DRAFT Specification for BIF Hardware Abstraction Layer

Version 0.9 Revision 03 – 24 January 2013

* NOTE TO IMPLEMENTERS *

This document is a Draft Specification. MIPI member companies' rights and obligations apply to this Draft Specification as defined in the MIPI Membership Agreement and MIPI Bylaws.

DRAFT Specification for BIF Hardware Abstraction Layer

Version 0.9

Revision 03

24 January 2013

Further technical changes to this document are expected as work continues in the BIF HAL Subgroup Working Group.

NOTICE OF DISCLAIMER

The material contained herein is not a license, either expressly or impliedly, to any IPR owned or controlled by any of the authors or developers of this material or MIPI®. The material contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, this material is provided AS IS AND WITH ALL FAULTS, and the authors and developers of this material and MIPI hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence.

All materials contained herein are protected by copyright laws, and may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of MIPI Alliance. MIPI, MIPI Alliance and the dotted rainbow arch and all related trademarks, tradenames, and other intellectual property are the exclusive property of MIPI Alliance and cannot be used without its express prior written permission.

ALSO, THERE IS NO WARRANTY OF CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THIS MATERIAL OR THE CONTENTS OF THIS DOCUMENT. IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF THIS MATERIAL OR THE CONTENTS OF THIS DOCUMENT OR MIPI BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT, SPECIFICATION OR DOCUMENT RELATING TO THIS MATERIAL, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

Without limiting the generality of this Disclaimer stated above, the user of the contents of this Document is further notified that MIPI: (a) does not evaluate, test or verify the accuracy, soundness or credibility of the contents of this Document; (b) does not monitor or enforce compliance with the contents of this Document; and (c) does not certify, test, or in any manner investigate products or services or any claims of compliance with the contents of this Document. The use or implementation of the contents of this Document may involve or require the use of intellectual property rights ("IPR") including (but not limited to) patents, patent applications, or copyrights owned by one or more parties, whether or not Members of MIPI. MIPI does not make any search or investigation for IPR, nor does MIPI require or request the disclosure of any IPR or claims of IPR as respects the contents of this Document or otherwise.

Questions pertaining to this document, or the terms or conditions of its provision, should be addressed to:

MIPI Alliance, Inc.
c/o IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08854
Attn: Board Secretary

Contents

Figures

Tables

Release History

Date	Version	Description
		No board Approved Releases.

Development History

Date	Release Target	Description
2012-12-17	v0.9 r01	Editor reformatted of working submission into standard MIPI specification template in preparation for board review.
2013-01-10	v0.9 r02	Fixed six reported issues, page numbering and moved FAQ to a separate document.
2013-01-24	v0.9 r03	Fixed two reported issues.

1 Introduction

1 The Hardware Abstraction Layer (HAL) has been defined to complement the Battery Interface Specification ***[MIPI01]***. It describes the standard software interface on top of the BIF Master signaling layer consisting of protocol and physical layer.

2 The specification is organized as follows:

3 - ***Section 2*** provides a glossary of terms and abbreviations used in the specification.

4 - ***Section 3*** provides a list of referred documents.

5 - ***Section 4*** provides an overview of a possible host-side BIF stack architecture.

6 - ***Section 5*** defines the data structures and the function declarations of the HAL.

7 - ***Section 6*** provides an overview of the interrupts mode function and operation.

8 - ***Section 7*** provides system considerations for developing practical applications.

1.1 Scope

9 The scope of the HAL specification is limited to the BIF HAL interface which defines the data types and function declaration of the HAL.

1.2 Purpose

10 The HAL enables BIF Slave suppliers to develop the BIF client driver for BIF Slave control on the host platform. Similarly, a mobile device maker can develop BIF Master high level software to control BIF Master devices from different suppliers.

11 The HAL for BIF is defined to support major Operating Systems for mobile platforms, but it can also be used in non mobile applications with MCUs that have less processing power.

2 Terminology

12 The MIPI Alliance has adopted Section 13.1 of the *IEEE Standards Style Manual*, which dictates use of the words “shall”, “should”, “may”, and “can” in the development of documentation, as follows:

> 13 The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the Specification and from which no deviation is permitted (*shall* equals *is required to*).
>
> 14 The use of the word *must* is deprecated and shall not be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.
>
> 15 The use of the word *will* is deprecated and shall not be used when stating mandatory requirements; *will* is only used in statements of fact.
>
> 16 The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).
>
> 17 The word *may* is used to indicate a course of action permissible within the limits of the Specification (*may* equals *is permitted to*).
>
> 18 The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

19 All sections are normative, unless they are explicitly indicated to be informative.

2.1 Definitions

20 **Bus:** All Master and Slave devices connected to a BCL.

21 **DeviceID:** Eight byte device identification code assigned by BIF Slave device manufacturer.

22 **Host:** System in mobile device that includes a BIF Master device and optionally BIF Slave devices.

23 **ManufacturerID:** Two byte manufacturer identification code assigned by MIPI.

24 **Master:** A device that controls the digital communication over the BCL.

25 **Mobile Device:** A complete operational mobile device including system Host with BIF Master device and optionally BIF Slave devices, with complete Battery Pack.

26 **Slave:** A device that communicates with a Master device through BCL, but do not control the digital communication.

27 **UniqueID:** World-wide unique ten byte identification number across all BIF products. Consists of an eight byte **DeviceID** and two byte **ManufacturerID.**

2.2 Abbreviations

28	e.g.	For example (Latin: exempli gratia)
29	etc.	And so forth and so on
30	i.e.	That is (Latin: id est)
31	wrt.	With respect to

2.3 Acronyms

BCL	Battery Communication Line
BIF	Battery Interface
DDB	Device Descriptor Block
ECL	Elementary Control Layer
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
ISTO	Industry Specifications and Technology Organization
LSB	Least Significant Byte
MIPI	Mobile Industry Processor Interface
ML	Manager Layer
MSB	Most Significant Byte
NVM	Non-volatile Memory
SID	SlaveID
SL	Signaling Layer
UID	UniqueID

3 References

47 [MIPI01] *MIPI Alliance Specification for Battery Interface,* Version 1.0, MIPI Alliance Inc., 23 December 2011.

48 [MIPI02] *MIPI Alliance Specification for Device Descriptor Block (DDB),* Version 1.0, MIPI Alliance, Inc., Board approved 29 October 2008, with editorial corrections authorized on 12 October 2011.

4 Overview of a Host-side BIF Stack Architecture

49 A possible host-side BIF software stack is shown in ***Figure 1***. It has four layers, the BIF Master Signaling Layer, the BIF Transaction Layer, the BIF Elementary Control Layer and the BIF Manager Layer. The HAL specification only specified the HAL interface as shown in the blue dotted line in ***Figure 1*** which defines the data types and functions declaration. The blue box in ***Figure 1*** is the actual implementation of the HAL which is entirely left to the flexibility of user implementation.

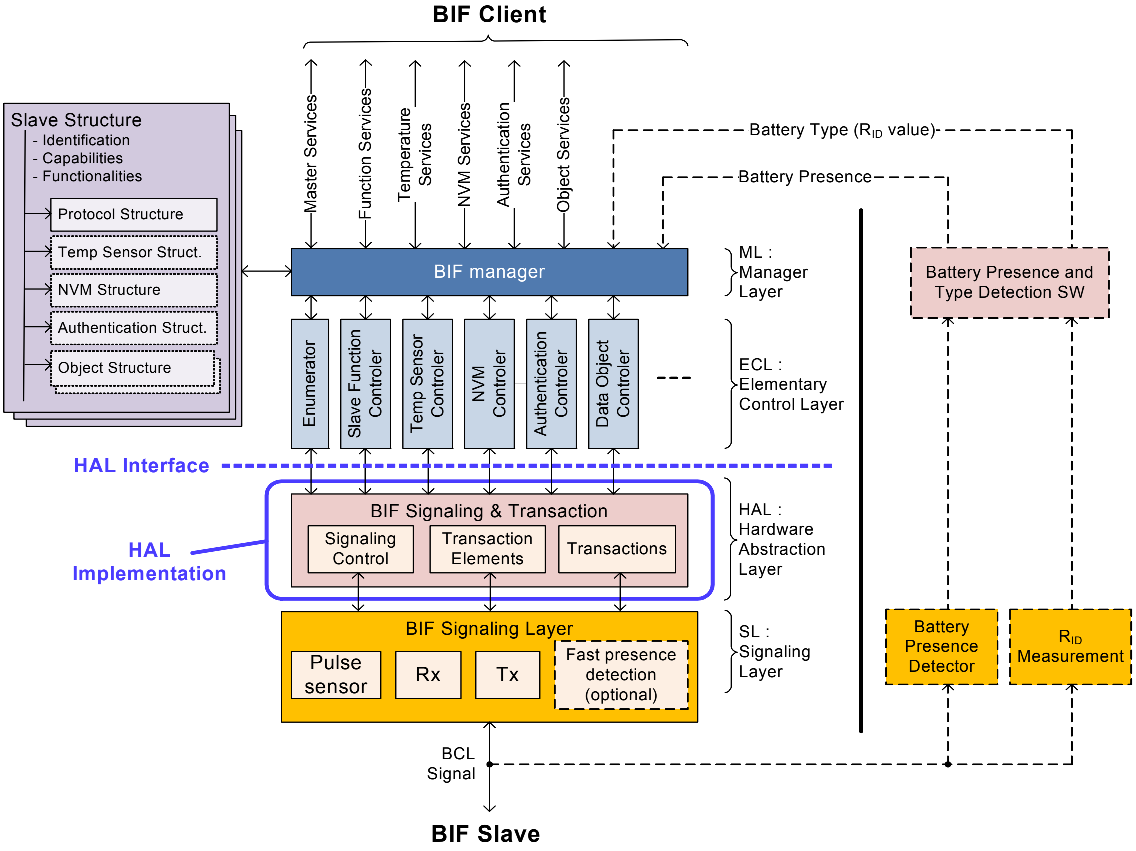

Figure 1 BIF Stack Possible Implementation Architecture

50 The Signaling Layer (SL) is typically implemented as hardware IP in the power management IC or the application processor of the mobile platform. It can also be implemented in software using a GPIO. The SL is mainly responsible for transmitting and receiving the BIF words between the Master and Slave devices. It also handles the interrupt from the Slave devices. The SL guarantees the protocol timing requirements as defined in the BIF specification. Fast presence detection is used to detect whether the battery pack with R_{ID} pull-down resistor is connected or not. The fast presence detection is optionally specified function in BIF specification ***[MIPI01]***.

51 The Hardware Abstraction Layer (HAL) provides access to the BIF protocol. It comprises the HAL interface where the data types and function calls are declared, as well as the HAL implementation. The HAL implementation consists of signaling control, transaction elements and transaction. The signaling control contains low level signaling like power-down, interrupts and other functions.

52 The transaction elements include command packet, data packet, address packet, read data packet and write data packet. The transaction consists of a sequence of transaction elements, like burst read. In addition, some data structures are defined to describe the capability and the state of the BIF layer. BIF data words are constructed and sent to the SL for transmission. BIF data words received by the SL are decoded by the HAL.

53 The Elementary Control Layer (ECL) manages various functions of a BIF device as described in the BIF specification ***[MIPI01]***. Commonly used BIF functions that can be supported in this layer are Temperature Sensor Controller, NVM Controller, Authentication Controller, etc. A data object controller can be populated to handle the BIF objects allocated in Slave and instantiate the associated object structure collection.

54 The Manager Layer (ML) provides overall control of all BIF devices present. It contains the DDB structure ***[MIPI02]*** that describes the capabilities and functions of the devices. It detects all the Slave devices and manages the UID list. It also provides the service interface to the BIF client drivers.

55 In addition to the four layers composing the BIF software stack, the Battery Presence and Type Detection block is interfaced with the battery presence detector and a R_{ID} (identification resistance) measurement. It triggers the BIF manager upon battery insertion and removal and informs about the battery type (based on R_{ID} value). The definition of R_{ID} is described in BIF specification in Section 5.1 ***[MIPI01]***.

56 The battery presence detection is not necessarily the same as the fast presence detection in SL of ***Figure 1***. The battery presence detection can alternatively be implemented in many ways, for example, by a battery cover case switch or by monitoring the battery voltage.

57 The Battery Presence and Type Detection blocks are optional. The BIF manager can be informed about battery removal to reset any Slave structures it has populated.

5 HAL Data Types and Functions

58 The HAL interface declaration is available in a directly usable header file. This header file is written in C/C++ language and is shown in ***Annex A.1***

59 The HAL header file contains several parts as described in the following subsections.

5.1 Data Types

5.1.1 Basic Types

60 This section provides local declaration of basic types used in the HAL (for example, N bits integers, signed or unsigned). The declaration corresponds to standard "stdint.h" integer declaration available for most platforms.

61 The HAL basic types can be specifically modified to fit a particular application. This is done by declaring the types in another file and defining the BIF_BASIC_TYPES_DEFINED before including the BIF HAL header file. In this case, the declaration of BIF basic types in HAL header file is over-ridden by the new declarations.

62 The HAL basic data types shall follow the definitions of ***Table 1***.

Table 1 Basic Types Declaration

Type Name	Description
BIFint8	Signed 8-bit integer
BIFint16	Signed 16-bit integer
BIFint32	Signed 32-bit integer
BIFuint8	Unsigned 8-bit integer
BIEuint16	Unsigned 16-bit integer
BIFuint32	Unsigned 32-bit integer
BIFchar	8-bit ASCII character
BIFbool	Boolean (**BIF_TRUE** or **BIF_FALSE**)

5.1.2 Default Handle Types

63 This section provides a default declaration of the BIFhandle type. Except for simple platforms, the BIFhandle default type will not match the requirements of BIF HAL integration in an operating system. This BIFhandle type should be declared specifically for each target system and an external BIFhandle server may have to be designed.

64 Using a target system specific BIFhandle type consists of declaring the BIFhandle type and defining BIF_BIFHANDLE_TYPE_DEFINED prior to including the BIF HAL header file. In this case the declaration of BIFhandle type in the HAL header file is over-ridden by the new declarations.

65 For example, the corresponding BIF handles can be returned by a function based on system level information such as BCL number. The detailed handle acquiring mechanism can be different in each system and is beyond the scope of this specification.

5.1.3 HAL Specific Types

66 This section declares types used to cover the HAL functionality.

67 Normally it is not required to have a target system specific declaration for these types.
If BIF_ HAL_TYPES_DEFINED is defined prior to including the HAL header file, the declaration of types in HAL header file is over-ridden by the new declarations.

68 The HAL type shall follow the definitions of ***Table 2***, ***Table 3***, ***Table 4*** and ***Table 5***.

Table 2 *BIFresult* Type

Type Name	Description
BIFresult	Result of BIF HAL function call. **BIFresult** is a 16-bit data which will be: **BIFRESULT_NO_ERROR** : no error **BIFRESULT_TIME_OUT** : time-out occurred **BIFRESULT_ABORTED** : command has been aborted **BIFRESULT_CONTACT_BROKEN** : BCL has been disconnected. This result can be reported only if contact break detector is embedded in the hardware (see capabilities). **BIFRESULT_NO_EVENT**: No interrupt or contact-broken is detected **BIFRESULT_ERR_VENDOR_SPEC** : vendor specific errors **BIFRESULT_ERR_HW_INIT** : Hardware initialization error **BIFRESULT_ERR_HW_FAIL** : Hardware failure **BIFRESULT_ERR_HW_BUSY** : Hardware busy **BIFRESULT_ERR_SLAVE_NO_RESP** : Slave has not responded to a read in time **BIFRESULT_ERR_SLAVE_NACK** : Slave reported an error at read (ACK is 0) **BIFRESULT_ERR_SLAVE_EOT** : Slave terminated transmission earlier than burst length requested **BIFRESULT_ERR_SLAVE_SIG** : A signaling error is seen during Slave transmission (parity, inversion, BCF) **BIFRESULT_ERR_NOT_SUPPORTED** : The called function is not implemented **BIFRESULT_ERR_OUT_OF_RANGE** : Function parameter is out of range

Table 3 *BIFslaveError* Type

Type Name	Description
BIFslaveError	Error responded by the Slave in case of NACK response during read or in case of TQ stating error. **BIFslaveError** is a 8-bit data which will be: **BIFSLAVEERROR_NO_ERROR** : no error **BIFSLAVEERROR_GEN_COM** : general communication error **BIFSLAVEERROR_PARITY** : parity error **BIFSLAVEERROR_INV** : Inversion error **BIFSLAVEERROR_LENGTH** : Invalid word length error **BIFSLAVEERROR_TIMING** : Timing error **BIFSLAVEERROR_UNKWN_CMD** : un-known command error **BIFSLAVEERROR_SEQ** : Wrong sequence error **BIFSLAVEERROR_BUS_COL** : Bus collision error **BIFSLAVEERROR_BUSY** : Slave busy error **BIFSLAVEERROR_FATAL** : Fatal error **BIFSLAVEERROR_VENDOR_BASE** : Slave vendor specific error base

Table 4 ***BIFsigCapabilities*** **Type**

Type Name	Description
BIFsigCapabilities	Structure to inform about the signaling layer capabilities.
BIFsigCapabilities::LowPowerTauBifSupport	**BIF_TRUE** if the low power TauBif mode is supported else **BIF_FALSE**.
BIFsigCapabilities::OptimizedReadBurstLength	Describes the read FIFO length of BIF Master. Making read operation length equal to or shorter than this value ensures that the read operation will be completed in a single read burst.
BIFsigCapabilities::ManufacturerId	Manufacturer identifier of BIF Master.
BIFsigCapabilities::DeviceId	Device identifier of BIF Master.
BIFsigCapabilities::ContactBreakDetTime	Informs about detection time guaranteed by the BIF Master contact break sensor. Time will be expressed in 100 ns unit (for example, 134 means 13.4 µs detection time). If the Master doesn't embed contact break detector, the reported time will be 0.
BIFsigCapabilities::NonBlockingInterruptSupported	**BIF_TRUE** if non-blocking interrupt is supported else **BIF_FALSE.**

Table 5 ***BIFsigState*** **Type**

Type Name	Description
BIFsigState	Structure informing about the signaling layer state.
BIFsigState::TauBifNormalMode	If low power TauBif mode is selected, it will be **BIF_FALSE**. If normal TauBif mode is selected, it will be **BIF_TRUE**.
BIFsigState::TauBif	TauBif for the normal mode operation. TauBif will be expressed in 100 ns unit (for example, 25 means 2.5 µs).
BIFsigState::LowPowerTauBif	If low power TauBif operation is not supported, the value will be 0. If low power TauBif is supported, the value reports TauBif for this mode will be expressed in 100 ns unit (for example, 25 means 2.5 µs).
BIFsigState::BCLpadState	It will report the current BCL pad state (**BIF_FALSE**: logic low, **BIF_TRUE**: logic high).
BIFsigState::SlaveAddressSelected	It will report the latest Slave selection address (done with SDA command). This address is the reference for Slave selection cache functionality.
BIFsigState::SlaveSelectionCache	If **BIF_TRUE**, Slave selection cache will be enabled. In such case, any SDA command selecting an address equal to **SlaveAddressSelected** is ignored. If **BIF_FALSE**, all SDA commands will be emitted even if they select again same Slave.

5.2 Function Prototypes

69 All functions in the BIF HAL shall return a **BIFresult** data stating about:

- **70** Success of the function execution (no error, no time-out)
- **71** Error during function execution
- **72** Time-out

73 All functions in the BIF HAL shall take a handle **BIFhandle** as the first parameter.

74 The HAL function prototypes shall follow the definition of ***Table 6*** to ***Table 54***.

5.2.1 Signaling Layer Function Declarations

75 This section declares functions which interact with signaling layer. They allow controlling the BIF Master for some generic operations.

Table 6 *bifHALGetVersion* Function Prototype

Topic	Description
Prototype	BIFresult **bifHALGetVersion**(BIFhandle handle, BIFuint16 * version);
Role	This function returns the HAL interface version implemented. This should match the **BIF_HAL_HEADER_VERSION** defined version in the header file.
Parameters	**version** : pointer populated with the implemented version of the interface.

Table 7 *bifHALGetString* Function Prototype

Topic	Description
Prototype	BIFresult **bifHALGetString**(BIFhandle handle, BIFuint8 stringID, BIFchar * string);
Role	This function populates the "string" pointer with the corresponding constant "stringID" string. "stringID" is one of defined **BIF_STRING_ID_**... The string reported has length below or equal **BIF_STRING_MAX_LENGTH**. If stringID is out of supported range, the function returns **BIFRESULT_ERR_OUT_OF_RANGE.**
Parameters	**stringID** : ID of the string to be retrieved. **string** : string pointer which will be populated with the corresponding string.

Table 8 *bifHALGetLastError* Function Prototype

Topic	Description
Prototype	BIFresult **bifHALGetLastError**(BIFhandle handle, BIFuint8 * error);
Role	Report the error code reported by Slave. If precedent function call was given **BIFresult** at **BIFRESULT_ERR_SLAVE_NACK**, it means that corresponding transaction went well at signaling level but the slave has not understood it correctly. With NACK reporting from Slave, an error code is provided which can be retrieved with this **bifHALGetLastError** function.
Parameters	**error** : the pointer to be populated with the last error code reported by Slave. The error code follows the defined **BIFSLAVEERROR_**...

Table 9 ***bifHALGetCapabilities*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifHALGetCapabilities**(BIFhandle handle, BIFsigCapabilities * capabilities);
Role	Retrieves the signaling layer capabilities.
Parameters	**capabilities** : the pointer to be populated with the signaling layer capabilities.

Table 10 ***bifHALInit*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifHALInit**(BIFhandle handle);
Role	Initializes the signaling layer to a default inactive state.
Parameters	None

Table 11 ***bifHALEnabling*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifHALEnabling**(BIFhandle handle, BIFbool enable); Signaling layer enabling.
Role	Calling this function with "enable" at **BIF_FALSE** places the signaling layer in inactive state (BCL is not driven). If "enable" is **BIF_TRUE**, the signaling layer Masters the BCL.
Parameters	**enable** : requested enabling state of the signaling layer

Table 12 ***bifHALSetTauBif*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifHALSetTauBif**(BIFhandle handle, BIFuint16 tauBif);
Role	TauBif setting. Configure and select the normal TauBIF timing. The effective TauBif configured is reported in field **TauBif** of **BIFsigState** structure
Parameters	**tauBif** : The tauBif parameter is expressed in 100 ns unit (2.3 µs corresponds to tauBif = 23).

Table 13 ***bifHALSetLowPowerTauBif*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifHALSetLowPowerTauBif**(BIFhandle handle, BIFuint16 tauBif);

Table 13 *bifHALSetLowPowerTauBif* Function Prototype

Topic	Description
Role	Set and select TauBif to the most power efficient value supported by the HW. If **LowPowerTauBifSupport** is supported in the **BIFsigCapabilities**, calling the **bifHALSetLowPowerTauBif** function configures the hardware to run with the best compromise of transmission speed vs power consumption. Typically, **bifHALSetLowPowerTauBif** will configure the hardware to use a low power clock tree which is generally at low frequency. As a result, TauBif would be clamped to a long value. The effective TauBif for this mode is reported in field **LowPowerTauBif** of **BIFsigState** structure.
Parameters	**tauBif** : The tauBif parameter is expressed in 100 ns unit (100 µs corresponds to tauBif=1000).

Table 14 *bifHALSlaveSelectionCache* Function Prototype

Topic	Description
Prototype	BIFresult **bifHALSlaveSelectionCache**(BIFhandle handle, BIFbool enable);
Role	Slave selection cache control. If cache is enabled, any SDA command selecting a Slave which has already been selected with a previous SDA command are ignored and not emitted. If cache is disabled, all SDA commands are emitted even if they select again same Slave. The known current selected Slave and state of selection cache are mentioned inside the **BIFsigState** structure reported with **bifHALGetState** function.
Parameters	**enable** : **BIF_TRUE** enables the selection cache, **BIF_FALSE** disables it.

Table 15 *bifHALGetState* Function Prototype

Topic	Description
Prototype	BIFresult **bifHALGetState**(BIFhandle handle, BIFsigState * state);
Role	Get state of signaling layer.
Parameters	**state** : the pointer to be populated with the signaling layer state.

Table 16 *bifHALVendorSpecific* Function Prototype

Topic	Description
Prototype	BIFuint16 **bifHALVendorSpecific**(BIFhandle handle, BIFuint32 command, void * data_in, void * data_out);
Role	Generic access point to vendor specific signaling layer functionalities. This function returns a 16-bit data defined by vendor implementation. Some vendor implementations can define the function to be called as below. This typical implementation allows retrieving exact vendor error code if another function responds **BIFRESULT_ERR_VENDOR_SPEC** as **BIFresult**. *BIFuint16 vendorError = bifHALVendorSpecific(handle, 0, (void*)0, (void*)0);*
Parameters	**command** : vendor specific command to be executed. **data_in** : generic pointer to the data input used by the vendor specific command. **data_out** : generic pointer populated by the result of the vendor specific command.

5.2.2 Transaction Elements Function Declarations

76 This section declares functions processing transaction elements on BIF bus as defined in the BIF specification ***[MIPI01]*** in the sections on BIF commands and raw data access.

Table 17 *bifTrans_BRES* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_BRES**(BIFhandle handle);
Role	Send BIF command BRES (Bus Reset)
Parameters	None

Table 18 *bifTrans_PWDN* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_PWDN**(BIFhandle handle);
Role	Send BIF command PWDN (Power down)
Parameters	None

Table 19 *bifTrans_HardRST* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_HardRST**(BIFhandle handle);
Role	Keep BCL low for t_{PDL}
Parameters	None

Table 20 *bifTrans_STBY* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_STBY**(BIFhandle handle);
Role	Send BIF command STBY (standby)
Parameters	None

Table 21 *bifTrans_WakeUpPulse* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_WakeUpPulse**(BIFhandle handle);
Role	Produces activation signal in order to go out of power down or standby mode. This function will not wait for the time needed to re-activate the Slaves. The timing will be managed at the caller side.
Parameters	None

Table 22 ***bifTrans_WakeUpFromPWDN*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_WakeUpFromPWDN**(BIFhandle handle);
Role	Produces an activation signal and wait for tPUP delays (BIF Specification Table 36) in order to go out of power-down mode.
Parameters	None

Table 23 ***bifTrans_WakeUpFromSTBY*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_WakeUpFromSTBY**(BIFhandle handle);
Role	Produces an activation signal and wait for tACT delays (BIF Specification Table 36) in order to go out of standby mode.
Parameters	None

Table 24 ***bifTrans_ISTS*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_ISTS**(BIFhandle handle, BIFbool * intStatus);
Role	Send BIF command ISTS (interrupt status)
Parameters	**intStatus** : pointer populated with **BIF_TRUE** if Slave reports a pending interrupt, else **BIF_FALSE**.

Table 25 ***bifTrans_RBL*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_RBL**(BIFhandle handle, BIFuint8 length);
Role	Send BIF command RBL (read burst length).
Parameters	**length**: Length of the read burst.

Table 26 ***bifTrans_RBE*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_RBE**(BIFhandle handle, BIFuint8 length);
Role	Send BIF command RBE (read burst extended).
Parameters	**length**: Length of the read burst.

Table 27 *bifTrans_DASM* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DASM**(BIFhandle handle);
Role	Send BIF command DASM (multiple device address selection).
Parameters	None

Table 28 *bifTrans_DISS* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DISS**(BIFhandle handle);
Role	Send BIF command DISS (start of ID search).
Parameters	None

Table 29 *bifTrans_DILC* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DILC**(BIFhandle handle, BIFbool * dilc);
Role	Send BIF command DILC (Probe that last bit of ID search is reached).
Parameters	**dilc** : pointer populated with **BIF_TRUE** if last bit of ID is reached else populated with **BIF_FALSE**.

Table 30 *bifTrans_DIE0* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DIE0**(BIFhandle handle);
Role	Send BIF command DIE0 (Enter in branch bit at 0 for ID search).
Parameters	None

Table 31 *bifTrans_DIE1* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DIE1**(BIFhandle handle);
Role	Send BIF command DIE1 (Enter in branch bit at 1 for ID search).
Parameters	None

Table 32 *bifTrans_DIP0* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DIP0**(BIFhandle handle, BIFbool * dip0);
Role	Send BIF command DIP0 (Probe if current bit is 0 for ID search).

Table 32 *bifTrans_DIP0* Function Prototype

Topic	Description
Parameters	**dip0** : pointer populated with **BIF_TRUE** if a Slave responds positively, else populated with **BIF_FALSE**.

Table 33 *bifTrans_DIP1* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DIP1**(BIFhandle handle, BIFbool * dip1);
Role	Send BIF command DIP1 (Probe if current bit is 1 for ID search).
Parameters	**dip1** : pointer populated with **BIF_TRUE** if a Slave responds positively, else populated with **BIF_FALSE**.

Table 34 *bifTrans_DRES* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_DRES**(BIFhandle handle);
Role	Send BIF command DRES (Device reset).
Parameters	None

Table 35 *bifTrans_TQ* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_TQ**(BIFhandle handle, BIFslaveError * errorCode, BIFuint8 * dataCount);
Role	Send BIF command TQ (transaction query).
Parameters	**errorCode** : if Slave reports error, the pointer is populated with the reported error code, else populated with **BIFSLAVEERROR_NO_ERROR** value. **dataCount** : if Slave doesn't report error, the pointer is populated with the reported data count, else populated with 0.

Table 36 *bifTrans_AIO* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_AIO**(BIFhandle handle);
Role	Send BIF command AIO (Address auto-increment disable).
Parameters	None

Table 37 *bifTrans_EDA* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_EDA**(BIFhandle handle, BIFuint8 add);

Table 37 ***bifTrans_EDA*** **Function Prototype**

Topic	Description
Role	Send BIF command EDA (extended Slave address).
Parameters	**add** : Slave address extension.

Table 38 ***bifTrans_SDA*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_SDA**(BIFhandle handle, BIFuint8 add);
Role	Send BIF command SDA (Slave address).
Parameters	**add** : Slave address.

Table 39 ***bifTrans_ERA*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_ERA**(BIFhandle handle, BIFuint8 add);
Role	Send BIF command ERA (extended address).
Parameters	**add** : address extension.

Table 40 ***bifTrans_WRA*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_WRA**(BIFhandle handle, BIFuint8 add);
Role	Send BIF command ERA (write address).
Parameters	**add** : address.

Table 41 ***bifTrans_WD*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_WD**(BIFhandle handle, BIFuint8 data);
Role	Send BIF command WD (write data).
Parameters	**data** : data to be written.

Table 42 ***bifTrans_RRA*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_RRA**(BIFhandle handle, BIFuint8 add, BIFuint8 * data, BIFuint8 length);
Role	Send BIF command RRA (read data at address). ***Note:*** *It is not recommended to use RRA directly as the data count transmitted by Slave depends directly on previous command sent to it (burst length configuration). Use a high-level read functions instead.*

Table 42 ***bifTrans_RRA*** **Function Prototype**

Topic	Description
Parameters	**add** : address of the read operation to be performed. **data** : pointer populated with the data read from the Slave. **length** : number of data expected to be read.

Table 43 ***bifTrans_EINT*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_EINT**(BIFhandle handle, BIFuint16 timeOut);
Role	Send BIF command EINT (enable interrupt mode) and wait for completion. Completion can be: • Interrupt signalled by Slave (function returns **BIFRESULT_NO_ERROR**) • Time-out (function returns **BIFRESULT_TIME_OUT**) • Abort (function returns **BIFRESULT_ABORTED**) • BCL contact broken if such detection is supported by Master (function returns **BIFRESULT_CONTACT_BROKEN**) In case of time-out or contact-broken completion, the function aborts the interrupt mode on BIF bus so that any command can be send to the Slave without any additional action. The interrupt mode can be aborted by the **bifTrans_Abort_EINT** function called from another thread (useful in case of infinite time-out value). In such case, the function returns **BIFRESULT_ABORTED**.
Parameters	**timeOut** : timeout is expressed in ms. If timeOut is 0, the function waits forever (infinite time-out). As such this should be used with care especially in single tasking system or system that does bit-banging where there is no way to break the infinite loop and to abort the IRQ.

Table 44 ***bifTrans_Async_EINT*** **Function Prototype**

Topic	Description
Prototype	BIFresult **bifTrans_Async_EINT** (BIFhandle handle);
Role	Sending command EINT (enable interrupt mode) and don't wait for completion. This function is effectively implemented only if non-blocking interrupt mode is supported by the Master. This support is reported in the field **NonBlockingInterruptSupported** of **BIFsigCapabilities** structure after calling **bifHALGetCapabilities** function. After sending the EINT command to Slaves, the function engages BCL monitoring on a background process (Hardware cell commonly) and returns. The BIF bus is not available for more transactions. The function caller must manage resuming from interrupt mode. The function returns **BIFRESULT_NO_ERROR** if the background process is correctly started. Otherwise, it returns the appropriate error information. The interrupt survey engaged on the background process is stopped when: • The bifTrans_Abort_EINT function is called. • Slaves produce interrupt. A BCL contact break detection (if such detection is supported by the Master) does not stop the interrupt survey background process, a call to bifTrans_Abort_EINT function is required. By using the **bifTrans_Query_EINT** function, it is possible to get the state of the background interrupt survey process. If the Master does not support non-blocking interrupt mode, the function returns **BIFRESULT_ERR_NOT_SUPPORTED**

Table 44 *bifTrans_Async_EINT* Function Prototype

Topic	Description
Parameters	None

Table 45 *bifTrans_Query_EINT* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_Query_EINT** (BIFhandle handle);
Role	Probe the background interrupt survey process state. This function is effectively implemented only if non-blocking interrupt mode is supported by the Master. This support is reported in the field **NonBlockingInterruptSupported** of **BIFsigCapabilities** structure after calling **bifHALGetCapabilities** function. When interrupt survey background process is engaged after a successful call of **bifTrans_Async_EINT**, the function returns the following: • **BIFRESULT_NO_ERROR** if a Slave interrupt has been detected. • **BIFRESULT_CONTACT_BROKEN** if a BCL contact break has been detected (if supported by Master). • **BIFRESULT_NO_EVENT** if no interrupt or contact-broken is detected. If the Master does not support non-blocking interrupt mode, the function returns **BIFRESULT_ERR_NOT_SUPPORTED.**
Parameters	None

Table 46 *bifTrans_Abort_EINT* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_Abort_EINT**(BIFhandle handle);
Role	Abort the interrupt mode engaged by **bifTrans_EINT** or **bifTrans_Async_EINT** commands.
Parameters	None

Table 47 *bifTrans_SlaveVendorSpecific* Function Prototype

Topic	Description
Prototype	BIFresult **bifTrans_SlaveVendorSpecific**(BIFhandle handle, BIFuint8 commandNibble);
Role	Send Slave vendor specific command.
Parameters	**commandNibble** : command code from 0x00 to 0x0F

5.2.3 Transaction Function Declarations

77 This section declares functions processing transactions on the BIF bus. These functions are typically called transaction elements. These functions code popular operations to select Slave or to access data on BIF Slaves.

Table 48 *bifTransWriteUint8* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransWriteUint8**(BIFhandle handle, BIFuint16 address, BIFuint8 data);

Table 48 *bifTransWriteUint8* Function Prototype

Topic	Description
Role	Write an 8-bit data at a 16-bit address into the Slave.
Parameters	**address** : 16-bit address where the transaction acts **data** : 8-bit data to be written

Table 49 *bifTransWriteUint16* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransWriteUint16**(BIFhandle handle, BIFuint16 address, BIFuint16 data);
Role	Write a 16-bit data at a 16-bit address into the Slave.
Parameters	**address** : 16-bit address where the transaction acts **data** : 16-bit data to be written. Most significant byte [bit 15:8] shall be written first, followed by least significant byte [bit 7:0].

Table 50 *bifTransWriteMultUint8* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransWriteMultUint8**(BIFhandle handle, BIFuint16 address, BIFuint8 * data, BIFuint8 size);
Role	Write an array of 8bits data starting at a 16bits address into the Slave.
Parameters	**address** : 16-bit address where the transaction starts **data** : pointer to the array of 8-bit data to be written. Least significant byte (data [0]) shall be written first, and most significant byte (data [size-1]) shall be written last. **size** : size of the array to write.

Table 51 *bifTransReadUint8* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransReadUint8**(BIFhandle handle, BIFuint16 address, BIFuint8 * data);
Role	Read an 8-bit data at a 16-bit address from the Slave.
Parameters	**address** : 16-bit address where the transaction acts. **data** : pointer populated with the 8-bit data read.

Table 52 *bifTransReadUint16* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransReadUint16**(BIFhandle handle, BIFuint16 address, BIFuint16 * data);
Role	Read a 16-bit data at a 16-bit address from the Slave.
Parameters	**address** : 16-bit address where the transaction acts. Most significant byte [bit 15:8] shall be read first, followed by least significant byte [bit 7:0]. **data** : pointer populated with the 16-bit data read

Table 53 *bifTransReadMultUint8* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransReadMultUint8**(BIFhandle handle, BIFuint16 address, BIFuint8 * data, BIFuint16 size);
Role	Read an array of 8-bit data starting at a 16-bit address from the Slave. BIF specification [MIPI0] supports burst read of 1 to 256 bytes. If the size is specified greater than 256 bytes, the function returns **BIFRESULT_ERR_OUT_OF_RANGE.**
Parameters	**address** : 16-bit address where the transaction starts. **data** : pointer to an array populated by the 8-bit data read. Least significant byte (data[0]) shall be read first, and most significant byte (data[size-1]) shall be read last. **size** : count of 8-bit data to read (size of the read array).

Table 54 *bifTransSelUid* Function Prototype

Topic	Description
Prototype	BIFresult **bifTransSelUid**(BIFhandle handle, BIFuint8 * Uid);
Role	Select a Slave by its UID.
Parameters	**Uid** : pointer to a 10 x 8-bit data corresponding to the UID of the Slave to select.

6 Interrupts and Events

78 MIPI BIF Slave devices have an interrupt mechanism for informing the host that interaction with the Slave is required, for example, when the Slave has new data available for the host to read. The host can either directly read the Slaves' interrupt status from the interrupt control registers, query the Slaves to see if an interrupt is pending or direct the bus to an interrupt mode, where the Slaves can actively notify the host of a flagged interrupt. This chapter describes the MIPI BIF interrupt mechanism from the point of view of MIPI BIF HAL. More information can be found in the BIF specification ***[MIPI01]***.

6.1 Accessing the Interrupt Control Register

79 The Slave interrupt control registers accessed via the Slave Control Function, which is discoverable via the Function Directory as specified in the BIF specification ***[MIPI01]***. These features are accessed through Slave device registers. The HAL operations for reading and writing Slave registers are described in ***Section 5***.

80 If the host discovers that there is a Slave that has a pending interrupt by using either the interrupt status query or the interrupt mode, the host should read the interrupt control registers of all of the Slaves to determine which Slave has issued the interrupt and which function caused the interrupt.

6.2 Querying the Interrupt Status

81 The host can issue the Bus Command Interrupt Status (ISTS) to query if any Slave has a pending interrupt. The Slaves respond with a Bus Query pulse. If no Slaves respond within the allotted time tBUSQ, no interrupts are pending on any of the Slaves. When the HAL function **bifTrans_ISTS** is called it issues an ISTS command on the bus:

82
```
BIFresult bifTrans_ISTS(BIFhandle handle, BIFbool * intStatus)
```

83 If the function call was successful, **bifTrans_ISTS** returns the result of the query in the **intStatus** argument. The value that is stored in **intStatus** will be **BIF_TRUE** if one or more Slaves responded to the query. If no Slaves responded, **intStatus** will be **BIF_FALSE**. The host should then go through the Slaves' interrupt control registers to locate the source or sources of the interrupt. The value **BIF_FALSE** signifies that no Slave responded within t_{BUSQ}.

6.3 Blocking (Synchronous) Interrupt Mode

84 The blocking (synchronous) interrupt mode is similar to the interrupt status query with the difference that there is no implicit timeout. Instead the bus enters the interrupt mode when the host issues the Bus Command Enable Interrupt (EINT). The bus exits the interrupt mode only when one of the Slaves or the host issues an interrupt pulse on the BCL. The host should check the interrupt control registers of the Slaves for any pending interrupts. The host should do this even if the host issued the interrupt pulse, because it is possible that one of the Slaves also happened to send an interrupt pulse at the same time.

85
```
BIFresult bifTrans_EINT(BIFhandle handle, BIFuint16 timeOut)
```

86 The HAL function **bifTrans_EINT** is used to enter the interrupt mode. The optional parameter **timeOut** can be used to force the host the break out of the interrupt mode after a certain amount of time. The unit of **timeOut** is milliseconds and a value of 0 issued when no timeout is required. If **bifTrans_EINT** timeouts, the host will automatically issue an interrupt pulse to bring the bus out of the interrupt mode and **bifTrans_EINT** returns **BIFRESULT_TIME_OUT**. If **bifTrans_EINT** exits due to a Slave interrupt pulse, **BIFRESULT_NO_ERROR** is returned. If **bifTrans_EINT** exits because **bifTrans_Abort_EINT** was called, **bifTrans_Abort_EINT** returns **BIFRESULT_ABORT**. In all of these cases either a Slave or the host has sent an interrupt pulse to the bus and the bus is no longer in the interrupt mode.

87 After **bifTrans_Abort_EINT** terminates, the host should always remember to check for any pending interrupts. Even if **bifTrans_Abort_EINT** terminated due to a timeout or an abort, one of the Slaves might have issued a concurrent interrupt pulse. The host should use the ISTS query to check if any Slaves have pending interrupts.

88
```
    BIFresult bifTrans_Abort_EINT(BIFhandle handle)
```

89 The function **bifTrans_Abort_EINT** is used to abort a pending **bifTrans_EINT** and force and exit from the interrupt mode. **bifTrans_Abort_EINT** should be called from another thread or an interrupt handler or other such context to break the **bifTrans_EINT** blocking wait. The BIF bus can be accidentally in active mode, e.g., due to a short contact break.

90 When the **bifTrans_EINT** function call is pending, other functions that send or receive data from the bus cannot be used. These functions shall return **BIFRESULT_ERR_HW_BUSY**.

6.4 Non-Blocking (Asynchronous) Interrupt Mode

91 In addition to the blocking (synchronous) interrupt mode described in ***Section 6.3***, a HAL can implement a non-blocking (asynchronous) interrupt mode. The HAL shall indicate support for this feature by setting the field **NonBlockingInterruptSupported** in **BIFsigCapabilities** to **BIF_TRUE**. If non-blocking interrupt mode is not supported, **NonBlockingInterruptSupported** shall be **BIF_FALSE**.

92 The non-blocking interrupt mode is entered by calling **bifTrans_Async_EINT**. When this function is called, the HAL shall issue an EINT Bus Command on the BCL and the function shall be immediately completed after the transmission. A HAL that does not support non-blocking interrupt mode shall return **bifTrans_Async_EINT** with **BIFRESULT_ERR_NOT_SUPPORTED**.

93 Once in the non-blocking interrupt mode, the state of the bus can be queried with **bifTrans_Query_EINT**. The function returns **BIFRESULT_NO_ERROR** if an interrupt has been detected on the BCL, **BIFRESULT_CONTACT_BROKEN** if a contact break has been detected or **BIFRESULT_NO_EVENT** if HAL has detected neither a contact break nor interrupt. A HAL that does not support non-blocking interrupt mode shall return **bifTrans_Query_EINT** with **BIFRESULT_ERR_NOT_SUPPORTED**.

94 The host can interrupt the non-blocking interrupt mode by calling **bifTrans_Abort_EINT**, which shall cause the HAL to issue an interrupt pulse on the BCL and terminate the non-blocking interrupt mode as if an interrupt had been detected on the BCL.

95 When the HAL is in non-blocking interrupt mode, other functions that send or receive data from the bus cannot be used. These functions shall return **BIFRESULT_ERR_HW_BUSY**.

7 System Considerations (Informative)

7.1 Typical Examples of BIF HAL Usage

7.1.1 Basic Set-up of Transaction Layer

96 After checking the correspondence of implemented version with the header file value, the layer is initialized and enabled. TauBif is set and BIF bus is reset. The coding example is shown in ***Annex B.1.1***

7.1.2 Vendor Specific Access

97 The vendor specific access verifies the Master vendor and device names. If they match the known device, specific commands are issued. The coding example is shown in ***Annex B.1.2***

7.1.3 Low Power Interrupt Mode

98 The system switches to low power tauBif if supported by the Master HW and not already selected. Then, an EINT command is sent. As low power timing was selected prior to send the EINT command, the Slave will signal its interrupt with this slow timing. Hence, the Master is ensured to catch the interrupt even if running at slow clock. After interrupt signaling from Slave, previous tauBif clocking is selected again. The coding example is shown in ***Annex B.1.3***.

7.1.4 Selecting a Slave

99 The Slave can be selected by its known logic address. Otherwise, it can be selected by its UID. The coding example is shown in ***Annex B.1.4***.

7.1.5 Read DDBL1

100 The coding example for reading the DDBL1 value is shown in ***Annex B.1.5***.

7.1.6 Output Error Code to Console

101 The coding example for output of error code to console is shown in ***Annex B.1.6***.

7.1.7 UID Device Search

102 The coding example for performing a UID search is shown in ***Annex B.1.7***.

Annex A Header File Source Code

A.1 HAL Header File

```
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//
// This header file declare types and functions of BIF Hardware Abstraction Layer
// //
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------

#ifndef BIF_HAL_H
#define BIF_HAL_H

//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
// Header file version
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------

#define BIF_HAL_HEADER_VERSION    (11)

//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
// Basic types declaration
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------

#ifndef BIF_BASIC_TYPES_DEFINED
#define BIF_BASIC_TYPES_DEFINED

#include <stdint.h>

typedef int8_t     BIFint8;
typedef int16_t    BIFint16;
typedef int32_t    BIFint32;
typedef uint8_t    BIFuint8;
typedef uint16_t   BIFuint16;
typedef uint32_t   BIFuint32;

typedef uint8_t    BIFchar;

typedef uint8_t    BIFbool;
#define BIF_TRUE   (1)
#define BIF_FALSE  (0)

#endif

//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
// Default handle type define
//
// The handle exact definition is application specific. It should be redefined at
//   target implementation
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------

#ifndef BIF_HANDLEBIF_TYPE_DEFINED
#define BIF_HANDLEBIF_TYPE_DEFINED

typedef BIFuint8 BIFhandle;
```

```

#endif

//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
// HAL specific types
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#ifndef BIF_HAL_TYPES_DEFINED
#define BIF_HAL_TYPES_DEFINED

//...................................................................................
// BIFresult
//...................................................................................
#define BIFRESULT_NO_ERROR        (0x0000)   // no error
#define BIFRESULT_TIME_OUT        (0x0001)   // time-out occurred
#define BIFRESULT_ABORTED         (0x0002)   // command has been aborted
#define BIFRESULT_CONTACT_BROKEN  (0x0003)   // BCL has been disconnected.
                                             // This result can be reported only if
                                             // contact break detector is embedded in
                                             // the HW.
#define BIFRESULT_NO_EVENT        (0x0004)   // No interrupt or contact-broken is
                                             // detected

#define BIFRESULT_ERR_VENDOR_SPEC (0x4000)   // vendor specific error

#define BIFRESULT_ERR_HW_INIT     (0x8000)   // HW initialization error
#define BIFRESULT_ERR_HW_FAIL     (0x8001)   // HW failure
#define BIFRESULT_ERR_HW_BUSY     (0x8002)   // HW busy
#define BIFRESULT_ERR_SLAVE_NO_RESP (0x8003) // Slave has not responded to a read in
                                             // time
#define BIFRESULT_ERR_SLAVE_NACK  (0x8004)   // Slave reported an error at read (ACK is
                                             // 0)
#define BIFRESULT_ERR_SLAVE_EOT   (0x8005)   // Slave terminated transmission earlier
                                             // than burst length requested
#define BIFRESULT_ERR_SLAVE_SIG   (0x8006)   // A signalling error is seen during Slave
                                             // transmission (parity, inversion or BCF)
#define BIFRESULT_ERR_NOT_SUPPORTED (0x8007) // The called function is not implemented
#define BIFRESULT_ERR_OUT_OF_RANGE (0x8008)  // Function parameter is out of range

typedef BIFuint16 BIFresult;      // Teach about success/error of function call.


//...................................................................................
// BIFslaveError
//...................................................................................
#define BIFSLAVEERROR_NO_ERROR    (0x00)     // no error
#define BIFSLAVEERROR_GEN_COM     (0x10)     // general communication error
#define BIFSLAVEERROR_PARITY      (0x11)     // parity error
#define BIFSLAVEERROR_INV         (0x12)     // Inversion error
#define BIFSLAVEERROR_LENGTH      (0x13)     // Invalid word length error
#define BIFSLAVEERROR_TIMING      (0x14)     // Timing error
#define BIFSLAVEERROR_UNKWN_CMD   (0x15)     // unknown command error
#define BIFSLAVEERROR_SEQ         (0x16)     // Wrong sequence error
#define BIFSLAVEERROR_BUS_COL     (0x1F)     // Bus collision error
#define BIFSLAVEERROR_BUSY        (0x20)     // Slave busy error
#define BIFSLAVEERROR_FATAL       (0x7F)     // Fatal error

#define BIFSLAVEERROR_VENDOR_BASE (0x80)     // Slave vendor specific error

typedef BIFuint8          BIFslaveError;     // Error reported from Slave

//...................................................................................
```

```
// Signaling layer capabilities
//...................................................................................
typedef struct
{
    BIFbool    LowPowerTauBifSupport;        // is BIF_TRUE if this mode is
                                             // supported else BIF_FALSE
    BIFuint8   OptimizedReadBurstLength;     // Teach indirectly about the read HW
                                             // FIFO length of BIF Master. Making
                                             // read operation length shorter than
                                             // this value ensures read operation
                                             // to be completed in a single read
                                             // burst.
    BIFuint16  ManufacturerId;               // Manufacturer identifier of BIF
                                             // Master solution.
    BIFuint16  DeviceId;                     // Device identifier of BIF Master
                                             // solution.
    BIFuint16  ContactBreakDetTime;          // Informs about detection time
                                             // guaranteed by the BIF Master
                                             // contact break sensor.
                                             // Time is expressed in 100ns unit
                                             // (for example, 134 means 13.4µs
                                             // detection time).
                                             // If the Master solution doesn't
                                             // embed contact break detector,
                                             // the reported time is 0.
    BIFbool    NonBlockingInterruptSupported; // is BIF_TRUE if the Master
                                             // implementation support non blocking
                                             // interrupt mode.

} BIFsigCapabilities;

//...................................................................................
// Signaling layer state
//...................................................................................
typedef struct
{
    BIFbool    TauBifNormalMode;             // BIF_FALSE: low power TauBif mode selected,
                                             // BIF_TRUE: normal TauBif selected
    BIFuint16  TauBif;                       // TauBif for the normal mode
                                             // in 100ns unit (for example, 25 means
                                             // 2.5µs)
    BIFuint16  LowPowerTauBif;               // If low power TauBif is not supported,
                                             // value is 0 else value is TauBif for this
                                             // mode in 100ns unit
                                             // (for example, 25 means 2.5µs)
    BIFbool    BclPadState;                  // reports the current BCL pad state
                                             // (BIF_FALSE: low, BIF_TRUE: high)
    BIFuint8   SlaveAddressSelected;         // reports the latest Slave selection
                                             // address (done with SDA command)
    BIFbool    SlaveSelectionCache;          // If BIF_TRUE, Slave selection cache is
                                             // enabled.
                                             // In such case, any SDA command selecting an
                                             // address equal to SlaveAddressSelected is
                                             // ignored. If BIF_FALSE, all SDA commands
                                             // are emitted even if they select again same
                                             // Slave.
} BIFsigState;


//...................................................................................
// Signaling layer reported strings
//...................................................................................

```

```
#define BIF_STRING_MAX_LENGTH     (0x40)     // Max length of string (64 chars
                                             // including null terminating char)
#define BIF_STRING_ID_MANUFACTURER_NAME (0x00) // Manufacturer name
#define BIF_STRING_ID_DEVICE_NAME (0x01)     // Device name

#endif

//----------------------------------------------------------------------------------
//----------------------------------------------------------------------------------
// Signaling layer function declaration
//----------------------------------------------------------------------------------
//----------------------------------------------------------------------------------

//..................................................................................
// Getting implemented version of interface.
// This function informs about the interface version implemented. This should match
// the BIF_HAL_HEADER_VERSION version of this header file.
//..................................................................................
BIFresult bifHALGetVersion(BIFhandle handle, BIFuint16 * version);

//..................................................................................
// Getting information string.
// This function populates the "string" pointer with the corresponding constant
//  "stringID".
// "stringID" is one of defined BIF_STRING_ID_...
// The string reported has length below or equal BIF_STRING_MAX_LENGTH
//..................................................................................
BIFresult bifHALGetString(BIFhandle handle, BIFuint8 stringID, BIFchar * string);

//..................................................................................
// Getting error code reported by Slave.
// If BIFresult is BIFRESULT_ERR_SLAVE_NACK, it means that transaction went well at
// signaling level but the salve was not understanding it correctly. With NACK
// reporting from Slave, an error code is provided which can be retrieved with the
// following function.
//..................................................................................
BIFresult bifHALGetLastError(BIFhandle handle, BIFslaveError * error);

//..................................................................................
// Retrieving the signaling layer capabilities
//..................................................................................
BIFresult bifHALGetCapabilities(BIFhandle handle, BIFsigCapabilities * capabilities);

//..................................................................................
// Signaling layer initialization.
// Calling this function initializes the signaling layer to a default inactive
// state.
//..................................................................................
BIFresult bifHALInit(BIFhandle handle);

//..................................................................................
// Signaling layer enabling.
// Calling this function with "enable" at BIF_FALSE place the signaling layer in
// inactive state (BCL is not driven). If "enable" is BIF_TRUE, the
// signaling layer Masters the BCL.
//..................................................................................
BIFresult bifHALEnabling(BIFhandle handle, BIFbool enable);

//..................................................................................
// TauBif setting.
// This function configures and selects the normal TauBiF timing.
// The "tauBif" parameter is expressed in 100ns unit (2.3µs corresponds to
//  tauBif = 23).
```

```
//
// The effective TauBif configured is reported in field TauBif of BIFsigState.
//.....................................................................................
BIFresult bifHALSetTauBif(BIFhandle handle, BIFuint16 tauBif);

//.....................................................................................
// Set and select TauBif to the most power efficient value supported by the HW.
//
// If LowPowerTauBifSupport is supported in the BIFsigCapabilities, calling the
// bifSigSetLowPowerTauBif configures the HW to run with the best compromise of
// transmission speed vs power consumption.
//
// Typically, bifHALSetLowPowerTauBif configures the HW to use a low power clock tree
// which is generally at low frequency. By consequence, TauBif would be clamped to a
// long value.
//
// The "tauBif" parameter is expressed in 100ns unit (100µs corresponds to
// tauBif = 1000). The effective TauBif for this mode is reported in field
// LowPowerTauBif of BIFsigState.
//.....................................................................................
BIFresult bifHALSetLowPowerTauBif(BIFhandle handle, BIFuint16 tauBif);

//.....................................................................................
// Slave selection cache control.
//
// If enable is BIF_TRUE, Slave selection cache is enabled. In such case, any SDA
// command selecting a Slave which has already been selected with a previous SDA
// command are ignored and not emitted. If enable BIF_FALSE, all SDA commands are
// emitted even if they select again same Slave.
//
// The known current selected Slave and state of selection cache are mentioned inside
//  the BIFsigState structure reported with bifHALGetState function.
//.....................................................................................
BIFresult bifHALSlaveSelectionCache(BIFhandle handle, BIFbool enable);

//.....................................................................................
// Get state of signaling layer.
// The function set-up the data pointed by "state" pointer.
//.....................................................................................
BIFresult bifHALGetState(BIFhandle handle, BIFsigState * state);

//.....................................................................................
// Generic access point to vendor specific signaling layer functionalities
//.....................................................................................
BIFuint16 bifHALVendorSpecific(BIFhandle handle, BIFuint32 command, void * data_in,
    void * data_out);


//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
// Transaction Elements layer function declaration
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------


//.....................................................................................
// Sending command BRES (Bus Reset)
//.....................................................................................
BIFresult bifTrans_BRES(BIFhandle handle);

//.....................................................................................
// Driving BCL low for tPDL (Hard Reset)
//.....................................................................................
```

```
BIFresult bifTrans_HardRST(BIFhandle handle);

//...................................................................................
// Sending command PWDN (Power down)
//...................................................................................
BIFresult bifTrans_PWDN(BIFhandle handle);

//...................................................................................
// Sending command STBY (standby)
//...................................................................................
BIFresult bifTrans_STBY(BIFhandle handle);

//...................................................................................
// Produces an activation signal only
//...................................................................................
BIFresult bifTrans_WakeUpPulse(BIFhandle handle);

//...................................................................................
// Produces an activation signal and wait for tPUP delays in order to go out of
// power-down
//...................................................................................
BIFresult bifTrans_WakeUpFromPWDN(BIFhandle handle);

//...................................................................................
// Produces an activation signal and wait for tACT delays in order to go out of
// standby mode
//...................................................................................
BIFresult bifTrans_WakeUpFromSTBY(BIFhandle handle);

//...................................................................................
// Sending command ISTS (interrupt status).
// The function reports BIF_TRUE if Slave reports interrupt pending else it report
// BIF_FALSE.
//...................................................................................
BIFresult bifTrans_ISTS(BIFhandle handle, BIFbool * intStatus);

//...................................................................................
// Sending command RBL (read burst length) with the command parameter
//...................................................................................
BIFresult bifTrans_RBL(BIFhandle handle, BIFuint8 length);

//...................................................................................
// Sending command RBE (read burst extended) with the command parameter
//...................................................................................
BIFresult bifTrans_RBE(BIFhandle handle, BIFuint8 length);

//...................................................................................
// Sending command DASM (device multiple select)
//...................................................................................
BIFresult bifTrans_DASM(BIFhandle handle);

//...................................................................................
// Sending command DISS
//...................................................................................
BIFresult bifTrans_DISS(BIFhandle handle);

//...................................................................................
// Sending command DILC.
// Report the DILC result.
//...................................................................................
BIFresult bifTrans_DILC(BIFhandle handle, BIFbool * dilc);

//...................................................................................
```

```
// Sending command DIE0
//...................................................................................
BIFresult bifTrans_DIE0(BIFhandle handle);

//...................................................................................
// Sending command DIE1
//...................................................................................
BIFresult bifTrans_DIE1(BIFhandle handle);

//...................................................................................
// Sending command DIP0.
// Reports BIF_TRUE if a Slave responds to probing else BIF_FALSE.
//...................................................................................
BIFresult bifTrans_DIP0(BIFhandle handle, BIFbool * dip0);

//...................................................................................
// Sending command DIP1.
// Reports BIF_TRUE if a Slave responds to probing else BIF_FALSE.
//...................................................................................
BIFresult bifTrans_DIP1(BIFhandle handle, BIFbool * dip1);

//...................................................................................
// Sending command DRES (device reset)
//...................................................................................
BIFresult bifTrans_DRES(BIFhandle handle);

//...................................................................................
// Sending command TQ (Transaction query).
// If no error, reports transaction "dataCount" value ("errorCode" will be
// BIFSLAVEERROR_NO_ERROR).
// If error, returns "errorCode" from Slave ("dataCount" will be 0).
//...................................................................................
BIFresult bifTrans_TQ(BIFhandle handle, BIFslaveError * errorCode,
    BIFuint8 * dataCount);

//...................................................................................
// Sending command AIO
//...................................................................................
BIFresult bifTrans_AIO(BIFhandle handle);

//...................................................................................
// Sending command EDA.
// Function takes the EDA command parameter.
//...................................................................................
BIFresult bifTrans_EDA(BIFhandle handle, BIFuint8 add);

//...................................................................................
// Sending command SDA (Slave select by address).
// Function takes the Slave address as parameter.
//...................................................................................
BIFresult bifTrans_SDA(BIFhandle handle, BIFuint8 add);

//...................................................................................
// Sending command ERA.
// Function takes the command parameter.
//...................................................................................
BIFresult bifTrans_ERA(BIFhandle handle, BIFuint8 add);

//...................................................................................
// Sending command WRA.
// Function takes the command parameter.
//...................................................................................
BIFresult bifTrans_WRA(BIFhandle handle, BIFuint8 add);
```

```

//...................................................................................
// Sending command WD.
// Function takes the command parameter.
//...................................................................................
BIFresult bifTrans_WD(BIFhandle handle, BIFuint8 data);

//...................................................................................
// Sending command RRA.
// read data are placed in data array with specified length
//
// * It is not recommended to use RRA directly as data count transmitted by Slave*
// * depends directly on previous command sent to it. Prefer using high-level read*
// * functions instead.           *
//
//...................................................................................
BIFresult bifTrans_RRA(BIFhandle handle, BIFuint8 add, BIFuint8 * data,
    BIFuint8 length);

//...................................................................................
// Sending command EINT (enable interrupt mode) and wait for completion.
//
// Completion can be:
// - interrupt signalled by Slave (function returns BIFRESULT_NO_ERROR)
// - time-out (function returns BIFRESULT_TIME_OUT)
// - abort (function returns BIFRESULT_ABORTED)
// - BCL contact broken if such detection is supported by Master (function returns
//  BIFRESULT_CONTACT_BROKEN)
//
// Function takes timeOut data as parameter. timeOut is expressed in ms. If timeout
// ellapses before interrupt signalled by Slave, the function returns
// BIFRESULT_TIME_OUT as BIFresult.
//
// !! In case of time-out, the function aborts the interrupt mode on BIF bus so that
// !! any command can be send to the Slave
//
// The interrupt mode can be aborted by the BifTrans_Abort_EINT function called from
// another thread. In such case, the function returns BIFRESULT_ABORTED.
//
// If timeOut is 0, the function waits forever (infinite time-out)
//...................................................................................
BIFresult bifTrans_EINT(BIFhandle handle, BIFuint32 timeOut);

//..................................................................................
// Sending command EINT (enable interrupt mode) and don't wait for completion.
//
// !! this function is effectively implemented only if non-blocking interrupt mode
// !! is supported by the Master. This support is reported in the field
// !! NonBlockingInterruptSupported of BIFsigCapabilities structure after calling
// !! bifHALGetCapabilities function.
//
// After sending the EINT command to Slaves, the function engages BCL monitoring on a //
background process (HW cell commonly) and returns. The BIF bus is no more available // for
any transactions. It's up to function caller to manage resuming from interrupt
// mode.
//
// The function returns BIFRESULT_NO_ERROR if background process is correctly started. //
Else, it returns appropriate error information.
//
// The interrupt survey engaged on background process is resumed when:
// - the bifTrans_Abort_EINT function is called.
// - Slaves produce interrupt.
//
```

```
// A BCL contact break detection (if such detection is supported by the Master)
// doesn't stop the interrupt survey background process. A call to bifTrans_Abort_EINT
// function is required.
//
// By using bifTrans_Query_EINT function, it is possible to get the state of the
// background interrupt survey process.
//....................................................................................
BIFresult bifTrans_Async_EINT(BIFhandle handle);

//....................................................................................
// Probe the background interrupt survey process state.
//
// !! this function is effectively implemented only if non-blocking interrupt mode
// !! is supported by the Master. This support is reported in the field
// !! NonBlockingInterruptSupported of BIFsigCapabilities structure after calling
// !! bifHALGetCapabilities function.
//
// When interrupt survey background process is engaged after a sucessfull call of
// bifTrans_Async_EINT, this function returns the following:
// - BIFRESULT_NO_ERROR if a slave interrupt has been detected.
// - BIFRESULT_CONTACT_BROKEN if a BCL contact break has been detected (if supported
// by Master).
// - BIFRESULT_BG_RUNNING if background process is running and has not detected
// anything.
//....................................................................................
BIFresult bifTrans_Query_EINT(BIFhandle handle);

//....................................................................................
// Abort the interrupt mode engaged by bifTrans_EINT command or bifTrans_EINT_Engage
//
// This function emits the interrupt mode abort pulse on the BIF bus.
//....................................................................................
BIFresult bifTrans_Abort_EINT(BIFhandle handle);

//....................................................................................
// Sending Slave vendor specific command with commandNibble code (from 0x00 to 0x0F)
//....................................................................................
BIFresult bifTrans_SlaveVendorSpecific(BIFhandle handle, BIFuint8 commandNibble);


//------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------
// Transaction function declaration
//------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------

//....................................................................................
// Writing a 8bits "data" at a 16bits "address" into the Slave.
// After write done, a TQ command is sent and its result is reported.
//....................................................................................
BIFresult bifTransWriteUint8(BIFhandle handle, BIFuint16 address, BIFuint8 data);

//...................................................................................
// Writing a 16bits "data" at a 16bits "address" into the Slave.
// After write done, a TQ command is sent and its result is reported.
//....................................................................................
BIFresult bifTransWriteUint16(BIFhandle handle, BIFuint16 address, BIFuint16 data);

//....................................................................................
// Read BIFuint8 "data" at a 16 bits "address" into the Slave.
//....................................................................................
BIFresult bifTransReadUint8(BIFhandle handle, BIFuint16 address, BIFuint8 * data);
```

```
//...................................................................................
// Read BIFuint16 "data" at a 16 bits "address" into the Slave.
//...................................................................................
BIFresult bifTransReadUint16(BIFhandle handle, BIFuint16 address, BIFuint16 * data);

//...................................................................................
// Writing multiple 8bits "data" from a 16bits "address" into the Slave.
// After write done, a TQ command is sent and its result is reported.
//...................................................................................
BIFresult bifTransWriteMultUint8(BIFhandle handle, BIFuint16 address,
    BIFuint8 * data, BIFuint8 size);

//...................................................................................
// Reading multiple 8bits "data" from a 16bits "address" into the Slave.
//...................................................................................
BIFresult bifTransReadMultUint8(BIFhandle handle, BIFuint16 address, BIFuint8 * data,
    BIFuint16 size);

//...................................................................................
// Select a Slave by its " UID " passed in parameter as an array of 8bits data
// (10 bytes).
//...................................................................................
BIFresult bifTransSelUid(BIFhandle handle, BIFuint8 * uid);



#endif
```

Annex B Examples Source Code

B.1 Examples of Using HAL

B.1.1 Basic Setup of Transaction Layer

```
bool bifRawSetUpTransactionLayer(BIFhandle handle)
{
    // HAL implemented version
    BIFuint16 HALVersion;
    // state structure
    BIFsigState layerState;

    // Get HAL implemented version
    if (bifHALGetVersion(handle, &HALVersion) != BIFRESULT_NO_ERROR) return false;

    // Check HAL version compatibility
    if (HALVersion != BIF_HAL_HEADER_VERSION) return false;

    // initialize HAL layer
    if (bifHALInit(handle) != BIFRESULT_NO_ERROR) return false;

    // activate HAL layer
    if (bifHALEnabling(handle, BIF_TRUE) != BIFRESULT_NO_ERROR) return false;

    // set-up tauBIF to 3µs
    if (bifHALSetTauBif(handle, 30) != BIFRESULT_NO_ERROR) return false;

    // get transaction layer state
    if (bifSigGetState(handle, &layerState) != BIFRESULT_NO_ERROR) return false;

    // reset the BIF bus
    if (bifTrans_BRES(handle) != BIFRESULT_NO_ERROR) return false;

    return true; // BIF BUS ready to be used
}

```

B.1.2 Vendor Specific Access

```
bool bifVendorSpecificAccess(BIFhandle handle)
{
    // example of vendor specific command data
    BIFuint16      dataIn[4] = {0, 1, 2, 3};
    BIFuint16      dataOut[8];
    BIFuint16      returnedValue;

    // string instantiation
    BIFchar        vendorName[BIF_STRING_MAX_LENGTH];
    BIFchar        deviceName[BIF_STRING_MAX_LENGTH];

    // string initialization
    vendorName[0] = 0;
    deviceName[0] = 0;

    // get strings
    if (bifHALGetString(handle, BIF_STRING_ID_MANUFACTURER_NAME, vendorName) !=
               BIFRESULT_NO_ERROR) return false;
    if (bifSigGetString(handle, BIF_STRING_ID_DEVICE_NAME, deviceName) !=
               BIFRESULT_NO_ERROR) return false;

    // check strings in order to use the vendor specific commands only with
    // the good hardware
    if (strcmp((char*)vendorName, "MyVendorName") == 0 &&
        strcmp((char*)deviceName, "MyDeviceName") == 0)
    {
        // call vendor command 0 which doesn't take dataIn neither dataOut
        returnedValue = bifHALVendorSpecific(handle, /*command*/ 0, (void*)0, (void*)0);

        // Do treatment linked to command 0
        // ...
        // ...

        // call vendor command 4 which takes dataIn & dataOut
        returnedValue = bifHALVendorSpecific(handle, /*command*/ 4,
                            (void*)dataIn,(void*)dataOut);

        // Do treatment linked to command 4
        // ...
        // ...
    }
    return true;
}

```

B.1.3 Low Power Interrupt Mode

```
bool bifLowPowerInterruptMode(BIFhandle handle)
{
    // Capabilities of hardware
    BIFsigCapabilities capabilities;
    // state structure
    BIFsigState     layerState;

    // Get capabilities
    if (bifHALGetCapabilities(handle, &capabilities) != BIFRESULT_NO_ERROR) return false;

    // Engage low power TauBIF if possible
    if (capabilities.LowPowerTauBifSupport == BIF_TRUE)
    {
        // Get the signaling state to memorize the current tauBif indirectly
        if (bifHALGetState(handle, &layerState) != BIFRESULT_NO_ERROR) return false;

        // set low power TauBif (150µs target) if not already engaged
        if (layerState.TauBifNormalMode == BIF_TRUE)
        {
            if (bifHALSetLowPowerTauBif(handle,1500) != BIFRESULT_NO_ERROR)
                return false;
        }
    }

    // Engage BIF interrupt mode without time-out.
    // Function below is blocking while no interrupt happens from a Slave.
    if (bifTrans_EINT(handle, 0xFFFF) != BIFRESULT_NO_ERROR) return false;

    // Resume normal tauBIF mode if possible and if engaged before
    if (capabilities.LowPowerTauBifSupport == BIF_TRUE &&
        layerState.TauBifNormalMode == BIF_TRUE)
    {
        // set normal TauBif
        if (bifHALSetTauBif(handle, layerState.TauBif) != BIFRESULT_NO_ERROR)
            return false;
    }

    return true;
}

```

B.1.4 Selecting a Slave

```
bool bifSelectingDevice(BIFhandle handle, BIFuint8 logicAddress, BIFuint8 * Uid)
{
    // if no logic address known, select by UID
    if (logicAddress == 0)
    {
        if (bifTransSelUid(handle, Uid) != BIFRESULT_NO_ERROR) return false;
    }
    // else select by logic address
    else
    {
        if (bifTrans_SDA(handle, logicAddress) != BIFRESULT_NO_ERROR) return false;
    }
    return true; // device selected
}
```

B.1.5 Read DDBL1

```
bool bifFetchDbbL1(BIFhandle handle)
{
    // ddbL1 structure
    struct {
        BIFuint8            revision;
        BIFuint8            level;
        BIFuint16           deviceClass;
        BIFuint16           manufacturerId;
        BIFuint16           productId;
    } ddbL1;

    // read Slave BIF revision
    if (bifTransReadUint8(handle, 0x00, &(ddbL1.revision)) != BIFRESULT_NO_ERROR)
        return false;
    // read Slave BIF level
    if (bifTransReadUint8(handle, 0x01, &(ddbL1.level)) != BIFRESULT_NO_ERROR)
        return false;
    // read Slave device class
    if (bifTransReadUint16(handle, 0x02, &(ddbL1.deviceClass)) != BIFRESULT_NO_ERROR)
        return false;
    // read Manufacturer ID
    if (bifTransReadUint16(handle, 0x04, &(ddbL1.manufacturerId)) != BIFRESULT_NO_ERROR)
        return false;
    // read Product ID
    if (bifTransReadUint16(handle, 0x06, &(ddbL1.productId)) != BIFRESULT_NO_ERROR)
        return false;
    return true;
}
```

B.1.6 Output Error Code to Console

```
BIFresult bifErrorDisplay(BIFhandle handle, BIFresult returnedResult)
{
    // Slave error
    BIFslaveError slaveError;
    switch (returnedResult)
    {
    case BIFRESULT_NO_ERROR :
        // no error, no printf
        break;
    case BIFRESULT_ERR_HW_INIT :
        printf("ERROR : Master not initialized");
        break;
    case BIFRESULT_ERR_HW_FAIL :
        printf("ERROR : Master failure");
        break;
    case BIFRESULT_ERR_HW_BUSY :
        printf("ERROR : Master is busy");
        break;
    case BIFRESULT_ERR_SLAVE_NO_RESP :
        printf("ERROR : slave didn't respond in time");
        break;
    case BIFRESULT_ERR_SLAVE_NACK :
        printf("ERROR : slave didn't ACK the command");
        // get Slave error
        bifSigGetLastError(handle, &slaveError);
        switch (slaveError)
        {
        case BIFSLAVEERROR_GEN_COM :
            printf("  -> general communication error");
            break;
        case BIFSLAVEERROR_PARITY :
            printf("  -> parity error");
            break;
        case BIFSLAVEERROR_INV :
            printf("  -> inversion error");
            break;
        case BIFSLAVEERROR_LENGTH :
            printf("  -> invalid length");
            break;
        case BIFSLAVEERROR_TIMING :
            printf("  -> timing out of limits");
            break;
        case BIFSLAVEERROR_UNKWN_CMD :
            printf("  -> unknown command");
            break;
        case BIFSLAVEERROR_SEQ :
            printf("  -> sequence of commands invalid");
            break;
        case BIFSLAVEERROR_BUS_COL :
            printf("  -> bus collision");
            break;
        case BIFSLAVEERROR_BUSY :
            printf("  -> slave is busy");
            break;
        case BIFSLAVEERROR_FATAL :
            printf("  -> slave fatal error");
            break;
        default:
            if (slaveError >= BIFSLAVEERROR_VENDOR_BASE)
                printf("  -> vendor specific error");
            else
                printf("  -> invalid error code");
            break;
        }
        break;
    case BIFRESULT_ERR_SLAVE_EOT :
        printf("ERROR : early termination of read burst");
        break;
    case BIFRESULT_ERR_SLAVE_SIG :
        printf("ERROR : Slave signaling error");
        break;
    case BIFRESULT_TIME_OUT :
        printf("Time-out");
        break;
    case BIFRESULT_ABORTED :
        printf("Command aborted");
        break;
    case BIFRESULT_CONTACT_BROKEN :
        printf("Contact broken during command");
        break;
    case BIFRESULT_ERR_VENDOR_SPEC :
        printf("ERROR : Vendor specific");
        break;
    }
    return returnedResult;
}

```

B.1.7 UID Device Search

```
typedef struct BIFuid_ {
    BIFuint8        u[10];
} BIFuid;

#define UID_LENGTH 80
/**
 * Search for a Slave UID.
 * If there are multiple Slaves on the bus, the UID with the least numeric value is chosen.
 * Returns BIF_TRUE if UID was found, BIF_FALSE otherwise.
 * Error handling omitted.
 */
BIFbool bifSearchForDevice (BIFhandle handle, BIFuid * uid)
{
BIFuint8 index=0;             // Current UID bit index.
    BIFuint8        bit;       // Current bit index in byte.
    BIFuint8        pos;       // Current byte index.
    BIFuint8        val;       // Current bit value
    BIFbool         res;       // Result of DIP0, DIP1 and DILC.

     // Clear the uid buffer.
    for (pos=0; pos<10; pos++) {
        uid->u[pos]=0;
    }

    // Reset the bus.
    bifTrans_BRES(handle);

    // Start UID search.
    bifTrans_DISS(handle);

    // UID_LENGTH = 80
    for (index = 0; index < UID_LENGTH; index++) {
        bit = index%8;
        pos = index/8;

        // Check if current bit is 0.
        bifTrans_DIP0(handle, &res);

        if (res) {
            // Current bit is 0.
            val = 0;
        } else {
            // Not 0, check if 1.
            bifTrans_DIP1(handle, &res);
            if (res) {
                // Current bit is 1.
                val = 1;
            } else {
                // Error, no Slave responded to either bit value.
                return BIF_FALSE;
            }
        }
        // Set the current bit value to the UID buffer.
        uid->u[pos] |= (val << bit);

        // Move to the next bit.
        if (val == 0) {
            bifTrans_DIE0(handle);
        } else {
            bifTrans_DIE1(handle);
        }
    } // for index

    // All done. Check that Slave agrees.
    bifTrans_DILC(handle, &res);

    if (!res) {
        // Slave didn't respond to DILC although we're done with all of the bits.
        return BIF_FALSE;
    }
    return BIF_TRUE;
}
```

Participants

The following list includes those persons who participated in the Working Group that developed this Specification and who consented to appear on this list.

Bell, Stewart – Panasonic Corporation

Chiang, Sie Boo – Infineon Technologies AG

Chun, Christopher – Qualcomm Incorporated

Cimaz, Lionel – ST-Ericsson

Furtner, Wolfgang – Infineon Technologies AG

Leinonen, Pekka E. – Nokia Corporation

Littow, Markus – ST-Ericsson

Rajala, Jarno – Nokia Corporation

Schaecher, Stephan – Infineon Technologies AG

Sunyi, Imre – Sony Mobile Communications

Tang, Yiming – Infineon Technologies AG

Waldstein, Steve – Fairchild Semiconductor Int’l