Software for CC1100/CC2500 and MSP430

Examples and Function Library User's Guide

By Magnus Wines

Abstract

The "Software for CC1100/CC2500 and MSP430 - Examples and Function Library" is the recommended starting point for developing software for the CC1100/CC2500 and MSP430 combination. The software demonstrates how to use some of the many advanced digital features and packet handling capabilities of the CC1100/CC2500. The software examples have been developed for the MSP430 Experimenter's Board, but can easily be ported to another hardware platform.

Table of contents

1 Introduction

This application note describes different ways to interface and use the TI Chipcon RF modules CC1100 and CC2500 with an MSP430. The accompanying software contains a function library allowing quick prototyping of radio protocols and wireless applications. A set of examples, running out-of-the-box on the MSP430 Experimenter's Board, is also included.

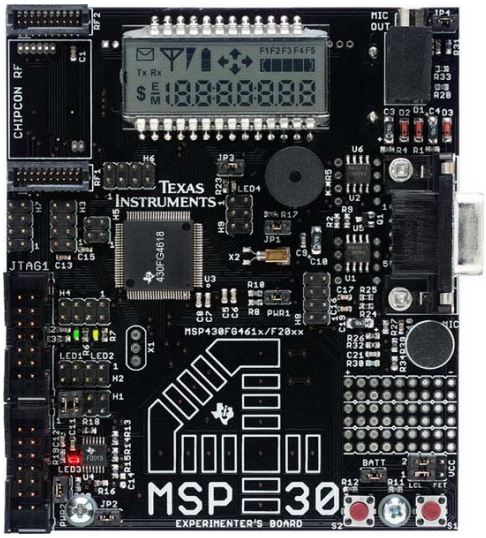

Figure 1 MSP430 Experimenter's Board

The MSP430 Experimenter's Board from Texas Instruments (see [8]) features selected MSP430 devices and additional hardware components for easy system evaluation and prototyping. It is an ideal platform for learning a new architecture or testing the capabilities of a device.

The TI Chipcon CC100/CC2500 low power RF transceivers are market-leading RF devices operating in the ISM band (Industrial, Scientific, and Medical) and they are an optimal match for the MSP430 ultra low power microcontrollers.

The software provided in this library is a starting point for developers wanting to get the most out of the MSP430 and the Chipcon RF devices. It shows how to communicate with the CC1100/CC2500EM using the SPI interface and how to use various features of the transceivers for packet transmission and reception.

After reading this document, you will be familiar with

- Different ways to send and receive packets using the Chipcon RF transceivers.
- Different ways to combine RF functions with sleep modes and interrupt service routines on the MSP430.

2 Using the software

2.1 Prerequisites

To successfully compile, download and run the software described in this document, the following material is needed

- 2 MSP430 Experimenter's Boards.
- 2 CC1100EMs or CC2500EMs with appropriate antennas.
- An MSP430 Debug Interface (e.g. MSP430-FET430UIF).
- IAR Embedded Workbench for MSP430.
- 4 AAA batteries

A free, code size limited, but fully functional edition of IAR Embedded Workbench (IAR Kickstart) is available from the IAR Systems website (www.iar.com) or from the TI MSP430 homepage www.ti.com/msp430.

As an alternative to IAR Embedded Workbench, it would be possible to use Code Composer 2.0. Minor modifications of the software handling interrupts would be necessary. A trial edition of Code Composer is available from the TI MSP430 homepage.

2.2 Getting started

Follow these simple steps to get your application up and running:

1. Install IAR Workbench.
2. Download the source code for this application note and unzip the files to your working directory.
3. Open IAR Embedded Workbench and create a new project:
    a. Select Project -> Create new project…
    b. Select tool chain MSP430.
    c. Base the project on the empty project template.
    d. Save (you will also be asked to save the current workspace).
4. Add the following C files from the software that you downloaded and unzipped in step 2:
    - All C files from the **hal/hal_exp430** directory
    - All C files from the **hal/common** directory
    - All C files from *one* of the sub-directories in the **app** directory
5. Modify the file "my_rf_settings.h", which should be located in the directory of the application you have chosen. The file is set up with register settings for the CC2500. If you have another chip, operating at another frequency band, use SmartRF Studio to generate the register settings appropriate for your application. Use the code export facility of SmartRF Studio to get an initialized version of the HAL_RF_CONFIG structure.
6. Open the "options…" dialog for the new project by right clicking the project name in the workspace window. A window should appear.
    a. Under "General options", select the MSP device. For the Experimenter Board, use MSP430FG4618.
    b. Under "C/C++ compiler", click on the "Preprocessor" pane.

i. The “Ignore standard include directories” tick box should *not* be ticked.

ii. In the “Additional include directories”, add include paths telling the compiler where to find the header files included by the C files. You should add the **hal\include**, **hal\common** and **hal\hal_exp430** directories.[1]

c. Under “Debugger”, select “FET Debugger” from the “Driver” drop down list.

d. Under “FET Debugger”, in the “Connection” section, choose the connection type of the FET tool (e.g. Texas Instruments USB-IF). Leave the rest of the settings as is.

7. Close the options window.
8. Select Project -> Rebuild All. There should be no errors or warnings when IAR builds the executables.
9. Attach the CC1100EM or CC2500EM to the MSP430 Experimenter Board.
10. Attach the MSP430 FET to your PC. If you are running Windows and using the USB FET tool for the first time, you will be asked to install some drivers for the tool. For Windows XP, they are located in $IAR_INSTALL_DIR$\430\drivers\TIUSBFET\WinXP.
11. Attach the MSP430 FET to the MSP430 Experimenter Board using the JTAG1 connector. The power select jumpers should be set according to the description in [9]. The figure below shows one possible jumper position where the board is powered from the FET alone.

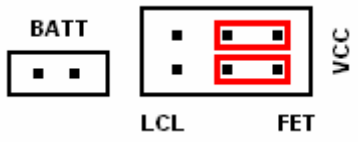

Figure 2 Jumper position when board powered from FET

12. Select Project -> Debug. IAR will now establish a connection with the target MCU, download the application and program the MSP430. The debugger will be started, halting the target at main().
13. Start the application by selecting Debug -> Go. The LCD should display the part and revision number of the EM (e.g. 0003 for a CC1100EM and 8003 for a CC2500EM).
14. Stop debugging by selecting Debug -> Stop Debugging.
15. The unit can now be operated independently from the debugger by disconnecting the FET tool and using the AAA batteries as power source. Attach the BATT jumper (position of VCC jumpers irrelevant)
16. Repeat steps 8 through 14 for the other board.
17. Depending on the application you are running, you now have two boards capable of transmitting and receiving packets. See the subsequent chapters for an explanation of the different applications.

[1] You may use the variable $PROJ_DIR$ in the path to specify the directory of the current project.

3 Software Library Reference

The design of the software in this package is based on the layered architecture as depicted in Figure 3.

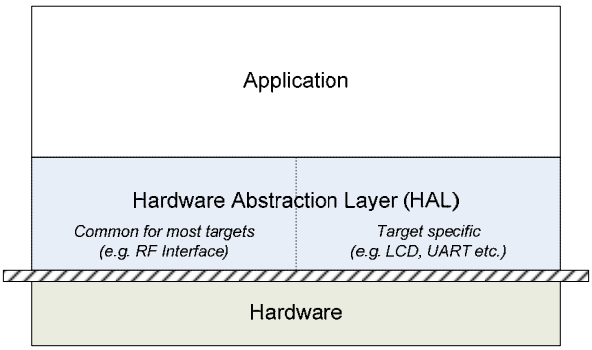

Figure 3 Architecture

The Application Layer contains example code for transmitting and receiving packets.

The Hardware Abstraction Layer (HAL) is divided in two parts: The common part (common) and the target specific part (hal_exp430). The common part contains software that is portable within a large range of hardware platforms, and covers the functions for accessing the CC1100 and CC2500 RF ICs and generic SPI functions. The common part contains most of the functions needed for successfully developing applications and experimenting with the RF devices. The target specific part of HAL contains the "glue" software needed for making it possible to run software on that particular platform. It contains functions to interact with the board's user interface, such as digital I/O, LEDs, LCD panels and UART.

The HAL has a generic, board independent interface, defined by the header files in the include directory.

The target specific part can easily be replaced such that other hardware platforms can be used. The software includes a HAL template that can be used when developing a HAL for a new target.

The layered software architecture is reflected in the directory structure of the source code.

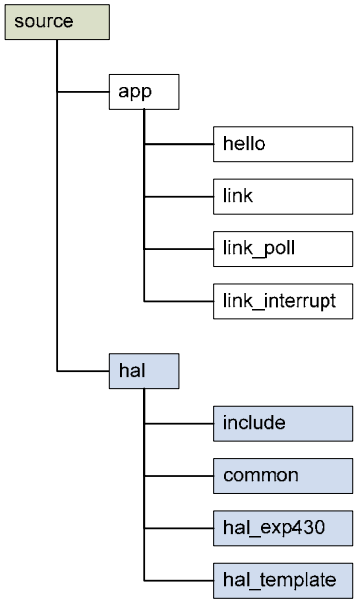

Figure 4 Software organization

3.1 Application Examples

The software for the experimenter's board is bundled with a number of example applications demonstrating how to use the various features of CC1100 or CC2500 to send and receive packets. The following examples are provided:

"hello"	**"**Hello world" example. Displays the CC1100/CC2500 part number and revision number on the LCD and writes a string to the UART.
"link"	Transmit or receive packets with variable length, but shorter than the size of the CC1100/CC2500 packet FIFO.
"link_poll"	Transmit or receive variable length packets (up to 255 bytes) using state information from chip.
"link_interrupt"	Transmit or receive variable length packets (up to 255 bytes) using advanced features of the CC1100/CC2500 FIFOs.

Details about the three link examples can be found in the subsequent paragraphs.

Please note that the intention of these examples is to show how various features of the RF chip can be used to send and receive data. Thus, a unit operates either as a transmitter or as a receiver. It is left to the user to combine these features to create full scale RF protocols with units being able to both send and receive data.

3.1.1 "link" test application

This application demonstrates how to set up a simple RF link between two units. One board will operate as the transmitter and the other board will operate as the receiver. Select mode of operation by pushing button S1 (receiver) or S2 (transmitter). The transmitter will send one packet every time one of the buttons is pushed. In this example, the packet is smaller than the size of the internal FIFO on the RF chip, so the entire packet is written at once. The receiver assumes the same, thus it waits until the chip signals that a complete packet is received before starting to read from the FIFO.

The number of packets transmitted and received is shown on the LCD display on the TX and RX unit, respectively. On the receiver side, several error conditions (FIFO overflow, illegal packet length and CRC error) are checked before updating the display.

The figure below shows the program flow both when the unit is operating in transmit mode and when it is operating in receive mode.

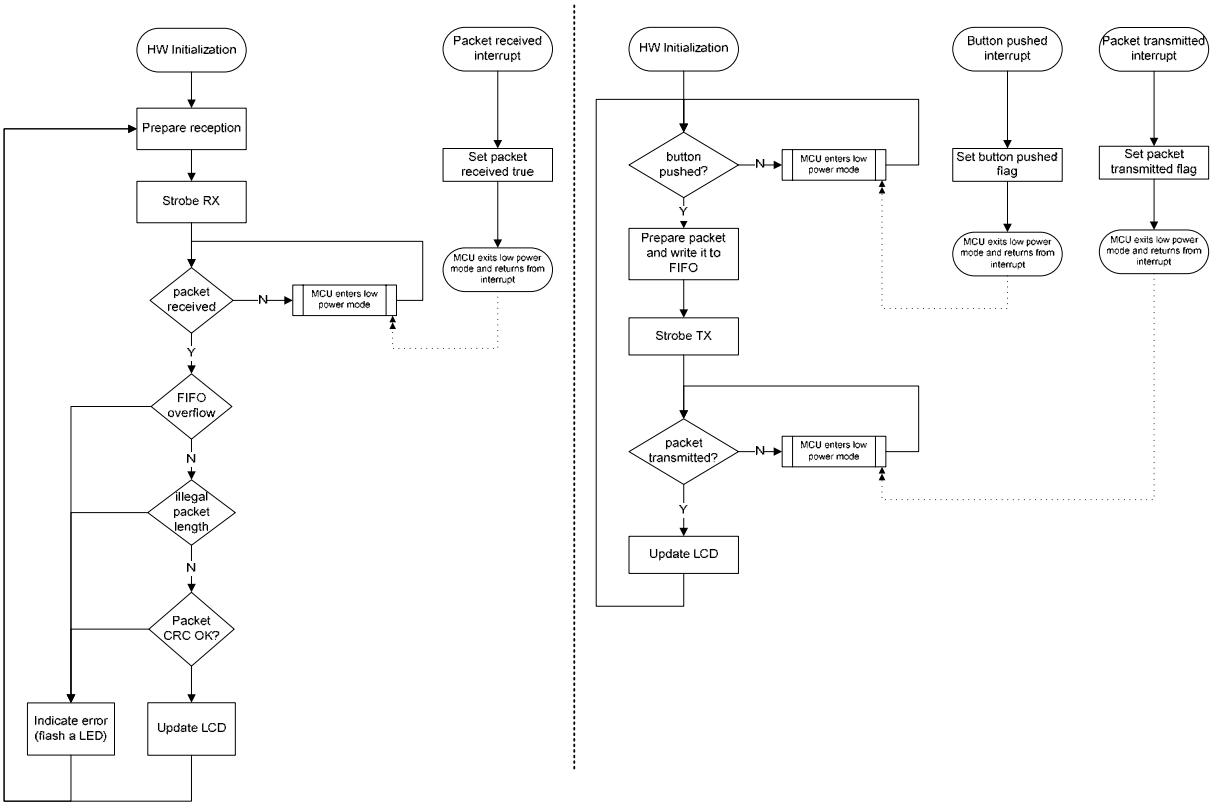

Figure 5 Receiving and transmitting packets

Note that the MCU always goes into a low power state while waiting for an event. The interrupt service routine will wake up the MCU, making it run normally after exiting the interrupt context.

3.1.2 "link_poll" test application

This application demonstrates how it is possible to send packets up to 255 bytes relying exclusively on the status byte returned by the RF chip. The status byte is returned whenever the chip is addressed or when sending a command (strobe) to the chip. The status byte is explained in detail in [1] and [2] (section 10.1).

NB! When reading a status register over the SPI interface while the register is updated by the radio hardware (e.g. MARCSTATE or TXBYTES), there is a small, but finite, probability that a single read from the register is being corrupt. This also applies to the chip status byte. [3] and [4] explain the problem and propose several workarounds. In this application example, the status byte is read repeatedly until getting the same value twice.

As for the "link" example, one of the boards is set in RX and the other board is set in TX by pushing either S1 (RX) or S2 (TX). On the transmitter unit, a packet (with arbitrary length from 2 to 255 bytes) [2] will be posted and the MCU waits until the packet has been written to the FIFO before transmitting a new one. The actual writing of the packet is performed by a timer interrupt service routine (ISR), scheduled to run every 200 microseconds (approximately). The ISR starts by checking the state of the radio chip. If the chip is in one of the states IDLE, CALIBRATE or TX, and if there is space available in the FIFO, the ISR writes data from the posted packet to the FIFO. Once the complete packet is written to the FIFO, the MCU is notified. The ISR runs at a rate such that a TX underflow does not occur - i.e. avoiding an empty FIFO as long as the packet has not been transmitted.

On the receiver side, the same state polling mechanism is used. Every 200 microseconds, the timer ISR will check the state of the radio. If it is either in IDLE or RECEIVE state, data from the FIFO is read and appended to the current packet buffer. This continues until a complete packet has arrived, at which point the ISR sets a global flag indicating completion. The main routine polls this flag, and returns the packet to the caller once it is set.

An element that slightly complicates the software example is that the FIFO cannot be emptied while the chip is writing data to it. The RX FIFO problem is described in [3] and [4]. The workaround used in this example is to always make sure there is at least one more byte in the FIFO unless the complete packet has been received (and the chip is no longer in RX state).

The figures on the following page illustrate the program flow on the receiver and transmitter side.

[2] Length includes payload and length byte

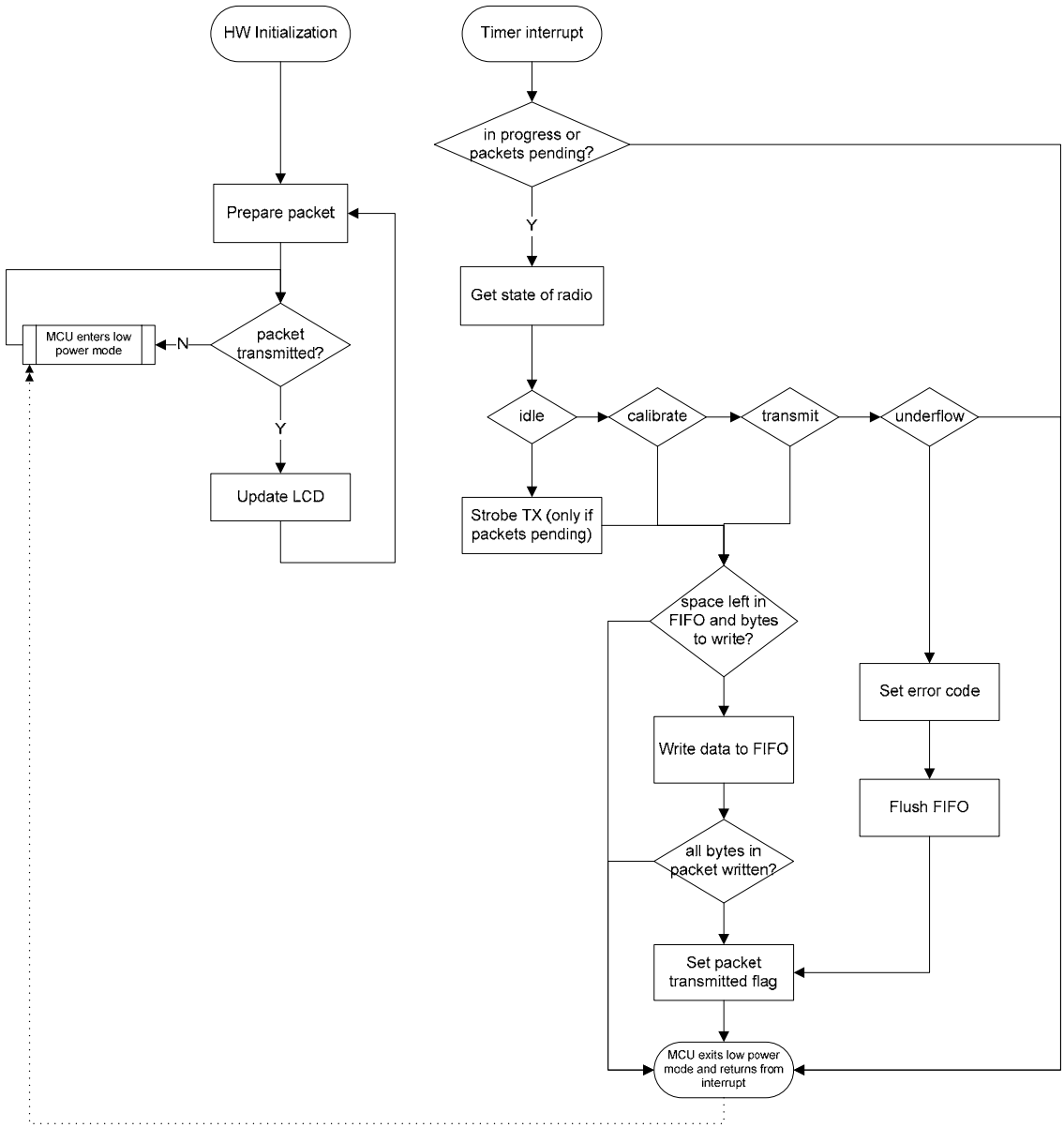

Figure 6 Transmitting packets in the "link_poll" example

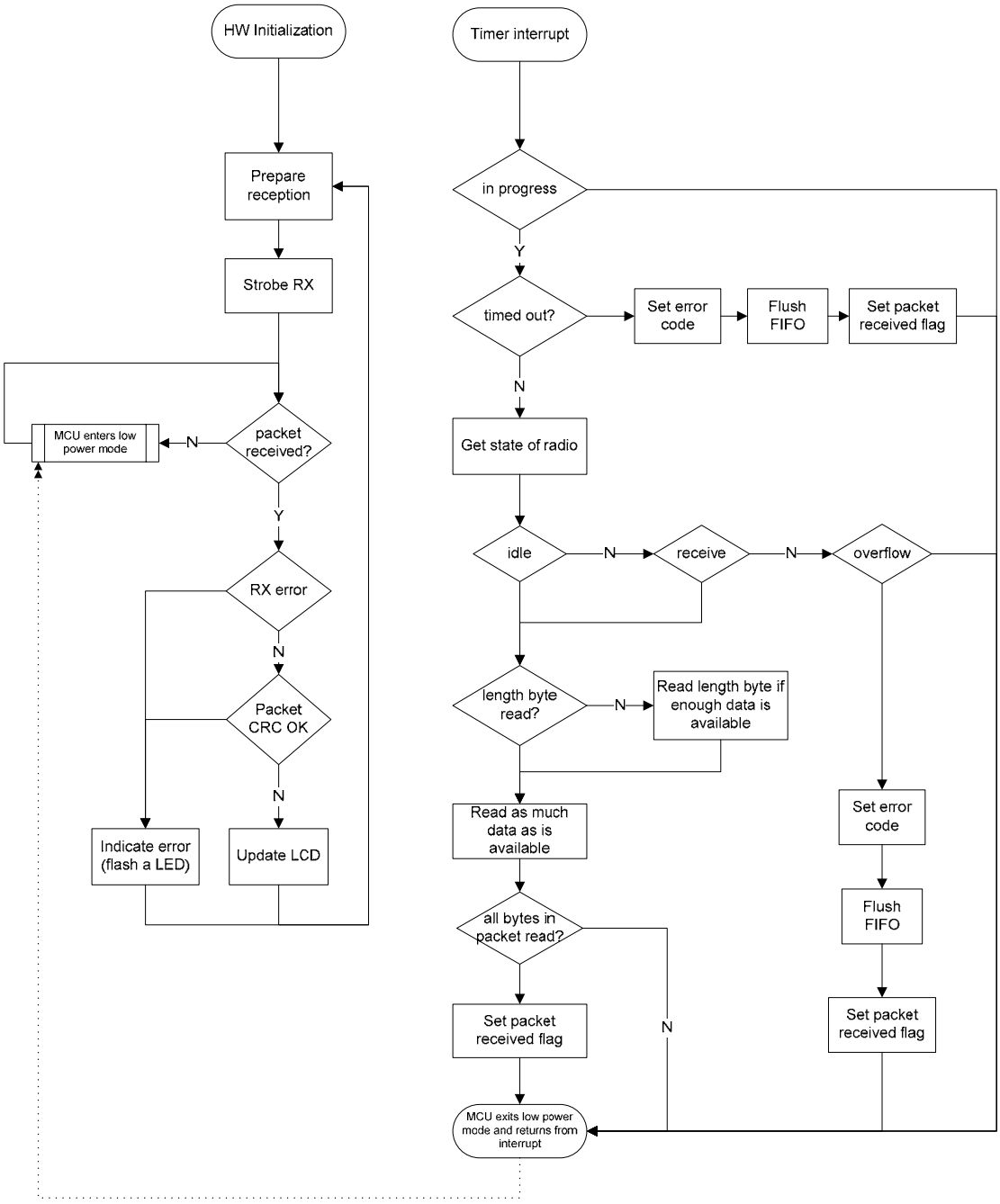

Figure 7 Receiving packets in the “link_poll” example

3.1.3 "link_interrupt" test application

This application demonstrates how it is possible to use the digital output signals from the CC1100 and CC2500 for efficient transmission and reception of packets up to 255 bytes. For details about I/O pin usage, see [7].

The board will act as a receiver or transmitter by pressing button S1 (RX) or S2 (TX) respectively. Unlike the "link_poll" example, this application uses two events generated by the chip to figure out whether data can be written to or read from the FIFO.

The main event is *"packet received/transmitted"*. The RF chip is configured to generate a signal that is set high (1) when the radio starts receiving data or transmitting data. The signal is set low (0) when the complete packet is either received or transmitted. The software configures the MCU to generate an interrupt on a falling edge of the signal.

The other event is *"FIFO half full"*. The RF chip is configured to generate a signal that is set high (1) when there are more than 32 bytes in the FIFO and low (0) when there are less than 33 bytes in the FIFO. In RX mode, an interrupt is generated on a rising edge of the signal (the buffer is being filled up), whereas in transmit mode, an interrupt is generated on a falling edge of the signal (the buffer is being emptied).

When transmitting, the MCU may send new packets as soon as the previous packet has been written to the FIFO, i.e. it does not wait for the packet to be sent over the air. The MCU keeps track of how many bytes it has written to the FIFO and will never attempt to write if the FIFO is full.

In reception mode, the board will set the radio in RX mode and wait until one of two events occurs: Either, the FIFO is half full (and)/or a complete packet is received. In the first case, and if the length byte has not already been read, read the first byte of the packet, which should indicate the length of the incoming packet. Then read all available bytes in the FIFO (but avoid emptying the FIFO, due to the RX FIFO problem mentioned in the previous example). Go back to sleep and wait for a new event. If the event indicates reception of a complete packet, read (the rest of) the packet and check for eventual CRC errors.

Note on the threshold value: In this example, we have chosen a value for the threshold such that we get an interrupt when the FIFO is half full. The value to use in "real life" should be adapted to the specific needs of the application. The developer should consider the following:

	Interrupt on many bytes in FIFO (a few bytes free)	**Interrupt on a few bytes in FIFO (many bytes free)**
Transmit	The MCU wakes up just being able to write a small amount of data to the FIFO. Efficient when only a few bytes of an outgoing packet remain, letting the transmit function finish earlier.	The MCU can write a large amount of data to the FIFO, but will have to wait for an unnecessarily long time if only a few bytes of a packet remain. Timing becomes critical: If not additional data is written to the FIFO quickly, you might get a TX underflow.
Receive	The MCU can read many bytes from FIFO, thus avoiding waking up the MCU just to do almost nothing. Timing becomes critical: If the FIFO is not read quickly, you might get RX overflow.	The MCU can only read a few bytes from FIFO, but might let the receiver start processing the packet at an earlier stage.

TX Scenario: Transmit packet A with 123 bytes of data, then immediately afterwards send packet B with 10 bytes of data.

1. Prepare packet. Make sure the packet length is pre-pended to the packet payload. The length in this case will be 123 + 1 = 124 (payload + length byte).
2. Check number of bytes in FIFO. We assume this is the first packet, so there is room for 64 bytes in the FIFO.
3. Write 64 bytes to FIFO. 124 – 64 = 60 bytes of packet remaining.
4. Set RF chip in transmit mode (strobe TX).
5. Go to Sleep
6. **ISR: A FIFO Threshold interrupt occurs, setting the number of bytes in the FIFO to 32. We have configured this interrupt to trigger on falling edge, so the chip is now emptying the FIFO. Wake up MCU.**
7. The MCU now writes 64 – 32 = 32 bytes to the FIFO. 60 – 32 = 28 bytes of packet remaining.
8. The radio is still in TX, so there is no need to strobe TX.
9. Go to sleep.
10. **ISR: A FIFO Threshold interrupt occurs, setting the number of bytes in the FIFO to 32. Wake up MCU.**
11. Write the remaining 28 bytes to the FIFO. Don't strobe TX.
12. Packet A written to FIFO. Prepare next packet with a total of 10 + 1 = 11 bytes.
13. There is (at least) room for 64 – (32 + 28) = 4 bytes in the FIFO. Write 4 bytes to FIFO. 7 bytes of packet remaining.
14. Don't strobe TX, since a packet sent interrupt has not yet occurred (i.e. the previous packet not sent).
15. Go to sleep.
16. **ISR: A FIFO Threshold interrupt occurs, setting the number of bytes in the FIFO to 32. Wake up MCU.**
17. Write the remaining 7 bytes to FIFO. Packet B written to FIFO.
18. **ISR: A FIFO Threshold interrupt occurs, setting the number of bytes in the FIFO to 32. Wake up MCU.**
19. Nothing to do. Go back to sleep.
20. **ISR: A Packet Transmitted interrupt occurs (packet A sent over the air). Read number of bytes in FIFO (should be 11). Since there is a packet pending for transmission, strobe TX. Wake up MCU.**
21. Nothing to do. Go back to sleep.
22. **ISR: A Packet Transmitted interrupt occurs (packet B sent over the air). Read number of bytes in FIFO (should be 0). No packets pending, so don't strobe TX. Wake up MCU.**
23. Scenario finished

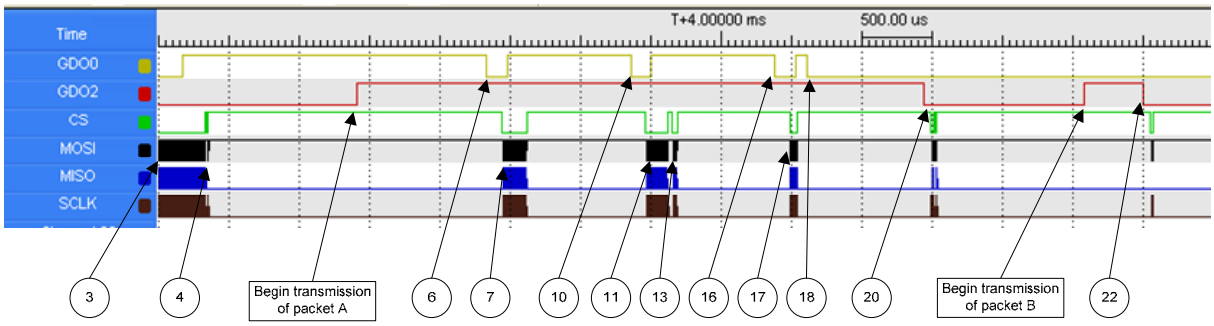

Figure 8 Capture of TX scenario using a logic analyzer

RX Scenario: Receive packet with arbitrary length.

1. Prepare reception of new packet.
2. Set RF chip in receive mode (strobe RX).
3. Go to sleep
4. **ISR: FIFO Threshold interrupt. Set data pending flag.**
5. Data is pending. There is at least 32 bytes in the RX FIFO. Read length byte. For this example, we will assume that the length is 123 (i.e. 124 bytes in packet).
6. Read remaining data in FIFO, but don't empty FIFO completely. Read 32 – 1 (length byte) – 1 (don't empty FIFO) = 30 bytes. 124 – 1 – 30 = 93 bytes remaining.
7. Go to sleep
8. **ISR: FIFO Threshold interrupt. Set data pending flag.**
9. Read data in FIFO, but don't empty FIFO completely. Read 32 – 1 = 31 bytes. 62 bytes remaining.
10. Go to sleep.
11. **ISR: FIFO Threshold interrupt. Set data pending flag.**
12. Read data in FIFO, but don't empty FIFO completely. Read 32 – 1 = 31 bytes. 31 bytes remaining.
13. Go to sleep.
14. **ISR: FIFO Threshold interrupt. Set data pending flag.**
15. **ISR: Packet complete interrupt. Set packet received flag.**
16. Read remaining data from FIFO.
17. Check packet integrity by reading the two appended status bytes (RSSI and LQI).
18. Go to 1.

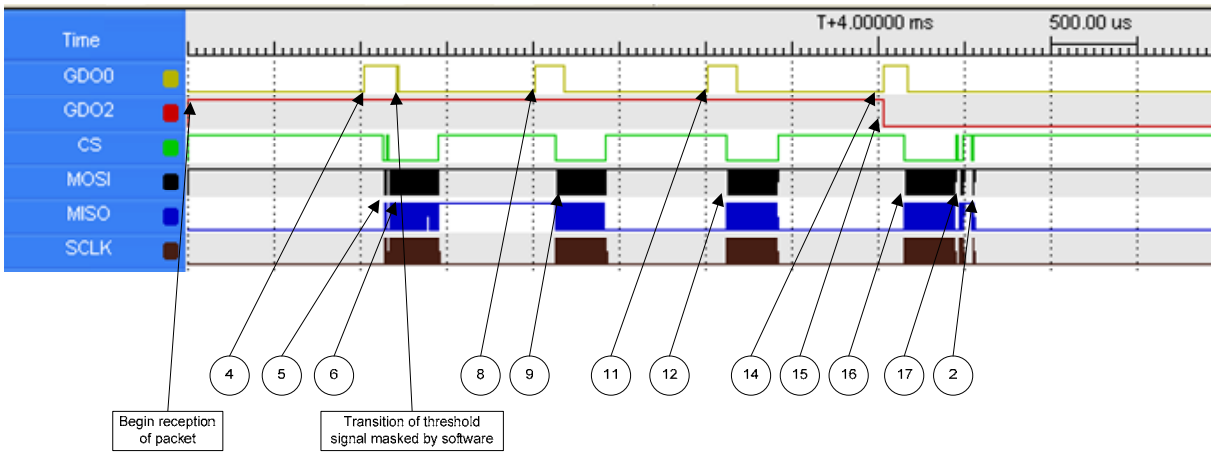

Figure 9 Capture of RX scenario using a logic analyzer

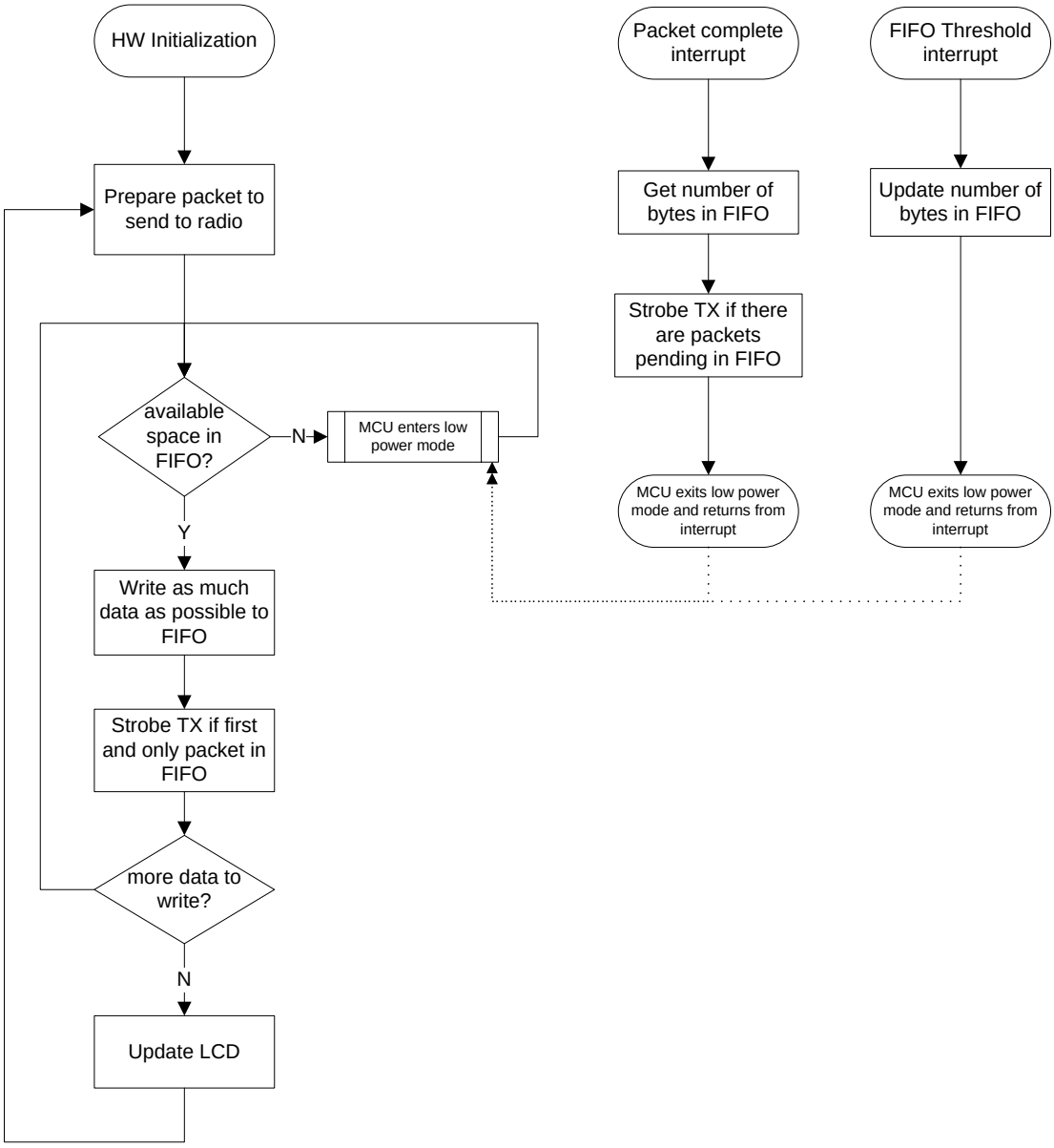

Figure 10 Transmitting packets in the "link_interrupt" example

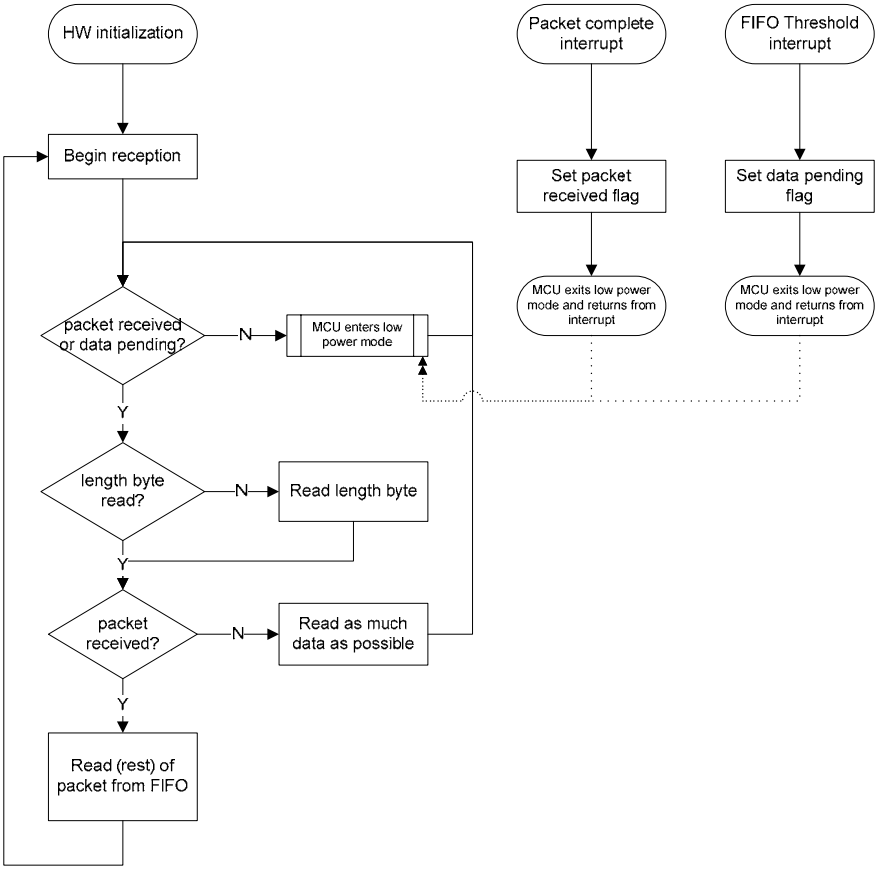

Figure 11 Receiving packets in the “link_interrupt” example

3.2 Hardware Abstraction Layer

The idea of a hardware abstraction layer (HAL) is to provide a standard interface to hardware resources on any board.

The common part of the HAL contains the core functionality of the library, that is the functions for accessing the CC1100 and the CC2500 using the SPI interface. The common part can be reused on any MSP430 based target board.

The target specific part of the HAL is necessary in order to configure the MCU – it sets up the clock system, configures I/O pins for general purpose use and dedicated peripherals etc. In addition, it contains functions for accessing target or board specific units, such as UART, LCD and LEDs. The HAL for the MSP430 Experimenter board (hal_exp430) provides an efficient interface to some of its many features, limited to the interest of the RF examples described in this application note.

The following sections will describe the individual components of the HAL.

3.2.1 CC1100/CC2500 RF Interface

Description

This component contains functions for interfacing the CC1100/CC2500 family of RF transceivers. Note that CC1100 and CC2500 are identical in terms of SW access, i.e. they have the same registers (identical memory map).

The functions in this component use SPI read and write functions for accessing the chip, and it would be possible to skip this layer and use the SPI functions directly in the application to reduce function call overhead (in order to reduce execution time and code size).

Include file

hal_rf.h
cc1100.h or cc2500.h (for register definitions)

Functions

```
void halRfResetChip(void)
```

Software reset of the chip, using the method described in the CC1100 and CC2500 datasheet.

```
void halRfConfig(          const HAL_RF_CONFIG* rfConfig,
                           const uint8* rfPaTable,
                           uint8 rfPaTableLen)
```

The function configures the chip according to the rfConfig structure. The configuration structure can be exported directly from SmartRF Studio, using the code export feature. rfPaTable and rfPaTable defines what to write into the chips PA table. One entry (the first) is sufficient if PA ramping is not used. See the datasheet for details.

```
void halRfBurstConfig(     const HAL_RF_BURST_CONFIG rfConfig,
                           const uint8* rfPaTable,
                           uint8 rfPaTableLen)
```

The function configures the chip according to the rfConfig structure. Instead of using the structure exported by SmartRF Studio directly, this function takes a predefined array of all registers, sorted in order to match the memory map of the registers, and writes them in one burst write to the chip. This reduces the function call overhead and the time it takes to configure the chip.

```
uint8 halRfGetChipId(void)
```

Returns the id of the chip (contents of PARTNUM register).

```
uint8 halRfGetChipVer(void)
```

Returns the chip version (contents of the VERSION register).

`uint8 halRfReadReg(uint8 addr)`

Read contents of one of the configuration registers of the chip. Returns the value of the specified register.

`uint8 halRfReadStatusReg(uint8 addr)`

Read contents of one of the status registers of the chip. Returns the value of the specified register.

`HAL_RF_STATUS halRfWriteReg(uint8 addr, uint8 value)`

Write the given value to one of the configuration registers of the chip. Returns the status byte of the chip (status before writing begins, with TX FIFO information in the last four bits). The status byte is described in [1] and [2].

`HAL_RF_STATUS halRfStrobe(uint8 cmd)`

Sends a command strobe to the chip. Returns the status byte of the chip (status before strobe was sent, with TX FIFO information in the last four bits).

`HAL_RF_STATUS halRfGetRxStatus(void)`

Returns the chip status bytes with RX FIFO information (available bytes in FIFO) in the last four bits.

`HAL_RF_STATUS halRfGetTxStatus(void)`

Returns the chip status bytes with TX FIFO (free space in FIFO) information in the last four bits.

`HAL_RF_STATUS halRfReadFifo(uint8* data, uint16 length)`

Read length bytes from the RX FIFO and store in buffer provided by caller. NB! The user should not attempt to read more data than there is currently in the RX FIFO. Otherwise, the FIFO might stop working as expected. Returns the status byte of the chip (status before read begins, with RX FIFO information in the last four bits).

`HAL_RF_STATUS halRfWriteFifo(uint8* data, uint16 length)`

Write length bytes from user provided buffer to the TX FIFO. NB! The user should avoid writing more data than there is currently room for in the FIFO. Otherwise, data will be overwritten and the FIFO might stop working as expected. Returns the status byte of the chip (status before write begins, with TX FIFO information in the last four bits).

3.2.2 SPI – Serial Peripheral Interface

Description

The SPI module implements the interface between the Chipcon RF IC and the MSP430. The SPI interface uses one of the available serial channels on the MSP430, be it USART, USCI or USI. Traditional BITBANG is also supported. The interface to use is configured in hal_board.h. One, *and only one*, of the following symbols must be defined:

HAL_SPI_INTERFACE_USART0	USART0 will be used to interface the RF IC
HAL_SPI_INTERFACE_USART1	USART1 will be used to interface the RF IC
HAL_SPI_INTERFACE_USCIA0	USCIA0 will be used to interface the RF IC
HAL_SPI_INTERFACE_USCIA1	USCIA1 will be used to interface the RF IC
HAL_SPI_INTERFACE_USCIB0	USCIB0 will be used to interface the RF IC
HAL_SPI_INTERFACE_USCIB1	USCIB1 will be used to interface the RF IC
HAL_SPI_INTERFACE_USI	USI will be used to interface the RF IC
HAL_SPI_INTERFACE_BITBANG	BITBANG will be used to interface the RF IC

The pins associated with these interfaces are MCU and hardware specific. Thus, the port and pin number for each of the signals needed by the SPI interface (SOMI, SIMO, CLK and CS) must also be set up in hal_board.h:

HAL_SPI_SOMI_PORT	MCU IO Port used by SPI SOMI signal
HAL_SPI_SOMI_PIN	The pin number on the port used by the SOMI signal
HAL_SPI_SIMO_PORT	MCU IO Port used by SPI SIMO signal
HAL_SPI_SIMO_PIN	The pin number on the port used by the SIMO signal
HAL_SPI_CLK_PORT	MCU IO Port used by SPI clock
HAL_SPI_CLK_PIN	The pin number on the port used by the SPI clock
HAL_SPI_CS_PORT	MCU IO Port used by the chip select signal
HAL_SPI_CS_PIN	The pin number on the port used by the CS signal

On the MSP430 Experimenter Board, USART0 is used. The macros above are set accordingly.

The SPI module has been implemented to support all possible hardware configurations. Thus, there is one halSpiInit() function for each of the interfaces. In addition, the generic SPI functions use a set of macros for all SPI transactions. The specific macros are implemented in hal_spi_config.c and hal_spi_config.h. The generic functions are implemented in hal_spi.c.

The generic macros can be used directly in the user code, allowing full control of the SPI interface.

HAL_SPI_BEGIN	Start SPI transaction. Assert CSn
HAL_SPI_END	End SPI transaction. Deassert CSn.
HAL_SPI_WAIT_RXFIN	Wait until a complete byte is clocked in to the RX buffer
HAL_SPI_WAIT_TXFIN	Wait until a complete byte has been clocked out on the wire
HAL_SPI_WAIT_TXBUF	Wait until a complete byte can be written to the TX buffer
HAL_SPI_TXBUF	For direct access to the TX buffer. Assign a value to the buffer to start writing over the interface. The MCU will generate the clock and clock out the individual bits on the MOSI pin.
HAL_SPI_RXBUF	For direct access to the RX buffer. When an SPI transaction has finished, read the value of the buffer to see what was clocked out from the peer.

Note that this software configures the MCU to operate as SPI master. Also note that, for all SPI transactions, the slave select signal is asserted at the beginning of a transfer and de-asserted at the end of a transfer.

Include file

hal_spi.h

Functions

```
void halSpiInit(uint8 speed)
```

Configure and initialize SPI interface.

The speed argument is currently not supported.

```
uint8 halSpiRead(uint8 address, uint8* buffer, uint16 length)
```

Read length bytes from the chip at the given address. Data will be written to buffer. The function returns the byte that the slave returns during the addressing phase (the byte returned by the slave while the master writes the first byte (address or header) to the slave, i.e. the status byte).

```
uint8 halSpiWrite(uint8 address, const uint8* buffer, uint16 length)
```

Write length bytes of data from buffer to the chip at the given address. The function returns the byte that the slave returns during the addressing phase (the byte returned by the slave while the master writes the first byte (address or header) to the slave, i.e. the status byte).

```
uint8 halSpiStrobe(uint8 cmd)
```

Send single byte to chip. Returns whatever data the chip sent on the MISO line during the single byte access.

3.2.3 MCU – Micro Controller Unit

Description

The MCU component implements basic functions for configuring the MCU clock system, and setting the MCU in a low power mode.

Include file

hal_mcu.h

Functions

```
void halMcuInit(void)
```

This function stops the watchdog timer and configures the clock system. The function makes sure that the oscillators are running stable before returning.

This function should be called prior to doing anything else in software.

```
void halMcuSetPowerMode(uint8 mode)
```

Sets the MCU in a low power mode. Use one of the following values when setting the mode.

```
HAL_MCU_LPM_0
HAL_MCU_LPM_1
HAL_MCU_LPM_2
HAL_MCU_LPM_3
HAL_MCU_LPM_4
```

The mapping of the above power modes to the actual power mode of the MCU is implementation dependent. For the MSP430, `HAL_MCU_LPM_0` corresponds to LPM0, `HAL_MCU_LPM_1` corresponds to LPM1 etc. For another MCU the mapping would probably be different. Note that not all MCUs support all of these low power modes.

Also note that the peripherals that are running in the different low power modes depend on the MCU. The application developer should take this into account when using these modes. That is, it is difficult to reliably port applications using low power modes, from one family of MCUs to another.

NB! Global interrupts will be enabled when entering one of the low power modes.

```
void halMcuWaitUs(uint16 usec)
```

This function waits (busy wait) for the specified number of microseconds. The accuracy will depend on the hardware.

3.2.4 Global Interrupt Control

Include file

hal_int.h

Functions

`void halIntOn(void)`

Unconditionally turns interrupts on (regardless of previous interrupt state).

`void halIntOff(void)`

Unconditionally turns interrupts off (regardless of previous interrupt state).

`uint16 halIntLock(void)`

Turns interrupts off and returns current state of interrupts.

`void halIntUnlock(uint16 key)`

Sets interrupt state back to previous state (using return value from halIntLock() function). Avoids turning interrupts on if interrupts were already turned off when calling halIntLock().

To avoid function call overhead, the functions above are also implemented as macros:

```
HAL_INT_ON()
HAL_INT_OFF()
HAL_INT_LOCK(x)     // x is a 16 bit unsigned integer
HAL_INT_UNLOCK(x)   // x is a 16 bit unsigned integer
```

3.2.5 Periodic Timer

Include file

hal_timer.h

Functions

```
void halTimerInit(uint16 rate)
```

Sets up a timer to generate an interrupt with the rate given by the input argument. The rate is given in microseconds.

```
void halTimerRestart(void)
```

Stops the timer and restarts it, counting up from 0.

```
void halTimerConnect(ISR_FUNC_PTR isr)
```

Connects an ISR to the timer interrupt. Each time the timer interrupt occurs, the connected function is called. Note that this function does not enable timer interrupts.

```
void halTimerEnable(void)
```

Enable timer interrupt.

```
void halTimerDisable(void)
```

Disable timer interrupt.

3.2.6 Digital IO Management

Description

The digital I/O (digio) component implements an abstraction access to hardware pins configured as digital I/O without needing to handle individual bits and hardware dependent registers. Its main purpose is to simplify pin configuration for any given combination of MCU and board layout.

This component also provides simple functions for setting up interrupt service routines for events on digital inputs that supports interrupts.

The interrupt framework will always wake up the MCU from any low power mode it is currently in.

The main data structure of the component is the digioConfig structure.

```
typedef struct {
    uint8    port;      // The port number (1, 2, 3 ...)
    uint8    pin;       // The pin number (1, 2, 3, ...)
    uint8    pin_bm;    // Pin bitmask
    uint8    dir;       // Direction (INPUT or OUTPUT)
    uint8    initval;   // Initial value (valid if output)
} digioConfig;
```

For a given combination {MCU, PCB}, set up the digital I/O pins used by the software by initializing pin variables, giving a value to all structure members. See below for an example.

Example

Suppose that pin 1 on port 5 of an MCU is connected to a LED, and that pin 3 on port 1 is connected to a button. To toggle the LED by means of an interrupt generated by pressing the button, the software would look something like this.

```
const digioConfig pinLed    = {5, 1, BIT1, HAL_DIGIO_OUTPUT, 0};
const digioConfig pinButton = {1, 3, BIT3, HAL_DIGIO_INPUT,  0};

void myIsr(void)
{
   halDigioToggle(&pinLed);
}

void main(void)
{
   // do something

   halDigioConfig(&pinLed);
   halDigioConfig(&pinButton);

   halDigioIntConnect(&pinButton, &myIsr);
   halDigioIntSetEdge(&pinButton, HAL_DIGIO_INT_EDGE_FALLING);
   halDigioIntEnable(&pinButton);

   // do something else
}
```

On a different board, it would suffice to change the initial values of pinLed and pinButton and the rest of the code would remain unchanged. This greatly improves portability, reusability, readability and maintainability of the code, at the cost of a few extra CPU cycles due to function calls and data fetches.

Include file

hal_digio.h

Functions

```
uint8 halDigioConfig(const digioConfig* pin)
```

Given an initialized digio structure, this function will set and clear the required bits in the appropriate configuration registers. This function must be called prior to using the digital I/O signal.

```
uint8 halDigioSet(const digioConfig* pin)
```

If the pin is configured as a digital I/O output, the output is set high (logic 1).

```
uint8 halDigioClear(const digioConfig* pin);
```

If the pin is configured as a digital I/O output, the output is set low (logic 0).

```
uint8 halDigioToggle(const digioConfig* pin);
```

If the pin is configured as a digital I/O output, the output changes state (high if it was low, low if it was high).

```
uint8 halDigioGet(const digioConfig* pin)
```

Returns the current value (high(1) or low(0)) of a digital I/O input pin.

```
uint8 halDigioIntConnect(const digioConfig* pin, ISR_FUNC_PTR isr)
```

Connects an interrupt service routine to an input pin. The ISR will be invoked if the interrupt has been enabled and if the trigger condition is met (rising or falling edge of pin).

```
uint8 halDigioIntEnable(const digioConfig* pin)
```

Enables interrupts for the specified input pin.

```
uint8 halDigioIntDisable(const digioConfig* pin)
```

Disables interrupts for the specified input pin.

```
uint8 halDigioIntClear(const digioConfig* pin)
```

Clears pending flag for given interrupt source. Note that the flag also will be cleared when the interrupt service routine is invoked.

```
uint8 halDigioIntSetEdge(const digioConfig* pin, uint8 edge)
```

Set to either HAL_DIGIO_INT_FALLING_EDGE or HAL_DIGIO_INT_RISING_EDGE to select triggering event.

3.2.7 LCD – Liquid Crystal Display

Description

Depending on hardware, this module lets the user operate the LCD on the board. The following functions are provided by the interface (but not all will be implemented on all hardware platforms). The MSP430 Experimenter Board uses the Softbaugh SBLCDA4 display.

Include file

hal_lcd.h

Functions

```
void halLcdInit(void)
```

Set up LCD and prepare hardware for use.

```
void halLcdWriteScreen(const char* line1, const char* line2)
```

Write two strings to a two line multi segment LCD (not supported on Experimenter's board).

```
void halLcdWriteValue(uint16 value, uint8 radix, uint8 line)
```

Display a value on the LCD. Select either decimal (`HAL_LCD_RADIX_DEC`) or hexadecimal format (`HAL_LCD_RADIX_HEX`). The last argument (line) is ignored in the current implementation, but is added to be compliant with other LCDs with more than one line.

```
halLcdClear(void)
```

Clear display

```
void halLcdWriteSymbol(uint8 symbol, uint8 on)
```

Display special symbols supported by the LCD. Turn on by setting `on` to any value different from 0. Turn off by setting `on` to 0. The following symbols are currently supported by the API

`LCD_SYMBOL_ANT`	Antenna
`LCD_SYMBOL_RX`	RX
`LCD_SYMBOL_TX`	TX

3.2.8 LED – Light Emitting Diode

Description

Basic operation of LEDs on a PCB.

Include file

hal_led.h

Functions

```
void halLedSet(uint8 id)
```

Turn led with ID id on (ID is hardware dependent)

```
void halLedClear(uint8 id)
```

Turn led with ID id off (ID is hardware dependent)

```
void halLedToggle(uint8 id)
```

Change state of LED.

3.2.9 UART – Universal Asynchronous Receive Transmit

Include file

hal_uart.h

Functions

```
void halUartInit(uint8 speed, uint8 options)
```

Set up UART interface with given baudrate and communication options.

The speed and option arguments are currently not supported. The current implementation for the MSP430 Experimenter Board configures the UART to operate at 115200 8-N-1.

```
void halUartWrite(const uint8* buf, uint16 length)
```

Write length bytes from buffer to the UART interface. Only polled mode “byte-banging” is implemented.

```
void halUartRead(uint8* buffer, uint16 length)
```

Read data from UART (not yet implemented).

4 References

[1] CC1100 Single-Chip Low Cost Low Power RF-Transceiver, Data sheet (cc1100.pdf)

[2] CC2500 Single-Chip Low Cost Low Power RF-Transceiver, Data sheet (cc2500.pdf)

[3] CC1100 Errata Notes (swrz012.pdf)

[4] CC2500 Errata Notes (swrz002.pdf)

[5] DN500 Packet Transmission Basics (swra109.pdf)

[6] DN503 SPI Access (swra112.pdf)

[7] DN506 GDO Pin Usage (swra121.pdf)

[8] MSP430 Experimenter's Board Product Page (msp-exp430fg4618)

[9] MSP430 Experimenter's Board User's Guide (slau213)

[10] MSP430 Interface to CC1100/CC2500 (slaa325)

5 Document History

Revision	Date	Description/Changes
SWRA141	2007-05-31	Initial release

Appendix A Porting the software to another board

As noted in the beginning of this document, the examples provided by this application note run out-of-the-box on the MSP430 Experimenter's Board. This appendix will briefly state what you will need to modify in order to adapt the examples and library to another board.

The common part of the HAL is the core of the library and does not need to be modified to work on other boards. However, they are dependent on some "glue" functions and macros in the target specific part of the HAL.

The main files for board specific configuration are hal_board.h, hal_board.c and hal_mcu.c. They all need to change if you want to use this software on another board. Optionally, hal_timer.c should also be modified in order to support the "link_poll" example. The table below gives an overview of their role and what you need to change.

File	Purpose	Modifications needed
hal_board.h	Contains definitions and macros used by other modules in the HAL.	Modify macros to fit your hardware. In particular, the macros that define the pin and ports used by the SPI interface must match the board layout, in addition to the serial interface used for communication with the RF IC.
hal_board.c	Defines and configures digital IO pins used on your board.	Modify declaration of digital IO pins to fit the actual hardware. Add or remove pins to your convenience. **halBoardInit()** should do all board level initialization.
hal_mcu.c	This file contains a function for setting up the clock on the microcontroller and a busy wait function. Both are heavily dependent on the microcontroller in use.	**halMcuInit()** sets the system clock rate and determines which clock sources to use. As this will depend on your hardware and the MCU in use, the function should be modified.
		halMcuWaitUs() implements a tight busy wait function, depending on the system clock rate and instruction timings. Must be modified for correct operation.
hal_timer.c	Sets up a timer on the board to generate a periodic interrupt.	**halTimerInit()** should be modified in order to set up an appropriate timer for a given board, depending on system clock rate and available external crystals. The resolution and accuracy of the timer ticks will depend on all of the above factors.

The functions for accessing the LEDs, LCD and UART are target specific. They were added to support the examples with methods for displaying data and interacting with the user. Removing these functions, replacing them by stubs or implementing the same function for different hardware will not affect the basic functionality of the library.

The accompanying HAL template (hal\hal_template) can be used as a starting point when developing a HAL for a new target.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information 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.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2007, Texas Instruments Incorporated