Si4010 API Additional Library Description

1. Purpose

This document describes an additional library of the Application Programming Interface (API) for the Si4010 firmware which is not included in ROM.

The library described can only be used with ROM version 0x02rr (02.rr in binary coded decimal notation) as returned by the `wSys_GetRomId()` function. The major ROM version number must be 0x02. The minor version rr can be any number.

The document describes additional API functions not present in ROM, provided in a form of a linkable library for Keil toolchain, and how to use them.

2. Building an Application with Additional Library

2.1. Base Files Needed for Building an Application

The main API document (AN370) describes what files are needed to build a customer application. The `api_add` library requires those files to be present. Table 1 lists these files along with their descriptions. Table 1 is the same as in the original API document and is duplicated here for reference.

Table 1. Base Files

File Name	Description
si4010_types.h	Header file that declares type definitions used in the API. See Type Definitions section for more information.
si4010.h	Device header file that declares all SFR and XREG registers. It also defines masks and bit indices which are to be used when accessing fields within registers. This is a C header file. Must be included in all C files using the Si4010.
si4010.inc	Same as si4010.h, but for use with an assembler. This file should be included in all assembly source files while using Si4010.
si4010_api_rom.h	C header declaring all the API functions. Must be included in the application which uses the API.
startup.a51	Simplified assembly startup file for Keil and Raisonance tool-chains. Customers may want to modify this file. It must be included in the application build.
si4010_rom_keil.a51	Assembly ROM symbol map that must be assembled and linked into the application build if the API functions are being used. It tells the linker the API functions are located in ROM. This file is tailored to the Keil toolchain. It also includes references to some of the Keil library functions.

Table 1. Base Files (Continued)

File Name	Description
si4010_rom_all.a51	Same as above, but for any other toolchain. It can also be used with the Keil toolchain if the user does not desire to use Keil library functions in ROM. With Keil toolchain use either this one or the one above, but not both.
si4010_data.c	Data file related to the si4010.h defining the XREG register in XDATA area. This file must be included in the application build.
si4010_link.c	File with dummy array variables to force linker to avoid DATA and IDATA spaces used and reserved by ROM API. If this file is not used the linker area avoidance directive must be used. Users may want to augment this file to notify linker that the end of CODE/XDATA RAM is also reserved for API use. A commented section on how to achieve that is included at the end of the file.
If the Si4010 revision B silicon (Si4010-B1-GS or Si4010-B1-GT) is used, then the following file is also required:	
si4010_fix_rom_keil.lib	This file is needed only for Rev B. Keil library file containing the fixed `vFCast_FskAdj` function. Without it the frequency modulation will not work. This file must be included in the application build if FSK modulation is used (i.e. the vFCast_FskAdj function is called). In that case, only the Keil toolchain is supported.

These files are in the directory

`...\common\src\`

in the Si4010 installation tree.

2.2. Additional Library Files Needed for Building an Application

While using the API additional library there are additional files the user has to include in the files or build to be able to use the functions in the library. See Table 2.

Table 2. Additional Library Files

File Name	Description
si4010_api_add.h	Header file that declares additional library types and functions. This file resides in the …\common\src\ directory.
si4010_api_add_keil.lib	Keil library to be included in the project for function linking. The library is model agnostic, but it was compiled using the SMALL memory model. It is recommended to use the SMALL model. Only the Keil toolchain is supported at this time. This file resides in …\common\lib\ directory.

2.3. Compiling an Application

Refer to the section of building the application in the main API document. This section just reiterates some building steps and provides additional information for API additional library use.

To use the Si4010 API to build a user application in C, the user has to do the following:

1. To be able to use the Silicon Laboratories IDE for debugging, the user must use the Keil BL51 linker or toolchain with the standard OMF-51 output file format. The user cannot use the LX51 linker, since the Keil proprietary output format, OMF-2, is not understood by the Silicon Laboratories IDE.
2. Add the path `...\common\src` to the C compiler include directive. This is where the Si4010 files are installed.
3. Include `si4010.h` and `si4010_api_rom.h` headers in every C source file of the application. These files include the si4010_types.h header automatically:
   ```
   #include "si4010.h"
   #include "si4010_api_rom.h"
   #include "si4010_api_add.h"
   ```
4. Add the following files, which have to be assembled or compiled, into the application build. These files must be compiled and/or linked with the user application:
   ```
   si4010_rom_keil.a51
   si4010_data.c
   si4010_link.c
   startup.a51
   si4010_api_add_keil.lib
   ```

For the Keil toolchain if the user does not desire to use some of the Keil library functions in the ROM then the file `si4010_rom_all.a51` must be used with the Keil toolchain instead.

5. Build the application as usual.

2.4. Using the Library Functions

The additional library functions discussed here are distributed as a library. Therefore, when the functions are used, the linker will pull the functions from the library and link them with the user code to be put in the CODE/XDATA RAM. There is only a little over 4 kB of RAM available. The library functions are of varying sizes.

The library functions also require their DATA/IDATA and XDATA variables to be stored in the respective memories. Those variables are not placed on fixed memory locations in the library. Instead, it is up to the linker to assign the proper locations of those variables and share the locations with the user application based on the calling tree to save space.

The user has to provide DATA/IDATA and XDATA space where these variables can be placed. It is recommended that the user uses the SMALL memory model, although it is not strictly required.

2.5. Stack Size Requirements

Table 3 shows the additional stack requirements when the user code is calling an API library function. The number of bytes in the table is in addition to the 2 bytes return address storage requirements for the return address to be stored on top of the stack when the function is called. For example, if the function is not using any additional stack storage (not calling any other function, and not using PUSH/POP instructions), then the function internal stack requirement is listed as 0.

Note: The maximum stack size requirement is determined by the interrupt service routines and if the application is using one or two interrupt priorities. The worst case stack requirement would come from the application using two levels of interrupt levels and lower priority ISR was interrupted by the higher priority ISR.

The user is required to leave at least an additional 4 bytes of stack space, 2 bytes as a guard and 2 bytes for possible Silicon Labs use in the future.

If the function additional stack requirement is zero, then the table shows " —" for reading clarity.

See the main API document for the stack size requirements of the ROM API functions.

Table 3. Stack Requirements

Function	Internal stack use [bytes]
fFc_GetFrfRatio	4
fFc_CountFintOverFrf	6
bFStep_Collect	—
vFStep_Apply	6
vFStep_ApplyFast	4
bNvm_LoadBlock	4
wSys_GetApiAddId	—

3. Additional Library Module Descriptions

The following sections describe the library functions in detail.

3.1. Frequency Counter Module

The additional functions for frequency counter provide a way to use the known RF frequency set by the `vFcast_Tune()` function as a reference to measure other frequencies, which are inputs to the interval counter part of the frequency counter.

3.1.1. Frequency Counter Module Additional Functions

fFc_GetFrfRatio

Description: Returns a ratio of the frequency counter input frequency, which is LC or divider output, to the actual output RF frequency set by the `vFCast_Tune()` function. This function just looks at the current hardware setting of the chip and returns the ration factor. It does not actually run a frequency counter.

Note: This is a helper function. The user is advised to use the `fFc_CountFintOverFrf()` function instead in the user application.

In order to calculate the frequency of the "slow" clock, f_interval, which is the frequency input to the interval counter based on the frequency counter mode for a known RF carrier f_RF, the user could use a code like this after the frequency counter was run:

```
/* Calculate the interval counter count number based on
 * current hardware setting */
lIntervalNum = (2 + (FC_INTERVAL & 0x01))
                  * (1UL << (FC_INTERVAL >> 1));

/* Calculate the frequency of the interval counter input signal
 * based on the knowledge of the RF signal as input to the
 * frequency counter. */
fFint = (fFrf * fFc_GetFrfRatio() * lIntervalNum)
           / lFc_GetCount();
```

Inputs: None

Outputs: f_RF to f_LC or f_DIV ratio: (float) Ratio between the f_RF frequency and the frequency of the currently selected input to the frequency counter, which is f_LC or f_DIV.

fFc_CountFintOverFrf

Description: Run frequency counter to measure the interval counter input frequency f_interval with respect to the output f_RF frequency and return the ration f_interval / f_RF after the single frequency counter run.

This function always forces the f_LC to be the input to the main frequency counter.

This function requires that the frequency counter is set up prior to calling this function by the `vFc_Setup()` function. The user application is responsible for setting the frequency counter before calling this function. This function runs the frequency counter and calculates the f_interval / f_RF ratio.

Note: If the frequency counter does not have either interval frequency input f_interval of the main frequency counter input f_LC running then this function will stall, waiting indefinitely for the frequency counter to finish the calculation. It is the responsibility of the main application to set up the frequency counter such that the frequency count will finish or provide a timer and associated interrupt service routine to set the FC_CTRL.FC_DONE bit to 1 to relieve this function from the infinite loop waiting for the frequency counter to be done. The same applies to the `vFCast_Tune()` usage, since it uses the frequency counter in the same fashion.

Example of usage of this function:

```
/* Setup frequency casting .. needed once per boot */
  vFCast_Setup();

/* Tune the RF frequency to desired output. The function
 * uses frequency counter internally. */
fFrf = 435.23e6;  /* Desired RF frequency [Hz] */
   vFCast_Tune( fFrf );

/* Prepare frequency counter to measure 24MHz clock.
 * In this example we measure the 24MHz clock directly. The system
 * clock can be division of that. One can also measure the
 * current system clock instead.
 * The value 29 results in 3 x 2^14 = 49152 interval counter
 * cycles for the main counter to count the LC oscillator
 * output. For divides system clock as input, bFc_ModeClkSys_c,
 * one has to use lower numbers not to overflow the main
 * frequency counter. Adjust the interval value according
 * to the application particular needs. */

vFc_Setup( bFc_ModeClkOsc_c, 29 );


/* Run frequency counter and calculate the oscillator
 * clock frequency based on the known output frequency.
 * The value fFosc should be close to 24 MHz. */

fFosc = fFrf x fFc_CountFintOverFrf();  /* [Hz] */

/* Subsequent runs do not need to set up the frequency counter.
 * However, the frequency counter must be set up again after
 * vFCast_Tune() is called. */

fFosc = fFrf * fFc_CountFintOverFrf();  /* [Hz] */
```

Inputs: None

Outputs: f_interval / f_RF ratio: (float) Ration of the interval counter clock frequency to the output RF frequency after the frequency counter was run once.

3.2. NVM Module

The non-volatile memory (NVM) module copies formatted (composed) code and data blocks from the NVM. The data within NVM block can be copied to CODE/XDATA RAM, as well as to the internal DATA/IDATA RAM.

The sole purpose of this module is for the user to load user application code and data overlays from the NVM by the application at runtime.

If using overlays, the user application consists of the main program (User Boot), which is loaded to the CODE/XDATA RAM upon boot and run. The application can be written, then it can load new or additional code and/or data from the NVM in a form of an overlay. That mechanism will allow the use of the 7 kB of NVM available code/data for user application, while having only 4 kB of RAM from which the code can be run.

The function provided is a wrapper around the `vNvm_McEnableRead()`, `bNvm_CopyBlock()`, and `vNvm_McDisableRead()` functions to ease the user loading of the overlay at runtime.

It usually takes 3.6 ms per 1 kB of NVM data to be copied from the NVM by the `bNvm_CopyBlock()` function.

Note: It is highly recommended that interrupts are disabled around the call to the `bNvm_LoadBlock()` and function. If not possible, then it is highly recommended that the most time-consuming interrupts are disabled. The interrupt disruption of the NVM load process should be kept at a minimum in the order of units of microseconds.

3.2.1. User Overlay Loading and Error Checking

The user can load the NVM block by application at runtime. That block had to be put into the NVM at a specific address using the NVM burner. The NVM composer/burner provides the user the length (in bytes) of the block in the NVM. Therefore, for each block starting address the user will know the exact length of the block in NVM. When the copy is done, the NVM address returned by the `wNvm_GetAddr()` call is the sum of the block start address and the block length.

For example, the user wants to put the overlay block at NVM address 0xF400 and the NVM composer/burner returns NVM block length at 0x345. The NVM overlay load error checking in the user application could be done as follows:

```
/* Load the overlay block */
EA = 0;  /* Disable interrupts */
  bStatus = bNvm_LoadBlock( 0xF400 );
wNvmAddr = wNvm_GetAddr();
EA = 1;  /* Enable interrupts */

/* Check the error status */
bError = 0;  /* Clear error status */
if ( 0xFF == bStatus
     || (0xF400U + 0x345U) != wNvmAddr )
{
  bError = 1;  /* NVM load block failed */
}
```

3.2.2. NVM Module Functions

bNvm_LoadBlock

Description: Wrapper function around the `vNvm_McEnableRead()`, `bNvm_CopyBlock()`, and `vNvm_McDisableRead()` functions to ease the user loading of the overlay at runtime. See the `bNvm_CopyBlock()` function description in the main API user document.

Inputs: wiNvmAddr: (WORD) The 16-bit NVM start address of the NVM block to be copied from the NVM.

Outputs: Return value/status: (BYTE) Return value of the `bNvm_CopyBlock()` call. Either the error flag (0xFF), or the return value in the byte following the block end flag in NVM. For user NVM block (overlay) the NVM block return value is fixed to 1.

3.3. System Module

This module contains a grouping of functions which perform tasks that are too specific and individual to justify creating an entire module for them.

3.3.1. System Module Functions

wSys_GetApiAddId

Description: Returns the version of the API additional library. The value is a 2-byte word value coded in BCD fashion. The upper byte is the major number, the lower byte is the minor number. For example, 0x0201 represents library version 02.01, with the major number being 02 and the minor number being 01.

The upper byte matches the upper byte of the ROM version number as returned by the `wSys_GetRomId()` function. Therefore, the major ROM identification number matches the major library version number. The minor library number is therefore the sequential release number for the library.

For example, if the ROM identification number is 0x02rr then the library identification number will be 0x02mm where rr and mm are unrelated minor numbers for their respective identification numbers.

The ROM and library major identification numbers must match for things to work.

Inputs: None.

Outputs: Library ID: (WORD) Word in binary coded decimal format which holds the library identification. For example, the hexadecimal value 0x0201 translates to the binary coded decimal library version value of 02.01.

3.4. Two Step Frequency Casting Module

This module implements collection of the tuning data generated by the call to the `vFCast_Tune()` function and application of the same data in the main application.

The main function tuning the device to desired frequency, `vFCast_Tune()`, is a complicated function which takes 5-6 ms to execute. If the user desires to switch in between frequencies quickly during transmission then it is possible to run the `vFCast_Tune()` for each desired frequency before transmission, collect and save the tuning parameters, and then quickly apply them during transmission.

The tuning values generated by the call to the `vFCast_Tune()` are stable in a moderate range of temperatures. It is therefore possible to call the `vFCast_Tune()` function separately, collect and record the generated tuning data by calling the `bFStep_Collect()` function, and use the data to set up the chip for transmission in the main application by calling the `vFStep_Apply()` function.

Therefore, during "fast switching tuning" in the main application the user will not call the time-intensive `vFCast_Tune()` function, but a small and fast `vFStep_Apply()` function to apply the tuning data previously generated by the `vFCast_Tune()` function and collected by the `bFStep_Collect()` function.

The user can use overlay to do the initial tuning and then switch to the main application using the collected data.

See the `fstep_demo` example application for how to use the two step tuning process. The application can be found in the directory

…`\fstep_demo\src\`

in the Si4010 installation tree. In the associated directory

…`\fstep_demo\bin\`

there is a Keil project file with proper compiler and linker settings along with Silicon Labs IDE projects.

3.4.1. Two Step Frequency Casting Required Data

Usage of the two step frequency casting requires that the user reserve a 15 byte data array for collecting the data generated by the `vFCast_Tune` function. For convenience, the byte array size constant is provided as #define constant

`bFStep_DataLength_c`

in the

`…\common\src\si4010_api_add.h`

header file. Note that both FStep functions take a pointer to a byte array in XDATA. The data byte array must be located in XDATA. Use of generic 3-byte pointer resulted in a code size increase of over 300 bytes for the user application.

The only thing required is that the same data collected by the `bFStep_Collect()` function is passed to the `bFStep_Apply()` function. See more discussion in the section related to the application of the two step frequency casting mechanism.

3.4.2. Two Step Frequency Casting Module Functions

bFStep_Collect

Description: Collection of data generated by the call to the `vFCast_Tune()` function.

This function must be called immediately after the call `vFCast_Tune()` call. It copies the critical data generated by the frequency casting tuning and puts them to the user-defined 15-byte array. That information is then used by the counterpart function, `vFStep_Apply()`, when the same data is restored and the Single Tx Loop can therefore function correctly as if the `vFCast_Tune()` was called in the main application.

Inputs: pboData: (pointer to BYTE array in XDATA) Pointer to the pre-allocated 15-byte data array where the results are going to be stored.

Outputs: Number of copied bytes: (BYTE) Number of bytes copied to the output array. It will always be 15. This return value is for user convenience and software design consistency. It can be ignored.

vFStep_Apply

Description: Application of data generated by the call to the `vFCast_Tune()` function and collected by the `bFStep_Collect()` function.

In the main user application, instead of calling the `vFCast_Tune()` the user would call this function with the input pointer pointing to the 15-byte data array previously collected by the `bFStep_Collect()` function.

Note that this function should be used in place of the `vFCast_Tune()` function and therefore it behaves identically with respect to the hardware control. Consult the main API document, the Transmission Control Section, for details.

This function forcibly enables the LC oscillator and also forces the DMD TS interrupt to be enabled and the TS to be set for temperature measurement. It leaves the LC oscillator forcibly on!

This function is a wrapper function around the following calls:

```
/* Tune and mimic vFCast_Tune() */
  vFStep_ApplyFast( pbiData );
vDmdTs_RunForTemp( 3 );
vSys_ForceLc();
```

Inputs: pbiData: (pointer to BYTE array in XDATA) Pointer to the previously collected 15-byte data array generated by `vFCast_Tune()` and subsequently filled by the `bFStep_Collect()` function call.

Outputs: None

vFStep_ApplyFast

Description: Application of data generated by the call to the `vFCast_Tune()` function and collected by the `bFStep_Collect()` function.

This function applies the new LC oscillator tuning values, but it does not enable the DMD TS interrupt, and does not change the LC force status.

However, if the LC oscillator is forcibly turned on when this function is called, the function applies the new settings and waits for 125 ms for the LC to stabilize. If the LC is not forcibly turned on then this function returns immediately after it applies new frequency values. This function does not modify the LC oscillator on/off state.

Therefore, this is a lower-level function and it cannot be used in place of the `vFCast_Tune()`. See the `vFStep_Apply()` function for that purpose.

This function can be used to fast tune the LC oscillator when the user knows that the DMD TS is running and does not desire to force the LC oscillator on. For example, new antenna tuning is not required and the user just wants to change frequencies quickly during transmission for slow OOK data rate.

Inputs: pbiData: (pointer to BYTE array in XDATA) Pointer to the previously collected 15-byte data array generated by `vFCast_Tune()` and subsequently filled by the `bFStep_Collect()` function call.

Outputs: None.

4. Additional Reference Resources

- Si4010 Data Sheet
- Si4010 Development Kit User’s Guide
- AN370: Si4010 Software Programming Guide

Document Change List

Revision 0.1 to Revision 0.2

- Updated Table 1, “Base Files,” on page 1.

NOTES:

Simplicity Studio

One-click access to MCU tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

www.silabs.com/simplicity

MCU Portfolio
www.silabs.com/mcu

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com