UM2354
User manual

ST framework for connecting to Alexa Voice Service, software expansion for STM32Cube

Introduction

This user manual describes the content of the STM32Cube Expansion Package that enables interfacing to the Alexa Voice Service (AVS) from an STM32 application.

The X-CUBE-VS4A Expansion Package for STM32Cube provides application examples that connect STMicroelectronics boards to Amazon servers in order to ease implementation of AVS-oriented products on STM32 devices.

The X-CUBE-VS4A Expansion Package consists of a set of libraries and application examples for STM32F7 Series microcontrollers acting as Alexa-enabled devices.

X-CUBE-VS4A runs on the 32F769IDISCOVERY board. It features a ready-to-run firmware example demonstrating the implementation of a simple smart speaker.

X-CUBE-VS4A offers the following features:

- Board configuration interface
- TCP/IP connectivity
- AVS-protocol encapsulation for the easy implementation of applications
- Application specific services
- STMicroelectronics framework for the Alexa Voice Service
- Creation of AVS-oriented STM32 applications
- Replaceable basic audio acquisition
- Example of a limited audio player

Note: X-CUBE-VS4A does not include software for audio front-end enhancement, neither does it include the complete media player that is needed to be compatible with all different audio services.

Contents

1 General information 6

2 Important note regarding the security 7

3 Package description 8
3.1 General description 8
3.2 Architecture 9
3.3 Folder structure 11

4 Integration in the application 12
4.1 Configuration files 12
4.2 Platform initialization 12
4.3 STVS4A events 14
4.4 STVS4A persistent objects 15
4.5 Service implementation 15
4.5.1 Simple service implementation 16
4.5.2 Threaded service implementation 16
4.5.3 Service with AVS directive/event implementation 16
4.5.4 Send simple AVS event 16
4.5.5 AVS custom event stream 17
4.5.6 Manage the synchronization event state 18
4.6 JSON and JANSSON 19
4.7 Debugging JSON 22

5 Application example 23
5.1 Application description 23
5.2 Alexa Voice Service account 23
5.3 Network setup and authentication 23
5.4 Flash programming 25
5.5 Using the application 26
5.6 Endurance tests 27
5.7 Getting the printf-like traces 28

6 Revision history 29

List of figures

Figure 1. X-CUBE-VS4A software architecture . . . 9
Figure 2. Project file structure . . . 11
Figure 3. STVS4A life cycle . . . 13
Figure 4. Alexa transport protocol . . . 17
Figure 5. Network connections . . . 24
Figure 6. ST-LINK Option Bytes . . . 25
Figure 7. Virtual COM port selection . . . 28
Figure 8. Virtual COM port configuration . . . 28

List of tables

Table 1. List of terms and acronyms . . . 6
Table 2. Minimum code to start an AVS session . . . 13
Table 3. AVS application event handler . . . 14
Table 4. Persistent storage application service callbacks . . . 15
Table 5. Simple event sending to AVS . . . 17
Table 6. Functions allowing to create custom event or stream . . . 18
Table 7. Event creation pseudo-code . . . 18
Table 8. JANSSON string extraction . . . 19
Table 9. JSON event creation example . . . 20
Table 10. JSON event parsing . . . 21
Table 11. Document revision history . . . 29

1 General information

This user manual describes the X-CUBE-VS4A Expansion Package and STVS4A voice-service middleware. It focuses on their use and neither explains the Alexa architecture, nor the creation of an AVS account. Such descriptions are available on the Amazon and developer websites at:

- https://alexa.amazon.com
- https://developer.amazon.com

The X-CUBE-VS4A Expansion Package runs on STM32 32-bit microcontrollers based on the Arm®(a) Cortex®-M processor.

Table 1 presents the definition of terms and acronyms that are relevant for a better understanding of this document.

Table 1. List of terms and acronyms

Term	Definition
Alexa	Amazon's cloud-based voice service
API	Application programming interface
AVS	Alexa Voice Service
BSP	Board support package
DHCP	Dynamic host configuration protocol
EVT	Event
HAL	Hardware abstraction layer
IDE	Integrated development environment
IP	Internet protocol
JSON	JavaScript object notation
STVS4A	STMicroelectronics SDK for designing STM32 -based AVS devices
SDK	Software development kit

.

arm

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Important note regarding the security

Caution: Application developers must take care of security aspects, and put mechanisms in place to protect the tokens and secrets used for the connection to AVS.

The application example provided in the X-CUBE-VS4A Expansion Package does not implement such mechanisms. It only presents a basic implementation for an easy understanding of the stack interface.

3 Package description

This chapter details the content and use of the X-CUBE-VS4A Expansion Package.

3.1 General description

The X-CUBE-VS4A Expansion Package is based on STVS4A, which is a software development kit supporting the design of STM32-based Alexa Voice Service devices. STVS4A features the Service API for the connection to the AVS server and the negotiation of authentication with the server. STVS4A also provides a support to receive directives and send events to the server. In addition, STVS4A features a set of audio support including microphone acquisition and audio playback. STVS4A support can be extended to word-spotting recognition with the addition of an external component.

STVS4A supports Alexa Voice Service API version v20160207.

The following integrated development environments are supported:

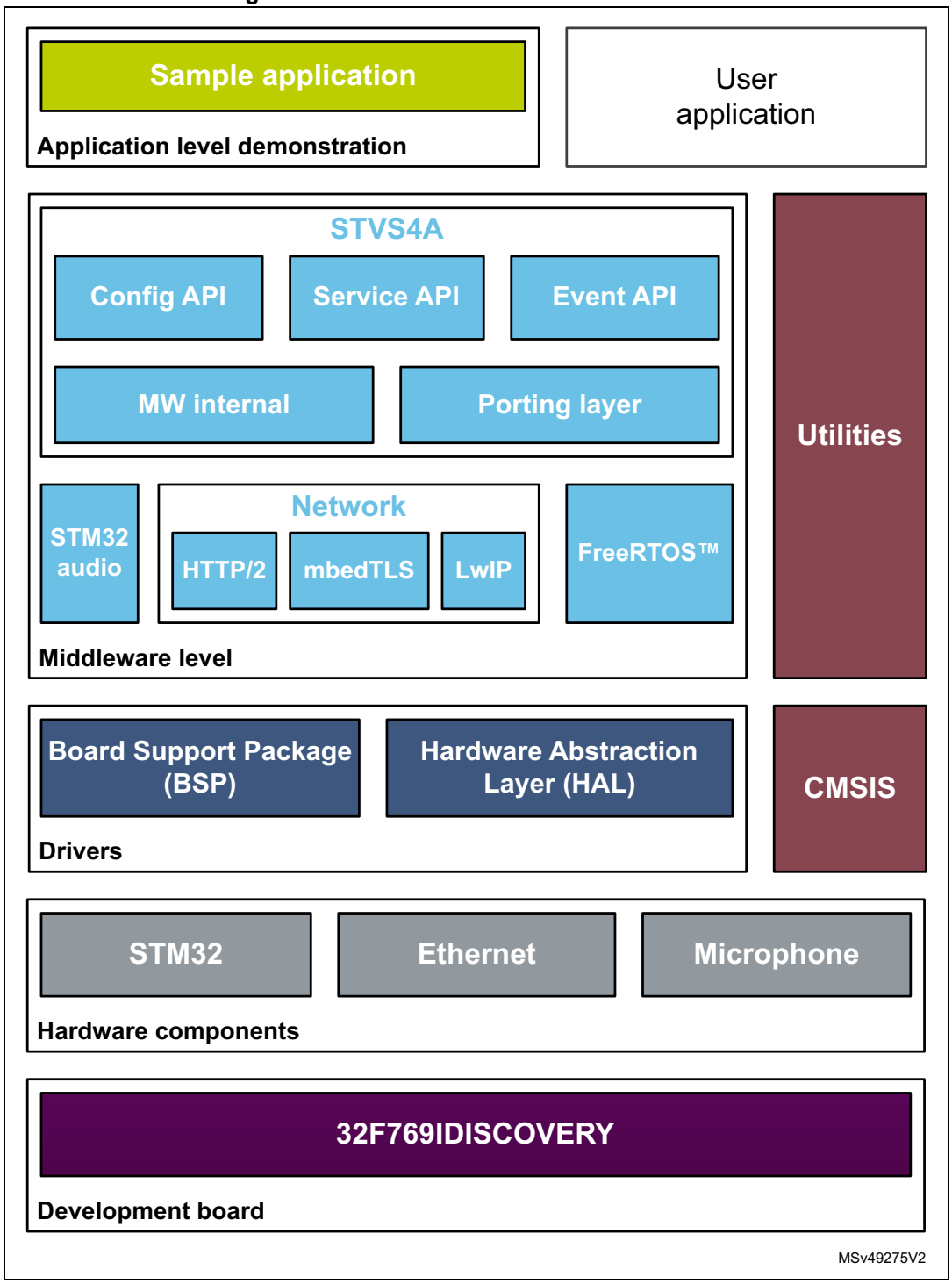
- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® Microcontroller Development Kit (MDK-ARM)
- System Workbench for STM32

Note: refer to the release note available in the delivery package root folder for information about the IDE versions supported.

3.2 Architecture

Figure 1 outlines X-CUBE-VS4A software architecture.

Figure 1. X-CUBE-VS4A software architecture



X-CUBE-VS4A is an Expansion Package for STM32Cube, which:

- Fully complies with STM32Cube architecture
- Expands STM32Cube for the development of AVS-enabled applications
- Relies on the STM32CubeHAL, which is the hardware abstraction layer for STM32 microcontrollers

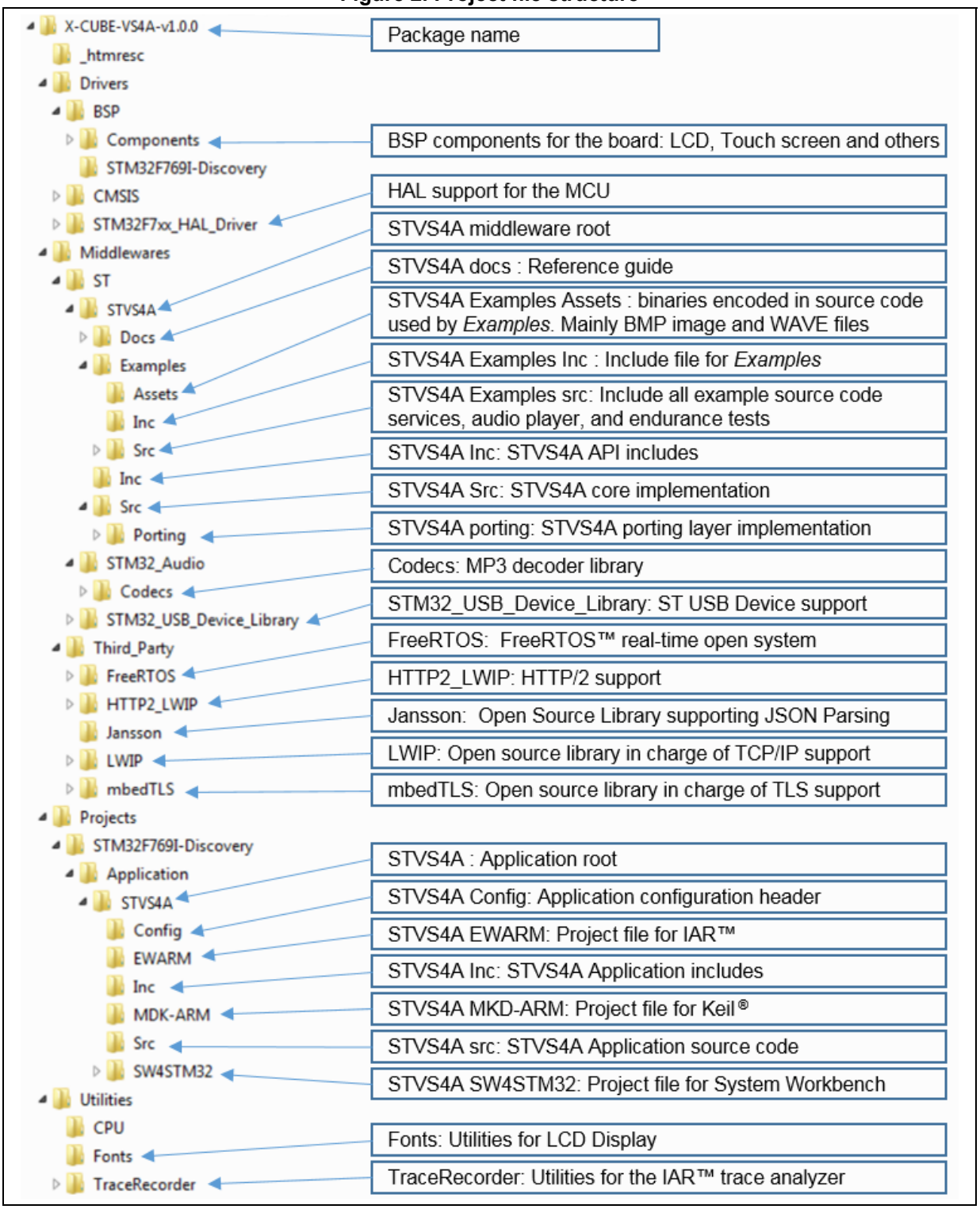
The X-CUBE-VS4A Expansion Package and STVS4A middleware exhibit the following features in addition to those presented in *Section 3.1: General description*:

- STVS4A offers an API for configuration, which is used mainly at initialization time.
- STVS4A offers an API for the customization of some services by the application such as persistent memory access to load and store AVS tokens.
- STVS4A uses an event-driven architecture. After the initialization, STVS4A communicates with the user application using mainly notifications. A notification is a code number with one or several parameters. The parameter is an opaque variant depending on the event. It can be interpreted as an integer, a structure pointer, a handle, or any other meaning. The meaning of the parameter is documented in the event documentation.
- The porting layer is used to customize STVS4A middleware for the application specific environment. The customization ensures the interfacing with potentially different MCUs, networks, and audio interfaces.
- The application and middleware components run in the FreeRTOS™ environment.
- HTTP/2 is the main communication protocol used to interact with an AVS server. X-CUBE-VS4A provides a library to manage this protocol.
- LwIP is in charge of TCP/IP communication services. The encryption is provided by the MbedTLS component.
- STM32Audio provides libraries for audio input and output, as well as MP3 decoding.
- The X-CUBE-VS4A Expansion Package is part of the STM32Cube ecosystem. It relies on its BSP and HAL drivers.

3.3 Folder structure

Figure 2 presents the folder structure of the X-CUBE-VS4A Expansion Package.

Figure 2. Project file structure



4 Integration in the application

The sections in this chapter review the way to integrate the content of the X-CUBE-VS4A Expansion Package in the user application.

4.1 Configuration files

STVS4A middleware relies on other components, such as HTTP/2, LwIP, mbedTLS, and FreeRTOS™:

- HTTP/2 middleware relies on the three other components. It comes as a binary.
- The configuration files for LwIP, mbedTLS, and FreeRTOS™ that have been used during the library generation are provided in the *HTTP2_LWIP* folder as references.

Caution: The user must not modify these configuration files in a way that changes the component APIs.

4.2 Platform initialization

The application is responsible for the initialization of the platform together with all dependencies. This covers for instance such components as:

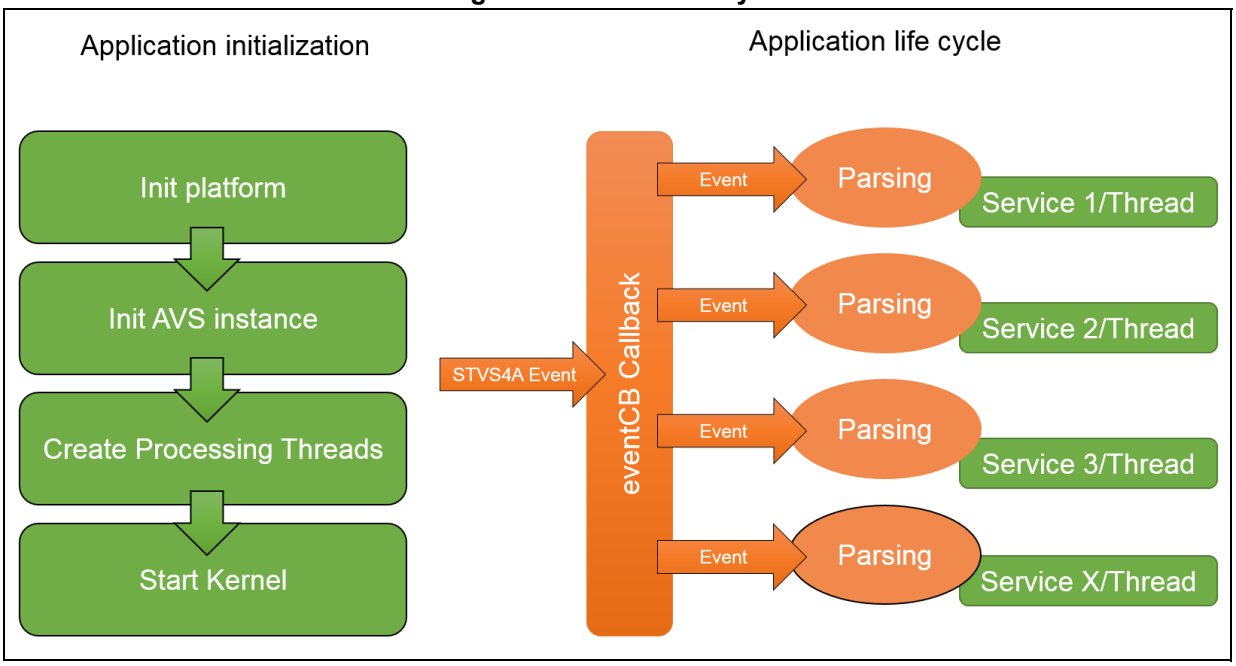
- SystemClock
- GPIO
- MPU_Config
- HAL
- Flash
- Serial

In the demonstration application provided within the Expansion Package, the *platform_init.c* code illustrates the basic initialization for a platform.

STVS4A has no dedicated activity related to board initialization. It assumes that some mandatory modules are initialized. As soon as the platform is initialized, the STVS4A SDK can be initialized as well. The application starts STVS4A initialization using `AVS_Create_Instance()`.

Figure 3 presents an overview of STVS4A life cycle.

Figure 3. STVS4A life cycle



STVS4A initialization consists in filling the Instance Factory with the AVS account IDs and creating a STVS4A instance.

Most of the parameters of the Instance Factory are mandatory only when customizing its standard behavior. Only a few parameters are mandatory for a standard use case while all other are filled with a default values at instance creation. Once these parameters are set, STVS4A starts on the next `osStartKernel()`.

Table 2 shows the minimum code to start STVS4A.

Table 2. Minimum code to start an AVS session

```
platform_Init();
osInitKernel();
AVS_VERIFY(AVS_Init());
sInstanceFactory.clientId     = MY_CLIENT_ID;
sInstanceFactory.clientSecret = MY_CLIENT_SECRET;
sInstanceFactory.productId    = MY_PRODUCT_ID;
sInstanceFactory.eventCB      = appEventHandler;
AVS_VERIFY(hInstance = AVS_Create_Instance(&sInstanceFactory));
osStartKernel();
```

Caution: The Factory must be defined as a global and not as a constant so that it is always in RAM, because STVS4A updates the Factory during the application life cycle. For instance, when STVS4A receives an IP address, the application consequently receives a notification via the callback and the factory is updated with a valid IP address.

Using the simple initialization presented in *Table 2*, Alexa is able to listen and speak as soon as the STVS4A state is changed by means of the function `AVS_Set_State(hInstance, AVS_STATE_START_CAPTURE)`. However, a final product implies more interaction. The

application needs to manage events to support more complex use cases and be capable of interacting fully with the server and local components.

4.3 STVS4A events

A STVS4A event (EVT) is a notification coming from the internal state machine or from the Alexa server. In order to receive events, the application needs to set the `eventCB` field from the `AVS_Instance_Factory` as shown in the code example in *Table 3*.

Table 3. AVS application event handler

```
AVS_Result appEventHandler(AVS_Handle handle, uint32_t pCookie, AVS_Event_t evt,
                           uint32_t  pparam)
{
  return AVS_OK;
}
…
  static AVS_Instance_Factory sInstanceFactory;
  memset(&sInstanceFactory, 0, sizeof(sInstanceFactory));
  sInstanceFactory.clientId     = MY_CLIENT_ID;
  sInstanceFactory.clientSecret = MY_CLIENT_SECRET;
  sInstanceFactory.eventCB      = appEventHandler;
  AVS_VERIFY(hInstance          = AVS_Create_Instance(&sInstanceFactory));
  osStartKernel();
```

STVS4A notifies the application using an EVT via the `eventCB`. The EVT behavior depends on the EVT context. The return value is important since, in some context, it drives the engine response:

- The `AVS_OK` return value means that all is fine and STVS4A executes its normal processing.
- The `AVS_EVT_HANDLED` return value means that the application caught the event and that it is not mandatory for AVS to continue the parsing of events.

The following event-handling use case is provided as an example:

- When STVS4A receives a directive from Alexa, it first sends `EVT_DIRECTIVE`, the parameter being the JSON handle. The application can parse the JSON and manage it directly. STVS4A also parses the JSON and can emit a more detailed EVT concerning this directive. For instance, if the JSON is an alarm event. STVS4A sends `EVT_DIRECTIVE_ALERT` and the application is notified only for `ALERT` directives. If the application catches `EVT_DIRECTIVE` and its handler returns `AVS_EVT_HANDLED`, STVS4A stops the parsing and the `EVT_DIRECTIVE_ALERT` is never received.

EVTs are very sensitive. Since they are synchronous, it is not possible to block their messages for a long period in the callback. All EVTs are dispatched directly from the STVS4A core to the user application. If the application needs to process a heavy job within an EVT, it must delegate the job to a task or create a FreeRTOS™ queue. Otherwise, the STVS4A state management is corrupted, and the overall behavior is impacted.

Most EVTs are re-entrant, meaning that an EVT can be sent from within an EVT. Still, some EVTs are not re-entrant to avoid that local resources used by the system or the event

regenerate other EVTs making the application fall into a dead lock. The EVTs that are not re-entrant start from `EVT_NO_REENTRANT_START` and finishes to `EVT_NO_REENTRANT_END`.

The STVS4A package provides several services that are given as examples showing EVT management.

4.4 STVS4A persistent objects

The AVS architecture manages persistent objects in order to save a context that resists reset. It is the case for tokens and TLS root certificates, which are objects that can be updated during the STVS4A life cycle. After end-user authentication, tokens are obtained from the AVS server and refreshed regularly. The root certificates are obtained from the developer's reference API web sites and can be revoked by Amazon at any time. STVS4A provides a mechanism to store these objects through an application service. For example, this avoids that the full authentication process is run after each reset. The application must store the given information in Flash or in another persistent storage, using a secured mechanism. A single callback is exposed in `AVS_Factory` for this purpose, `persistentCB`.

The `persistentCB` callback manages the persistent storage for different objects. Among the various objects, at least the tokens and the root certificates must be persistent. It is up to the application to manage safely these informations. STVS4A calls `persistentCB` with the `AVS_PERSIST_SET` parameter when the application has to store an object, and `AVS_PERSIST_GET` or `AVS_PERSIST_GET_POINTER` when STVS4A wants to retrieve an object. It is up to the application to manage these events in the appropriate way to serve the API with the right objects. An object takes the form of an array and size to load or store.

In the demonstration provided within the package, the service *service_persistent_storage.c* is the direct illustration of the management of persistent objects using the Flash. The provided example stores in Flash in a non-secure way.

Table 4. Persistent storage application service callbacks

```
platform_Init();
osInitKernel();
AVS_VERIFY(AVS_Init());
  sInstanceFactory.clientId      = MY_CLIENT_ID;
  sInstanceFactory.clientSecret  = MY_CLIENT_SECRET_ID;
  sInstanceFactory.productId     = MY_CLIENT_PRODUCT_ID;
  sInstanceFactory.eventCB       = appEventHandler;
  sInstanceFactory. persistentCB = appPersistCB;
AVS_VERIFY(hInstance = AVS_Create_Instance(&sInstanceFactory));
osStartKernel();
```

4.5 Service implementation

A service is an Alexa application. Typically, a service connects an AVS directive to a device, and the device reports status to AVS using events. There are standard and custom interfaces, which are described from *Section 4.5.1* to *Section 4.5.6*.

4.5.1 Simple service implementation

At least, a service needs to implement an event parser using the `eventCB`. The implementation takes the form of a switch/case where all events you are interested in are caught and processed. In the demo provided with the package, the service *service_serial_trace.c* illustrates such simple interaction.

4.5.2 Threaded service implementation

If a services needs to process information or change the STVS4A state, the application needs to create one or more FreeRTOS™ tasks. These task are able to query or change the STVS4A state, process the information, and notify Alexa about the new status.

In the demonstration provided within the package, the service *service_wakeup.c* is the direct illustration of a basic thread changing the STVS4A state. In this service, the task waits for a a key or a button action to change the STVS4A state from *IDLE* to *Capture* and waits for the end of the dialogue.

4.5.3 Service with AVS directive/event implementation

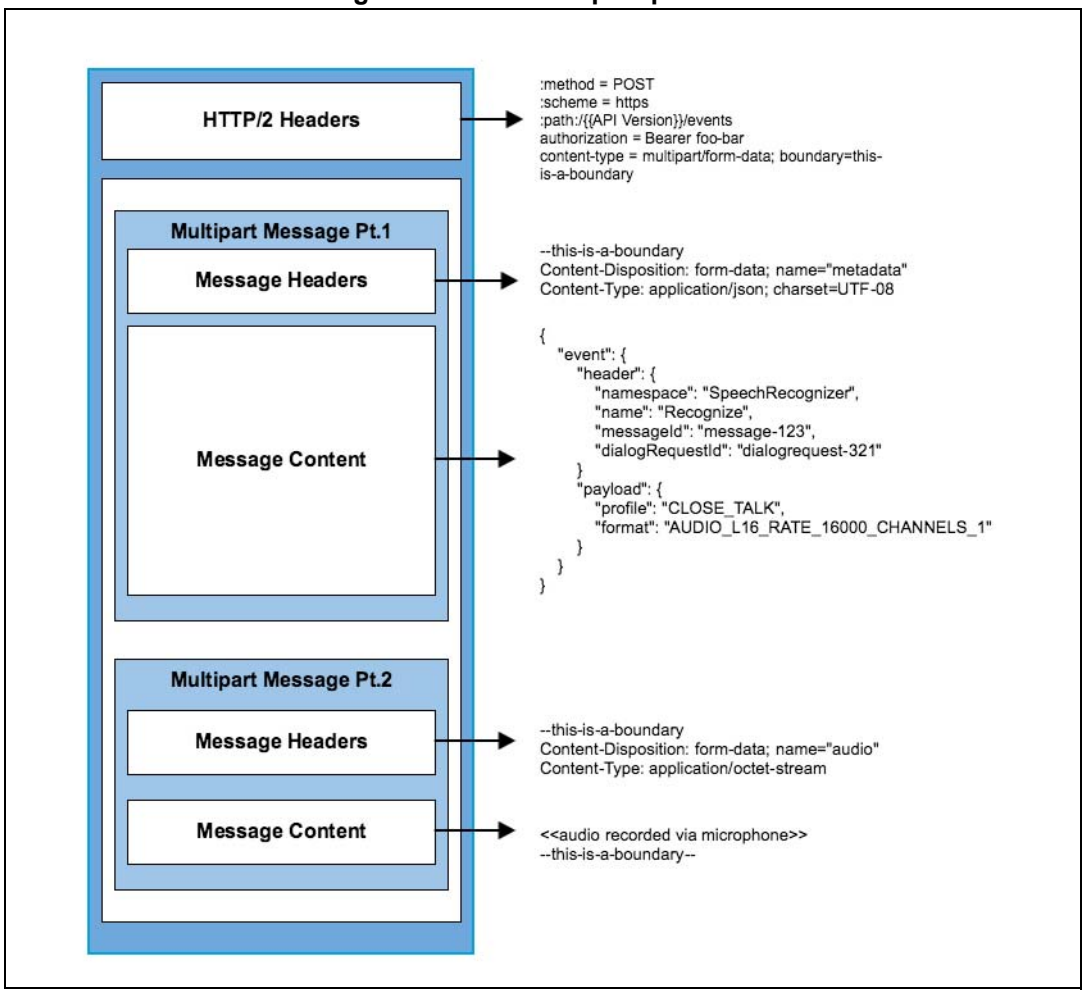
The complex use cases also need an interaction with Alexa in addition to the interaction with STVS4A. Such cases are more sophisticated and require to listen to AVS directives and notify the server about the service states.

Directives coming from AVS are notified using an EVT and the `eventCB` callback. A JSON handle parameter is passed to the application. The parameter is a root `(json_t *)` that can be parsed to extract the information of interest.

4.5.4 Send simple AVS event

Most of the time, Alexa expects a response after a directive. The response takes the form of an AVS event. STVS4A exposes the function `AVS_Send_JSon()` for this purpose. `AVS_Send_JSon` sends a simple event to Alexa using the Alexa transport protocol presented in *Figure 4*.

Figure 4. Alexa transport protocol



1. This illustration is copied from the AVS web site.

`AVS_Send_JSon()` sends a simple event. This means that the *Multipart Message Pt2* is omitted and the stream is closed immediately after the JSON body. *Table 5* shows the sending of a simple event to AVS.

Table 5. Simple event sending to AVS

```
const char *pJson = create_my_event();
   if(AVS_Send_JSon(handle,pJson) != AVS_OK)
   {
     AVS_TRACE_ERROR("Send Json Event");
   }
   free((void *)pJson);
```

4.5.5 AVS custom event stream

If the application needs to create a custom event or stream, STVS4A exposes for this purpose a set of specific functions presented in *Table 6*.

Table 6. Functions allowing to create custom event or stream

```
void *          AVS_Get_Http2_Instance(AVS_Handle hInstance);
AVS_HStream *   AVS_Open_Stream(AVS_Handle hHandle);
AVS_Result      AVS_Add_Body_Stream(AVS_Handle hInstance, AVS_HStream hStream,
                                    const char *pJson);
AVS_Result      AVS_Read_Stream(AVS_Handle hInstance, AVS_HStream hStream,
                                void *pBuffer, uint32_t szInSByte, uint32_t *retSize);
AVS_Result      AVS_Write_Stream(AVS_Handle hInstance, AVS_HStream hStream,
                                  const void *pBuffer, size_t lengthInBytes);
AVS_Result      AVS_Stop_Stream(AVS_Handle hInstance, AVS_HStream hStream);
AVS_Result      AVS_Close_Stream_(AVS_Handle hInstance, AVS_HStream hStream);
AVS_Result      AVS_Process_Json_Stream(AVS_Handle hInstance, AVS_HStream hStream);
const char_t *  AVS_Get_Reponse_Type_Stream(AVS_Handle hInstance,
                                            AVS_HStream hStream);
```

The typical sequence to create a custom event takes the form of the pseudo-code presented in *Table 7*.

Table 7. Event creation pseudo-code

```
/* main HTTP/2 instance */
void *hClient=AVS_Get_Http2_Instance(hInstance);
/* Create the stream & add HTTP2/2 header */
AVS_HStream *hStream = AVS_Open_Stream(hInstance,hClient);
/* Create the Multipart PL1 and send the message content */
AVS_Add_Body_Stream(hInstance,hStream,create_my_event());
/* Prepare multiPart-PL2 */
/* Customize push/pull payload */
AVS_Read_Stream(hInstance,hStream,...);
AVS_Write_Stream/Pull(hInstance,hStream,...);
....
/* End multiPart-PL2 */
AVS_Stop_Stream(hInstance,hStream);
```

4.5.6 Manage the synchronization event state

Some events or services require to send a synchronization status to Alexa. A synchronization status is a reporting of all service contexts. STVS4A exposes a function forcing the engine to post a status as soon as it is possible: `AVS_Post_Sychro()`. It is possible that a synchronization status is requested at any time from the STVS4A core. The synchronization status must reflect the exact context of all services.

Additionally, STVS4A exposes a mechanism to add a custom context to the synchronization status. Before sending the synchronization status and after creating the JSON headers, STVS4A sends the `EVT_SEND_SYNCHRO_STATE` message with the root (`json_t *`) as parameter. Each service is notified and free to update the `contex` JSON array and add its own events before the synchronization event is sent to Alexa by the STVS4A core.

When an event must be sent to Alexa with function `AVS_Send_JSon()`, it is mandatory to add the context state to each message. This is done through the use of function `AVS_Json_Add_Context()` before the conversion of the `(json_t *)` to a string.

In the demonstration provided in the package, the service *service_alarm.c* shows an example of `EVT_SEND_SYNCHRO_STATE` management.

4.6 JSON and JANSSON

The main job implementing a service is to parse and create AVS JSON events. For these purposes, STVS4A uses the JANSSON Open Source API to manage JSON strings.

Information about the JANSSON API is available at http://www.digip.org/jansson/.

The API is simple to use and has a low footprint. STVS4A always gives a root handle JANSSON when it sends directive events to the application. It is then easy to get a JSON string from this handle as shown in *Table 8*.

Table 8. JANSSON string extraction

```
AVS_Result service_alarm_event_cb(AVS_Handle handle, uint32_t pCookie, AVS_Event_t
evt, uint32_t  pparam)
{
If (evt==EVT_DIRECTIVE_ALERT)
    {
    // the json is the parameter
    json_t *root = (json_t *)pparam;
    const char *pJson = json_dumps(root, 0);
    AVS_TRACE_INFO(" %s ", pJson) ;
    }
…
}
```

Creating a JSON event is a common task in a service. It implies some mandatory piece of code to create a syntonically perfect JSON string. The piece of code in table *Table 9* illustrates such a creation.

Table 9. JSON event creation example

```
//{
//"event": {
//    "header": {
//        "namespace": "Alerts",
//        "name": "SetAlertSucceeded",
//        "messageId": "{{STRING}}"
//    },
//    "payload": {
//        "token": "{{STRING}}"
//    }
//}
// send an alarm event to AVS
static    void service_alarm_event(AVS_Handle handle,const char *pToken, const char
*pName)
{
   uint32_t err=0;
   json_t *root  = json_object();
   json_t *event = json_object();
   json_t *header = json_object();
   json_t *payload = json_object();
   static char_t msgid[32];
   sprintf(msgid, FORMAT_MESSAGE_ID, messageIdCounter++);
   err |= json_object_set_new(header, "namespace", json_string("Alerts"));
   err |= json_object_set_new(header, "name", json_string(pName));
   err |= json_object_set_new(header, "messageId", json_string(msgid));
   err |= json_object_set_new(payload, "token", json_string(pToken));
   // links
   err |= json_object_set_new(root, "event", event);
   err |= json_object_set_new(event, "header", header);
   err |= json_object_set_new(event, "payload", payload);

   /* Add the context state to the event */
   AVS_VERIFY(AVS_Json_Add_Context(handle, root));

   const char *pJson = json_dumps(root, 0);
   json_decref(root);
   if(AVS_Send_JSon(handle,pJson) != AVS_OK)
   {
     AVS_TRACE_ERROR("Send Json Event");
   }
   free((void *)pJson);
}
```

Parsing a JSON event is also an easy task to perform, for instance when a directive is received from Alexa. Such parsing is illustrated by the piece of code in *Table 10*.

Table 10. JSON event parsing

```
//"directive": {
   //    "header": {
   //        "namespace": "Alerts",
   //        "name": "SetAlert",
   //        "messageId": "{{STRING}}",
   //        "dialogRequestId": "{{STRING}}"
   //    },
   //    "payload": {
   //        "token": "{{STRING}}",
   //        "type": "{{STRING}}",
   //        "scheduledTime": "{{STRING}}",
   //        "assets": [
   //            {
   //                "assetId": "{{STRING}}",
   //                "url": "{{STRING}}"
   //            },
   //            {
   //                "assetId": "{{STRING}}",
   //                "url": "{{STRING}}"
   //            },
   //        ],
   //        "assetPlayOrder": [ {{LIST}} ],
   //        "backgroundAlertAsset": "{{STRING}}",
   //        "loopCount": {{LONG}},
   //        "loopPauseInMilliseconds": {{LONG}}
   //    }
   //}
   // the directive EVT_DIRECTIVE_ALERT manages timer and alarms
   // we can overload this event to add and delete items

 case EVT_DIRECTIVE_ALERT:
   {
     // the json is the parameter
     json_t *root = (json_t *)pparam;
     // parse it to extract information
     json_t *directive = json_object_get(root, "directive");
     json_t *payload    = json_object_get(directive,"payload");
     json_t *header    = json_object_get(directive,"header");
     json_t *scheduledTime = json_object_get(payload,"scheduledTime");
     json_t *token = json_object_get(payload,"token");
     json_t *type = json_object_get(payload,"type");

     json_t *name    = json_object_get(header,"name");
     const char_t *pToken = json_string_value(token);
     const char_t *pTime = json_string_value(scheduledTime);
     const char_t *pType = json_string_value(type);
…
```

4.7 Debugging JSON

STVS4A provides a mechanism to dump JSON events and directives. STVS4A can dump JSON script on the serial console. By default, this option is disabled. It is enabled with the following piece of code:

```
AVS_Set_Debug_Level(AVS_TRACE_LVL_DEFAULT |
                    AVS_TRACE_LVL_JSON |
                    AVS_TRACE_LVL_JSON_FORMATED);
```

There are two options to visualize the JSON scripts, which are always sent or received in compact mode by Alexa:

- With `AVS_TRACE_LVL_JSON`, JSON script is shown exactly as it was sent or received
- With the addition of `AVS_TRACE_LVL_JSON_FORMATED`, STVS4A formats the JSON script so that it is readable

Note: Using `AVS_TRACE_LVL_JSON_FORMATED` takes time to print and format, and can be intrusive if there are a too many traces at the same time.

5 Application example

This chapter describes the smart-speaker-like application.

Note: The application does not implement the full set of features of the smart speaker.

5.1 Application description

The user can talk to Amazon's Alexa with this application.

He can ask Alexa to sing a song, hear the news, check the weather forecasts, control smart home devices, and more.

5.2 Alexa Voice Service account

Two kinds of account may be used:

- A developer needs an Amazon account to create his device on the Alexa server. Refer to https://developer.amazon.com for details.
  The application example is provided with device identifiers and *secrets* as examples.
- The end-user of the application uses an Amazon account to login (refer to *Section 5.3: Network setup and authentication*), and use the application.

Note: During the development phase, both the developer and the end-user accounts may be the same.

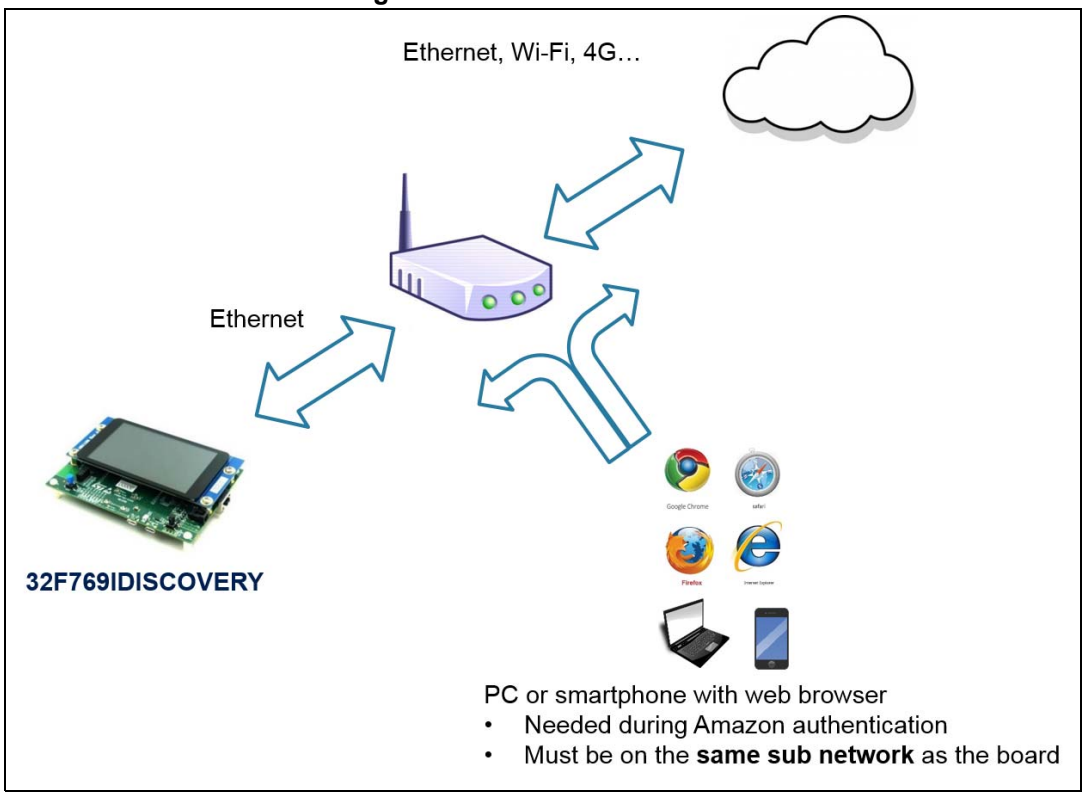
5.3 Network setup and authentication

In order to run the application, the board must be plug to a router that has direct access to the Internet, without proxy.

The router must be configured as DHCP server. The application needs one DHCP server available for setting its IP address.

A PC or a smartphone is used for the authentication phase. *Figure 5* shows the network and authentication environment.

Figure 5. Network connections



The sequence for the authentication phase is:

1. From the PC or smartphone, connect to the STM32 board internal web server
   – http://xxx.xxx.xxx.xxx where xxx.xxx.xxx.xxx is the board IP address. The board IP address is displayed on the user interface after the initialization phase
2. When the navigator is redirected to the Amazon login page, submit valid Amazon credentials
3. Then, the navigator tries to load content from a long URL starting with *http://stvs4a/grant_me?code=*. Replace the *stvs4a* string in the URL by the IP address of the board and send the request
4. After connection, the browser shows a successful message informing that it has got a token, and displays the first characters of the token
5. At this point, the board stores the information needed to connect to AVS server, even after next reset. This information is not the Amazon credentials

Caution: Security aspects are not supported in that example that is assumed to be run in a trusted environment. For a final product, the user must develop and put in place security mechanisms that are aligned with the product environment.

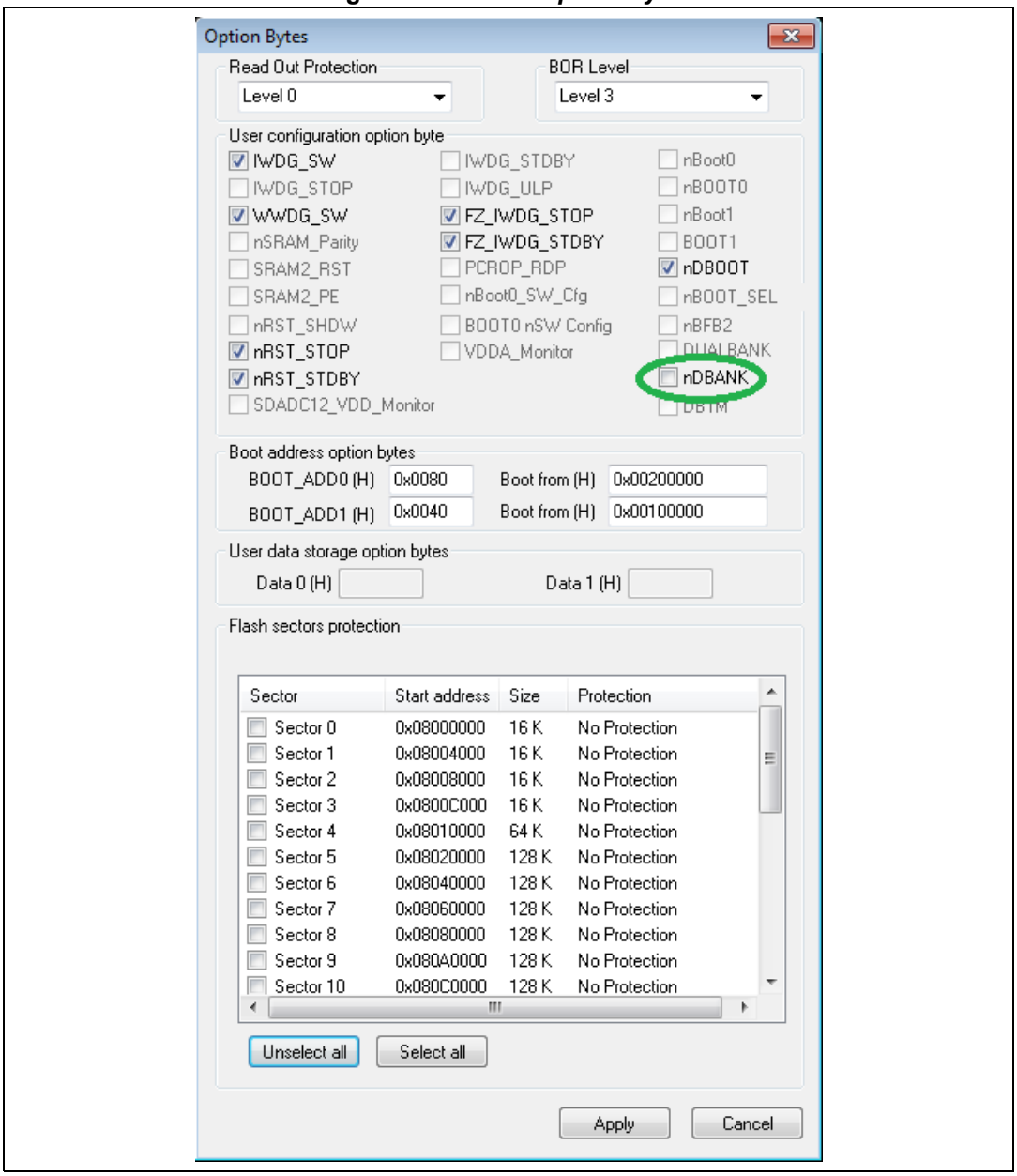
5.4 Flash programming

The application uses the Flash as DUAL BANK. This mode is configured through an option byte as follows:

1. Run the ST-LINK tool
2. Connect to the board using the *Target* menu
3. Display the *Option Bytes* window using the *Target* menu
4. Uncheck *nDBANK* and apply the change if needed

Figure 6 shows the *Option Bytes* window of ST-LINK and the location of the *nDBANK* check box.

Figure 6. ST-LINK *Option Bytes*



The application embeds an endurance test suite. These tests need some data that are stored in the Flash. In most build configurations, the data are stored in the external *QSPI* flash. In such a case, the debugger does not program it and the executable file needs to be programmed through ST-LINK by means of the following sequence of operations:

1. *Erase chip command in the Target menu*
2. *Open File command in the File menu*
3. *Program & Verify command in the Target menu*

Note: If the test assets have not been programmed, the 'Test start' button does not work while other feature do.

5.5 Using the application

When the application is initialized, it is possible to interact with Alexa with questions like:

- "When was Harry Potter published?"
- "What time is it in Tokyo?"
- "How old is Chuck Norris?"

It is also possible to exercise alarm and timer skill with such requests as:

- "Set an alarm for 7:00 AM!"
- "Cancel alarm"
- "Set a timer in two minutes"
- "Set an alarm"

When the request does not contain all needed details to set the alarm, Alexa asks for details.

The volume of the board audio output can be controlled as well as in the following examples:

- "Volume up"
- "Volume down"
- "Set volume to 3"

Unlike a real product, which is generally designed for a precise usage, the application example supports both button-based modes *TAP TO TALK* and *PUSH TO TALK*. The *PUSH TO TALK* mode is also known as *PRESS AND HOLD*. The mode can be selected on the touchscreen user interface.

The blue push button is used for interactions:

- In *TAP TO TALK* mode, tap-and-ask triggers instant Alexa's response
- In *PRESS AND HOLD* mode, press-and-ask triggers instant Alexa's response
- A very short tap on the button is used to stop the alarm buzzer

The touchscreen displays a two-page user interface.

Both user interface pages display:

- A heart that beats when the application is running
- Version information
- An initiator mode button
  - By default the mode is *TAP TO TALK*
  - Pressing this button switches to *PUSH TO TALK* (*PRESS AND HOLD*)
  - Since the wake-up word detection is not integrated in the application, *VOICE INITIATED* works as *TAP TO TALK*
- The IP address of the board
- The internal state (*Idle* when ready to run)
- The date
- The user interface page number to switch between both pages

Additionally, the first user interface page displays the following informations:

- The alarm status displays when an alarm has been set through a vocal command
- The audio player status displays information during music playback

The second user interface page displays the following additional informations and buttons with respect to the common set:

- Endurance test status

  Note: timeouts may happen when the expected answer is not provided fast enough.
- *Test start* button: launches the embedded endurance test suite (data must have been flashed)
- *Set link up/down* button: simulates the Ethernet link loss
- *Network Sim Off* button: removes the network loss simulation from endurance tests

Note: The second user interface page does not exist in some build configuration.

5.6 Endurance tests

The application comes with an endurance test harness, which is implemented as a service.

The endurance tests can be launched by means of a button on the touchscreen, from the second page of the user interface. When the start tests button is pushed, a list of predefined tests are executed. The sequence is managed in the *source file service_endurance_test.c*.

The endurance tests rely on assets stored in Flash memory. Refer to *Section 5.4* for details about Flash programming.

During endurance tests, vocal commands are pushed from audio files (refer to file *service_assets.c*) into the input pipe, instead of being obtained through microphone capture via the BSP.

Network disconnections are also simulated during endurance tests, leading to the display of error notifications and network re-initialization when the specific test is run.

Note: The test engine loops until the stop test button is pushed.

5.7 Getting the printf-like traces

When the board is connected to a PC, a Virtual COM port is created on the PC, which simulates a serial port.

The STVS4A stack embeds printf-like macros to send debugging information to the connected PC. To get those traces on the PC, the serial port must be configured with:

- COM port number
- 921600 baud rate
- 8-bit data
- Parity none
- 1 stop bit
- No flow control

The setting of the Virtual COM port for printf-like tracing is illustrated in *Figure 7* and *Figure 8*.

Figure 7. Virtual COM port selection

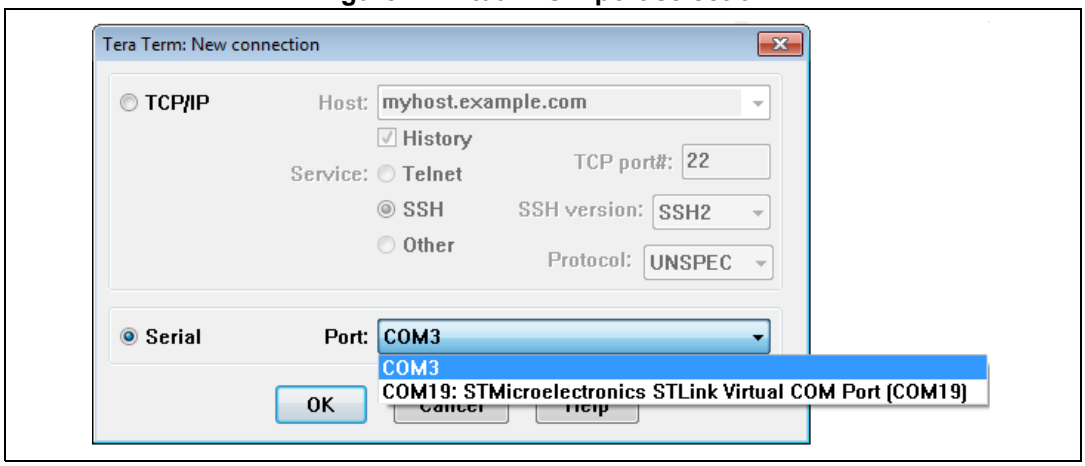
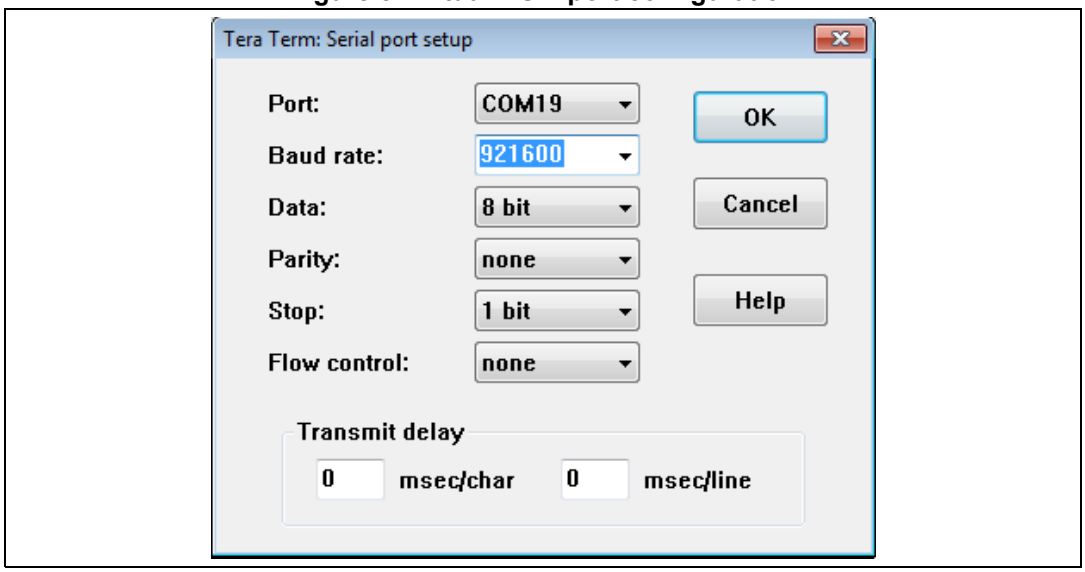


Figure 8. Virtual COM port configuration



6 Revision history

Table 11. Document revision history

Date	Revision	Changes
20-Mar-2018	1	Initial release.
5-Jun-2018	2	Updated *Section 5.4: Flash programming*. Updated STM32Cube Expansion Package reference to X-CUBE-VS4A. Updated voice-service middleware name to STVS4A.