Application Report
SWRA193 – September 2015

HAL Driver Application Programming Interface (API)

ABSTRACT

This document describes the application programming interface for HAL Drivers release with RemoTI software development kit. 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.

Contents

List of Tables

1 Drivers API Overview

1.1 Function Calls

1.1.1 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.

1.1.2 Service Access Function Calls

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

1.1.3 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.

1.2 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.

1.3 Acronyms and Definitions

Table 1. Acronyms and Definitions

Acronym	Definition
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 Service Access Function Calls

Table 2. Summary List

Table 2. Summary List (continued)

3 ADC Services

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

HalAdcInit()

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.

Prototype `void HalAdcInit (void)`

Paramater Details None

Return None

HalAdcRead()

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

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

Paramater Details

- `channel` – Input channels (see Table 3)
- `resolution` – Resolution of the conversion (see Section 3.1.2)

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

HalAdcSetReference()

Description This function sets the reference voltage for the ADC and initializes the service.

Prototype `void HalAdcSetReference (uint8 reference)`

Paramater Details

- `reference` – Reference voltage to be used by the ADC

Return None

HalAdcCheckVdd()

Description Check for minimum Vdd specified.

Prototype `bool HalAdcCheckVdd(uint8 vdd)`

Paramater Details

- `vdd` – Board-specific Vdd reading to check for

Return TRUE, if the Vdd measured is greater than the 'vdd' minimum parameter FALSE if not.

3.1 Constants

3.1.1 Channels

Table 3. HAL ADC Channels

Parameter	Description
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

3.1.2 Resolutions

Table 4. HAL ADC Resolutions

Parameter	Description
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.

HalLcdInit()

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.

Prototype `void HalLcdInit (void);`

Paramater Details None

Return None

HalLcdWriteString()

Description This routine writes a string to the LCD.

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

Paramater 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 (see Section 4.1.1).

Return None

HalLcdWriteString()

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

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

Paramater Details

- `value` – Value that will be displayed on the LCD
- `radix` – 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 (see Section 4.1.1).

HalLcdWriteScreen()

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

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

Paramater 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

Return None

HalLcdWriteStringValue()

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

Prototype

```
void HalLcdWriteStringValue( char *title, uint16 value, uint8 format, uint8 line
);
```

Paramater 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` – Line number where the string (title and value) will be displayed on the LCD display

Example

```
HalLcdWriteStringValue (“Count:”, 180, 10, 2);
```

Display Count: 180 (on line 2 of the LCD display)

Return None

HalLcdWriteStringValueValue()

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

Prototype

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

Paramater Details

- `title` – String that will be displayed on the LCD display
- `value1` – First value that will be displayed following the “title”
- `format1` – Format of the 'value1'. It can be 8, 10, and 16 (Octal, Decimal and Hex)
- `value2` – Second 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

Return None

HalLcdDisplayPercentBar()

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

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

Paramater 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 [||||||||50+]

Return None

4.1 Constants

4.1.1 Options

Table 5. HAL LCD 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.

HalLedInit()

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.

Prototype `void HalLedInit (void);`

Paramater Details None

Return None

HalLedSet()

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.

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

Paramater Details

- `led` – Bit mask of LEDs to be turned ON (see Section 5.1.1)
- `mode` – New mode for the LEDs (see Section 5.1.2)

Return None

HalLedBlink()

Description	This function will blink the given LEDs based on the provided parameters.
Prototype	`void HalLedBlink (uint8 leds, uint8 numBlinks, uint8 percent, uint16 period);`
Paramater Details	• `leds` – Bit mask of leds to be blinked (see Section 5.1.1) • `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.
Return	None

HalLedGetState()

Description	This function returns the current state of the LEDs.
Prototype	`uint8 HalLedGetState (void);`
Paramater Details	None
Return	None

HalLedEnterSleep()

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.
Prototype	`void HalLedEnterSleep (void);`
Paramater Details	None
Return	8-bit contains the current state of the LEDs. Each bit indicates the corresponding LED status.

HalLedExitSleep()

Description	This function restores the original state of the LEDs before the device entered sleep mode.
Prototype	`void HalLedExitSleep (void);`
Paramater Details	None
Return	None

5.1 Constants

5.1.1 LEDs

Table 6. HAL 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

5.1.2 Modes

Table 7. HAL LED 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 and joysticks.

HalKeyInit()

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

Prototype

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

Paramater Details None

Return None

HalKeyConfig()

Description

This function is used to configure the keys, switches and 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 100 ms. Keys, switches and joystick will be polled every 100 ms . If interrupt is used, an ISR will handle the case. There is a delay of 25 ms after the interrupt occurs to eliminate de-bounce.

Prototype

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

Paramater Details

- `interruptEnable` – TRUE or FALSE. Enable or disable the interrupt. If interrupt is disabled, the keys, switches and joysticks will be polled. Otherwise, the keys, switches and 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 and joystick that will cause the callback when it's triggered. Key code varies per platform. For the typical key definitions for evaluation/development board, see Section 6.1.1.
  - `state` – Current state of the key, switch and joystick that causes the callback (see Section 6.1.3).

Return

None

HalKeyRead()

Description

This function is used to read the current state of the key, switch and joystick. If the Key Service is set to polling, this function will be called by the Hal Driver Task every 100 ms. If the Key Service is set to interrupt driven, this function will be called by the Hal Driver Task 25 ms 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.

Prototype

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

Paramater Details

None

Return

Return the key code. Key code varies per platform (see Section 6.1.1) for typical key code definitions for evaluation/development board). Key code could be comprised of bits indicating individual key and 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.

HalKeyEnterSleep()

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.

Prototype

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

Paramater Details

None

Return

None

HalKeyExitSleep()

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.

Prototype

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

Paramater Details None

Return None

HalKeyPoll()

Description This routine is used internally by hal driver.

Prototype

```
void HalKeyPoll ( void );
```

Paramater Details None

Return None

6.1 Constant

6.1.1 Keys

Table 8. HAL KEY 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

6.1.2 Joystick

Table 9. HAL KEY 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

6.1.3 States

Table 10. 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.

HalSleep()

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

Prototype `void halSleep(uint16 osal_timeout)`

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

Return None

HalSleepWait()

Description This routine performs a blocking wait.

Prototype `void halSleepWait(uint16 duration)`

Paramater Details `duration` – Duration of the blocking in micro seconds

Return None

8 Timer Service

This service supports up to four hardware timers, two 8-bit timers and two 16-bit timers.

8.1 Operation Modes

Timer service supports Normal and Clear Timer on Compare (CTC) 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 three channels for output compare and one channel for input capture or overflow channel mode for 16-bit timers. For 8-bit timers, the Timer Service has one channel for output capture or overflow channel mode.

8.3 Interrupts/Channel Modes

Timer service supports three interrupt sources: overflow, output compare, and input capture. However, only 16-bit timers support input capture.

HalTimerInit()

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.

Prototype `void HalTimerInit (void)`

Paramater Details `duration` – Duration of the blocking in micro seconds

Return None

HalTimerConfig()

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

Prototype

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

Paramater Details

- `timerId` - HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers (see Section 8.4.1)
- `opMode` – Operation Mode, can be Normal or CTC (see Section 8.4.4).
- `channel` – Channel to be configured (see Section 8.4.2)
- `channelMode` – The channel mode, Input Capture Mode, Output Compare Mode or Overflow Mode (see Section 8.4.3)
- `intEnable` – TRUE or FALSE – Enable and 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 Section 8.4.1).
  – `channel` – The channel where the interrupt occurs (see Section 8.4.2)
  – `channelMode` – Interrupt source of the event

Return Status of the configuration (see Section 8.4.6)

HalTimerStart()

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.

Prototype

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

Paramater Details

- `timerId` – HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers (see Section 8.4.1).
- `timerPerTick` - Number of micro-seconds per tick

Return If the Timer Service is not configured, `HAL_TIMER_NOT_CONFIGURED` will be returned. Otherwise, `HAL_TIMER_OK` will be returned (see Section 8.4.6)

HalTimerStop()

Description This function is called to stop a timer/counter.

Prototype

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

Paramater Details

- `timerId` – HAL_TIMER_X, identification of the timer that will be stopped (see Section 8.4.1)
- `timerPerTick` - Number of micro-seconds per tick

Return If the timerId is not valid, `HAL_TIMER_INVALID_ID` will be returned. Otherwise, `HAL_TIMER_OK` will be returned (see Section 8.4.6).

HalTimerTick()

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()* sends 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()*.

Prototype

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

Paramater Details None

Return None

HalTimerInterruptEnable()

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

Prototype

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

Paramater Details

- `timerId` – HAL_TIMER_X, identification of the timer. Different processors support different numbers of timers (see Section 8.4.1).
- `channelMode` - The channel mode, Input Capture Mode, Output Compare Mode or Overflow Mode (see Section 8.4.3).
- `enable` – TRUE or FALSE

Return If the timerId is not valid, `HAL_TIMER_INVALID_ID` will be returned. Otherwise, `HAL_TIMER_OK` will be returned (see Section 8.4.6).

8.4 Constants

8.4.1 Timer ID

Note that description of timer ID enumeration is specific to CC253x. The actual timer mapped to the ID may vary per platform.

Table 11. HAL TIMER Timer ID

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

8.4.2 Channels

Table 12. HAL TIMER 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

8.4.3 Channel Modes

Table 13. HAL TIMER 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

8.4.4 Operation Modes

Table 14. HAL TIMER Operational 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.

8.4.5 Prescale

Table 15. HAL TIMER Prescale 8 Bit

Prescale – 8 Bit	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

Table 16. HAL TIMER Prescale 16 Bit

Prescale – 16 Bit	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

8.4.6 Status

Table 17. HAL TIMER 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…and so forth. It also reports framing and overrun errors.

HalUARTInit()

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.

Prototype `void HalUARTInit (void)`

Paramater Details None

Return None

HalUARTOpen()

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.

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

Paramater Details

- `port` – Specified serial port to be opened (see Section 9.1.1)
- `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;
  halUARTBufControl_t rx;
  halUARTBufControl_t 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` – Baud rate of the port to be opened (see Section 9.1.2)
- `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 CC253x 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 CC253x 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 CC253x, 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 or disable interrupt. It can be set as TRUE or FALSE. TRUE value will enable the interrupt and FALSE value will disable the interrupt. For CC253x, 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 (see Section 9.1.1).
  – `event` – Event that causes the callback (see Section 9.1.6).

Return Status of the function call (see Section 9.1.5).

HalUARTClose()

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.

Prototype `void HalUARTClose (uint8 port);`

Paramater Details `port` - Specified serial port to be closed (see Section 9.1.1)

Return None

HalUARTRead()

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.

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

Paramater Details

- `port` - Specified serial port where data will be read (see Section 9.1.1)
- `buf` – Pointer to the buffer of the data
- `length` – Requested length

Return Returns the length of the read data or 0 otherwise.

HalUARTWrite()

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.

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

Paramater Details

- `port` – Specified serial port where data will be read (see Section 9.1.1)
- `buf` – Buffer of the data
- `length` – Length of the data

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

HalUARTPoll()

Description This routine simulates the polling mechanism for UART.

Prototype `void HalUARTPoll (void);`

Paramater Details None

Return None

HalUARTSuspend()

Description This function aborts UART when entering sleep mode.

Prototype

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

Paramater Details None

Return None

HalUARTBusy()

Description Query the UART hardware & buffers before entering PM mode 1, 2 or 3.

Prototype

```
uint8 HalUARTBusy (void);
```

Paramater Details None

Return TRUE if the UART H/W is busy or buffers are not empty; FALSE otherwise.

HalUARTResume()

Description This function resumes UART after wakeup from sleep.

Prototype

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

Paramater Details None

Return None

9.1 Constants

9.1.1 UART Ports

Table 18. HAL UART Ports

Port	Description
HAL_UART_PORT_1	UART port 1
HAL_UART_PORT_2	UART port 2

9.1.2 Baud Rates

Table 19. HAL UART Baud Rates

Parameter	Description
HAL_UART_BR_57600	Baud rate 57600 bps
HAL_UART_BR_115200	Baud rate 115200 bps
HAL_UART_BR_230400	Baud rate 230400 bps

9.1.3 Parity

Table 20. HAL UART Parity

Parameter	Description
HAL_UART_NO_PARITY	No parity
HAL_UART_ODD_PARITY	Odd parity
HAL_UART_EVEN_PARITY	Even parity

9.1.4 Stop Bits

Table 21. HAL UART Stop Bits

Parameter	Description
HAL_UART_ONE_STOP_BIT	Stop Bits 1
HAL_UART_TWO_STOP_BITS	Stop Bits 2

9.1.5 Status

Table 22. HAL UART Status

Parameter	Description
HAL_UART_SUCCESS	Success
HAL_UART_UNCONFIGURED	Uart not configured
HAL_UART_NOT_SUPPORTED	Not supported
HAL_UART_MEM_FAIL	Fail to allocate memory
HAL_UART_BAUDRATE_ERROR	Baud rate is bad

9.1.6 Callback Events

Table 23. HAL UART Callback Events

Event	Description
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 CC253x. 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.

HAL_PA_LNA_RX_LGM()

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

Prototype `#define HAL_PA_LNA_RX_LGM()`

Paramater Details None

Return None

HAL_PA_LNA_RX_HGM()

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

Prototype `#define HAL_PA_LNA_RX_HGM()`

Paramater Details None

Return None

11 I2C Service

This service supports I2C data read and write to a peripheral device (MASTER MODE), or from a host (SLAVE MODE). 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.

Master Mode or Slave mode is chosen with the compilation flag `HAL_I2C_MASTER` = `TRUE or FALSE`.

HalI2CInit()

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.

In master mode, the initialization set up the clock speed only.

In slave mode, the initialization set up the address that the device will answer to, and the necessary callback function for the application to parse the incoming traffic.

Prototype

In Master mode:

```
#define HAL_PA_LNA_RX_LGM()
```

In Slave Mode:

```
void HalI2CInit(uint8 address, i2cCallback_t i2cCallback)
```

Paramater Details

- `clockrate` – In master mode, specify the clock rate (I2C Master Clock speed)
- `address` – in slave mode I2C address the device will answer to
- `callBackFunc` – This callback is called when a Master Tx is ready to be read. uint8 i2cCallback (uint8 cnt);
  - `cnt` - Zero indicates Master read request and slave Tx buffer empty. Non-zero indicates the number of bytes received from a Master write, this data is in the slave Rx buffer, ready to be read.
  - `output` – N/A when cnt is non-zero; otherwise TRUE if I2C should continue to clock stretch, FALSE to send a zero with last byte indication.

Return

None

HalI2CRead()

Description

This function read from the bus.

Prototype

In Master mode:

```
i2cLen_t HalI2CRead(uint8 address, i2cLen_t len, uint8 *pBuf);
```

In Slave Mode:

```
i2cLen_t HalI2CRead(i2cLen_t len, uint8 *pBuf);
```

Paramater Details

- `address` – In master mode, 8-bit address of the slave device to read from
- `pbuf` – Target array for read bytes
- `len` – Maximum number of bytes to read. Depending of the maximum buffer size allowed (`HAL_I2C_BUF_MAX`), can be a uint8 or a uint16.

Return

The number of bytes successfully read.

HalI2CWrite()

Description This function write to the bus.

Prototype In Master mode:

`i2cLen_t HalI2CWrite(uint8 address, i2cLen_t len, uint8 *pBuf;`

In Slave Mode:

`i2cLen_t HalI2CWrite(i2cLen_t len, uint8 *pBuf);`

Paramater Details

- `address` – In master mode, 8-bit address of the slave device to read from
- `pbuf` – Pointer to buffered data to send
- `len` – Number of bytes in buffer

Return In master mode: The number of bytes successfully written.

In slave mode: The number of bytes setup to write.

HalI2CEnterSleep()

Description I2C register are not save in PM2 and PM3 mode. This function saves them just before going to sleep. Valid for both Master and Slave mode.

Prototype `void HalI2CEnterSleep (void);`

Paramater Details None

Return None

HalI2CExitSleep()

Description Restore I2C state after exiting a Power Mode. Valid for both Master and Slave mode.

Prototype `void HalI2CExitSleep (void);`

Paramater Details None

Return None

HalI2CReady2Sleep()

Description Determine whether the I2C is ready to sleep. Valid for both Master and Slave mode.

Prototype `uint8 HalI2CReady2Sleep (void);`

Paramater Details None

Return 1 if the I2C is ready to sleep; 0 otherwise.

11.1 Constants

11.1.1 I2C Master Clock Speed

Table 24. I2C Master Clock Speed

Port	Description
i2cClock_33KHZ	33 kHz
i2cClock_123KHZ	123 kHz
i2cClock_144KHZ	144 kHz
i2cClock_165KHZ	165 kHz
i2cClock_197KHZ	197 kHz
i2cClock_267KHZ	267 kHz
i2cClock_533KHZ	533 kHz

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		