

A2B STACK USER GUIDE

Document Status	Approved
Approved by	Siva S

ANALOG DEVICES, INC.

www.analog.com

Revision List

Table 1: Revision List

Revision	Date	Description
0.1	29-Sep-2016	Draft Version – EVAL Release of A2B Stack
0.2	12-Oct-2016	Incorporated Review comments and elaborated relevant sections.
1.0	21-Oct-2016	Approved for 13.0.0_EVAL
1.1	08-Nov-2016	Draft Version of 13.0.0
1.2	10-Nov-2016	Incorporated Review comments
2.0	10-Nov-2016	Approved and baselined for Rel 13.0.0
2.1	30-Nov-2016	Updated Diagnostics and debugging for sample demo – Rel 13.1.0
2.2	02-Dec-2016	Incorporated review comments
3.0	09-Dec-2016	Approved and baselined for Rel 13.1.0
3.1	17-Jan-2017	Updated document to include BERT, disabling power diagnostics for Rel 14.0.0
3.2	18-Jan-2017	Addressed review comments
4.0	20-Jan-2017	Approved and baselined for Rel 14.0.0
4.1	23-Feb-2017	Minor changes in section 5.2.1.2.1 and 4.1.1
5.0	03-Mar-2017	Baselined for Rel15.0.0
5.1	30-Apr-2018	Document re-written
5.2	09-May-2018	Review comments incorporated
6.0	06-Jun-2018	Baselined for Rel19.0.0
6.1	24-Oct-2018	Updated for release 19.1.0
7.0	31-Oct-2018	Baselined for Rel19.1.0
7.1	31-Aug-20	Updated for release 19.4.0
8.0	02-Sep-20	Baselined for Rel19.4.0

Copyright, Disclaimer Statements

Copyright Information

Copyright (c) 2009-2020 Analog Devices, Inc. All Rights Reserved. This software is proprietary and confidential to Analog Devices, Inc. and its licensors. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Table of Contents

Revision List.....	2
Copyright, Disclaimer Statements	3
Table of Contents.....	4
List of Figures	6
List of Tables.....	6
List of Code Snippets	6
1 Introduction.....	8
1.1 Scope	8
1.2 Organization of the Guide	8
2 A2B Software Stack	9
2.1 Host-Target Software Workflow	10
3 Building an A2B Application on a custom platform	12
3.1 Designing A2B schematic on SigmaStudio	12
3.2 Building Target software for a custom platform	13
3.2.1 Porting A2B Software Stack to a custom platform	15
3.2.2 Apply A2B Network configuration	19
3.2.3 Modify Application call back functions	20
3.2.3.1 Discovery completion Callback function	20
3.2.3.2 Power/Line Fault Callback function	22
3.2.3.3 Interrupt Callback function.....	25
3.2.3.4 Node Discovery Callback function	26
3.3 Summary of Building A2B Application on custom platform	28
4 Application Integration	29
4.1 Stack States	32
4.1.1 Initialize/Allocate	32
4.1.2 Load.....	32
4.1.3 Start.....	32
4.1.4 Discover	32
4.1.5 Interrupt Poll	32
4.1.6 Stop	33
4.1.7 Free	33
4.2 Application Extensions to Environment control block	34

4.3 Plugin Architecture.....	35
4.3.1 Plugin Examples	35
4.3.1.1 Master Plugin	35
4.3.1.2 Generic Slave Plugin.....	35
4.3.2 Handling Interrupts in a Plugin	35
4.3.3 Writing a Custom Plugin.....	36
4.3.4 Loading Plugins into the Stack.....	38
4.4 Using A2B Stack for Multi-Master Network	38
4.5 Inter-Processor Communication over A2B Mailbox	39
4.5.1 Pre-Requisites for Inter-Processor Communication.....	40
4.6 Post discovery APIs.....	43
5 Appendix A: Diagnostics and Debugging	44
5.1 Generating Sequence Diagrams.....	44
5.1.1 Sequence diagram support in the Stack.....	44
5.1.2 Enabling Sequence Chart in Sample Demo Applications	45
5.2 Capturing Trace Messages.....	47
5.2.1 Trace support in the Stack	47
5.2.2 Enabling Trace in Sample Demo Applications.....	48
5.3 Stack scalability and optimization options	49
6 Appendix B: Messages	50
6.1 Request Message.....	50
6.2 Notify Message.....	51
6.3 Sending custom messages and notifications	52
6.4 Receiving custom messages and notifications	53
7 Appendix C: Target Debug Features	54
7.1 Bit Error Rate Test (BERT)	54
8 Appendix D: Auto Configuration from EEPROM.....	55
Terminology	56
References.....	56

List of Figures

Figure 1: A2B Software Stack Architecture.....	9
Figure 2: Host Tool – Target Software Workflow	11
Figure 3: A2B Target Software Examples.....	13
Figure 4: A2B Target Project directory structure	14
Figure 5: a2bapp_onDiscoveryComplete callback registration.....	21
Figure 6: a2bapp_onPowerFault callback registration	23
Figure 7: a2bapp_onInterrupt callback registration	25
Figure 8: a2bapp_onNodeDiscovery callback registration	26
Figure 9: Building A2B Application on a Custom Platform	28
Figure 10: Message exchange between Master and Slave node processor	39
Figure 11: A2B Target project directory structure with communication channel includes	41
Figure 12: A2B Target project directory structure with communication channel sources.....	42
Figure 13: Setting Path in Environment Variables.....	46
Figure 14: Sample Sequence Chart.....	47
Figure 15: Request Message Example.....	50
Figure 16: Notify Message Example.....	51

List of Tables

Table 1: Revision List	2
Table 2: Target Example Projects	13
Table 3: PAL Functions to be Re-implemented	16
Table 4: ECB components.....	34
Table 5: Plugin Functions	36
Table 6: Supported post discovery APIs.....	43
Table 7: PAL Logging Functions – Info.....	45
Table 8: Trace Levels Description	48
Table 9: Trace Domains Description.....	48
Table 10: Request Message Commands.....	50
Table 11: Notify Message Commands.....	52
Table 12: Terminology.....	56
Table 13: References	56

List of Code Snippets

Code Snippet 1: a2bapp_onDiscoveryComplete sample implementation	22
Code Snippet 2: a2bapp_onPowerFault sample implementation.....	24
Code Snippet 3: a2bapp_onInterrupt sample implementation	25

Code Snippet 4: a2bapp_onNodeDiscovery sample implementation.....	27
Code Snippet 5: Wrapper Services Layer 1 usage	30
Code Snippet 6: a2b_pluginInterrupt dummy implementation.....	36
Code Snippet 7: Custom PluginInit implementation	37
Code Snippet 11: Sending Custom Message Example.....	53
Code Snippet 12: Receiving Custom Message Example	53

1 Introduction

This document guides the user in porting A2B Stack on to custom platforms. The document provides details of stack layers which needs to be reused and the layers which needs to be re-implemented when porting to a custom platform. The document also provides code snippets to enable user in building an A2B application on a custom platform using the stack.

A2B Stack is a highly portable and flexible framework for developing and deploying A2B networks in automotive environments. The Stack embodies the following set of features:

- Full power and line fault diagnostics
- Extensive logging and debug capability
- Modular plugin architecture
- Well defined Platform Abstraction Layer (PAL)
- Message based command and control
- Extensible command set

1.1 Scope

The scope of this document is to briefly illustrate the steps to integrate the A2B Stack (Core A2B Network stack referred in Figure 1) to any platform and build an A2B application using exported '*Bus configuration files*' from SigmaStudio.

Refer to [2] for the steps to draw customized A2B schematics using SigmaStudio tool.

Refer to [3] for the detailed API documentation for A2B Stack.

1.2 Organization of the Guide

Section 1 : Introduction to A2B Stack and the organization of the document.

Section 2 : A2B Software Stack architecture and workflow with Target software.

Section 3 : Building an A2B Application on a custom platform.

Section 4 : Application Integration.

2 A2B Software Stack

A2B Software Stack or simply referred as A2B Stack is a collection of functional blocks designed to efficiently configure, troubleshoot, and deploy A2B networks. Figure 1 shows the architecture of the stack software.

A2B Stack is platform agnostic. The functionalities of the core stack remain same irrespective of the target platform. The stack can be used to build applications specific to any platform by re-implementing the Platform abstraction layer (PAL) as per the targeted platform.

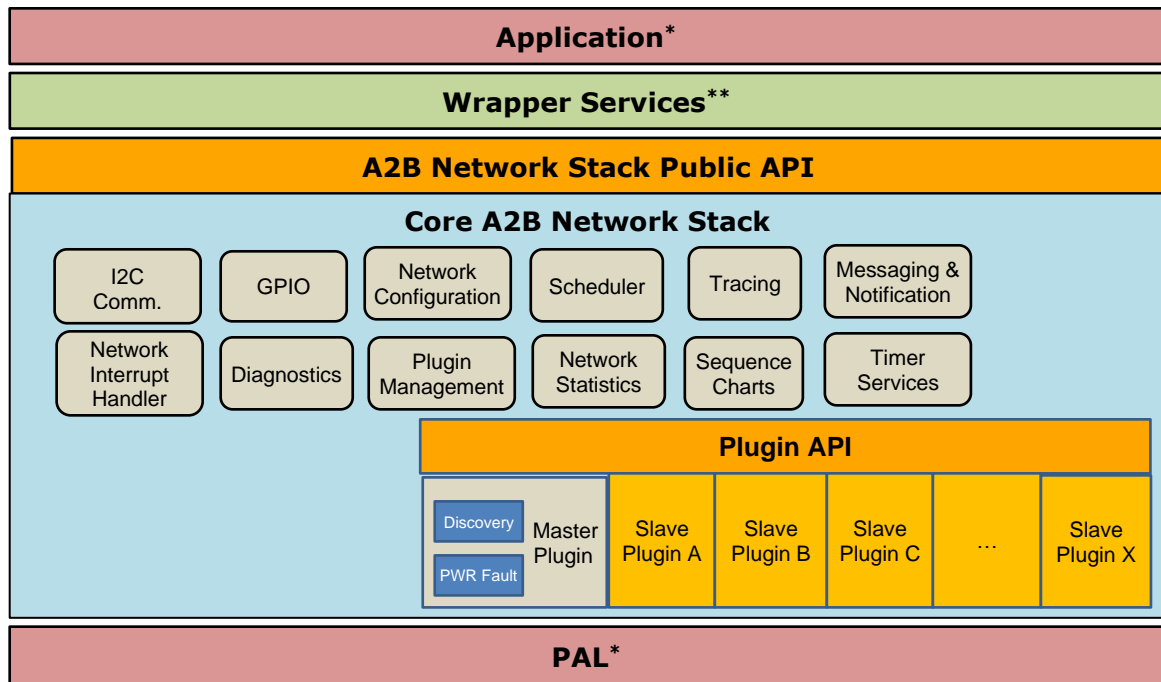


Figure 1: A2B Software Stack Architecture

* PAL and Application layers need to be re-implemented depending on the target platform and the end application requirements.

** Wrapper Services Layer helps in easy stack integration. It is integrator's choice to use it as-is or can be re-implemented. This layer can be further visualized as Wrapper Services Layer 1 (top-most layer) and Wrapper Services Layer 2 (layer just above the A2B stack and below Wrapper Services Layer 1).

The Stack is comprised of the following subsystems.

1. Scheduler

- The scheduler is designed to efficiently coordinate network activities, especially during the discovery and configuration phase, and execute units of work encapsulated in messages and jobs.

2. Plugin Management

- Plugin Management initializes and integrates a single master plugin and zero or more slave plugins into the Stack.

3. Diagnostics

- The diagnostic APIs provide a uniform means for slave plugins, and the Stack itself, to transfer diagnostic information to the Application software.

4. Logging/Tracing

- The logging and tracing subsystems provide a uniform way for plugins, and the Stack itself, to log interesting events throughout the network lifecycle.

5. Bus Configuration Parsing

- The bus configuration parsing subsystem, which is external to the Stack, is responsible for parsing the output (BCF/BDD) of the Host Tool like SigmaStudio.

6. Slave I2C communication

- The slave I2C communication subsystem provides a more direct interface for plugins to communicate with slave devices. Depending on the role of software entity (e.g. master plugin, slave plugin, or application) protective limits are placed on device access over I2C to minimize the potential of I2C issues.

7. Logging/Tracing

- Optional Trace Support
 - Trace domains (e.g., I2C, Plugin, Stack) and severity levels
- Optional Sequence Chart Generation
 - Utilize PlantUML for Presentation
 - Preprocessing support for enhanced charts
- Stack APIs
 - Register dumps, BER counts, node GPIO manipulation

2.1 Host-Target Software Workflow

Target software is a framework which hosts A2B Stack, Application and other components specific to the targeted processor platform.

A2B stack, running on the Target Software framework, requires a bus configuration file produced by a Host network design tool like SigmaStudio to configure an A2B network. Refer [2] for more details on drawing customized A2B schematics using SigmaStudio.

Once an A2B schematic, corresponding to the Target system, is modelled and validated in SigmaStudio, the schematic information is then exported (`adi_a2b_busconfig.c` file) and added to the Target software project. The Stack running on the Target parses the information in this file and programs the A2B network accordingly.

The Host-Target software workflow in building an A2B application is as shown in Figure 2.

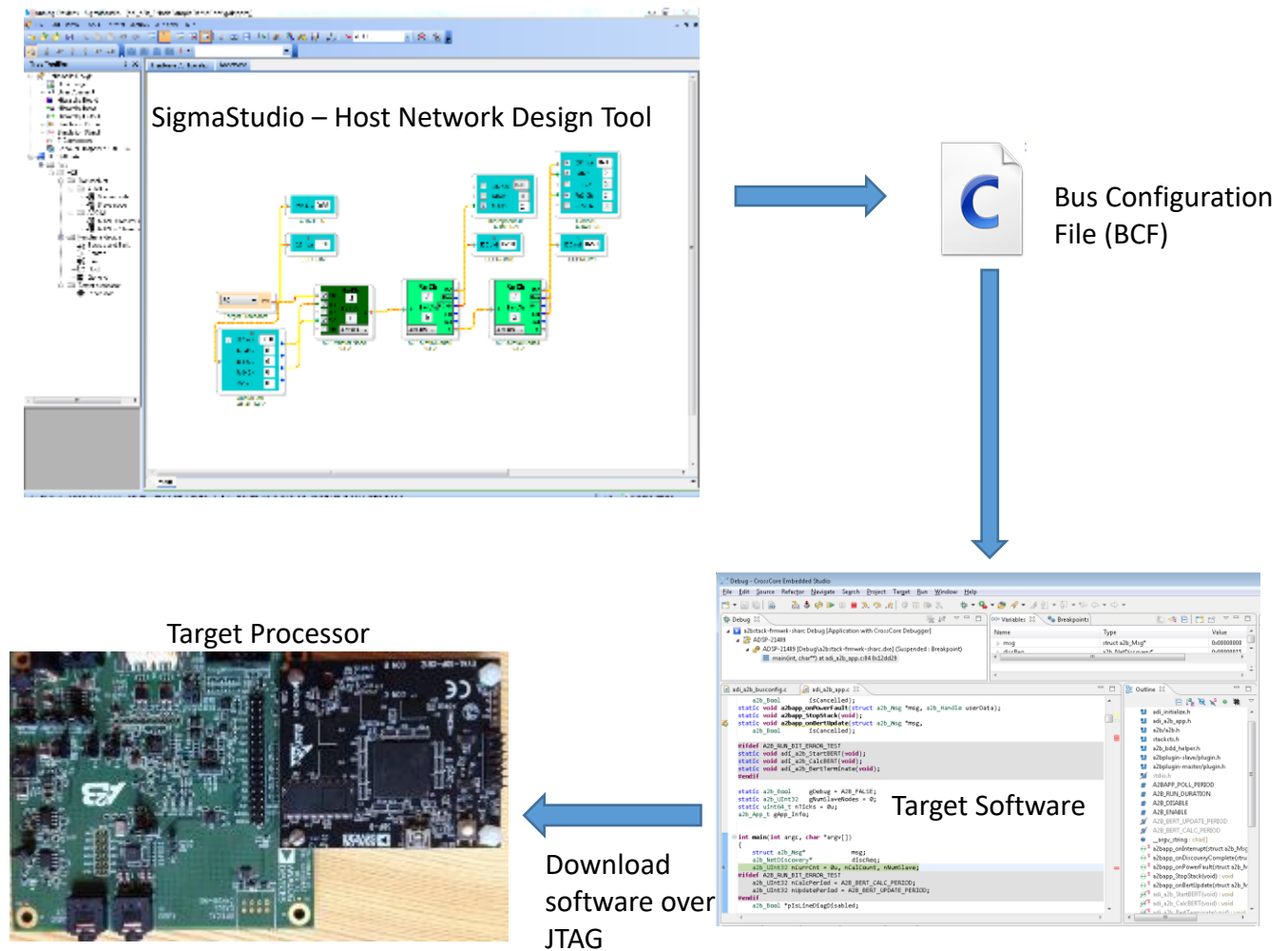


Figure 2: Host Tool – Target Software Workflow

3 Building an A2B Application on a custom platform

Building an A2B application on a custom platform involves two major steps

1. **Step-1 (Concept):** Designing A2B schematic on SigmaStudio.
2. **Step-2 (System Implementation):** Building Target software for the custom platform.

3.1 Designing A2B schematic on SigmaStudio

This step is required to create a bus configuration file that stores the complete A2B network information required by the Target software running on the custom platform. An A2B schematic, corresponding to the Targeted application shall be designed and validated on SigmaStudio before exporting the bus configuration file. The steps involved in this process are as follows

1. Build an A2B Schematic on SigmaStudio matching your final A2B system
 - Refer [2] for drawing an A2B schematic on SigmaStudio. Ensure
 - Audio streams are defined and assigned for the network.
 - Configuration is provided for all A2B nodes and connected peripheral devices.
2. Validate A2B Schematic using PC as the host processor
 - Refer to the example in Section 5 of 1 to discover A2B network using PC as Host
 - Link-compile-download and confirm successful network discovery, configuration and audio routing.
 - Debug discovery issues (if any) using 'Tracing', 'Sequence Chart' and other features.
3. Perform Network analysis to ensure the drawn schematic matches requirements of the end system
 - Check for Bandwidth usage per Node/Network.
 - Run Bit error Test for the network.
 - Check Power usage for the network.
4. Define application response to line faults (if required)
 - Auto-rediscovery upon faults, no. of attempts etc.
 - Verify line fault handling/rediscovery upon line faults.
5. After successful validation, export **Bus Configuration .C file** for the validated A2B schematic.
6. Bus configuration file can also be exported as a binary file using "**Dump as .dat**" option.

3.2 Building Target software for a custom platform

The next step is to build Target software for a custom platform that hosts A2B stack and the application. The A2B stack is responsible for discovering and configuring the A2B network as per the configuration provided and to handle any run-time events/faults. The subsequent sections describe the steps involved in porting the Stack. It may be necessary to implement additional responsibilities in the Target software depending on the end-system requirements which is beyond the scope of this document.

The best way of building Target software for a custom platform is to port a matching demo project (available in A2B Software package under .\Target). Figure 1Figure 3 shows Target software examples on different ADI platforms available within the software package.

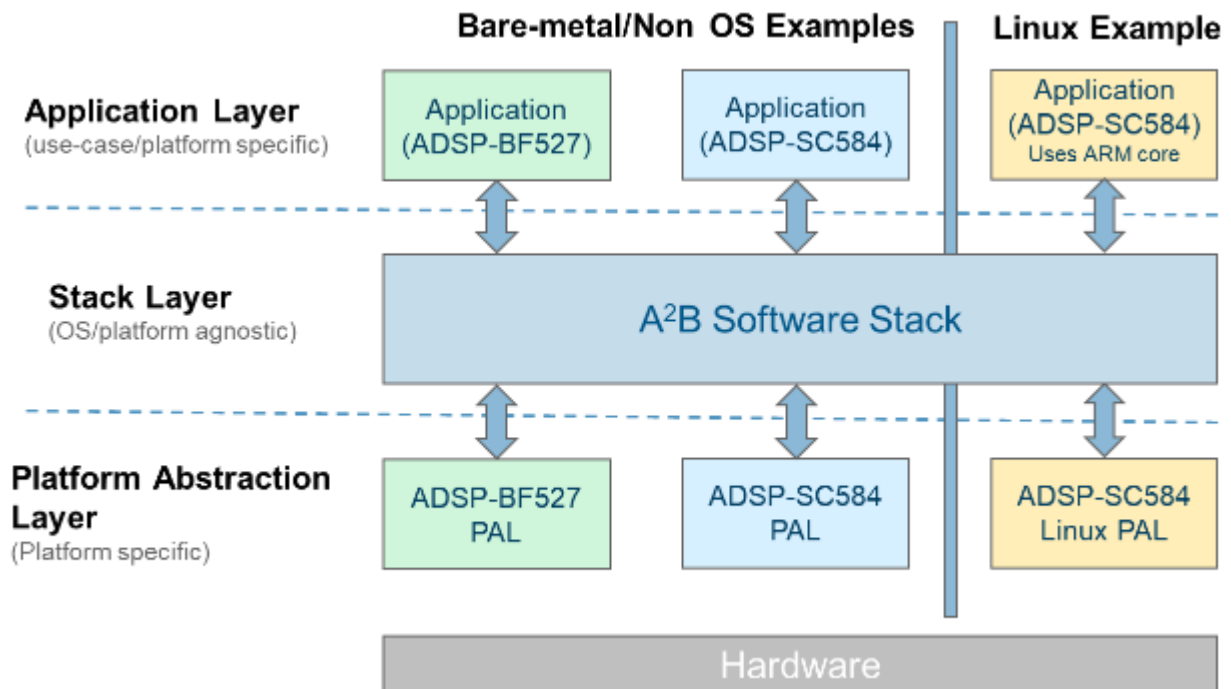


Figure 3: A2B Target Software Examples

Table 2 provides A2B controller, audio host details for each example. Refer to Section 5 of [1] to run the example projects.

Table 2: Target Example Projects

Example Project Name	Platform	A2B Controller	Audio Host
a2bapp-bf	SDP-B + EVAL-AD242xWDZ	BF527	ADAU1452
a2bapp-adsp-sc58x	ADSP-SC584 Ez Kit	ARM A5 (Core 0)	SHARC (Core 1)
a2bapp-linux	ADSP-SC584 Ez Kit	ARM A5 (Core 0)	SHARC (Core 1)

All Target example projects provided in the A2B Software package have a similar directory structure as shown in Figure 4. The figure also shows the folders which constitute core A2B “Stack” and

“Application” files. When porting Stack on to a custom platform, modifications are required only for the files under ‘a2bstack-pal’ and ‘Application’ folders while the rest shall be moved as-is.

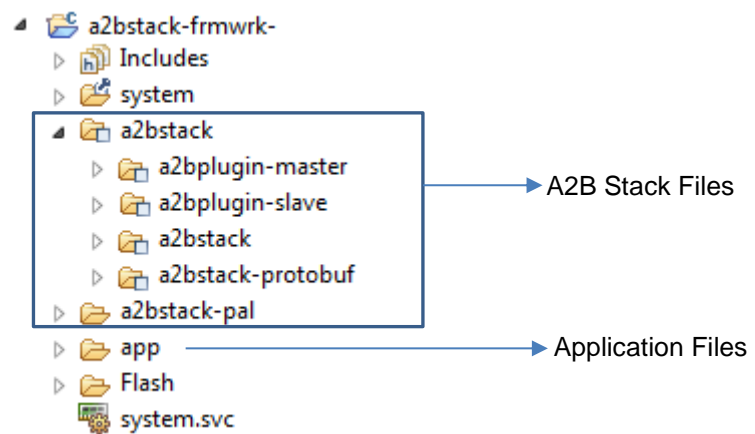


Figure 4: A2B Target Project directory structure

- ▶ a2bstack
 - The generic or target agnostic portions of the A2B Stack. Holds a scheduler designed to efficiently coordinate network activities, especially during the discovery and configuration phase, and execute units of work encapsulated in messages and jobs.
- ▶ a2bplugin-master
 - The sources for the A2B Stack master node plugin. The A2B network discovery algorithms and line fault diagnostics are encapsulated within these sources.
- ▶ a2bplugin-slave
 - The sources for simple A2B stack slave node plugin. These sources are a simple example of a slave plug-in for use as a launching pad for developing custom plugins.
- ▶ a2bstack-pal
 - The platform adaptation layer (PAL) for the A2B Software Stack.
- ▶ a2bstack-protobuf
 - The source code for parsing A2B Bus Configuration File (BCF) from ADI SigmaStudio tool.
 - Source code for parsing and decoding Google Protobuf (Nanopb) encoded A2B configuration file generated by Host Tool.

The steps involved in porting A2B stack and defining application response to A2B events/faults are as follows

1. Port A2B Software Stack to a custom platform. [3.2.1].

2. Apply A2B system configuration. [3.2.2].
3. Modify the application callback code (if necessary). [3.2.3].

3.2.1 Porting A2B Software Stack to a custom platform

Step-by-step approach in porting A2B stack on to a custom platform are as follows

1. Copy all files from folders ***a2bplugin-master***, ***a2bplugin-slave***, ***a2bstack***, ***a2bstack-protobuf***, ***a2bstack-pal***, ***app***, ***inc*** of demo software to corresponding folders of your target platform project “***as-is***”.
2. Re-Implement ***adi_a2b_SystemInit()*** in ***main()*** to perform Target platform specific initializations as required.
 - Replace ADI platform specific Board Support Package (BSP) with Target platform BSP.
 - Ensure to generate and provide Bit Clock and SYNC signals for the master A2B Transceiver chip.
 - Define the stack and heap memory for the Target platform project using the options provided by your development environment (IDE).
 - 4K stack and 5K heap are the typical requirements. If the number of nodes in the system is fixed, then memory can be statically allocated instead of using the Heap.
 - Enabling the macro ***A2B_APP_STATIC_MEMORY_FOR_STACK***, makes use of static memory allocation instead of dynamic memory allocation as preferred by many automotive customers (set to typical values with margin in `.\Target\examples\demo\app-plugin\src\a2bapp.c`)
 - Stack memory - ***A2BAPP_STACK_NW_MEMORY*** (4864 bytes)
 - Plugin memory - ***A2BAPP_PLUGIN_NW_MEMORY*** (704 bytes)
 - BCF File/EEPROM buffer (optional) - ***A2BAPP_EEPROM_BLOCK_MEMORY*** (4096 bytes)
3. Optionally, configure A2B Stack for the Target platform by modifying necessary macros in
 - “features.h” in ***Target/examples/demo/<a2b-xx>/a2bstack-pal/platform/a2b/***
 - “conf.h” in ***Target/examples/demo/<a2b-xx>/a2bstack-pal/platform/a2b/***
4. Re-Implement PAL functions in the file ***a2bstack-pal\adi_a2b_pal.c***
 - This would require implementing drivers (I2C, Timers, SPORT etc) specific to the Target platform under ***a2bstack-pal*** folder. The list of PAL functions to be re-implemented are listed in Table 2.

- Refer for the implementation in the Example projects provided within the software package.
- Each re-implemented function shall be unit tested to confirm that it is working as per the function description before going to the next step.

In the table below, those functions marked as “Required” must be implemented to have a minimally functional Stack. The remaining functions provide developers with convenient points in the Stack operation to insure portability to a wide array of platforms.

Table 3: PAL Functions to be Re-implemented

PAL Function	Required	Description
I2C Functions		
<code>a2b_pal_I2cInit</code>	No	This routine is called to do initialization required by the I2C subsystem.
<code>a2b_pal_I2cOpenFunc</code>	No	This routine is called to do post initialization of the I2C subsystem during the Stack allocation process. This routine is called immediately following a successful call to the <code>pal_i2cInitFunc</code> .
<code>a2b_pal_I2cCloseFunc</code>	No	This routine is called to de-initialize the I2C subsystem.
<code>a2b_pal_I2cReadFunc</code>	No	This routine reads bytes from an I2C device.
<code>a2b_pal_I2cWriteFunc</code>	Yes	This routine writes bytes to an I2C device. Application can implement either a blocking or a non-blocking platform I2C driver call. Non-blocking is preferred since it saves CPU cycles as I2C speed is typically in kHz.
<code>a2b_pal_I2cWriteReadFunc</code>	Yes	This routine performs an atomic repeated start I2C write/read transaction to an I2C device. Application can implement either a blocking or a non-blocking platform I2C driver call. Non-blocking is preferred since it saves CPU cycles as I2C speed is typically in kHz.
<code>a2b_pal_I2cShutdownFunc</code>	No	This routine is called to shut down the I2C subsystem.
Timer Functions		

<code>a2b_pal_TimerInitFunc</code>	No	This routine is called to do initialization the timer
<code>a2b_pal_TimerGetSysTimeFunc</code>	Yes	This routine returns the current "system" time in milliseconds. The underlying system time is platform specific.
<code>a2b_pal_TimerShutdownFunc</code>	No	This routine is called to shut down the timer subsystem during the Stack destroy process.
Audio Functions		
<code>a2b_pal_AudioInitFunc</code>	No	This routine is called to do initialization the audio subsystem during the Stack allocation process.
<code>a2b_pal_AudioOpenFunc</code>	No	This routine is called to do post-initialization the audio subsystem during the Stack allocation process. This routine is called immediately after a successful call to the <code>pal_audioInitFunc</code> .
<code>a2b_pal_AudioCloseFunc</code>	No	This routine is called to de-initialization the audio subsystem during the Stack destroy process.
<code>a2b_pal_AudioConfigFunc</code>	No	This routine is called to configure the audio subsystem master node during the discovery process. This routine is called during the "NetComplete" process after all nodes are discovered and before the master node "NodeComplete" process which fully initializes the master A2B registers and starts the up/downstream flow.
<code>a2b_pal_AudioShutdownFunc</code>	No	This routine is called to shut down the audio subsystem during the Stack destroy process. This routine is called immediately after a successful call to the <code>pal_audioCloseFunc</code> .
Memory Functions <i>(only when A2B_FEATURE_MEMORY_MANAGER is disabled in <code>features.h</code>)</i>		
<code>a2b_pal_MemMgrInitFunc</code>	No	This routine is called to do initialization required by the memory manager service during the Stack allocation process. A PAL implementation has the option of implementing their own (or custom) memory allocation strategy. Another option is to leverage the built-in memory manager feature of the generic Stack if <code>A2B_FEATURE_MEMORY_MANAGER</code> is defined. This manager allocates memory blocks

		from a fixed size heap whose size is derived in part from settings in 'conf.h'.
a2b_pal_MemMgrOpenFunc	No	This routine opens a memory managed heap located at the specified address and of the specified size. If the Stack's heap cannot be opened and managed at the specified location (perhaps because the size is insufficient) then the returned handle will be A2B_NULL. The (optional) built-in memory manager enabled via the A2B_FEATURE_MEMORY_MANAGER definition will use memory pools to avoid fragmentation within a managed region. NOTE: If A2B_NULL is returned, memory allocation must have failed, and the Stack allocation will fail.
a2b_pal_MemMgrMallocFunc	Yes	This routine is called to allocate a fixed amount of memory. Note: Only needed if A2B_FEATURE_MEMORY_MANAGER is disabled.
a2b_pal_MemMgrFreeFunc	Yes	This routine is called to free previously allocated memory. Note: Only needed if A2B_FEATURE_MEMORY_MANAGER is disabled.
a2b_pal_MemMgrCloseFunc	No	This routine is called to de-initialization the memory management subsystem during the Stack destroy process. All resources associated with the heap are freed.
a2b_pal_MemMgrShutdown	No	This routine is called to shut down the memory manager subsystem during the Stack destroy process. This routine is called immediately after a successful call to the pal_memMgrCloseFunc.
Logging Functions <i>(only when A2B_FEATURE_SEQ_CHART or A2B_FEATURE_TRACE is enabled in features.h)</i>		
a2b_pal_LogInitFunc	No	This routine is called to do initialization the log subsystem during the Stack allocation process.
a2b_pal_LogOpenFunc	No	This routine opens a log channel.
a2b_pal_LogCloseFunc	No	This routine is closes a log channel.
a2b_pal_LogWriteFunc	No	This routine writes to a log channel.

<code>a2b_pal_LogShutdownFunc</code>	No	This routine is called to shut down the log subsystem during the Stack destroy process. This routine is called immediately after a successful call to the <code>pal_logCloseFunc</code>
Plugin Functions (Generally not required to be modified. Default implementation should suffice)		
<code>a2b_pal_PluginsLoadFunc</code>	No	This routine returns a list of all available plugins. The plugins returned are queried during discovery as slave nodes are found.
<code>a2b_pal_PluginsUnloadFunc</code>	No	This routine is called to unload previously loaded plugins from <code>pal_pluginsLoad</code> .
<code>a2b_pal_PalGetVersionFunc</code>	No	This routine returns version information related to the PAL.
<code>a2b_pal_PalGetBuildFunc</code>	No	This routine returns build information related to the PAL.

File read Functions (only when A2B_BCF_FROM_FILE_IO is enabled in features.h)		
<code>a2b_pal_FileOpen</code>	No	This routine opens the binary file in read mode and shall be modified as per the file system used.
<code>a2b_pal_FileRead</code>	No	This routine reads the binary file and shall be modified as per the file system used.
<code>a2b_pal_FileClose</code>	No	This routine closes the binary file.

3.2.2 Apply A2B Network configuration

After completing all steps as mentioned in Section 3.2.1 , the next step is to apply bus configuration to the Target software.

1. In the Target platform project, include the validated bus configuration file (**`adi_a2b_busconfig.c`**), exported by following Section 3.1 .
 - Replace the existing in `.\Target\A2BStack\demo\<a2b-xx>\app`.
2. Optionally, if bus configuration is read from a binary file, replace the exported .dat format of bus configuration file into the file system path (`A2B_CONF_BINARY_BCF_FILE_URL`).
3. Optionally, audio routing table (`.\app\adi_a2b_audioroutingtable.c`) may need to be modified if the audio streams are to be routed by the audio host.
 - In case where the A2B controller is also the audio host for the network then modify the audio routing table as explained in Section 5.3.7 of [1]. Otherwise, the routing has to be modified in the audio host. Section 5.1.1 of [1] explains this process when using

ADAU1452 as audio host on ADI A2B evaluation boards such as EVAL-AD2425WDZ and EVAL-AD2428WD1BZ.

Note: This step is not required if stream definition, routing is defined in SigmaStudio where streams are sourced and consumed within A2B nodes and not routed by the Audio Host.

4. Build and Run the Target project.

- Use the build/flash tools provided by your development environment (IDE) to build and run the executable image.
- A2B Network should get discovered and configured as per the added bus configuration file. Refer to Section 5 for Debugging help.

3.2.3 Modify Application call back functions

By this time, we should have completed the porting of A2B Stack as explained in Section 3.2.1 . At this stage, the A2B Stack ported on the custom platform should be capable of discovering and configuring a connected A2B network as per the added bus configuration file.

The A2B Stack offers provision for the application running on the Target software to register callback functions for important network activities. Three important application callback functions are registered with the Stack. These functions can be modified by the user to perform an action specific to the application.

Note: All examples provided in A2B Software package come with default implementations for these callback functions. Modifications to these functions are required only if the default implementation doesn't match with your targeted system requirement. When requiring additional functionality, it is recommended to add on top of the existing implementation unless rewriting completely.

The three application callback functions are explained in the following sub-sections.

3.2.3.1 Discovery completion Callback function

The discovery completion callback function is invoked by the stack upon completing the discovery and configuration of the whole A2B network. Figure 5 shows the application registration of a discovery completion callback function with the stack. The status of the discovery is notified by this function allowing the application to perform any additional tasks based on the notified status.

```
/* send message to start discovery */
nResult = a2b_msgRtrSendRequest(msg, A2B_NODEADDR_MASTER, a2bapp_onDiscoveryComplete);
```

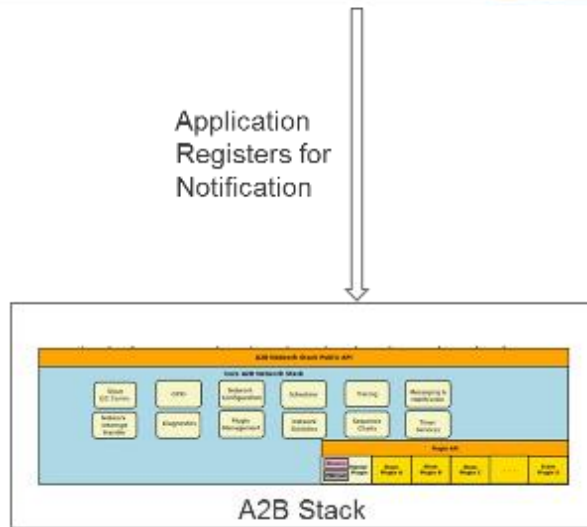


Figure 5: a2bapp_onDiscoveryComplete callback registration

Note: a2bapp_onDiscoveryComplete() comes with default implementation for post discovery bus drop monitoring and rediscovery upon faults (if it was set in SigmaStudio while exporting bus configuration file). Modify this function only to override default functionality (if required).

Code Snippet 1 shows a sample implementation of this callback function.

```

a2bapp_onDiscoveryComplete
(
    struct a2b_Msg* msg,
    a2b_Bool      isCancelled
)
{
    a2b_NetDiscovery* results;
    a2b_Bool* discDone;

    if ( A2B_NULL == msg )
    {
#ifdef A2B_PRINT_CONSOLE
        /* This should *never* happen */
        fprintf(stderr, "Error: no response for network discovery\n");
#endif
    }
    else
    {
        discDone = &discDone;
        if ( isCancelled )
        {
#ifdef A2B_PRINT_CONSOLE
            fprintf(stderr, "Discovery request was cancelled.\n");
#endif
        }
        else
        {
            results = (a2b_NetDiscovery*)a2b_msgGetPayload(msg);
            if ( A2B_SUCCEEDED(results->resp.status) )
            {
                printf("Discovery succeeded with %d nodes discovered\n",
                    results->resp.numNodes);
                gNumSlaveNodes = results->resp.numNodes;
            }
            else
            {
#ifdef A2B_PRINT_CONSOLE
                fprintf(stderr, "Discovery failed!\n");
#endif
            }
        }

        /* Force the main loop to exit */
        *discDone = A2B_TRUE;
    }
}

```

Code Snippet 1: a2bapp_onDiscoveryComplete sample implementation

3.2.3.2 Power/Line Fault Callback function

The power fault callback function is invoked by the stack upon detecting a power related fault in any node of the network. An application callback function can be registered with the Stack for power fault notifications as shown in Figure 6.

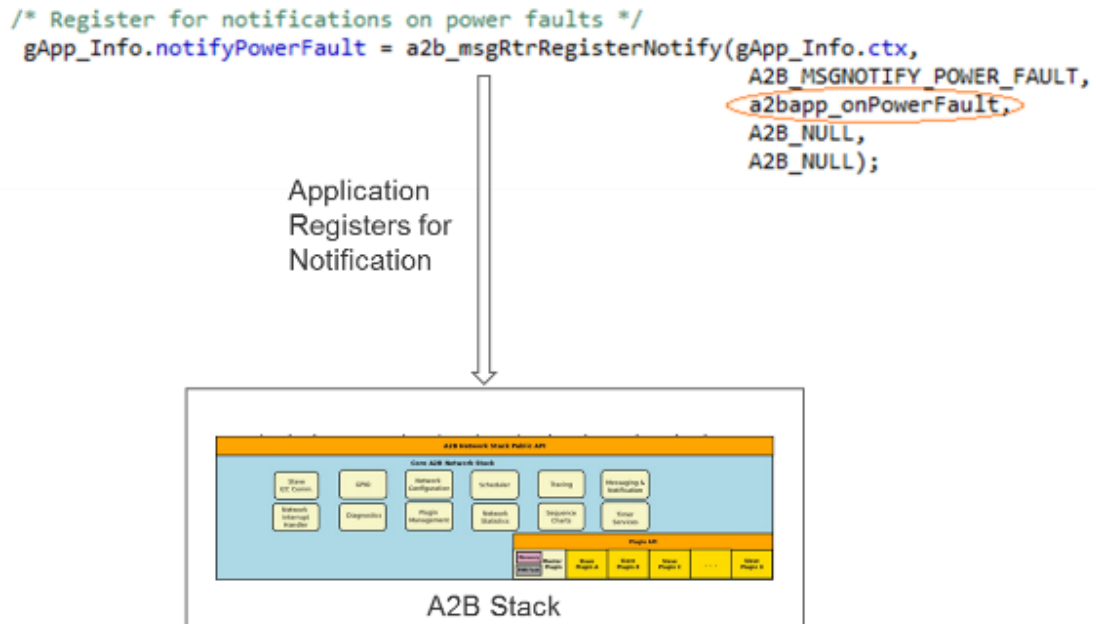


Figure 6: a2bapp_onPowerFault callback registration

The Stack provides callback function to the application layer upon the occurrence of a fault in the A2B System. The stack performs necessary diagnostics and fault localization (in case of concealed faults) and reports the fault type and location to the application for further handling.

Note that the Stack performs all necessary actions to handle the fault as recommended by A2B Transceiver Programmer's reference manual and finally invokes the application callback.

The function is invoked under the following fault conditions during and post discovery.

- Critical faults
 - Cable terminal shorted to GND
 - Cable terminal shorted to VBat
- Non-Critical faults
 - Cable terminals shorted together
 - Cable disconnected or open circuit
 - Cable is reverse connected
- Indeterminate faults
- Bus/Node drop condition

Code Snippet 2 shows a sample implementation of `a2bapp_onPowerFault` callback function.

The information about the presence of local powered slave is made known to the stack through the BDD (please refer SigmaStudio user guide – node properties description to set “Local powered”). In case of critical faults (Cable terminal shorted to GND, Cable terminal shorted to VBat), the stack switches of the bus from the immediate upstream local powered slave onwards.

Partial bus operation is possible between master and this upstream local powered slave.

```

a2bapp_onPowerFault (struct a2b_Msg *msg, a2b_Handle userData)
{
    A2B_UNUSED(userData);
    a2b_PowerFault *fault;
    const char *faultString;

    if ( msg )
    {
        fault = (a2b_PowerFault *)a2b_msgGetPayload(msg);
        if ( fault )
        {
            if ( A2B_SUCCEEDED(fault->status) )
            {
                switch (fault->intrType)
                {
                    case A2B_ENUM_INTTYPE_PWRERR_CS_GND:
                        faultString = "Cable Shorted to GND";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR_CS_VBAT:
                        faultString = "Cable Shorted to VBat";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR_CS:
                        faultString = "Cable Shorted Together";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR_CS_SC:
                        faultString = "Cable Shorted or Open Circuit";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR_CREV:
                        faultString = "Cable Reverse Connected or Wrong Port";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR_FAULT:
                        faultString = "Indeterminate Fault";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR-NLS_GND:
                        faultString = "Non-Localized Short to GND";
                        /* Add your code to handle fault */
                        break;
                    case A2B_ENUM_INTTYPE_PWRERR-NLS_VBAT:
                        faultString = "Non-Localized Short to VBat";
                        /* Add your code to handle fault */
                        break;
                    default:
                        faultString = "Unknown";
                        /* Add your code to handle fault */
                        break;
                }
                gApp_Info.faultNode = fault->faultNode;
            }
        }
    }
}

```

Sample Callback implementation

Code Snippet 2: a2bapp_onPowerFault sample implementation

The Interrupt callback function is invoked by the Stack upon seeing any interrupts at the master node. Figure 7 shows the application registration of an interrupt callback function with the Stack.

Application registers for notification



```
static void a2bapp_onInterrupt(struct a2b_Msg* msg, a2b_Handle userData)
{
    a2b_Interrupt* interrupt;

    A2B_UNUSED(userData);

    if (msg)
    {
        interrupt = a2b_msgGetPayload(msg);

        if (gDebug)
        {
            if (interrupt)
            {
#ifdef A2B_PRINT_CONSOLE
                printf("Interrupt type=%u nodeAddr=%d\n",
                    interrupt->intrType, interrupt->nodeAddr);
#endif
                /* your code to handle interrupt */
            }
        }
    }

#ifdef A2B_PRINT_CONSOLE
    fprintf(stderr, "INTERRUPT: failed to retrieve payload\n");
#endif
}
}
```

Note: Any interrupt on the slave node can be handled within `a2bplugin_slave\ a2bslave plugin.c` file in the function `a2b pluginInterrupt` as explained in Section 4.3.2

3.2.3.4 Node Discovery Callback function

The node discovery callback function is an optional callback, which is invoked by the stack upon each node discovery or when node authentication fails. Figure 8 shows the application registration of this callback function with the Stack.

```
/* Register for notifications on node discovery */
pApp_Info->notifyNodeDiscvry = a2b_msgRtrRegisterNotify(pApp_Info->ctx,
A2B_MSGNOTIFY_NODE_DISCOVERY, a2bapp_onNodeDiscovery, pApp_Info, A2B_NULL);
```

Application
Registers for
Notification

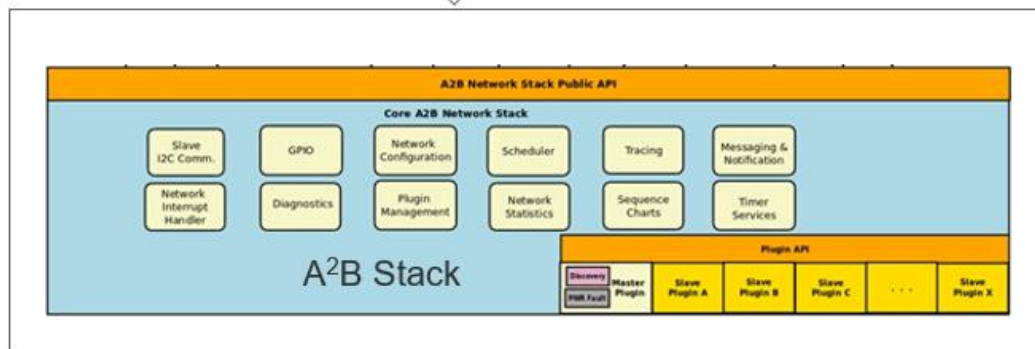


Figure 8: a2bapp_onNodeDiscovery callback registration

Code Snippet 34 shows a sample implementation of this callback function. The application can decide whether to continue with discovery or not and has more control with this callback function.

```
static void a2bapp_onNodeDiscovery(struct a2b_Msg* msg, a2b_Handle userData)
{
    a2b_Nodedscvry* dscvrdNodeMsg;

#ifdef A2B_PRINT_CONSOLE
    a2b_Int16 nodeAddr;
    bdd_Node *bddnode;
    bdd_Network *bddNetwork;
#endif
    A2B_UNUSED(userData);

    if (msg)
    {
        /* details of the currently discovered node */
        dscvrdNodeMsg = a2b_msgGetPayload(msg);

#ifdef A2B_PRINT_CONSOLE
        nodeAddr = dscvrdNodeMsg->nodeAddr; //the address of slave node discovered
        bddNetwork = (bdd_Network *)dscvrdNodeMsg->bddNetwork;
        bddnode = (bdd_Node *)(&bddNetwork->nodeAddr);
#endif
        if (dscvrdNodeMsg)
        {
            A2B_APP_LOG("Node Discovery: nodeType=%u nodeAddr=%d discoveryCompleteCode=%u\n\r",
                        dscvrdNodeMsg->nodeType, nodeAddr, dscvrdNodeMsg->discoveryCompleteCode);

#ifdef ENABLE_SUPERBCF
            /* SuperBCF: Populate the further action which is required to be taken by stack
             * bContinueDisc to true if required to proceed with discovery process
             * bContinueDisc to false if required to end the discovery process
             */
            dscvrdNodeMsg->bContinueDisc = A2B_TRUE;
#endif /* ENABLE_SUPERBCF */
        }
        else
        {
            A2B_APP_LOG("NODE DISCOVERY: failed to retrieve payload\n\r");
        }
    }
}
```

Sample Callback implementation

Code Snippet 4: a2bapp_onNodeDiscovery sample implementation

3.3 Summary of Building A2B Application on custom platform

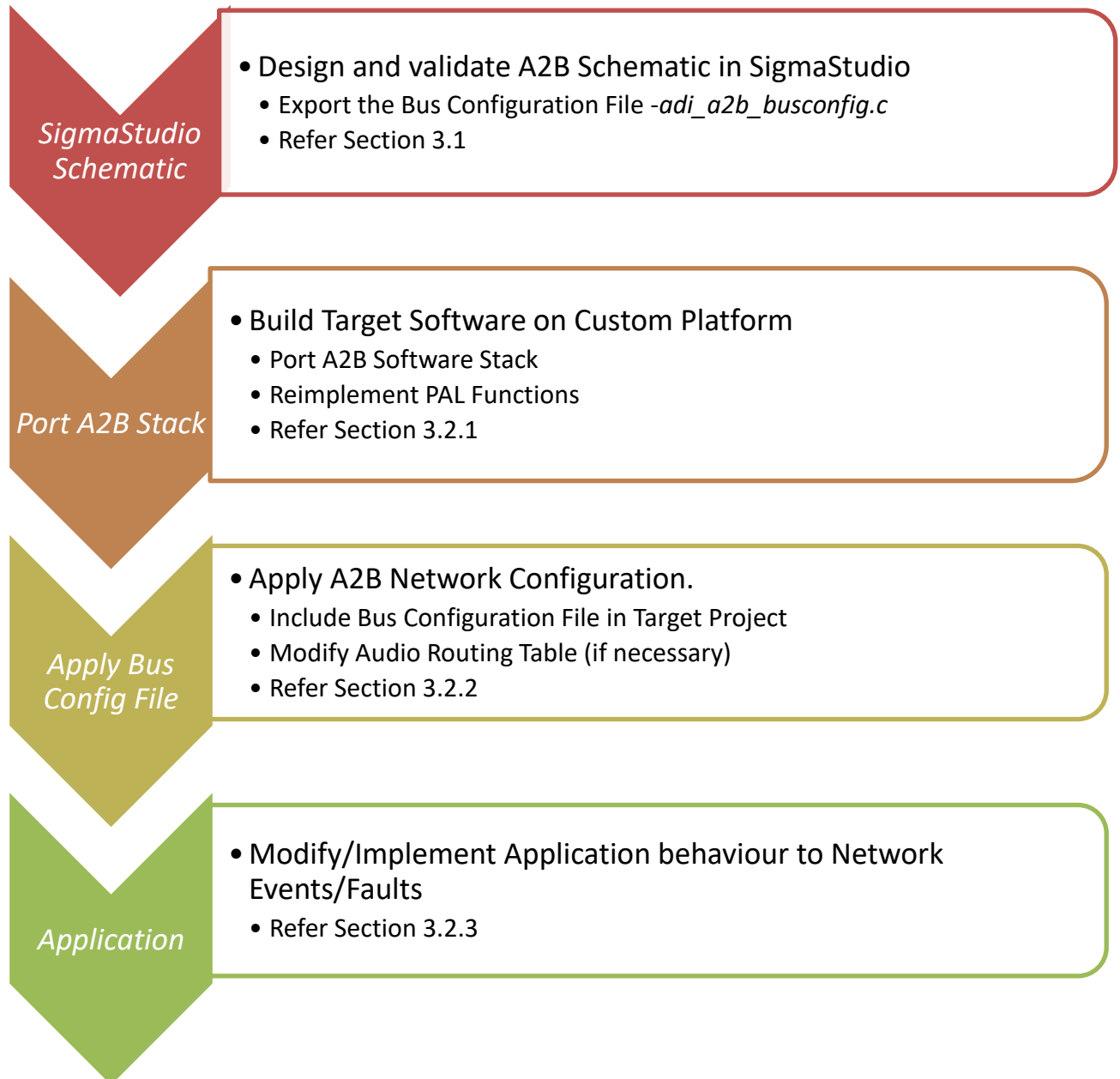


Figure 9: Building A2B Application on a Custom Platform

4 Application Integration

Integrator can take 3 approaches to integrate A2B stack:

1. Using Wrapper Services Layer 1
2. Using Wrapper Services Layer 2
3. A2B Network Stack Public APIs

1. Using Wrapper Services Layer 1:

This is the top-most layer below Application (refer Figure 1). This provides ease of integration with minimal service APIs. Example implementation of this layer is available in `.\Target\examples\demo\app-plugin\src\a2bapp.c`.

The main services provided in this layer are:

1. A2B network setup - **a2b_setup()**
 - Implements the higher level service of network setup including stack memory allocation, initialization and discovery
2. A2B fault monitor - **a2b_fault_monitor()**
 - If line diagnostics is enabled this function checks if a line fault occurred post discovery and initiates re-discovery.
3. A2B stack time tick - **a2b_stackTick()**
 - Stack tick function ensures that the stack is periodically called to keep all the processes/states rolling within the stack.
4. A2B stop - **a2b_stop()**
 - This function stops stack, un-registering call-backs, turning off interrupt polling, disabling sequence charts, and freeing resources associated with the application context.

Below code snippet shows a sample usage of Wrapper services layer 1 APIs as in `.\Target\examples\demo\a2b-bf\appc\a2bapp_bf.c`.



```

void main(int argc, char *argv[])
{
    a2b_UInt32    nResult = 0;
    bool         bRunFlag = true;
    A2B_SPI_RET   eSpiRet;

    adi_initComponents();

    /* system/platform specific initialization */
    nResult = adi_a2b_SystemInit();
    if(nResult != 0)
    {
        assert(nResult == 0);
    }

    /* A2B Network Setup. Performs discovery and configuration of A2B nodes and its peripherals */
    nResult = a2b_setup(&gApp_Info);

    if (nResult)
    {
        /* failed to setup A2B network */
        assert(nResult == 0);
    }

    while(1)
    {
        /* Monitor a2b network for faults and initiate re-discovery if enabled */
        nResult = a2b_fault_monitor(&gApp_Info);

        /* tick keeps all process rolling.. so keep ticking */
        a2b_stackTick(gApp_Info.ctx);
    }
}

a2b_UInt32 a2b_fault_monitor(a2b_App_t *pApp_Info)
{
    a2b_UInt32 nResult = 0;
    a2b_UInt8 nChainIndex;

    /* If line diagnostics enabled and non-zero re-attempts configured */
    /* If fault has occurred */
    if ((pApp_Info->bRetry == A2B_TRUE) && (pApp_Info->bfaultDone == A2B_TRUE))
    {
        /* delay between re-discovery attempt */
        a2b_ActiveDelay(pApp_Info->ctx, pApp_Info->pTargetProperties->nRediscInterval);

        /* stop a2b stack */
        nResult = a2b_stop(pApp_Info);

        /* Re-discover the network */
        pApp_Info->ecb.palEcb.nChainIndex = nChainIndex;
        nResult = a2b_setup(pApp_Info);
    }
    return (nResult);
}

```

Code Snippet 5: Wrapper Services Layer 1 usage

2. Using Wrapper Services Layer 2:

.\Target\examples\demo\app-plugin\src\a2bapp.c provides wrapper services for achieving stack functionality. The services of this layer is invoked by Wrapper Services Layer 1. This layer invokes the A2B Network Stack Public APIs, which are core A2B stack services.

Figure 10 shows the state transition diagram for different Stack states at Wrapper Services Layer 2. The stack starts network discovery after it has been allocated and loaded with the configuration file. After discovery, the stack continues to poll indefinitely for interrupts/events until it is stopped or encounters a critical error in any of the earlier states.

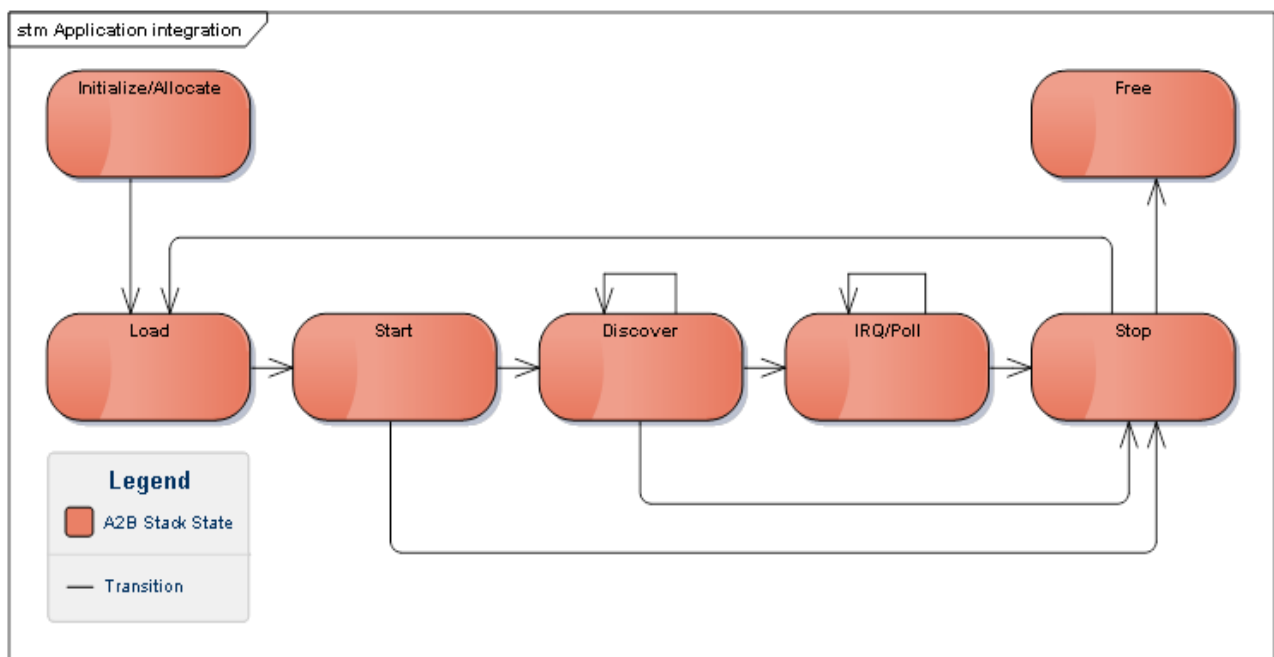


Figure 10: Application Level State Transition Diagram

Each Stack state is explained in the following sub section.

3. A2B Network Stack Public APIs:

Once an integrator is familiar with the wrappers services like Initialize, Load, Discover, etc. as described above, he/she can implement his/her own services using the A2B Network Stack Public APIs directly (refer Figure 1).

4.1 Stack States

4.1.1 Initialize/Allocate

The Stack requires the application to allocate and initialize one or more contexts to run. Each individual context maintains the complete Stack state for an A2B network. Multiple contexts allow the Stack to manage multiple A2B networks simultaneously.

This state corresponds to the function `a2b_allocate()` in the application file `a2bapp.c`.

4.1.2 Load

This application state loads A2B bus configuration data into the Stack context. A bus configuration file exported from SigmaStudio (or Bus Description Data from Mentor A2B Analyzer Application) contains all of the information needed to perform a successful discovery and configuration of an A2B network.

SigmaStudio generated BCF file can be optionally encoded in a Google Protocol Buffer (Protobuf) format. Mentor generated BDD file is always encoded in Protobuf format. More information on Google Protocol Buffers can be found here: <https://developers.google.com/protocol-buffers/>. This function decodes and loads bus configuration when the included file is protobuf encoded.

This state corresponds to the function `a2b_load()` in the application file `a2bapp.c`.

4.1.3 Start

Starting the Stack involves instructing the Stack to begin polling for interrupts, enabling sequence charts and debugging output, and hooking in application level call-backs.

This corresponds to the function `a2b_start()` in the application file `a2bapp.c`.

4.1.4 Discover

Discovery starts when the application sends an `A2B_MSGREQ_NET_DISCOVERY` message to the Stack. Once this message has been sent, the application should transition to the Polling state in order to complete the discovery process.

This state corresponds to the function `a2b_discover()` in the application file `a2bapp.c`.

4.1.5 Interrupt Poll

Polling for system events is simply calling the tick function of the Stack on a regular basis. The Stack will enforce the interrupt poll time established in the Start state if called too often.

The calls to `a2b_stackTick()` drive the internal scheduler which ultimately drives all aspects of the Stack. The internal scheduler runs every `A2B_CONF_SCHEDULER_TICK_MULTIPLE` ticks as defined in `conf.h`.

By default, `A2B_CONF_SCHEDULER_TICK_MULTIPLE` is set to two (2). Therefore, if `a2b_stackTick()` is called every 5ms a job will be scheduled every 10ms. Change this value to one (1) to have the scheduler run on each tick. While the Stack itself is neither thread nor interrupt safe, it can be called as a result of an interrupt to minimize system latency to A2B events. Following an interrupt, one should call `a2b_intrQueryIRQ()`. This call into the Stack must be from the same thread of execution as all of the other Stack calls.

This call will service up to `A2B_CONF_CONSECUTIVE_INTERRUPTS` as defined in `conf.h`.

We can also have interrupt-based event handling instead of polling system events by enabling the macro `ENABLE_INTERRUPT_PROCESS`. The `a2b_processIntrpt()` function will periodically check if a pin interrupt is latched and will process them. An example is provided in. `\Target\examples\demo\app-plugin\src\A2Bapp.c`. Platform specific interrupt callback shall be implemented by the integrator to latch the pin interrupt.

4.1.6 Stop

Stopping the Stack involves un-registering call-backs, turning off interrupt polling, disabling sequence charts, and freeing resources associated with a particular network configuration.

This state corresponds to the function `a2b_stop()` in the application file `A2Bapp.c`.

4.1.7 Free

Freeing the Stack is simply a matter of freeing the application context container.

This state corresponds to the function `a2b_free()` in the application file `A2Bapp.c`.

4.2 Application Extensions to Environment control block

The environmental control block (or ECB) is the container for all platform and environment data passed throughout the Stack. Most PAL functions receive a pointer to the ECB making this a central data structure for the PAL.

The core Stack ECB is defined by the `a2b_Ecb` structure which is comprised of two other sub structure definitions as shown in Table 4.

Table 4: ECB components

Data Type	Description
<code>a2b_BaseEcb</code>	This contains the basic platform independent environment parameters. This structure must be defined first in the ECB structure.
<code>a2b_PalEcb</code>	These are the platform specific environment properties defined by the PAL. This structure must be defined second in the ECB structure.

Application specific extensions can be added to the standard Stack ECB by adding custom application fields in the appropriate structures of `a2b_Ecb`.

4.3 Plugin Architecture

One of the most powerful elements of the Stack is the plug-in architecture for initializing and managing slave nodes throughout an A2B network lifecycle. Plugins can initialize peripheral hardware on slave nodes, manipulate GPIO, communicate directly with I2C devices, create timers for periodic events, service interrupts, and monitor A2B diagnostic registers. Plugins can also send/receive notifications to/from the application to enable rich interactive system features. Custom plugins can be developed to support new A2B hardware.

4.3.1 Plugin Examples

The Stack in all example Target projects comes with a Master and a generic Slave plugin designed using the plugin architecture.

4.3.1.1 Master Plugin

Slave node discovery and diagnostics are coordinated and enabled by the Master plugin. The Master plugin supports a variety of discovery modes, line diagnostics, as well as slave EEPROM configuration processing and custom node authentication.

If necessary, the Master plugin can be customized. It is always recommended to add customizations on top of the existing implementation rather than replacing with a newer version as the Master plugin is responsible for some important functions like discovery, diagnostics etc.

4.3.1.2 Generic Slave Plugin

The default slave plugin provided with the Stack example projects is generic in nature and has minimal command handling with support for initializing and de-initializing peripherals connected on slave nodes. The plugin always responds affirmately to query requests during discovery making it “default” plugin when included within a system.

A Custom Slave plugin may be necessary only in cases where the slave node needs to run a complex functionality specific to the slave board capabilities. Otherwise, a common generic slave plugin may be sufficient for all cases as used in Stack example projects in the software package.

4.3.2 Handling Interrupts in a Plugin

In some cases, it may be necessary for the plugins to handle interrupts generated by A2B nodes. The function `a2b_pluginInterrupt()` in the plugin shall be implemented to handle such interrupts. The Stack takes care of passing the interrupt to appropriate plugin depending on the interrupt type and location.

The Master plugin in the Stack comes with default implementation to handle master interrupts and to invoke appropriate application callback functions if registered.

The generic Slave plugin in the Stack doesn't come with default implementation for interrupts generated by a slave node (GPIO pin). If a specific functionality is required on a slave node upon an

interrupt, it can be implemented in `a2b_pluginInterrupt()` function of the slave plugin after checking the Interrupt Type and Source node as shown in Code Snippet 6.

```
static void a2b_pluginInterrupt (
    struct a2b_StackContext*   ctx,
    a2b_Handle                 hnd,
    a2b_UInt8                  intrSrc,
    a2b_UInt8                  intrType
)
{
    a2b_Plugin* plugin = a2b_pluginFind(hnd);

    A2B_UNUSED(ctx);

    if ( A2B_NULL != plugin )
    {
        switch ( intrType )
        {
            /* Slave plugins *only* receive GPIO interrupts */
            case A2B_ENUM_INTTYPE_IO0PND:
            case A2B_ENUM_INTTYPE_IO1PND:
            case A2B_ENUM_INTTYPE_IO2PND:
            case A2B_ENUM_INTTYPE_IO3PND:
            case A2B_ENUM_INTTYPE_IO4PND:
            case A2B_ENUM_INTTYPE_IO5PND:
            case A2B_ENUM_INTTYPE_IO6PND:
            case A2B_ENUM_INTTYPE_IO7PND:
                a2b_emitSlaveDtc(plugin, intrSrc, intrType);
                break;

            default:
                break;
        }
    }
} /* a2b_pluginInterrupt */
```

Sample code snippet

Code Snippet 6: `a2b_pluginInterrupt` dummy implementation

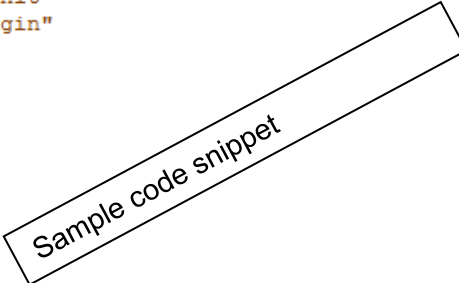
4.3.3 Writing a Custom Plugin

A custom plugin can be developed to support a new A2B hardware based on an example plugin. A Stack plugin must export a set of functions as indicated in Table 5.

Table 5: Plugin Functions

Function Name	Use
<code>a2b_pluginInit()</code>	Called by the Stack once to initialize the <code>a2b_PluginApi</code> structure allowing the plugin to register the remaining entry-point functions. This is the only function that should be exported by a plugin.
<code>a2b_pluginOpen()</code>	Called during network discovery to see if the plugin handles a specific node. Any plugin related resources should be allocated here.
<code>a2b_pluginExecute()</code>	Called when a job needs to be processed by this plugin.
<code>a2b_pluginInterrupt()</code>	Called to process an interrupt for the slave associated with this plugin. Slave plugins only receive GPIO related interrupts.
<code>a2b_pluginClose()</code>	Called to close the plugin. Any plugin related resources should be freed here.

A typical `a2b_pluginInit()` function for a plugin looks like this:



```

#define MY_PLUGIN_INIT a2b_myPluginInit
#define MY_PLUGIN_NAME "My Slave Plugin"

typedef struct a2b_Plugin
{
    struct a2b_StackContext* ctx;
    a2b_NodeSignature nodeSig;
    a2b_Bool inUse;
    struct a2b_Timer* timer;
} a2b_Plugin;

/* My plugin context pool */
static a2b_Plugin gsPlugins[A2B_CONF_MAX_NUM_SLAVE_NODES];

a2b_Bool MY_PLUGIN_INIT(struct a2b_PluginApi* api)
{
    a2b_Bool status = A2B_FALSE;
    a2b_UInt32 idx;

    if ( A2B_NULL != api )
    {
        api->open = a2b_pluginOpen;
        api->close = a2b_pluginClose;
        api->execute = a2b_pluginExecute;
        api->interrupt = a2b_pluginInterrupt;
        a2b_strncpy(api->name, MY_PLUGIN_NAME, sizeof(api->name) - 1);
        api->name[sizeof(api->name) - sizeof(a2b_Char)] = '\0';

        /* Initialize my context pool */
        for ( idx = 0; idx < A2B_ARRAY_SIZE(gsPlugins); ++idx )
        {
            a2b_memset(&gsPlugins[idx], 0, sizeof(gsPlugins[idx]));
        }

        status = A2B_TRUE;
    }

    return status;
}

```

Code Snippet 7: Custom PluginInit implementation

Implementation details for other plugin functions can be referred in file `.\a2bplugin-slave\src\slaveslave_plugin.c`.

4.3.4 Loading Plugins into the Stack

Plugins are loaded into the Stack through the `a2b_PluginsLoadFunc` PAL function. This function returns a structured list of pointers pointing to the `a2b_pluginInit()` function of each plugin.

During the discovery process, each registered plugin is queried, in order, when a new slave node is discovered to determine whether or not that plugin can service the node. The first plugin to respond affirmatively by returning a non-NULL value, will be assigned by the Stack to that node.

Since each discovered slave node carries its own context, **a single plugin can service more than one slave node concurrently**. It's important to note, however, that each slave plugin instance is responsible for maintaining its context.

4.4 Using A2B Stack for Multi-Master Network

In cases where the Host processor is controlling multiple A2B Masters, it is necessary to maintain multiple stack instances. One stack and application context is mapped per network master. Each individual context maintains the complete Stack state for an A2B network. Multiple contexts allow the Stack to manage multiple A2B networks simultaneously.

Each application context can either register separate callback functions (for Discovery completion, Power Fault or Interrupt events) or have single function with unique callback parameter for each network chain.

An example project to demonstrate multi-master bus set up is provided in '*ADI_A2B_Software-RelX.Y.Z\Target\examples\advancedappl\multimaster*' of the A2B Software package. This example uses ADSP-SC584 processor to discover and route the audio between two A2B networks. In the example project, the structure `a2b_App_t` represents application level context. Separate objects of this structure are created for each network instance. Each instance is identified with an index – '*nChainIndex*', starting with 0. This parameter is used inside notification callback & PAL functions to differentiate the handling between two A2B networks.

*Note that when requiring to support multiple A2B Masters on a different platform, it is not just sufficient to change the macro '*A2B_CONF_MAX_NUM_MASTER_NODE*' in *Target/examples/demo/<a2b-xx>/a2bstack-pal/platform/a2b/conf.h* but also would require modifications to the functions in *adi_a2b_pal.c*.*

4.5 Inter-Processor Communication over A2B Mailbox

The Stack running on the master node target processor can communicate with an intelligent slave node having a connected processor. This can be achieved by using the Mailbox Communication Channel module. The module enables exchange of control and command messages between the two processors.

An example project for demonstrating the inter-processor communication using mailbox communication channel is provided in 'ADI_A2B_Software-RelX.Y.Z\ Target\examples\ advancedapp\mbboxcommch' of the A2B Software package. Refer [4] for more details on running the demo. The document also provides details of the module APIs and the integration approach.

One example use case of this module is when Custom node authentication using Mailbox option is set in SigmaStudio. In this case, the Stack running on the Target/Host processor queries a slave node processor for Node Identifier using `A2B_COMMCH_MSG_REQ_SLV_NODE_SIGNATURE` and the slave node responds with `A2B_COMMCH_MSG_RSP_SLV_NODE_SIGNATURE` using the module API as shown in Figure 10. The figure also shows an example sequence of exchange when the slave initiates a request message transmission with message ID say `A2B_COMMCH_MSG_REQ_MSTR_VERSION` and the master responds back with the response message ID `A2B_COMMCH_MSG_RES_MSTR_VERSION`.

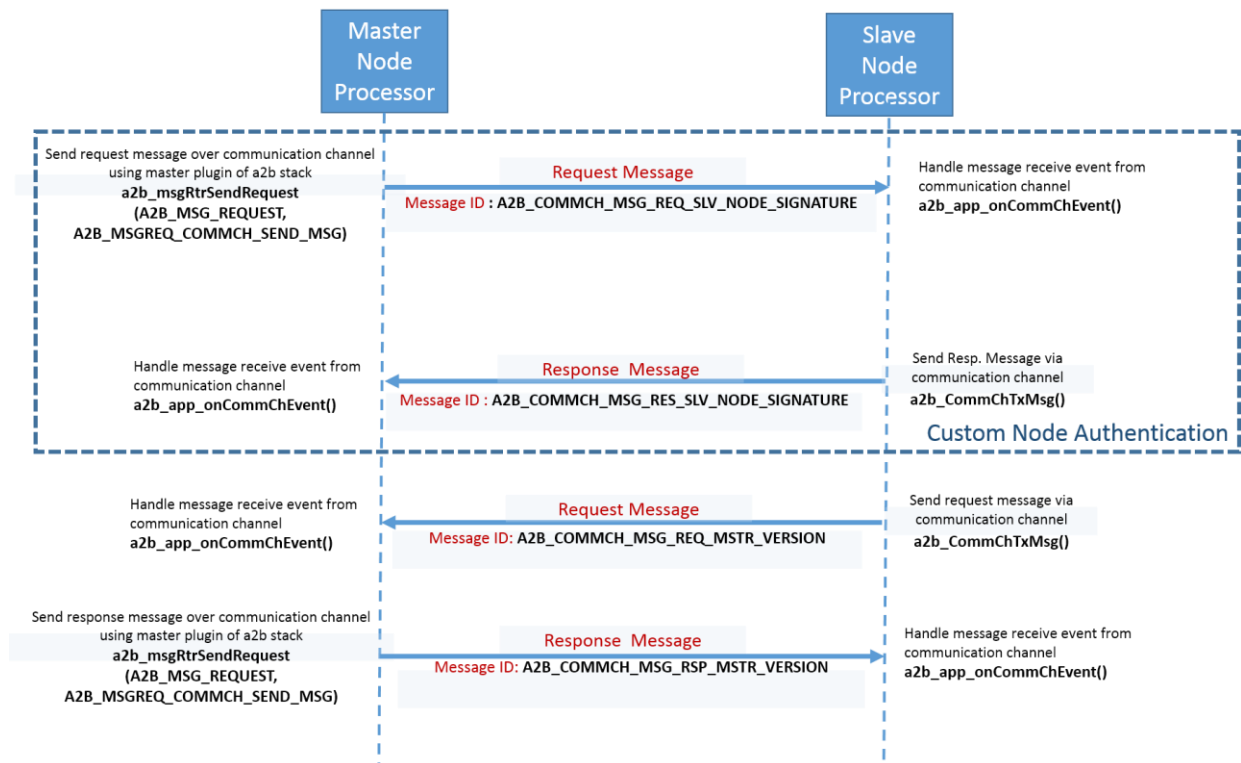


Figure 10: Message exchange between Master and Slave node processor

Every message communicated has a unique ID assigned that is common across master and slave nodes. Message ID is 6 bits in length. Message IDs up to 0xA are reserved to be used for communication with the Master Plugin and should not be used by the application. All these message IDs are defined in 'ADI_A2B_Software-RelX.Y.Z/Target/a2bcommchannel/inc/adi_a2b_commch_interface.h' file.

The Target example projects in 'ADI_A2B_Software-RelX.Y.Z/Target/examples/demo' does not come with the mailbox communication channel module integrated. To use the module for inter-processor communication the following pre-requisites shall be met.

4.5.1 Pre-Requisites for Inter-Processor Communication

1. The mailbox registers in the slave node needs to be configured appropriately during discovery. This is controlled by the bus configuration file (*adi_a2b_busconfig.c*) exported from Sigma Studio schematic. While designing the A2B schematic in Sigma Studio the mailbox registers *MBOX0_CTL* and *MBOX1_CTL* should be set to the values 0x3D and 0x3F from the register tab view for the slave node to which communication channel messages via mailbox is to be exchanged. The above register settings ensure the following configurations of mailbox:
 - Mailbox data length should be 4 bytes
 - Mailbox full and empty interrupts should be enabled
 - Mailbox 0 should be configured as receive mailbox (where master transmits to slave) and mailbox 1 should be configured as transmit (where slave transmits to master).
2. The communication channel feature must be enabled in the Target project by defining the macro *A2B_FEATURE_COMM_CH* in the stack configuration file *features.h*.
3. The following header files need to be included in the Target project. The header files are available in the folder 'ADI_A2B_Software-RelX.Y.Z/Target/a2bcommchannel/inc/' in the release package.
 - *adi_a2b_commch_interface.h* - Contains the message identifiers that are reserved to be exchanged by A2B master plugin of A2B stack on the master node with communication channel on slave nodes. User should not modify these macros
 - *adi_a2b_commch_mstr.h* - Contains the structures, data types and function declarations for master communication channel. User configurable macros are also present in this file.
 - *adi_a2b_commch_engine.h* - Contains the structures, data types and function declarations of the communication channel engine. The communication engine runs the framing/de-framing protocol.

The directory structure of the Target example project with the above include files are shown in Figure 11.

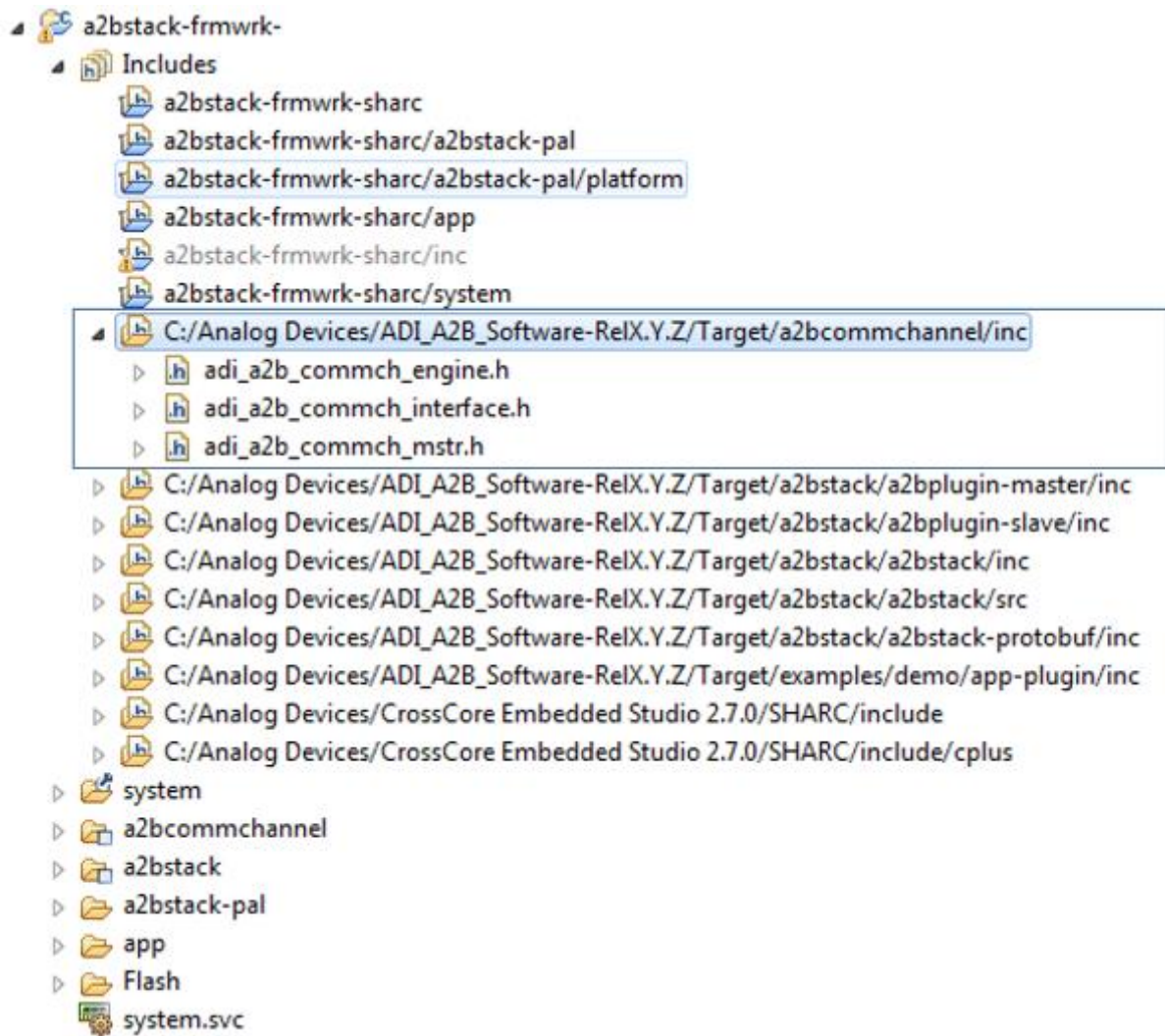


Figure 11: A2B Target project directory structure with communication channel includes

4. The following source files need to be included in the application. The source files are available in the folder '*ADI_A2B_Software-RelX.Y.Z/Target/a2bcommchannel/src*' in the release package
 - *adi_a2b_commch_mstr.c* - Contains the function definitions for master communication channel which are used by the Master Plugin to create and use the communication channel for transmitting & receiving messages.
 - *adi_a2b_commch_engine.c* - Contains the function definitions of the communication channel engine. The communication engine runs the framing/de-framing protocol.

The directory structure of the Target example project with the above source files included as a part of the virtual folder '*a2bcommchnl*' are shown in Figure 12.

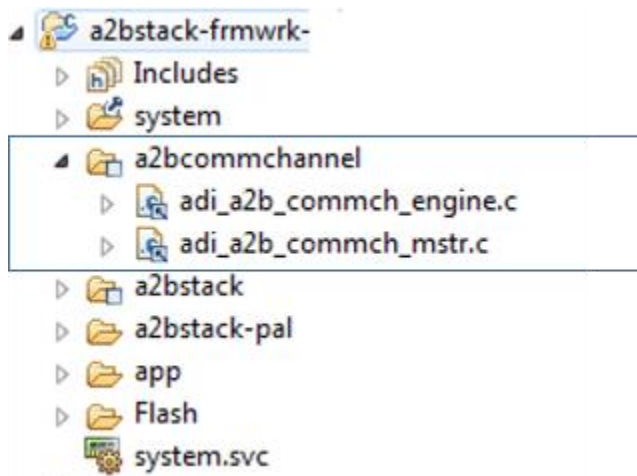


Figure 12: A2B Target project directory structure with communication channel sources

For details on integration of A2B mailbox into application, please refer [4].

4.6 Post discovery APIs

A2B network's biggest advantage is "setup once and forget". After network discovery and setup of all the nodes, minimal host intervention is required. Audio streams get transmitted across nodes seamlessly. So host intervention is only limited to fault monitoring. Host action is required only when bus faults are detected.

To ease applications in performing some basic post discovery operations, the following APIs are provided to the application. Post discovery APIs are provided in `.\Target\examples\demo\app-plugin\src\A2BApp.c`.

Table 6: Supported post discovery APIs

API name	Use
<code>a2b_reset()</code>	This function does A2B network soft reset.
<code>a2b_AppWriteReg()</code>	This function writes a register value to a particular A2B node.
<code>a2b_AppReadReg()</code>	This function reads a register value from a particular A2B node.
<code>a2b_app_handle_becovf()</code>	This routine periodically resets BECNT and BECOVF registers and checks for bus drop.
<code>a2b_AppDetectBusDrop()</code>	This function reads the Vendor Id register of all A2B nodes discovered and declares a bus drop at a particular node where the read value is not the expected.

5 Appendix A: Diagnostics and Debugging

Diagnostic and debugging is one of the most powerful aspects of the Stack. Every logged event is timestamped and presented to the PAL for storage or reporting. Internally logged items include:

- All I2C transactions
- All messages
- All timer events
- All application interactions
- Discovery details
- Power fault details

The Stack included two discrete classes of logging. The first class of logs are used to build structured sequence diagrams. The second class are unstructured trace messages. While both types utilize the `pal_logXXX()` functions, separate handles allow the PAL layer to distinguish between the two logging classes.

Sequence diagrams provide unparalleled transparency into the complex interactions between the master and slaves throughout the entire lifecycle of an A2B network. This includes all aspects of discovery and steady-state operation after discovery.

Once the deployment of an A2B network reaches a certain state of stability, trace messages can be utilized to log Stack operations in a less detailed manner than the Sequence diagrams. Trace messages are routinely integrated into larger system level logging frameworks where message types and severity can be monitored and filtered.

Additionally, the Stack automatically performs power and line fault diagnostics whenever a network discovery fails. The diagnostics are reported back to the application through the diagnostic event handler registered with the Stack.

5.1 Generating Sequence Diagrams

The sequence diagrams created by the Stack are compatible with an open source tool called PlantUML (<http://plantuml.com/>). The raw syntax for PlantUML is human readable and friendly for processing with diff tools or checking into document repositories. When post-processed by the PlantUML tool, extremely rich graphical sequence diagrams, can be created.

The script to post process the sequence diagrams to a more readable format is given in '*Target/tools*'. Note: Python must be installed on the developer's system for the post processing script to function.

5.1.1 Sequence diagram support in the Stack

- Sequence diagram support is an optional Stack feature and must be enabled by defining `A2B_FEATURE_SEQ_CHART` in '*features.h*' prior to compiling the Stack.
- Once sequence diagram support is included in the Stack, sequence diagrams must be enabled as part of the Stack start-up as explained in 4.1.3 .

- All logging, including sequence diagrams and tracing, go through the `pal_logXXX()` set of functions in the PAL. The table below illustrates the primary functions:

Table 7: PAL Logging Functions – Info

PAL Function	Notes
a2b_LogOpenFunc	<p>The second argument passed to <code>a2b_seqChartStart()</code> is passed back into this function.</p> <p>This argument is a URI that the PAL code should recognize and open. A handle must be returned back to the Stack from this function. The handle will be passed along to the <code>a2b_LogCloseFunc</code> and <code>a2b_LogWriteFunc</code> functions.</p> <p>NOTE: The URI passed into <code>a2b_seqChartStart()</code> can be a pointer to any object, not just a constant string. For systems without an underlying filesystem, a common trick is to pass a pointer to a “logging” structure that contains an application level log buffer. This pointer should then be returned to the Stack by this function as the handle. The Stack will then forward the pointer to the logging structure to the <code>a2b_LogWriteFunc</code> and <code>a2b_LogCloseFunc</code> functions whenever any sequence chart data needs to be written.</p>
a2b_LogCloseFunc	Closes the device handle returned by <code>a2b_LogOpenFunc</code>
a2b_LogWriteFunc	Writes a line of sequence chart data to the device handle returned by <code>a2b_LogOpenFunc</code>

5.1.2 Enabling Sequence Chart in Sample Demo Applications

- Pre-Requisites
 - Set the PATH environment variable for running '`java.exe`'
 - `C:\Program Files (x86)\Java\jre<<xx>>\bin`

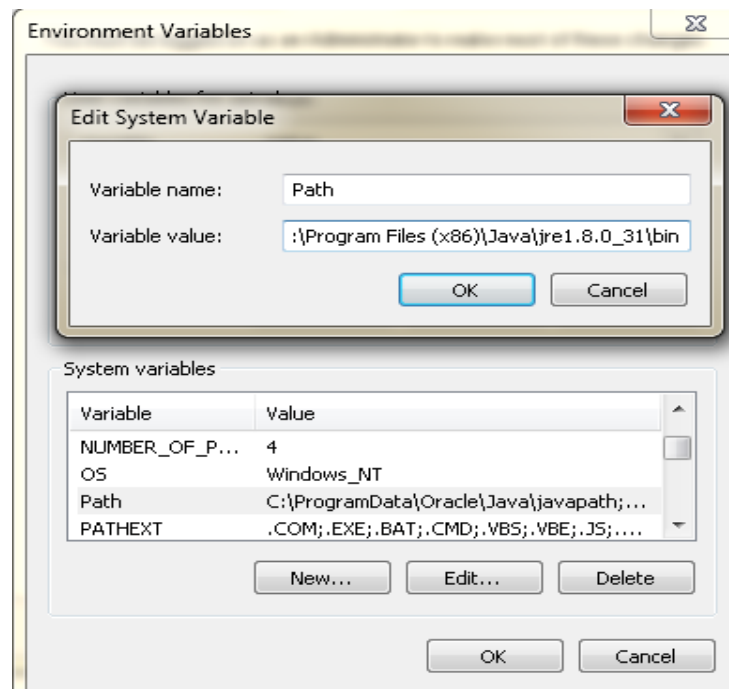


Figure 13: Setting Path in Environment Variables

- Enable 'A2B_FEATURE_SEQ_CHART' macro in '**Target\examples\demo<a2b-xx>\a2bstack-pal\platform\platform\features.h**'
- Build, load and execute the a2bstack application on to the Target in Emulator mode using JTAG (ICE1000/ICE2000).
 - Refer [1] for more details on running sample demo.
- Run the '**Target\tools\SeqChartProcess_<platform>.bat**' once discovery is done and nodes are configured.
- 'SequenceFile.detailed.png' is created in '**Target/examples/demo/<a2b-xx>**' folder

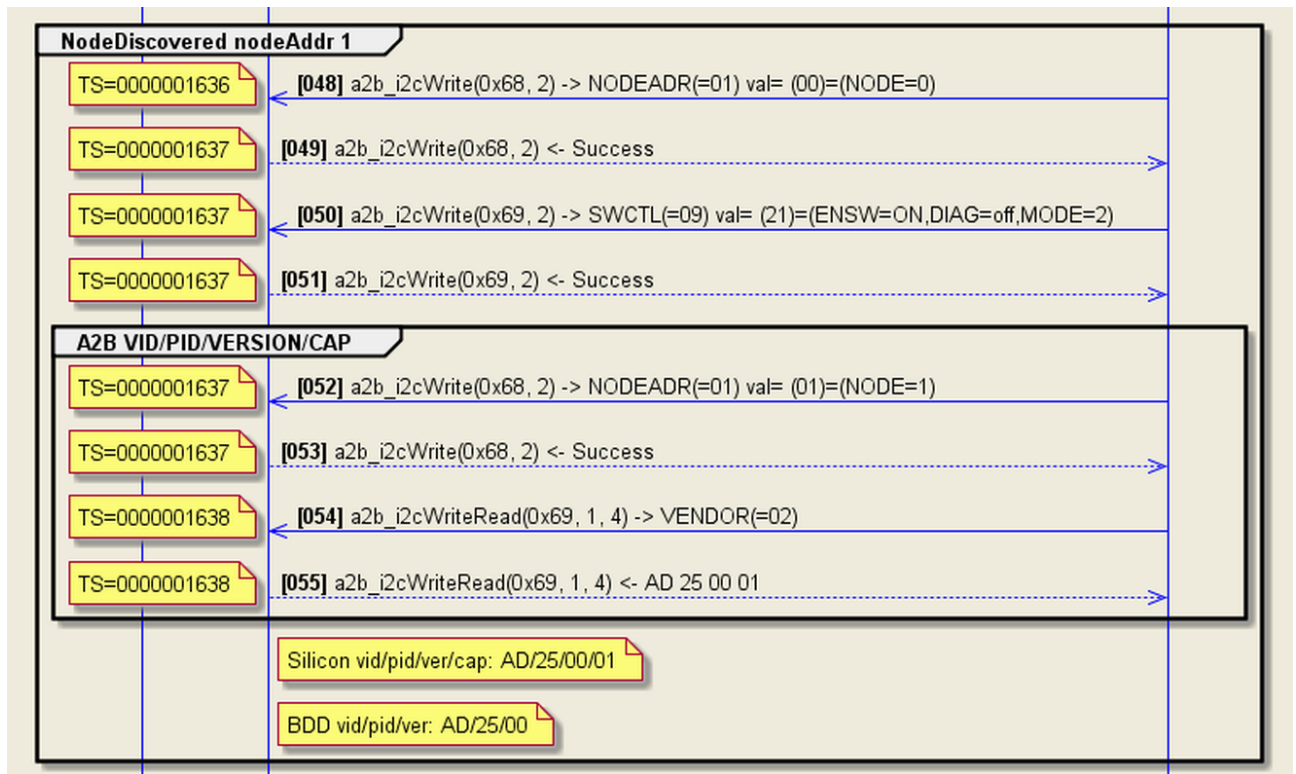


Figure 14: Sample Sequence Chart

5.2 Capturing Trace Messages

In addition to sequence charts, the Stack also provides mechanisms to emit trace messages. Like sequence charts, trace messages are also sent through the logging subsystem of the PAL. It is important that a unique URI be used for trace messages, so the PAL can distinguish between trace messages and sequence charts. Similarly, if one is running two stack contexts concurrently, insure that each context has a unique URI to keep the trace outputs from mixing together.

5.2.1 Trace support in the Stack

- Trace support is an optional Stack feature. Support for tracing must be enabled by defining `A2B_FEATURE_TRACE` in `features.h` prior to compiling the Stack.
- Both the tracing URI as well as the default trace level mask are configured within the Environment Control Block (ECB).
- For tracing, both `traceUrl` and `traceLvl` should be initialized along with the PAL.
- Trace levels can be changed by the application at any time.

5.2.2 Enabling Trace in Sample Demo Applications

- Enable A2B_FEATURE_TRACE macro in '*Target\examples\demo\<a2b-xx>\a2bstack-pal\platform\<a2b-xx>\a2bfeatures.h*'
- Optionally, modify the Trace Level in the Application header file
 - *Target\examples\demo\<a2b-xx>\appla2bapp_defs.h*
- Build, load and execute the a2bstack application on Target in Emulator mode using JTAG (ICE1000/ICE2000).
 - Refer [1] for more details on running sample demo.
- Halt the target in CCES after discovery and nodes are configured (after audio is configured)
- By default, trace messages will be stored in '*Target/a2bstack/demo/<a2b-xx>/a2b_trace.txt*'

Table 8: Trace Levels Description

TRACE LEVELS (Macros)	Description
A2B_TRC_LVL_DEFAULT	Log fatal errors and warnings. This is a combination of A2B_TRC_LVL_WARN, A2B_TRC_LVL_ERROR, A2B_TRC_LVL_FATAL
A2B_TRC_LVL_INFO	Log information wrt A2B node properties, slave plugin processing
A2B_TRC_LVL_DEBUG	Log discovery related messages and interrupts
A2B_TRC_LVL_TRACE1	Log typical function In/Out messages
A2B_TRC_LVL_TRACE2	Log verbose messages
A2B_TRC_LVL_TRACE3	Log Interrupt Mask for Master Plugin
A2B_TRC_LVL_ALL	Log all messages

Table 9: Trace Domains Description

TRACE Domains (Macros)	Description
A2B_TRC_DOM_STACK	Log messages or events from Stack alone. The messages logged will be for a failure case. Hence A2B_TRC_LVL_DEFAULT should be enabled to log these messages.
A2B_TRC_DOM_TICK	Log information with respect to Stack Tick. Enable A2B_TRC_LVL_TRACE2 to log these messages.
A2B_TRC_DOM_TIMERS	Log timer related messages. Enable A2B_TRC_LVL_TRACE1 to log the timer functions.
A2B_TRC_DOM_MSGRTR	Log information with respect to message Request and Notifications. Enable A2B_TRC_LVL_TRACE1 to log the events with respect to message transactions.

A2B_TRC_DOM_PLUGIN	Log messages or events from all Plugins
A2B_TRC_DOM_I2C	Log I2C transactions only. Enable <code>A2B_TRC_LVL_TRACE2</code> to log the I2C transactions.
A2B_TRC_DOM_ALL	Log messages from all domains

5.3 Stack scalability and optimization options

A2B stack is scalable for small micro-controllers to large SoC running complex OS. In this section, we provide options to optimize memory based on few configurations.

1. In '***Target/examples/demo/<a2b-xx>/a2bstack-pal/platform/a2b/features.h***', undefine the following macro(s) :
 - `A2B_FEATURE_COMM_CH`,
 - `ENABLE_PERI_CONFIG_BCF`,
 - `A2B_FEATURE_TRACE`,
 - `A2B_FEATURE_SEQ_CHART`
2. In '***Target/examples/demo/<a2b-xx>/a2bstack-pal/platform/a2b/conf.h***',
 - Set `A2B_CONF_MAX_NUM_MASTER_NODES` to 1
 - Set `A2B_CONF_MAX_NUM_SLAVE_NODES` to 2
3. Remove Slave plugin usage in `a2bapp.c` if no slave peripheral configuration is required.
 - In function `a2bapp_pluginsLoad`, remove `A2B_SLAVE_PLUGIN_INIT(&appPlugins[i])` and set `*numPlugins = 1`;
4. Use compressed BCF export. Refer Section 4.1.1 of A2B SigmaStudio User guide
 - Ensure `ADI_A2B_BCF_COMPRESSED` is defined in `a2bapp_defs.h`

6 Appendix B: Messages

Messages are shared between the Stack, Plugins and the Application to request or notify specific events.

6.1 Request Message

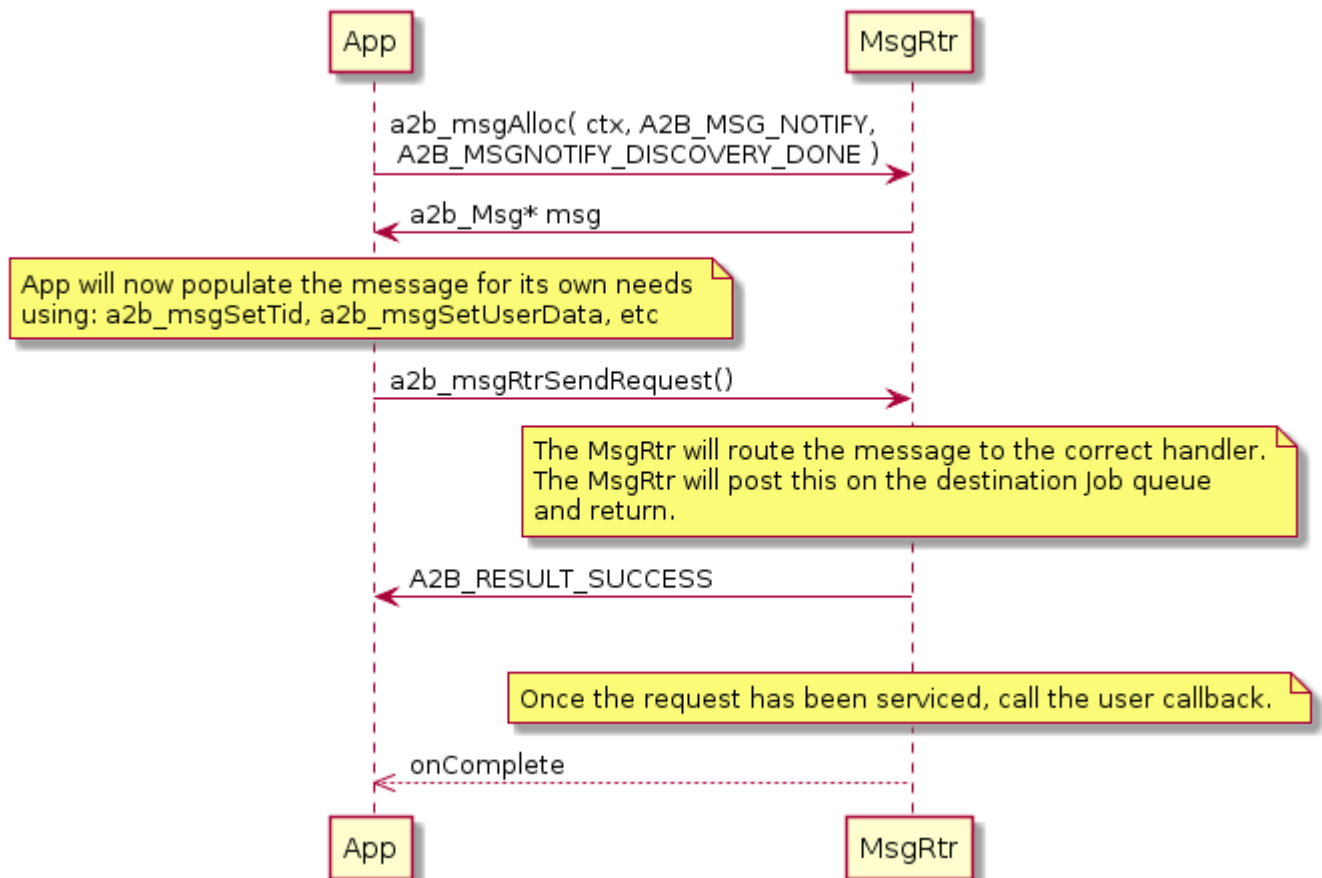


Figure 15: Request Message Example

The list of Request message commands is listed below.

Table 10: Request Message Commands

Commands	Description	Payload Type
<code>A2B_MSGREQ_UNKNOWN</code>	Unknown message request command, typically to indicate error.	-
<code>A2B_MSGREQ_NET_RESET</code>	Reset the A ² B network.	-
<code>A2B_MSGREQ_NET_DISCOVERY</code>	Start A ² B network discovery.	<code>a2b_NetDiscovery</code>

A2B_MSGREQ_NET_DISCOVERY_DIAGMODE	Start A ² B network discovery in diag mode.	Not supported
A2B_MSGREQ_PLUGIN_PERIPH_INIT	Request directed to a slave plugin to complete any necessary initialization of peripherals attached to the slave node.	a2b_PluginInit
A2B_MSGREQ_PLUGIN_PERIPH_DEINIT	Request directed to a slave plugin to de-initialize any peripherals attached to the slave node.	a2b_PluginDeinit
A2B_MSGREQ_PLUGIN_VERSION	Request directed to a master or slave plugin for version and build information about the plugin itself.	a2b_PluginVerInfo
A2B_MSGREQ_CUSTOM	Arbitrary custom command.	-

6.2 Notify Message

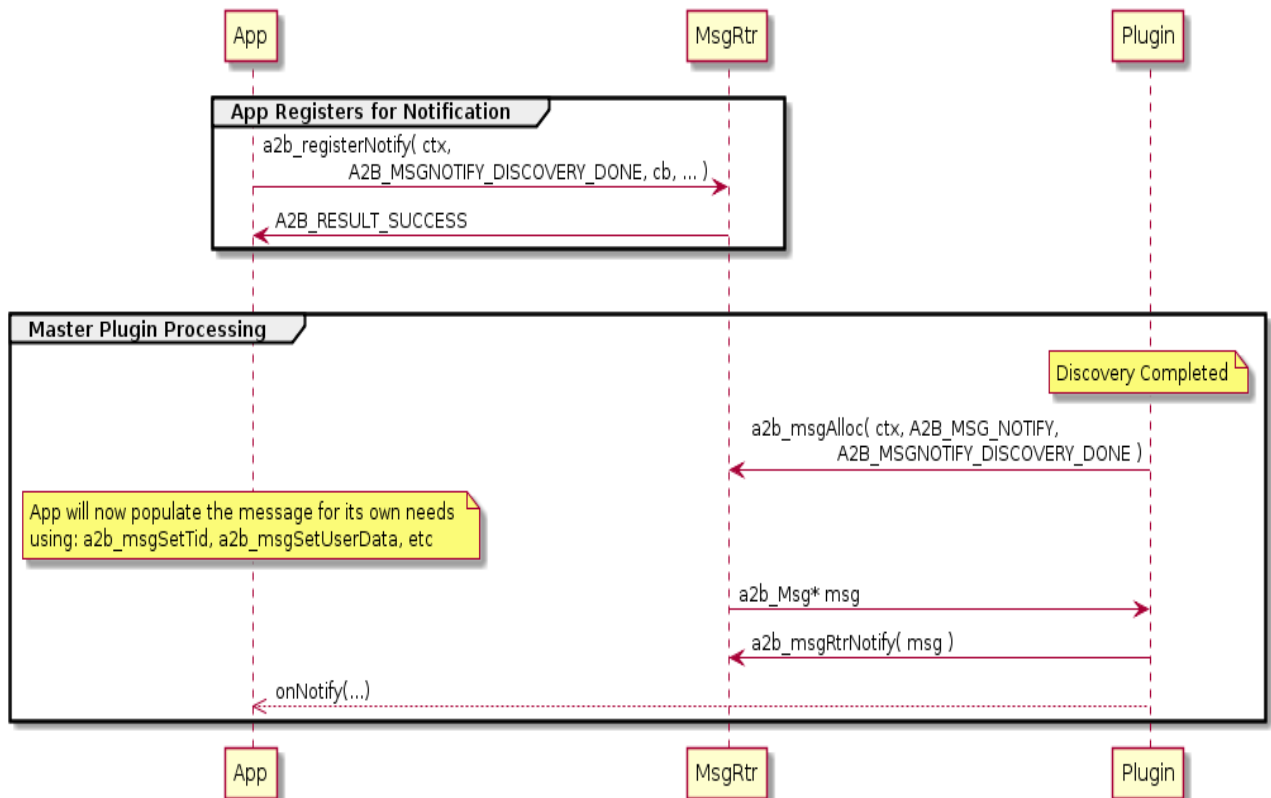


Figure 16: Notify Message Example

The list of Notify Message Commands is listed in Table 11.

Table 11: Notify Message Commands

Notify Message Commands	Description	Payload Type
A2B_MSGNOTIFY_GPIO_INTERRUPT	Notify type used when a plugin triggers a GPIO interrupt. Causes an interrupt notification to be emitted.	a2b_Interrupt
A2B_MSGNOTIFY_POWER_FAULT	Notify type used when a plugin sends a power fault notification.	a2b_PowerFault
A2B_MSGNOTIFY_INTERRUPT	Notify type used when the stack detects any interrupt and emits a notification. This also includes any GPIO related interrupts.	a2b_Interrupt
A2B_MSGNOTIFY_DISCOVERY_DONE	Notification that is emitted at the end of discovery whether it resulted in success or failure.	a2b_DiscoveryStatus
A2B_MSGNOTIFY_CUSTOM	Arbitrary custom command. Anything at or beyond this value is considered a custom command.	-
A2B_MSGNOTIFY_NODE_DISCOVERY	Notify type used when the stack discovers a node OR custom node authentication fails for a particular node.	a2b_Nodedscvry

6.3 Sending custom messages and notifications

To send a custom message, one must allocate the message, find the payload area, deposit the message contents, send the message, and optionally release the reference to the message. Sending a typical message looks like this:

```

#define A2B_MSG_MY_MESSAGE (A2B_MSGREQ_CUSTOM + 1)

struct a2b_Msg *msg;
a2b_HResult result;
a2b_UInt32 *data;
a2b_UInt16 slaveNode = 1;

msg = a2b_msgAlloc(a2b>ctx, A2B_MSG_REQUEST, A2B_MSG_MY_MESSAGE);
if (msg != A2B_NULL)
{
    data = (a2b_UInt32 *)a2b_msgGetPayload(msg);
    *data = 0xABCDABCD;
    result = a2b_msgRtrSendRequest(msg, slaveNode, NULL);
    a2b_msgUnref(msg);
}

```

Code Snippet 8: Sending Custom Message Example

As illustrated in the example code above, the caller unreferences the message following the call to *a2b_msgRtrSendRequest()*. This is because the Stack adds its own reference to the message following the send request. In the event that caller requires that the message live longer, i.e. because it is carrying pointers to data, then one should register a complete callback with the *a2b_msgRtrSendRequest()* and unreference the message there.

The callee of the message is free to modify the contents of the message within the callee's message handler. A callback optionally registered by the caller can be called when the callee completes processing of the message. This callback can be used by the caller to process return values from the callee.

6.4 Receiving custom messages and notifications

Messages are handled exclusively through the Execute method of a plugin. A plugin simply has to have defined entries for custom message within the switch/case statement. The message payload can be extracted from the message using '*a2b_msgGetPayload()*'.

```

#define A2B_MSGNOTIFY_DATA (A2B_MSGREQ_CUSTOM + 1)

struct a2b_MsgNotifier *notifierHandle;

/* Register for notifications from the Remote plugin */
notifierHandle = a2b_msgRtrRegisterNotify(a2b>ctx, A2B_MSGNOTIFY_DATA,
                                          myCallback,
                                          myUserData,
                                          myDestroyFunction);

```

Code Snippet 9: Receiving Custom Message Example

7 Appendix C: Target Debug Features

7.1 Bit Error Rate Test (BERT)

An example application for running BERT after discovery is provided in ‘*ADI_A2B_Software-RelX.Y.Z\Target\examples\advancedapp\bert*’ of the A2B Software package. The macro ‘*A2B_RUN_BIT_ERROR_TEST*’ is enabled in ‘*a2bstack-pal\platform\la2b\features.h*’ to start the BERT test after discovery for a period defined by *A2B_BERT_CALC_PERIOD* and constantly updated after a time period mentioned in *A2B_BERT_UPDATE_PERIOD*. The BERT handler structure must be defined in the Application structure. *ADI_A2B_BERT_HANDLER oBertHandler*.

```
typedef struct
{
    /*! BERT window in micro-seconds */
    a2b_UInt32 nReadTime;

    /*! Test Mode */
    a2b_UInt32 nBERTMode;

    /*! PRBS error counter */
    a2b_UInt32 nPRBSCount[A2B_CONF_MAX_NUM_SLAVE_NODES + 1];

    /*! Counter for various errors */
    a2b_UInt32 nErrorCount[A2B_CONF_MAX_NUM_SLAVE_NODES + 1];

    /*! Auto reset flag */
    a2b_UInt8 bResetFlag;

    /*! Auto reset window in Microseconds */
    a2b_UInt32 nAutoResetWindowTime;

    /*! Read interval counter */
    a2b_UInt32 nCount;

    /*! Overflow flag */
    a2b_UInt32 bOverFlowCount[A2B_CONF_MAX_NUM_SLAVE_NODES + 1];
}ADI_A2B_BERT_HANDLER;

ADI_A2B_BERT_HANDLER oBertHandler;
```

8 Appendix D: Auto Configuration from EEPROM

Auto configuration of slave nodes allows the register and peripheral configuration of a slave node to be programmed from an EEPROM attached to the node during discovery by the Stack. To enable auto configuration feature in the Stack, define the macro *A2B_FEATURE_EEPROM_PROCESSING* in '*Target\examples\demo\<a2b-xx>\a2bstack-pal\platform\<a2b>features.h*'.

The pre-requisite for this feature is that the configuration needs to be programmed in the EEPROM attached to the slave node and the exported bus configuration file from Sigma Studio should have the auto configuration enabled for the node. Refer [2] to configure the same in Sigma Studio.

Terminology

Table 1213: Terminology

Term	Description
A2B	Automotive Audio Bus
A2B node	Refers to AD241x/AD242x.
BCF	Bus Configuration File exported from SigmaStudio.
BDD	Bus Description Data exported from Mentor A2B Analyzer application.
Master Node	A2B transceiver that is connected to the host processor is considered as the master A2B node.
Slave Node	A2B Slave Transceiver with local peripherals such as speakers and microphones.
I2C	Is a multi-master single-ended serial bus used for attaching low-speed peripherals to a processor. In TWI / I2C protocol the serial data transmission is done in asynchronous mode. This protocol uses only two wires named <i>SDA</i> (serial data) and <i>SCL</i> (serial clock) for communicating between two or more ICs.
PAL	Platform Abstraction Layer. The code below this layer is platform specific.

References

Table 14315: References

Reference No.	Description
[1]	AE_09_A2B_QuickStartGuide.pdf
[2]	AE_09_A2B_SigmaStudio_UserGuide.pdf
[3]	AE_09_A2B_Stack_API_Reference.chm
[4]	AE_09_A2B_CommChannel_IntegrationGuide.pdf