Voice Intelligent Technology

Integration Guide

Document information	
Title	VIT - Integration Guide
Status	
Version	2.0
Date	December 09, 2021

1 Reference & Abbreviations

References	
[1]	Release Notes

Table 1: Reference documents

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

Table 2: Abbreviations

2 Introduction

Voice Intelligent Technology product is providing Voice services aiming to wake up and control IOT, Home devices.
Current version of VIT is supporting a low power VAD (Voice Activity Detection), 2/3 MICs Audio Front-End, a WakeWord Text2Model and a Voice Commands Text2Model functionalities.

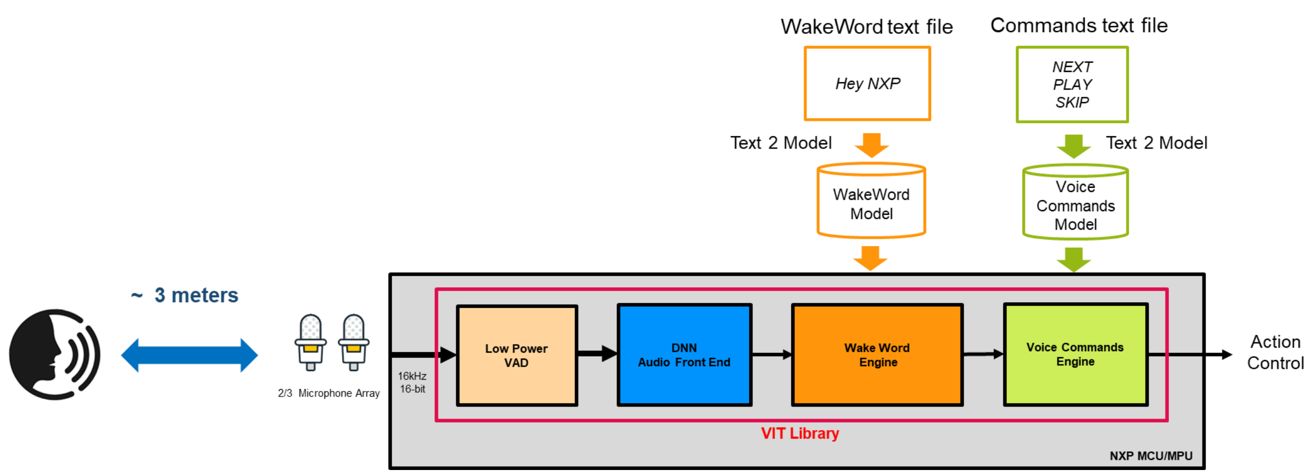


Figure 1 : VIT overview

The role of the Audio Front-End (AFE) is to reduce noise in far-field condition to improve WakeWord and Voice commands detection. The AFE is supporting 2 or 3 Mics.

The WakeWord model and the Voice Commands Model are built from a Text2Model approach which does not require any audio dataset.

The VIT lib is provided with two models:

- VIT_Model_en.h for English support : "Hey NXP" key word and Voice Commands
- VIT_Model_cn.h for the Mandarin support : detection of the *你好 恩 智 浦 (*Nǐ hǎo *NXP : Hello NXP)* key word and Voice Commands.

The commands supported are listed in the different model files : VIT_Model.h.

The role of the Low Power VAD is too 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).

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

- Wake Word detection only : “Hey NXP”
- WakeWord + Voice Commands detection
  E.g. “Hey NXP – Play Music” - “Hey NXP – Next”

The Voice command should be pronounced in a 3 seconds time frame after the Wake word. See Appendix section for further details.
VIT will return an “UNKOWN” command if the audio captured after the WakeWord does not correspond to any targeted command.

The VIT lib is processing 10ms audio frame @16kHz - 16-bit data - mono

The VIT Lib has been ported on 3 cores :

- The VIT lib has been built for Cortex-M7 core and validated on the i.MXRT1060, i.MXRT1160 and i.MX RT1170 platforms.
- The VIT lib has been built for HIFI4 core and validated on the i.MXRT600 platform.
- The VIT lib has been built for FUSIONF1 core and validated on the i.MXRT500 platform.
- The VIT lib has been built for Cortex-M4 core and validated on the i.MX8M mini platform.

Notes 1:
Enabling the LPVAD can impact the first keyword detection, this is dependent on the ambient conditions (silence / noisy). The LPVAD decision is maintained during a hangover time of 15s after the latest burst detection.

AFE support per platform :

- RT1060/RT1160/RT1170/RT600 : 2 and 3 MICs AFE
- RT500 : No AFE

Notes 2:
VIT version running on the i.MX8M mini – Cortex-M4 is integrating only the VAD feature.

Notes 3 : EVK limitation
RT500 (VIT lib No AFE): 1 MIC support
RT600 (VIT lib supporting 3 MICs): only 2 MICs are enabled on EVK
RT1170 (VIT lib supporting 3 MICs): only 1 MIC is enabled on EVK
RT1160 (VIT lib supporting 3 MICs): only 1 MIC is enabled on EVK
RT1060 (VIT lib supporting 3 MICs): only 1 MIC is enabled on EVK

3 Release description

The VIT release is including following files :

- Lib/libVIT_*PLATFORM_VERSION*.a : PLATFORM can be either HIFI4_FUSIONF1 or Cortex-M7
- Lib/VIT.h : file describing VIT public API
- Lib/VIT_Model.h : file containing VIT model description for the WakeWord and Voice Commands Engines – The commands supported are also listed in this file.
- Lib/Inc : folder integrating additional VIT public interface definitions
- ExApp/VIT_ExApp.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 in RAM.

- If the Model is stored in flash, VIT will make 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, VIT will use directly the model from its original memory location. : HIFI4 and FusionF1 case.

4.1.3 PL_platformTypes_CortexM7.h

PL_platformTypes_CortexM7.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_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_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, usage…) – Please refer to the VIT.h file.

4.2.1 Main APIs

The Main VIT APIs has to be called (in the right sequence) in order to instantiate, control and exercise VIT algorithm.

4.2.1.1 VIT_SetModel

VIT_ReturnStatus_en VIT_SetModel (PL_UINT8* pVITModelGroup, VIT_Model_Location_en Location)

4.2.1.1.1 Goal:

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

4.2.1.1.2 Input parameters:

The address of the VIT Model in memory.
The location of the Model (ROM or RAM).

4.2.1.1.3 Output parameters:

None

4.2.1.1.4 Return value:

A value of type 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:

Goal is to inform the SW application about the required memory needed by the VIT library.
4 kinds of memory are identified:

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

4.2.1.2.2 Input parameters:

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.

4.2.1.2.2.1 Output parameters:

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

4.2.1.2.3 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:

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:

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 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 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:

Set or modify the control parameter of VIT instance. The New parameters won't be set immediately. Indeed, to avoid processing artifact due to the new parameters themselves the update sequence is under internal processing condition and will occur as soon as possible.

4.2.1.4.2 Input parameters:

1- VIT Handle
2- Pointer to a control parameter structure : VIT_ControlParams_st
   a. OperatingMode : control enablement of the different VIT features (VAD, AFE, VoiceCommand modules)
   b. MIC1_MIC2_Distance : Distance between MIC2 and the reference MIC (in mm)
   c. MIC1_MIC3_Distance : Distance between MIC3 and the reference MIC (in mm)

MIC1_MIC2_Distance shall be different than zero when AFE 2 Mics is used.
MIC1_MIC2_Distance & MIC1_MIC3_Distance shall be different than zero when AFE 3 Mics is used.

Operating mode supported : `see VIT.h`

4.2.1.4.3 Output parameters:

None

4.2.1.4.4 Return value:

A value of type PL_ReturnStatus_en.

If PL_SUCCESS then control parameter structure has been considered and will be effective as soon as possible.

4.2.1.5 VIT_Process

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

4.2.1.5.1 Goal:

Analyse the audio flow to detect a “Hot Word” or a Voice command.

4.2.1.5.2 Input parameters:

1- VIT Handle
2- Temporal audio samples (160 samples @16kHz – 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 will switch in a Voice commands detection phase for ~3s.

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. See Appendix section for further details.

4.2.1.5.4 Return value:

A value of type PL_ReturnStatus_en.
If 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:

Retrieve the command ID and name (when present) detected by VIT.
The function shall 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:

1- VIT Handle
2- Pointer to a Voice Commands struct type

4.2.1.6.3 Output parameters:

pVoiceCommand will be filled with the ID and name of the command detected.
A "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 PL_ReturnStatus_en. If PL_SUCCESS then pVoiceCommand 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 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:

1- Pointer to a VIT_LibInfo structure

4.2.2.1.3 Output parameters:

VIT_LibInfo will be filled with the details on VIT library.
See VIT.h for further information.

4.2.2.1.4 Return value:

A value of type PL_ReturnStatus_en. If 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 lib. The function shall be called only when VIT_SetModel() is informing that the model is correct. (ReturnStatus == VIT_SUCCESS).

4.2.2.2.2 Input parameters:

2- Pointer to a VIT_Model_Info structure

4.2.2.2.3 Output parameters:

VIT_Model_Info will be filled with the details on VIT_Model.
See VIT.h for further information.

4.2.2.2.4 Return value:

A value of type PL_ReturnStatus_en. If 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 called. The reset doesn't take effect immediately. Indeed, to avoid processing artifact due to the reset itself the reset sequence is under internal processing condition and will occur 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:

VIT Handle

4.2.2.3.3 Output parameters:

None

4.2.2.3.4 Return value:

A value of type PL_ReturnStatus_en.

If PL_SUCCESS then the reset has been considered and will 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:

1- VIT Handle
2- Pointer to a control parameter structure

4.2.2.4.3 Output parameters:

Parameter structure is updated

4.2.2.4.4 Return value:

A value of type PL_ReturnStatus_en.
If PL_SUCCESS then parameter structure has been 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:

1- VIT Handle
2- Pointer to a status parameter buffer

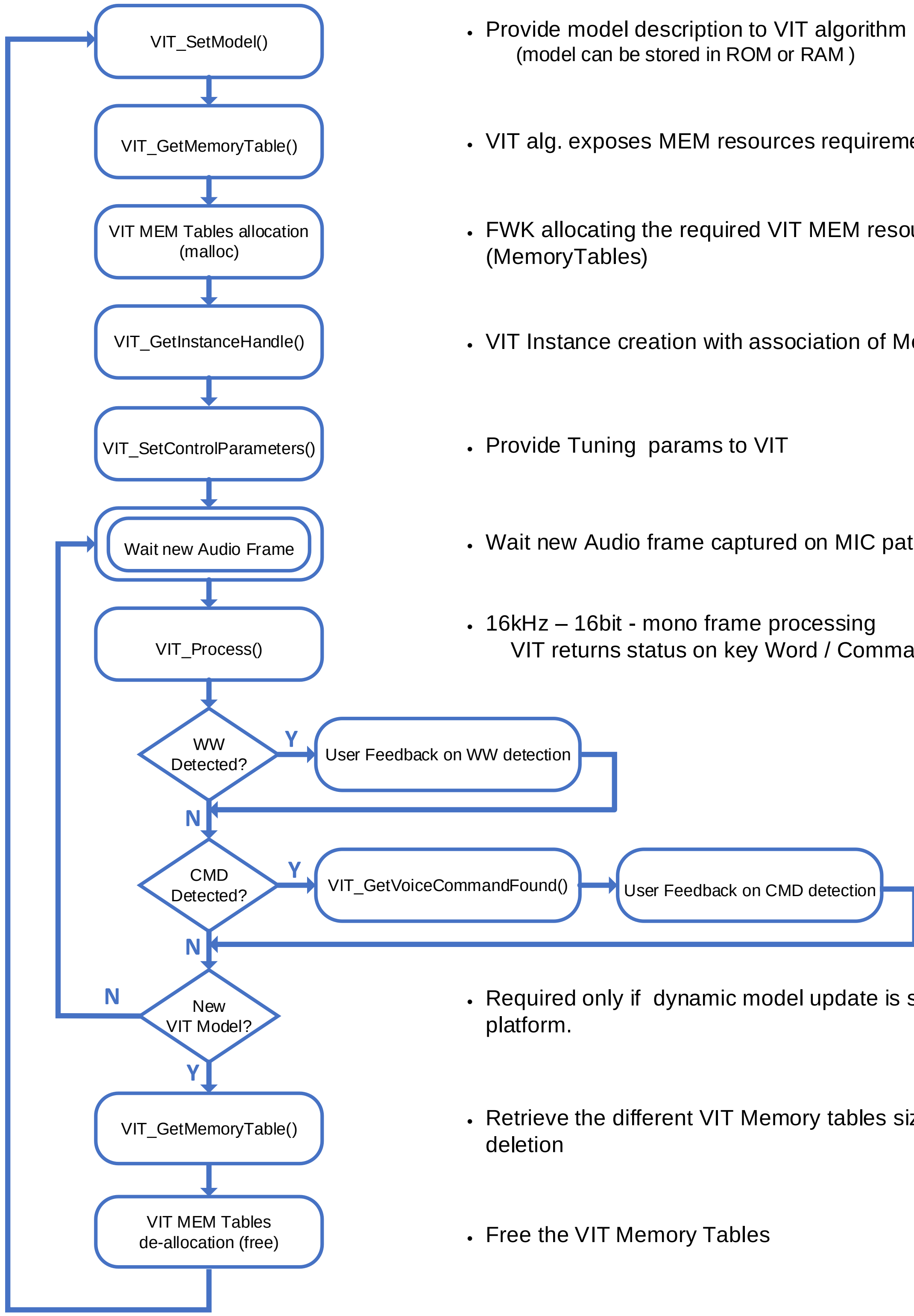
4.2.2.5.3 Output parameters:

Fill the status parameter structure.

4.2.2.5.4 Return value:

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

4.3 Programming sequence



- Provide model description to VIT algorithm
  (model can be stored in ROM or RAM)
- VIT alg. exposes MEM resources requirement to FWK
- FWK allocating the required VIT MEM resources (MemoryTables)
- VIT Instance creation with association of MemoryTables
- Provide Tuning params to VIT
- Wait new Audio frame captured on MIC path
- 16kHz – 16bit - mono frame processing
  VIT returns status on key Word / Commands detection
- Required only if dynamic model update is supported by the platform.
- Retrieve the different VIT Memory tables size for deletion
- Free the VIT Memory Tables

4.4 Code Sample

The code sample below is aimed to explain the configuration and usage of the main VIT interfaces. See ExApp.c for further 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_FRAME;
VITInstParams.NumberOfChannel  = _1CHAN;
VITInstParams.DeviceId         = VIT_IMXRT600;
```

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 informations, 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:

SW application code set the new control parameters and call VIT_SetControlParameters:

```
VITControlParams.OperatingMode = VIT_WAKEWORD_ENABLE | VIT_VOICECMD_ENABLE;
VITControlParams.MIC1_MIC2_Distance = 0;
VITControlParams.MIC1_MIC3_Distance = 0;

Status = VIT_SetControlParameters( VITHandle,
                                   &VITControlParams);
```

4.4.2 Process phase

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

```
VIT_InputBuffers.pBuffer_Chan1 = Audio_Buffer;  //temporal data(10ms @16khz mono 16-bit)
VIT_InputBuffers.pBuffer_Chan2 = PL_NULL;
VIT_InputBuffers.pBuffer_Chan3 = PL_NULL;
```

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

Check status of the detection:

```
 if (VIT_DetectionResults == VIT_WW_DETECTED)
 {
     // WakeWord detected - possible action :
     printf("WakeWord detected \n");
 }
 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 simply stopping calling VIT_Process.
There is 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 will have 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 Enabling VIT Audio Front End.

The example above (section 4.4) is considering VIT configuration for 1 Mic support.

VITInstParams, VITControlParams and InputBuffers shall be adapted to enable AFE with 2 Mics support as follow:

```
VITInstParams.SampleRate_Hz    = VIT_SAMPLE_RATE;
VITInstParams.SamplesPerFrame = VIT_SAMPLES_PER_FRAME;
VITInstParams.NumberOfChannel = _2CHAN;
VITInstParams.DeviceId         = VIT_IMXRT600;

VITControlParams.OperatingMode = VIT_AFE_ENABLE | VIT_WAKEWORD_ENABLE | VIT_VOICECMD_ENABLE;
VITControlParams.MIC1_MIC2_Distance = 63;     //to be changed based on MIC geometry
VITControlParams.MIC1_MIC3_Distance = 0;

VIT_InputBuffers.pBuffer_Chan1 = Audio_Buffer;  //temporal data(10ms @16khz mono 16-bit)
```

```
VIT_InputBuffers.pBuffer_Chan2 = &Audio_Buffer[VIT_SAMPLES_PER_FRAME];
VIT_InputBuffers.pBuffer_Chan3 = PL_NULL;
```

VITInstParams, VITControlParams and InputBuffers shall be adapted to enable AFE with 3 Mics support as follow:

```
VITInstParams.SampleRate_Hz   = VIT_SAMPLE_RATE;
VITInstParams.SamplesPerFrame = VIT_SAMPLES_PER_FRAME;
VITInstParams.NumberOfChannel = _3CHAN;
VITInstParams.DeviceId        = VIT_IMXRT600;

VITControlParams.OperatingMode = VIT_AFE_ENABLE | VIT_WAKEWORD_ENABLE | VIT_VOICECMD_ENABLE;
VITControlParams.MIC1_MIC2_Distance = 63;     //to be changed based on MIC geometry
VITControlParams.MIC1_MIC3_Distance = 45;     //to be changed based on MIC geometry

VIT_InputBuffers.pBuffer_Chan1 = Audio_Buffer;  //temporal data(10ms @16khz mono 16-bit)
VIT_InputBuffers.pBuffer_Chan2 = &Audio_Buffer[VIT_SAMPLES_PER_FRAME];
VIT_InputBuffers.pBuffer_Chan3 = &Audio_Buffer[VIT_SAMPLES_PER_FRAME*2];
```

5 VIT Profiling

Profiling example for a English model supporting 12 commands (with WW in Text2Model and Voice commands in Text2Model). The MHz figures are built from platform measurements and simulations.

5.1 VIT figures on RT1060:

- 1 MIC solution :

MHz		Code	Data Memory		
Peak	Avg		ROM Model storage	RAM persistent	RAM scratch
240	156	70kB	353kB	348kB	47kB

- 2 MICs solution :

MHz		Code	Data Memory		
Peak	Avg		ROM Model storage	RAM persistent	RAM scratch
343	260	70kB	353kB	627kB	47kB

- 3 MICs solution :

MHz		Code	Data Memory		
Peak	Avg		ROM Model storage	RAM persistent	RAM scratch
426	317	70kB	353kB	627kB	47kB

5.2 VIT figures on RT1170:

- 1 MIC solution :

MHz		Code	Data Memory		
Peak	Avg		ROM Model storage	RAM persistent	RAM scratch
200	137	70kB	353kB	348kB	47kB

- 2 MICs solution :

MHz		Code	Data Memory		
Peak	Avg		ROM Model storage	RAM persistent	RAM scratch
286	221	70kB	353kB	627kB	47kB

- 3 MICs solution :

MHz		Code	Data Memory		
Peak	Avg		ROM Model storage	RAM persistent	RAM scratch
332	264	70kB	353kB	627kB	47kB

5.3 VIT figures on RT600:

- 1 MIC solution :

MHz		Code	Data Memory	
Peak	Avg		RAM Model storage	RAM
65	36	57kB	353kB	253kB

- 2 MICs solution :

MHz		Code	Data Memory	
Peak	Avg		RAM Model storage	RAM
168	120	57kB	353kB	532kB

- 3 MICs solution :

MHz		Code	Data Memory	
Peak	Avg		RAM Model storage	RAM
270	198	57kB	353kB	532kB

5.4 VIT figures on RT500:

- 1 MIC solution :

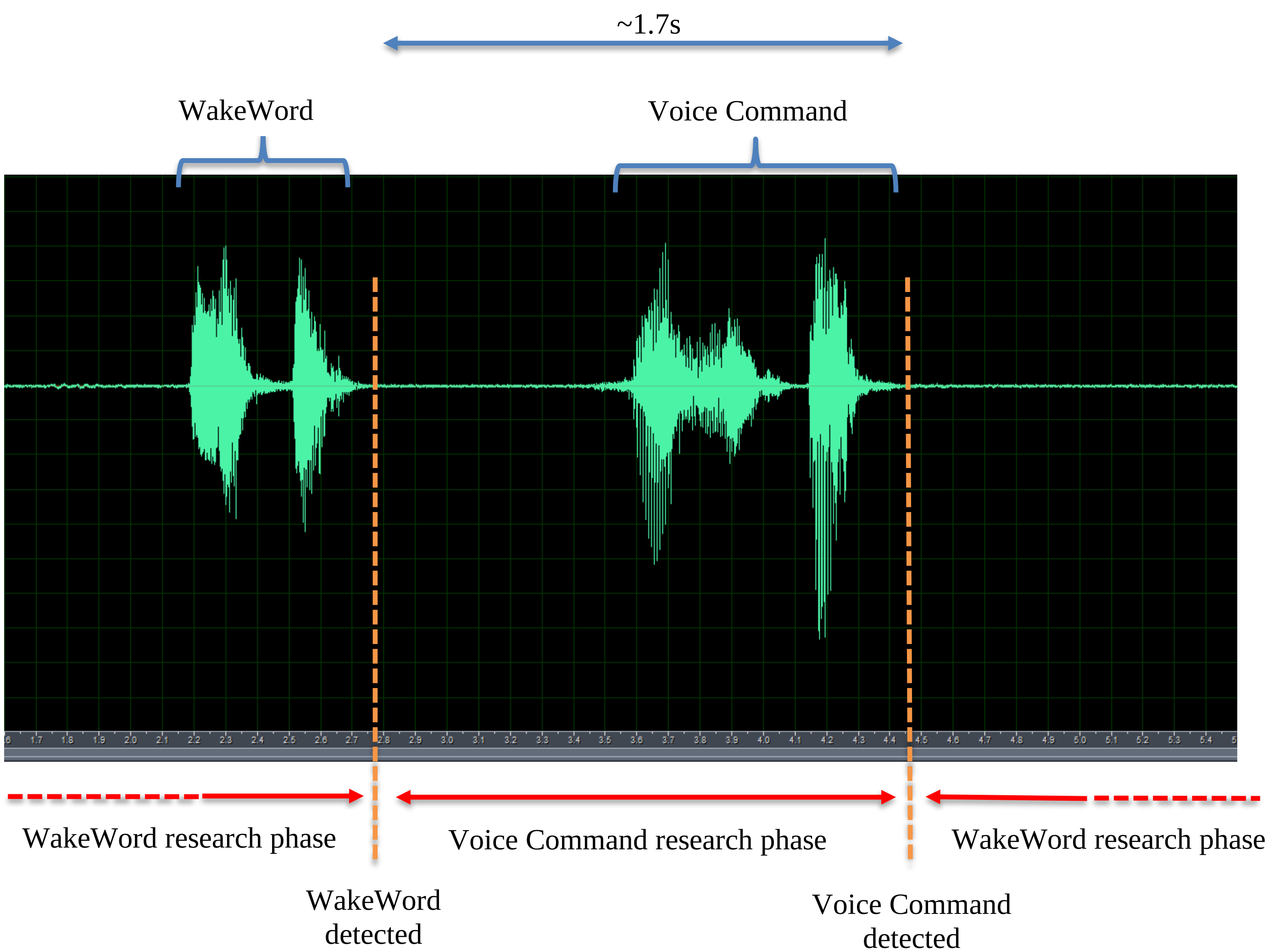
MHz		Code	Data Memory	
Peak	Avg		RAM Model storage	RAM
84	46	32kB	353kB	253kB

VIT Stack usage < 2kB

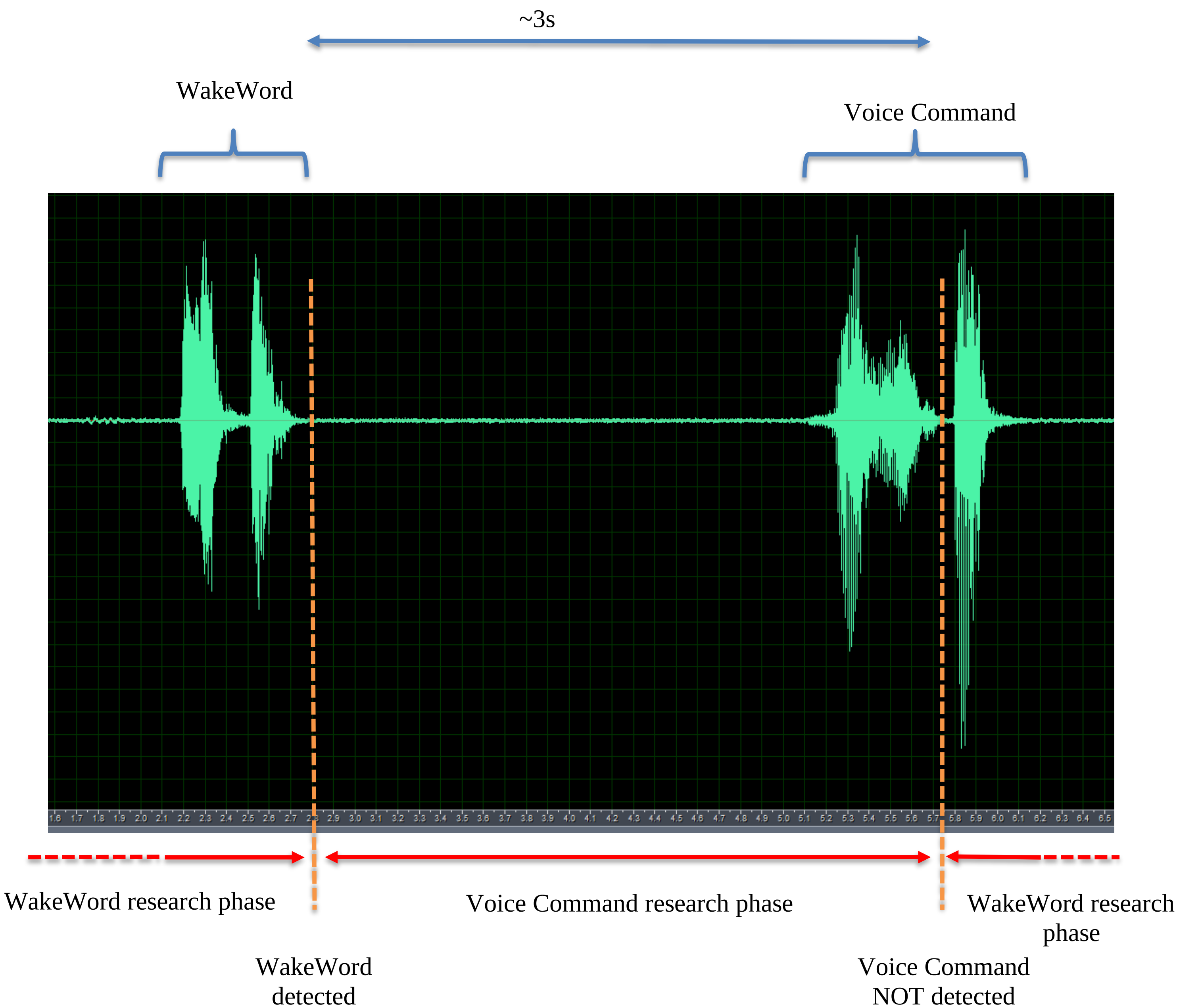
6 Appendix

The example below is illustrating the voice command research window : end of Voice Command utterance shall occur in a ~3s window from the wake word.

Example 1 :
The voice command utterance is ending 1.7s after the WakeWord :
After having detected the WakeWord, VIT will switch to the Voice Command research mode. VIT will detect the Voice command, and switch back to the Wake word detection mode.



Example 2:
The voice command utterance is ending 3s after the WakeWord :
After having detected the WakeWord, VIT will switch to the Voice Command research mode. VIT will not be able to detect the Voice command, since the Command is not fitting in the 3s window.
At the end of the 3s research window, VIT will return an “UNKNOWN” command and switch back to the WakeWord detection mode.



How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

NXP reserves the right to make changes without further notice to any products herein. NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.