) can be selected to clock the timing generator. The connected clock sources applied to the timing generator may vary with the device, see data sheet for details. The clock source selected should be active from the beginning of write or erase until the operation is fully completed.

Figure 5–6. Block Diagram of the Timing Generator in the Flash Memory Module



The selected clock source should be divided to meet the frequency requirement f_x of the flash timing generator.

If the clock signals are not available throughout the duration of the write or erase operation, or their frequencies change drastically, the result of the write or erase may be marginal, or the flash memory module may be stressed above the limits of reliable operation.

Table 5–1 shows all useful combinations of control bits for proper write and erase operation:

Table 5–1. Control Bits for Write or Erase Operation

FUNCTION PERFORMED	SEGWRT	WRT	Meras	Erase	BUSY	WAIT	Lock
Write word or byte	0	1	0	0	0	0	0
Write word or byte in same segment, segment write mode	1	1	0	0	0	1	0
Erase one segment by writing to any address in the target segment (0 to n or A or B)	0	0	0	1	0	0	0
Erase all segments (0 to n) but not the information memory (SegmentA and SegmentB)	0	0	1	0	0	0	0
Erase all segments (0 to n and A and B) by writing to any address in the flash memory module	0	0	1	1	0	0	0

Note: A write to flash memory performed with any other combination of bits SEGWRT, WRT, Meras, Eras, BUSY, WAIT, and Lock will result in an access violation. ACCVIFG is set and an NMI is requested if ACCVIE=1.

5.2.4 Flash Memory Status During Code Execution

The flash memory module delivers data for code execution in the same manner as any masked ROM or RAM. The flash memory module should be in read mode, with no write (programming) or erase operation active. By default, power-on reset (POR) puts the flash memory into read mode. No control bits need to be defined in the flash memory control registers after POR for code execution.

5.2.5 Flash Memory Status During Erase

The default bit level of the flash memory is 1. Any successful erase sets all bits of a segment or a block to this default level. Once a bit is programmed to the 0-level, only the erase function can reset it back to 1. Erase can be performed for one segment, a group of segments, or for an entire module. This can vary for each device configuration, and the exact implementation should be noted in the data sheet.

The erase operation starts with the following sequence:

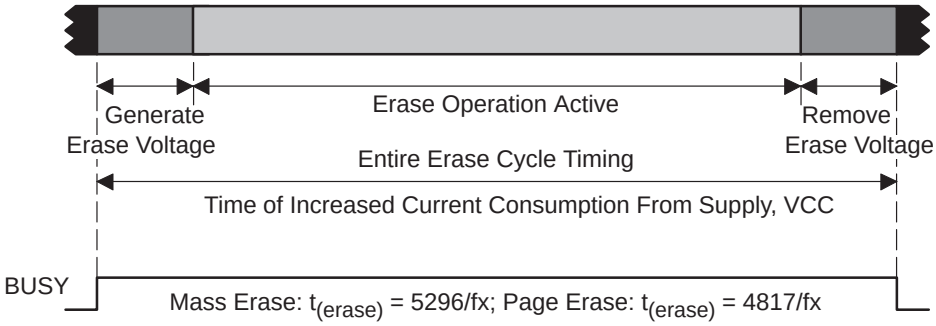
1) Set the correct input-clock frequency of the timing generator by selecting the clock source and predivider.
2) Reset the LOCK control bit, if set.
3) Watch the BUSY bit. Continue to the next steps only if the BUSY bit is reset.
4) Set the erase control bit *Erase* to erase a segment, or
5) Set the *mass-erase* control bit MEras to erase a group of segments or
6) Set the *mass-erase* (MEras) and erase (Erase) control bits to erase the entire flash memory
7) Execute a dummy write to any address in the range to be erased.

The *dummy write* starts the erase cycle. An example of dummy write is *CLR &0F012h*.

Note that a dummy write is ignored in a segment where the selected operation can not be executed successfully.

An example of such a situation can take place when Segment 1 is to be erased: the control bits are set properly, but the *dummy write* is sent to the information memory. No flag indicates this unsuccessful erase situation.

Figure 5–7. Basic Flash EEPROM Module Timing During the Erase Cycle



The erase cycle completes successfully when none of the following restrictions is violated:

- ❏ The selected clock source is available until the cycle is completed
- ❏ The predivider should not be modified during the operation
- ❏ No further access to the flash memory module is performed while BUSY is set
  - ■ No read of data from this block
  - ■ No write into this block
  - ■ No further erase of this block

  An access will result in setting the KEYV bit and requesting an NMI interrupt. The NMI interrupt routine should handle such violations.
- ❏ The supply voltage should be within the devices' electrical specifications defined in the respective data sheet; however, slight variations can be tolerated.

Control bit BUSY indicates an active erase cycle. It is set immediately after a dummy write starts the timing generator. It remains set until the entire erase cycle is completed and the erased segment or block is ready to be accessed again. The BUSY bit can not be set by software. But it can be reset. In case of emergency, set the emergency exit (EMEX) bit and the erase operation will be stopped immediately; BUSY bit is reset. One example of stop erase by software is when the supply voltage drops drastically and the operating conditions of the controller are exceeded. Another example is when the timing of the erase cycle gets out of control, for example, when the clock-source signal is lost.

Note:

When the erase cycle is stopped before its normal completion by the hardware, the timing generator is stopped and erasure of the flash memory can be marginal. An incomplete erasure can be verified. But an erase level of 1 can be inconsistently read as valid when supply voltage, temperature, access time (instruction execution, data read), and frequency vary.

5.2.6 Flash Memory Status During Write (Programming)

The flash memory erase bit level is 1. Bits can only be written (programmed) to a 0-level. Once a bit is programmed, only the erase function can reset it back to the 1-level. The byte or word 0-level can not be written (programmed) in one cycle. Any bit can be programmed from 1 to 0 at any time, but not from 0 to 1.

Two slightly different write operations can be performed: write a single byte or word of data, or write a sequence of bytes or words. A write sequence of bytes or words can be performed as multiple sequential, or as a segment write. The segment write is approximately twice as fast as a multiple sequential write algorithm.

The write (program) operation starts with the following sequence:

- ❑ Set the correct input clock frequency of the timing generator by selecting the clock source and predivider.
- ❑ Reset the LOCK control bit, if set
- ❑ Watch the BUSY bit. Continue with the next steps only if the BUSY bit is reset.
- ❑ Set the write-control bit WRT when a single byte of word data is to be written.
- ❑ Set the write WRT and SEGWRT control bits when segment write is chosen to write multiple bytes or words to the flash memory module.
- ❑ Writing the data to the selected address starts the timing generator. The data is written (programmed) while the timing generator proceeds.

Note:

Whenever the write cycle is stopped before its normal ending by the hardware, the timing generator is stopped and the data written to the flash memory can be marginal. The data may be incorrect, which can be verified, or the data are verified to be correct but the programming is marginal. Reading of the data may be inconsistently valid when varying the supply voltage, the temperature, the access time (instruction execution, data read), or the time.

Figure 5–8. Basic Flash Memory Module Timing During Write (Single Byte or Word) Cycle

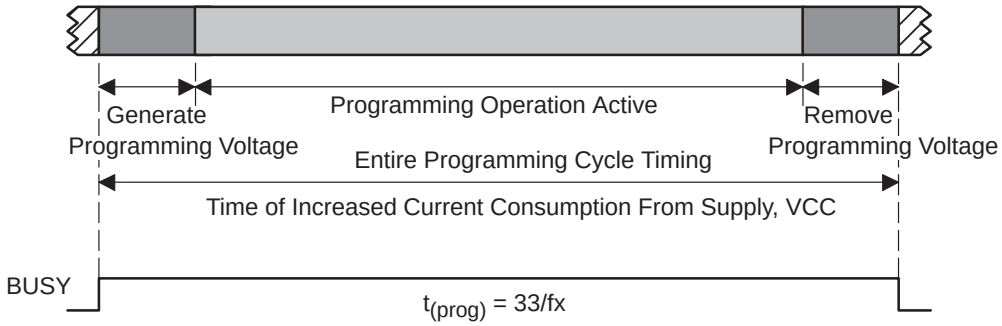
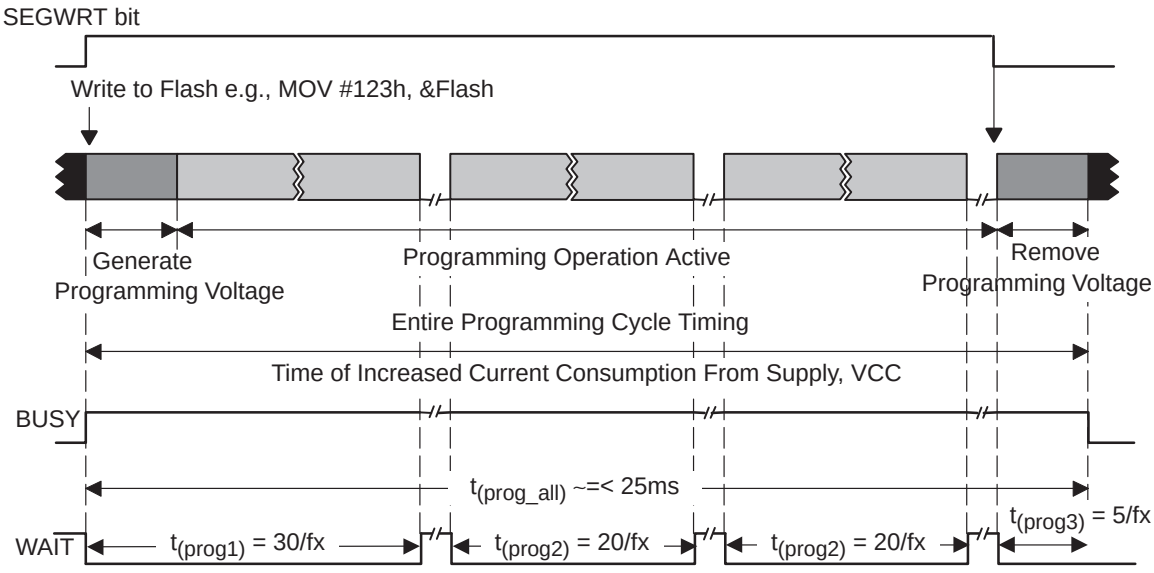


Figure 5–9. Basic Flash Memory Module Timing During a Segment-Write Cycle



The segment write can be used on sequential addresses of the memory module. One segment is 64 bytes long, starting at 0xx00h, 0xx40h, 0xx80h, or 0xxC0h, and ending at 0xx3Fh, 0xx7Fh, 0xxBFh, or 0xxFFh. Examples of sequential segment addresses are:

```
0F000h to 0F03Fh, 0F040h to 0F07Fh,0F080h to 0F0BFh,
0F0C0h to 0F0FFh, 0F100 to 0F13Fh,
```

The segment-write (program) operation at the 64-byte boundaries needs special software support (test of address 0xx3Fh, 0xx7Fh, 0xxBFh, or 0xxFFh was successful):

- ❑ Wait until the WAIT bit is set, indicating that the write of the last byte or word was completed.
- ❑ Reset control bit SEGWRT.
- ❑ The BUSY bit remains set until the programming voltage is removed from the flash memory module and overstress is avoided.
- ❑ Wait the recovery time $t_{(rcv)}$ before another segment write is started.

The write cycle is successfully completed if none of the following restrictions is violated:

- ❑ The selected clock source is available until the cycle is completed.
- ❑ The predivider is not modified.
- ❑ The access to the flash memory module is restricted as long as BUSY is set.

The conditions to read data from the flash memory with and without access violation are listed in Table 5–2.

Table 5–2. Conditions to Read Data From Flash Memory

Flash Operation	Instruction Fetch (see Note 1)	BUSY	WAIT	Data on Memory Data Bus (MDB)	Action
Byte/word program cycle (see Note 2)	No	1	0	3FFF	Access violation
	Yes	1	0	3FFF → JMP $	Nothing
Flash read mode		0	0	Memory contents from applied address	PC = PC + 2
Page erase cycle (see Note 3)	No	1	0	3FFF	Access violation
	Yes	1	0	3FFF → JMP $	Nothing
Mass-erase cycle (see Note 3)	No	1	0	3FFF	Access violation
	Yes	1	0	3FFF → JMP $	Nothing
All erase (mass and information memory)	No	1	0	3FFF	Access violation
	Yes	1	0	3FFF → JMP $	Nothing
Segment write (see Note 4)	N.A.	1	0	3FFF	Access violation and LOCK (see Note 5)
	No	1	1	3FFF	Nothing
	Yes	1	1	3FFF	Access violation and LOCK (see Note 5)

Notes: 1) Instruction fetch refers to the fetch part of an instruction, and reads one word. The instruction fetch reads the first word of instructions with more than one word. The JMP instruction has one word. The data fetched (3FFFh) is used by the CPU as an instruction.

2) Ensure that the programmed data does not result in unpredictable program execution, such as destruction of executable code sequences.

3) If the PC points to the memory location being erased, no access violation indicates this situation. After erase, no executable code is available and an unpredictable situation occurs.

4) Any software located in a flash memory module can not use the SEGWRT mode to program the same flash memory module. Using the byte or word programming mode allows programming data in the flash memory module holding the software code currently executing.

5) The access violation sets the LOCK bit to 1. Setting the LOCK bit allows completion of the active segment write operation in the normal manner.

- ❑ The supply voltage should be within the devices' electrical conditions and can only vary slightly, as specified in the applicable data sheet

The control bit BUSY indicates that the write or segment-write cycle is active. It is set by the instruction that writes data to the flash memory module and starts the timing generator. It remains set until the write cycle is completed and the programming voltage is removed. In the write mode the BUSY bit indicates if the flash memory is ready for another write operation. In segment write mode the WAIT bit indicates if the flash memory is ready for another write operation and the BUSY bit indicates the segment write operation is completed. In case of emergency, the emergency exit bit EMEX is set and stops the write cycle immediately. The programming voltage is switched off. One situation where the write cycle should be stopped by software is when the supply voltage drops drastically and the controller's operating conditions may be exceeded. Another case is when the flash memory timing gets out of control, as when the clock-source signal is lost.

Note:

Whenever the write cycle is stopped before its normal ending by the hardware, the timing generator is stopped and the data written in flash memory may be marginal. Data reading may be inconsistently valid when varying the supply voltage, the temperature, the access time (instruction execution, data read), or the time.

5.3 Flash Memory Control Registers

Defining the correct control bits of three control registers enables write (program), erase, or mass-erase. All three registers should be accessed using word instructions only. The control registers are protected against false write or erase cycles via a key word. Any violation of this keyword sets the KEYV bit and requests a nonmaskable interrupt (NMI). The keyword is different from the keyword used with the Watchdog Timer.

All control bits are reset during PUC. PUC is activated after V_{CC} is applied, a reset condition is applied to the RST/NMI pin or watchdog, or a flash operation was not performed normally.

5.3.1 Flash Memory Control Register FCTL1

Any write to control register FCTL1 during erase, mass-erase, or write (programming) will end in an access violation with ACCVIFG=1. In an active segment-write mode, the control register can be written if wait mode is active (WAIT=1). In an active segment write mode and while WAIT=0, writing to control register FCTL1 will also end in an access violation with ACCVIFG=1.

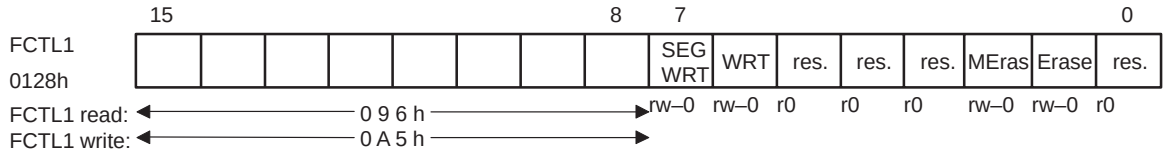
Read access is possible at any time without restrictions.

Any write to control register FCTL1 during erase, mass-erase, or write (programming) will end in an access violation with ACCVIFG=1. In an active seg-

ment write mode, the control register can be written if wait mode is active (WAIT=1). In an active segment write mode and while WAIT=0, writing to control register FCTL1 will also end in an access violation with ACCVIFG=1.

Read access is possible at any time without restrictions.

The control bits of control register FCTL1 are:



Erase 0128h, bit1, Erase a segment

0: No segment erase is started.

1: Erase of one segment is enabled. The segment n to be erased is defined by a *dummy* write into any address within the segment. The erase bit is automatically reset when the erase operation is completed.

Note: Instruction fetch access during erase is allowed. Any other access to the flash memory during erase results in setting the ACCVIFG bit, and an NMI interrupt is requested. The NMI interrupt routine should handle such violations.

MEras 0128h, bit2, Mass-erase, Segment0 to Segmentn are erased together.

0: No erase is started

1: Erase of Segment0 to Segmentn is enabled. When a *dummy* write into any address in Segment0 to Segmentn is executed, mass-erase is started. The MEras bit is automatically reset when the erase operation is completed.

Note: Instruction fetch access during mass-erase is allowed. Any other access to the flash memory during erase results in setting the ACCVIFG bit, and an NMI interrupt is requested. The NMI interrupt routine should handle such violations.

WRT 0128h, bit6, The bit WRT should be set to get a successful write execution.

If bit WRT is reset and write access to the flash memory is performed, an access violation occurs and ACCVIFG is set.

Note: Instruction fetch access during erase is allowed. Any other access to the flash memory during erase results in setting the ACCVIFG bit, and an NMI interrupt is requested. The NMI interrupt routine should handle such violations.

SEGWRT 0128h, bit7, Bit SEGWRT can be used to reduce total programming time.

The segment-write bit SEGWRT is useful if larger sequences of data have to be programmed. If programming of one segment is completed, a reset and set sequence should be performed to enable access to the next segment. The WAIT bit should be high before the next write instruction is executed. See also paragraph 16.1.1 and Figure 5–9.

0: No segment write accelerate is selected.

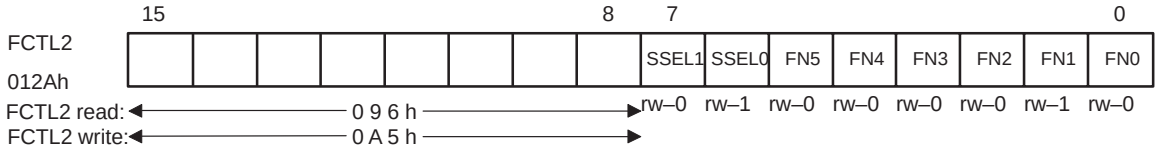
1: Segment write is used. This bit needs to be reset and set between segment borders.

5.3.2 Flash Memory Control Register FCTL2

A PUC resets the flash timing generator. The generator is also reset if the emergency exit bit EMEX is set.

The timing generator generates the timing necessary to write, erase, and mass-erase from a selected clock source. Two control bits SSEL0 and SSEL1 in control register FCTL2 can select one of three clock sources. The clock source selected should be divided to meet the frequency requirements for f_X, as specified in the device's data sheet.

Writing to control register FCTL2 should not be attempted if the BUSY bit is set; otherwise an access violation will occur (ACCVIFG=1). Read access to FCTL2 is possible at any time without restrictions.

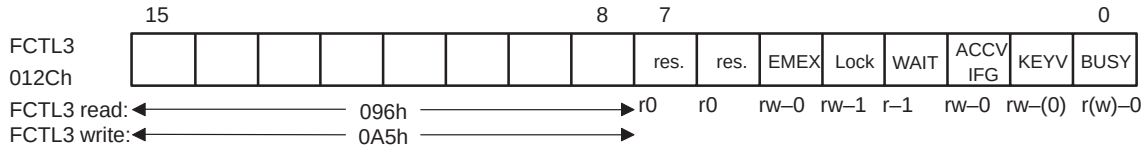


The control bits are:

FN0 to	012Ah, bit0,	These six bits define the division rate of the clock signal. The division rate can be 1 to 64, depending on the digital value of FN5 to FN0
FN5	012Ah, bit5,	plus one
SSEL0	012Ah, bit0,	Determine the clock source
SSEL1		0: ACLK
		1: MCLK
		2,3 SMCLK

5.3.3 Flash Memory Control Register FCTL3

There are no restrictions on modifying this control register. The control bits are reset or set (WAIT) by a PUC, but key violation bit KEYV is reset by POR.



BUSY 0128h, bit0, The bit BUSY shows if an access to the flash memory is possible (BUSY=0), or if an access violation can occur. The BUSY bit is read only, but a write operation is allowed. The BUSY bit should be tested before each write and erase cycle. The flash-timing generator hardware immediately sets the BUSY bit after the start of a write operation, a segment-write operation, a segment erase, or a mass-erase. Once the timing generator has completed its function, the BUSY bit is reset by hardware.

The program and erase timing are shown in Figures 5–7, 5–8, and 5–9.

0: Flash memory is not busy. Read, write, erase and mass-erase are possible without any violation of the internal flash timing. The BUSY bit is reset by POR and by the flash timing generator.

1: Flash memory is busy. Remains in busy state if segment write function is in *wait* mode.

The conditions for access to the flash memory during BUSY=1 are described in paragraph 5.2.6.

KEYV, 012Ch, bit1, Key Violated.

0: Key 0A5h (high byte) was not violated.

1: Key 0A5h (high byte) was violated. Violation occurs when a write access to register FCTL1, FCTL2 or FCTL3 is executed and the *high byte* is not equal to 0A5h. If the security key is violated, bit KEYV is set and a PUC is performed. The KEYV bit can be used to determine the source that forced a start of the program at the reset vector's address. The KEYV bit is not automatically reset and should reset by software.

Note: Any key violation results in a PUC, independent of the state of the KEYV bit. To avoid endless software loops, the flash memory control registers should not be written during a key violation service routine.

Note: The software can set the KEYV bit. A PUC is also performed if it is set by software.

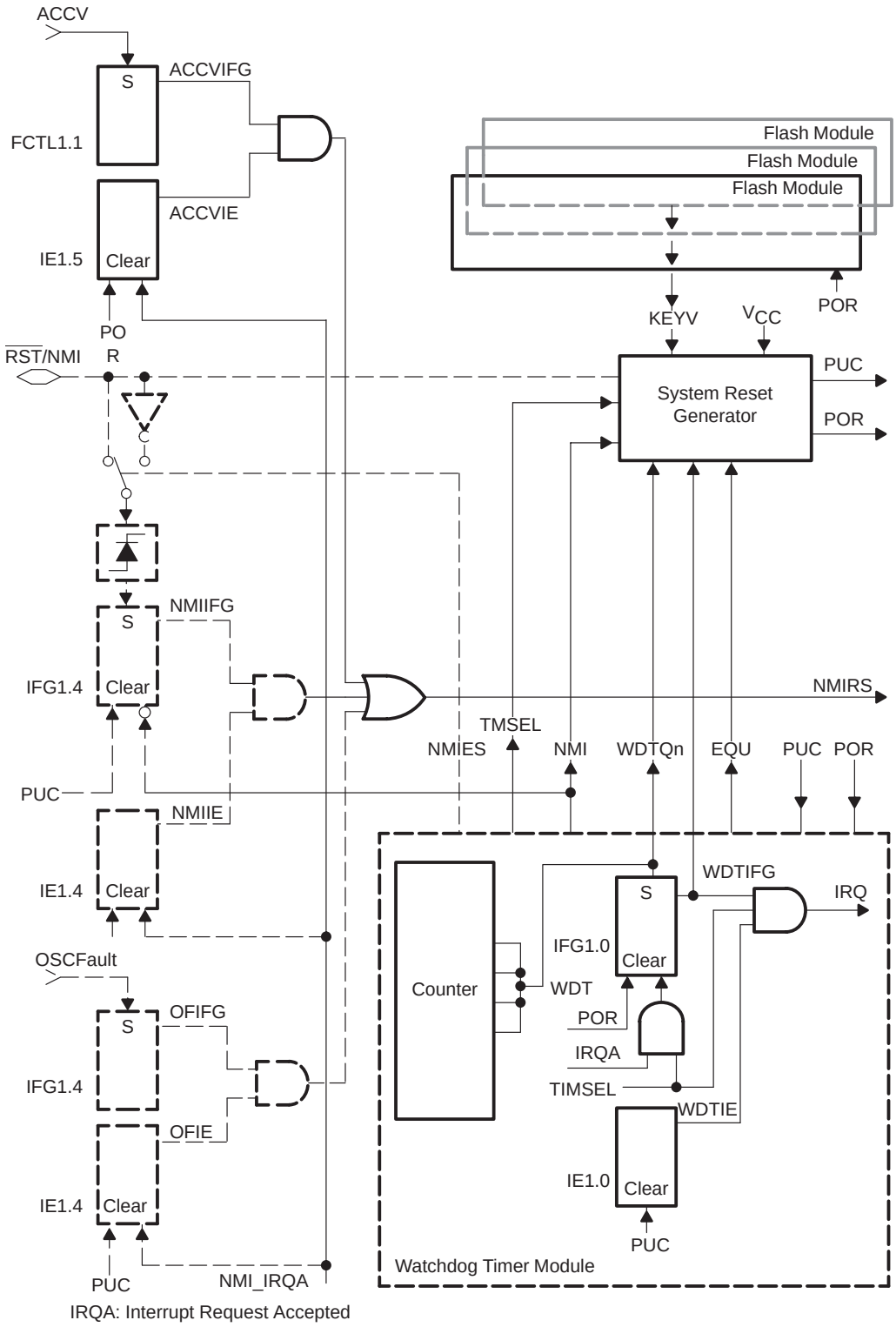
ACCVIFG	bit2,	Access violation interrupt flag The access-violation interrupt flag is set when the flash memory module is improperly accessed while a write or erase operation is active. The violation situations are described in paragraph 5.2.3. When the access-violation interrupt-enable bit is set, the interrupt-service request is accepted and the program continues at the NMI interrupt-vector address. Reading the control registers will not set the ACCVIFG bit. Note: The proper interrupt-enable bit ACCVIE is located in interrupt-enable register IE1 of the special-function register. Software can set the ACCVIFG bit; in this case, an NMI is also executed.
WAIT	012Ch, bit3,	Wait. In the segment write mode the WAIT bit indicates that the flash memory is ready to receive the (next) data for programming. The WAIT bit is read only, but a write to the WAIT bit is allowed. The WAIT bit is automatically reset if the SEGWRT bit is reset or the LOCK bit is set. Segment-write operation is completed, and then the WAIT bit returns to 1. Condition, SEGWRT=1 (see Figure 5-9): After each successful write operation, the BUSY bit is reset to indicate that another byte or word can be written (programmed). The BUSY bit does not indicate the condition when the timing generator has completed the entire programming. The high-voltage portion and voltage generator remain active. The maximum time $t_{(CPT)}$ should not be violated. 0: Segment-write operation has started and programming is in progress. 1: Segment-write operation is active and programming of data is completed. Waiting for the next data to be programmed.
Lock	012Ch, bit4,	The lock bit can be set during any write, erase of a segment, or mass-erase request. The active sequence is completed normally. In segment-write mode, if the lock bit is set and SEGWRT and WAIT are set, the SEGWRT and WAIT bits are reset and the mode ends normally. The WAIT bit is 1 after segment-write mode has ended. Software or hardware can control the lock bit. If an access violation occurs (see conditions described in paragraph 16.1.1), the ACCVIFG and the lock bit are set. 0: Flash memory can be read, programmed, erased, and mass-erased. 0: Flash memory can be read but not programmed, erased, or mass-erased. A current program, erase, or mass-erase operation is completed normally. The access-violation interrupt flag ACCVIFG is set when the flash memory module is accessed while the lock bit is set.

EMEX	012Ch, bit5,	Emergency exit. The emergency exit should only be used when a flash memory write or erase operation is out-of-control. 0: No function. 0: Stops the active operation immediately and shuts down all internal parts of the flash memory controller. Current consumption immediately drops back to the active mode. All bits in control register FCTL1 are reset. Since the EMEX bit is automatically reset by hardware, the software always reads EMEX as 0.

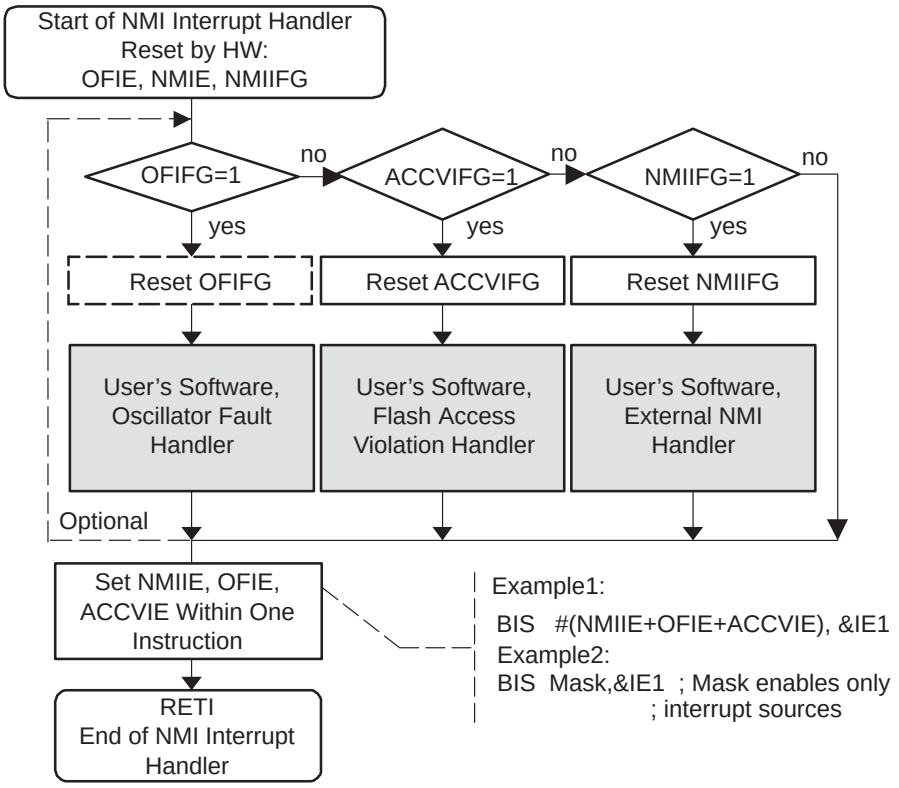
5.4 Flash Memory, Interrupt, and Security Key Violation

One NMI vector is used for three non-maskable interrupt (NMI) events, RST/NMI, oscillator fault (OFIFG), and flash-access violation (ACCVIFG). The software can determine the source of the interrupt request by testing interrupt flags NMIIFG, OFIFG, and ACCVIFG. They remain set until reset by software.

Figure 5–10. Basic Flash Memory Module Timing During Segment-Write Cycle



5.4.1 Example of an NMI Interrupt Handler



The NMI handler takes care of all sources requesting a nonmaskable interrupt. The NMI interrupt is a multiple-source interrupt per MSP430 definition. The hardware resets the interrupt-enable flags: the external nonmaskable interrupt enable NMIIE, the oscillator fault interrupt enable OFIE, and the flash memory access-violation interrupt enable. The individual software handlers reset the interrupt flags and reenables the interrupt enable bits according to the application needs. After all software is processed, the interrupt enable bits have to be set if another NMI event is to be accepted. Setting the interrupt enable bits should be the last instruction before the return-from-interrupt instruction RETI. If this rule is violated, the stack can grow out of control while other NMI requests are already pending. Setting the interrupt enable bits can be accomplished by using a bit-set-instruction BIS using immediate data or a mask. The mask data can be modified anywhere via software (for example in RAM); this constitutes the nonmaskable interrupt processing.

5.4.2 Protecting One-Flash Memory-Module Systems From Corruption

MSP430 configurations having one flash memory module use this module for program code and interrupt vectors. When the flash memory module is in a write, erase, or mass-erase operation and the program accesses it, an access violation occurs. This violation will request an interrupt service; however, when the interrupt vector is read from the flash memory, 03FFFh will be read independent of the data in the flash memory at the vector's memory location.

To protect the software from this error situation, all interrupt sources have to be disabled since all interrupt requests will fail. The flash memory returns the vector 03FFFh. Before the interrupt enable bits are modified, they can be stored in RAM to be restored when the flash memory is ready for access again.

The following interrupt enable bits should be reset to stop all interrupt service requests:

- GIE = 0
- NMIIE = ACCVIE = OFIE = 0

Additionally the watchdog should be halted to prevent its expiration when flash memory is busy:

- WDTHOLD = 1

When the flash memory is ready, the interrupt sources can be enabled again. Before they are enabled, critical interrupt flags should be checked and, if necessary, served or reset by software.

- GIE = 1 or left disabled, or be restored to the previous level
- NMIIE = ACCVIE = OFIE = 1 or left disabled, or be restored to the previous level
- WDTHOLD = 0 or left disabled, or be restored to the previous level

5.5 Flash Memory Access via JTAG and Software

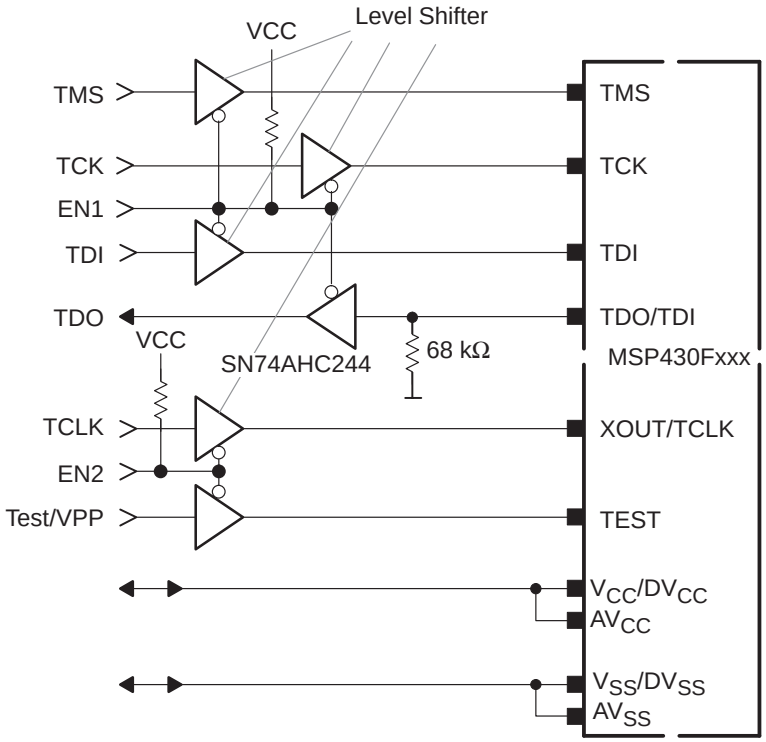
5.5.1 Flash Memory Protection

Flash memory access via the serial test and programming interface JTAG can be inhibited when the *security fuse* is activated. The *security fuse* is activated via serial instructions shifted into the JTAG. Activating the fuse is not reversible, and any access to the internal system is disrupted. The bypass function described in the IEEE1149.1 standard is active.

5.5.2 Program Flash Memory Module via Serial Data Link Using JTAG Feature

The hardware interconnection to the JTAG pins is done via four separate pins, plus the ground or V_{SS} reference level. The JTAG pins are TMS, TCK, TDI (/VPP), and TDO (/TDI).

Figure 5–11. Signal Connections to MSP430 JTAG Pins

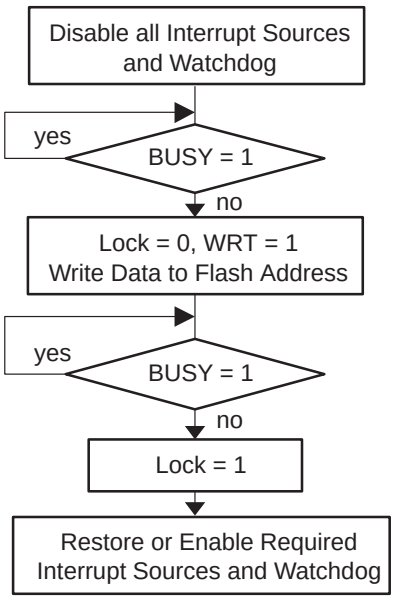


5.5.3 Programming a Flash Memory Module via Controller Software

No special external hardware is required to program a flash memory module. The power supply at pin V_{CC} should supply the higher current during write (program) and erase modes. The software algorithm is simple. The embedded timing generator in the flash memory module controls the program and erase cycles. Software can not run in the same flash memory module where data is to be written. Such background software needs to run on other memory device, such as a ROM module, a RAM module, or another flash memory module.

5.5.3.1 Example: Programming One Word Into a Flash Memory Module via Software Execution Outside This Module

This example assumes that the code to program the flash location is not executed from the target flash memory module.




```
Fxkey    .set    03300h
Fwkey    .set    0A500h
; No interrupt request may happen while the flash is programmed

Test_Busy1
        BIT     #BUSY,&FCTL3
        JNZ     Test_Busy1

        MOV     #Fwkey,&FCTL3             : Clear lock bit
        MOV     #(Fwkey+WRT),&FCTL1:      : Enable write to flash
        MOV     #123h,&0FF1Eh             : Write a word to flash

Test_Busy2
        BIT     #BUSY,&FCTL3              ; still busy?
        JNZ     Test_Busy2                ; yes, repeat busy test
        MOV     #Fwkey,&FCTL1             : Reset write bit
        XOR     #(Fxkey+Lock),&FCTL3      : Change lock bit to 1

; Enable those interrupt sources that should be accepted
```

The BUSY bit can be tested before the write to the flash memory module is done, or after a write (program) starts:

- For flash memory locations that hold data, it is a good practice to test the BUSY bit before the write is executed. This has some time benefits, since the write process is executed via the flash memory timing generator without further CPU intervention. It is important that the clock source remains active until BUSY is reset by the flash memory hardware.

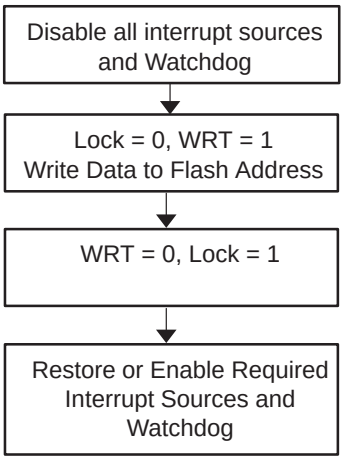
  The power or clock management, responsible for entering low-power modes, has to make sure that it does not switch off the clock source used by the flash controller.

- For flash memory blocks that hold program code, it is a good practice to test the BUSY bit after the write is executed. The program can only proceed if the module can be accessed again. No special attention is needed during execution of software code. Every write to the flash memory module has to leave the programming cycle with the BUSY bit reset.

  Testing the BUSY bit before writing to a flash memory block that holds program code ensures that the active program will not access the flash memory module. Two types of access are visible: execute program code, or read and write data on this flash memory module.

5.5.3.2 Example: Programming One Word Into the Same Flash Memory Module via Software

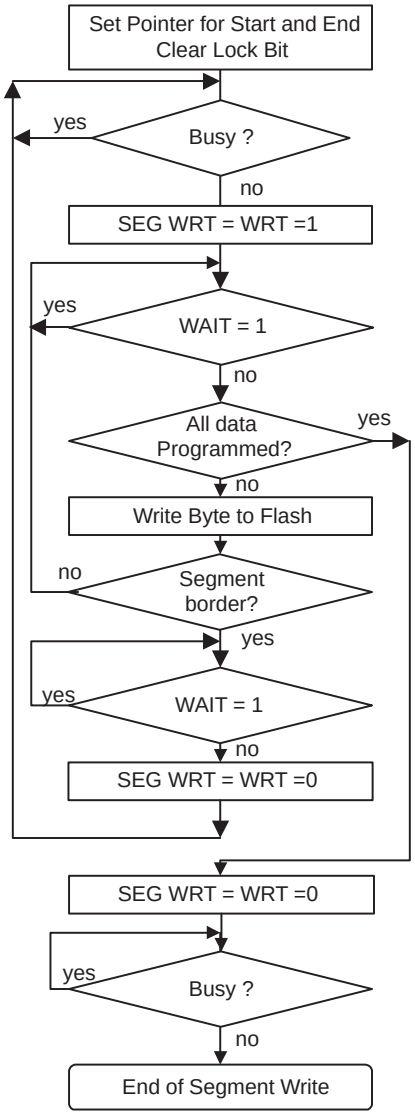
The program execution waits after the write-to-flash instruction (MOV #123h,&0FF1Eh) until the busy bit is reset again. If no other write-to-flash instruction method is used the BUSY bit test may not be needed to ensure correct flash-write handling.




```
Fxkey   .set    03300h
Fwkey   .set    0A500h

; No interrupt request may happen while the flash is programmed

        MOV     #(Fwkey+WRT),&FCTL1:  : Enable Write to flash
        MOV     #123h,&0FF1Eh         : Write a word to flash

        MOV     #Fwkey,&FCTL1         : Reset Write bit
        XOR     #(Fxkey+Lock),&FCTL3  : Change Lock bit to 1

; Enable those interrupt sources that should be accepted
```

5.5.3.3 Example, Programming Byte Sequences Into a Flash Memory Module via Software

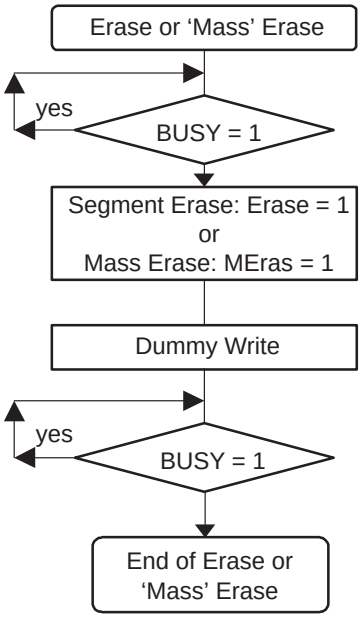
Sequences of data, bytes, or words can use the segment-write feature. This reduces the programming time by about one half.




```
Fxkey   .set   03300h
Fwkey   .set   0A500h
Frkey   .set   09600h

; Ensure that neither Watchdog Timer, nor
; interrupts nor Low–Power Modes may corrupt
; proper execution

RAM2FLASH
        MOV    #Start_Ptr,Rx
        MOV    #End_Ptr,Ry
        MOV    #Fkey,&FCTL3              ; Clear lock bit
Test_Busy1
        BIT    #BUSY,&FCTL3              ; Flash busy? SEGWRT ended?
        JNZ    Test_Busy1

        MOV    #(Fwkey+WRT+SEGWRT),&FCTL1;Segment write

Test_WAIT1
        BIT    #WAIT,&FCTL3
        JNE    Test_WAIT1

        CMP    Rx,Ry                     ; All data programmed?
        JZ     End_Seg_Write

       ; Program data: in this example one byte
        MOV.B  @Rx+,(Flash_Start_Ptr–Start_ptr–1) (Rx)

        BIT    #03Fh,Rx                  ; Segment border?
        JNZ    Test_Wait1

Test_Wait2
        BIT    #WAIT,&FCTL3              ; Test if data written
        JZ     Test_Wait2

        MOV    #Fwkey,&FCTL1             ; Stop segment write
        JMP    Test_busy1                ; Segment write ends if busy=0

End_Seg_Write
                                         ; All data are programmed
        MOV    #Fwkey,&FCTL1             ; Stop segment write

Test_Busy2
        BIT    #BUSY,&FCTL3              ; Segment write ended?
        JNZ    Test_Busy2
        XOR    #(Fxkey+Lock),&FCTL3      ; Change Lock bit to '1
```

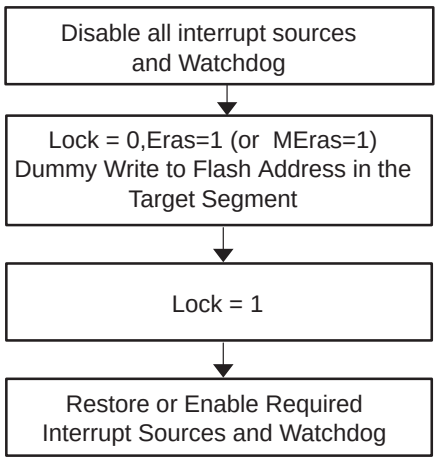
5.5.3.4 Example, Erase Flash Memory Segment or Module via Software Execution Outside This Flash Module

The following sequence can be used to erase a segment, or mass-erase a block of segments.




```
BIC     #( Fwkey+Lock),&FCTL3  ;Reset Lock bit

Test_Busy1
        BIT       #BUSY,&FCTL3
        JNZ       Test_Busy1

        MOV       #(Fwkey+Erase),&FCTL1  ; select segment erase


        CLR    &0F000h

Test_Busy2
        BIT       #BUSY,&FCTL3
        JNZ       Test_Busy2


        XOR    #(Fxkey+Lock),&FCTL3
```

5.5.3.5 Example, Erase Flash Memory Segment Module in the Same Flash Memory Module via Software

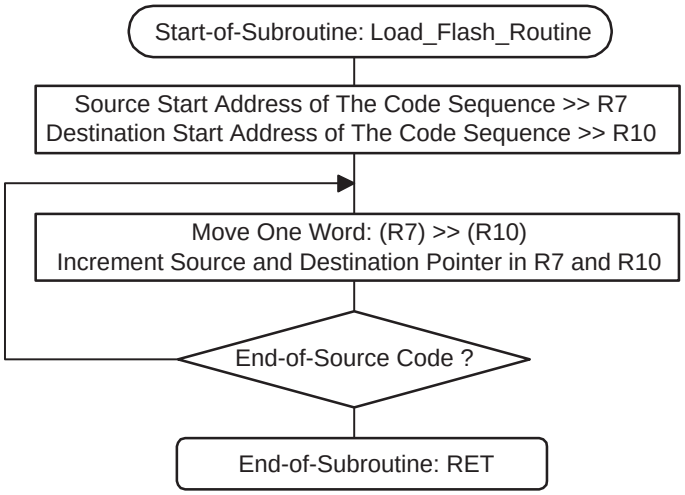



```
; Disable all possible interrupt sources and watchdog

MOV #(Fwkey+Eras),&FCTL1          ;Enable Erase of Flash
CLR  &0FA00h                      ;Dummy Write to Flash
                                  ;Erase Segment 2
; Program execution in information memory if MEras=1 (Eras=0)

XOR  #(Fwkey+Lock),&FCTL3         : Change Lock bit to 1

; The erase bit Eras is automatically reset

; Enable those interrupt sources that should be accepted
```

5.5.3.6 Code for Write (Program), Erase, and Mass-Erase

Software that is active during write, erase, or mass-erase may not run in the flash memory module where it is written or erased. Software that controls write, erase, or mass-erase can be located in the flash memory module and copied during execution into RAM. In this case the code should be written position-independent, and should be loaded (for instance, to RAM) before it is used. The algorithm runs in RAM during the programming sequence to avoid conflict when the flash memory is written or erased.

The target flash memory module can not execute the programming code sequence while data is being written to it. In the following example, a subroutine moves the programming-code sequence to another memory such as RAM.




```
;-------------------------------------------------------------
; Definitions used in Subroutine:
; Move programming code sequence into RAM (load_flash_routine)
;-------------------------------------------------------------
Flash_ram          .set   0222h  ; Start address of flash
                                 ; program in the RAM
                                 ; program in the RAM
Prg_source_start   .set   0xxxxh ; Start address of code
                                 ; in the flash to be prg'ed
Prg_source_end     .set   0yyyyh ; End address of code
                                 ; in the flash to be prg'ed
Prg_dest_start     .set Flash_ram
load_flash_routine    ; The code of the program which moves
                      ; Flash access code (write, erase,..)
                      ; starts at label load_flash_routine
   push   r9
   push   r10
   mov    #Prg_source_start,R9  ; load pointer source
   mov    #Prg_dest_start,R10   ; load pointer destination
load_flash_prg
   mov    @R9,0(R10)            ; move a word
   incd   R10                   ; destination  pointer + 2
   incd   R9                    ; source pointer + 2
   cmp    # Prg_source_end,R9   ; compare to end_of_code
   jne    load_ flash_prg
   pop    r9
   pop    r10
   ret
```

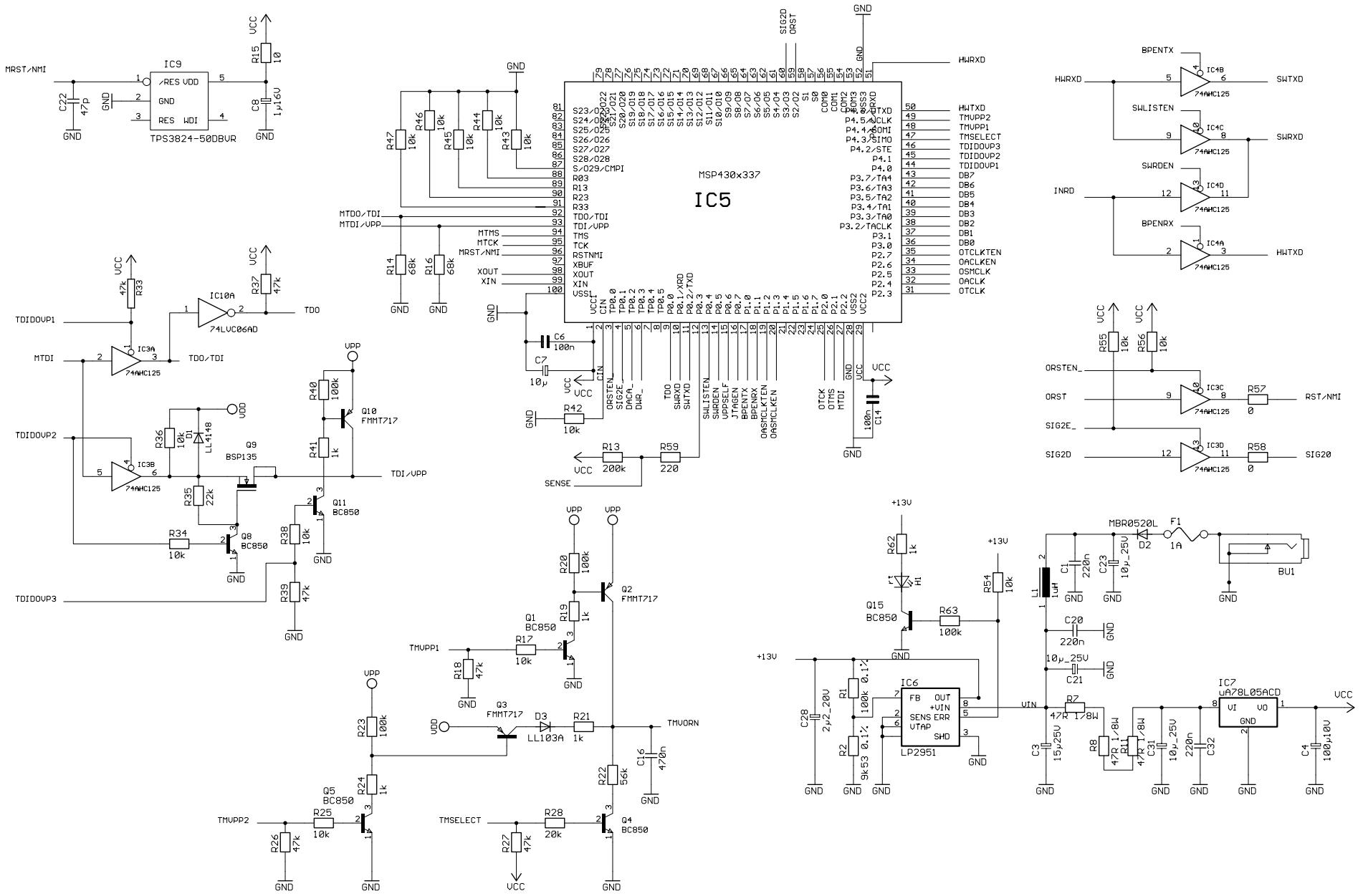
Appendix A

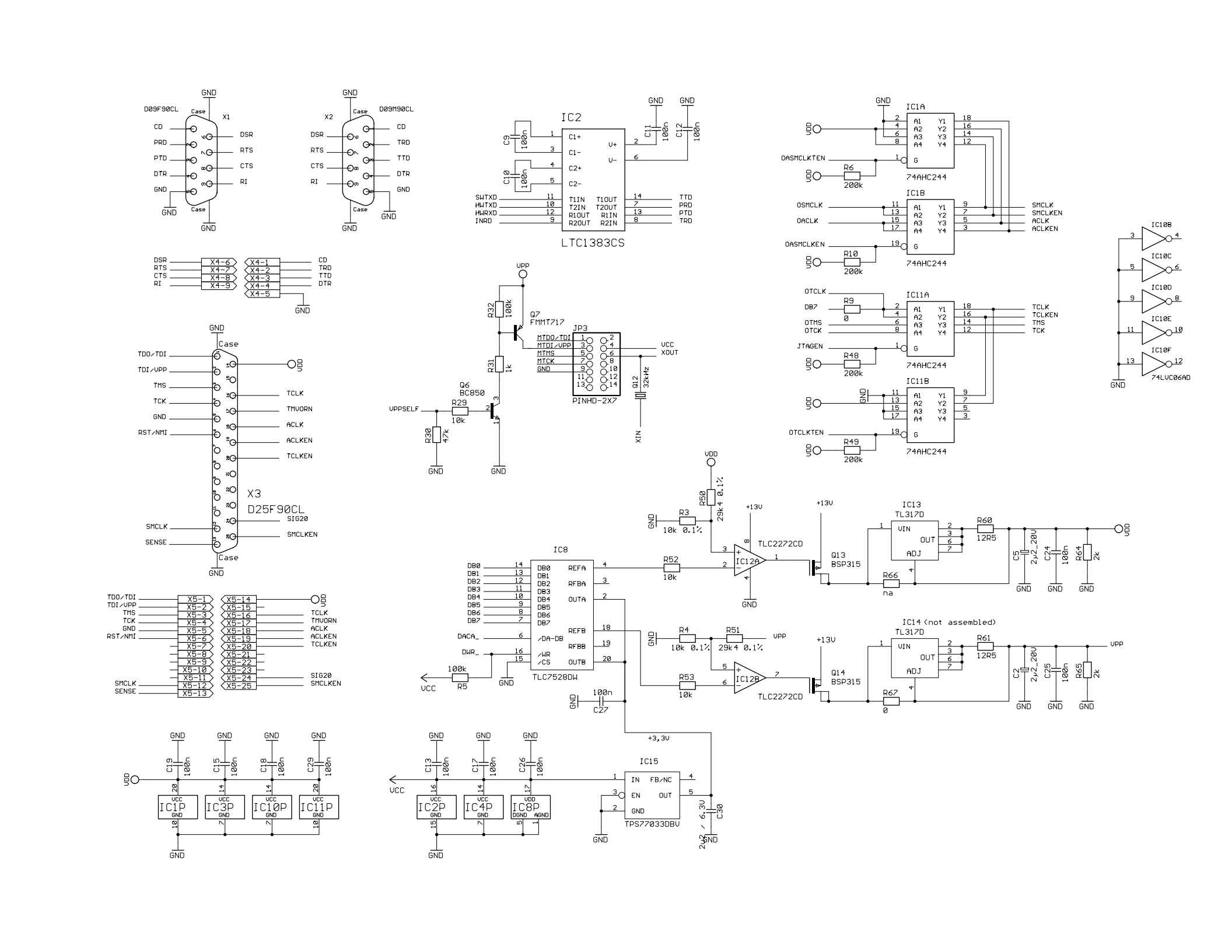
Schematics

This appendix contains the schematic diagrams for the serial programming adapter.

Topic **Page**

SERIAL PROGRAMMING ADAPTER




D09F90CL
X1
CD
PRD
PTD
DTR
GND
DSR
RTS
CTS
RI
Case
D09M90CL
X2
TRD
TTD
IC2
LTC1383CS
C1+
C1-
C2+
C2-
V+
V-
T1IN
T2IN
R1OUT
R2OUT
T1OUT
T2OUT
R1IN
R2IN
SWTXD
HWTXD
HWRXD
INRD
TTD
PRD
PTD
TRD
IC1A
IC1B
74AHC244
OASMCLKTEN
OSMCLK
OACLK
OASMCLKEN
SMCLK
SMCLKEN
ACLK
ACLKEN
IC11A
IC11B
OTCLK
DB7
OTMS
OTCK
JTAGEN
OTCLKTEN
TCLK
TCLKEN
TMS
TCK
IC10B
IC10C
IC10D
IC10E
IC10F
74LVC06AD
X4-6
X4-7
X4-8
X4-9
X4-1
X4-2
X4-3
X4-4
X4-5
TDO/TDI
TDI/VPP
RST/NMI
TMVORN
SIG20
SENSE
X3
D25F90CL
Q7
FMMT717
Q6
BC850
JP3
MTDO/TDI
MTDI/VPP
MTMS
MTCK
PINHD-2X7
VPP
VPPSELF
VCC
XOUT
XIN
Q12
32kHz
IC8
TLC7528DW
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
REFA
RFBA
OUTA
REFB
RFBB
OUTB
/DA-DB
/WR
/CS
DACA_
DWR_
IC12A
IC12B
TLC2272CD
Q13
Q14
BSP315
IC13
IC14 (not assembled)
TL317D
VIN
OUT
ADJ
IC15
TPS77033DBV
IN
FB/NC
EN
+13V
+3,3V
IC1P
IC3P
IC10P
IC11P
IC2P
IC4P
IC8P
VCC
VDD
GND
R3
R4
R5
R6
R9
R10
R29
R30
R31
R32
R48
R49
R50
R51
R52
R53
R60
R61
R64
R65
R66
R67
C5
C2
C24
C25
C27
C30
C9
C10
C11
C12
C13
C15
C17
C18
C19
C26
C29
2µ2_20V
100n
2k
12R5
200k
10k
47k
1k
100k
29k4 0.1%
10k 0.1%
na