VITIUG

Voice Intelligent Technology Integration User's Guide

Rev. 3 — 10 November 2022 **User guide**

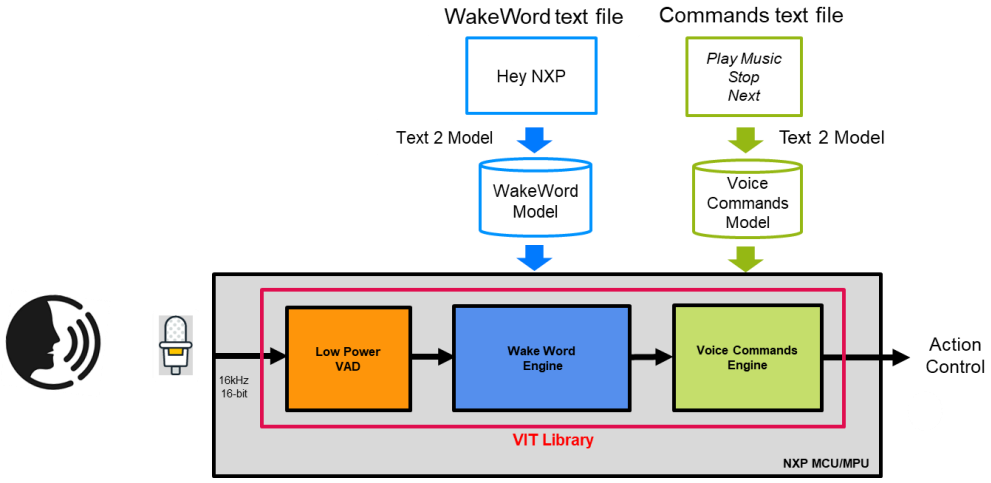
Document information

Information	Content
Keywords	Voice Intelligent Technology
Abstract	The Voice Intelligent Technology (VIT) product provides voice services aiming to wake up and control the IOT devices.

1 Introduction

The Voice Intelligent Technology (VIT) product provides voice services aiming to wake up and control the IOT devices.

The current version of VIT supports a low-power VAD (Voice Activity Detection), a WakeWord Text2Model, and voice commands Text2Model functionalities, see figure below



The WakeWord model and the voice commands model are built from a Text2Model approach, which does not require any audio dataset.

VIT can support the detection of up to 3 WakeWords in parallel.

The VIT library is provided with two models:

- `VIT_Model_en.h` for English support: “Hey NXP” keyword and voice commands.
- `VIT_Model_cn.h` for the Mandarin support: detection of the 你好恩智浦 (Nǐ hǎo NXP, Hello NXP) keyword and voice commands.

Supported commands are listed in the different model files: `VIT_Model.h`.

The role of the low-power VAD is to limit CPU load with minimizing WakeWord / voice command processing in silence conditions.

The enablement of the different features of VIT can be controlled via `VIT_OperatingMode`, see VIT.h file.

Scenario supported by the VIT library (English model example):

- WakeWord detection only: “Hey NXP”.
- WakeWord + voice commands detection. For example, “Hey NXP – Play Music” - “Hey NXP – Next”.

- WakeWord followed by multi commands Voice Command recognitions. For example, “Hey NXP – Play Music – Volume Up”(see Multiturn Voice command feature description in section

The Voice command should be pronounced in a fix time span which should be adapted to the maximum command length. This time span is controlled via `VIT_ControlParams`. See section 4.2.1.4 and section 7 for further details.

VIT returns an “UNKOWN” command if the audio captured after the WakeWord does not correspond to any targeted command.

The VIT library is processing 10ms and 30ms audio frame @16 kHz - 16-bit data

mono.

The VIT library has been ported on 5 cores:

- The VIT lib has been built for Cortex-M7 core and validated on the i.MXRT1050, i.MXRT1060, i.MXRT1160, and i.MX RT1170 platforms.
- The VIT library has been built for HIFI4 core and validated on the IMXRT600 platform.
- The VIT library has been built for FUSIONF1 core and validated on the IMXRT500 platform.
- The VIT lib has been built for Cortex-A53 core and validated on the i.MX8MMini and i.MX8MPlus platforms.
- The VIT lib has been built for Cortex-A55 core and validated on the i.MX93.

Note: *Enabling the LPVAD can impact the first keyword detection, it is dependent on the ambient conditions (silence / noisy).*

The LPVAD decision is maintained during a hangover time of 15 s after the latest burst detection.

2 Acronyms and abbreviations

Table 1. Acronyms and abbreviations

Acronym	Definition
AFE	Audio Front End
VAD	Voice Activity Detection
VCE	Voice Commands Engine
VIT	Voice Intelligent Technology
WWE	Wake Word Engine

3 Release description

The VIT release includes the following files:

- `lib/libVIT_PLATFORM_VERSION.a`: PLATFORM can be either HIFI4, FUSIONF1, Cortex-M4, Cortex-M7, Cortex-A53 or Cortex-A55.
- `lib/VIT.h`: file describes VIT public API.
- `lib/VIT_Model.h`: file contains the VIT model description for the WakeWord and voice commands engines, also this file lists the supported commands.
- `lib/Inc`; folder integrates additional VIT public interface definitions.
- `ExApp/VIT_ExApp.c or ExApp/VIT_alsa_test_app.c` : VIT integration example.

4 Public interfaces description

4.1 Header files

4.1.1 VIT.h

`VIT.h` describes all the definitions required for VIT configuration and usage:

- Operating mode to enable VIT features.
- Detection status enumerator.
- Instance parameters structure.
- Control parameters structure.
- Status parameters structure.
- All VIT public functions.

4.1.2 VIT_Model.h

`VIT_Model.h` contains the model array.

The `VIT_Model` array can be stored in ROM (flash) or RAM.

- If the model is stored in flash, then VIT makes the necessary memory reservation to copy part of the model in RAM before using it : current Cortex-M7 case.
- If the model is stored in RAM, then VIT uses the model directly from its original memory location; HIFI4 and FusionF1 cases.

4.1.3 PL_platformTypes_CortexM.h

`PL_platformTypes_CortexM.h` describes the dedicated platform definition for VIT library.

4.1.4 PL_platformTypes_HIFI4_FUSIONF1.h

`PL_platformTypes_HIFI4_FUSIONF1.h` describes the dedicated platform definition for VIT library.

4.1.5 PL_platformTypes_CortexA.h

`PL_platformTypes_CortexA.h` describes the dedicated platform definition for VIT library.

4.1.6 PL_memoryRegion.h

`PL_memoryRegion.h` describes all the memories definition dedicated to the VIT handle allocation.

4.2 Public APIs

The VIT library present different public functions to control and exercise the library:

- `VIT_SetModel`
- `VIT_GetMemoryTable`
- `VIT_GetInstanceHandle`
- `VIT_SetControlParameters`
- `VIT_Process`
- `VIT_GetVoiceCommandFound`
- `VIT_GetWakeWordFound`
- `VIT_GetLibInfo` (subsidiary interface)
- `VIT_GetModelInfo` (subsidiary interface)
- `VIT_ResetInstance` (subsidiary interface)
- `VIT_GetControlParameters` (subsidiary interface)
- `VIT_GetStatusParameters` (subsidiary interface)

For detailed description of the different APIs (Parameters, return values, and usage), see section 4.1.1.

4.2.1 Main APIs

The main VIT APIs must be called (in the right sequence) in order to instantiate, control, and exercise VIT algorithms.

4.2.1.1 VIT_SetModel

To set the model location: `VIT_ReturnStatus_en VIT_SetModel` (`PL_UINT8* pVITModelGroup`, `VIT_Model_Location_en`).

4.2.1.1.1 Goal

Save the address of VIT model and check whether the model provided is supported by the VIT library.

4.2.1.1.2 Input parameters

To set the input parameters:

- The address of the VIT model in memory.
- The location of the model is in ROM or RAM.

4.2.1.1.3 Output parameters

The output parameter is: None.

4.2.1.1.4 Return value

A value of type is `PL_ReturnStatus_en`. If `PL_SUCCESS` is returned, then:

- VIT model address is saved.

- VIT model is supported by the VIT library.

4.2.1.2 VIT_GetMemoryTable

```
VIT_ReturnStatus_en VIT_GetMemoryTable(VIT_Handle_t            phInstance,
                                       PL_MemoryTable_st       *pMemoryTable,
                                       VIT_InstanceParams_st  *pInstanceParams);
```

4.2.1.2.1 Goal

The goal is to inform the software application about the required memory, needed by the VIT library.

There are 4 kinds of memory are identified:

- Fast data
- Slow data
- Fast coefficient
- Temporary or scratch

4.2.1.2.2 Input parameters

The input parameters are:

1. A pointer to an instance of VIT. It must be a null pointer as instance is not reserved yet.
2. A pointer to a memory table structure.
3. The instance parameter of the VIT library.

4.2.1.2.3 Output parameters

The memory table structure is filled. It informs about the memory size required for each memory type.

4.2.1.2.4 Return value

A value of type `PL_ReturnStatus_en`. If `PL_SUCCESS` is returned, then VIT is succeeding to get memory requirement of:

- Each sub module.
- The VIT model.

4.2.1.3 VIT_GetInstanceHandle

```
VIT_ReturnStatus_en VIT_GetInstanceHandle(
                           VIT Handle t           *phInstance,
                           PL_MemoryTable_st      *pMemoryTable,
                           VIT_InstanceParams_st *pInstanceParams);
```

4.2.1.3.1 Goal

The goal is to set and initialize the instance of VIT before processing call.

All memory is mapped to the required buffer of each sub module.

4.2.1.3.2 Input parameters

The Input parameters are:

1. Pointer to the future instance of VIT.
2. A pointer to the memory table structure. Memory allocation must be done and memory address per memory type has been saved in the table.
3. The instance parameter of the VIT library.

Depending on the value of the instance parameter, sub module initialization is different.

4.2.1.3.3 Output parameters

Address of the VIT instance is set.

4.2.1.3.4 Return value

A value of type is `PL_ReturnStatus_en`. If `PL_SUCCESS` is returned, then:

- VIT instance has been set and initialize correctly.
- VIT model layers are copied in dedicated memory.

4.2.1.4 VIT_SetControlParameters

```
VIT_ReturnStatus_en VIT_SetControlParameters(
                          VIT_Handle_t                phInstance,
                          const VIT_ControlParams_st  *const pNewParams);
```

4.2.1.4.1 Goal

The goal is to set or modify the control parameter of VIT instance.

The new parameters do not set immediately. Indeed, to avoid processing artifact due to the new parameters themselves the update sequence is under internal processing condition and occurs as soon as possible.

4.2.1.4.2 Input parameters

The Input parameters are:

1. VIT handle.
2. Pointer to a control parameter structure: VIT_ControlParams_st
   - OperatingMode: control enablement of the different VIT features (VAD, AFE, VoiceCommand modules)
   - Command_Time_Span: Voice command recognition time span (in second)

Voice command recognition time span must be adapted to the maximum command length targeted.

For operating mode supported, see VIT.h.

4.2.1.4.3 Output parameters

The output parameter is: None.

4.2.1.4.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then control parameter structure has been considered and it has to be effective very soon.

4.2.1.5 VIT_Process

```
VIT_ReturnStatus_en VIT_Process (  VIT_Handle_t           phInstance,
                                   void                   *pVIT_InputBuffers,
                                   VIT_DetectionStatus_en *pVIT_DetectionResults
                                );
```

4.2.1.5.1 Goal

Analyze the audio flow, to detect a “Hot Word” or a voice command.

4.2.1.5.2 Input parameters

The input parameters are:

1. VIT handle.
2. Temporal audio samples (160 or 480 samples @16 kHz – 16-bit data).

4.2.1.5.3 Output parameters

Detection status can have 3 different states:

- `VIT_NO_DETECTION`: No detection.
- `VIT_WW_DETECTED`: WakeWord has been detected.
- `VIT_VC_DETECTED`: A voice command has been detected.

When `VIT_WW_DETECTED` is returned; VIT switches in a voice commands detection phase for a duration controlled by the Command_Time_Span.

When `VIT_VC_DETECTED` is returned; `VIT_GetVoiceCommandFound()` shall be called to know which command has been detected.

`VIT_VC_DETECTED` is also indicating the end of the voice command research period and the switch to a WakeWord detection phase until the WakeWord is detected again. For further details, see section 7.

4.2.1.5.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then the process of the new audio frame has successfully been done.

4.2.1.6 VIT_GetVoiceCommandFound

```
VIT_ReturnStatus_en VIT_GetVoiceCommandFound (
                           VIT_Handle_t         pVIT_Instance,
                           VIT_VoiceCommands_t *pVoiceCommand);
```

4.2.1.6.1 Goal

The goal is to retrieve the command ID and name (when present) detected by VIT.

The function must be called only when `VIT_Process()` is informing that a voice command has been detected (*`pVIT_DetectionResults==VIT_VC_DETECTED`).

4.2.1.6.2 Input parameters

The Input parameters are:

1. VIT handle.
2. Pointer to a voice commands struct type.

4.2.1.6.3 Output parameters

`pVoiceCommand` must be filled with the ID and name of the command detected.

The “UNKNOWN” command is returned if VIT does not identify any targeted command during the voice command detection phase.

4.2.1.6.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then `pVoiceCommand` can be considered.

4.2.1.7 VIT_GetWakeWordFound

```
VIT_ReturnStatus_en VIT_GetWakeWordFound (VIT_Handle_t     pVIT_Instance,
                                          VIT_WakeWord_st *pWakeWord);
```

4.2.1.7.1 Goal

Retrieve the WakeWord ID and name (when present) detected by VIT.

The function must be called only when `VIT_Process()` informs that a WakeWord has been detected `(*pVIT_DetectionResults==VIT_WW_DETECTED)`

4.2.1.7.2 Input parameters

3. VIT Handle

4. Pointer to a WakeWord struct type

4.2.1.7.3 Output parameters

`pWakeWord` will be filled with the ID and name of the command detected.

4.2.1.7.4 Return value

A value of type `PL_ReturnStatus_en`. If `PL_SUCCESS`, then `pWakeWord` can be considered.

4.2.2 Secondary APIs

The secondary VIT APIs are not mandatory for good usage of VIT algorithms. They can be used in order to reset VIT in case of discontinuity in the audio recording flow,

See VIT_ResetInstance section 4.2.2.3 description), get information on the VIT library, VIT model and get information on the internal state of VIT.

4.2.2.1 VIT_GetModelInfo

```
VIT_ReturnStatus_en VIT_GetModelInfo (VIT_LibInfo_t *pLibInfo);
```

4.2.2.1.1 Goal

This function returns different information of the VIT library.

4.2.2.1.2 Input parameters

The input parameter is: Pointer to a `VIT_LibInfo` structure.

4.2.2.1.3 Output parameters

`VIT_LibInfo` must be filled with the details on VIT library, see vit.h file.

4.2.2.1.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then *`pLibInfo` can be considered.

4.2.2.2 VIT_GetModelInfo

```
VIT_ReturnStatus_en VIT_GetModelInfo (VIT_ModelInfo_t *pModel_Info);
```

4.2.2.2.1 Goal

This function returns different information of the VIT model registered within VIT library. The function must be called only when `VIT_SetModel()` is informing that the model is correct (`ReturnStatus == VIT_SUCCESS`).

4.2.2.2.2 Input parameters

The input parameter is: Pointer to a `VIT_Model_Info` structure.

4.2.2.2.3 Output parameters

`VIT_Model_Info` must be filled with the details on `VIT_Model`, see vit.h file.

4.2.2.2.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then *`pModel_Info` can be considered.

4.2.2.3 VIT_ResetInstance

```
VIT_ReturnStatus_en VIT_ResetInstance(VIT_Handle_t phInstance);
```

4.2.2.3.1 Goal

Reset the instance of VIT with instance parameters saved while `VIT_GetInstanceHandle` is called. The reset does not take effect immediately. Indeed, to avoid processing artifact due to the reset itself the reset sequence is under internal processing condition and occurs as soon as possible.

The `VIT_ResetInstance` function should be called whenever there is a discontinuity in the input audio stream. A discontinuity means that the current block of samples is not contiguous with the previous block of samples.

Examples are:

- Calling the VIT process function after a period of inactivity.
- Buffer underrun or overflow in the audio driver.

After resetting VIT instance, VIT shall be reconfigured (call to `VIT_SetControlParameters()`) before continuing the VIT detection process (`i.e VIT_Process()`).

4.2.2.3.2 Input parameters

The input parameter is: VIT handle.

4.2.2.3.3 Output parameters

The output parameter is: None.

4.2.2.3.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then the reset has been considered and must be effective as soon as possible.

4.2.2.4 VIT_GetControlParameters

```
VIT_ReturnStatus_en VIT_GetControlParameters(
                          VIT_Handle_t          *phInstance,
                          VIT_ControlParams_st *pControlParams);
```

4.2.2.4.1 Goal

Get the current control parameter of VIT instance.

4.2.2.4.2 Input parameters

The input parameters are:

1. VIT handle.
2. Pointer to a control parameter structure.

4.2.2.4.3 Output parameters

The output parameter structure is updated.

4.2.2.4.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then parameter structure must be updated correctly.

4.2.2.5 GET_StatusParameters

```
VIT_ReturnStatus_en VIT_GetStatusParameters(
                          VIT_Handle_t         phInstance,
                          VIT_StatusParams_st *pStatusParams);
```

4.2.2.5.1 Goal

Get the status parameters of the VIT library.

4.2.2.5.2 Input parameters

The input parameters are:

1. VIT handle.
2. Pointer to a status parameter buffer.

4.2.2.5.3 Output parameters

Fill the status parameter structure.

4.2.2.5.4 Return value

A value of type is `PL_ReturnStatus_en`. If it is `PL_SUCCESS`, then the status parameters are valid and can be considered.

4.3 Programming sequence

See Figure 2 for programming sequence.

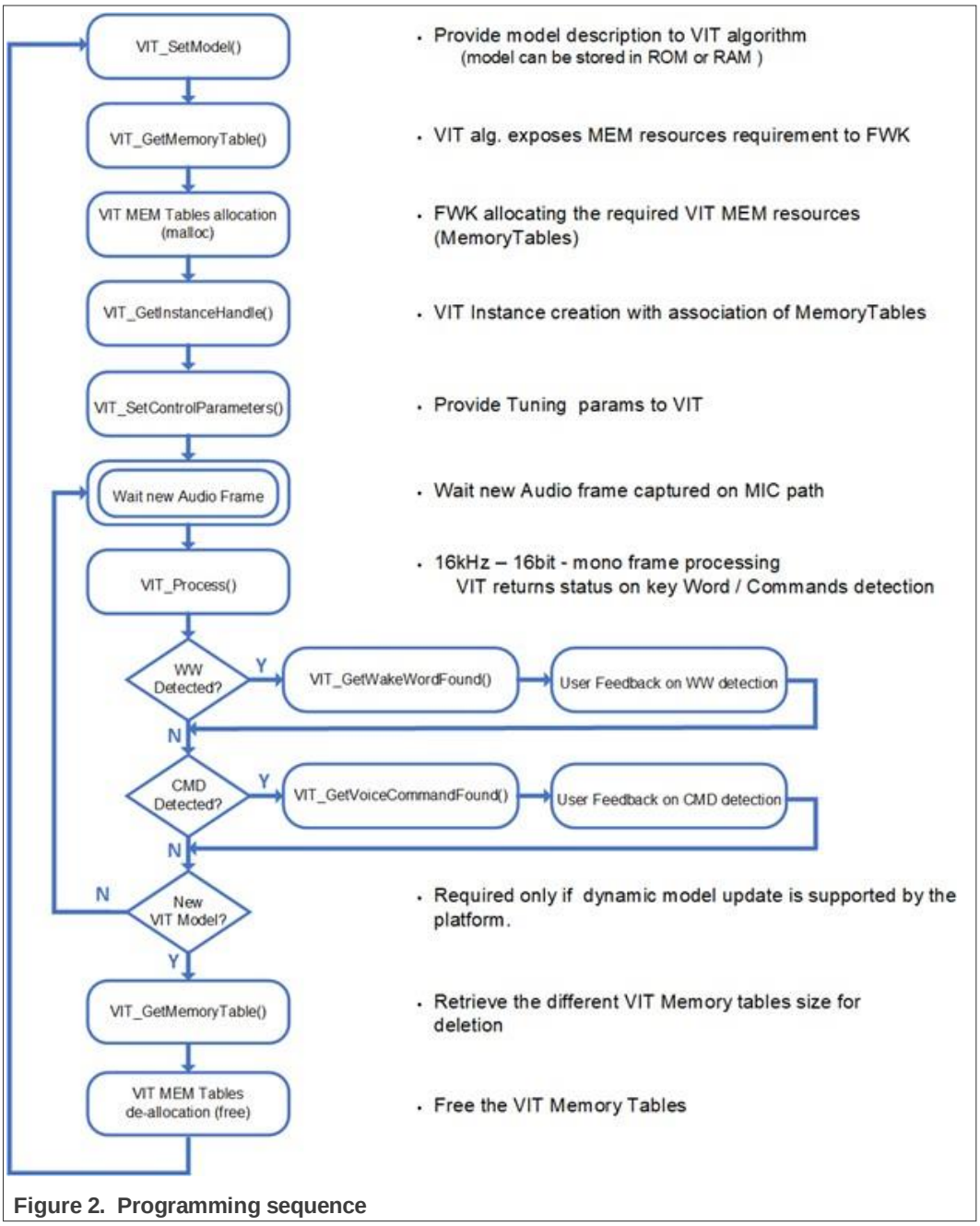


Figure 2. Programming sequence

4.4 Code sample

The code sample in this section aimed to explain the configuration and usage of the main VIT interfaces. See `ExApp.c` (available as part of the SDK project) for details.

4.4.1 Initialization phase

Initialization sequence permits to set an instance of VIT. After initialization sequence, VIT is ready to process audio data. Initialization sequence is in the application code and must respect the following order:

1. Local variable declaration:

```
VIT_Handle_t            VITHandle;        // VIT handle pointer
VIT_InstanceParams_st   VITInstParams;    // VIT instance parameters structure
VIT_ControlParams_st    VITControlParams; // VIT control parameters structure
PL_MemoryTable_st       VITMemoryTable;   // VIT memory table descriptor
PL_ReturnStatus_en      Status;           // status of the function
VIT_VoiceCommands_t     VoiceCommand;
VIT_DetectionStatus_en VIT_DetectionResults = VIT_NO_DETECTION;
                                          // VIT detection result
VIT_DataIn_st           VIT_InputBuffers = { PL_NULL, PL_NULL, PL_NULL }
```

2. Set the instance parameters:

Software application code set the instance parameters of VIT function.

As an example:

```
VITInstParams.SampleRate_Hz    = VIT_SAMPLE_RATE;
VITInstParams.SamplesPerFrame =
VIT_SAMPLES_PER_10MS_FRAME;
VITInstParams.NumberOfChannel = _1CHAN;
VITInstParams.DeviceId         = VIT_IMXRT600;
VITInstParams.APIVersion       = VIT_API_VERSION;
```

3. Set model address:

```
Status = VIT_SetModel(VIT_Model, VIT_MODEL_IN_ROM);
                                  // Pass the address of the VIT Model
```

4. Get memory size and location requirement:

```
Status = VIT_GetMemoryTable(PL_NULL,
                            &VITMemoryTable,
                            &VITInstParams);
```

5. Reserve memory space:

Based on the `VITMemoryTable` information, the software application reserve memory space in the required memory type. The start address of each memory type is saved in `VITMemoryTable` structure.

```
#define MEMORY_ALIGNMENT 4

//Following pseudo code applied to MemType =
//PL_MEMREGION_PERSISTENT_SLOW_DATA, PL_MEMREGION_PERSISTENT_COEF and
//PL_MEMREGION_TEMPORARY
if (VITMemoryTable.Region[MemType].Size != 0)
  {
     pMemory = malloc_in_SLOW_MEMORY (VITMemoryTable.Region[MemType].Size +
               MEMORY_ALIGNMENT);
       VITMemoryTable.Region[MemType].pBaseAddress = (void *) pMemory;
  }
}

//Following pseudo code applied to MemType =
//PL MEMREGION PERSISTENT FAST DATA
```

```
if (VITMemoryTable.Region[MemType].Size != 0)
  {
     pMemory = malloc_in_FAST_MEMORY (VITMemoryTable.Region[MemType].Size +
               MEMORY_ALIGNMENT);
     VITMemoryTable.Region[MemType].pBaseAddress = (void *) pMemory;
  }
}
```

6. Get instance of VIT:

```
VITHandle = PL_NULL;   // force to null address for correct initialization
Status = VIT_GetInstanceHandle(     &VITHandle,
                                    &VITMemoryTable,
                                    &VITInstParams);
```

7. Set control parameters:

Software application code set the new control parameters and call `VIT_SetControlParameters`:

```
VITControlParams.OperatingMode = VIT_WAKEWORD_ENABLE | VIT_VOICECMD_ENABLE;
VITControlParams.MIC1_MIC2_Distance = 0;          // in mm
VITControlParams.MIC1_MIC3_Distance = 0;          // in mm
VITControlParams.Command_Time_Span = 3.0;         // in second
Status = VIT_SetControlParameters( VITHandle,
                                   &VITControlParams);
```

4.4.2 Process phase

For each new input audio frame, `VIT_Process` is called by the application code.

```
Status = VIT_Process( VITHandle,
                      &VIT_InputBuffers,
                      &VIT_DetectionResults );   // VIT detection results
```

Check status of the detection:

```
if (VIT_DetectionResults == VIT_WW_DETECTED)
{
   // a Wakeword detected - Retrieve information :
   Status = VIT_GetWakeWordFound(VITHandle, &WakeWord);
   printf("Wakeword : %d detected \n", WakeWord.WW_Id);

   // Retrieve Wakeword name : OPTIONAL
   // Check first if CMD string is present
   if (WakeWord.WW_Name != PL_NULL)
   {
      printf(" %s\n", WakeWord.WW_Name);
   }
}
```

```
else if (VIT_DetectionResults == VIT_VC_DETECTED)
{
     // a Voice Command detected – Retrieve command information :
     Status = VIT_GetVoiceCommandFound(VITHandle, &VoiceCommand);
     printf("Voice Command : %d detected \n", VoiceCommand.Cmd_Id);

    // Retrieve CMD name : OPTIONAL
    // Check first if CMD string is present
     if (VoiceCommand.Cmd_Name != PL_NULL)
     {
       printf(" %s\n", VoiceCommand.Cmd_Name);
     }
}
else
{
    // No specific action since VIT did not detect anything for this frame
}
```

4.4.3 Delete phase

The framework can delete the environment process / task of VIT with stopping calling `VIT_Process`.

There are no specific VIT APIs in order to free VIT internal memory since the memory allocation is owned by the framework itself (no internal memory allocation).

The framework has to free the memory associated with the different VIT `memoryTables`.

If the framework did not save the `MemoryTables` properties, `VIT_GetMemoryTable` can be called with `VITHandle` in order to retrieve base addresses and size of the different `MemoryTables`.

```
Status = VIT_GetMemoryTable(VITHandle,
                            &VITMemoryTable,
                            &VITInstParams);
// Free memory
for (i = 0; i<PL_NR_MEMORY_REGIONS; i++)
{
    if (VITMemoryTable.Region[i].Size != 0)
    {
        free((PL_INT8 *)VITMemoryTable.Region[i].pBaseAddress);
    }
}
```

4.4.4 Additional code snippet (secondary APIs)

`VIT_GetSatusParameters`

```
VIT_StatusParams_st  VIT_StatusParams_Buffer;
VIT_StatusParams_st* pVIT_StatusParam_Buffer =
 (VIT_StatusParams_st*)&VIT_StatusParams_Buffer;

VIT_GetStatusParameters(VITHandle, pVIT_StatusParam_Buffer,
 sizeof(VIT_StatusParams_Buffer));
printf("\nVIT Status Params\n");
printf(" VIT LIB Release    = 0x%04x\n", pVIT_StatusParam_Buffer-
>VIT_LIB_Release);
```

```
printf(" VIT Model Release = 0x%04x\n", pVIT_StatusParam_Buffer-
>VIT_MODEL_Release);
printf(" VIT Features = 0x%04x\n", pVIT_StatusParam_Buffer-
>VIT_Features_Supported);
printf(" VIT Features Selected = 0x%04x\n", pVIT_StatusParam_Buffer-
>VIT_Features_Selected);
printf(" Nb of channels supported = %d\n", pVIT_StatusParam_Buffer-
>NumberOfChannels_Supported);
printf(" Nb of channels selected = %d\n", pVIT_StatusParam_Buffer-
>NumberOfChannels_Selected);
printf(" Device Selected : device id = %d\n", pVIT_StatusParam_Buffer-
>Device_Selected);
if (pVIT_StatusParam_Buffer->WakeWord_In_Text2Model)
     printf(" VIT WakeWord in Text2Model\n ");
else
     printf(" VIT WakeWord in Audio2Model\n ");
```

4.5 MultiTurn Voice command support

The example above (section 4.4) considers a wake-up word followed by recognition of a single voice command.

VITlib also supports the multi-turn voice command feature, which means that a wake-up word can be followed by multiple recognizable commands.

The amount of time to detect each command is controlled via the `VIT_SetControlParameters` API, see section 4.2.1.4. The end of the multi-turn sequence (for example, from VIT lib back to wake-up word detection) can be freely controlled by the integrator depending on the specific target use case. The end of the multiturn mode can be based on the detection of a specific command (see example below) or after a global timeout.

Considering the stage of the process:

In this example, multiturn will be re-enabled by default after each Voice command is detected and disabled after a specific command is recognized: "START" (`VoiceCommand.Cmd_Id == START_CMD_ID`)

See below the special code that controls the multi-turn sequence, consider the additional code in the gray area at the step of defining the command:

For each new input audio frame, `VIT_Process` is called by the application code.

```
Status = VIT_Process( VITHandle,
                      &VIT_InputBuffers,
                      &VIT_DetectionResults ); // VIT detection results
```

See below the special code that controls the multi-turn sequence, consider the additional code in bold at the command detection phase:

```
if (VIT_DetectionResults == VIT_WW_DETECTED)
{
    // a Wakeword detected – Retrieve information :
    Status = VIT_GetWakeWordFound(VITHandle, &WakeWord);
    printf("Wakeword : %d detected \n", WakeWord.WW_Id);

   // Retrieve Wakeword name : OPTIONAL
   // Check first if CMD string is present
   if (WakeWord.WW_Name != PL_NULL)
   {
         printf(" %s\n", WakeWord.WW_Name);
   }
}
else if (VIT_DetectionResults == VIT_VC_DETECTED)
{
     // a Voice Command detected – Retrieve command information :
     Status = VIT_GetVoiceCommandFound(VITHandle, &VoiceCommand);
     printf("Voice Command : %d detected \n", VoiceCommand.Cmd_Id);

     // Retrieve CMD name : OPTIONAL
```

```
     // Check first if CMD string is present
     if (VoiceCommand.Cmd_Name != PL_NULL)
{
          printf(" %s\n", VoiceCommand.Cmd_Name);
}
//VIT is in command detection phase - we will switch back to WW detection only
// when START
// cmd is detected - otherwise we force to continue in CMD detection mode
if (VoiceCommand.Cmd_Id == START_CMD_ID) // we detect the START cmd here
{
    // back to the default WW/Voice command detection sequence
    VITControlParams.OperatingMode = VIT_WAKEWORD_ENABLE | VIT_VOICECMD_ENABLE;

    VIT_Status = VIT_SetControlParameters(VITHandle, &VITControlParams);
}
else
{
    // force command detection mode (Multiturn voice command mode)
    VITControlParams.OperatingMode = VIT_VOICECMD_ENABLE;

VIT_Status = VIT_SetControlParameters(VITHandle, &VITControlParams);
}

}
else
{
     // No specific action since VIT did not detect anything for this frame
}
```

5 VIT profiling

The profiling example for the English model supports 12 commands (with WW in Text2Model and voice commands in Text2Model). The MHz figures are built from platform measurements.

- VIT figures on RT1060:

Table 2. 1 MIC solution

MHz		Code	Data memory		
Peak	Avg	70 kB	ROM model storage	RAM persistent	RAM scratch
240	156		353 kB	352 kB	47 kB

- VIT figures on RT600:

Table 8. 1 MIC solution

MHz		Code	Data memory	
Peak	Avg	57 kB	RAM model storage	RAM
65	36		353 kB	257 kB

- VIT figures on RT500:

Table 11. 1 MIC solution

MHz		Code	Data memory	
Peak	Avg	32 kB	RAM model storage	RAM
84	46		353 kB	257 kB

VIT stack usage < 2 kB

6 Revision history

Table 13 summarizes the changes done to this document since the initial release.

Revision history

Revision number	Date	Substantive changes
3	20 November 2022	Remove VIT AFE description, add 30ms input frame support
2	10 October 2022	Updated for the next version
1	19 May 2022	Updated the VIT profiling and platform support list corresponding to VIT in SDK2.11.
0	10 September 2021	Initial release

7 Appendix

The example shown in Figure 3 and Figure 4 illustrates the voice command research window: end of voice command utterance shall occur in a ~3 s window from the WakeWord. (for more information on the window size controlled via Command_Time_Span, see section 4.2.1.4)

Example 1:

The voice command utterance is ending 1.7 s after the WakeWord: Once the WakeWord is detected, VIT switches to the voice command research mode. VIT detects the voice command and switches back to the WakeWord detection mode.

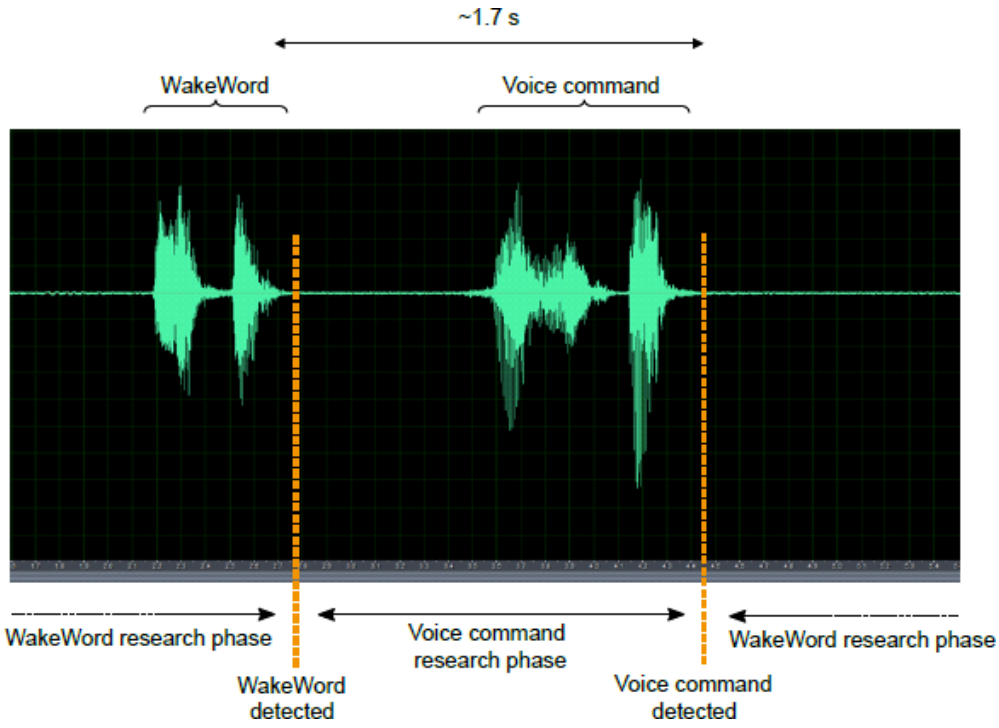


Figure 3. Example 1

Example 2:

The voice command utterance is ending 3 s after the WakeWord: Once the WakeWord is detected, VIT switches to the voice command research mode. VIT would not be able to detect the voice command, since the command is not fitting in the 3 s window. (for more information on the window size, see section 4.2.1.4s)

At the end of the 3 s research window, VIT returns an “UNKNOWN” command and switch back to the WakeWord detection mode.

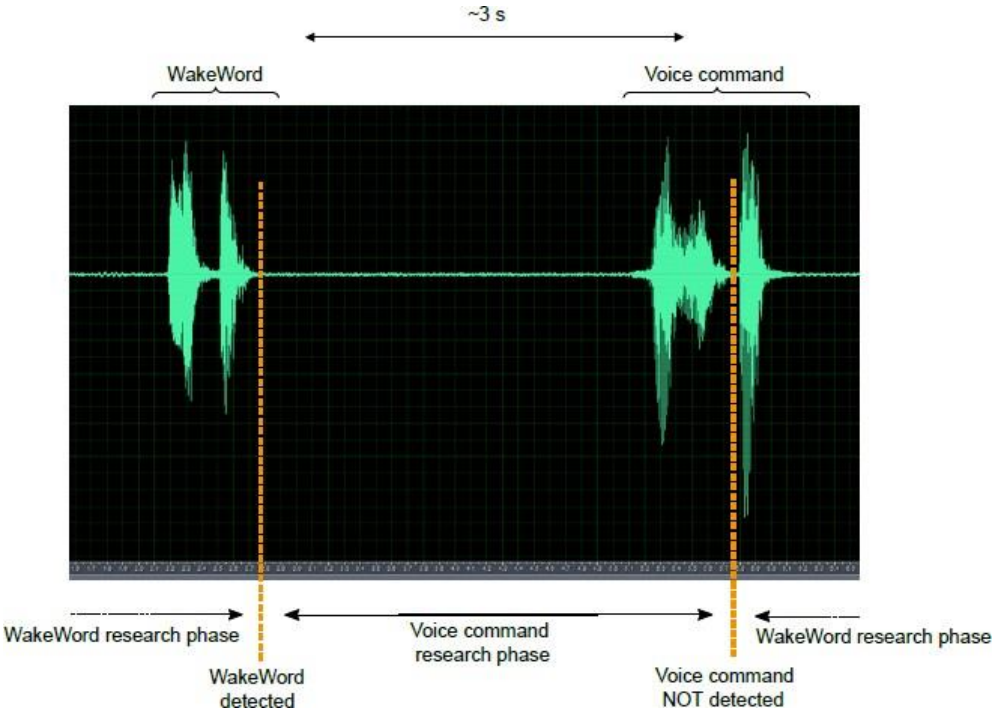


Figure 4. Example 2

8 Legal information

8.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

8.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

8.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

For more information, please visit: http://www.nxp.com

Date of release: 10 October 2022
Document identifier: VITIUG