MSP430L092 Loader Code

The MSP430L092 microcontroller (MCU) is a development, prototyping, and small series member of the MSP430x09x device family. It contains a loader code as ROM firmware. This user’s guide describes how the MSP430L092 loader code is used to build an autonomous microcontroller solution. The loader approach is chosen as nonvolatile memory is not available for native ultra-low supply voltage.

Contents
1 Loader Code Introduction .............................................................................................................. 3
   1.1 Typical Two-Chip Application ................................................................................................. 3
   1.2 Code Generation, Conventions, and Restrictions ................................................................. 4
2 Loader Code Operation .................................................................................................................. 5
   2.1 Start-up Behavior and Timing ................................................................................................. 7
   2.2 Start-up Code (SUC) .............................................................................................................. 8
   2.3 Data Structure of SPI Memory ............................................................................................... 9
   2.4 Data and Program Containers ............................................................................................ 10
   2.5 Failsafe Mechanism ........................................................................................................... 11
   2.6 API Functions of Loader .................................................................................................... 12
   2.7 Software Registers and Public Data Elements .................................................................. 30
   2.8 Interrupt Handling ............................................................................................................. 35
   2.9 Sockets and Plug-Ins ........................................................................................................... 35
   2.10 Power Efficient Program Loading ...................................................................................... 35
   2.11 Programming With Overlays ............................................................................................. 36
   2.12 Debugging Checkpoints for Code Development ............................................................... 37
   2.13 Inner Mechanism of Loader ............................................................................................. 38
   2.14 SPI Commands Used by Loader ....................................................................................... 38
3 Target Hardware .......................................................................................................................... 39
   3.1 Booster Converters ............................................................................................................. 39
   3.2 Adaptation Networks ......................................................................................................... 41

List of Figures
1 Debugging Scenarios With MSP430x09x Devices ...................................................................... 3
2 Component-Optimized Application Circuit for 0.9-V Supply ..................................................... 4
3 Structure of Loader Code With API .......................................................................................... 5
4 Flow Chart of Loader ............................................................................................................... 6
5 Flow Chart of XOVL Function .................................................................................................. 6
6 Timing of Successful Load Operation ...................................................................................... 7
7 Flow Chart of the Start-Up Code for ‘C09x devices ................................................................ 8
8 Data Structures in SPI Memory ................................................................................................ 9
9 Data/Program Container ........................................................................................................ 10
10 Timing of Error Signature .................................................................................................... 11
11 Stack Behavior of XOVL() .................................................................................................. 17
12 Stack Behavior of LOVL() ...................................................................................................... 18
13 Stack Behavior of COVL() ................................................................................................... 19
14 Stack Behavior of ROVL() .................................................................................................. 20
15 CurOvlSpiH Register ............................................................................................................. 31
List of Tables

1. Debugging Scenarios With MSP430x09x Devices .................................................. 3
2. Public API Functions ......................................................................................... 13
3. Software Registers ......................................................................................... 30
4. Secondary Interrupt Vectors ........................................................................... 30
5. CurOvlSpiL Register Description ................................................................. 31
6. CurOvlSpiHLRegister Description ............................................................... 31
7. CurOvlAdrH Register Description ................................................................. 32
8. CurOvlAdrL Register Description ................................................................. 32
9. Status_Reg Register Description .................................................................. 33
10. LedOffPtr Register Description ................................................................... 34
11. LedOnPtr Register Description ..................................................................... 34
12. UnexpCnt Register Description .................................................................... 34
13. SPI Commands Used by Loader ................................................................. 38
14. Values of the Components ............................................................................. 39

Trademarks

MSP430 is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.
1 Loader Code Introduction

The loader code in the MSP430L092 MCU is ROM code from Texas Instruments that provides a series of services. The loader code lets customers build autonomous applications without the need for a custom ROM mask. Such an application consists of a MSP430™ MCU containing the loader (for example, MSP430L092) and an SPI memory device (for example, the ‘95512 or ’25AA40). These and similar memory devices are available from various manufacturers.

The most common reasons to make an application that uses a loader device and external SPI memory for native 0.9-V supply voltage are late development, prototyping, and small series production. Table 1 shows various debugging scenarios possible for ultra-low supply voltage. A loader approach is the only choice for an autonomous application, because no nonvolatile memories are available on the market for native ultra-low supply voltage, as of this writing.

Table 1. Debugging Scenarios With MSP430x09x Devices

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Early Development</th>
<th>Late Development</th>
<th>Prototyping</th>
<th>Small Series</th>
<th>Mass Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Units</td>
<td>Up to 10</td>
<td>Up to 100</td>
<td>Up to 1000</td>
<td>Up to 100000</td>
<td>100000 or more</td>
</tr>
<tr>
<td>Device</td>
<td>MSP430L092</td>
<td>MSP430L092</td>
<td>MSP430L092</td>
<td>MSP430L092</td>
<td>MSP430C091/C092</td>
</tr>
<tr>
<td>Cost per Unit</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Code Stored in</td>
<td>IDE, RAM</td>
<td>External memory, RAM</td>
<td>External memory, RAM</td>
<td>External memory, RAM</td>
<td>External memory, ROM, RAM</td>
</tr>
<tr>
<td>Galvanic Separation</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RAM Size (typical)</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>1024 or 2048 bytes</td>
</tr>
<tr>
<td>Overlays</td>
<td>Supported by L092</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Depends on customer code</td>
</tr>
</tbody>
</table>

The user can determine the type of SPI memory device to use with the MSP430 device with loader code. SPI-EEPROM, SPI-Flash, SPI-SRAM, SPI-FRAM, and SPI-byte-alterable flash devices with supply voltages ranging from 1.8 V to 6 V and various memory sizes are on the market.

1.1 Typical Two-Chip Application

An application with the MSP430L092 device can be as simple as the one in Figure 2. The loader code initializes the MSP430 device, generates an external clock on port P1.2; this clock enables an external boost converter that generates the necessary supply voltage for the SPI device containing the user program. After approximately 500 µs, the loader code starts to load the user code into the L092 RAM. After a successful load procedure, the user code is started. During the code loading process, the LED that is used to stabilize the voltage for the SPI device lights briefly. The LED may be used for regular signaling purposes in the application, because the SPI device is inactive after the initial loading process.
1.2 **Code Generation, Conventions, and Restrictions**

The application code is generated with the standard tools for MSP430 devices. The user application may (but does not have to) use other service provided by the loader API. If the API services are used, then special conventions must be followed (see Section 2.6); otherwise, the user code can be written without any restrictions.
## Loader Code Operation

The L092 loader code consists of two blocks, the loader application and the loader API, as shown in Figure 3. After the loader gains control, it initializes itself, load the user application code from the image in the SPI device into the internal RAM of the MSP430 MCU, and invokes the user code (see Figure 4). The user application has full access to the public functions and services of the loader using a standardized API interface. The API consists of the hardware abstraction layer (HAL), the API core functions, and the API interface (API IF).

- The HAL is a set of low-level device- and platform-oriented functions that all other higher-level functions are based on.
- The API core functions are higher-level functions that provide more complex operations such as copying memory.
- The API IF provides a standardized way to provide the HAL and API core functions to the user. The API IF should remain the same in any versions that the loader API core might go through in future. The API IF should also remain the same in all of the devices into which the loader is ported in the future. The API IF might be extended if the functions of the API grows over time; the functions or the earlier versions should still behave the same way in the newer code versions (so that backward and forward compatibility for common functions is maintained) (see Figure 5).

![Figure 3. Structure of Loader Code With API](image-url)
Loader Code Operation

Figure 4. Flow Chart of Loader

Figure 5. Flow Chart of XOVL Function
2.1 Start-up Behavior and Timing

Immediately after startup, devices with the loader code behave the same as devices with any other user code. After \( V_{CC} \) ramp-up or reset release, the control is given to the start-up code (SUC) (for details, see Section 2.2). The SUC initializes the device and performs device integrity checks, then it passes control to the code in ROM by branching to the location where the ROM code start vector is pointing (in this case, the vector points to the loader code).

The loader performs its initialization and turns on a 250-kHz clock on port P1.2. After approximately 500 µs, the user application code residing in the external SPI memory is loaded into the internal MSP430 RAM. When stored in the external SPI memory, the application code is embedded in a data container that is protected with checksums (see Section 2.3 and Section 2.3). During the loading process, the checksum is verified, and control is passed to the application code only if the checksum is correct (see Figure 6).

![Figure 6. Timing of Successful Load Operation](image)

**NOTE:** The ROM code start vector is located at 0xF840 for the MSP430x09x devices. It is a reduced length 16-bit address pointer that points to the start of the loader.
2.2 **Start-up Code (SUC)**

The SUC is firmware that is provided by TI and is the first code that is invoked after reset. The SUC configures the device, checks the JTAG password to allow debugging on correct password, performs a checksum-based code integrity check, calibrates the oscillator based on optional available calibration data, and finally invokes the application (that is, the user code).

The SUC of the L092 device differs slightly from that of the 'C09x devices. L092 devices are always open and ready for debugging. On C092 devices, the password must be provided through the JTAG mailbox input registers before reset release. A valid password causes the device to enter LPM4 and wait for JTAG emulation (see Figure 7).

![Flow Chart of the Start-Up Code for 'C09x devices](image-url)

**Figure 7. Flow Chart of the Start-Up Code for 'C09x devices**
2.3 **Data Structure of SPI Memory**

The application code is typically kept in external SPI memory when using the loader approach (see Figure 8). One-bit-wide SPI devices with 16-bit and 24-bit address ranges are supported. At location 0x0, a format indication is expected for both address types. The loader code automatically adapts its SPI address width to the identified SPI memory device size by checking for the format indicator at memory location at addresses 0x0 and 0x1. The first boot data/program container is expected at address 0x2. Other data/program containers may be stored anywhere in the SPI memory. Loading data/program containers from SPI addresses below 0x800 automatically causes an LED turn off operation and an password check with a stop for debugging purposes. Data/program containers loaded from SPI addresses at or above 0x800 do not cause an automated LED off operation; this area is typically used for overlay programming.

![Figure 8. Data Structures in SPI Memory](image-url)
2.4 Data and Program Containers

A data or program container is a structure stored in SPI memory that contains data or program (code) elements as payload (see Figure 9). The header of the data container consists of a 16-bit length field, a 16-bit destination address where the code should be loaded, and a 16-bit start address field that is invoked after code load.

The length field represents the count of the payload in bytes. The payload itself is always of even length. Zero padding at the end of payload is used if the length is an odd value. The theoretical maximum block length is 65536 bytes.

The load address points to the MSP430 memory location where the payload should be written (when not overridden). This is between 0x0 and 0xFFFF.

The start address points to the start of code when loaded into the MSP430 memory in the case of the first bootable data/program container for proper operation. For all the other containers loaded later, it may point outside the loaded destination address.

The trailer of the container provides two copies of the checksum that is based on the header and payload. The checksum is calculated using a word-wide XOR operation initialized with zero.

![Figure 9. Data/Program Container](image-url)
2.5 Failsafe Mechanism

A visible error signature is generated (see Figure 10) if no SPI device is connected or if the SPI power is not generated or if an error during user code load is detected. The voltage stabilization LED blinks three times with a frequency of approximately 1 Hz, the user's application code is not executed, and the device enters LPM4. This mechanism prevents execution of erroneous code.

Figure 10. Timing of Error Signature
2.6 API Functions of Loader

The API interface is implemented as a set of pointers to the core functions. This method lets the API core functionality be extended or changed without changing the access conventions. Although an application can call each function with an pointer to an absolute address, TI recommends using symbolic addressing to help simplify future migration of the application. The file loader.h provides the required definitions for assembler and ANSI-C.

Before calling the API functions, several peripherals must be initialized, and the API software registers must be unlocked and initialized. Perform the following steps to configure the clock, GPIO, Timer_A, and API software registers:

1. Configure the Compact Clock System:
   a. Unlock the Compact Clock System registers by writing the CCSKEY to the CCSCTL0 register.
   b. Clear any XOFG and HOFG faults by setting CCSCTL7 to 0.
   c. Clear any pending system interrupts by setting the SYSIFG1 register to 0.
   d. Set ACLK = SMCLK = MCLK = DCO at 1 MHz by setting CCSCTL4 and CCSCTL5 to 0.
2. Disable JTAG communication:
   a. Set P2SEL0 and P2SEL1 to 0
3. Configure Port 2 for SPI communication:
   a. P2.0, P2.1, P2.2, P2.3 are CS, MOSI, CLK, and MISO signals, respectively.
   b. Set CS high by setting bit 2 of P2OUT to 1.
   c. Set MOSI and CLK low by setting bits 1 and 2 of P2OUT to 0.
   d. Set CS, MOSI, and CLK pins to output by setting bits 0, 1, and 2 in P2DIR to 1.
   e. Set MISO to an input by setting bit 3 in P2DIR to 0.
4. Configure the API software registers:
   a. Unlock registers by setting the RAMLCK1 and RAMLCK0 bits in SYSCNF to 0.
   b. Set Status_Reg, CurOvlSpiH, and CurOvlAdrL to 0.
   c. Set CurOvlSpiL to 0x0002.
   d. Set LedOnPtr to user defined function for turning on power to the EEPROM (see Section 2.7.7).
   e. Set LedOffPtr to user defined function for turning power off to the EEPROM (see Section 2.7.6).
   f. Lock registers by setting the RAMLCK1 and RAMLCK0 bits in SYSCNF to 1.

Before calling any API function, software must unlock the API software registers using the process described in step 4a. If calling multiple API functions, it is common practice to unlock the API registers one time before calling the multiple functions and then locking them after all calls have completed. See the examples provided in MSP430Ware for MSP430x09x devices for the correct implementation of API initialization and function calls.
### Table 2. Public API Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Target Function Number</th>
<th>Stack Bytes Used</th>
<th>Vector Position</th>
<th>Context Save</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loader</td>
<td>Loader entry</td>
<td>–</td>
<td>2</td>
<td>0xF880</td>
<td>–</td>
</tr>
<tr>
<td>ApiCall</td>
<td>API call of target function</td>
<td>–</td>
<td>20 + TF&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>0xF882</td>
<td>16 bit</td>
</tr>
<tr>
<td>SWID</td>
<td>Software ID</td>
<td>0</td>
<td>2</td>
<td>0xF884</td>
<td>16 bit</td>
</tr>
<tr>
<td>LedOn</td>
<td>Turn on SPI boost voltage</td>
<td>1</td>
<td>2</td>
<td>0xF886</td>
<td>16 bit</td>
</tr>
<tr>
<td>LedOff</td>
<td>Turn off SPI boost voltage</td>
<td>2</td>
<td>2</td>
<td>0xF888</td>
<td>16 bit</td>
</tr>
<tr>
<td>XOVL</td>
<td>Load and execute overlay</td>
<td>3</td>
<td>36</td>
<td>0xF88A</td>
<td>16 bit</td>
</tr>
<tr>
<td>LOVL</td>
<td>Load overlay and continue at old</td>
<td>4</td>
<td>38</td>
<td>0xF88C</td>
<td>16 bit</td>
</tr>
<tr>
<td>COVL</td>
<td>Load overlay and execute as call</td>
<td>5</td>
<td>42</td>
<td>0xF88E</td>
<td>16 bit</td>
</tr>
<tr>
<td>ROVL</td>
<td>Load old overlay and resume execution</td>
<td>6</td>
<td>36</td>
<td>0xF890</td>
<td>16 bit</td>
</tr>
<tr>
<td>Spi.ReadByte</td>
<td>Read byte from SPI memory</td>
<td>7</td>
<td>18</td>
<td>0xF892</td>
<td>16 bit</td>
</tr>
<tr>
<td>Spi.ReadWord</td>
<td>Read word from SPI memory</td>
<td>8</td>
<td>20</td>
<td>0xF894</td>
<td>16 bit</td>
</tr>
<tr>
<td>SpiReadStream</td>
<td>Read stream from SPI memory</td>
<td>9</td>
<td>16</td>
<td>0xF896</td>
<td>16 bit</td>
</tr>
<tr>
<td>SpiStreamEnd</td>
<td>Terminate read stream from SPI</td>
<td>10</td>
<td>2</td>
<td>0xF898</td>
<td>16 bit</td>
</tr>
<tr>
<td>Spi.WriteByte</td>
<td>Write byte to SPI memory</td>
<td>11</td>
<td>22</td>
<td>0xF89A</td>
<td>16 bit</td>
</tr>
<tr>
<td>SpiReadWrite</td>
<td>Read/write swap with SPI memory</td>
<td>12</td>
<td>4</td>
<td>0xF89C</td>
<td>16 bit</td>
</tr>
<tr>
<td>CpyToSpi</td>
<td>Copy from MSP430 to SPI</td>
<td>13</td>
<td>30</td>
<td>0xF89E</td>
<td>16 bit</td>
</tr>
<tr>
<td>CpyTo430</td>
<td>Copy from SPI to MSP430</td>
<td>14</td>
<td>18</td>
<td>0xF8A0</td>
<td>16 bit</td>
</tr>
<tr>
<td>SpiGenerateImage</td>
<td>Generate bootable SPI memory image</td>
<td>15</td>
<td>40</td>
<td>0xF8A2</td>
<td>partly</td>
</tr>
<tr>
<td>CRC</td>
<td>CRC function</td>
<td>16</td>
<td>2</td>
<td>0xF8A4</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

<sup>(1)</sup> TF = Stack bytes used by target function
2.6.1  Loader(), Loader Entry Function

This function starts the loader application, which performs the tasks as shown in Figure 4. It is not a true function, as it does not return to its caller.

Assembler access absolute
CALL &0F880h
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &Loader

ANSI-C
void Loader(void);

2.6.2  ApiCall(), Low Convention API Call Function

This function may be used to call all of the other API functions with only one argument. All of the other arguments and return values are put into a memory structure. The pointer to that memory structure is then the only argument for ApiCall(). The content of the structure depends on the called target function and is described in each function description in the following sections.

The loader.h file also provides type definitions for these structures for each function callable by the ApiCall function.

Assembler access absolute
CALL &0F882h
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &ApiCall
...with R12.A pointing to the argument structure

ANSI-C
void ApiCall(unsigned short *args);
2.6.3  **SWID(), Software Identifier Function**

This function runs an integrity check and identifies the type and version of the software code. This function returns the value 0x28435000 as identifier for the first revision of the code and the value –1/0xFFFFFFFF if the integrity check fails.

The return value is composed of four fields:

- **Upper byte:** 0x28  IEEE identifier for Texas Instruments
- **Upper middle byte:** 0x43  Identifies MSP430
- **Lower middle byte:** 0x50  Identifies loader software (may vary depending on software revision)
- **Lower byte:** 0x00  Identifies revision code (may vary depending on software revision)

### Assembler access absolute

CALL &0F884h

(valid for MSP430L092 devices only)

### Assembler access symbolic

CALL &SWID

...with R12.W = 2843h and R13.W = 5000h as return value on success

### ANSI-C

```c
unsigned long SWID(void);
```

### For calls by ApiCall

```c
typedef struct {
   unsigned short SWIDNum; // Function number (here #0)
   unsigned short IdLow; // Reserved for returned low byte
   unsigned short IdHigh; // Reserved for returned high byte
} swid_api_args;
```
2.6.4  LedOn(), Turn LED On HAL Function

This function enables the clock on the port pin P1.2, which signals an external boost circuit to generate
the voltage for the SPI memory. In most circuits, an LED turns on as well as. For this API function, a
socket is implemented that lets the user provide a customized plug-in function (see Section 2.9).

Assembler access absolute
CALL &0F886h
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &LedOn

ANSI-C
void LedOn(void);

For calls by ApiCall
typedef struct
{
    unsigned short LedOnNum; // Function number (here #1)
}ledon_api_args;

2.6.5  LedOff(), Turn LED Off HAL Function

This function disables the clock on the port pin P1.2, which stops the voltage generation for the SPI
memory. In most circuits, an LED turns off as well as. For this API function, a socket is implemented that
lets the user provide a customized plug-in function (see Section 2.9).

Assembler access absolute
CALL &0F888h
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &LedOff

ANSI-C
void LedOff(void);

For calls by ApiCall
typedef struct
{
    unsigned short LedOffNum; // Function number (here #2)
}ledoff_api_args;
2.6.6 **XOVL(), Execute Code Overlay Function**

This function loads a code image into the internal MSP430 memory and invokes it after a successful load process.

On errors, it generates an error signature on port P1.2 as shown in Figure 10. The error signature is generated by using LedOn() and LedOff() and may, therefore, be generated on other ports, depending on the plug-in functions the user may have installed (see Section 2.9).

When ladr (alternate load address in MSP430 memory space) is not zero, ladr is used instead of the address given in the data image loaded. When sadr (alternate start address of code after load) is not zero, sadr is used instead of the address given in the data image loaded.

SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

XOVL() is not a true function, as it does not return to the caller.

Program execution on the new overlay continues at the same stack depth, as shown in Figure 11.

**Assembler access absolute**

CALL &0F88Ah

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &XOVL

**Arguments**

- R12.W for SPI address upper part
- R13.W for SPI address lower part
- R14.W for alternate loading address
- R15.W for alternate start address

**ANSI-C**

```c
void XOVL (unsigned short SpiAdrHigh, unsigned short SpiAdrLow,  
          unsigned short ladr, unsigned short sadr);
```

**For calls by ApiCall**

```c
typedef struct 
{
  unsigned short XOVLNum;   // Function number (here #3)
  unsigned short SpiAdrHigh; // SPI address high word
  unsigned short SpiAdrLow;  // SPI address low word
  unsigned short ladr;       // Alternate 430 load address when != 0
  unsigned short sadr;       // Alternate 430 start address when != 0
}xovl_api_args;
```

![Figure 11. Stack Behavior of XOVL()](image-url)
2.6.7 LOVL(), Load Code Overlay Function

This function loads a code image into the internal MSP430 memory and continues at the old overlay segment.

On errors, it generates an error signature on port P1.2 as shown in Figure 10. The error signature is generated by using LedOn() and LedOff() and may, therefore, be generated on other ports, depending on the plug-in functions the user may have installed (see Section 2.9).

When ladr (alternate load address in MSP430 memory space) is not zero, ladr is used instead of the address given in the data image loaded.

SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

The stack behavior is shown in Figure 12.

**Assembler access absolute**

CALL &0F88Ch

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &LOVL

Arguments

R12.W for SPI address upper part
R13.W for SPI address lower part
R14.W for alternate loading address

**ANSI-C**

void LOVL(unsigned short SpiAdrHigh, unsigned short SpiAdrLow, unsigned short ladr);

**For calls by ApiCall**

typedef struct
{
    unsigned short LOVLNum; // Function number (here #4)
    unsigned short SpiAdrHigh; // SPI address high word
    unsigned short SpiAdrLow; // SPI address low word
    unsigned short ladr; // Alternate 430 load address when != 0
}lovl_api_args;

![Figure 12. Stack Behavior of LOVL()](image-url)
### 2.6.8 COVL(), Call Code Overlay Function

This function stores the current code position (SPI address of overlay and position within overlay) on the stack, loads a code image as overlay into the internal MSP430 memory, and invokes it after a successful load process.

On errors, it generates an error signature on port P1.2 as shown in Figure 10. The error signature is generated by using LedOn() and LedOff() and may, therefore, be generated on other ports, depending on the plug-in functions the user may have installed (see Section 2.9).

When ladr (alternate load address in MSP430 memory space) is not zero, ladr is used instead of the address given in the data image loaded.

SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

The stack behavior is shown in Figure 13.

**Assembler access absolute**

CALL &0F88Eh

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &COVL

**Arguments**

R12.W for SPI address upper part  
R13.W for SPI address lower part  
R14.W for alternate loading address

**ANSI-C**

void COVL (unsigned short SpiAdrHigh, unsigned short SpiAdrLow, unsigned short ladr);

**For calls by ApiCall**

typedef struct
{
    unsigned short COVLNum; // Function number (here #5)  
    unsigned short SpiAdrHigh; // SPI address high word  
    unsigned short SpiAdrLow; // SPI address low word  
    unsigned short ladr; // Alternate 430 load address when != 0
} covl_api_args;

![Figure 13. Stack Behavior of COVL()](image-url)
2.6.9 ROVL(), Return and Resume Previous Code Overlay Function

This function terminates the code execution of the current overlay, loads the previous code image as an overlay into the internal MSP430 memory, and continues code execution where it was interrupted by the last COVL() function. SPI memory loading address of previous overlay and continuation address of that overlay is found on the stack.

This function performs the complementary operation to COVL() on the stack as shown in Figure 14.

Assembler access absolute
CALL &0F890h
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &ROVL

ANSI-C
void ROVL(void);

For calls by ApiCall
typedef struct
{
    unsigned short ROVLNum; // Function number (here #6)
}rovl_api_args;

Figure 14. Stack Behavior of ROVL()
2.6.10 SpiReadByte(), SPI Memory Read Byte Function

This function reads one byte of data from an SPI memory at location SpiAdr. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

**Assembler access absolute**

CALL &0F892h

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &SpiReadByte

**Arguments**

- R12.W for SPI address upper part
- R13.W for SPI address lower part

**Return Values**

- R12.B byte value read at SpiAdr

**ANSI-C**

```c
unsigned char SpiReadByte(unsigned short SpiAdrHigh, unsigned short SpiAdrLow);
```

**For calls by ApiCall**

```c
typedef struct
{
    unsigned short Spi.ReadByteNum; // Function number (here #7)

    union
    {
        unsigned short SpiAdrHigh; // SPI address high word
        unsigned char SpiRxData;    // Reserved for returned byte
    }

    unsigned short SpiAdrLow;     // SPI address low word
}spireadbyte_api_args;
```
2.6.11 SpiReadWord(), SPI Memory Read Word Function

This function reads one word of data from an SPI memory at location SpiAdr. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

**Assembler access absolute**

CALL &0F894h

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &SpiReadWord

**Arguments**

- R12.W for SPI address upper part
- R13.W for SPI address lower part

**Return Values**

- R12.B word value read at SpiAdr

**ANSI-C**

```c
unsigned short SpiReadWord(unsigned short SpiAdrHigh, unsigned short SpiAdrLow);
```

**For calls by ApiCall**

```c
typedef struct
{
    unsigned short SpiReadWordNum; // Function number (here #8)

    union
    {
        unsigned short SpiAdrHigh; // SPI address high word
        unsigned short SpiRxData; // Reserved for returned word
    };

    unsigned short SpiAdrLow; // SPI address low word
}spreadword_api_args;
```
2.6.12 SpiReadStream(), SPI Memory Read Stream Function

This function opens a read stream and returns the first byte of the stream from an SPI memory at location SpiAdr. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space. All the other elements of the stream are read using SpiXfer. A read stream is terminated with SpiStrEnd.

**Assembler access absolute**

CALL &0F986h

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &SpiReadStream

Arguments

R12.W for SPI address upper part
R13.W for SPI address lower part

Return Values

R12.B byte value read at SpiAdr

**ANSI-C**

unsigned char SpiReadStream (unsigned short SpiAdrHigh, unsigned short SpiAdrLow);

For calls by ApiCall

typedef struct
{
    unsigned short SpiReadStreamNum; // Function number (here #9)
    union
    {
        unsigned short SpiAdrHigh; // SPI address high word
        unsigned char SpiRxData; // Reserved for returned byte
    };
    unsigned short SpiAdrLow; // SPI address low word
}spireadstream_api_args;
2.6.13 SpiStreamEnd(), SPI Memory Read Stream End HAL Function

This function terminates a read stream from SPI memory. A read stream is opened using SpiRdStr. Elements of the read stream are received using SpiXfer.

Assembler access absolute
CALL &0F898h
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &SpiStreamEnd

ANSI-C
void SpiStreamEnd(void);

For calls by ApiCall
typedef struct
{ 
  unsigned short SpiStreamEndNum; // Function number (here #10)
}spistreamend_api_args;

2.6.14 SpiWriteByte(), SPI Byte Write-Function

This function writes one byte to an SPI device. The address in SPI memory space is given in SpiAdr. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

Assembler access absolute
CALL &0F89Ah
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &SpiWriteByte

Arguments
  R12.W for SPI address upper part
  R13.W for SPI address lower part
  R14.B for byte to be written to SPI address

ANSI-C
void SpiWriteByte(unsigned short SpiAdrHigh, unsigned short SpiAdrLow, unsigned char data);

For calls by ApiCall
typedef struct
{ 
  unsigned short SpiWriteByteNum; // Function number (here #11)
  unsigned short SpiAdrHigh; // SPI address high word
  unsigned short SpiAdrLow; // SPI address low word
  unsigned char SpiData; // Byte to write to SPI address
}spiwritebyte_api_args;
2.6.15 SpiReadWrite(), SPI Byte Read Write HAL Function

This function exchanges one byte with an SPI device. The data argument provided by the caller is transmitted to the SPI device, while the data pattern received from the SPI device during transmit is returned to the caller of this function. The chip select of the SPI device is not affected by this function.

NOTE: SPI devices may ignore some transmitted data depending on the internal state of the executed SPI command. See the data sheet of the SPI device for details.

Assembler access absolute
CALL &0F89Ch
(valid for MSP430L092 devices only)

Assembler access symbolic
CALL &SpiReadWrite

Arguments
- R12.B for byte data to be transmitted

Return Values
- R12.B byte value read during transmission

ANSI-C

unsigned char SpiReadWrite(unsigned char TxData);

For calls by ApiCall

typedef struct
{
    unsigned short SpiReadWriteNum; // Function number (here #12)
    union
    {
        unsigned char SpiTxData; // Byte to send to SPI device
        unsigned char SpiRxData; // Reserved for byte received from SPI device
    }
} spiReadWrite_api_args;
2.6.16 CpyToSpi(), Copy From MSP430 to SPI Function

This function copies a data block from MSP430 memory space to SPI memory space. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

**Assembler access absolute**

CALL &0F89Eh

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &CpyToSpi

**Arguments**

- R12.W for MSP430 source address
- R13.W for SPI address lower part
- R14.W for SPI address upper part
- R15.W length of block in bytes

**ANSI-C**

```c
void CpyToSpi(unsigned short SrcAdr, unsigned short SpiAdrHigh, unsigned short SpiAdrLow, unsigned short Count);
```

**For calls by ApiCall**

```c
typedef struct
{
    unsigned short CpyToSpiNum; // Function number (here #13)
    unsigned short SrcAdr; // Source address for MSP430
    unsigned short SpiAdrHigh; // SPI address high word
    unsigned short SpiAdrLow; // SPI address low word
    unsigned char Count; // Length in bytes
}cpytospi_api_args;
```
2.6.17 CpyTo430(), Copy SPI to MSP430 Function

This function copies a data block from SPI memory space to MSP430 memory space. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space.

Assembler access absolute

CALL &0F8A0h

(valid for MSP430L092 devices only)

Assembler access symbolic

CALL &CpyTo430

Arguments

R12.W for SPI address upper part
R13.W for SPI address lower part
R14.A for MSP430 destination address
R15.W length of block in bytes

ANSI-C

void CpyTo430(unsigned short SpiAdrHigh, unsigned short SpiAdrLow, unsigned short DstAdr, unsigned short Count);

For calls by ApiCall

typedef struct
{
    unsigned short CpyTo430Num; // Function number (here #14)
    unsigned short SpiAdrHigh; // SPI address high word
    unsigned short SpiAdrLow; // SPI address low word
    unsigned short DstAdr; // Destination address for MSP430
    unsigned char Count; // Length in bytes
}cpyto430_api_args;
2.6.18 SpiGenerateImage(), Generate Program Image in SPI Memory Utility Function

This function generates a bootable program image in SPI memory. SpiAdr uses the 16 least significant bits for SPI devices with up to 64KB of memory space and the 24 least significant bits for SPI devices with more than 64KB of memory space. This function is not a true function, as it does not return to the caller. This function stops with an LPM4 state.

**Assembler access absolute**

CALL &0F8A2h

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &SpiGenerateImage

**Arguments**

- R12.W for program address in MSP430 space
- R13.W for SPI destination address upper word
- R14.W for SPI destination address lower word
- R15.W length of block in bytes
- R11.W start address of code

**ANSI-C**

```c
void SpiGenerateImage(unsigned short SrcAdr, unsigned short SpiAdrHigh, unsigned short SpiAdrLow, unsigned short Count, unsigned short Sadr);
```

**For calls by ApiCall**

```c
typedef struct
{
    unsigned short SpiGenerateImageNum; // Function number (here #15)
    unsigned short SrcAdr; // Program address in MSP430 space
    unsigned short SpiAdrHigh; // SPI address high word
    unsigned short SpiAdrLow; // SPI address low word
    unsigned char Count; // Length in bytes
    unsigned short StartAdr; // Start address of code
}spigenerateimage_api_args;
```
2.6.19 CRC(), Calculate CRC

This function calculates the CRC over the given parameters and returns the calculated value.

**Assembler access absolute**

CALL &0F8A4h

(valid for MSP430L092 devices only)

**Assembler access symbolic**

CALL &CRC

**Arguments**

- R12.W for initial CRC seed
- R13.W for CRC start address
- R14.W for CRC end address

**Return Values**

- R12.W for calculated CRC value

**ANSI-C**

```
unsigned short CRC(unsigned short Seed, unsigned short StartAdr, unsigned short EndAdr);
```

**For calls by ApiCall**

```
typedef struct
{
    unsigned short CRCNum;  // Function number (here #16)

    union
    {
        unsigned short Seed;  // Initial CRC seed
        unsigned short CRCVal; // Reserved for returned CRC value
    };

    unsigned short StartAdr; // Address to start calculating CRC from
    unsigned short EndAdr;  // Address to stop calculating CRC
}
spireadword_api_args;
```
2.7 Software Registers and Public Data Elements

Table 3 lists the software registers. The software registers allow observation and operation mode control of the loader and its API for debug purposes and advanced API use.

Table 3. Software Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Acronym</th>
<th>Register Type</th>
<th>Register Access</th>
<th>Address (in L092)</th>
<th>Initial State</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current overlay SPI address</td>
<td>CurOvlSpiH</td>
<td>read/write</td>
<td>word</td>
<td>1C50h</td>
<td>0000h</td>
<td>Section 2.7.1</td>
</tr>
<tr>
<td></td>
<td>CurOvlSpiL</td>
<td>read/write</td>
<td>word</td>
<td>1C52h</td>
<td>0002h</td>
<td>Section 2.7.2</td>
</tr>
<tr>
<td>Current overlay MSP430 program address</td>
<td>CurOvlAdrH</td>
<td>read/write</td>
<td>word</td>
<td>1C54h</td>
<td>0000h</td>
<td>Section 2.7.3</td>
</tr>
<tr>
<td></td>
<td>CurOvlAdrL</td>
<td>read/write</td>
<td>word</td>
<td>1C56h</td>
<td>0000h</td>
<td>Section 2.7.4</td>
</tr>
<tr>
<td>Status register</td>
<td>Status_Reg</td>
<td>read/write</td>
<td>word</td>
<td>1C58h</td>
<td>0000h</td>
<td>Section 2.7.5</td>
</tr>
<tr>
<td>LED off function pointer</td>
<td>LedOffPtr</td>
<td>read/write</td>
<td>word</td>
<td>1C5Ah</td>
<td>Undefined</td>
<td>Section 2.7.6</td>
</tr>
<tr>
<td>LED on function pointer</td>
<td>LedOnPtr</td>
<td>read/write</td>
<td>word</td>
<td>1C5Ch</td>
<td>Undefined</td>
<td>Section 2.7.7</td>
</tr>
<tr>
<td>Unexpected interrupt count</td>
<td>UnexpCnt</td>
<td>read/write</td>
<td>word</td>
<td>1C5Eh</td>
<td>0000h</td>
<td>Section 2.7.8</td>
</tr>
</tbody>
</table>

Table 4 lists the secondary interrupt vectors. These vectors provide a vector field, similar to the INTVECS section, that allows a dynamic attachment of interrupt handlers on devices with a loader code.

Table 4. Secondary Interrupt Vectors

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Acronym</th>
<th>Register Type</th>
<th>Register Access</th>
<th>Address (in L092)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved vectors</td>
<td>RESERVED_xx_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C60h to 1C68h</td>
</tr>
<tr>
<td>P2IFG.0 to PGIFG.3 interrupt vector</td>
<td>PORT2_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C6Ah</td>
</tr>
<tr>
<td>TA0CCR1 CCIFG1 interrupt vector</td>
<td>TIMER0_A1_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C6Ch</td>
</tr>
<tr>
<td>TA0CCR0 CCIFG0 interrupt vector</td>
<td>TIMER0_A0_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C6Ah</td>
</tr>
<tr>
<td>P1IFG.0 to P1IFG.6 interrupt vector</td>
<td>PORT1_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C70h</td>
</tr>
<tr>
<td>CxIFG interrupt vector</td>
<td>APOOL_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C72h</td>
</tr>
<tr>
<td>WDT interrupt vector</td>
<td>WDT_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C74h</td>
</tr>
<tr>
<td>TA1CCR1 CCIFG1 interrupt vector</td>
<td>TIMER1_A1_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C76h</td>
</tr>
<tr>
<td>TA1CCR0 CCIFG0 interrupt vector</td>
<td>TIMER1_A0_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C78h</td>
</tr>
<tr>
<td>User NMI vector</td>
<td>UNMI_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C7Ah</td>
</tr>
<tr>
<td>System NMI vector</td>
<td>SYSNMI_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C7Ch</td>
</tr>
<tr>
<td>Reset vector</td>
<td>RST_IV2</td>
<td>read/write</td>
<td>word</td>
<td>1C7Eh</td>
</tr>
</tbody>
</table>
2.7.1 CurOvlSpiH, Current Overlay SPI Source Address High Register

**Figure 15. CurOvlSpiH Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>Reserved</td>
<td>RW</td>
<td>0h</td>
<td>Reserved. Reads as 0.</td>
</tr>
<tr>
<td>7-0</td>
<td>CurOvlSpiH</td>
<td>RW</td>
<td>0h</td>
<td>Bit [23:16] of the SPI address that the last data/program container was loaded from. This field is usually observed by debugging tools. This field is updated by the load image private function of the loader.</td>
</tr>
</tbody>
</table>

2.7.2 CurOvlSpiL, Current Overlay SPI Source Address Low Register

**Figure 16. CurOvlSpiL Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>CurOvlSpiL</td>
<td>RW</td>
<td>2h</td>
<td>Bit [15:0] of the SPI address that the last data/program container was loaded from. This field is usually observed by debugging tools. This field is updated by the load image private function of the loader.</td>
</tr>
</tbody>
</table>
2.7.3 CurOvlAdrH, Current Overlay Running Address High Register

**Figure 17. CurOvlAdrH Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-4</td>
<td>Reserved</td>
<td>RW</td>
<td>0h</td>
<td>Reserved. Reads as 0.</td>
</tr>
<tr>
<td>3-0</td>
<td>CurOvlAdrH</td>
<td>RW</td>
<td>0h</td>
<td>Bit ([19:16]) of the MSP430 address at which the current data/program container is executed (also see Section 2.7.4). This field is usually observed by debugging tools. This field is updated by the load image private function of the loader.</td>
</tr>
</tbody>
</table>

Table 7. CurOvlAdrH Register Description

2.7.4 CurOvlAdrL, Current Overlay Running Address Low Register

**Figure 18. CurOvlAdrL Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>CurOvlAdrL</td>
<td>RW</td>
<td>2h</td>
<td>Bit ([15:0]) of the MSP430 address at which current data/program container is executed (also see Section 2.7.3). This field is usually observed by debugging tools. This field is updated by the load image private function of the loader.</td>
</tr>
</tbody>
</table>
2.7.5 Status_Reg, Status Register

### Figure 19. Status_Reg Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-3</td>
<td>Reserved</td>
<td>RW</td>
<td>0h</td>
<td>Reserved. Reads as 0.</td>
</tr>
<tr>
<td>2</td>
<td>SpiSize</td>
<td>RW</td>
<td>0h</td>
<td>This bit indicates the address size of the SPI memory device detected. 0 = SPI memory device with 16-bit address size detected, 1 = SPI memory device with 24-bit address size detected.</td>
</tr>
<tr>
<td>1</td>
<td>DebStp</td>
<td>RW</td>
<td>0h</td>
<td>This bit controls the execution flow for debugging purposes. 0 = Program execution may be continued after data/program container load, 1 = Program execution is stopped after data/program container load and LPM4 is entered.</td>
</tr>
<tr>
<td>0</td>
<td>LedState</td>
<td>RW</td>
<td>0h</td>
<td>This bit reflects the state of the external boost circuit for SPI device supply. This bit is controlled by LedOn() and LedOff() or their custom plug-ins. 0 = Boost circuit is off (LED is off), 1 = Boost circuit is on (LED is on).</td>
</tr>
</tbody>
</table>
2.7.6 LedOffPtr, Pointer to LedOff() Function Register

Figure 20. LedOffPtr Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>LedOffPtr</td>
<td>RW</td>
<td>F800h</td>
<td>This field points to the location where the default LedOff() function is located. This field is initialized during loader startup with the address of the LedOff() function of the loader itself. This location is used to redirect to a customer's variant of a LedOff() function as a plug-in.</td>
</tr>
</tbody>
</table>

Table 10. LedOffPtr Register Description

2.7.7 LedOnPtr, Pointer to LedOn() Function Register

Figure 21. LedOnPtr Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>LedOnPtr</td>
<td>RW</td>
<td>F800h</td>
<td>This field points to the location where the default LedOn() function is located. This field is initialized during loader startup with the address of the LedOn() function of the loader itself. This location is used to redirect to a customer's variant of a LedOn() function as a plug-in.</td>
</tr>
</tbody>
</table>

Table 11. LedOnPtr Register Description

2.7.8 UnexpCnt, Unexpected Interrupt Count Register

Figure 22. UnexpCnt Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>UnexpCnt</td>
<td>RW</td>
<td>0h</td>
<td>This field reflects the number of unexpected interrupts counted by the dummy interrupt handler. By default, all possible interrupts are terminated after startup. This value can be cleared by the debugging tool after the user application is loaded and initialized. If this field is incremented, then at least one interrupt source is not covered by the user application.</td>
</tr>
</tbody>
</table>

Table 12. UnexpCnt Register Description
2.8 Interrupt Handling

The loader provides a method that lets an application program use all of the interrupt resources of a device. A hardware interrupt causes the interrupt service routine at which the corresponding interrupt vector is pointing to be run. In the case of the loader, a simple instruction (called a software stub or SW-stub) is placed in that location and forwards control to the interrupt service routine at which the secondary interrupt vector is pointing. The secondary interrupt vector is a software element in RAM that points to the user’s interrupt service routine (see Figure 23). Such SW-stubs and secondary interrupt vectors are implemented for all interrupt sources. The secondary interrupt vectors are initialized to point to a dummy interrupt handler. This ensures that all interrupts, even unexpected ones, are terminated before the user application takes control. The dummy interrupt handler counts the number of unexpected interrupts (see Section 2.7.8).

There are slight differences between the L092 interrupt handling behavior and the C091/C092 behavior.

- The interrupt response of the L092 takes four cycles longer due to the redirection by the SW-stub.
- Unexpected interrupts are always terminated on the L092. If the user code does not terminate the interrupts, the default L092 loader terminates them. The C091 and C092 do not terminate interrupts, so the user code on these devices must handle all interrupts.

NOTE: It is strongly recommended that production applications terminate all interrupt vectors.

![Figure 23. Secondary Interrupt Vectors](image)

2.9 Sockets and Plug-Ins

The user can provide optimized functions to generate the signals for the boost converter by replacing the default LedOn() and LedOff() functions that are supplied with the loader. To use custom functions:

1. Load the custom LedOn() and LedOff() functions into the MSP430 internal memory.
2. Set the function pointer registers, LedOnPtr (see Section 2.7.7) and LedOffPtr (see Section 2.7.6), to the addresses of the custom functions.

The custom LedOn() and LedOff() functions are now used by the API core functions.

2.10 Power Efficient Program Loading

The original LedOn() and LedOff() functions generate a signal of 250 kHz with an 50% duty cycle. This signal allows a variety of charge pumps to be used; however, this signal might not be the most power-efficient one. An application might need a 125-kHz signal with a duty cycle of 25%, because this is the optimum setting for the inductor used in the charge pump. A two-step boot approach can be used to save power. In Figure 24 the first boot portion “B1e” installs own LedOn() or simply changes the duty cycle of the signal generator, and the secondary boot process called with XOVL() then loads the real application with a significantly reduced power budget.
2.11 Programming With Overlays

The loader API allows the use of applications that are split into overlay sections. The functions XOVL() (execute overlay), COVL() (call overlay), and ROVL() (return overlay), allow hierarchical program structures using overlaid code. Figure 25 shows a complex application using the overlay mechanisms.

![Figure 24. Two-Level Boot Approach](image)

![Figure 25. Complex Application With Three Overlays Segments](image)
2.12 Debugging Checkpoints for Code Development

Three checkpoints are implemented to ease debugging of code for regular and overlaid code (see Figure 26).

Checkpoint CP1
This checkpoint is implemented inside the SUC. The execution is stopped when the application password is found in the JTAG mailbox. The device enters LPM4 and waits for a debugger to start a session. This stop is done before application execution on C091 and C092 and before loader execution on L092.

Checkpoint CP2
This checkpoint is implemented in the loader core. The execution is stopped when the application password is found in the JTAG mailbox. The device enters LPM4 and waits for an debugger to start a session. This stop is done before application execution on L092.

Checkpoint CPx
This checkpoint is implemented inside the loader API. The execution is stopped when OpMode1 is set. This stop is done after overlay load and before program continuation. This stop is done before application execution on L092. The device enters LPM4 and waits for a debugger to continue the debug session.

![Figure 26. Debugging Checkpoints](image-url)
2.13 **Inner Mechanism of Loader**

Figure 27 shows the inner mechanism of loader API, start-up code, and loader core.

![Diagram showing the inner mechanism of loader API, start-up code, and loader core.]

2.14 **SPI Commands Used by Loader**

MOSFET SPI memory devices share a common command set. The loader uses only the common command set; special device-dependent commands are not used.

<table>
<thead>
<tr>
<th>SPI Command</th>
<th>Code</th>
<th>EEPROMS</th>
<th>Flash</th>
<th>FRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read status register</td>
<td>0x05</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
<tr>
<td>Write status register</td>
<td>0x01</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
<tr>
<td>Write Enable</td>
<td>0x06</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
<tr>
<td>Read memory</td>
<td>0x03</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
<tr>
<td>Write memory</td>
<td>0x02</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
<tr>
<td>Bulk Erase</td>
<td>0xC7</td>
<td>ignored (exp.)</td>
<td>used</td>
<td>ignored (exp.)</td>
</tr>
<tr>
<td>Read Stream</td>
<td>0x03</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
</tbody>
</table>
3 Target Hardware

Devices with the loader like the MSP430L092 require a target hardware to operate. Figure 2 shown in the introduction is such a target hardware optimized for a particular device. A more generic block diagram for such a target hardware is shown in Figure 28. It is the user's choice to select one of the proposed SPI device voltage supply booster circuits and adapter networks or to develop different circuits. It is also the user's choice to select the type of SPI memory device used.

![Generic Block Diagram of Target Hardware](image)

3.1 Booster Converters

The circuits in Figure 29 through Figure 34 represent a variety of booster circuits that have been verified and can be used to generate SPI device supply voltages from 1.9 V to 6 V.

Table 14 lists the values of the components shown in the following figures.

<table>
<thead>
<tr>
<th>Component</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1 kΩ</td>
</tr>
<tr>
<td>R2</td>
<td>47 kΩ</td>
</tr>
<tr>
<td>C1</td>
<td>330 nF</td>
</tr>
<tr>
<td>C2</td>
<td>330 nF</td>
</tr>
<tr>
<td>C3</td>
<td>10 nF</td>
</tr>
<tr>
<td>D1</td>
<td>1N4148</td>
</tr>
<tr>
<td>D2</td>
<td>1N4148</td>
</tr>
<tr>
<td>L</td>
<td>33 μH/160 mA</td>
</tr>
<tr>
<td>Q</td>
<td>BC807/BC817</td>
</tr>
</tbody>
</table>

Table 14. Values of the Components
0.9-V supply
C1
P1.2
R1
R2

Orange on EPCOS
PQ2 330 K02
33 µH / 160 (1.17=>1.89)

Figure 29. Booster Converter Type A

0.9-V supply
C1
P1.2
R1
R2

Yellow on Coilcraft
0805PS-333KL
33 µH / 160 (1.17=>3.19)

Figure 30. Booster Converter Type B

0.9-V supply
C1
P1.2
R1
R2

Red on Murata
LQH2MCN330K02
33 µH / 160 (1.17=>2.79)

Figure 31. Booster Converter Type C

0.9-V supply
C1
P1.2
R1
R2

Green on EPCOS
SIMID 1210-100
33 µH / 105 (1.17=>3.80)

Figure 32. Booster Converter Type D

0.9-V supply
C1
P1.2
R1
R2

Blue on EPCOS
SIMID 1210-T
33 µH / 70 (1.17=>5.98)

Figure 33. Booster Converter Type E
3.2 Adaptation Networks

The circuits in Figure 35 through Figure 38 represent a variety of adaptation networks circuits suitable for level adaptation for an SPI device being supplied from 1.8 V to 6 V.
Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from March 26, 2011 to July 18, 2018

<table>
<thead>
<tr>
<th>Changes</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Added the paragraph that begins &quot;Before calling the API functions...&quot; and the following steps and paragraph in Section 2.6, API Functions of Loader</td>
<td>12</td>
</tr>
<tr>
<td>• Removed former &quot;Calling Method&quot; column from Table 2, Public API Functions</td>
<td>13</td>
</tr>
<tr>
<td>• Corrected the &quot;Vector Position&quot; value for all rows from &quot;LedOff&quot; to the end of Table 2, Public API Functions</td>
<td>13</td>
</tr>
<tr>
<td>• Changed ANSI-C example in Section 2.6.1, Loader(), Loader Entry Function</td>
<td>14</td>
</tr>
<tr>
<td>• Added the second paragraph in Section 2.6.2, ApiCall(), Low Convention API Call Function</td>
<td>14</td>
</tr>
<tr>
<td>• Changed ANSI-C example in Section 2.6.2, ApiCall(), Low Convention API Call Function</td>
<td>14</td>
</tr>
<tr>
<td>• Changed all examples from &quot;Assembler access symbolic&quot; to the end of section and deleted &quot;Returned structure&quot; in Section 2.6.3, SWID(), Software Identifier Function</td>
<td>15</td>
</tr>
<tr>
<td>• Updated ANSI-C and ApiCall examples in Section 2.6.4, LedOn(), Turn LED On HAL Function</td>
<td>16</td>
</tr>
<tr>
<td>• Updated ANSI-C and ApiCall examples in Section 2.6.5, LedOff(), Turn LED Off HAL Function</td>
<td>16</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall in Section 2.6.6, XOVL(), Execute Code Overlay Function</td>
<td>17</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall in Section 2.6.7, LOVL(), Load Code Overlay Function</td>
<td>18</td>
</tr>
<tr>
<td>• Deleted sadr (alternate start address) in Section 2.6.8, COVL(), Call Code Overlay Function</td>
<td>19</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall in Section 2.6.8, COVL(), Call Code Overlay Function</td>
<td>19</td>
</tr>
<tr>
<td>• Updated for ANSI-C and ApiCall in Section 2.6.9, ROVL(), Return and Resume Previous Code Overlay Function</td>
<td>20</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall and deleted Returned Structure in Section 2.6.10, Spi.ReadByte(), SPI Memory Read Byte Function</td>
<td>21</td>
</tr>
<tr>
<td>• Added Section 2.6.11, SpiReadWord(), SPI Memory Read Word Function</td>
<td>22</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall and deleted Returned Structure in Section 2.6.12, SpiReadStream(), SPI Memory Read Stream Function</td>
<td>23</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall in Section 2.6.13, SpiStreamEnd(), SPI Memory Read Stream End HAL Function</td>
<td>24</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall in Section 2.6.14, Spi.WriteByte(), SPI Byte Write-Function</td>
<td>24</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall and removed Returned Structure in Section 2.6.15, Spi.WriteByte(), SPI Byte Write Read HAL Function</td>
<td>25</td>
</tr>
<tr>
<td>• Added ANSI-C, and ApiCall and removed Returned Structure in Section 2.6.16, CpyToSpi(), Copy from MSP430 to SPI Function</td>
<td>26</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall and removed Returned Structure in Section 2.6.17, CpyTo430(), Copy SPI to MSP430 Function</td>
<td>27</td>
</tr>
<tr>
<td>• Updated examples in assembler, ANSI-C, and ApiCall and removed Returned Structure in Section 2.6.18, SpiGenerateImage(), Generate Program Image in SPI Memory Utility Function</td>
<td>28</td>
</tr>
<tr>
<td>• Added Section 2.6.19, CRC(), Calculate CRC</td>
<td>29</td>
</tr>
<tr>
<td>• Changed Operation mode to Status register and changed Initial State of LED off function pointer and LED on function pointer in Table 3, Software Registers</td>
<td>30</td>
</tr>
<tr>
<td>• Updates throughout Table 4, Secondary Interrupt Vectors</td>
<td>30</td>
</tr>
<tr>
<td>• Changed title of Section 2.7.5, Status_Reg, Status Register</td>
<td>33</td>
</tr>
<tr>
<td>• Changed Figure 29</td>
<td>40</td>
</tr>
</tbody>
</table>
IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated (‘TI”) technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, “TI Resources”) are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI’s provision of TI Resources does not expand or otherwise alter TI’s applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT. AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED “AS IS” AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include, without limitation, TI’s standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2018, Texas Instruments Incorporated