HAL Drivers
Application Programming Interface

Document Number: SWRA193

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial release.	03/02/2006
1.1	Hal sleep updates	11/20/2006
1.2	Updated LCD and timers API	11/22/2006
1.3	Added CC2591 PA/LNA control APIs	06/11/2008
1.4	Added I2C interface and IR signal generation APIs Updated UART API and made editorial changes	04/03/2009
1.5	Updated halSleep API Removed halSleepWait API Added a note in timer service	09/23/2011
1.6	Remove references to deprecated platform (CC2430).	05/02/2013
1.7	Added halSleepWait	02/20/2015

TABLE OF CONTENTS

1. Introduction

1.1 Purpose

This document describes the application programming interface for HAL Drivers. The API provides application the interface to access timers, GPIO, UART and ADC. This is a platform independent API that provides a superset of features for each service. Not all features will be available for all platforms.

NOTE: This document does not currently cover any details specific to the CC2538.

1.2 Acronyms

ADC	Analog to Digital Conversion
API	Application Programming Interface
CD	Carrier Detect
CTC	Clear Timer on Compare
CTS	Clear To Send
DMA	Direct Memory Access
DSR	Data Set Ready
DTR	Data Terminal Ready
GPIO	General Purpose Input Output
HAL	Hardware Abstract Layer
HGM	High Gain Mode
I2C	Inter-IC Bus
IC	Integrated Circuit
IR	Infra-red
ISR	Interrupt Service Routine
LGM	Low Gain Mode
LNA	Low Noise Amplifier
MAC	Medium Access Control.
OSAL	Operating System Abstraction Layer
PA	Power Amplifier
RI	Ring Indicator
RTS	Ready To Send

2. Drivers API Overview

2.1 Function Calls

2.1.1 Initialization Function Calls

These function calls are used to initialize a service and /or to setup optional parameters for platform-specific data. Initialization functions are often called at the beginning stage when the device powers up.

2.1.2 Service Access Function Calls

These function calls can directly access hardware registers to get/set certain value of the hardware (i.e. ADC) or control the hardware components (i.e. LED).

2.1.3 Callback Function Calls

These functions must be implemented by the application and are used to pass events that are generated by the hardware (interrupts, counters, timers…) or by polling mechanism (UART poll, Timer poll…) to upper layers. Data accessed through callback function parameters (such as a pointer to data) are only valid for the execution of the function and should not be considered valid when the function returns. If these functions execute in the context of the interrupt, it must be efficient and not perform CPU-intensive operations or use critical sections.

2.2 Services

HAL drivers provide Timer, GPIO, LEDs, key switches, UART, ADC, IR signal generation and I2C interface service for MAC and upper layers. Not all the features in the service are available in every platform. Features in each service can be configured for different platforms through initialization function.

3. ADC Service

This service supports 8, 10, and 14-bit analog to digital conversion on 8 channels (0-7).

3.1 HalAdcInit ()

3.1.1 Description

This ADC initialization function is called once at the startup. This function has to be called before any other ADC functions can be called. It enables ADC to be initialized with both required and optional parameters.

3.1.2 Prototype

```
void HalAdcInit (void)
```

3.1.3 Parameter Details

None.

3.1.4 Return

None.

3.2 HalAdcRead ()

3.2.1 Description

This function reads and returns the value of the ADC conversion at specified channel and resolution.

3.2.2 Prototype

```
uint16 HalAdcRead (uint8 channel,
                   uint8 resolution);
```

3.2.3 Parameter Details

channel – input channels (Check Channels Table)

resolution – resolution of the conversion. (Check Resolutions Table)

3.2.4 Return

Return 16-bit value of the conversion at the given channel and resolution.

3.3 Constants

3.3.1 Channels

Parameter	Desription
HAL_ADC_CHANNEL_0	Input channel 0
HAL_ADC_CHANNEL_1	Input channel 1
HAL_ADC_CHANNEL_2	Input channel 2
HAL_ADC_CHANNEL_3	Input channel 3

HAL_ADC_CHANNEL_4	Input channel 4
HAL_ADC_CHANNEL_5	Input channel 5
HAL_ADC_CHANNEL_6	Input channel 6
HAL_ADC_CHANNEL_7	Input channel 7

3.3.2 Resolutions

Parameter	Desription
HAL_ADC_RESOLUTION_8	8-bit resolution
HAL_ADC_RESOLUTION_10	10-bit resolution
HAL_ADC_RESOLUTION_12	12-bit resolution
HAL_ADC_RESOLUTION_14	14-bit resolution

4. LCD Service

This service allows writing text on the LCD. Not every board supports LCD.

4.1 HalLcdInit()

4.1.1 Description

This initialization function is called once at the startup. This function has to be called before any other LCD function can be called. It enables LCD to be initialized with both required and optional parameters.

4.1.2 Prototype

```
void HalLcdInit (void);
```

4.1.3 Paraneter Details

None.

4.1.4 Return

None.

4.2 HalLcdWriteString()

4.2.1 Description

This routine writes a string to the LCD.

4.2.2 Prototype

```
void HalLcdWriteString (unit8* str,
                        uint8 option);
```

4.2.3 Parameter Details

str – pointer to the string that will be displayed on the LCD. Max length of the string is defined under HAL_LCD_MAX_CHARS. If the length is greater than HAL_LCD_MAX_CHARS, then only HAL_LCD_MAX_CHARS will be displayed.

option – option for the string to be displayed on the LCD. (Check Options table).

4.2.4 Return

None.

4.3 HalLcdWriteValue()

4.3.1 Description

This routine writes a 32-bit value to the LCD.

4.3.2 Prototype

```
void HalLcdWriteValue (uint32 value,
                       uint8 radix,
                       uint8 option);
```

4.3.3 Parameter Details

value – value that will be displayed on the LCD.

radix – representation of the value. It can be 8, 10 or 16. (Octal, Decimal, or Hex)

option – option for the value to be displayed on the LCD. (Check Options table)

4.4 HalLcdWriteScreen()

4.4.1 Description

This routine writes 2 lines of text on the LCD display.

4.4.2 Prototype

```
void HalLcdWriteScreen( char *line1,
                        char *line2 );
```

4.4.3 Parameter Details

Line1 – pointer to the string that will be displayed on the 1st line of the LCD

Line2 – pointer to the string that will be displayed on the 2nd line of the LCD

4.4.4 Return

None.

4.5 HalLcdWriteStringValue()

4.5.1 Description

This routine writes a string followed by a 16-bit value on the specified line number on the LCD display.

4.5.2 Prototype

```
void HalLcdWriteStringValue( char *title,
```

```
                                    uint16 value,
                                    uint8 format,
                                    uint8 line );
```

4.5.3 Parameter Details

title – string that will be displayed on the LCD display.

value – value that will be displayed following the "title".

format – format of the value that will be displayed. It can be 8, 10, and 16 (Octal, Decimal and Hex)

line – the line number where the string (title and value) will be displayed on the LCD display.

Example: To display "Count: 180" on line 2 of the LCD display

HalLcdWriteStringValue ("Count:", 180, 10, 2);

4.5.4 Return

None.

4.6 HalLcdWriteStringValueValue()

4.6.1 Description

Write two 16-bit values back to back on the specified line on the LCD display underneath the a title.

4.6.2 Prototype

```
void HalLcdWriteStringValueValue( char *title,
                                  uint16 value1,
                                  uint8 format1,
                                  uint16 value2,
                                  uint8 format2,
                                  uint8 line );
```

4.6.3 Parameter Details

title – string that will be displayed on the LCD display.

value1 – 1st value that will be displayed after the "title".

format1 – format for the "value1". It can be 8, 10, and 16 (Octal, Decimal, and Hex).

value2 – 2nd value that will be displayed after "value1".

format2 – format for the "value2". It can be 8, 10, and 16 (Octal, Decimal, and Hex).

line – line number where the string (title, value1 and value2) will be displayed on the LCD display.

Example: `HalLcdWriteStringValueValue ("Test: ", 2, 10, 30, 10, 1);`

Display: Test: 2 30

4.6.4 Return

None.

4.7 HalLcdDisplayPercentBar()

4.7.1 Description

Simulating a percentage bar on the LCD with the percentage in numerical figure in the middle of the bar.

4.7.2 Prototype

```
void HalLcdDisplayPercentBar( char *title,
                              uint8 value );
```

4.7.3 Parameter Details

title – string that will be displayed on the 1st line of the LCD display

value – percentage value that will be displayed in the middle of the bar

Example: `HalLcdDisplayPercentBar ("Rate:", 50);`

Display:
```
Rate:
[||||||||50+          ]
```

4.7.4 Return

None.

4.8 Constants

4.8.1 Options

Option	Description
HAL_LCD_LINE_1	Display the text on line 1 of the LCD
HAL_LCD_LINE_2	Display the text on line 2 of the LCD

5. LED Service

This service allows LEDs to be controlled in different ways. LED service supports ON, OFF, TOGGLE, FLASH and BLINK. Not all platforms support these modes.

5.1 HalLedInit()

5.1.1 Description

This LED initialization function is called once at the startup. This function has to be called before any other LED function can be called. It enables LED to be initialized with both required and optional parameters.

5.1.2 Prototype

```
void HalLedInit (void);
```

5.1.3 Parameter Details

None.

5.1.4 Return

None.

5.2 HalLedSet()

5.2.1 Description

This function will set the given LEDs ON, OFF, BLINK, FLASH, or TOGGLE. If BLINK and FLASH mode are used, a set of default parameters will be used. To customize these parameters, HalLedBlink() has to be used.

5.2.2 Prototype

```
void HalLedSet (uint8 led, uint8 mode);
```

5.2.3 Parameter Details

led – bit mask of LEDs to be turned ON (See LEDs Table)

mode – new mode for the LEDs. (See Modes Table)

5.2.4 Return

None.

5.3 HalLedBlink()

5.3.1 Description

This function will blink the given LEDs based on the provided parameters.

5.3.2 Prototype

```
void HalLedBlink (uint8 leds,
                  uint8 numBlinks,
                  uint8 percent,
                  uint16 period);
```

5.3.3 Parameter Details

leds – bit mask of leds to be blinked (See LEDs Table)

numBlinks – number of times the LED will blink

percent – percentage of the cycle that is ON.

period – time in milliseconds of one ON/OFF cycle.

5.3.4 Return

None.

5.4 HalLedGetState()

5.4.1 Description

This function returns the current state of the LEDs.

5.4.2 Prototype

```
uint8 HalLedGetState (void);
```

5.4.3 Parameter Details

None.

5.4.4 Return

8-bit contains the current state of the LEDs. Each bit indicates the corresponding LED status.

5.5 HalLedEnterSleep()

5.5.1 Description

This function stores the current state of the LEDs and turn off all the LEDs to conserve power. It also sets a global state variable indicating sleep mode has been entered. This global state variable will stop the interrupt from processing the LEDs during while in sleep mode.

5.5.2 Prototype

```
void HalLedEnterSleep (void);
```

5.5.3 Parameter Details

None.

5.5.4 Return

None.

5.6 HalLedExitSleep()

5.6.1 Description

This function restores the original state of the LEDs before the device entered sleep mode.

5.6.2 Prototype

```
void HalLedExitSleep (void);
```

5.6.3 Parameter Details

None.

5.6.4 Return

None.

5.7 Constants

5.7.1 LEDs

LED	Description
HAL_LED_1	LED 1
HAL_LED_2	LED 2

HAL_LED_3	LED 3
HAL_LED_4	LED 4
HAL_LED_ALL	All LEDs

5.7.2 Modes

Mode	Description
HAL_LED_MODE_OFF	Turn OFF the LED
HAL_LED_MODE_ON	Turn ON the LED
HAL_LED_MODE_BLINK	BLINK the LED
HAL_LED_MODE_FLASH	FLASH the LED
HAL_LED_MODE_TOGGLE	TOGGLE the LED

6. KEY Service

This service provides services for keys, switches and joysticks. The service can be polling or interrupt driven. A callback function has to be registered in order for the service to inform the application of the status of the keys/switches/joysticks.

6.1 HalKeyInit()

6.1.1 Description

This initialization function is called once at the startup. This function has to be called before any other function that uses keys/switches/joysticks can be called. It enables Keys/switches/joysticks to be initialized with both required and optional parameters.

6.1.2 Prototype

```
void HalKeyInit (void *init);
```

6.1.3 Parameter Details

None.

6.1.4 Return

None.

6.2 HalKeyConfig()

6.2.1 Description

This function is used to configure the Keys/Switches/Joysticks service as polling or interrupt driven. It also sets up a callback function for the service. If interrupt is not used, polling starts automatically after 100ms. Keys/switches/joystick will be polled every 100ms. If interrupt is used, an ISR will handle the case. There is a delay of 25ms after the interrupt occurs to eliminate de-bounce.

6.2.2 Prototype

```
void HalKeyConfig (bool interruptEnable,
                   halKeyCBack_t *cback);
```

6.2.3 Parameter Details

interruptEnable – TRUE or FALSE. Enable or disable the interrupt. If interrupt is disabled, the keys/switches/joysticks will be polled. Otherwise, the keys/switches/joysticks will be on interrupt. For maximum power savings, ensure that interruptEnable is set to TRUE so that the background key poll timer is not running keeping the device from entering sleep for long periods of time.

cback – this call back occurs when a key, switch or joystick is active. If the callback is set to NULL, the event will not be handled.

```
typedef void (halKeyCback_t) (uint8 key, uint8 state);
```

key – the key/switch/joystick that will cause the callback when it's triggered. Key code varies per platform (Check Keys table for typical key definitions for evaluation/development board).

state – current state of the key/switch/joystick that causes the callback. (Check States table)

6.2.4 Return

None.

6.3 HalKeyRead()

6.3.1 Description

This function is used to read the current state of the Key/Switch/Joystick. If the Key Service is set to polling, this function will be called by the Hal Driver Task every 100ms. If the Key Service is set to interrupt driven, this function will be called by the Hal Driver Task 25ms after the interrupt occurs. If a callback is registered during HalKeyConfig(), that callback will be sent back to the application with the new status of the keys. Otherwise, there will be no further action.

6.3.2 Prototype

```
uint8 HalKeyRead ( void );
```

6.3.3 Parameter Details

None.

6.3.4 Return

Return the key code. Key code varies per platform (Check Keys table for typical key code definitions for evaluation/development board). Key code could be comprised of bits indicating individual key/switch, as is the case for the typical key code for evaluation/development board. Some platforms, which have more keys than can be represented by 8 bits, define key code as enumerated value or as a value comprised of bits indicating row number and bits indicating column number of a key matrix.

6.4 HalKeyEnterSleep()

6.4.1 Description

This function sets a global state variable indicating sleep mode has been entered. This global state variable is used to stop the interrupt from processing the keys during sleep mode.

6.4.2 Prototype

```
void HalKeyEnterSleep (void);
```

6.4.3 Parameter Details

None.

6.4.4 Return

None.

6.5 HalKeyExitSleep()

6.5.1 Description

This function sets a global state variable indicating sleep mode has been exited. It also processes those keys that are stored by the key interrupt.

6.5.2 Prototype

```
void HalKeyExitSleep (void);
```

6.5.3 Parameter Details

None.

6.5.4 Return

None.

6.6 HalKeyPoll()

6.6.1 Description

This routine is used internally by hal driver.

6.6.2 Prototype

void HalKeyPoll (void);

6.6.3 Parameter Details

None.

6.6.4 Return

None.

6.7 HalKeyPressed()

6.7.1 Description

This routine is used internally by hal sleep.

6.7.2 Prototype

```
bool HalKeyPressed( void );
```

6.7.3 Parameter Details

None.

6.7.4 Return

None.

6.8 Constant

6.8.1 Keys

Key	Description
HAL_KEY_SW_1	Key #1 is pressed
HAL_KEY_SW_2	Key #2 is pressed
HAL_KEY_SW_3	Key #3 is pressed
HAL_KEY_SW_4	Key #4 is pressed
HAL_KEY_SW_5	Key #5 is pressed
HAL_KEY_SW_6	Key #6 is pressed

6.8.2 Joystick

Key	Description
HAL_KEY_UP	Joystick is up
HAL_KEY_RIGHT	Joystick is down
HAL_KEY_CENTER	Joystick is center
HAL_KEY_LEFT	Joystick is left
HAL_KEY_DOWN	Joystick is down

6.8.3 States

State	Description
HAL_KEY_STATE_NORMAL	The key is pressed normally
HAL_KEY_STATE_SHIFT	The key is shift and pressed

7. Sleep Service

This service is part of the power saving mechanism. Osal uses these routines to exercise low power mode when POWER_SAVING symbol is compiled.

7.1 HalSleep()

7.1.1 Description

This routine is called from the OSAL task loop through the OSAL interface to set the low power mode of the MAC.

7.1.2 Prototype

```
void halSleep(uint32 osal_timeout)
```

7.1.3 Parameter Details

osal_timeout – The next OSAL timer timeout. This will be used to determine how long the MAC needs to sleep or stay awake.

7.1.4 Return

None.

7.2 halSleepWait

7.2.1 Description

Perform a blocking wait for the specified number of microseconds. It uses assumptions about number of clock cycles needed for the various instructions. This function assumes a 32 MHz clock. This function is highly dependent on architecture and compiler and is available only on CC2530, CC2531 and CC2533.

7.2.2 Prototype

```
void halSleepWait(uint16 duration)
```

7.2.3 Parameter Details

duration – time in microseconds to wait.

7.2.4 Return

None.

8. Timer Service

This service supports up to four hardware timers, two 8-bit timers and two 16-bit timers. **Note:** Z-Stack and TIMAC no longer use CC253x Timer 1, Timer 3, and Timer 4. The supporting timer driver module is removed and left for the users to implement their own application timer functions.

8.1 Operation Modes

Timer service supports Normal and CTC (Clear Timer on Compare) mode. In Normal mode, the Timer/Counter is always up, and no Timer/Counter clear is performed. The Timer/Counter simply overruns when it passes its maximum and then restarts from zero again. In CTC mode, the Timer/Counter will be cleared to zero when the Timer/Counter matches the given value.

8.2 Channels

Timer service has up to 3 channels for output compare and 1 channel for input capture or overflow channel mode for 16bit timers. For 8bit timers, the Timer Service has 1 channel for output capture or overflow channel mode.

8.3 Interrupts/Channel Modes

Timer service supports 3 interrupt sources, Overflow, Output Compare, and Input Capture. However, only 16bit timers support Input Capture.

8.4 HalTimerInit ()

8.4.1 Description

This timer initialization function is called once at the startup. It should be called before any other timer function can be called. It allows hardware timers to be initialized with both required and optional parameters.

8.4.2 Prototype


```
void HalTimerInit (void)
```

8.4.3 Parameter Details

None.

8.4.4 Return

None.

8.5 HalTimerConfig()

8.5.1 Description

This function allows the channels to be configured in different modes.

8.5.2 Prototype

```
halTimerStatus_t HalTimerConfig ( uint8 timerId,
                                  uint8 opMode,
                                  uint8 channel,
                                  uint8 channelMode,
                                  bool intEnable,
                                  halTimerCBack_t cback);
```

8.5.3 Parameter Details

timerId - HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers. (See TimerId Table)

opMode – Operation Mode, can be Normal or CTC (See Operation Modes Table)

channel – Channel to be configured. (See Channels Table)

channelMode – The channel mode, Input Capture Mode, Output Compare Mode or Overflow Mode. (See Channel Modes Table)

intEnable – TRUE or FALSE – Enable / Disable the interrupt

cback – pointer to the callback function. Callback function is used to inform the application whenever an interrupt occurs.

```
typedef void (halTimerCback_t) (uint8 timerId,
                                uint8 channel,
                                uint8 channelMode);
```

`timerId` – HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers. (See TimerId Table)

`channel` – the channel where the interrupt occurs. (See Channels Table)

`channelMode` – Interrupt source of the event.

8.5.4 Return

Status of the configuration. (See Status Table)

8.6 HalTimerStart ()

8.6.1 Description

This function starts the timer/counter with the operation mode, channel, channel mode, prescale that are provided by HalTimerConfig(). In other words, HalTimerConfig() has to be called before HalTimerStart() is called. In case of polling, timer ticks are updated by the Hal driver task calling HalTimerTick(). In case timer interrupt is used, timer ticks are updated by the interrupt every time an interrupt occurs.

8.6.2 Prototype

```
uint8 HalTimerStart ( uint8 timerId, uint32 timePerTick );
```

8.6.3 Parameter Details

`timerId` – HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers. (See TimerId Table)
`timerPerTick` – number of micro-seconds per tick

8.6.4 Return

If the Timer Service is not configured, HAL_TIMER_NOT_CONFIGURED will be returned. Otherwise, HAL_TIMER_OK will be returned. (See Status Table)

8.7 HalTimerStop()

8.7.1 Description

This function is called to stop a timer/counter.

8.7.2 Prototype

```
uint8 HalTimerStop (uint8 timerId);
```

8.7.3 Parameter Details

timerId – HAL_TIMER_X, identification of the timer that will be stopped. (See Timer ID Table)

8.7.4 Return

If the timerId is not valid, HAL_TIMER_INVALID_ID will be returned. Otherwise, HAL_TIMER_OK will be returned. (See Status Table)

8.8 HalTimerTick()

8.8.1 Description

This function is called by the Hal Driver task when the interrupt is disabled in order to create a tick for the application. To use HalTimerTick(), Timer Service has to be configured using HalTimerConfig() with intEnable set to FALSE before calling HalTimerTick(). HalTimerTick() will send back to the application using the provided callback function at every tick. The duration of the tick is setup using the information provided by HalTimerConfig().

8.8.2 Prototype

```
void HalTimerTick (void);
```

8.8.3 Parameter Details

None.

8.8.4 Return

None.

8.9 HalTimerInterruptEnable()

8.9.1 Description

This function will enable or disable timer interrupt of the timerId and channelMode.

8.9.2 Prototype

```
uint8 HalTimerInterruptEnable ( uint8 timerId,
                                 uint8 channelMode,
                                 bool enable );
```

8.9.3 Parameter Details

`timerId` – HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers. (See TimerId Table)

channelMode – The channel mode, Input Capture Mode, Output Compare Mode or Overflow Mode. (See Channel Modes Table)

enable – `TRUE or FALSE`

8.9.4 Return

If the timerId is not valid, HAL_TIMER_INVALID_ID will be returned. Otherwise, HAL_TIMER_OK will be returned. (See Status Table)

8.10 Constants

8.10.1 Timer ID

Note that description of timer ID enumeration is specific to CC2530. The actual timer mapped to the ID may vary per platform.

ID	Description
HAL_TIMER_0	8-bit timer ID
HAL_TIMER_1	16-bit timer ID – this is MAC timer and not supported by HAL
HAL_TIMER_2	8-bit timer ID
HAL_TIMER_3	16-bit timer ID

8.10.2 Channels

Channel	Description
HAL_TIMER_CHANNEL_SINGLE	Single Channel Timer
HAL_TIMER_CHANNEL_A	Timer Channel A
HAL_TIMER_CHANNEL_B	Timer Channel B
HAL_TIMER_CHANNEL_C	Timer Channel C

8.10.3 Channel Modes

Channel Mode	Description
HAL_TIMER_CH_MODE_INPUT_CAPTURE	Input Capture Mode
HAL_TIMER_CH_MODE_OUTPUT_COMPARE	Output Compare Mode
HAL_TIMER_CH_MODE_OVERFLOW	Overflow Mode

8.10.4 Operation Modes

Operation Mode	Description
HAL_TIMER_MODE_NORMAL	In Normal Mode, there is no counter clear performed. If the counter passes the max, it will restart again at zero
HAL_TIMER_MODE_CTC	In CTC Mode, the counter is cleared to zero when the counter matches with the specified value.

8.10.5 Prescale

Prescale – 8bit	Description
HAL_TIMER_8_TC_STOP	No clock, timer stopped
HAL_TIMER_8_TC_DIV1	No clock pre-scaling
HAL_TIMER_8_TC_DIV8	Clock pre-scaled by 8
HAL_TIMER_8_TC_DIV32	Clock pre-scaled by 32
HAL_TIMER_8_TC_DIV64	Clock pre-scaled by 64
HAL_TIMER_8_TC_DIV128	Clock pre-scaled by 128
HAL_TIMER_8_TC_DIV256	Clock pre-scaled by 256
HAL_TIMER_8_TC_DIV1024	Clock pre-scaled by 1024

Prescale – 16bit	Description
HAL_TIMER_16_TC_STOP	No clock, timer stopped
HAL_TIMER_16_TC_DIV1	No clock pre-scaling
HAL_TIMER_16_TC_DIV8	Clock pre-scaled by 8
HAL_TIMER_16_TC_DIV64	Clock pre-scaled by 64
HAL_TIMER_16_TC_DIV256	Clock pre-scaled by 256
HAL_TIMER_16_TC_DIV1024	Clock pre-scaled by 1024
HAL_TIMER_16_TC_EXTFE	External clock (T2), falling edge

HAL_TIMER_16_TC_EXTRE	External clock (T2), rising edge

8.10.6 Status

Status	Description
HAL_TIMER_OK	OK status
HAL_TIMER_NOT_OK	NOT OK status
HAL_TIMER_PARAMS_ERROR	Parameters are mismatched or no correct
HAL_TIMER_NOT_CONFIGURED	Timer is not configured
HAL_TIMER_INVALID_ID	Invalid Timer ID
HAL_TIMER_INVALID_CH_MODE	Invalid channel mode
HAL_TIMER_INVALID_OP_MODE	Invalid operation mode

9. UART Service

This service configures several things in the UART such as Baud rate, flow control, CTS, RTS, DSR, DTR, CD, RI…etc. It also reports framing and overrun errors.

9.1 HalUARTInit ()

9.1.1 Description

This UART initialization function is called once at the startup. This function has to be called before any other UART function can be called. It enables UART to be initialized with both required and optional parameters.

9.1.2 Prototype

```
void HalUARTInit (void)
```

9.1.3 Parameter Details

None

9.1.4 Return

None

9.2 HalUARTOpen ()

9.2.1 Description

This function opens a port based on the configuration that is provided. A callback function is also registered so events can be handled correctly.

9.2.2 Prototype

```
uint8 HalUARTOpen (uint8 port,
                   halUARTCfg_t *config);
```


9.2.3 Parameter Details

port – specified serial port to be opened. (Read UART Ports Table)

config – Structure that contains the information that is used to configure the port

```
typedef struct
{
  bool   configured;
  uint16 baudRate;
  bool   flowControl;
  uint16 flowControlThreshold;
  uint8  idleTimeout;
  uint16 rx;
  uint16 tx;
  bool   intEnable;
  uint32 rxChRvdTime;
  halUARTCBack_t callBackFunc;
}halUARTCfg_t;
```

config.configured – Set by the function when the port is setup correctly and read to be used.

config.baudRate – The baud rate of the port to be opened. (Check UART Baud Rate Table)

config.flowControl – UART flow control can be set as TRUE or FALSE. TRUE value will enable flow control and FALSE value will disable flow control.

config.flowControlThreshold – This parameter indicates number of bytes left before Rx buffer reaches maxRxBufSize. When Rx buffer reaches this number (maxRxBufSize – flowControlThreshold) and flowControl is TRUE, a callback will be sent back to the application with HAL_UART_RX_ABOUT_FULL event. This parameter is supported only by MSP platforms. For CC2530 platforms, compile flag HAL_UART_ISR_HIGH (in case interrupt is used) or HAL_UART_DMA_HIGH (in case DMA is used) determines the threshold of number of received bytes to trigger callback.

config.idleTimeout – This parameter indicates rx timeout period in milliseconds. If Rx buffer hasn't received new data for idleTimout amount of time, a callback will be issued to the application with HAL_UART_RX_TIMEOUT event. The application can choose to read everything from the Rx buffer or just a part of it. This parameter is supported only by MSP platforms. For CC2530 platforms, this parameter is replaced with compile flag HAL_UART_ISR_IDLE (in case interrupt is used) or HAL_UART_DMA_IDLE (in case DMA is used).

config.rx – Contains `halUARTBufControl_t` structure that is used to manipulate Rx buffer.

config.tx – Contains `halUARTBufControl_t` structure that is used to manipulate Tx buffer.

```
typedef struct
{
  uint16 bufferHead;
  uint16 bufferTail;
  uint16 maxBufSize;
```



```
  uint8 *pBuffer;
}halUARTBufControl_t;
```

bufferHead – This parameter is obsolete.

bufferTail – This parameter is obsolete.

maxBufSize – holds maximum size of the Rx/Tx buffer that the UART can hold at a time. When this number is reached for Rx buffer, HAL_UART_RX_FULL will be sent back to the application as an event through the callback system. If Tx buffer is full, HalUARTWrite() function will return 0. This parameter is supported only by MSP platforms. For CC2530, compile flag HAL_UART_ISR_RX_MAX (in case interrupt is used) or HAL_UART_DMA_RX_MAX (in case DMA is used) determines receive buffer size, and compile flag HAL_UART_ISR_TX_MAX (in case interrupt is used) or HAL_UART_DMA_TX_MAX (in case DMA is used) determines transmit buffer size.

***pBuffer –** This parameter is obsolete

config.intEnable – enable/disable interrupt. It can be set as TRUE or FALSE. TRUE value will enable the interrupt and FALSE value will disable the interrupt. For CC2530, compile flags HAL_UART_ISR and HAL_UART_DMA determine interrupt usage. To use interrupt for the driver, HAL_UART_ISR has to be defined with non-zero value matching the corresponding USART block enumeration (1 or 2) and HAL_UART_DMA has to be defined as zero. To use DMA for the driver, HAL_UART_ISR has to be defined as zero and HAL_UART_DMA has to be defined with corresponding USART block (1 or 2).

rxChRvdTime – This parameter is obsolete.

callBackFunc – This callback is called when there is an event such as Tx done, Rx ready…

```
void HalUARTCback (uint8 port, uint8 event);
```

port - specified serial port that has the event. (Check UART Ports Table).

event – event that causes the callback (Check Events table).

9.2.4 Return

Status of the function call. (Check Status Table).

9.3 HalUARTClose ()

9.3.1 Description

This function closes a given port. This function may be followed by HalUARTOpen() call in order to reconfigure port settings. Or, this function can be used in order to turn off UART for power conservation.

9.3.2 Prototype

```
void HalUARTClose (uint8 port);
```

9.3.3 Parameter Details

port - specified serial port to be closed. (Check UART Ports Table)

9.3.4 Return

None.

9.4 HalUARTRead ()

9.4.1 Description

This function reads a buffer from the UART. The number of bytes to be read is determined by the application. If the requested length is larger than the Rx Buffer, then the requested length will be adjusted to the Rx buffer length and the whole Rx buffer is returned together with the adjusted length. If the requested length is smaller than the Rx buffer length, then only requested length buffer is sent back. The Rx buffer will be updated after the requested buffer is sent back. The function returns the length of the data if it is successful and 0 otherwise.

9.4.2 Prototype

```
uint16 HalUARTRead (uint8 port,
                    uint8 *buf,
                    uint16 length);
```

9.4.3 Parameter Details

port – specified serial port that data will be read. (Check UART Ports Table)

buf – pointer to the buffer of the data.

length – requested length.

9.4.4 Return

Returns the length of the read data or 0 otherwise.

9.5 HalUARTWrite ()

9.5.1 Description

This function writes a buffer of specific length into the serial port. The function will check if the Tx buffer is full or not. If the Tx Buffer is not full, the data will be loaded into the buffer and then will be sent to the Tx data register. If the Tx buffer is full, the function will return 0. Otherwise, the length of the data that was sent will be returned.

9.5.2 Prototype

```
uint16 HalUARTWrite (uint8 port,
                     uint8 *buf,
                     uint16 length);
```

9.5.3 Parameter Details

port – specified serial port that data will be read. (Check UART Ports Table)

buf – buffer of the data.

length– the length of the data

9.5.4 Return

Returns the length of the data that is successfully written or 0 otherwise.

9.6 HalUARTPoll()

9.6.1 Description

This routine simulates the polling mechanism for UART.

9.6.2 Prototype

```
void HalUARTPoll (void);
```

9.6.3 Parameter Details

None.

9.6.4 Return

None.

9.7 Hal_UART_RxBufLen()

9.7.1 Description

This function returns the number of bytes currently in the Rx buffer.

9.7.2 Prototype

```
uint16 Hal_UART_RxBufLen (uint8 port);
```

9.7.3 Parameter Details

port – serial port whose Rx buffer length is requested. (Check UART Ports Table)

9.7.4 Return

16 bit size of the Rx buffer.

9.8 Hal_UART_TxBufLen()

This function returns the number of bytes currently in the Tx buffer. This function is supported only by MSP platforms.

9.8.1 Description

This function returns the number of bytes currently in the Tx buffer.

9.8.2 Prototype

```
uint16 Hal_UART_TxBufLen (uint8 port);
```

9.8.3 Parameter Details

port – serial port whose Tx buffer length is requested. (Check UART Ports Table)

9.8.4 Return

16 bit size of the Tx buffer.

9.9 Hal_UART_FlowControlSet()

9.9.1 Description

This function enables or disables the flow control for the UART. This function is supported only by MSP platforms and the function is left blank in MSP platforms as well. Customers should fill the function body for MSP platforms to add implementation-specific flow control code.

9.9.2 Prototype

```
void Hal_UART_FlowControlSet  (uint8 port,
                               uint8 status);
```

9.9.3 Parameter Details

port – specified serial port, for which flow control will be set. (Check UART Ports Table)

status – `TRUE or FALSE,` enable or disable the flow control

9.9.4 Return

None.

9.10 HalUARTSuspend()

9.10.1 Description

This function aborts UART when entering sleep mode.

9.10.2 Prototype

```
void HalUARTSuspend (void);
```

9.10.3 Parameter Details

None.

9.10.4 Return

None.

9.11 HalUARTResume()

9.11.1 Description

This function resumes UART after wakeup from sleep.

9.11.2 Prototype

```
void HalUARTResume (void);
```

9.11.3 Parameter Details

None.

9.11.4 Return

None.

9.12 Constants

9.12.1 UART Ports

Port	Description
HAL_UART_PORT_1	UART port 1

HAL_UART_PORT_2	UART port 2

9.12.2 Baud Rates

Parameter	Desription
HAL_UART_BR_9600	Baud rate 9600 bps
HAL_UART_BR_19200	Baud rate 19200 bps
HAL_UART_BR_38400	Baud rate 38400 bps
HAL_UART_BR_57600	Baud rate 57600 bps
HAL_UART_BR_115200	Baud rate 115200 bps

9.12.3 Parity

Parameter	Desription
HAL_UART_NO_PARITY	No parity
HAL_UART_ODD_PARITY	Odd parity
HAL_UART_EVEN_PARITY	Even parity

9.12.4 Stop Bits

Parameter	Desription
HAL_UART_ONE_STOP_BIT	Stop Bits 1
HAL_UART_TWO_STOP_BITS	Stop Bits 2

9.12.5 Status

Parameter	Desription
HAL_UART_SUCCESS	Success
HAL_UART_MEM_FAIL	Fail to allocate memory
HAL_UART_BAUDRATE_	Baud rate is bad

9.12.6 Callback Events

Event	Desription
HAL_UART_RX_FULL	Rx Buffer is full

HAL_UART_RX_ABOUT_FULL	Rx Buffer is at **maxRxBufSize - flowControlThreshold**
HAL_UART_RX_TIMEOUT	Rx is idle for **idleTimout** time
HAL_UART_TX_FULL	Tx Buffer is full
HAL_UART_TX_EMPTY	Tx Buffer is free to write more data

10. PA/LNA Service

A CC2591 PA/LNA may be added to designs using the CC2520. The CC2591 is a range extender for all existing and future low-power 2.4-GHz RF transceivers, transmitters, and SoC products from Texas Instruments. The CC2591 increases the link budget by providing a PA for improved output power and a LNA with a low noise figure for improved receiver sensitivity. When an EM module has a CC2591 installed, the HAL_PA_LNA compiler switch must be globally defined.

10.1 HAL_PA_LNA_RX_LGM()

10.1.1 Description

This macro selects CC2591 RX low gain mode. To use this macro, "hal_board.h" must be included in your application.

10.1.2 Prototype

```
#define HAL_PA_LNA_RX_LGM()
```

10.1.3 Parameter Details

None.

10.1.4 Return

None.

10.2 HAL_PA_LNA_RX_HGM()

10.2.1 Description

This macro selects CC2591 RX high gain mode. To use this macro, "hal_board.h" must be included in your application.

10.2.2 Prototype

```
#define HAL_PA_LNA_RX_HGM()
```

10.2.3 Parameter Details

None.

10.2.4 Return

None.

11. I2C Service

This service supports I2C data read and write to a peripheral device. Note that support of this service is limited to RemoTI platforms as of today. RemoTI is Texas Instruments product offering for ZigBee RF4CE standard compliant platform.

11.1 HalI2CInit ()

11.1.1 Description

This I2C initialization function is called once at the startup. This function has to be called before any other I2C functions can be called. It enables I2C hardware resources to be initialized with both required and optional parameters.

11.1.2 Prototype

```
void HalI2CInit (void)
```

11.1.3 Parameter Details

None.

11.1.4 Return

None.

11.2 HalI2CReceive ()

11.2.1 Description

This function polls and receives data into a buffer from an I2C slave device. This function is a blocking function.

11.2.2 Prototype

```
int8 HalI2CReceive(uint8 address,
                   uint8 *buf
                   uint16 len);
```

11.2.3 Parameter Details

address – 8-bit address of the slave device

buf – target array for read bytes

len – maximum number of bytes to read

11.2.4 Return

Returns zero when successful and non-zero when receipt of data failed.

11.3 HalI2CSend ()

11.3.1 Description

This function sends buffer contents to an I2C slave device. This function is a blocking function.

11.3.2 Prototype

```
int8 HalI2CSend(uint8 address,
                uint8 *buf
                uint16 len);
```

11.3.3 Parameter Details

address – 8-bit address of the slave device

buf – pointer to buffered data to send

len – number of bytes in buffer

11.3.4 Return

Returns zero when successful and non-zero when transmission of data failed.

11.4 Constants

None.

12. IR Signal Generation Service

This service support IR signal generation in a particular format. Note that support of this service is limited to RemoTI platforms as of today.

12.1 HalIrGenInit()

12.1.1 Description

This initialization function is called once at the startup. This function has to be called before any other IR signal generation function can be called. It enables IR signal generation hardware resources to be initialized with both required and optional parameters.

12.1.2 Prototype

```
void HalIrGenInit (void);
```

12.1.3 Paraneter Details

None.

12.1.4 Return

None.

12.2 HalIrGenCommand()

12.2.1 Description

This routine generates an IR signal format corresponding to a command. This function is a non-blocking function and the signal generation completion will be notified by a callback function. Application shall not call HalIrGenCommand() before getting the callback function call since last HalIrGenCommand() call.

12.2.2 Prototype

```
void HalIrGenCommand (halIrGenCmd_t command );
```

12.2.3 Parameter Details

command – An upto 32 bit command. The data type of this parameter is determined by HAL_IRGEN_CMD_LENGTH compile flag which indicates the length of a command.

12.2.4 Return

None.

12.3 HalIrGenComplete()

12.3.1 Description

This function is a callback function that a user application has to define. This function notifies completion of IR signal generation corresponding to a command byte that was initiated by HalIrGenCommand() call.

12.3.2 Prototype

```
void HalIrGenComplete (void);
```

12.3.3 Paraneter Details

None.

12.3.4 Return

None.

12.4 Constants

None.