# Broadcast Video Layer 0 API

## Broadcast Video Low-Level Communication Software
## Layer 0 Application Programming Interface

## Features

- Driver initialization
- 3 possible communication modes: USB, SIMU, CUSTomer
- Dynamic communication mode selection
- User-selectable Read/Write tracing
- low-level trace mechanism
- traces tagging
- Level-selectable traces
- 0, 1 or n bytes addressing capability

## Applications

- Digital terrestrial & cable STB and NIM
- Digital terrestrial & cable iDTV set
- Personal Video Recorder (DVD or HDD-based)
- Digital terrestrial PC-TV tuner peripherals
- Portable DVB-T receiver / DVD player
- Any Broadcast Video Product

EVALUATION AND USE OF THIS SOFTWARE IS SUBJECT TO THE TERMS AND CONDITIONS OF THE SOFTWARE LICENSE AGREEMENT IN THE DOCUMENTATION FILE CORRESPONDING TO THIS SOURCE FILE.
IF YOU DO NOT AGREE TO THE LIMITED LICENSE AND CONDITIONS OF SUCH AGREEMENT, PLEASE RETURN ALL SOURCE FILES TO SKYWORKS SOLUTIONS.

# 1 Introduction

All Skyworks Solutions Broadcast Video evaluation boards connect to a PC via a USB interface.
For development purpose, it is possible to use a simulator mode that allows easy checking of the upper layers, in addition to USB communication.
To make easy the customer code development, a customer' communication mode is also included, to wrap the customer low-level I2C access. This mode will need to be adapted to the final i2c communication.

## 1.1 Overview

The architecture of the Layer 0 API is illustrated in the diagram below.
The current document intends to describe the byte-level Layer 0 I2C API functions and how they must be mapped to the final application.
Once the required Layer 0 functions are available to the upper layers, the Broadcast Video APIs and the related example applications can be run easily on the final hardware.

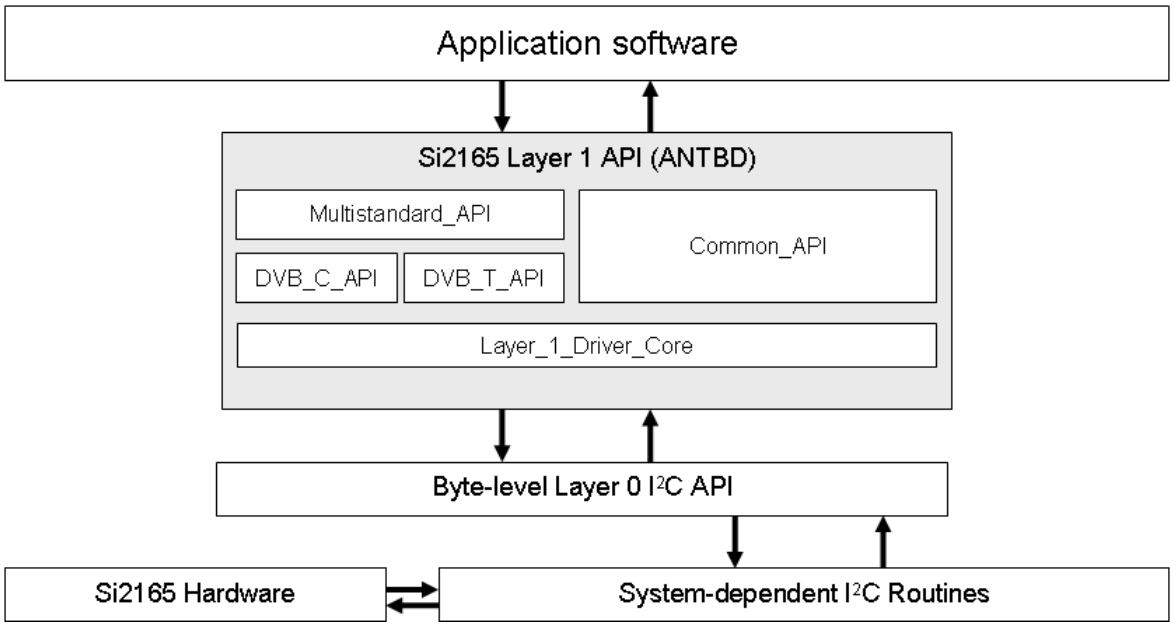



**Figure 1: Software Architecture used by the Si2165 software**

## Table of Contents

## Table of figures

# 2 Change log

## 2.1 As from V5.0.7 (2018/03/15)

In L0_WriteRegister (legacy function for register access):
Typo corrected in L0_WriteRegister. This only had an impact on register-based parts.

## 2.2 As from V5.0.6 (2017/06/19)

In L0_WriteRegister (legacy function for register access):
Changing initial shift value. The previous code did not shift enough bits. This had no impact on the result, but it's cleaner now. (It would only change the written bits if provided an out of range register value)

## 2.3 As from V5.0.5 (201703/29)

Changing some char* to const char* to avoid compilation warnings on some compilers.
In traceToBuffer: using strncat instead of strcat
In SiTraceFunction: using strncpy instead of strcpy, using vsnprintf instead of vsprintf

## 2.4 As from V5.0.4 (2016/09/27)

<correction>[multiple/memory] Removing ucBuffer from global scope. This may create issues for setups with multiple frontends.
ucBuffer is now locally declared in L0_WriteBytes and L0_WriteString (with a size reduced to 100 bytes instead of the previous 1000).

## 2.5 As from V5.0.3 (2015/12/03)

<new_feature>[TS/slave] Update to CypressUSB.dll to V15.49 to enable 'TS SLAVE' feature (using the TS output as with a SoC). Also requires updating the Cypress FW to V15.49.

## 2.6 As from V5.0.2 (2015/11/19)

<compatibility>[Linux/adapter_nr]
In L0_SetAddress: setting adapter_nr as add[15:8] (only if LINUX_I2C_Capability)
When using adapter_nr, provide it from L3 together with the i2c address:
adapter_nr = add[15:8]
i2c address = add[ 7:0]

## 2.7 As from V5.0.1 (2015/10/12)

<compatibility>[Linux/Time] In system_time: now compatible with standard Linux with USB support (using FX2LP) as well as with ST_SDK2 in kernelspace.

### 2.8 As from V5.0.0 (2015/10/05)

<compatibility>[Linux/USB] Adding information on Linux installation allowing using the Cypress FX2LP implemented on the Skyworks EVBs to control the EVBs using USB.

### 2.9 As from V4.0.1 (2015/06/16)

<compatibility>[Linux/gettimeofday] Code compiling under Linux with Cypress EZUSB compatibility.
(requires NO_WIN32 / LINUX / USB_Capability flags and compiling with libCypressUSB.so)
When connected in LINUX_USB mode, L0 traces will show 'LUSB'.

### 2.10 As from V4.0.0 (2015/01/22/)

Adding L0_Cypress_VendorCmd and L0_Cypress_VendorRead (for Cypress chip only, when USB_Capability is defined)
<compatibility>[Tizen/int]
Explicitly declaring all function parameters are 'signed int' instead fo 'int'.
This is because Tizen interpret 'int' as 'unsigned int'
All other compilers interpret 'int' as 'signed int', so this change doesn't affect other compilers.

### 2.11 As from V3.5.1 (2014/11/17)

<improvement> [Code checker/warnings] In L0_WriteString: removing unused pucBuffer.

### 2.12 As from V3.5.0 (2014/09/05)

<improvement>In L0_ReadRegister/L0_WriteRegister: checking input parameter to avoid having iNbBytes = 0.
NB: This would only occur if improper parameters are sent to the functions.
An error message is now generated to inform the user that the input parameters are incorrect.

### 2.13 As from V3.4.9 (not uploaded)

<correction/system_wait> In system_wait: really waiting for the expected amount of time (regression introduced in V3.4.6).

### 2.14 As from V3.4.8 (2014/07/18)

<new_feature>[SiTRACES/<porting>] In Silabs_L0_API.h: defining SiLOGS to SiTRACE when traces are defined and printf otherwise.
This is to be able to get minimal traces even with release versions (i.e. compiled without SITRACES)
<improvement>[tag] Consistency in tagging between memory/stdout/file (the tag was duplicated in file)
<improvement>[Compatibility/traces] In L0_StoreTag: returning directly when SiTRACES is not defined.

### 2.15 As from V3.4.7 (2014/06/19)

<porting>[Windows] In system_time: replacing 1000/CLOCKS_PER_SEC by 1/1 on Windows, since this provides a much longer time before the returned value folds and returns a negative value.

## 2.16 As from V3.4.6 (2014/05/16)

<new_feature> [tag/level] Adding support for tag and level
This allows adding a tag at the beginning of trace lines to identify the frontend and DTV/TER/SAT, what is very useful to use traces generated by multiple frontend applications.
In addition to this, it is now possible to display the 'level' as well, as well as select which level is visible in traces (level is from 0 to 32, commonly 0 to 3 in our console code).
Adding SiTraceLevelEnabled (int level): to check whether a given trace level is enabled
Adding SiTraceLevel (int level, unsigned char on_off): to set the 'enable' flag for the trace level
Redefinition of SiTraceFunction to have it take the level and tag as well.
Thanks to new macros, the changes in the other layers are minimal.
Use SiTRACE_X to use the 'same' traces as before (defaulting to level 32, tag "")
<improvement> [SiTRACE config] In SiTraceConfiguration: adding check on number of arguments for configuration strings

## 2.17 As from V3.4.5 (2014/04/16)

<improvement> [snprintf] Adding STRING_APPEND_SAFE to replace snprintf, since this function behaves differently between platforms when called with:
snprintf(msg, 1000, "%s%d", msg, value);
Under Windows: the text for 'value' is APPENDED to msg.
Under Linux: the text for 'value' is REPLACING the original msg.

## 2.18 As from V3.4.4 (2014/04/12)

<new feature> [tag/level] Adding support for tag and level
This allows adding a tag at the beginning of trace lines to identify the frontend and DTV/TER/SAT> This is very useful to use traces generated by multiple frontend applications.
In addition to this, it is now possible to display the 'level' as well, as well as select which level is visible in traces (level is from 0 to 32, commonly 0 to 3 in our console code).
Adding `SiTraceLevelEnabled (int level)`: to check whether a given trace level is enabled
Adding `SiTraceLevel (int level, unsigned char on_off)`: to set the 'enable' flag for the trace level
Re-definition of SiTraceFunction to have it taking the level and tag as well.
Thanks to new macros, the changes in the other layers are minimal.
Use SiTRACE_X to use the 'same' traces as before (defaulting to level 32, tag "")
<improvement> [SiTRACE config] In SiTraceConfiguration: adding check on number of arguments for configuration strings.

## 2.19 As from V3.4.3 (2013/12/19)

<new_feature> Adding SPI control functions
(initially only implemented for the Cypress Fx2lp chip):

- L0_EnableSPI: enabling SPI port selected pins
- L0_LoadSPI: loading bytes over SPI
- L0_DisableSPI: disabling SPI pins

## 2.20 As from V3.4.2

<improvement>In L0_WriteString: removed one dead code line

### 2.21 As from V3.4.1

Moving buffer allocation inside functions using them, to be thread-safe.
NB: There is one remaining area which is not thread-safe in the functions used by the SiERROR and CHECK_FOR_ERRORS macros.
If this is an issue, redefine the SiERROR and CHECK_FOR_ERRORS macros inside SiLabs_L0_API.h to empty strings

### 2.22 As from V3.4.0

Adding copyright notice to source code delivery

### 2.23 As from V3.3.9

In strcmp_nocase_n: using "%s" in sprintf (greater Linux compatibility)

### 2.24 As from V3.3.8

Declaring parameter strings as 'const char*' instead of 'char*' to limit -Wwrite-strings warnings

### 2.25 As from V3.3.7

Adding L0_SlowI2C and L0_FastI2C for easier control of the i2c speed from the above layers

### 2.26 As from V3.3.6

Improved porting comments in L0_ReadBytes

### 2.27 As from V3.3.5

casting calls to strlen as (int) for greater compatibility with VisualStudio

### 2.28 As from V3.3.4

Added help for 'resume' / "suspend'

### 2.29 As from V3.3.3

Adding 'suspend' / 'resume' in traces to keep control of the traces destination at application level.
This is required to avoid having the traces routed to file after each FW download with 'command mode' parts.

### 2.30 As from V3.3.2

Correction on last character when changing traces file name
Correction in traces file name management

# 3 Trace features

## 3.1 SiTRACES, the general use trace mechanism

The general use traces mechanism is equivalent to having access to the printf function only when this is required, while having the ability to remove all the corresponding code from the compilation by commenting a single line in Silabs_L0_API.h.

The traces mechanism provides a number of features to:

- Select the trace destination
- Display source code information
- Display time information
- Manage the traces

### 3.1.1 Compiling without traces

To remove all traces from the source code a single line needs to be commented in Silabs_L0_API.h:

```
/* Uncomment the following line to activate all traces in the code */
/*#define SiTRACES*/
```

This simple operation removes all calls to the following functions from the compilation:

```
SiTRACE
SiTRACE_X
SiTraceConfiguration
TRACES_PRINTF
ALLOCATION_ERROR_MESSAGE
TREAT_ERROR_MESSAGE
TRACES_ERR
TRACES_TRACE
TRACES_SHOW
TRACES_USE
```

### 3.1.2 Compiling with traces

To allow traces from the source code a single line needs to be activated in Silabs_L0_API.h:

```
/* Uncomment the following line to activate all traces in the code */
#define SiTRACES
```

### 3.1.3 Compiling with minimal features

To disable *printf* and *fprintf* functions, the following macro in Silabs_L0_API.h has to be changed:

```
/* Replace 'SiTRACES_FULL' by 'SiTRACES_MINIMAL' in the following line to de-activate full
features mode. WARNING: the minimal features mode disable the display of functions in stdout
and the extern file */
```

```
#define SiTRACES_FEATURES      SiTRACES_FULL
```

### 3.1.4 Selecting the default trace configuration

The default trace configuration can be changed using the trace_config flags in Silabs_L0_Connection.c and the trace_output_type flag in traceDefaultConfiguration.

```
. . .

unsigned char trace_config_lines       = 0;
unsigned char trace_config_files       = 0;
unsigned char trace_config_rename      = 0;
unsigned char trace_config_time        = 0;
unsigned char trace_config_tags        = 1;
unsigned char trace_config_level       = 1;
unsigned char trace_config_verbose     = 0;

. . .

void traceDefaultConfiguration(void) {
. . .
    trace_output_type = TRACE_MEMORY;
. . .
}
```

### 3.1.5 Selecting a trace output

The trace output at startup can be set in traceDefaultConfiguration

```
. . .

void traceDefaultConfiguration(void) {
. . .
    trace_output_type = TRACE_MEMORY;
. . .
}
```

It can also be changed during execution using dedicated SiTraceConfiguration strings:

```
SiTraceConfiguration ("traces -output memory");
SiTraceConfiguration ("traces -output file");
SiTraceConfiguration ("traces -output stdout");
SiTraceConfiguration ("traces -output none");
```

### 3.1.6 Traces options

Using ‘traces’/’help’ in the console application the following help text is displayed, describing all possible trace options:

```
-------------------------------------------------------------------------------------------
 Possible traces commands:
-------------------------------------------------------------------------------------------
traces  -output   <memory/stdout/file/none>. select the trace destination (don't use 'stdout' in wish)
traces  -file     <on/off> . . . . . . . . . select file name   mode (adding C source file name)
traces  -line     <on/off> . . . . . . . . . select line number mode (adding C source file line number)
traces  -function <on/off> . . . . . . . . . select function    mode (adding C source file function name)
```

```
traces  -time     <on/off> . . . . . . . . . select time tag    mode (adding elapsed time as hh:mm:ss.ms)
traces  -tag      <on/off> . . . . . . . . . select tag        mode (adding user-defined tag, often with frontend index)
traces  -level    <on/off> . . . . . . . . . select level      mode (adding trace level indication)
traces  -name     <file_name>. . . . . . . . select traces file name (default 'Skyworks_traces.txt' and '' gives the current file used)
traces  -verbose  <on/off> . . . . . . . . . select verbose mode (traces in console even if '-output' not 'stdout')
traces  show . . . . . . . . . . . . . . . . display buffered traces in console (not visible in wish)
traces  get. . . . . . . . . . . . . . . . . display buffered traces (visible in wish)
traces  count. . . . . . . . . . . . . . . . display number of traces since start
traces  suspend. . . . . . . . . . . . . . . suspend tracing until 'resume'
traces  resume . . . . . . . . . . . . . . . resume  tracing after 'suspend'
traces  lost . . . . . . . . . . . . . . . . display number of traces lost since start
traces  save . . . . . . . . . . . . . . . . save    buffered traces to file
traces  flush. . . . . . . . . . . . . . . . erase   buffered traces
traces  erase. . . . . . . . . . . . . . . . erase   file content
traces  status . . . . . . . . . . . . . . . display the current traces flags
traces  help . . . . . . . . . . . . . . . . display this help
```

#### 3.1.6.1 'traces –output <memory/stdout/file/none>'

The traces can be routed to several direction:

- 'memory' : to be retrieved only if required
- 'stdout' : direct console output
- 'file' : traces are written to a file (useful when the application crashes, since the file will still contain the last trace before the crash).
- 'none' : no tracing

#### 3.1.6.2 'traces –file <on/off>'

Each trace line can contain the name of the source code file it's originating from.
By default, the filename is Skyworks_Traces.txt, but it can be changed using 'traces –name <new_file_name>'

#### 3.1.6.3 'traces –line <on/off>'

Each trace line can contain the number of the line in the source code file it's originating from.

#### 3.1.6.4 'traces –file <on/off>'

Each trace line can contain the name of the source code file it's originating from.

#### 3.1.6.5 'traces –function <on/off>'

Each trace line can contain the name of the function it's originating from.

#### 3.1.6.6 'traces –time <on/off>'

Each trace line can contain the time it was generated, starting from the application's startup time. Units in ms.

#### 3.1.6.7 'traces –tag <on/off>'

Each trace line can contain the tag of the front-end it's originating from.
This is particularly useful for multiple front-ends, to identify which front-end is the source of each trace line. It's of less use for single front-ends, but can still be interesting to sort out Skyworks traces amongst many other.
The configuration macros delivered with Skyworks source code generally use 'FE[<x>]' as the tag for each front-end, but this can be 'main'/'sub', etc. Any text string
This is set using a L3 configuration function as the 'tag' text string (limited to SILABS_TAG_SIZE = 20 characters max by default):

```
SiLabs_API_Set_Index_and_Tag (SILABS_FE_Context* front_end, unsigned char index, char* tag);
```

#### 3.1.6.8 'traces –level <on/off>'

Each trace line can contain the level of the function it's originating from (L0, L1, L2, L3).

This is set using a L3 configuration function as the 'index' value:

```
SiLabs_API_Set_Index_and_Tag (SILABS_FE_Context* front_end, unsigned char index, char* tag);
```

Level tracing allows reducing the amount of traces once the application is working fine, to keep tracing the L3 accesses for instance.

#### 3.1.6.9 'traces –function <on/off>'

Each trace line can contain the name of the function it's originating from.

#### 3.1.6.10 'traces –name <new_file_name>'

The traces are routed to a file when using 'traces –output file'.
By default, the filename is Skyworks_Traces.txt, but it can be changed using 'traces –name <new_file_name>'

#### 3.1.6.11 'traces –verbose <on/off>'

In 'verbose' mode, trace lines are always traced to stdout in addition to all possible destinations ('none', 'memory' or 'file').
'-verbose on' plus '-file on' allows following the application progress in the console while in parallel saving the traces to the file for later analysis.

### 3.1.7 Suspending trace messages

In some cases, to avoid too many traces lines during execution it is preferable to suspend the trace logging without changing its configuration.
This is particularly useful while downloading firmware to a chip.
It is interesting to trace the very first lines to make sure the process is running fine, and then traces can be suspended until the download is complete.

This is possible using the 'suspend' option:

```
SiTraceConfiguration ("traces suspend");
```

To resume tracing, use:

```
SiTraceConfiguration ("traces resume");
```

### 3.1.8 Configuring the trace buffer size

The trace buffer size can only be changed at startup by modifying the following macro value in Silabs_L0_Connection.c. :

```
. . .
#define SiTRACES_BUFFER_LENGTH 100000
. . .
```

## 3.1.9 Traces examples

### 3.1.9.1 -file off –line off –function off–time off –tag off –level off

```
. . .
>> WRITE baud_rate_unfrz_cond baud_rate_unfrz_cond_disable : 0 (0x0000)  @[0x298:1]-[0x299:0]
. . .
<< READ  pll_divm                              :   16 (0x0010)  @[0xc5:4-0]
<< READ  pll_divl                              :   22 (0x0016)  @[0xc4:4-0]
. . .
>> WRITE timing_unfrz_cond    timing_unfrz_cond_all_agc_lock : 2 (0x0002)  @[0x298:3]-[0x299:2]
writing 0xc8  USB> 0x02 0x99
reading 0xc9  <USB 0x08
writing 0xc8  USB> 0x02 0x99 0x08
. . .
```

### 3.1.9.2 -file on –line on –function off –time off –tag off –level off

```
. . .
./Si2167/Si2167_L1_Multistandard_API.c   183 Si2167_L1_Demod_init_after_reset starting...
. . .
>> WRITE dcom_data            patch_table[table_index++] : -805101344 (0xd00320e0)  @[0x48f:7]-[0x490:0]
writing 0xc8  USB> 0x04 0x90 0xe0 0x20 0x03 0xd0
. . .
./Si2167/Si2167_L1_Multistandard_API.c   332 Si2167_L1_Demod_init_after_reset done...
. . .
```

### 3.1.9.3 -file on –line on –function off –time on –tag off –level off

```
. . .
00:09:30.159   >> WRITE clock_mode           mode                       : 33 (0x0021)  @[0xffffffff:6]-
[0x0:0]
00:09:30.159  writing 0xc8  USB> 0x00 0x00 0x21
. . .
  ./Si2167/Si2167_L1_Multistandard_API.c  312 00:09:30.549  Si2167_L1_Demod_init_after_reset calling
Si2167_L1_Demod_Re_Init...
  ./Si2167/Si2167_L1_Multistandard_API.c  348 00:09:30.549  Si2167_L1_Demod_Re_Init starting...
. . .
  ./Si2167/Si2167_L1_Common_Demod_API.c   167 00:09:41.173
DOWNLOAD IS OK, CRC is good and equal to 0xbbca
. . .
```

### 3.1.9.4 -file off –line off –function off –time on –tag off –level off

```
. . .
00:13:26.470  Si2167_L1_Demod_init_after_reset starting...
. . .
00:13:30.011   >> WRITE dcom_data            patch_table[table_index++] : -1206910952 (0xb8100018)
@[0x48f:7]-[0x490:0]
00:13:30.011  writing 0xc8  USB> 0x04 0x90 0x18 0x00 0x10 0xb8
. . .
00:13:37.828  Si2167_L1_Demod_init_after_reset done...
```

### 3.1.9.5 –file off –line off –function on –time off –tag off –level off

```
. . .
..67_L1_Demod_init_after_reset Si2167_L1_Demod_init_after_reset: Board connection test OK
. . .
reading 0xc9  <USB 0x01
..67_L1_Demod_init_after_reset Si2167_L1_Demod_init_after_reset init complete
writing 0xc8  USB> 0x00 0x5b 0x00
. . .
switch_to_standard             DVB-T->DVB-T {<S1T1i>D1i}| ( 7332 ms)
switch_to_standard             | TER: 2    0 ms | SAT: 1    0 ms | DTV: 4 7332 ms | ( 7332 ms)
. . .
```

#### 3.1.9.6 –file on –line on –function on –time off –tag off –level off

```
. . .
writing 0xc8 at 0x04 0x88 USB> 0x3f 0x00 0x00 0xc0 0x10 0x09 0x00 0x60 0x01 0x00 0x18 0x15
. . .
./Si2167/Si2167_L1_Multistandard_API.c     119 check_setup                ll_divk = 24
./Si2167/Si2167_L1_Multistandard_API.c     121 check_setup                adc_clk       =   56000000 Hz
./Si2167/Si2167_L1_Multistandard_API.c     122 check_setup                Tuner_IF_HZ   =    5000000 Hz
. . .
./Si2167/Si2167_L3_Test.c                 1027 switch_to_standard            | TER: 2    0 ms | SAT: 1
0 ms | DTV: 4 7317 ms | ( 7317 ms)
. . .
./Si2167/Si2167_L3_Test.c                 1111 relock_to_standard            relock_to_standard DVB-T
sequence: [Tb8000000p1r]
. . .
```

#### 3.1.9.7 –file off –line off –function off –time off –tag on –level on

```
. . .
fe[0] DTV L1    Si2183 DD_BER RST 0
fe[0] DTV L1    Si2183 DD_PER RST 0
fe[0]     L3    Checking FE_SPECIFIC
fe[0] DTV L1    Si2183 DVBT_STATUS INTACK 0
fe[0] DTV L1    Si2183 DVBT_TPS_EXTRA
fe[0]     L3    Checking FE_QUALITY
fe[0] DTV L1    Si2183 DD_SSI_SQI TUNER_RSSI -59
. . .
```

#### 3.1.9.8 –file off –line off –function off –time off –tag on –level off

```
. . .
fe[0] DTV Si2183 DD_BER RST 0
fe[0] DTV Si2183 DD_PER RST 0
fe[0]     Checking FE_SPECIFIC
fe[0] DTV Si2183 DVBT_STATUS INTACK 0
fe[0] DTV Si2183 DVBT_TPS_EXTRA
fe[0]     Checking FE_QUALITY
fe[0] DTV Si2183 DD_SSI_SQI TUNER_RSSI -59
. . .
```

#### 3.1.9.9 –file off –line off –function off –time off –tag off –level on

```
. . .
L1    Si2183 DD_BER RST 0
L1    Si2183 DD_PER RST 0
L3    Checking FE_SPECIFIC
L1    Si2183 DVBT_STATUS INTACK 0
L1    Si2183 DVBT_TPS_EXTRA
L3    Checking FE_QUALITY
L1    Si2183 DD_SSI_SQI TUNER_RSSI -59
. . .
```

### 3.1.10 Traces impact on the final application

### 3.1.11 Execution time

The tests are made with L0 traces activated. The results give a rough estimate of the execution time and traces impact on the application, during a typical fw download in a component with 'register level' traces activated:

| | #undef SiTRACES | #define SiTRACES | | | |
|---|---|---|---|---|---|
| -output | X | none | memory | file | stout |
| Average execution time (in ms) | 2045 | 2068 | 2073 | 11616 | 22240 |

NB: These results have been obtained with the executable being on an external network disk. Figures are much better when running the application on a local disk, in fact quite close to the '-output memory' case.

Activating traces options (e.g. –line on, –file on) obviously increases the execution time. Nevertheless, the difference is minimal.

### 3.1.12 Application size

| | | |
|---|---|---|
| Size of Si2167_L3_Test.exe: | #undef SiTRACES | 212 Ko |
| | #define SiTRACES | 231 Ko |

## 3.2 SiERROR, the general use ERROR logging mechanism

The error logging mechanism has been implemented in addition to the normal traces to store the last error messages and make them available to the application upon request.

The error logging mechanism is much simpler than the normal traces, as it is not configurable and allows only simple strings (no 'printf-type' formatting is possible).

This last restriction makes it compatible with old version of some compilers, not supporting variadic-macros, i.e. macros with variable number of arguments.

The feature is useful to track initialization errors, which can be tricky to detect during the development phase.
The source code should contain enough dedicated error messages (i.e. SiERROR lines) to find out what the error is and make it easy to debug.

### 3.2.1 SiERROR features

Storing error messages is done calling the L0_StoreError function, which is made available via the SiERROR macro:

```
#define SiERROR           L0_StoreError
```

Using SiERROR makes it easy to remove the SiERROR messages from the application, just by commenting the macro definition in SiLabs_L0_API.h.

Retrieving error messages is done calling the L0_ErrorMessage function and displaying the returned string (an empty string if there no error logged).

Be careful using this, as the errors will be cleared when calling L0_ErrorMessage, so they need to be processed immediately as it will not be possible to retrieve them a second time.

## 3.3 Low-level (I2C bytes) traces

To allow easy control of the i2c bytes sent over the i2c bus, a set of 2 flags are present in the L0_Context:

- `int trackWrite;` a flag to control whether i2c traces are generated or not during write operations.
- `int trackRead;` a flag to control whether i2c traces are generated or not during read operations.

Setting these flags to '1' allows traces to display the i2c bytes sent/received for a particular chip.
It is therefore possible to track low-level i2c per chip, without the need for an i2c spy or sniffer tool.

## 3.4 Register-level traces

A set of preprocessing macros is available for tracing the software behavior during software development. They can be easily disabled in a final release.

```
#define RWTRACES
```

When RWTRACES is defined a C file, the usual L1_READ and L1_WRITE macros are replaced by more 'verbose' versions that will call L1_ReadRegisterTrace and L1_WriteRegisterTrace instead of L1_ReadRegister and L1_WriteRegister.

L1_ReadRegisterTrace and L1_WriteRegisterTrace trace the register name and address as well as the value read from or written into the register.

For example, defining RWTRACES at the beginning of Si2165_L1_Multistandard_API.c (before including Silabs_L0_API.h) enables the 'register' traces generated from within this file.

# 4 Code description

## 4.1 Layer 0 Context

The L0_Context structure contains all the variables used by the Layer 0 code to manage i2c communication. It is defined in Silabs_L0_API.h.

```
typedef struct L0_Context
{
    unsigned char   address;
    int             indexSize;
    CONNECTION_TYPE connectionType;
    int             trackWrite;
    int             trackRead;
    int             mustReadWithoutStop;
    unsigned char   tag_index;
    char            tag[SILABS_TAG_SIZE];
} L0_Context;
```

Members of this structure are:

- unsigned char address; the i2c address of the chip (0xc8 for the Si2167 base address)
- int indexSize; the number of bytes used to index the i2c bytes. 2 for the Si2167, usually 1 for most tuners.
- CONNECTION_TYPE connectionType; the value of the current connection mode. Set to 'USB' for Skyworks EVBs, it can also be 'SIMU' for software debug purposes or 'CUSTOMER' for customer hardware.
- int trackWrite; a flag to control whether i2c traces are generated or not during write operations.
- int trackRead; a flag to control whether i2c traces are generated or not during read operations.
- int mustReadWithoutStop; a flag indicating whether a read operation can include a STOP condition (the normal i2c behavior) or not. (So far, only the RDA5812 satellite tuner does not allow the normal mode and this flag must be set as 1).
- tag_index: a byte used to print the 'level' of the trace
- tag: the tag string (which can be set by the application as required, with a limit of SILABS_TAG_SIZE (default size is 20)

## 4.2 Layer 0 functions

The layer 0 code is delivered with simulator capabilities and can be adapted to the customer communication layer.

This part is probably where the biggest effort is required in the initial phase, to get proper communication using a dedicated method.

In case of difficulties getting communication with the HW, please contact Skyworks Solutions to get help from the software team.

Initially, it is possible to select between 3 communication modes during execution:

- 'SIMU' mode, to work without any hardware.
- 'USB' mode, to communicate with the Skyworks Solutions evaluation board.
- 'CUSTOMER' mode. This mode needs to be adapted to allow communication using the customer communication layer.
-

The layer 0 communication layer being 'generic' to all Skyworks Solutions Broadcast Video products, it is not related to any particular IC and can be used for any i2c component, whether using 0, 1 or 2 bytes indexing.

The layer 0 API functions are described below.

### 4.2.1 L0_Connect

Description: L0_Connect is the Core layer 0 connection function.
It is used to switch between various connection modes
It can be useful during SW development, to easily switch between the simulator and the actual HW.

Prototype: L0_Connect *i2c, CONNECTION_TYPE connType);

Porting: Depending on the i2c layer, this function could be removed if a single connection mode is allowed

Returns: 1 if connected to actual hw.

### 4.2.2 L0_Cypress_Configure

Description: L0_Cypress_Configure is the Cypress chip configuration function.
It is used to send a configuration command to the Cypress chip.
Comment: The cypress ship commands are defined in the underlying Cypress USB dll, and are not relevant to non-Windows cases or when the i2c layer is not using the Cypress chip (after porting).

Prototype: L0_Cypress_Configure *cmd, char *text, double dval, double *retdval, char **rettxt);

Porting: If not using the Cypress Fx2LP USB interface, comment the USB_Capability flag to avoid compiling this function

Returns: 1 if the command is unknow.

### 4.2.3 L0_Cypress_Process

Description: L0_Cypress_Process is the Cypress chip command channel function.
It is used to send a generic command to the Cypress chip.
Comment: The cypress ship commands are defined in the underlying Cypress USB dll, and are not relevant to non-Windows cases or when the i2c layer is not using the Cypress chip (after porting).

Prototype: L0_Cypress_Process *cmd, char *text, double dval, double *retdval, char **rettxt);

Porting: If not using the Cypress Fx2LP USB interface, comment the USB_Capability flag to avoid compiling this function

Returns: 1 if the command is unknow.

### 4.2.4 L0_Init

Description: L0_Init is the layer 0 initialization function.
It is used to set the layer 0 context parameters to startup values.
It must be called first and once per Layer 1 instance (i.e. once for the tuner and once for the demodulator).
It is automatically called by the Layer 1 init function.
Parameters: mustReadWithoutStop has been added to manage the case when some components do not allow a stop in a 'read'.
i2c usually allows 'write 0xc8 0x01 0x02' followed by 'read 0xc9 1' to read the byte at index 0x0102.
This should return the same data as 'read 0xc8 0x01 0x02 1'.
If this is not allowed, set mustReadWithoutStop at 1.
NB: at the date of writing, this behavior has only been detected in the RDA5812 satellite tuner.

Prototype: void L0_Init (L0_Context **ppContext) ;

Porting: If some members of the L0_Context structure are removed, they need to be removed from here too

Returns: Void.

### 4.2.5 L0_ReadBytes

Description: L0_ReadBytes is the lowest layer read function.
It is used to read a given number of bytes from the Layer 1 instance.
Parameters: i2c, a pointer to the Layer 0 context.
iI2CIndex, the index of the first byte to read.
iNbBytes, the number of bytes to read.
*pbtDataBuffer, a pointer to a buffer used to store the bytes.

Prototype: L0_ReadBytes i2c, unsigned int iI2CIndex, int iNbBytes, unsigned char *pucDataBuffer);

Porting: If a single connection mode is allowed, the entire switch can be replaced by a call to the final i2c read function

Returns: The number of bytes read.

### 4.2.6 L0_ReadCommandBytes

Description: L0_ReadCommandBytes is the 'command mode' bytes reading function.
It is used to read a given number of bytes from the Layer 1 instance in 'command mode'.
Comment: The 'command mode' is a specific mode where the indexSize is always 0 and the index is always 0x00
Parameters: i2c, a pointer to the Layer 0 context.
iNbBytes, the number of bytes to read.
*pucDataBuffer, a pointer to a buffer used to store the bytes.

Prototype: L0_ReadCommandBytes i2c, int iNbBytes, unsigned char *pucDataBuffer);

Returns: The number of bytes read.

### 4.2.7 L0_ReadRawBytes

Description: L0_ReadRawBytes is the raw i2c read function.
It is used to read a given number of bytes from the Layer 1 instance while managing the indexSize at this level.
Parameters: i2c, a pointer to the Layer 0 context.
iI2CIndex, the index of the first byte to read.
iNbBytes, the number of bytes to read.
*pbtDataBuffer, a pointer to a buffer used to store the bytes.
Comment: This function splits a 'read' operation in
1- 'writing' the chip's i2c adress plus the index bytes
2- 'reading' iNbBytes bytes from the chip.

Prototype: L0_ReadRawBytes i2c, unsigned int iI2CIndex, int iNbBytes, BYTE *pbtDataBuffer);

Porting: Can be used to manage the 16 bit case at this level

Returns: The number of bytes read.

### 4.2.8 L0_ReadRegister

Description: L0_ReadRegister is the register read function.
It is used to read a register based on its address, size, offset.

Prototype: L0_ReadRegister *i2c, unsigned int add, unsigned char offset, unsigned char nbbit, unsigned int isSigned);

Behavior: This function uses the characteristics of the register to read its value from hardware
it reads the minimum number of bytes required to retrieve the register based on the number of bits and the offset
it masks all bits not belonging to the register before returning
it handles the sign-bit propagation for signed registers

Parameters:
Add register address.
Offset register offset.
Nbbit register size in bits.
IsSigned register sign info.

Returns: The value read. O if error during the read.

### 4.2.9 L0_ReadRegisterTrace

Description: L0_ReadRegisterTrace is the trace and read function.
It is used to read a register based on its address, size, offset, with traces.

Prototype: L0_ReadRegisterTrace *i2c, char* name, unsigned int add, unsigned char offset, unsigned char nbbit, unsigned int isSigned);

Behavior: This function traces the register information, then calls L1_ReadRegister with all the register parameters
It is used only when tracing register reads is required

Comments: generally activated by changing the definition of the CONTEXT_READ function in the proper header file

Parameters:

Add register address.
Offset register offset.
Nbbit register size in bits.

Returns: The value written. O if error during the write.

### 4.2.10 L0_SetAddress

Description: L0_SetAddress is the function to set the device address.
It is used to set the I2C address of the component.
It must be called only once at startup per Layer 1 instance, as the addresses are not expected to change over time.

Prototype: L0_SetAddress i2c, unsigned int add, int addSize);

Returns: 1 if OK, 0 otherwise.

### 4.2.11 L0_TrackRead

Description: L0_TrackRead is the layer 0 initialization function.
It is used to toggle the read traces for the related Layer 1 instance.
It is useful for debug purpose, mostly to control that data is properly transmitted to the above layers.

Prototype: L0_TrackRead i2c, unsigned int track);

Returns: Void.

### 4.2.12 L0_TrackWrite

Description: L0_TrackWrite is the layer 0 initialization function.
It is used to toggle the write traces for the related Layer 1 instance.
It is useful for debug purpose, mostly to control that data is properly written to the desired Layer 1 instance.

Prototype: L0_TrackWrite i2c, unsigned int track);

Returns: Void.

### 4.2.13 L0_WriteBytes

Description: L0_WriteBytes is the lowest layer write function.
It is used to write a given number of bytes from the Layer 1 instance.
Parameters: i2c, a pointer to the Layer 0 context.

iI2CIndex, the index of the first byte to write.
iNbBytes, the number of bytes to write.
*pbtDataBuffer, a pointer to a buffer containing the bytes to write.

Prototype: L0_WriteBytes i2c, unsigned int iI2CIndex, int iNbBytes, unsigned char *pucDataBuffer);

Porting: If a single connection mode is allowed, the entire switch can be replaced by a call to the final i2c write function

Returns: The number of bytes read.

### 4.2.14 L0_WriteCommandBytes

Description: L0_WriteCommandBytes is the 'command mode' bytes writing function.
It is used to write a given number of bytes to the Layer 1 instance in 'command mode'.
Comment: The 'command mode' is a specific mode where the indexSize is always 0 and the index is always 0x00
Parameters: i2c, a pointer to the Layer 0 context.
iNbBytes, the number of bytes to write.
*pucDataBuffer, a pointer to a buffer containing the bytes.

Prototype: L0_WriteCommandBytes i2c, int iNbBytes, unsigned char *pucDataBuffer);

Returns: The number of bytes written.

### 4.2.15 L0_WriteRawBytes

Description: L0_WriteRawBytes is the raw i2c write function.
It is used to write a given number of bytes from the Layer 1 instance while managing the indexSize at this level.
Parameters: i2c, a pointer to the Layer 0 context.
iI2CIndex, the index of the first byte to write.
iNbBytes, the number of bytes to write.
*pbtDataBuffer, a pointer to a buffer containing the bytes.

Prototype: L0_WriteRawBytes i2c, unsigned int iI2CIndex, int iNbBytes, BYTE *pbtDataBuffer);

Porting: Can be used to manage the 16 bit case at this level

Returns: The number of bytes written.

### 4.2.16 L0_WriteRegister

Description: L0_WriteRegister is the register write function.
It is used to write a register based on its address, size, offset.

Prototype: L0_WriteRegister *i2c, unsigned int add, unsigned char offset, unsigned char nbbit, unsigned char alone, long Value);

Behavior: This function uses all characteristics of the register to write its value in the hardware
if the required value is out of range, no operation is performed and an error code is returned
before writing, the current register value is retrieved (if required because of adjacent registers),
in order to preserve the contain of adjacent registers
it reads the minimum number of bytes required to retrieve the register based on:
the number of bits and the offset
it keeps all bits not belonging to the register intact
it handles the sign-bit propagation for signed registers

Parameters:
Add register address.
Offset register offset.
Nbbit register size in bits.
Alone register loneliness info (1 if all other bits can be overwritten without checking).
Value the required value for the register.

Returns: The value written. O if error during the write.

### 4.2.17 L0_WriteRegisterTrace

Description: L0_WriteRegisterTrace is the register write and trace function.
It is used to write a register based on its address, size, offset, with traces.

Prototype: L0_WriteRegisterTrace *i2c, char* name, char* valtxt, unsigned int add, unsigned char offset, unsigned char nbbit, unsigned char alone, long Value);

Behavior: This function traces the register information, then calls L1_WriteRegister with all the register parameters
It is used only when tracing register writes is required

Comments: generally activated by changing the definition of the CONTEXT_WRITE function in the proper header file

Parameters:

Add register address.
Offset register offset.
Nbbit register size in bits.
Alone register loneliness info (1 if all other bits can be overwritten without checking).
Value the required value for the register.

Returns: The value written. 0 if error during the write.

### 4.2.18 SiTraceConfiguration

Description: SiTraceConfiguration is the SiTRACES configuration function.
It is used to configure the traces or trace a custom string.

Prototype: char *SiTraceConfiguration (char *config) ;

Comments: If the trace entered starts by 'traces', analyze the configuration string trace and define which type of arguments there are (each argument refers to a particular treatment). The trace message may include several series of (-<param> <value>) pairs.

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameter: config,the configuration string or trace to add (if not a configuration string).

Returns: The help if 'traces help' or 'traces'
The configuration status if 'traces -<param> <value>'
The traces buffer content if 'traces get'
The number of traces if 'traces count'
The number of lost traces if 'traces lost'
Nothing otherwise.

### 4.2.19 SiTraceFunction

Description: SiTraceFunction is the SiTRACES trace formatting function.
It formats the trace message with file name and line number and time if selected then saves it to the trace output.

Prototype: SiTraceFunction(char int number, char *fmt, .);

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameters: name the file name where the trace is written.
number the line number where the trace is written.
fmt string content of trace message. Others arguments are sent thanks to the ellipse.

Returns: void.

### 4.2.20 SiTraceLevelEnabled function

Description: SiTraceLevelEnabled is the SiTRACES level trace activation/de-activation check function.
It is used to know if tracing is enabled/disabled for the selected level.
Prototype: int SiTraceLevelEnabled (int level);
Parameters: level, the selected level
Returns: the flag for the selected level
Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

### 4.2.21 SiTraceLevel function

Description: SiTraceLevel is the SiTRACES level trace activation/de-activation function.
It is used to enable/disable traces for the selected level.
Prototype: int SiTraceLevel (int level, unsigned char on_off);
Parameters: level, the selected level to activate/de-activate
on_off: a flag indicating if the selected level traces are visible (1) or hidden (0)

Returns: the bitfield representing the currently active levels
Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

### 4.2.22 L0_EnableSPI function

Description: L0_EnableSPI is the SPI enable function.
It is used to allow sending a series of bytes over SPI

Porting: Needs to be completed to match the SPI HW. Initially supporting only the Cypress chip.

Parameters: SPI_config. One byte used to select with the cypress chip the initial port A (OEA) enable configuration.

Returns: 1 if OK, 0 if SPI is not available

### 4.2.23 L0_LoadSPI function

Description: L0_LoasSPI is the SPI download function
It is used to send a series of bytes over SPI

Prototype: int L0_LoadSPI (unsigned char *SPI_data, unsigned char length, unsigned short index)

Porting: Needs to be completed to match the SPI HW. Initially supporting only the Cypress chip.

Parameters: SPI_data, the buffer containing the SPI bytes to send.
length, the number of bytes to send
index, a byte used with the Cypress Chip to select the pins used for spi_clk and spi_data in the Cypress:

- 0x01 : SPI Data on PA_0, SPI clock on PA_1
- 0x10 : SPI Data on PA_1, SPI clock on PA_0
- 0x02 : SPI Data on PA_0, SPI clock on PA_2
- 0x20 : SPI Data on PA_2, SPI clock on PA_0
- 0x12 : SPI Data on PA_1, SPI clock on PA_2
- 0x21 : SPI Data on PA_2, SPI clock on PA_1

Returns: 0 if OK, otherwise an error code

### 4.2.24 L0_DisableSPI function

Description: L0_DisableSPI is the SPI disable function
It is used to disable sending bytes over SPI

Prototype: int L0_DisableSPI (void) ;

Porting: Needs to be completed to match the SPI HW. Initially supporting only the Cypress chip.

Returns: 1 if OK, 0 if SPI is not available

### 4.2.25 system_time

Description: system_time is the current system time retrieval function.
It is used to retrieve the current system time in milliseconds.

Prototype: int system_time(void) ;

Porting: Needs to use the final system call

Returns: The current system time in milliseconds.

### 4.2.26 system_wait

Description: system_wait is the current system wait function.
It is used to wait for time_ms milliseconds while doing nothing.

Prototype: int system_wait(int time_ms) ;

Porting: Needs to use the final system call for time retrieval

Parameter: time_ms the wait duration in milliseconds.

Returns: The current system time in milliseconds.

### 4.2.27 traceDefaultConfiguration

Description: traceDefaultConfiguration is the SiTRACES initialization function.
It is called on the first call to L0_Init (only once).
It defines the default output and inserts date and time in the default file.

Prototype: void traceDefaultConfiguration(void) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Returns: void.

### 4.2.28 traceElapsedTime

Description: traceElapsedTime is the SiTRACES time formatting function.
It allows the user to know when the trace has been treated.
It is used to insert the time before the trace when -time 'on'.

Prototype: char *traceElapsedTime(void) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Returns: Text containing the execution time in HH:MM:SS. ms format.

### 4.2.29 traceFlushBuffer

Description: traceFlushBuffer is the SiTRACES buffer erasing function.
It is called to empty the buffer.

Prototype: void traceFlushBuffer(char *buffer) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameter: buffer the trace buffer content.

Returns: void.

### 4.2.30 traceHelp

Description: traceHelp is the SiTRACES configuration help function.

Prototype: char* traceHelp (void);

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Returns: The help menu text.

### 4.2.31 traceToBuffer

Description: traceToBuffer is the SiTRACES buffer saving function.
It adds file name, line number, function name and time if selected.

Comment: The trace buffer length is limited (it is set by SiTRACES_BUFFER_LENGTH).
If the buffer is full, the oldest traces will be lost to write the new trace.

Prototype: void traceToBuffer(char* trace) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameter:

Returns: Void.

### 4.2.32 traceToDestination

Description: traceToDestination is the switch the trace in the selected output mode.

Comment: In verbose mode, the trace is always displayed in stdout.

Prototype: void traceToDestination(char* trace) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameter: trace, the trace string.

Returns: void.

### 4.2.33 traceToFile

Description: traceToFile is the SiTRACES file saving function.
It writes the current trace in an extern file.
It adds file name, line number, function name and time if selected.

Prototype: void traceToFile(char* trace) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameter:

Returns: void.

### 4.2.34 traceToStdout

Description: traceToStdout is the SiTRACES stdout display function.
It displays the current trace in the command window.
It adds file name, line number, function name and time if selected.

Prototype: void traceToStdout(char* trace) ;

Porting: Not compiled if SiTRACES is not defined in Silabs_L0_API.h.

Parameter:

Returns: void.

## 5 I2c porting

The connection to the new hardware is simplified by the Layer 0 API, which allows connection via several modes using the same application.

Only 2 functions need to be adapted to allow communication via the targeted communication method (inside Silabs_L0_Connection.c):

- `L0_WriteBytes`
- `L0_ReadBytes`

This is such that once the L0_WriteBytes and L0_ReadBytes functions are adapted to the customer hardware all the existing tools can be used directly, and the application should work identically on the Evaluation Board and on the Customer Hardware.

### 5.1 Getting help

In case of difficulties getting communication with the HW, please contact Skyworks Solutions to get help from the software team.

### 5.2 Compiling for non-Windows platforms

To compile the code on a non-windows platform (Linux, for instance), the ‘NO_WIN32’ flag must be defined in Silabs_L0_API.h (line 5 is NOT commented).

*NB: If compiling with the ‘NO_WIN32’ flag, access to the Cypress Fx2LP USB interface is not possible.*

### 5.3 Linux USB: Using the FX2LP driver on Linux

When using a Skyworks EVB on Linux, being able to connect the EVB using the USB connector is required.
To allow this, the FX2LP FW must be loaded in the Cypress part when the EVB is plugged on the USB bus.
This is achieved thanks to a set of udev rules.

udev is the typical Linux system for dynamic peripheral management.
udev associates peripherals with a given VID/PID (Vendor Id/ Product Id) to a set of rules, then these rules can execute scripts to perform any required action.

These rules are stored under `/etc/udev/rules.d`, while scripts are stored under `/user/local/bin`.

### 5.4 Linux USB: using fxload to load the Cypress FW

The program used to load the FW is fxload.
It can be installed using:
```
sudo apt-get install fxload
```

### 5.5 Linux USB: using libusb to communicate with the EVB

The program used to communicate with the EVB is libsub.
It can be installed using:
```
sudo apt-get install libusb-1.0-0-dev
```

### 5.6 Linux USB: Accessing the Cypress FX2LP

Once properly installed, the Cypress part will be accessible under `/dev/bus/$deviceBus/$deviceNumber`
`$deviceBus` and `$deviceNumber` will be set during execution of loadCypressFirmware.sh.

#### 5.6.1 Linux USB: loadCypressFirmware.sh

This first script allows loading the FX2LP FW on the part:
It's executed first for VID=0x10c4/PID=0x8496, then the part is reset to run the FW which has been downloaded. Upon this reset, the new PID is 0x8497.

```
#!/bin/bash
#This script finds the bus and the number of the device.
#Then the firmware is loaded on the device

# 10c4 is the device vendor id, 8496 is the device product id when the firmware is not loaded
deviceInfo=$(lsusb -d 10c4:8496)
deviceBus=$(expr "$deviceInfo" : '^Bus \([0-9]*\)')
deviceNumber=$(expr "$deviceInfo" : '^Bus [0-9]* Device \([0-9]*\)')
```

```
fxload -D /dev/bus/usb/$deviceBus/$deviceNumber -I /usr/local/bin/keil_asi2usb_project.hex  -t fx2

exit 0
```

### 5.6.2 Linux USB: changeAccessPermissionCypress.sh

This second script is executed once the FW has been downloaded and is running.
At this stage, the cypress chip enumerates as VID=0x10c4/PID=0x8497.
This rule is used to change the access permission on the cypress part, to allow all users to control the EVB.

```
#!/bin/bash
#This script finds the bus and the number of the device.
#Then access permission is changed for this device.

# 10c4 is the device vendor id, 8497 is the device product id when the firmware is loaded
deviceInfo=$(lsusb -d 10c4:8497)
deviceBus=$(expr "$deviceInfo" : '^Bus \([0-9]*\)')
deviceNumber=$(expr "$deviceInfo" : '^Bus [0-9]* Device \([0-9]*\)')

ret =$(chmod a+w /dev/bus/usb/$deviceBus/$deviceNumber)

exit 0
```

### 5.6.3 Linux USB: 10-EVB_Silabs-rule.rules

This is the udev rule which will trigger the above scripts depending on the cypress part's status.

When connecting the EVB, VID=0x10c4/PID=0x8496. loadCypressFirmware.sh is triggered.
Then once the FW is downloaded it's executed and VID=0x10c4/PID=0x8497.
Since VID=0x10c4/PID=0x8497, changeAccessPermissionCypress.sh is executed, to allow all users to access the EVB over the USB bus.

```
# 10c4 is the device vendor id, 8496 is the device product id when the firmware is not loaded
ACTION=="add", SUBSYSTEM=="usb", ATTR{idVendor}=="10c4", ATTR{idProduct}=="8496",
RUN+="/usr/local/bin/loadCypressFirmware.sh"

# 10c4 is the device vendor id, 8497 is the device product id when the firmware is loaded
ACTION=="add", SUBSYSTEM=="usb", ATTR{idVendor}=="10c4", ATTR{idProduct}=="8497", SYMLINK+="silabs_evb",
RUN+="/usr/local/bin/changeAccessPermissionCypress.sh"
```

### 5.6.4 Linux USB: Automated Cypress installation : install.sh

To make the process described above easy, the install.sh script should be executed.

```
#!/bin/bash
#This script copies the rules file in /etc/udev/rules.d
#It copies the two scripts and the firmware in /usr/local/bin
#It changes the owner of the script and rule files to root
#Finally it reloads the udev rules


# Make sure only root can run our script
if [[ $EUID -ne 0 ]]; then
   echo "This script must be run as root" 1>&2
   exit 1
fi
if ! type fxload > /dev/null; then
  echo "fxload is not installed"
  echo "use \"apt-get install\" to install it"
  exit 1
fi

cp 10-EVB_Silabs-rule.rules               /etc/udev/rules.d/
echo "cp 10-EVB_Silabs-rule.rules         /etc/udev/rules.d/"
```

```
cp loadCypressFirmware.sh                /usr/local/bin/
echo "cp loadCypressFirmware.sh          /usr/local/bin/"
cp changeAccessPermissionCypress.sh      /usr/local/bin/
echo "cp changeAccessPermissionCypress.sh  /usr/local/bin/"
cp keil_asi2usb_project.hex              /usr/local/bin/
echo "cp keil_asi2usb_project.hex        /usr/local/bin/"

chown root /etc/udev/rules.d/10-EVB_Silabs-rule.rules
echo "chown root /etc/udev/rules.d/10-EVB_Silabs-rule.rules"
chown root /usr/local/bin/loadCypressFirmware.sh
echo "chown root /usr/local/bin/loadCypressFirmware.sh"
chown root /usr/local/bin/changeAccessPermissionCypress.sh
echo "chown root /usr/local/bin/changeAccessPermissionCypress.sh"
udevadm control --reload-rules
```

### 5.6.5 Linux USB: Checking proper USB FW download

Using the following command the user can check the current PID, which should be 0x8497 if both scripts have been properly executed:

```
lsusb -d 10c4
```

## 5.7 Using USB communication with the cypress Fx2LP on Windows

To enable USB communication using the cypress Fx2LP chip used on Skyworks EVBs the following steps are required:

1. The code must be run on a Windows PC
2. The code must be compiled as a Windows application, i.e. the 'NO_WIN32' flag must **NOT** be defined in Silabs_L0_API.h (line 5 is commented).
3. The 'USB_Capability' flag must be defined in Silabs_L0_API.h (line 20 is not commented)
4. The application must be linked with the CypressUSB.lib library.

## 5.8 Removing USB communication capability with the cypress Fx2LP

*NB: We recommend keeping the USB communication capability in the code as much as possible during the development phase (i.e. as long as the code is run on a Windows PC), to allow using the Skyworks GUI and tools on the development platform for comparison purposes.*

To disable USB communication using the cypress Fx2LP chip used on Skyworks EVBs, 2 cases are possible:

1. The code is not compiled as a Windows application, i.e. the 'NO_WIN32' flag is defined in Silabs_L0_API.h (line 5 is NOT commented). This will be the case for Linux, for instance.
2. The 'USB_Capability' flag is not defined in Silabs_L0_API.h (line 20 is commented). This could be the case for a Windows VisualStudio project not requiring access to the Cypress USB interface (Fx2LP).

Both the above options will result in the USB_Capability flag not being defined, and the corresponding code will not be included in the compilation.

## 5.9 CUSTOMER i2c communication

### 5.9.1 L0_WriteBytes porting

To allow writing i2c bytes using custom functions, adapt the 'CUSTOMER' case in the L0_WriteBytes function.

```
int    L0_WriteBytes       (L0_Context* i2c, unsigned int iI2CIndex, int iNbBytes, unsigned char *pucDataBuffer) {
. . .
  switch (i2c->connectionType) {
. . .
    case CUSTOMER:
      nbWrittenBytes = 0;
       /* <porting> Insert here whatever is needed to
       (option 1)
       write iNbBytes bytes
       to the chip whose i2c address is i2c->address,
       starting at index iI2CIndex,
       with an index on i2c->indexSize bytes
       the data bytes being stored in pucDataBuffer.

       (option 2)
       Another option is to
       write iNbBytes + i2c->indexSize bytes
       to the chip whose i2c address is i2c->address,
       the index bytes and data bytes all being stored in pucBuffer.

       Make it such that on success nbWrittenBytes = iNbBytes + i2c->indexSize
       and on failure write_error is incremented.
       */
       break;
. . .
  }
. . .
}
```

When reaching this part of the code, the available variables are:

- `i2c->address` the chip's i2c address (on the 7 MSBs).
- `i2c->indexSize` the number of bytes used for the index (usually 2 for demodulators),
- `iI2CIndex` the index of the first byte to read,
- `iNbBytes` the number of bytes to read from the chip,
- `pucAddressBuffer` a pointer to a bytes buffer containing the index bytes
- `pucDataBuffer` a pointer to a iNbBytes bytes buffer containing the bytes to write
- `pucBuffer` a pointer to a i2c->indexSize+iNbBytes bytes buffer containing the index bytes plus the bytes to write

*Example when writing the standard register for a Si2167 at the base address to 'dvb_t':*

- *`i2c->address` = 0xC8*
- *`i2c->indexSize` = 2*
- *`iI2CIndex` = 264 (= 0x0108)*

- `iNbBytes` = *1*
- `pucAddressBuffer` *a pointer to a bytes buffer containing '0x01 0x08'*
- `pucDataBuffer` *a pointer to a iNbBytes bytes buffer containing '0x01' (the value for 'standard_dvb_t' for a Si2167)*
- `pucBuffer` *a pointer to a i2c->indexSize+iNbBytes bytes buffer containing '0x01 0x08 0x01' (the index bytes plus the value for 'standard_dvb_t' for a Si2167)*

Depending on what the customer i2c write function prototype is, the above variables can be used to have the iNbBytes written at i2c->address, starting at index iI2CIndex.

Upon success, the return value should be the number of data bytes (i.e. 1 in the example), and 0 in case of a write error, to enable error management from the above layer.

### 5.9.2 L0_ReadBytes porting

To allow reading i2c bytes, add a case in the L0_ReadBytes function

```
int    L0_ReadBytes       (L0_Context* i2c, unsigned int iI2CIndex, int iNbBytes, unsigned char *pucDataBuffer) {
. . .
  switch (i2c->connectionType) {
. . .
    case CUSTOMER:
      /* <porting> Insert here whatever is needed to
      read iNbBytes bytes
      from the chip whose i2c address is i2c->address,
      starting at index iI2CIndex,
      with an index on i2c->indexSize bytes
      the data bytes being stored in pucDataBuffer.

      Make it such that on success nbReadBytes = iNbBytes
      and on failure nbReadBytes = 0.
      */
      break;
. . .
  }
. . .
}
```

When reaching this part of the code, the available variables are:

- `i2c->address` the chip's i2c address (on the 7 MSBs).
- `i2c->indexSize` the number of bytes used for the index (usually 2 for demodulators),
- `iI2CIndex` the index of the first byte to read,
- `iNbBytes` the number of bytes to read from the chip,
- `pucAddressBuffer` a pointer to a bytes buffer containing the index bytes
- `pucDataBuffer` a pointer to a iNbBytes bytes buffer ready to receive the bytes read from the chip

*Example when reading the standard register for a Si2167 at the base address:*

- `i2c->address` = *0xC8*

- *i2c->indexSize* = *2*
- *iI2CIndex* = *264 (= 0x0108)*
- *iNbBytes* = *1*
- *pucAddressBuffer* *a pointer to a bytes buffer containing '0x01 0x08'*
- *pucDataBuffer* *a pointer to a iNbBytes bytes buffer ready to receive the bytes read from the chip*

Depending on what the customer i2c read function prototype is, the above variables can be used to have the iNbBytes read from i2c->address, starting at index iI2CIndex, and stored in pucDataBuffer.

Upon success, the return value should be the number of data bytes (i.e. 1 in the example), and 0 in case of a write error, to enable error management from the above layer.

### 5.10 I2C communication validation

To validate i2c communication with the platform it is recommended to test a write then a read to/from a register in one of the chips with the low_level traces activated

```
->trackWrite = 1;
->trackRead  = 1;
```

CAUTION: depending on the hardware configuration, the chip used for this test must be carefully chosen. The user needs to make sure that this chip is connected to the i2c bus and that the register used for the test can be written and read back even if no clock is present on the chip, is such is the case.

The sample top-level application delivered in the source code package should provide a set of options to allow easy i2c testing, such as:

```
------  i2c ---------
read     : read bytes from i2c
write    : write bytes to i2c
USB      : connect i2c in USB  mode
CUST     : connect i2c in CUST mode
trace    : toggle L0 traces
```

NB: when using the low-level i2c test options, the syntax is:

```
'write' 'i2c write <I2C_Address> Byte0 Byte1 Byte2'
```

- The i2c address of the chip is the first argument
- All bytes following the i2c address will be written to the chip, the first bytes having the role of setting the target index for the data bytes

```
'read' 'i2c read  <I2C_Address> <index> nbBytes'
```

- The i2c address of the chip is the first argument
- The number of bytes used for the i2c index will be equal to the number of arguments -2
- The number of bytes to read will be the last argument

A basic validation test for a Si2167 application would consist in writing the chip_mode register to any valid value.
*We don't really mind if the value is a valid functional value, as we don't want to make the chip work at this stage. What we want to achieve is validating i2c communication porting.*
The test sequence can be:
'trace' to activate the low-level traces,
'CUST' to connect in CUSTOMER mode (the mode to test),
then (abstract from the console application output):

```
write
i2c write 0xc8 0x00 0x00 32
writing 0xc8  USB> 0x00 0x00 0x20
3 bytes written
read
i2c read 0xc8 0x00 0x00 1
writing 0xc8  USB> 0x00 0x00
reading 0xc9  <USB 0x20
0x20
```

We can see here that the value '32' has been correctly written and read back from the chip.
What it means is:

1. There is a chip at address 0xc8.
2. The communication mode in use is working fine ('USB' here, as displayed in the traces)
3. I2C communication is OK with the hardware, and it is now possible to focus on building the complete application.

# 6 ANNEX 1: I2C communication in source code

This annex presents additional details on the Layer 0 and Layer 1 implementation, to illustrate the methods used in i2c communication in the source code.

## General Description

The registers are written using the following API function:
L1_WRITE(context->i2c, "reg_name", value);

The registers are read using the following API function:
L1_READ(context->i2c, "reg_name", &value);

The L1 context is a generic context containing (among other members):

- A pointer to an L0_context in order to drive the I²C module properly.
- An L0_context object which is linked to the pointer during the software initialization.

```
typedef struct L1_Context {
  L0_Context *i2c;
  L0_Context  i2cObj;
. . .
} L1_Context;
```

The most important parameter when i2c signals are concerned is the **indexSize**, indicating the number of bytes used to write the register address.

Legend

```
0x.. data transmitted by the host
0x.. data transmitted by the IP
s start
p stop
a ack from Host
a ack from IP
```

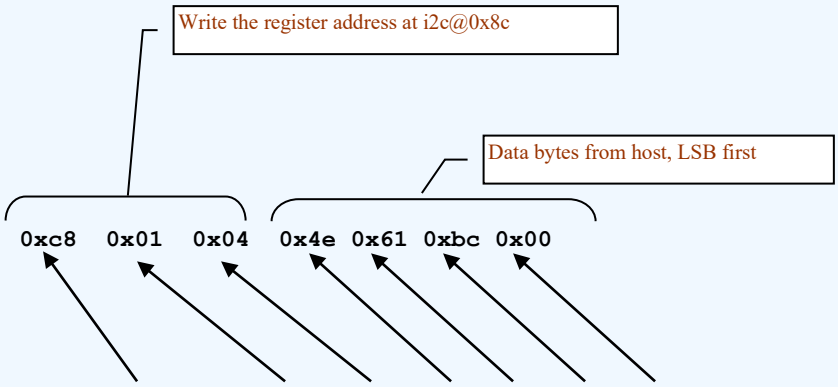
## Writing a register

An i2c write transaction does not really take into account the index size. It simply writes the bytes as a series made of:
'i2c_address(1 byte), index(indexSize bytes), data(n bytes)'
During a write, each byte sent by the host is acknowledged by the chip.
To end the transaction, the host sends a stop condition.

```
/* Enabling trackWrite mode */
L0_TrackWrite(demod->i2c,true);

/* 'standard' register definition */
/* standard                      */
```

```
 #define    standard_ADDRESS                 264
 #define    standard_OFFSET                    0
 #define    standard_NBBIT                     6
 #define    standard_ALONE                     1
 #define    standard_SIGNED                    0
  #define           standard_analog                          0
  #define           standard_dvb_t                           1
  #define           standard_dvb_h                           2
  #define           standard_dvb_c                           5
  #define           standard_dvb_s                           7
  #define           standard_dss                             8
  #define           standard_dvb_s2                         10

/* Setting the standard to 'dvb_s'*/
L1_WRITE(demod,standard,standard_dvb_s);

/* During pre-compilation processing (without traces) */
L0_WriteRegister (demod->i2c, standard_ADDRESS, standard_OFFSET, standard_NBBIT,
Standard_SIGNED, standard_dvb_s);
/* After pre-compilation processing (without traces) */
L0_WriteRegister (demod->i2c, 264, 0, 6, 0, 7);

/* During pre-compilation processing (with traces) */
L0_WriteRegister (demod->i2c, "standard", "standard_dvb_s2", standard_ADDRESS,
standard_OFFSET, standard_NBBIT, standard_SIGNED, standard_dvb_s);
/* After pre-compilation processing (without traces) */
L0_WriteRegister (demod->i2c, "standard", "standard_dvb_s2", 264, 0, 6, 0, 7);

# The resulting RWTRACES (if enabled)
>> WRITE standard standard_dvb_s: 7 (0x0007)  @[0x108:5-0]

# The resulting L0 traces (if enabled)
Layer0 writing   in 0xc8  USB> 0x01 0x08 0x07

# i2c data transmitted over the bus
s 0xc8 a 0x01 a 0x08 a 0x07 p

/* 'if_freq_shift' register definition */
/* if_freq_shift                    */
 #define    if_freq_shift_ADDRESS            260
 #define    if_freq_shift_OFFSET               0
 #define    if_freq_shift_NBBIT               29
 #define    if_freq_shift_ALONE                1
 #define    if_freq_shift_SIGNED               1

/* Setting if_freq_shift to 12345678 */
L1_WRITE(demod, if_freq_shift, 12345678);

/* During pre-compilation processing (without traces) */
L0_WriteRegister (demod->i2c, if_freq_shift_ADDRESS, if_freq_shift_OFFSET,
if_freq_shift_NBBIT, sf_freq_shift_SIGNED, if_freq_shift_dvb_s);
/* After pre-compilation processing (without traces) */
```

```
L0_WriteRegister (demod->i2c, 264, 0, 6, 0, 7);

/* During pre-compilation processing (with traces) */
L0_WriteRegister (demod->i2c, "if_freq_shift", "12345678", if_freq_shift_ADDRESS,
if_freq_shift_OFFSET, if_freq_shift_NBBIT, if_freq_shift_SIGNED,
if_freq_shift_dvb_s);

/* After pre-compilation processing (with traces) */
L0_WriteRegister (demod->i2c,"if_freq_shift","12345678", 264, 0, 6, 0, 7);

# The resulting RWTRACES (if enabled)
>> WRITE if_freq_shift 12345678: 12345678(0xbc614e) @[0x107:4]-[0x104:0]

# The resulting L0 traces (if enabled)
Layer0 writing   in 0xc8  USB> 0x01 0x04 0x4e 0x61 0xbc 0x00

# i2c data transmitted over the bus
s 0xc8 a 0x01 a 0x04 a 0x4e a 0x61 a 0xbc a 0x00 p

# meaning of i2c bytes
val = 12345678 = 0x00bc614e ➔
```




```
Layer0 writing   in 0xc8  USB> 0x01 0x04 0x4e 0x61 0xbc 0x00
```

## Reading a register

An i2c read transaction can be split in 3 parts:

1. Writing the index. This part is similar to writing 0 byte at a given index. It places the chip's internal byte index at the desired location for reading.
2. Writing the i2c_address+1 to indicate a 'read' is starting
3. The chip sending bytes until a STOP condition is issued by the host.

This means that the indexSize only changes the number of bytes sent in the first phase.

Another way to present it would be:

- Write 0 byte at the required index to position the index in the chip's state machine

- Directly read n bytes from the current index and issue a STOP when done.

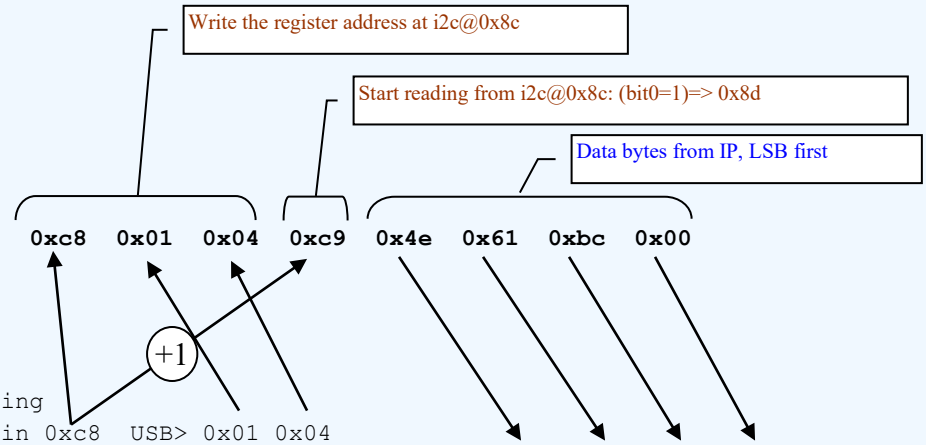
NB: For most (almost all) i2c chips, the ‘write’ and ‘read’ phases can be really treated as 2 different phases, with a STOP condition being issued after the ‘write’.

The RDA5812 (digital satellite Silicon Tuner from RDA) does not allow this method.
It requires no STOP condition between both phases to enable reads. Therefore, the i2c implementation must allow a specific case for this chip, with no STOP condition in a read.

Example in our code:

```
/* Enabling trackRead mode */
L0_TrackRead(demod->i2c,true);

/* Reading the value of 'standard' */
L1_READ(demod,standard);
---> return = 7;

# The resulting L0 traces (if enabled)
Layer0 writing   in 0xc8  USB> 0x01 0x08
Layer0 reading from 0xc8  <USB 0x07

# The resulting RWTRACES (if enabled)
<< READ  standard          :    7 (0x0007)  @[0x108:5-0]

/* Reading the value of 'if_freq_shift' */
L1_READ(demod,if_freq_shift);
---> return = 12345678;

# The resulting L0 traces (if enabled)
Layer0 writing   in 0xc8  USB> 0x01 0x04
Layer0 reading from 0xc8  <USB 0x4e 0x61 0xbc 0x00

# The resulting RWTRACES (if enabled)
<< READ  if_freq_shift     : 12345678 (0xbc614e)  @[0x107:4]-[0x104:0]

# I2c data transmitted over the bus
s 0xc8 a 0x01 a 0x04 p
s 0xc9 a 0x4e a 0x61 a 0xbc a 0x00 p

# I2c data transmitted over the bus
s 0xc8 a 0x01 a 0x04 p 0xc9 a 0x4e a 0x61 a 0xbc a 0x00 p
```

```
# meaning of i2c bytes
```




```
Layer0 writing
          in 0xc8  USB> 0x01 0x04
Layer0 reading from                0xc8  <USB 0x4e 0x61 0xbc 0x00
➔ val = 0x00bc614e = 12345678
```