Embedded SDK (Software Development Kit)

Voice Recognition (VRLite-1) Library

SDK129/D
Rev. 2, 07/23/2002

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Contents

Chapter 4
Building the VRLite-1 Library

Chapter 5
Linking Applications with the VRLite-1 Library

Chapter 6
VRLite-1 Applications

Chapter 7
License

List of Tables

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Figures

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Examples

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

About This Document

This manual describes the Voice Recognition (VRLite-1) algorithm for use with Motorola's Embedded Software Development Kit (SDK).

Audience

This document targets software developers implementing the voice recognition function within software applications.

Organization

This manual is arranged in the following sections:

- **Chapter 1, Introduction**—provides a brief overview of this document
- **Chapter 2, Directory Structure**—provides a description of the required core directories
- **Chapter 3, VRLite-1 Library Interfaces**—describes all of the VRLite-1 Library functions
- **Chapter 4, Building the VRLite-1 Library**—tells how to execute the system library project build
- **Chapter 5, Linking Applications with the VRLite-1 Library**—describes organization of the VRLite-1 Library
- **Chapter 6, VRLite-1 Applications**—describes the use of VRLite-1 Library through test/demo applications
- **Chapter 7, License**—provides the license required to use this product

Suggested Reading

We recommend that you have a copy of the following references:

- *DSP56800 Family Manual*, DSP56800FM/AD
- *DSP56824 User's Manual*, DSP56824UM/AD
- *Inside CodeWarrior: Core Tools*, Metrowerks Corp.

Conventions

This document uses the following notational conventions:

Typeface, Symbol or Term	Meaning	Examples
`Courier Monospaced Type`	Commands, command parameters, code examples, expressions, data types, and directives	...`*Foundational include files`... ...a data structure of type `vrlite1_sConfigure`...
Italic	Calls, functions, statements, procedures, routines, arguments, file names and applications	...the *pConfig* argument... ...defined in the C header file, *vrlite1.h*... ...makes a call to the *Callback* procedure...
Bold	Reference sources, paths, emphasis	...refer to the **Targeting DSP56824 Platform** manual.... ... see: **C:\Program Files\Motorola\Embedded SDK\help\tutorials**
Bold/Italic	Directory name, project name	...and contains these core directories: ***applications*** contains applications software.... ...CodeWarrior project, ***3des.mcp***, is.....
Blue Text	Linkable on-line	...refer to **Chapter 7**, License...
Number	Any number is considered a positive value, unless preceded by a minus symbol to signify a negative value	3V -10
ALL CAPITAL LETTERS	Variables, directives, defined constants, files libraries	INCLUDE_DSPFUNC #define INCLUDE_STACK_CHECK
Brackets [...]	Function keys	...by pressing function key [F7]...
Quotation marks "... "	Returned messages	...the message, "Test Passed" is displayed.... ...if unsuccessful for any reason, it will return "NULL"....

Definitions, Acronyms, and Abbreviations

The following list defines the acronyms and abbreviations used in this document. As this template develops, this list will be generated from the document. As we develop more group resources, these acronyms will be easily defined from a common acronym dictionary. Please note that while the acronyms are in solid caps, terms in the definition should be initial capped ONLY IF they are trademarked names or proper nouns

DSP Digital Signal Processor or Digital Signal Processing

FE Front End

HMM Hidden Markov Model

I/O Input/Output

LSB	Least Significant Bit
MIPS	Million Instructions Per Second
MSB	Most Significant Bit
OnCE™	On-Chip Emulation
OMR	Operating Mode Register
OOV	Out Of Vocabulary
RA	Rejection Analysis
SDK	Software Development Kit
SDSR	Speaker Dependent Speech Recognition
SRC	Source
VRLite-1	Voice Recognition-1

References

The following sources were referenced to produce this book:

1. *DSP56800 Family Manual*, DSP56800FM/AD
2. *DSP56824 User's Manual*, DSP56824UM/AD
3. *Embedded SDK Programmer's Guide*, SDK101/D

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 1
Introduction

Welcome to Motorola's Family of Digital Signal Processors (DSPs). This document describes the Voice Recognition (VRLite-1) Library, which is a part of Motorola's comprehensive Software Development Kit (SDK) for its DSPs. In this document, you will find all the information required to use and maintain the VRLite-1 Library interface and algorithms.

Motorola provides these algorithms to you for use on the Motorola Digital Signal Processors to expedite your application development and reduce the time it takes to bring your own products to market.

Motorola's Voice Recognition Library is licensed for your use on Motorola processors. Please refer to the standard Software License Agreement in **Chapter 7** for license terms and conditions; please consult with your Motorola representative for premium product licensing.

1.1 Quick Start

Motorola Embedded SDK is targeted to a large variety of hardware platforms. To take full advantage of a particular hardware platform, use **Quick Start** from the **Targeting DSP568xx Platform** documentation.

For example, the **Targeting DSP56824 Platform** manual provides more specific information and examples about this hardware architecture. If you are developing an application for the DSP56824EVM board or any other DSP56824 development system, refer to the **Targeting DSP56824 Platform** manual for **Quick Start** or any other information specific to the DSP56824.

1.2 Overview of VRLite-1

VRLite-1 is a memory-optimized, isolated-word, speaker-dependent speech recognition system. This means that the system must be trained to the voice of a particular user and that it can recognize only isolated words. For example, if the user trained the words "call" and "Bob", the algorithm can recognize both "call" and "Bob", but not the phrase "call Bob".

1.2.1 Background

Many products, such as a mobile phone, require a voice recognition system to operate the phone through voice. VRLite-1 provides a solution for this requirement.

Figure 1-1 shows the entire system in the context of a product and emphasizes the scope of the VRLite-1 software algorithm.

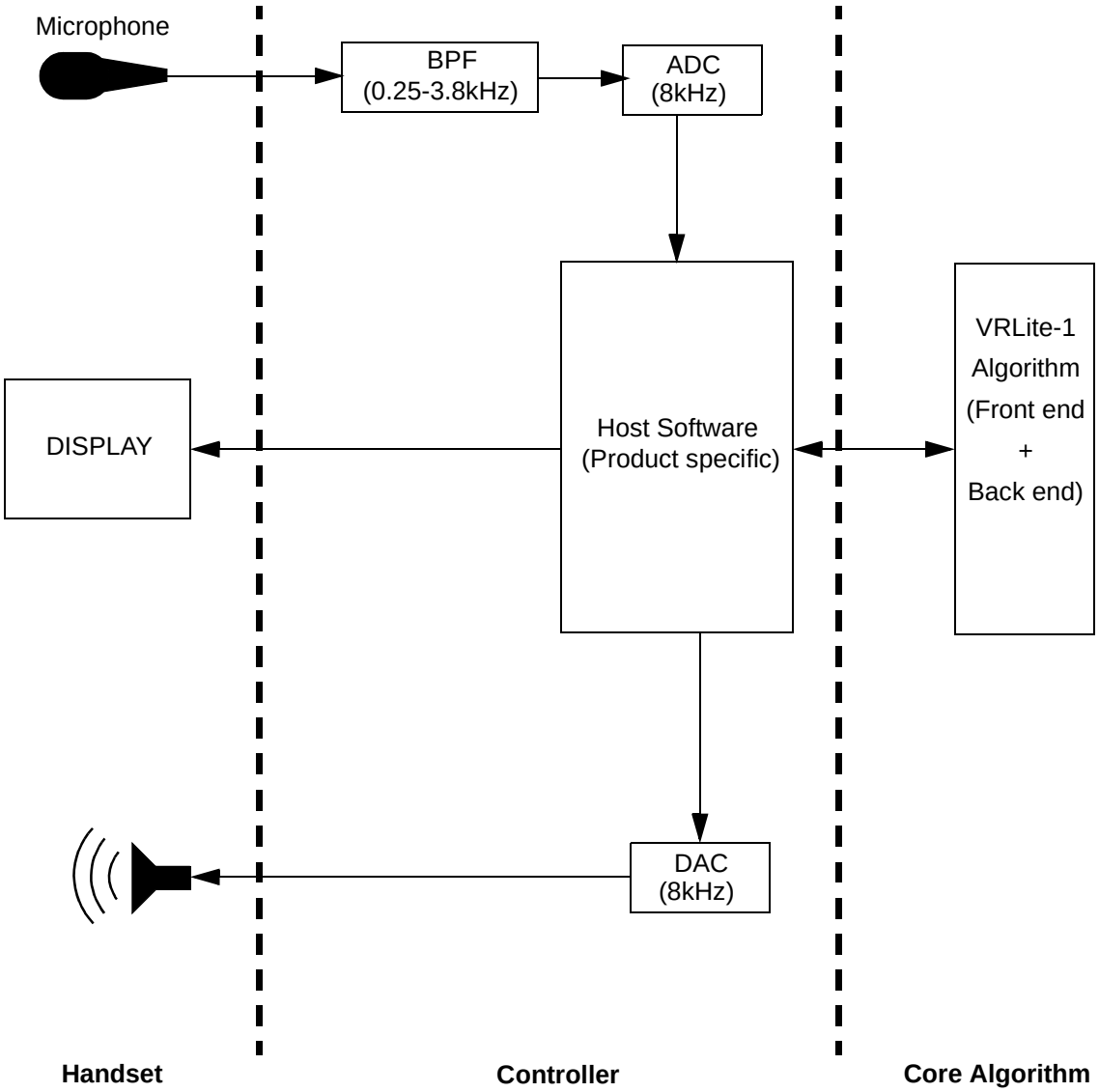

Figure 1-1. Typical Speaker-Dependent Speech Recognition Block Diagram

Handset: The input is a signal (through a microphone) containing speech, and will output a symbol that represents the recognized utterance or perform some task associated with the recognized speech. The display might be used to prompt the user for different inputs or to flag-out messages to the user.

Controller[1]**:** This part consists of host processor, A/D, and D/A converters. The input speech is filtered and digitized. The host software supplies one frame at a time to the front end of the VRLite algorithm. Note that front end processing in VRLite is real time. After front end processing, the host appropriately

1. The APIs provided in Chapter 3 are not for the "Handset" user but for the user who writes the "host software". The test files provided in the library somewhat serve as "controller".

invokes the back end for either training or recognition. The flow involved in training is shown in Figure 1-2; the flow involved in recognition is shown in Figure 1-3. Note that training requires two utterances of a word to be trained, while recognition requires only one utterance.

Core Algorithm: The core algorithm consists of frontend and backend of VRLite-1. The frontend is filter-bank based and extracts the features from the speech frames. The backend consists of HMM based training and recognition algorithms.

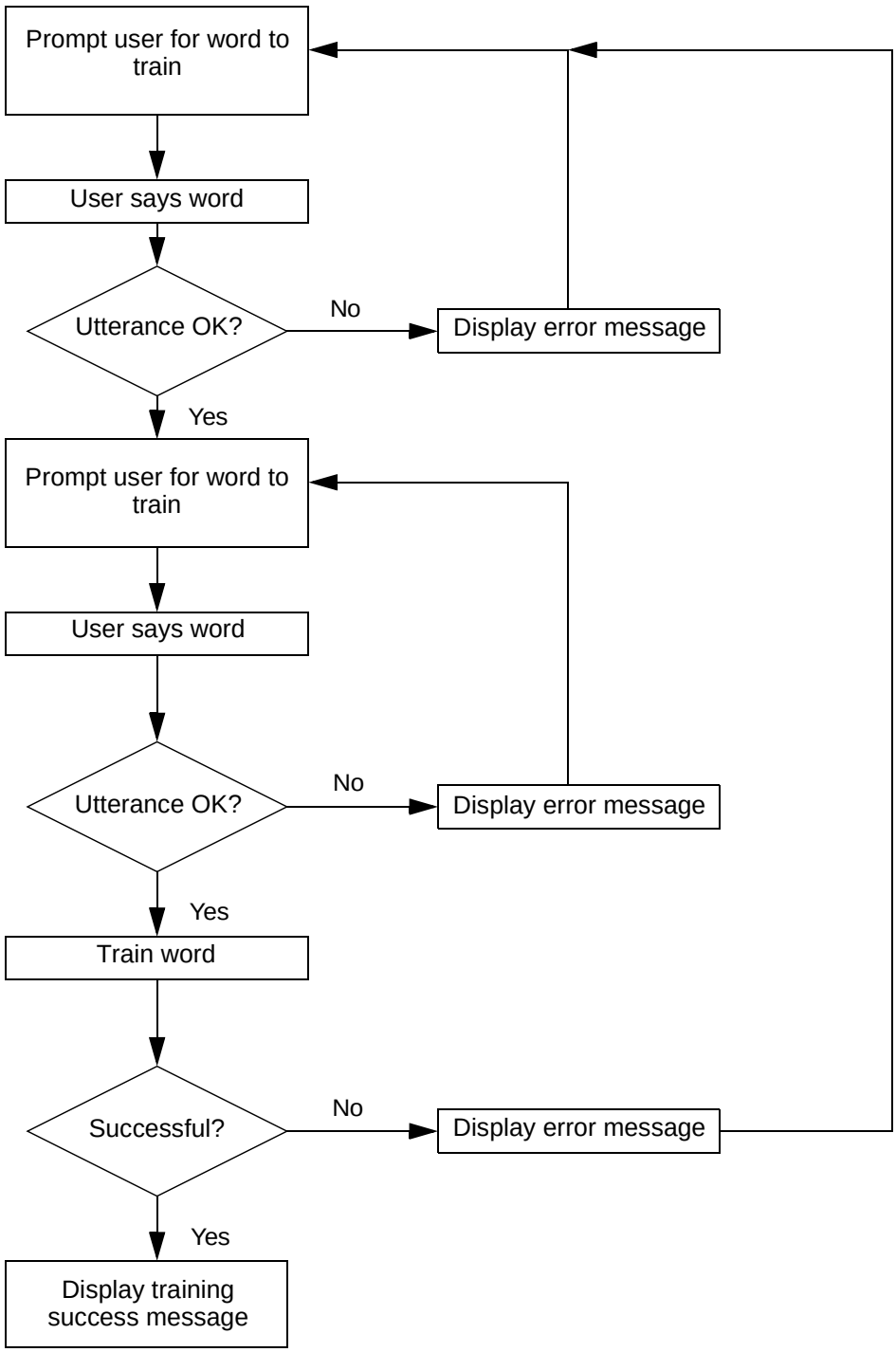

Figure 1-2. Training Flow Diagram

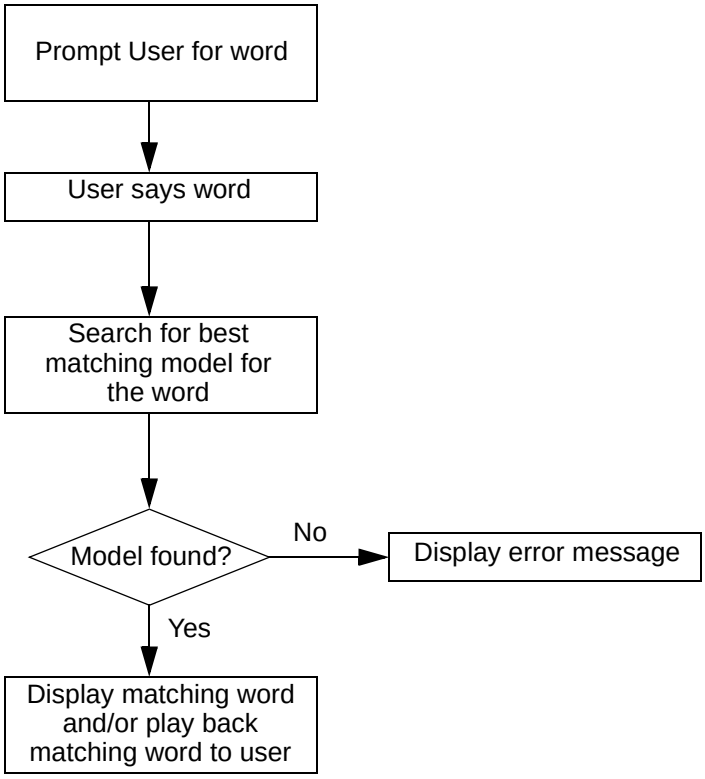

Figure 1-3. Recognition Flow Diagram

1.2.2 Features and Performance

Features:

- Speaker-dependent; isolated word recognition
- Reduced memory implementation
- Can train any number of models, as long as there is no constraint on host memory.

Note: The higher the number of trained models, the slower the recognition.

Performance:

For details on Memory and MIPS for a particular DSP, refer to the **Libraries** chapter of the appropriate Targeting manual.

Chapter 2
Directory Structure

2.1 Required Core Directories

Figure 2-1 details required platform directories:

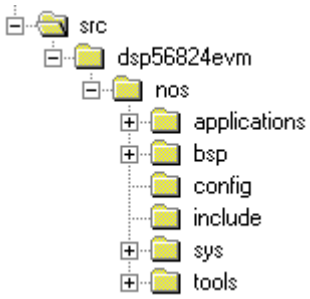

Figure 2-1. Core Directories

As shown in **Figure 2-1**, DSP56824EVM has no operating system (nos) support and contains these core directories:

- ***applications*** contains applications software that can be exercised on this platform
- ***bsp*** contains board support package specific for this platform
- ***config*** contains default hardware/software configurations for this platform
- ***include*** contains SDK header files which define the Application Programming Interface
- ***sys*** contains required system components
- ***tools*** contains utilities used by system components

There are also optional directories that include domain-specific libraries.

2.2 Optional (Domain-Specific) Directories

Figure 2-2 demonstrates how the VRLite-1 algorithm is encapsulated in the domain-specific directories under the directory ***speech***.

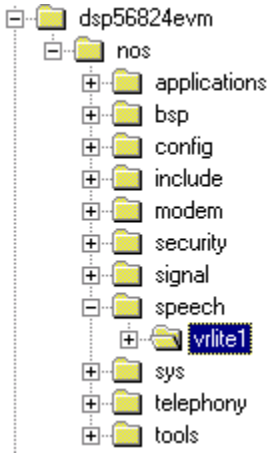

Figure 2-2. DSP56824 Directories

The ***speech*** directory includes speech recognition-specific algorithms. Figure 2-3 shows the ***vrlite1*** directory structure under the ***speech*** directory.

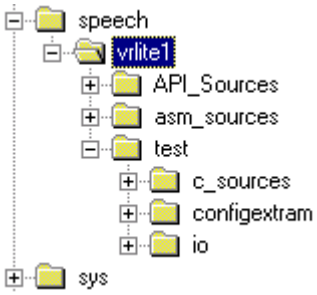

Figure 2-3. *vrlite1* Directory Structure

The ***vrlite1*** directory includes the following sub-directories:

- ***API_Sources*** contains APIs for VRLite-1
- ***asm_sources*** includes asm sources required for VRLite-1
- ***test*** includes C source files and configuration necessary for testing VRLite-1 library modules
  - — ***c_sources*** contains example test code for both training and recognition
  - — ***configextram*** contains the configuration files ***appconfig.c***, ***appconfig.h*** and ***linker.cmd*** specific to VRLite-1

Chapter 3
VRLite-1 Library Interfaces

3.1 VRLite-1 Services

The VRLite-1 library supports voice recognition functionality through two services: Training and Recognition. The data to be supplied must be in 16-bit word, fixed point (1.15) format, as shown below:

i = information bit

s = sign bit

3.2 Interface

The C interface for the VRLite-1 library services is defined in the C header file *vrlite1.h*, shown in **Code Example 3-1** as a reference.

Code Example 3-1. C Header File *vrlite1.h*

```
#ifndef __VRLITE1_H
#define __VRLITE1_H


/*****************************************************************
 Foundational Include Files
*****************************************************************/

#include "port.h"
```

```
/*****************************************************************
 Flags
 *****************************************************************/

/* VrControlFlag Bits. Please see user manual for details */

#define VR_ABORT                    0x0001 /* Not used */
#define VR_TRAINING                 0x0002
#define VR_RECOGNITION              0x0004
#define VR_REJECTION_ANALYSIS       0x0008
#define VR_RECOGNITION_LAST         0x0010 /* Not used */
#define VR_REJECTION_ANALYSIS_LAST  0x0020 /* Not used */
#define VR_OUT_OF_VOCAB             0x0040 /* Not used */
#define VR_HANDSFREE                0x0080

/* Flags which indicate the return status */

#define VR_FE_BUSY                  0x0001  /* Frontend busy flag */
#define VR_RA_BUSY                  0x0002  /* RA busy flag */
#define VR_RECOG_BUSY               0x0004  /* Recog. busy flag */

/* Training returns 91 HMM values for every word trained along
 * with 4 global noise mean values. These values MUST be stored
 * by the user in his callback function */

#define HMM_SIZE 91      /* Size of packed HMM buffer */
#define GLBL_STAT_SIZE 4 /* Size of global noise mean buffer */

/* After the recognition is complete, two values are returned through
 * callback. These values are the indices of the first best match
 * and the second best match, from the list of previously trained
 * words */

#define BW_SIZE 2        /* Size of best word table */

/*****************************************************************
 PASS/Error messages
*****************************************************************/

#define VR_FE_PASS                   0 /* Frontend PASS */
#define VR_TRAIN_PASS                1 /* Training PASS */
#define VR_RA_PASS                   2 /* RA PASS */
#define VR_RECOG_PASS                3 /* Recognition PASS */
#define VR_ACCEPT_MODEL              4 /* Accept the current model */

#define VR_TIME_OUT_ERROR           -2 /* Time out error from FE */
#define VR_BAD_SIGNAL_QUALITY_ERROR -3 /* Sig. Q. error from FE */
#define VR_CONFIG_ERROR             -4 /* Configuration error */
#define VR_TRAINING_ERROR           -5 /* Error in core training */
#define VR_REJECT_MODEL             -6 /* Reject the current model */

/*****************************************************************
 Structures for VRLite-1
*****************************************************************/
```

```
/* User configurable structure */

/* Call back structure */

typedef struct
{
    void (*pCallback) (void *pCallbackArg, void *pResult,
                       UWord16 NumResult);
    void *pCallbackArg;
} vrlite1_sCallback;


/* This structure must be used by the user
 * to configure VRLite-1 */

typedef struct
{
    UInt16 VrControlFlag;   /* Control flags, the bits are explained
                               above */
    Word16 GlobalStats[4];  /* Global statistics buffer; contains
                               4 words, namely, numModels, numReps,
                               MSW of NoiseMean, and LSW of
                               Noise Mean */
    vrlite1_sCallback Callback;
                            /* Callback structure */
} vrlite1_sConfigure;


/* VR-LITE1 handle structure */

/* This structure is used internally by VR-LITE1 for its
 * operation. The user should not set up this structure */

typedef struct
{
    UInt16 VrControlFlag;   /* Control flags, the bits are explained
                               above */
    UInt16 VrInitFlag;      /* Flag indicating the status of inits;
                               see possibilities above */
    UInt16 UtteranceNo;     /* Utterance number; first or second */
    Word16 *pGlobalStats;   /* Pointer to Global statistics; contains
                               4 words, namely, numModels, numReps,
                               MSW of NoiseMean, and LSW of
                               Noise Mean */
    Word16 *pContextBuf;    /* Context Buffer of length 80 */
    UWord16 ContextLen;     /* Length of context (<=80) */
    UInt16 NumPrevModels;   /* No. of previous models */
    UInt16 TempVar;         /* Temporary variable used as scratch */
    vrlite1_sCallback *Callback;
} vrlite1_sHandle;
```

```
/*****************************************************************
 Commands for VRLite-1 Control
*****************************************************************/

#define STOP_VRLITE1  1 /* Not used as of now */


/*****************************************************************
 Function Prototypes
*****************************************************************/

EXPORT vrlite1_sHandle *vrlite1Create (vrlite1_sConfigure *pConfig);

EXPORT Result vrlite1Init (vrlite1_sHandle *pVrlite1,
                           vrlite1_sConfigure *pConfig);

EXPORT Result vrlite1FrontendProcess (vrlite1_sHandle *pVrlite1,
                                      Word16 *pSamples,
                                      UWord16 NumSamples);

EXPORT Result vrlite1TrainingProcess (vrlite1_sHandle *pVrlite1);

EXPORT Result vrlite1RejAnalysisProcess (vrlite1_sHandle *pVrlite1,
                                         Word16 *pPrevModels);

EXPORT Result vrlite1RecognitionProcess (vrlite1_sHandle *pVrlite1,
                                         Word16 *pPrevModels);

EXPORT Result vrlite1Control (vrlite1_sHandle *pVrlite1,
                              UWord16 Command);

EXPORT void vrlite1Destroy (vrlite1_sHandle *pVrlite1);

#endif
```

3.3 Specifications

The following pages describe the VRLite-1 library functions.

Function arguments for each routine are described as *in*, *out*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the function. An *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, *inout* parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

3.3.1 *vrlite1Create*

Call(s):

```
vrlite1_sHandle *vrlite1Create (vrlite1_sConfigure *pConfig);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-1. *vrlite1Create* Arguments

pConfig	in	Points to the configuration data for VRLite-1

Description: The *vrlite1Create* function creates an instance of VRLite-1. During the *vrlite1Create* call, any dynamic resources required by the VRLite-1 algorithm are allocated.

The memory allocation is:

External Memory: 91 words

Internal Memory: 0 words

The *pConfig* argument points to the `vrlite1_sConfigure` structure used to configure VRLite-1 operation; for details on this structure, see **Section 3.3.2**.

The *vrlite1Create* function allocates memory dynamically using the *mem* library routines as shown in **Code Example 3-2**.

Code Example 3-2. Use of *mem* Library in *vrlite1Create* function

```
#include “port.h”
#include "vrlite1.h"
#include "mem.h"

vrlite1_sHandle *vrlite1Create (vrlite1_sConfigure *pConfig)
{
    vrlite1_sHandle *pVrlite1;

    /* Memory allocation for Handle */
    pVrlite1 = (vrlite1_sHandle *) memMallocEM (sizeof (vrlite1_sHandle));
    if (pVrlite1 == NULL) return (NULL);

    /* Force the pointers of members to NULL */
    pVrlite1->pContextBuf = NULL;
    pVrlite1->Callback = NULL;

    /* Allocate memory for context buffer */
    pVrlite1->pContextBuf = (Word16 *) memMallocEM (FRM_SIZE * sizeof (Word16));
    if (pVrlite1->pContextBuf == NULL)
    {
        vrlite1Destroy (pVrlite1);
        return (NULL);
    }
```

```
    /* Allocate memory for callback structure pointer */
    pVrlite1->Callback = (vrlite1_sCallback *) memMallocEM (sizeof
                                                    (vrlite1_sCallback));
    if (pVrlite1->Callback == NULL)
    {
        vrlite1Destroy (pVrlite1);
        return (NULL);
    }

    vrlite1Init (pVrlite1, pConfig);
    return (pVrlite1);
}
```

For details on the *vrlite1_sHandle* structure and constants used in **Code Example 3-2**, please refer to **Code Example 3-1**.

If the *vrlite1Create* function is called to create an instance, then *vrlite1Destroy* (see **Section 3.3.8**) should be used to destroy the instance.

Alternatively, the user can allocate memory statically, which requires duplicating all statements in the *vrlite1Create* function. In this case, the user can call the *vrlite1Init* function directly, bypassing the *vrlite1Create* function. If the user dynamically allocates memory without calling the *vrlite1Create* function, then the user himself must destroy the memory allocated.

Returns: Upon successful completion, the *vrlite1Create* function will return a pointer to the specific instance of VRLite-1created. If *vrlite1Create* is unsuccessful for any reason, it will return “NULL”.

Special Considerations:

- The VRLite-1 library is **not** re-entrant
- If *vrlite1Create* is called, then the user need not call *vrlite1Init* function, as it is called internally in the *vrlite1Create* function.

In **Code Example 3-3**, the application creates an instance of VRLite-1.

Code Example 3-3. Use of *vrlite1Create* Interface

```
#include “vrlite1.h”
#include “mem.h”

#define FRAME_LEN 80 /* NOTE: Frame length is defined to be 80 samples for
                             illustration purposes only. This value could be user’s
                             choice */

/* Function prototype */
void testVrlite1Training ();

/* The function below is an example for testing training with rejection analysis */

/* Input and output buffers */
Word16 inputTrn[FRAME_LEN]; /* Input buffer containing FRAME_LEN samples */
Word16 prevModelBuf[HMM_SIZE]; /* Each HMM model is always of size 91 */
```

```
/* Note: The structure of type WriteOutputTrn below is given here for illustration
* purposes only. The structure is to be decided by the user (who must write this
* kind of test file) depending on the callback function. This example assumes that the
* callback function just copies the output given by the VRLite-1 library to the
* hmmParam buffer in the structure below. */

typedef struct
{
    Word16 hmmParam[HMM_SIZE + GLBL_STAT_SIZE];
    UWord16 offset;
} WriteOutputTrn;  /* used in callback for collecting the output */

void testVrlite1Training ()
{
      vrlite1_sHandle *pVrlite1;
      vrlite1_sConfigure *pConfig;
      Int16 i;
      Result res;
      WriteOutputTrn hmmParamOut;

      hmmParamOut.offset = 0;
      /* Set up the configuration */
      pConfig = (vrlite1_sConfigure *) memMallocEM (sizeof (vrlite1_sConfigure));
      if (pConfig == NULL) assert (!"Out of memory");

      /* When there are no words trained, all the values of GlobalStats will
      * be zeros. These values are updated by the VRLite-1 algorithm
      * and returned to the calling function through callback. It is up to
      * the calling module (which calls Training API) to store these
      * returned values. The user (calling module) can define a buffer of
      * length 4 words, and keep overwriting with the latest returned
      * values. During configuration for VRLite-1 through pConfig
      * structure, these stored values must be assigned as shown below.
      * The values below are not zeros because there are some previously
      * trained models stored in the file prev_models.in located at
      * ...\nos\speech\vrlite1\test\io\ */

      pConfig->VrControlFlag = VR_TRAINING + VR_REJECTION_ANALYSIS;
      pConfig->GlobalStats[0] = 0x007e;
      pConfig->GlobalStats[1] = 0x000e;
      pConfig->GlobalStats[2] = 0x0000;
      pConfig->GlobalStats[3] = 0x00e0;
      pConfig->Callback.pCallback = CallbackTrain;
      pConfig->Callback.pCallbackArg = (WriteOutputTrn *) (&hmmParamOut);

      /* Vrlite1 instance creation and initialization */
      pVrlite1 = vrlite1Create (pConfig);
      if (pVrlite1 == NULL) assert (!"Out of Memory");

      /* Frontend processing for utterance 1 */
      ....
      res = VR_FE_BUSY;
      while (res == VR_FE_BUSY)
      {
          /* Read FRAME_LEN samples at a time into inputTrn (from file or codec) */
          ....
```

```
    res = vrlite1FrontendProcess (pVrlite1, inputTrn, FRAME_LEN);
}
....

/* Only if processing of first utterance is proper, process
* the second utterance */

if (res == VR_FE_PASS)
{
    /* Frontend processing for utterance 2 */
    ....
    res = VR_FE_BUSY;
    while (res == VR_FE_BUSY)
    {
       /* Read FRAME_LEN samples at a time into inputTrn
       * (from file or codec) */
       ....
       res = vrlite1FrontendProcess (pVrlite1, inputTrn, FRAME_LEN);
    }
    ....
}

if (res == VR_TIME_OUT_ERROR)
{
    /* Time out Error. Take appropriate action */
    ....
    vrlite1Destroy (pVrlite1);
    return;
}
else if (res == VR_BAD_SIGNAL_QUALITY_ERROR)
{
    /* Bad signal quality. Take appropriate action */
    ....
    vrlite1Destroy (pVrlite1);
    return;
}

/* If there is no error in frontend processing, continue
* with training */

/* Training */
res = vrlite1TrainingProcess (pVrlite1);

if (res != VR_TRAIN_PASS)
{
   if (res == VR_TRAINING_ERROR)
   {
       /* Error in core training */
   }

   /* Take appropriate action */
   ....
   vrlite1Destroy (pVrlite1);
   return;
}
```

```
        else
        {
            /* Training over. Take appropriate action */
            ....
        }

        /* NOTE: If training fails, there is no point in going
        * ahead with rejection analysis */

        /* Rejection Analysis */
        ....
        for (i = 0; i < pConfig->GlobalStats[0]; i++)
        {
            /* Fill prevModelBuf buffer with ONE previous model */
            ....
            res = vrlite1RejAnalysisProcess (pVrlite1, prevModelBuf);
            if (res != VR_RA_PASS)
            {
                break;
            }
        }

        if (res == VR_CONFIG_ERROR)
        {
            /* Configuration error. Take appropriate action */
            ....
        }
        else if (res == VR_REJECT_MODEL)
        {
            /* Reject the current model, i.e., do not add the current model
             * to the list of previous models */
            ....
        }
        else if (res == VR_ACCEPT_MODEL)
        {
            /* If VR_ACCEPT_MODEL, the HMM sent by training
             * module can be accepted (i.e., add the current model
             * to the list of previous models and increment the number of
             * models, i.e., the first value in the global statistics must
             * reflect this increment in the number-of-models) */
            ....
        }
        ....
}
```

For details on structures used in **Code Example 3-3**, see **Code Example 3-1**.

3.3.2 *Vrlite1Init*

Call(s):

```
Result vrlite1Init (vrlite1_sHandle *pVrlite1, vrlite1_sConfigure *pConfig);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-2. *vrlite1Init* Arguments

pVrlite1	in	Handle to an instance of VRLite-1
pConfig	in	A pointer to a data structure containing data for initializing the VRLite-1 algorithm

Description: The *vrlite1Init* function will initialize the VRLite-1 algorithm. During the initialization, all resources will be set to their initial values in preparation for VRLite-1 operation. Before calling the *vrlite1Init* function, a VRLite-1 instance must be created. The VRLite-1 instance (*pVrlite1*) can be created either by calling the *vrlite1Create* function (see **Section 3.3.1**) or by statically allocating memory, which does not require a call to the *vrlite1Create* function.

The parameter *pConfig* points to a data structure of type `vrlite1_sConfigure`; its fields initialize VRLite-1operation in the following manner:

VrControlFlag - The bits of the control flag are shown below. The numbers inside the box indicate the bit number.

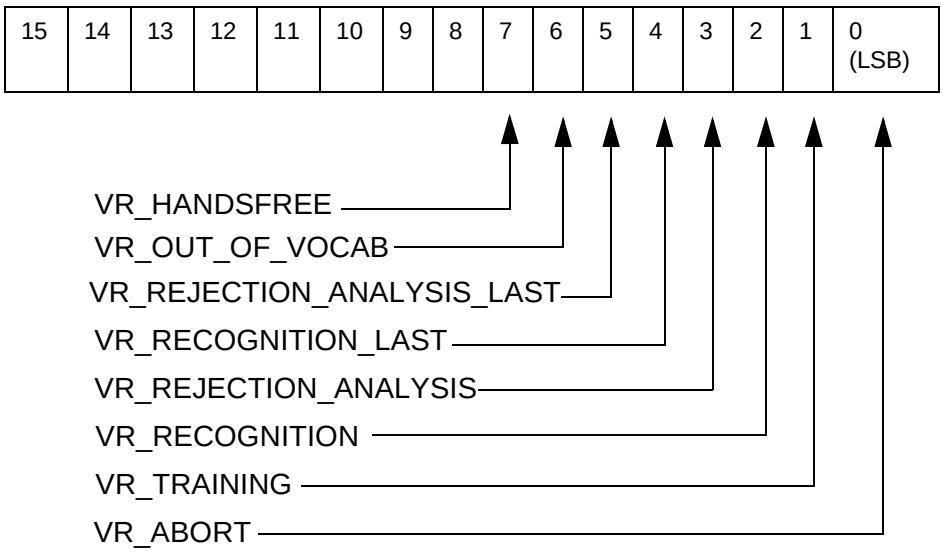

Table 3-3. Description of the fields of VrControlFlag

Bit Name	Bit Status	Description
VR_ABORT	set	Indicates that VRLite must be aborted; currently, this feature is **not applicable**
VR_TRAINING	set	Indicates that VRLite is configured for training
VR_RECOGNITION	set	Indicates that VRLite is configured for recognition
VR_REJECTION_ANALYSIS	set	Indicates that VRLite is configured for Training as well as Rejection Analysis; this bit **cannot** be set without setting the VR_TRAINING bit
VR_RECOGNITION_LAST	Not Applicable	
VR_REJECTION_ANALYSIS_LAST	Not Applicable	
VR_OUT_OF_VOCAB	Not Applicable	The feature of OOV rejection is permanently enabled in the library
VR_HANDSFREE	set	Indicates that training or recognition is done in handsfree mode

GlobalStats - Global statistics buffer of size 4. Each location in the buffer is explained below.

GlobalStats[0] = Number of previously trained models

GlobalStats[1] = Number of utterances for which noise update is done

GlobalStats[2] = Most Significant Word of Noise Mean

GlobalStats[3] = Least Significant Word of Noise Mean

This buffer must be filled by the host and given to API. These values must be zeros before training any model. The VRLite algorithm then updates these values during the training of every new model and returns them to the host through the callback function. The host stores the values given by the VRLite algorithm and ensures that these values are **not altered outside** the VRLite algorithm.

Callback - A structure of type `vrlite1_sCallback`; it describes the procedure which VRLite-1 will call once the training or recognition is complete. The training calls back twice, outputting 91 HMM values and 4 global noise mean values. The recognition does callback to output 2 best word values. The callback procedure has the following declaration:

```
void (*pCallback) (void *pCallbackArg, void *pResult,
                   UWord16 NumResult);
```

The callback procedure parameter, *pCallbackArg*, is supplied by the user in the `vrlite1_sCallback` structure; this value is passed back to the user during the call to the callback procedure. Typically, *pCallbackArg* points to context information used by the callback procedure, which the user must write.

An example callback procedure for training is shown in **Code Example 3-4** for reference. You must write your own callback procedure. This callback procedure stores the result in a buffer specified by the user or the host through a *pCallbackArg* pointer.

Code Example 3-4. Sample Callback Procedure for Training

```
void CallbackTrain (void *pCallbackArg, void *pResult, UWord16  NumResult)
{
      Int16 i;
      WriteOutputTrn *pTemp = (WriteOutputTrn *) pCallbackArg;
      Word16 *pTempIn = (Word16 *) pResult;
      for (i = 0; i < NumResult; i++)
      {
           pTemp->hmmParam[(pTemp->offset) + i] = pTempIn[i];
      }
      pTemp->offset += NumResult;
      return;
}
```

Code Example 3-4 is the callback function, **to be written by the user**, to collect the output given by the *vrlite1TrainingProcess* function (**Training** thread) of the VRLite-1 library. Training returns 91 HMM values, then returns 4 Global Statistics values through callback, which should be collected by the user[1]. In this example, `WriteOutputTrn` is a structure, illustrating one of the simple ways of collecting these outputs. In an embedded system environment, these outputs must be stored in non-volatile memory; the user can do this directly in the callback function if Rejection Analysis is *Off*. If Rejection Analysis is required (*On*), then the user can first copy the outputs into a temporary buffer as shown in **Code Example 3-4**, then copy them into non-volatile memory, depending upon the results of Rejection Analysis. In this example, during the first callback from the *vrlite1TrainingProcess* function, the CallbackTrain function stores the 91 values in the *hmmParam* buffer of the `WriteOutputTrn` structure, during which the `pTemp->offset` would be zero. While going out of this function, the offset would be incremented, so that during the next callback, the 4 Global Statistics output values are written from the 92nd location onwards.

An example callback procedure for recognition is shown in **Code Example 3-5** for reference.

1. In this case, “user” refers to the host function and not to the end user.

Code Example 3-5. Sample Callback Procedure for Recognition

```
void CallbackRecog (void *pCallbackArg, void *pResult, UWord16  NumResult)
{
       Int16 i;
       Word16 *pTemp = (Word16 *) pCallbackArg;
       Word16 *pTempIn = (Word16 *) pResult;
       for (i = 0; i < NumResult; i++)
       {
            pTemp[i] = pTempIn[i];
       }
       return;
}
```

Code Example 3-5 is the callback function, to be written by the user, which collects the output given by the *vrlite1RecognitionProcess* function (Recognition thread) of the VRLite-1 library. Recognition returns two values (first best word and second best word indices) through callback, which should be collected by the user[1]. This example illustrates just one of the simple ways of collecting these outputs. In an embedded system environment, these outputs can be used to take appropriate further action. During the callback, the *CallbackRecog* function stores the two values in the *BestWordOut* buffer pointed to by *pCallbackArg*.

Returns: Upon successful completion, *VRliteInit* returns “PASS”; *VRliteInit* returns “VR_CONFIG_ERROR” to indicate an error in the configuration of *pconfig* structure.

Special Considerations:

- If *vrlite1Create* is called, then the user need not call *vrlite1Init* function as it is called internally in the *vrlite1Create* function.

Code Example: See **Code Example 3-3** to learn how to use the *vrlite1Init* function.

1. In this case, “user” refers to the host function and not to the end user.

3.3.3 *vrlite1FrontendProcess*

Call(s):

```
Result vrlite1FrontendProcess (vrlite1_sHandle *pVrlite1,
                               Word16 *pSamples,
                               UWord16 NumSamples);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-4. *vrlite1FrontendProcess* Arguments

pVrlite1	in	Handle to an instance of VRLite-1
pSamples	in	Pointer to speech samples (samples of the uttered word)
NumSamples	in	The number of samples to be processed

Description: The *vrlite1FrontendProcess* function processes samples to generate feature vectors for the utterance. These feature vectors will be used in training/recognition. The user must call the *vrlite1FrontendProcess* function in a loop as long as the "VR_FE_BUSY" flag is returned from *vrlite1FrontendProcess* function.

Returns: The following are the return values:

- "VR_TIME_OUT_ERROR" indicates time out when valid speech cannot be found in the utterance for approximately the first 2 seconds
- "VR_BAD_SIGNAL_QUALITY_ERROR" indicates that the signal quality of the utterance is bad
- "VR_FE_PASS" indicates that the front end processing is complete
- "VR_FE_BUSY" indicates that the front end processing is not finished, and more samples are required to complete the front end processing

Special Considerations:

- For Training, this API must be invoked for two utterances; for Recognition, this API can be invoked for only one utterance
- To maximize the performance of recognition in noisy environments, the environment must be as quiet as possible for training.

Code Example: See **Code Example 3-3** to learn how to use the *vrlite1FrontendProcess* function.

3.3.4 *vrlite1TrainingProcess*

Call(s):

```
Result vrlite1TrainingProcess (vrlite1_sHandle *pVrlite1);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-5. *vrlite1TrainingProcess* Arguments

pVrlite1	in	Handle to an instance of VRLite-1

Description: The *vrlite1TrainingProcess* function will generate the HMM model representation for the uttered word. Training requires two utterances of the **same word** to generate the HMM model. The user must call the *vrlite1TrainingProcess* function after front end processing.

Returns: Upon successful completion, *vrliteTrainingProcess* will return “VR_TRAIN_PASS”; if there is a failure in the core training module, *vrliteTrainingProcess* returns “VR_TRAINING_ERROR”.

Special Considerations: None

Code Example: See **Code Example 3-3** to learn how to use the *vrlite1TrainingProcess* function.

3.3.5 *vrlite1RejAnalysisProcess*

Call(s):

```
Result vrlite1RejAnalysisProcess (vrlite1_sHandle *pVrlite1,
                                  Word16 *pPrevModels);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-6. *vrlite1RejAnalysisProcess* Arguments

pVrlite1	in	Handle to an instance of VRLite-1
pPrevModels	in	Pointer to previously-trained models

Description: The *vrlite1RejAnalysisProcess* function decides whether the HMM model and global noise mean values given by the *vrlite1TrainingProcess* function should be retained or rejected. Rejection Analysis requires the previously-trained models to do the analysis. The user must call the *vrlite1RejAnalysisProcess* function after Training, and it must be called for each previously-trained model.

Returns: The following are the return values:

- "VR_CONFIG_ERROR" indicates an error in the configuration of the *pConfig* structure, because Rejection Analysis cannot work without training
- "VR_REJECT_MODEL" indicates that the model is **rejected**; i.e., a similar model might already exist in the vocabulary (the set of previously trained models)
- "VR_RA_PASS" indicates that Rejection Analysis passed and the next model is to be supplied for analysis
- "VR_ACCEPT_MODEL" indicates that the rejection analysis is **complete** and the current model must be accepted

Special Considerations: One previous model is to be passed at a time. This implies that this API must be called previous-model-number-of-times from the calling module.

Code Example: See **Code Example 3-3** to learn how to use the *vrlite1RejAnalysisProcess* function.

3.3.6 *vrlite1RecognitionProcess*

Call(s):

```
Result vrlite1RecognitionProcess (vrlite1_sHandle *pVrlite1,
                                  Word16 *pPrevModels);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-7. *vrlite1RecognitionProcess* Arguments

pVrlite1	in	Handle to an instance of VRLite-1
pPrevModels	in	Pointer to previously-trained models

Description: The *vrlite1RecognitionProcess* function recognizes the word uttered with respect to the vocabulary (i.e., previously-trained models). Unlike training, recognition requires only one utterance for its operation. After configuring for recognition, the user must call the *vrlite1RecognitionProcess* function for each previously-trained model; see **Code Example 3-6**.

Returns: Upon successful completion of the recognition process, *vrlite1RecognitionProcess* returns "VR_RECOG_PASS"; if recognition is busy and the next previous model should be supplied, *vrlite1RecognitionProcess* returns "VR_RECOG_BUSY".

Special Considerations: Only one previous model can be passed at a time. This implies that this API is to be called previous-model-number-of-times from the calling module; see **Code Example 3-6**.

Code Example 3-6. Use of *vrlite1RecognitionProcess* Interface

```
#include "vrlite1.h"
#include "mem.h"

#define FRAME_LEN 80 /* NOTE: Frame length is defined to be 80 samples for
                              illustration purposes only. This value could be user's
                              choice */

/* Function prototype */
void testVrlite1Recog ();

/* The function below is an example for testing recognition */

/* Input and output buffers */
Word16 inputRecog[FRAME_LEN]; /* Input buffer containing FRAME_LEN samples */
Word16 prevModelBuf[HMM_SIZE]; /* Each HMM model is always of size 91 */


void testVrlite1Recog ()
{
      vrlite1_sHandle *pVrlite1;
      vrlite1_sConfigure *pConfig;
      Int16 i;
      Result res;
      Word16 BestWordOut[BW_SIZE];
```

```
/* Set up the configuration */
pConfig = (vrlite1_sConfigure *) memMallocEM (sizeof (vrlite1_sConfigure));
if (pConfig == NULL) assert (!"Out of memory");

/* When there are no words trained, all the values of GlobalStats will
* be zeros. These values are updated by the VRLite-1 algorithm
* and returned to the calling function through callback. It is up to
* the calling module (which calls Training API) to store these
* returned values. The user (calling module) can define a buffer of
* length 4 words, and keep overwriting with the latest returned
* values. During configuration for VRLite-1 through pConfig
* structure, these stored values have to be assigned as shown below.
* The values below are not zeros because we have some previously
* trained models stored in the file prev_models.in located at
* ...\nos\speech\vrlite1\test\io\ */

pConfig->VrControlFlag = VR_RECOGNITION;
pConfig->GlobalStats[0] = 0x007e;
pConfig->GlobalStats[1] = 0x000e;
pConfig->GlobalStats[2] = 0x0000;
pConfig->GlobalStats[3] = 0x00e0;
pConfig->Callback.pCallback = CallbackRecog;
pConfig->Callback.pCallbackArg = (Word16 *) (&BestWordOut);

/* Vrlite1 instance creation and initialization */
pVrlite1 = vrlite1Create (pConfig);
if (pVrlite1 == NULL)
{
   assert (!"Out of Memory");
}

/* Frontend processing for utterance */
....
res = VR_FE_BUSY;
while (res == VR_FE_BUSY)
{
   /* Read FRAME_LEN samples at a time into inputRecog buffer
    * (either from file or codec) */
   ....
   res = vrlite1FrontendProcess (pVrlite1, inputRecog, FRAME_LEN);
}

if (res == VR_TIME_OUT_ERROR)
{
    /* Time out Error. Take appropriate action */
    ....
    vrlite1Destroy (pVrlite1);
    return;
}
else if (res == VR_BAD_SIGNAL_QUALITY_ERROR)
{
    /* Bad signal quality. Take appropriate action */
    ....
    vrlite1Destroy (pVrlite1);
    return;
}
```

```
        /* Recognition */
        ....

        for (i = 0; i < pConfig->GlobalStats[0]; i++)
        {
           /* Get ONE previously trained model, which is always of size 91
            * into prevModelBuf buffer */
           ....
           res = vrlite1RecognitionProcess (pVrlite1, prevModelBuf);
           if (res == VR_CONFIG_ERROR)
           {
              break;
           }
        }

        if (res == VR_CONFIG_ERROR)
        {
           /* Configuration error, take appropriate action */
           ....
        }
        else if (res == VR_RECOG_BUSY)
        {
           /* Error: All prev. models are not supplied. Take appropraite action */
           ....
        }
        else
        {
           /* Recognition complete. Take appropraite action */
           ....
        }

        /* Destroy the VR-LITE1 instance */
        vrlite1Destroy (pVrlite1);
        return;
}
```

For details on structures used in **Code Example 3-6**, see **Code Example 3-1**.

3.3.7 *vrlite1Control*

Call(s):

```
Result vrlite1Control (vrlite1_sHandle *pVrlite1, UWord16 Command);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-8. *vrlite1Control* Arguments

pVrlite1	in	Handle to an instance of VRLite-1 generated by a call to *vrlite1Create*
Command	in	Command for controlling VRLite-1

Description: Reserved for future use

Returns: None

Special Considerations: None

3.3.8 *vrlite1Destroy*

Call(s):

```
void vrlite1Destroy (vrlite1_sHandle *pVrlite1);
```

Required Header: *vrlite1.h*

Arguments:

Table 3-9. *vrlite1Destroy* Arguments

pVrlite1	in	Handle to an instance of VRLite-1 generated by a call to *vrlite1Create*

Description: The *vrlite1Destroy* function destroys the instance of VRLite-1 originally created by a call to *vrlite1Create*. If the user bypassed the *vrlite1Create* function to create an instance on his own, the *vrlite1Destroy* function should not be called.

Returns: None

Special Considerations: None

Code Example 3-7. Use of *vrlite1Destroy* Interface

```
#include “vrlite1.h”
#include “mem.h”
```

See **Code Example 3-3** and **Code Example 3-6** to learn how to use the *vrlite1Destroy* function.

Chapter 4
Building the VRLite-1 Library

4.1 Building the VRLite-1 Library

The VRLite-1 library combines all of the components described in previous chapter into one library: *vrlite1.lib*. To build this library, a Metrowerks CodeWarrior project, ***vrlite1.mcp***, is provided. This project and all the necessary components to build the VRLite-1 library are located in the **...\nos\speech\vrlite1** directory of the SDK directory structure.

There are two methods to execute a system library project build: dependency build and direct build.

4.1.1 Dependency Build

Dependency build is the easiest approach and requires no additional work on the user's part. If you add the VRLite-1 library project, ***vrlite1.mcp,*** to your application project, as shown in **Figure 4-1**, the VRLite-1 library will automatically build when the application is built.

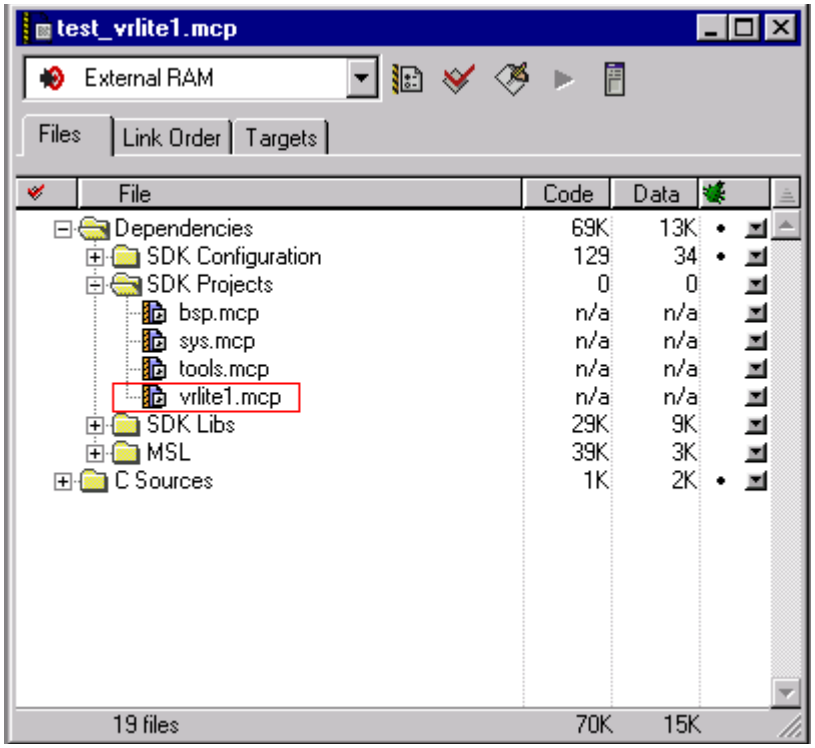

Figure 4-1. Dependency Build for VRLite-1 Project

4.1.2 Direct Build

Direct build allows you to build the VRLite-1 library independently of any other build. Follow these steps:

Step 1. Open ***vrlite1.mcp*** project, as shown in **Figure 4-2**.

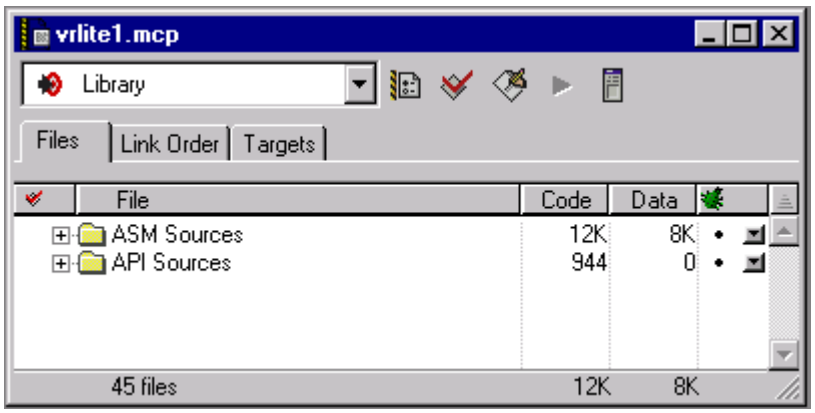

Figure 4-2. *vrlite1.mcp* Project

Step 2. Execute the build by pressing function key [F7] or by choosing the *Make* command from the Project menu; see **Figure 4-3**.

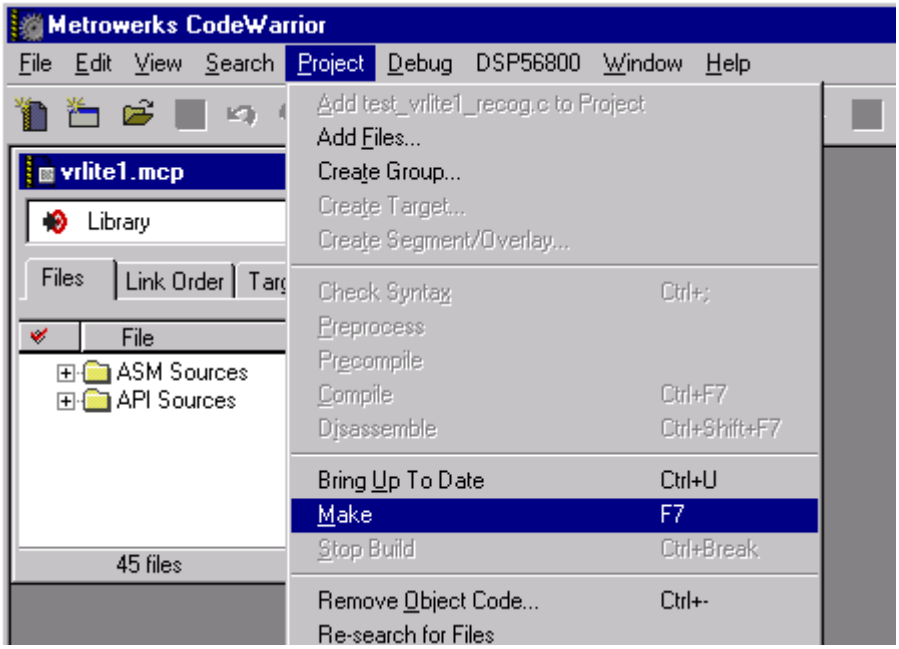

Figure 4-3. Execute Make

At this point, if the build is successful, the *vrlite1.lib* library file is created in the **...\nos\speech\vrlite1\Debug** directory.

Chapter 5
Linking Applications with the VRLite-1 Library

5.1 VRLite-1 Library

The library includes APIs, which define the interface between the user application and the VRLite-1 modules. As the VRLite-1 APIs are not a direct interface to the end user (for example, a handset user), the user application must be like host software. To invoke VRLite-1, APIs must be called in this order:

— vrlite1Create (.....);
— vrlite1Init (.....);
— vrlite1FrontendProcess (.....);
— vrlite1TrainingProcess (.....);
— vrlite1RejAnalysisProcess (.....);
— vrlite1RecognitionProcess (.....);
— vrlite1Destroy (.....);

5.1.1 Library Sections

The VRLite-1 Library contains the following sections:

- FRONTEND_ROM, a data ROM section
- VR_COMMON_ROM, a data ROM section
- VR_RECO_ROM, a data ROM section
- VR_Y_1_MEM_SHARE, a data RAM section
- VR_Y_2_MEM_SHARE, a data RAM section
- VR_Y_3_MEM_SHARE, a data RAM section
- VR_Y_4_MEM_SHARE, a data RAM section
- VR_Y_5_MEM_SHARE, a data RAM section
- VR_X_MEM_SHARE, a data RAM section
- VR_LONG_MEM_SHARE, a data RAM section

See **Code Example 5-1** for an example *linker.cmd* file used in the test application.

Code Example 5-1. *linker.cmd* File

```
# Linker.cmd file for DSP56824EVM External RAM
# using both internal and external data memory (EX = 0)
# and using external program memory (Mode = 3)

#*****************************************************************************
MEMORY {

       .pInterruptVector         (RWX) : ORIGIN = 0x0000, LENGTH = 0x002C
       .pExtRAM                  (RWX) : ORIGIN = 0x002C, LENGTH = 0xFFD4

       .xAvailable               (RW)  : ORIGIN = 0x0000, LENGTH = 0x0030
       .xCWRegisters             (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010
       .xIntRAM_DynamicMem1      (RW)  : ORIGIN = 0x0040, LENGTH = 0x07C0
       .xIntROM                  (R)   : ORIGIN = 0x0800, LENGTH = 0x0800
       .xIntRAM_DynamicMem2      (RW)  : ORIGIN = 0x1000, LENGTH = 0x0600
       .xHole                    (R)   : ORIGIN = 0x1600, LENGTH = 0x0A00
       .xExtRAM                  (RW)  : ORIGIN = 0x2000, LENGTH = 0xC000
       .xExtRAM_DynamicMem       (RW)  : ORIGIN = 0xE000, LENGTH = 0x1000
       .xStack                   (RW)  : ORIGIN = 0xF000, LENGTH = 0x0F80
       .xPeripherals1            (RW)  : ORIGIN = 0xFF80, LENGTH = 0x0040
       .xPeripherals2            (RW)  : ORIGIN = 0xFFC0, LENGTH = 0x0040

       .xExt1Vrlite1             (RW)  : ORIGIN = 0x4000, LENGTH = 0x0000
       .xExt2Vrlite1             (RW)  : ORIGIN = 0x4300, LENGTH = 0x0000
       .xExt3Vrlite1             (RW)  : ORIGIN = 0x4900, LENGTH = 0x0000
       .xExt4Vrlite1             (RW)  : ORIGIN = 0x4D00, LENGTH = 0x0000
       .xExt5Vrlite1             (RW)  : ORIGIN = 0x4E00, LENGTH = 0x0000
       .xExt6Vrlite1             (RW)  : ORIGIN = 0x5400, LENGTH = 0x0000
       }
#*****************************************************************************
FORCE_ACTIVE {FconfigInterruptVector}
SECTIONS {
#*****************************************************************************
       #
       # Data (X) Memory Layout
       #
       _EX_BIT     = 0;

       # Internal Memory Partitions (for mem.h partitions)
       _NUM_IM_PARTITIONS = 2;  # .xIntRAM_DynamicMem1 and .xIntRAM_DynamicMem2

       # External Memory Partition (for mem.h partitions)
       _NUM_EM_PARTITIONS = 1;   # .xExtRAM_DynamicMem
#*****************************************************************************
       .ApplicationInterruptVector :
       {
              vector.c (.text)

       } > .pInterruptVector
#*****************************************************************************
       .ApplicationCode :
       {
              # Place all code into External Program RAM
              * (.text)
              * (rtlib.text)
```

```
                * (fp_engine.text)
                * (user.text)

        } > .pExtRAM
#*****************************************************************************
        .ApplicationData :
        {
                # Define variables for C initialization code

                F_Xdata_start_addr_in_ROM = ADDR(.xIntROM) + SIZEOF(.xIntROM) / 2;
                F_StackAddr               = ADDR(.xStack);
                F_StackEndAddr            = ADDR(.xStack) + SIZEOF(.xStack) / 2  - 1;
                F_Xdata_start_addr_in_RAM = .;


                # Define variables for SDK mem library

                FmemEXbit = .;
                WRITEH(_EX_BIT);
                FmemNumIMpartitions = .;
                WRITEH(_NUM_IM_PARTITIONS);
                FmemNumEMpartitions = .;
                WRITEH(_NUM_EM_PARTITIONS);
                FmemIMpartitionList = .;
                WRITEH(ADDR(.xIntRAM_DynamicMem1));
                WRITEH(SIZEOF(.xIntRAM_DynamicMem1) / 2);
                WRITEH(ADDR(.xIntRAM_DynamicMem2));
                WRITEH(SIZEOF(.xIntRAM_DynamicMem2) / 2);
                FmemEMpartitionList = .;
                WRITEH(ADDR(.xExtRAM_DynamicMem));
                WRITEH(SIZEOF(.xExtRAM_DynamicMem) /2);

                # Place all data into External RAM

                * (.data)
                * (fp_state.data)
                * (rtlib.data)

                F_Xdata_ROMtoRAM_length = 0;

                F_bss_start_addr = .;
                _BSS_ADDR = .;

                * (rtlib.bss.lo)
                * (.bss)

                F_bss_length = . - _BSS_ADDR;  # Copy DATA

                } > .xExtRAM
#*****************************************************************************


        FArchIO   = ADDR(.xPeripherals2);
#*****************************************************************************
```

```
        .Vrlite1Data1 :
        {
                # Place all code into External Data RAM
                .=ALIGN(0x10);
                * (FRONTEND_ROM.data)
                * (VR_COMMON_ROM.data)
                * (VR_RECO_ROM.data)
                .=ALIGN(0x10);
                * (VR_Y_1_MEM_SHARE.data)
        } > .xExt1Vrlite1

        .Vrlite1Data2 :
        {
                # Place all code into External Data RAM
                .=ALIGN(0x100);
                * (VR_Y_2_MEM_SHARE.data)
        } > .xExt2Vrlite1

        .Vrlite1Data3 :
        {
                # Place all code into External Data RAM
                .=ALIGN(0x10);
                * (VR_Y_3_MEM_SHARE.data)
        } > .xExt3Vrlite1

        .Vrlite1Data4 :
        {
                # Place all code into External Data RAM
                .=ALIGN(0x10);
                * (VR_Y_4_MEM_SHARE.data)
        } > .xExt4Vrlite1

        .Vrlite1Data5 :
        {
                 # Place all code into External Data RAM
                .=ALIGN(0x20);
                * (VR_LONG_MEM_SHARE.data)
        } > .xExt5Vrlite1

        .Vrlite1Data6 :
        {
                 # Place all code into External Data RAM
                * (VR_X_MEM_SHARE.bss)
                * (VR_Y_5_MEM_SHARE.bss)
        } > .xExt6Vrlite1
}
```

Chapter 6
VRLite-1 Applications

6.1 Test and Demo Applications

To verify the VRLite-1 algorithm, test and demo applications have been developed. Refer to the **Targeting Motorola DSP568xx Platform** Manual for the DSP you are using to see if the test and demo applications are available for your target.

Chapter 7
License

7.1 Limited Use License Agreement

LIMITED USE LICENSE AGREEMENT

PLEASE READ THIS AGREEMENT CAREFULLY BEFORE USING THIS SOFTWARE. BY USING OR COPYING THE SOFTWARE, YOU AGREE TO THE TERMS OF THIS AGREEMENT.

The software in either source code form ("Source") or object code form ("Object") (cumulatively hereinafter "Software") is provided under a license agreement ("Agreement") as described herein. Any use of the Software including copying, modifying, or installing the Software so that it is usable by or accessible by a central processing unit constitutes acceptance of the terms of the Agreement by the person or persons making such use or, if employed, the employer thereof ("Licensee") and if employed, the person(s) making such use hereby warrants that they have the authority of their employer to enter this license agreement,. If Licensee does not agree with and accept the terms of this Agreement, Licensee must return or destroy any media containing the Software or materials related thereto, and destroy all copies of the Software.

The Software is licensed to Licensee by Motorola Incorporated ("Motorola") for use under the terms of this Agreement. Motorola retains ownership of the Software. Motorola grants only the rights specifically granted in this Agreement and grants no other rights. Title to the Software, all copies thereof and all rights therein, including all rights in any intellectual property including patents, copyrights, and trade secrets applicable thereto, shall remain vested in Motorola.

For the Source, Motorola grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to use, copy, and make derivatives of the Source solely in a development system environment in order to produce object code solely for operating on a Motorola semiconductor device having a central processing unit ("Derivative Object").

For the Object and Derivative Object, Motorola grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to copy, use, and distribute the Object and the Derivative Object solely for operating on a Motorola semiconductor device having a central processing unit.

Licensee agrees to: (a) not use, modify, or copy the Software except as expressly provided herein, (b) not distribute, disclose, transfer, sell, assign, rent, lease, or otherwise make available the Software, any derivatives thereof, or this license to a third party except as expressly provided herein, (c) not remove obliterate, or otherwise defeat any copyright, trademark, patent or proprietary notices, related to the Software (d) not in any form export, re-export, resell, ship or divert or cause to be exported, re-exported,

resold, shipped, or diverted, directly or indirectly, the Software or a direct product thereof to any country which the United States government or any agency thereof at the time of export or re-export requires an export license or other government approval without first obtaining such license or approval.

THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS AND WITHOUT WARRANTY OF ANY KIND INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MOTOROLA BE LIABLE FOR ANY LIABILITY OR DAMAGES OF ANY KIND INCLUDING, WITHOUT LIMITATION, DIRECT OR INDIRECT OR INCIDENTAL OR CONSEQUENTIAL OR PUNITIVE DAMAGES OR LOST PROFITS OR LOSS OF USE ARISING FROM USE OF THE SOFTWARE OR THE PRODUCT REGARDLESS OF THE FORM OF ACTION OR THEORY OF LIABILITY (INCLUDING WITHOUT LIMITATION, ACTION IN CONTRACT, NEGLIGENCE, OR PRODUCT LIABILITY) EVEN IF MOTOROLA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THIS DISCLAIMER OF WARRANTY EXTENDS TO LICENSEE OR USERS OF PRODUCTS AND IS IN LIEU OF ALL WARRANTIES WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE.

Motorola does not represent or warrant that the Software is free of infringement of any third party patents, copyrights, trade secrets, or other intellectual property rights or that Motorola has the right to grant the licenses contained herein. Motorola does not represent or warrant that the Software is free of defect, or that it meets any particular requirements or need of the Licensee, or that it conforms to any documentation, or that it meets any standards.

Motorola shall not be responsible to maintain the Software, provide upgrades to the Software, or provide any field service of the Software. Motorola reserves the right to make changes to the Software without further notice to Licensee.

The Software is not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Software could create a situation where personal injury or death may occur. Should Licensee purchase or use the Software for any such unintended or unauthorized application, Licensee shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the Software.

The term of this Agreement is for as long as Licensee uses the Software for its intended purpose and is not in default of any provisions of this Agreement. Motorola may terminate this Agreement if Licensee is in default of any of the terms and conditions of this Agreement.

This Agreement shall be governed by and construed in accordance with the laws of the State of Arizona and can only be modified in a writing signed by both parties. Licensee agrees to jurisdiction and venue in the State of Arizona.

By using, modifying, installing, compiling, or copying the Software, Licensee acknowledges that this Agreement has been read and understood and agrees to be bound by its terms and conditions. Licensee agrees that this Agreement is the complete and exclusive statement of the agreement between Licensee and Motorola and supersedes any earlier proposal or prior arrangement, whether oral or written, and any other communications relative to the subject matter of this Agreement.

Index

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1–303–675–2140 or 1–800–441–2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3–20–1, Minami–Azabu. Minato–ku, Tokyo 106–8573 Japan. 81–3–3440–3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852–26668334

Technical Information Center: 1–800–521–6274

HOME PAGE: http://www.motorola.com/semiconductors/

MOTOROLA

SDK129/D