Intel Recognition Primitives Library

Reference Manual

Issued in U.S.A.
Order Number 637785-002

Intel Recognition Primitives Library Reference Manual

Order Number: 637785-002

Revision	Revision History	Date
-001	Original issue.	8/95
-002	Added Error Handling chapter and additional FFT functions.	1/96
-003	Added HMM section to end of chapter 7.	4/96
-004	Version 3.0	8/96

How to Use This Online Manual


Click to hide or show subtopics when the bookmarks are shown.


Double-click to jump to a topic when the bookmarks are shown.


Click to display bookmarks.


Click to display thumbnails.


Click to close bookmark or thumbnail view.


Click and use on the page to drag the page in vertical direction.


Click and drag on the page to magnify the view.


Click and drag on the page to reduce the view.


Click and drag the selection cursor on the page.


Click to go to the first page of the manual.


Click to go to the previous page.


Click to go to the next page.


Click to go to the last page.


Click to return back to the previous view. Use this button when you need to go back after using the jump button (see below).


Click to go forward from the previous view.


Click to set 100% of the page view.


Click to display the entire page within the window.


Click to fill the width of the window.


Click to open a dialog to search for a word or multiple words.


Click jump button on manual pages to jump to the related subjects. Use the return back icon above to go back.

Printing an Online File. Select **Print** from the **File** menu to print an online file. The dialog that opens allows you to print full text, range of pages, or selection.

Viewing Multiple Online Manuals. Select **Open** from the **File** menu, and open a .PDF file you need. Select **Cascade** from the **Window** menu to view multiple files.

Resizing the Bookmark Area. Drag the double-headed arrow that appears on the area's border as you pass over it.

Jumping to Topics. Throughout the text of this manual, a title, a phrase that can be used as a reference to a section, or reference itself are printed in green, underlined style to indicate that you can jump to them. To return to the page from which you jumped, use the return back icon in the tool bar.
Try this example:

Vector operations are described in Chapter 3.

If you click the phrase printed in green color, underlined style, the first page of the "Vector Operations" chapter opens. To return to this page, click the  icon in the tool bar.

Intel Recognition Primitives Library Topics

The following is the list of topics presented in the *Intel Recognition Primitives Library Reference Manual*. To jump to a topic, click it.

Overview

.

Contents

Chapter 4 Signal Processing

Chapter 5 Recognition Basics

Figures

Tables

Overview 1

The Intel Recognition Primitives Library provides a set of recognition primitives and feature extraction functions targeted for use by speech and optical character recognition (OCR) application developers.

Manual Organization

This manual describes the functions in the Recognition Primitives Library. Each function is introduced by its name and a short description of its purpose. This is followed by a function prototype and definitions of its arguments. Finally there is a discussion of the algorithm and its implementation.

The chapters included in this manual are:

Related Publications

This manual is designed as a reference for the Intel Recognition Primitives Library. The routines described in this manual are tailored for speech signal analysis rather than general signal analysis. Thus, many of the potential signal processing variations involving complex input types, conjugate-symmetric input, in-place computation, not-in-place

computation, and so on, are not included here. If you need a comprehensive signal processing reference, see the *Intel Signal Processing Library Reference Manual*, order number 630508.

Many signal processing programs require other types of scalar and vector processing, array processing, and linear algebra functions which are not included in the Intel Recognition Primitives. The Intel Math Kernal Library provides such functions with a FORTRAN interface. For more information on this library, see the *Intel Math Kernal Library Reference Manual*, order number 630813.

This manual also contains numerous references to additional textbooks on filters and signal processing.

Notational Conventions

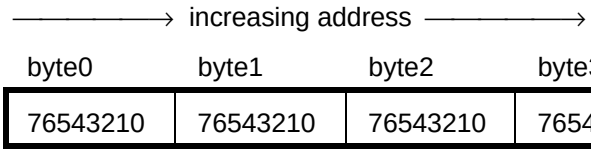
This section describes the notational conventions used by the Intel Recognition Primitives Library and the notational conventions for data types and function names used in this manual.

Data Types

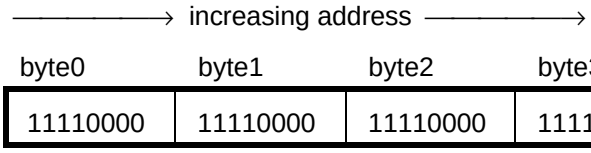
The most common data types used in the library are single precision floating point vectors (32 bits on Win32*) and short integer (16 bits on Win32) vectors - both for input and output. In some cases (for example, distance measures) long integer output is used. While short integers result in compact storage representation for data structures, they need to be augmented by a scaling strategy because of their limited bit capacity (namely, the range of -32768 to +32767 on Win32).

Other data types used in the library are 8-bit signed integer, 8-bit unsigned integer, 4-bit nibble and 1-bit.

Bit position is indicated with reference to the least significant bit in the smallest addressable unit (a byte). The figure below shows the order of bytes and bits within a byte:



Nibble position is indicated as follows:



Data Type Conventions

Many of the functions in the Recognition Primitives Library are available for a variety of integer vectors and for single-precision real and complex vectors. The Recognition Primitives Library distinguishes input vector types by the use of a character code. A character code embedded within the prefix of the function name indicates what type of vector can be used with a particular function. Table 1-1 lists the names of the vector types and their corresponding character codes.

Table 1-1 Vector Types and Corresponding Character Codes

Vector Type	Character Code
bit vector	RLb
4-bit nibble vector	RLn
unsigned byte vector	RLy
signed byte vector	RLt
16-bit integer vector	RLw
16-bit integer complex vector	RLv
single-precision real vector	RLs
single-precision complex vector	RLc
32-bit integer vector	RLi

Additionally, the output of some of the DFT and FFT functions are complex values formatted as a vector of type `WCplx` (for 16-bit integer valued inputs) and `SCplx` (for floating-point valued inputs). The C definitions for `WCplx` and `SCplx` are as follows:

```
typedef struct _WCplx {
    short int real;
    short int imag;
} WCplx;

typedef struct _SCplx {
    float real;
    float imag;
} SCplx;
```

Function Name Conventions

The names of functions in the Recognition Primitives Library always begin with the RL prefix and have the following general format:

RL < *character code* > < *flags* > < *name* > < *mods* > ()

where:

character code	One of the character codes described in Table 1-1 above (b, n, y, t, w, s, c, or i). The character code indicates which function to use with which data type.
flags	The flags field is optional. The only flag currently defined is b, which indicates a block (or vector) variety of the function. A block variety of a function is generally equivalent to multiple invocations of the non-block (scalar) function.
name	Indicates the core functionality, such as Add, Fft, or Cepstral.
mods	The *mods* field is optional and indicates a modification to the core functionality of the function group. Examples of *mods* are Nip (not-in-place) and Tr (truncation).

Integer Scaling

Most of the integer functions in the Recognition Primitives Library perform their internal computations using a higher precision than the integer data types used for input and output. For the Pentium processor, this higher precision is single- and double-precision floating point representation.

These integer functions posses two arguments, *doScaleOutput* and *scaleFactor*, which dictate how the internal representation is converted to integers before output. The *doScaleOutput* and *scaleFactor* arguments are described in greater detail below.

A typical integer function has the following format:

```
Rlw???(..., int doScaleOutput, int *scaleFactor);
```

doScaleOutput Indicates the scaling options to be used in returning the output. The following scaling options are currently allowed:

RL_NO_SCALE
Does not scale the output at all. This option gives the fastest performance. Truncation or wrap-around and other erroneous results will occur when overflow or underflow occur. A *scaleFactor* of 0 is returned.

RL_FIXED_SCALE
The output is always multiplied by $2^{-scaleFactor}$. The *scaleFactor* is returned without any alteration.

RL_AUTO_SCALE
The output is automatically scaled up or down to make the best use of the short integer output representation. Therefore, *scaleFactor* is chosen automatically, the output is multiplied by $2^{-scaleFactor}$, and *scaleFactor* is returned.

RL_SATURATE
When overflow or underflow occurs, the output is clipped to SHRT_MAX (that is, +32767 - long integers are clipped to LONG_MAX) or SHRT_MIN (that is, -32768 - long integers are clipped to LONG_MIN) respectively, otherwise it is not changed. A *scaleFactor* of 0 is returned.

scaleFactor The scale factor (an exponent of 2) that is either specified or chosen automatically depending on the *doScaleOutput* argument. If the option RL_AUTO_SCALE was chosen automatically, the actual output can be obtained from the returned *scaleFactor* as

$actual\_ouput = output * 2^{scaleFactor}$

Error Handling

2

This chapter describes the error handling facility supplied with the Intel Recognition Primitives Library. The Recognition Primitives Library functions report a variety of errors including bad arguments (`NULL` pointers and out-of-range parameters) and out of memory conditions. When a function detects an error, instead of returning a status code, the function signals an error by calling `RLSetStatus()`. This allows the error handling mechanism to be handled separately from the normal flow of the application code. The code is thus cleaner and more compact as shown in this example.

```
maximum = RLybMax(src, n, &position);
/* do error checking */
if(RLGetStatus()<0)
```

The error handling system is hidden within the function `RLybMax()`. Thus, this statement is uncluttered by error handling code and results in a statement which closely resembles a mathematical formula. The error is detected by calling `RLGetStatus()`.

The errors that a function may signal are implementation-dependent. Your application should assume that every library function call may result in some error condition. The Intel Recognition Primitives Library performs extensive error checks (for example, `NULL` pointers, out-of-range parameters, corrupted states) for every library function.

Error macros are provided to simplify the coding for error checking and reporting. You can modify the way your application handles errors by calling `RLRedirectError()` with a pointer to your own error handling function. For more information, see "Adding Your Own Error Handler" later in this chapter. For even more flexibility, you can replace the whole error handling facility with your own code. The source code of the default error handling facility is provided.

There are two versions of Intel Recognition Primitives Library: the debug version and the non-debug version. The debug version detects more errors than the non-debug version (for example, it checks bad parameters). The debug version can be used during application development and the non-debug version for the released application. The non-debug version detects much fewer errors (for example, failure of memory allocation) and is therefore faster.

Error Functions

The following sections describe the error functions in the Intel Recognition Primitives Library.

Error

Performs basic error handling.

```
RLStatus RLError(RLStatus status, const char * func,
const char * context, const char * file, int line)
```

`status`	Code that indicates the type of error (see Table 2-1, "RLError() Status Codes").
`func`	Name of the function where the error occurred.
`context`	Provides additional information about the context in which the error occurred. If the value of `context` is `NULL` or empty, this string will not appear in the error message.
`file`	Name of the source file whith the function text.
`line`	The line number where the error occurred.

Discussion

The `RLError()` function should be called whenever any of the library's functions encounters an error. The actual error reporting will be handled differently, depending on whether the program is running in Windows mode or in console mode. Within each invocation mode, you can set the error mode flag to alter the behavior of the `RLError()` function. See "SetErrMode" (for `RLSetErrMode()`) for more information on the defined error modes.

To simplify the coding for error checking and reporting, the error handling system supplied by the Intel Recognition Primitives Library supports a set of error macros. See "Error Macros" for a detailed description of the error handling macros.

The `RLError()` function calls the default error reporting function. You can change the default error reporting function by calling `RLRedirectError()`. For more information, see "RedirectError".

GetStatus, SetStatus

Gets and sets the error codes which describe the type of error being reported.

```
typedef int RLStatus;

RLStatus RLGetStatus(void);

void RLSetStatus(RLStatus status);
```

status Code that indicates the type of error (see Table 2-1, "RLError() Status Codes").

Discussion

The `RLGetStatus()` and `RLSetStatus()` functions get and set the error status codes which describe the type of error being reported. See "Status Codes" for descriptions of each of the error status codes.

GetErrMode, SetErrMode

Gets and sets the error modes which describe how an error is processed.

```
#define RL_ErrModeLeaf 0
#define RL_ErrModeParent 1
#define RL_ErrModeSilent 2
int RLGetErrMode(void);
void RLSetErrMode(int errMode);
```

errMode Indicates how errors will be processed. The possible values for *errMode* are RL_ErrModeLeaf, RL_ErrModeParent or RL_ErrModeSilent.

Discussion

NOTE. *This section describes how the default error handler handles errors for applications which run in console mode. If your application has a custom error handler, errors will be processed differently than described below.*

The RLSetErrMode() function sets the error modes which describe how errors are processed. The defined error modes are RL_ErrModeLeaf, RL_ErrModeParent and RL_ErrModeSilent.

If you specify RL_ErrModeLeaf, errors are processed in the “leaves” of the function call tree. The RLError() function (in console mode) prints an error message describing *status*, *func*, and *context*. It then terminates the program.

If you specify RL_ErrModeParent, errors are processed in the “parents” of the function call tree. When RLError() is called as the result of detecting an error, an error message will print but the program will not terminate.

Each time a function calls another function, it must check to see if an error has occurred. When an error occurs, the function should call `RLError()` specifying `RL_StsBackTrace`, and then return. The macro `RL_ERRCHK()` may be used to perform both the error check and back trace call. This passes the error "up" the function call tree until eventually some parent function (possibly `main()`) detects the error and terminates the program.

`RL_ErrModeSilent` is similar to `RL_ErrModeParent`, except that error messages are not printed.

`RL_ErrModeLeaf` is the default, and is the simplest method of processing errors. `RL_ErrModeParent` requires more programming effort, but provides more detailed information about where and why an error occurred. All of the functions in the library support both options (that is, they use `RL_ERRCHK()` after function calls). If an application uses the `RL_ErrModeParent` option, it is essential that it checks for errors after all library functions that it calls.

The status code of the last detected error is stored into the internal static variable *`status`* which can be returned by calling `RLGetStatus()`.

ErrorStr

Translates an error or status code into a textual description.

```
const char* RLErrorStr(RLStatus status);
```

status	Code that indicates the type of error (see Table 2-1, "RLError() Status Codes").

Discussion

The function `RLErrorStr()` returns a short string describing *status*. Use this function to produce error messages for users. The returned pointer is a pointer to an internal static buffer that may be over-written on the next call to `RLErrorStr()`.

RedirectError

Assigns a new error handler to call when an error occurs.

```
RLErrCallBack RLRedirectError(RLErrCallBack func);
```

func	Pointer to the function that will be called when an error occurs.

Discussion

The `RLRedirectError()` function assigns a new function to be called when an error occurs in the Intel Recognition Primitives Library. If *func* is `NULL`, `RLRedirectError()` installs the Intel Recognition Primitives Library's default error handler.

The return value of `RLRedirectError()` is a pointer to the previously assigned error handling function.

For the definition of the function typedef `RLErrCallBack`, see the include file `rlerror.h`. See "Adding Your Own Error Handler" for more information on the `RLRedirectError()` function.

Error Macros

The error macros associated with the `RLError()` function are described below.

```
#define RL_ERROR(status, func, context) \
      RLError((status), (func), (context), __FILE__, \
      __LINE__)

#define RL_ERRCHK(func, context) \
      ( (RLGetStatus() >= 0) ? RL_StsOk : \
      RL_ERROR(RL_StsBackTrace,( func),( context)) )

#define RL_ASSERT(expr, func, context)\
       ( (expr) ? RL_StsOk : \
       RL_ERROR(RL_StsInternal, (func), (context)) )

#define RL_RSTERR()  (RLSetStatus(RL_StsOk))
```

context — Provides additional information about the context in which the error occurred. If the value of *context* is `NULL` or empty, this string will not appear in the error message.

expr — An expression that checks for an error condition and returns `FALSE` if an error occurred.

func — Name of the function where the error occurred.

status — Code that indicates the type of error (see Table 2-1, "RLError() Status Codes.")

Discussion

The `RL_ASSERT()` macro checks for the error condition *`expr`* and sets the error status `RL_StsInternal` if the error occurred.

The `RL_ERRCHK()` macro checks to see if an error has occurred by checking the error status. If an error has occurred, `RL_ERRCHK()` creates an error back trace message and returns a non-zero value. This macro should normally be used after any call to a function that might have signaled an error.

The `RL_ERROR()` macro calls the `RLError()` function with current file name and line as last arguments. This macro is used by other error macros. By changing `RL_ERROR()` you can modify the error reporting behavior without changing a single line of source code.

The `RL_RSTERR()` macro resets the error status to `RL_StsOk`, thus clearing any error condition. This macro should be used by an application when it decides to ignore an error condition.

Status Codes

The status codes used by the Intel Recognition Primitives Library are described in Table 2-1. Status codes are integers, not an enumerated type. This allows an application to extend the set of status codes beyond those used by the library itself. Negative codes indicate errors, while non-negative codes indicate success.

Table 2-1 RLError() Status Codes

Status	Code	Description
RL_StsOk	0	No error. The RLError() function will do nothing if called with this status code.
RL_StsBackTrace	-1	Implements a backtrace of the function calls that lead to an error. If RL_ERRCHK() detects that a function call resulted in an error, it calls RL_ERROR() with this status code to provide further context information for the user.
RL_StsError	-2	An error of unknown origin, or of an origin not correctly described by the other error codes.
RL_StsInternal	-3	An internal "consistency" error, often the result of a corrupted state structure. These errors are typically the result of a failed assertion.
RL_StsNoMem	-4	A function attempted to allocate memory using malloc() or a related function and was unsuccessful. The message *context* indicates the intended use of the memory.
RL_StsBadArg	-5	One of the arguments passed to the function is invalid. The message *context* indicates which argument and why.
RL_StsBadFunc	-6	The function is not supported by the implementation, or the particular operation implied by the given arguments is not supported.
RL_StsNoConv	-7	An iterative convergence algorithm failed to converge within a reasonable number of iterations.
RL_StsOverflow	-20	The result of the calculation has been greater than the maximal value of data type.
RL_StsUnderflow	-21	The result of the calculation has been less than the minimal value of data type.

The status of the last error reported is stored into the internal static variable. Its value is the initially `RL_StsOk`. The last status can be set by calling `RLSetStatus()` and can be returned by calling `RLGetStatus()`.

If the application decides to ignore an error, it should reset the last status back to `RL_StsOk` (see `RL_RSTERR()` under "Error Macros"). An application-supplied error handling function must update the last status correctly; otherwise the Intel Recognition Primitives Library might fail.

Errors with status codes `RL_StsBadArg`, `RL_StsOverflow` or `RL_StsUnderflow` are detected only by the debug version of the Intel Recognition Primitives Library. Overflow and underflow cases are not always checked, but only in functions with special integer scaling (see "Interger Scaling" in Chapter 1).

Error Handling Example

Example 2-1 describes the default error handling for a console application. In the example `RLcFft()` represents a library function, `main()` and `appFunc()` represents application code.

The value of the error mode is set to `RL_ErrModeParent`. The `RL_ErrModeParent` option produces a more detailed account of the error conditions.

Example 2-1 Error Functions

```
/* application main function */
main()
{
      SCplx samps[1024]
      RLSetErrMode(RL_ErrModeParent);
      appFunc(5, 45, samps);
      if (RL_ERRCHK("main","compute something"))
      exit(1);
      return 0;
}
/* application subroutine */
void appFunc(int order1, int order2, SCplx *samps)
{
      RLcFft(samps, order1, RL_FORWARD);
      if (RL_ERRCHK("appFunc","compute using order1")) return;
      RLcFft(samps, order2, RL_FORWARD);
      if (RL_ERRCHK("appFunc","compute using order2")) return;
      /* do some more work */
}
/* library function */
void RLcFft(SCplx * samps, int order1, int flags);
{
      if (order > 31) {
            RL_ERROR(RL_StsBadArg, "RLcFft",
               "order must be less than 32");
            return;
      }
      /* code to do some real work goes here */
}
```

When the program is run, it produces the output illustrated in Example 2-2.

Example 2-2 Output for the Error Function Program (RL_ErrModeParent)

```
RPL Error: Bad argument
       in function [rlfft.c:231] RLcFft:
       order must be less than 32
       called from function [test.c:16] appFunc:
              compute using order2
       called from function [test.c:6] main:
              compute something
```

If the program had run with the `RL_ErrModeLeaf` option instead of `RL_ErrModeParent`, only the first three lines of the above output would have been produced before the program terminated.

Adding Your Own Error Handler

The Intel Recognition Primitives Library allows you to define your own error handler. User-defined error handlers are useful if you want your application to send error messages to a destination other than the standard error output stream. For example, you can choose to send error messages to a dialog box if your application is running under a Windows system or you can choose to send error messages to a special log file.

There are two methods of adding your own error handler. In the first method, you can replace the `RLError()` function or the complete error handling library with your own code. Note that this method can only be used at link time.

In the second method, you can use the `RLRedirectError()` function to replace the error handler at run time. The steps below describe how to create your own error handler and how to use the `RLRedirectError()`function to redirect error reporting.

1. Define a function with the function prototype, `RLErrorCallBack()`, as defined by the Intel Recognition Primitives Library.
2. Your application should then call the `RLRedirectError()`function to redirect error reporting for your own function. All subsequent calls to `RLError()`will call your own error handler.
3. To redirect the error handling back to the default handler, simply call `RLRedirectError()`with a `NULL` pointer.

Example 2-3 illustrates a user-defined error handler function, `ownError()`, which simply prints an error message constructed from its arguments and exits.

Example 2-3 A Simple Error Handler

```
RLStatus ownError(RLStatus status, const char *func,
       const char *context, const char *file, int line)
{
       fprintf(stderr, "IRPL error: %s, ", RLErrorStr(status));
       fprintf(stderr, "function %s, ", func ? func : "<unknown>");
       if (line > 0) fprintf(stderr, "line %d, ", line);
       if (file != NULL) fprintf(stderr, "file %s, ", file);
       if (context) fprintf(stderr, "context %s\n", context);
       exit(1);
}

main ()
{
       extern RLErrCallBack ownError;
       /* Redirect errors to your own error handler */
       RLRedirectError(ownError);
       /* Redirect errors back to the default error handler */
       RLRedirectError(NULL);
}
```

Vector Operations

3

The functions described in this chapter perform vector initialization, vector arithmetic, and logical operations on vectors.

Vector Initialization Functions

This section describes the functions in the Intel Recognition Primitives Library which perform vector initialization.

bCopy

Initializes a vector with the contents of a second vector.

```
void RLbbCopy(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
   /* bit vectors */

void RLnbCopy(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
   /* 4-bit nibble vectors */

void RLybCopy(const unsigned char *src, unsigned char
*dst, int n);
   /* unsigned byte vectors */

void RLtbCopy(const signed char *src, signed char *dst,
int n);
   /* signed byte vectors */
```

```
void RLwbCopy(const short int *src, short int *dst, int
n);
    /* 16-bit integer values */
void RLsbCopy(const float *src, float *dst, int n);
    /* single precision; real values */
```

src	Pointer to the source vector used to initialize *dst[i]*.
dst	Pointer to the vector to be initialized.
srcStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the destination vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to copy.

Discussion

The function RL?bCopy() copies the first *n* elements from a source vector *src[i]* into a destination vector *dst[i]* (where 0 <= i < *n*).

bSet

Initializes a vector with a scalar value.

```
void RLbbSet(unsigned char val, unsigned char *dst, int
startPos, int n);
    /* bit vectors */

void RLnbSet(unsigned char val, unsigned char *dst, int
startPos, int n);
    /* 4-bit nibble vectors */

void RLybSet(unsigned char val, unsigned char *dst, int
n);
    /* unsigned byte vectors */

void RLtbSet(signed char val, signed char *dst, int n);
    /* signed byte vectors */

void RLwbSet(short int val, short int *dst, int n);
    /* 16-bit integer values */

void RLsbSet(float val, float *dst, int n);
    /* single precision; real values */
```

val	The value used to initialize *dst[i]*.
dst	Pointer to the vector to be initialized.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to initialize.

Discussion

The function `RL?bSet()` initializes the first *n* elements of the vector *dst[i]* (where 0 <= i < *n*) to contain the same value *val*.

bZero

Initializes a vector to zero.

```
void RLbbZero(unsigned char *dst, int startPos, int n);
   /* bit vectors */

void RLnbZero(unsigned char *dst, int startPos, int n);
   /* 4-bit nibble vectors */

void RLybZero(unsigned char *dst, int n);
   /* unsigned byte vectors */

void RLtbZero(signed char *dst, int n);
   /* signed byte vectors */

void RLwbZero(short int *dst, int n);
   /* 16-bit integer vector */

void RLsbZero(float *dst, int n);
   /* single precision; real vector */
```

dst	Pointer to the vector to be initialized.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to initialize.

Discussion

The function `RL?bZero()` initializes the first *n* elements of the vector *dst[i]* (where 0 <= i < *n*) to 0.

GetBit

Gets a bit value from a vector.

```
unsigned char RLGetBit(const unsigned char *src, int n);
```

src	Pointer to the source vector from which a bit value is extracted.
n	Bit number (do not confuse with start position) from the start of vector *src[i]* aligned on a byte boundary.

Discussion

The function RLGetBit() gets the value of bit *n* from a vector *src[i]*.

SetBit

Initializes the value of a bit within a vector.

```
void RLSetBit(const unsigned char *src, int n, unsigned char val);
```

src	Pointer to the source vector where the bit value is set.
n	Bit number (do not confuse with start position) from the start of vector *src[i]* aligned on a byte boundary.
val	The value used to set the bit.

Discussion

The function RLSetBit() initializes bit *n* of a vector *src[i]* with value *val*.

GetNibble

Gets a nibble value from a vector.

```
unsigned char RLGetNibble(const unsigned char *src, int
n);
```

src	Pointer to the source vector from which the nibble value is extracted.
n	Nibble number (do not confuse with start position) from the start of vector *src[i]* aligned on a byte boundary.

Discussion

The function `RLGetNibble()` gets the value of nibble *n* from a vector *src[i]*.

SetNibble

Initializes the value of a nibble within a vector.

```
void RLSetNibble(const unsigned char *src, int n,
unsigned char val);
```

src	Pointer to the source vector where the nibble value is to be initialized.
n	Nibble number (do not confuse with start position) from the start of vector *src[i]* aligned on a byte boundary.
val	The value used to set the nibble.

Discussion

The function RLSetNibble() initializes nibble *n* of a vector *src[i]* with the value *val*.

Vector Arithmetic Functions

This section describes the Recognition Primitives Library functions which perform basic, element-wise operations between vectors. The library provides two versions of each function. One version performs the operation "in-place," while the other stores the results of the operation in a third vector.

bAdd2

Adds the elements of two vectors.

```
void RLnbAdd2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n, int
doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

void RLybAdd2(const unsigned char *src, unsigned char
*dst, int n, int doScale, int *scaleFactor);
    /* unsigned byte vectors */

void RLtbAdd2(const signed char *src, signed char *dst,
int n, int doScale, int *scaleFactor);
    /* signed byte vectors */

void RLwbAdd2(const short int *src, short int *dst, int
n, int doScale, int *scaleFactor);
    /* 16-bit integer vectors */

void RLsbAdd2(const float *src, float *dst, int n);
    /* single precision; real values */
```

src	Pointer to the vector to be added to *dst[i]*.
dst	Pointer to the vector *dst[i]* which stores the results of the addition *src[i]* + *dst[i]*.
srcStartPos	For nibble vectors, indicates the position of the element within the first byte of the source vector. This value can be 0 or 1 (0 for the least

	significant nibble and 1 for the most significant nibble).
dstStartPos	For nibble vectors, indicates the position of the element within the first byte of the destination vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be added.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?bAdd2() adds the first *n* elements of a source vector *src[i]* to the elements of destination vector *dst[i]* (where 0 <= i < *n*). The results of the operation are stored in *dst[i]*.

bAdd2s

Adds a scalar value to a vector.

```
void RLnbAdd2s(unsigned char val, unsigned char *dst, int
startPos, int n, int doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

void RLybAdd2s(unsigned char val, unsigned char *dst, int
n, int doScale, int *scaleFactor);
    /* unsigned byte vectors */

void RLtbAdd2s(signed char val, signed char *dst, int n,
int doScale, int *scaleFactor);
    /* signed byte vectors */

void RLwbAdd2s(short int val, short int *dst, int n, int
doScale, int *scaleFactor);
    /* 16-bit integer vectors */

void RLsbAdd2s(float val, float *dst, int n);
    /* single precision; real values */
```

val	The value to be added to each element of the vector *dst[i]*.
dst	Pointer to the vector *dst[i]* which stores the results of the addition *val* + *dst[i]*.
startPos	For nibble vectors, indicates the position of the element within the first byte of the vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?bAdd2s()` adds the scalar value *`val`* to each element of the first *`n`* elements of destination vector *`dst[i]`* (where 0 <= i < *`n`*). The results of the operation are stored in *`dst[i]`*.

bAdd3

Adds the elements of two vectors and stores the result in a third vector.

```
void RLnbAdd3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n, int doScale, int *scaleFactor);
   /* 4-bit nibble vectors */

void RLybAdd3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int n, int doScale, int
*scaleFactor);
   /* unsigned byte vectors */

void RLtbAdd3(const signed char *srcA, const signed char
*srcB, signed char *dst, int n, int doScale, int
*scaleFactor);
   /* signed byte vectors */

void RLwbAdd3(const short int *srcA, const short int
*srcB, short int *dst, int n, int doScale, int
*scaleFactor);
   /* 16-bit integer vectors */

void RLsbAdd3(const float *srcA, const float *srcB, float
*dst, int n);
   /* single precision; real values */
```

srcA, *srcB*	Pointers to the vectors whose elements are to be added together.
dst	Pointer to the vector *dst[i]* which stores the results of the addition *srcA[i]* + *srcB[i]*.
srcStartPos	For nibble vectors, indicates the position of the element within the first byte of the source vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

dstStartPos	For nibble vectors, indicates the position of the element within the first byte of the destination vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be added.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?bAdd3() adds the first *n* elements of the source vector *srcA[i]* to the elements of vector *srcB[i]* (where 0 <= i < *n*). The results of the operation are stored in the destination vector *dst[i]*.

bSub2

Subtracts the elements of two vectors.

```
void RLnbSub2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n, int
doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

void RLybSub2(const unsigned char *src, unsigned char
*dst, int n, int doScale, int *scaleFactor);
    /* unsigned byte vectors */

void RLtbSub2(const signed char *src, signed char *dst,
int n, int doScale, int *scaleFactor);
    /* signed byte vectors */

void RLwbSub2(const short int *src, short int *dst, int
n, int doScale, int *scaleFactor);
    /* 16-bit integer vectors */

void RLsbSub2(const float *src, float *dst, int n);
    /* single precision; real vectors */
```

src	Pointer to the vector to be subtracted from *dst[i]*.
dst	Pointer to the vector *dst[i]* which stores the results of the subtraction *dst[i] - src[i]*.
srcStartPos	For nibble vectors, indicates the position of the element within the first byte of the source vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For nibble vectors, indicates the position of the element within the first byte of the destination vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n	The number of elements to be subtracted.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?bSub2() subtracts the first *n* elements of the source vector *src[i]* from the elements of the destination vector *dst[i]* (where 0 <= i < *n*). The results of the operation are stored in *dst[i]*.

bSub2s

Subtracts a scalar value from a vector.

```
void RLnbSub2s(unsigned char val, unsigned char *dst, int
startPos, int n, int doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

void RLybSub2s(unsigned char val, unsigned char *dst, int
n, int doScale, int *scaleFactor);
    /* unsigned byte vectors */

void RLtbSub2s(signed char val, signed char *dst, int n,
int doScale, int *scaleFactor);
    /* signed byte vectors */

void RLwbSub2s(short int val, short int *dst, int n, int
doScale, int *scaleFactor);
    /* 16-bit integer vector */

void RLsbSub2s(float val, float *dst, int n);
    /* single precision; real vector */
```

val	The value to be subtracted from each element of the vector *dst[i]*.
dst	Pointer to the vector *dst[i]* which stores the results of the subtraction *dst[i] - val[i]*.
startPos	For nibble vectors, indicates the position of the element within the first byte of the vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?bSub2s()` subtracts the scalar value *`val`* from each element of the destination vector *`dst[i]`* (where 0 <= i < *`n`*). The results of the operation are stored in *`dst[i]`*.

bSub3

Subtracts the elements of two vectors and stores the result in a third vector.

```
void RLnbSub3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n, int doScale, int *scaleFactor);
   /* 4-bit nibble vectors */

void RLybSub3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int n, int doScale, int
*scaleFactor);
   /* unsigned byte vectors */

void RLtbSub3(const signed char *srcA, const signed char
*srcB, signed char *dst, int n, int doScale, int
*scaleFactor);
   /* signed byte vectors */

void RLwbSub3(const short int *srcA, const short int
*srcB, short int *dst, int n, int doScale, int
*scaleFactor);
   /* 16-bit integer vectors */

void RLsbSub3(const float *srcA, const float *srcB, float
*dst, int n, int doScale, int *scaleFactor);
   /* single precision; real values */
```

srcA, *srcB*	Pointers to the vectors whose elements are to be subtracted from each other.
dst	Pointer to the vector *dst[i]* which stores the results of the subtraction *srcB[i] - srcA[i]*.
srcStartPos	For nibble vectors, indicates the position of the element within the first byte of the source vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

dstStartPos	For nibble vectors, indicates the position of the element within the first byte of the destination vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?bSub3() subtracts the first *n* elements of the source vector *srcA[i]* from the elements of the vector *srcB[i]* (where 0 <= i < *n*). The results of the operation are stored in the destination vector *dst[i]*.

bMpy2

Multiplies the elements of two vectors.

```
void RLnbMpy2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n, int
doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

void RLybMpy2(const unsigned char *src, unsigned char
*dst, int n, int doScale, int *scaleFactor);
    /* unsigned byte vectors */

void RLtbMpy2(const signed char *src, signed char *dst,
int n, int doScale, int *scaleFactor);
    /* signed byte vectors */

void RLwbMpy2(const short int *src, short int *dst, int
n, int doScale, int *scaleFactor);
    /* 16-bit integer vectors */

void RLsbMpy2(const float *src, float *dst, int n);
    /* single precision; real vectors */
```

src	Pointer to the vector to be multiplied with *dst[i]*.
dst	Pointer to the vector *dst[i]* which stores the results of the multiplication *src[i] * dst[i]*.
srcStartPos	For nibble vectors, indicates the position of the element within the first byte of the source vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For nibble vectors, indicates the position of the element within the first byte of the destination vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n	The number of elements to be operated on.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?bMpy2() multiplies the first *n* elements of a source vector *src[i]* with the elements of a destination vector *dst[i]* (where 0 <= i < *n*). The results of the operation are stored in *dst[i]*.

bMpy2s

Multiplies a vector with a scalar value.

```
void RLnbMpy2s(unsigned char val, unsigned char *dst, int
startPos, int n, int doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

void RLybMpy2s(unsigned char val, unsigned char *dst, int
n, int doScale, int *scaleFactor);
    /* unsigned byte vectors */

void RLtbMpy2s(signed char val, signed char *dst, int n,
int doScale, int *scaleFactor);
    /* signed byte vectors */

void RLwbMpy2s(short int val, short int *dst, int n, int
doScale, int *scaleFactor);
    /* 16-bit integer vector */

void RLsbMpy2s(float val, float *dst, int n);
    /* single precision; real vector */
```

`val`	The value to be multiplied with each element of the vector `dst[i]`.
`dst`	Pointer to the vector `dst` which stores the results of the addition `val * dst[i]`.
`startPos`	For nibble vectors, indicates the position of the element within the first byte of the vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
`n`	The number of elements to be operated on.
`doScaleOutput`, `scaleFactor`	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?bMpy2s()` multiplies the scalar value *`val`* with each element of a destination vector *`dst[i]`* (where 0 <= i < *n*). The results of the operation are stored in *`dst[i]`*.

bMpy3

Multiplies the elements of two vectors and stores the result in a third vector.

```
void RLnbMpy3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n, int doScale, int *scaleFactor);
   /* 4-bit nibble vectors */

void RLybMpy3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int n, int doScale, int
*scaleFactor);
   /* unsigned byte vectors */

void RLtbMpy3(const signed char *srcA, const signed char
*srcB, signed char *dst, int n, int doScale, int
*scaleFactor);
   /* signed byte vectors */

void RLwbMpy3(const short int *srcA, const short int
*srcB, short int *dst, int n, int doScale, int
*scaleFactor);
   /* 16-bit integer vectors */

void RLsbMpy3(const float *srcA, const float *srcB, float
*dst, int n);
   /* single precision; real values */
```

srcA, *srcB*	Pointers to the vectors whose elements are to be multiplied together.
dst	Pointer to the vector *dst[i]* which stores the results of the multiplication *srcA[i] * srcB[i]*.
srcStartPos	For nibble vectors, indicates the position of the element within the first byte of the source vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

dstStartPos	For nibble vectors, indicates the position of the element within the first byte of the destination vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?bMpy3() multiplies the first *n* elements of the source vector *srcA[i]* with the elements of the vector *srcB[i]* (where 0 <= i < *n*). The results of the operation are stored in the destination vector *dst[i]*.

bAbs

Computes the absolute value of each vector element.

```
void RLtbAbs(signed char *dst, int n);
    /* signed byte vectors */

void RLwbAbs(short int *dst, int n);
    /* 16-bit integer vector */

void RLsbAbs(float *dst, int n);
    /* single precision; real vector */
```

dst	Pointer to the vector *dst[i]*. The absolute values of the first *n* elements in this vector will be computed.
n	The number of elements to be operated on.

Discussion

The function RL?bAbs() computes the absolute value of the first *n* elements in vector *dst[i]* (where 0 <= i < *n*). The results are stored in *dst[i]*.

bAbs2

Computes the absolute value of each vector element and stores the results in another vector.

```
void RLtbAbs2(const signed char *src, signed char *dst,
int n);
    /* signed byte vectors */

void RLwbAbs2(const short int *src, short int *dst, int
n);
    /* 16-bit integer vector */

void RLsbAbs2(const float *src, float *dst, int n);
    /* single precision; real vector */
```

src	Pointer to the vector *src[i]*. The absolute values of the elements of *src[i]* will be computed.
dst	Pointer to the vector that stores the absolute values of the elements of *src[i]*.
n	The number of elements to be operated on.

Discussion

The function RL?bAbs2() computes the absolute value of the first *n* elements in vector *src[i]* (where 0 <= i < *n*). The results of the operation are stored in the vector *dst[i]*.

bShiftL

Shifts the bits of each element in the input vector to the left by a specified number of bits.

```
void RLnbShiftL(unsigned char *dst, int dstStartPos, int
n, int nShift);
    /* 4-bit nibble vectors */

void RLybShiftL(unsigned char *dst, int n, int nShift);
    /* unsigned byte vectors */

void RLtbShiftL(signed char *dst, int n, int nShift);
    /* signed byte vectors */

void RLwbShiftL(short int *dst, int n, int nShift);
    /* 16-bit integer vectors */
```

dst	Pointer to the vector whose elements are to be shifted.
n	The length of the vector whose elements are to be shifted.
dstStartPos	For nibble vectors, indicates the position of the element within the nibble where the shifting will begin. The position can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
nShift	The number of bits by which the elements of the vector will be shifted.

Discussion

The function RL?bShiftL() shifts the bits of each element in the input vector to the left by *nShift* bits. The vacated bits are filled with zeros. For nibble vectors, the nibble where shifting will begin is indicated by *dstStartPos*.

bShiftR

Shifts the bits of each element in the input vector to the right by a specified number of bits.

```
void RLnbShiftR(unsigned char *dst, int dstStartPos, int
n, int nShift);
    /* 4-bit nibble vectors */

void RLybShiftR(unsigned char *dst, int n, int nShift);
    /* unsigned byte vectors */

void RLtbShiftR(signed char *dst, int n, int nShift);
    /* signed byte vectors */

void RLwbShiftR( short int *dst, int n, int nShift);
    /* 16-bit integer vectors */
```

dst	Pointer to the vector whose elements are to be shifted.
n	The length of the vector whose elements are to be shifted.
dstStartPos	For nibble vectors, indicates the position of the element within the nibble where the shifting will begin. The position can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
nShift	The number of bits by which the elements of the vector will be shifted.

Discussion

The function RL?bShiftR() shifts the bits of each element in the input vector to the right by *nShift* bits. The vacated bits are filled with zeros. For nibble vectors, the nibble where shifting will begin is indicated by *dstStartPos*.

Sum

Computes the sum of all elements in a vector.

```
int RLbSum(unsigned char *dst, int startPos, int n, int
doScale, int *scaleFactor);
    /* bit vectors */

int RLnSum(unsigned char *dst, int startPos, int n, int
doScale, int *scaleFactor);
    /* 4-bit nibble vectors */

int RLySum(unsigned char *dst, int n, int doScale, int
*scaleFactor);
    /* unsigned byte vectors */

int RLtSum(signed char *dst, int n, int doScale, int
*scaleFactor);
    /* signed byte vectors */

int RLwSum(short int *dst, int n, int doScale, int
*scaleFactor);
    /* 16-bit integer vector */

float RLsSum(float *dst, int n);
    /* single precision; real vector */
```

dst	Pointer to the vector whose elements are to be summed.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?Sum()` computes and returns the sum of the first *n* elements of the vector *dst[i]* (where 0 <= i < *n*).

Min

Finds the minimum valued element of a vector.

```
unsigned char RLnMin(unsigned char *src, int startPos,
int n, int *position);
   /* 4-bit nibble vectors */

unsigned char RLyMin(unsigned char *src, int n, int
*position);
   /* unsigned byte vectors */

signed char RLtMin(signed char *src, int n, int
*position);
   /* signed byte vectors */

short int RLwMin(short int *src, int n, int *position);
   /* 16-bit integer vector */

float RLsMin(float *src, int n, int *position);
   /* single precision; real vector */
```

src	Pointer to the vector *src[i]* whose minimum-valued element is to be found.
startPos	For nibble vectors, indicates the position of the element within the first byte of the vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
position	The index of the minimum-valued element.

Discussion

The function RL?Min() finds the minimum-valued element of the first *n* elements of the vector *src[i]* (where 0 <= i < *n*); its index is returned in *position*.

Max

Finds the maximum valued element of a vector.

```
unsigned char RLnMax(unsigned char *src, int startPos,
int n, int *position);
    /* 4-bit nibble vectors */

unsigned char RLyMax(unsigned char *src, int n, int
*position);
    /* unsigned byte vectors */

signed char RLtMax(signed char *src, int n, int
*position);
    /* signed byte vectors */

short int RLwMax(short int *src, int n, int *position);
    /* 16-bit integer vector */

float RLsMax(float *src, int n, int *position);
    /* single precision; real vector */
```

src	Pointer to the vector whose maximum-valued element is to be found.
startPos	For nibble vectors, indicates the position of the element within the first byte of the vector. This value can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.
position	The index of the maximum-valued element.

Discussion

The function `RL?Max()` finds the maximum-valued element of the first *n* elements of the vector *src[i]* (where 0 <= i < *n*); its index is returned in *position*.

Vector Logical Functions

This section describes the Recognition Primitives Library functions which perform basic, element-wise logical operations between vectors. The library provides two versions of each function. One version performs the operation "in-place," while the other stores the results of the operation in a third vector.

bAnd2

ANDs the elements of two vectors.

```
void RLbbAnd2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
    /* bit vectors */

void RLnbAnd2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
    /* 4-bit nibble vectors */

void RLybAnd2(const unsigned char *src, unsigned char
*dst, int n);
    /* unsigned byte vectors */

void RLwbAnd2(const short int *src, short int *dst, int
n);
    /* 16-bit integer vectors */
```

src — Pointer to the vector to be bitwise ANDed with elements of vector *dst[i]*.

dst — Pointer to the vector *dst[i]* which stores the results of the AND operation *src[i]* AND *dst[i]*.

srcStartPos — For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it

	can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the destination vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.

Discussion

The function RL?bAnd2() performs an element-wise logical AND of the first *n* elements of a source vector *src[i]* with the first *n* elements of the destination vector *dst[i]* (where 0 <= i < *n*). The results are stored in *dst[i]*. Each pair of elements is bit-wise ANDed.

bAnd2s

ANDs the elements of a vector with a scalar value.

```
void RLnbAnd2s(const unsigned char val, unsigned char
*dst, int startPos, int n);
    /* 4-bit nibble vectors */

void RLybAnd2s(const unsigned char val, unsigned char
*dst, int n);
    /* unsigned byte vectors */

void RLwbAnd2s(const short int val, short int *dst, int
n);
    /* 16-bit integer vectors */
```

val	The scalar which is ANDed with each vector element.
dst	Pointer to the vector *dst[i]* which stores the results of the AND operation *dst[i]* AND *val*.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.

Discussion

The function `RL?bAnd2s()` performs an element-wise logical `AND` of the scalar *`val`* with the first *`n`* elements of a destination vector *`dst[i]`*. The results are stored in *`dst[i]`* (where 0 <= i < *`n`*). The scalar is bit-wise `AND`ed with each of the first *`n`* elements of the vector.

bAnd3

ANDs the elements of two vectors and stores the result in a third vector.

```
void RLbbAnd3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n);
    /* bit vectors */

void RLnbAnd3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n);
    /* 4-bit nibble vectors */

void RLybAnd3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int n);
    /* unsigned byte vectors */

void RLwbAnd3(const short int *srcA, const short int
*srcB, short int *dst, int n);
    /* 16-bit integer vectors */
```

srcA, *srcB*	Pointers to the vectors whose elements are to be bitwise ANDed.
dst	Pointer to the vector *dst[i]* which stores the results of the AND operation *srcA[i]* AND *srcB[i]*.
srcStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the

destination vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n — The number of elements to be operated on.

Discussion

The function RL?bAnd3() performs an element-wise logical AND of the first *n* elements of a source vector *srcA[i]* with the first *n* elements of another vector *srcB[i]* (where 0 <= i < *n*). The results are stored in *dst[i]*. Each pair of elements is bit-wise ANDed.

bXor2

XORs the elements of two vectors.

```
void RLbbXor2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
    /* bit vectors */

void RLnbXor2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
    /* 4-bit nibble vectors */

void RLybXor2(const unsigned char *src, unsigned char
*dst, int n);
    /* unsigned byte vectors */

void RLwbXor2(const short int *src, short int *dst, int
n);
    /* 16-bit integer vectors */
```

src	Pointer to the vector to be bitwise XORed with *dst[i]*.
dst	Pointer to the vector *dst[i]* which stores the results of the XOR operation *src[i]* XOR *dst[i]*.
srcStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the destination vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n	The number of elements to be operated on.

Discussion

The function RL?bXor2() performs an element-wise logical XOR of the first *n* elements of a source vector *src[i]* with the first *n* elements of a destination vector *dst[i]* (where 0 <= i < *n*). The results of the operation are stored in *dst[i]*. Each pair of elements is bit-wise XORed.

bXor2s

XORs the elements of a vector with a scalar value.

```
void RLnbXor2s(const unsigned char val, unsigned char
*dst, int startPos, int n);
    /* 4-bit nibble vectors */

void RLybXor2s(const unsigned char val, unsigned char
*dst, int n);
    /* unsigned byte vectors */

void RLwbXor2s(const short int val, short int *dst, int
n);
    /* 16-bit integer vectors */
```

val	The scalar which is XORed with each vector element.
dst	Pointer to the vector *dst[i]* which stores the results of the XOR operation *dst[i]* XOR *val[i]*.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.

Discussion

The function RL?bXor2s() performs an element-wise logical XOR of the scalar *val* with the first *n* elements of a destination vector *dst[i]* (where 0 <= i < *n*). The results of the operation are stored in *dst[i]*. The scalar is bit-wise XORed with each element of the vector.

bXor3

XORs the elements of two vectors and stores the result in a third vector.

```
void RLbbXor3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n);
    /* bit vectors */

void RLnbXor3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n);
    /* 4-bit nibble vectors */

void RLybXor3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int n);
    /* unsigned byte vectors */

void RLwbXor3(const short int *srcA, const short int
*srcB, short int *dst, int n);
    /* 16-bit integer vectors */
```

srcA, *srcB* — Pointers to the vectors whose elements are to be bitwise XORed.

dst — Pointer to the vector *dst[i]* which stores the results of the XOR operation *srcA[i]* XOR *srcB[i]*.

srcStartPos — For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

dstStartPos — For packed bit and nibble vectors, indicates the position of the element within the first byte of the

destination vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n The number of elements to be operated on.

Discussion

The function RL?bXor3() performs an element-wise logical XOR of the first *n* elements of a source vector *srcA[i]* with the first *n* elements of another vector *srcB[i]* (where 0 <= *i* < *n*). The results of the operation are stored in *dst[i]*. Each pair of elements is bit-wise XORed.

bOr2

ORs the elements of two vectors.

```
void RLbbOr2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
   /* bit vectors */

void RLnbOr2(const unsigned char *src, unsigned char
*dst, int srcStartPos, int dstStartPos, int n);
   /* 4-bit nibble vectors */

void RLybOr2(const unsigned char *src, unsigned char
*dst, int n);
   /* unsigned byte vectors */

void RLwbOr2(const short int *src, short int *dst, int
n);
   /* 16-bit integer vectors */
```

src — Pointer to the vector to be bitwise ORed with elements of *dst[i]*.

dst — Pointer to the vector *dst[i]* which stores the results of the OR operation *src[i]* OR *dst[i]*.

srcStartPos — For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

dstStartPos — For packed bit and nibble vectors, indicates the position of the element within the first byte of the destination vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n	The number of elements to be operated on.

Discussion

The function RL?bOr2() performs an element-wise logical OR of the first *n* elements of a source vector *src[i]* with the first *n* elements of a destination vector *dst[i]* (where 0 <= *i* < *n*). The results are stored in *dst[i]*. Each pair of elements is bit-wise ORed.

bOr2s

ORs the elements of a vector with a scalar value.

```
void RLnbOr2s(const unsigned char val, unsigned char
*dst, int startPos, int n);
    /* 4-bit nibble vectors */

void RLybOr2s(const unsigned char val, unsigned char
*dst, int n);
    /* unsigned byte vectors */

void RLwbOr2s(const short int val, short int *dst, int
n);
    /* 16-bit integer vectors */
```

val	The scalar which is ORed with each vector element.
dst	Pointer to the vector *dst[i]* which stores the results of the OR operation *dst[i]* OR *val*.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.

Discussion

The function RL?bOr2s() performs an element-wise logical OR of the scalar *val* with the first *n* elements of a destination vector *dst[i]* (where 0 <= *i* < *n*). The results are stored in *dst[i]*. The scalar is bit-wise ORed with each element of the vector.

bOr3

ORs the elements of two vectors and stores the result in a third vector.

```
void RLbbOr3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n);
    /* bit vectors */

void RLnbOr3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int srcStartPos, int
dstStartPos, int n);
    /* 4-bit nibble vectors */

void RLybOr3(const unsigned char *srcA, const unsigned
char *srcB, unsigned char *dst, int n);
    /* unsigned byte vectors */

void RLwbOr3(const short int *srcA, const short int
*srcB, short int *dst, int n);
    /* 16-bit integer vectors */
```

srcA, *srcB*	Pointers to the vectors whose elements are to be bitwise OReed.
dst	Pointer to the vector *dst[i]* which stores the results of the OR operation *srcA[i]* OR *srcB[i]*.
srcStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the source vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
dstStartPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the destination vector. For bit vectors this value can

be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).

n	The number of elements to be operated on.

Discussion

The function `RL?bOr3()` performs an element-wise logical `OR` of the first *n* elements of a source vector *srcA[i]* with the first *n* elements of another vector *srcB[i]* (where 0 <= *i* < *n*). The results are stored in *dst[i]*. Each pair of elements is bit-wise `OR`ed.

bNot

Performs a logical NOT of the elements of a vector.

```
void RLbbNot(unsigned char *dst, int startPos, int n);
    /* bit vectors */

void RLnbNot(unsigned char *dst, int startPos, int n);
    /* 4-bit nibble vectors */

void RLybNot(unsigned char *dst, int n);
    /* unsigned byte vectors */

void RLwbNot(short int *dst, int n);
    /* 16-bit integer vectors */
```

dst	Pointer to the vector *dst[i]* which stores the results of the logical operation NOT *dst[i]*.
startPos	For packed bit and nibble vectors, indicates the position of the element within the first byte of the vector. For bit vectors this value can be 0 through 7 (0 for the least significant bit and 7 for the most significant bit) and for nibble vectors it can be 0 or 1 (0 for the least significant nibble and 1 for the most significant nibble).
n	The number of elements to be operated on.

Discussion

The function RL?bNot() performs a bit-wise logical NOT of the first *n* elements of the vector *dst[i]* (where 0 <= *i* < *n*). The results of the operation are stored in the vector *dst[i]*.

Signal Processing

4

The functions described in this chapter perform signal processing. The following signal processing tasks are supported.

- Windowing (Bartlett, Hamming, etc.)
- Fast Fourier Transform (FFT)
- Signal Pre-emphasis
- Cepstral Analysis

Windowing Functions

This section describes several of the windowing functions commonly used in digital signal processing. Windowing refers to the weighting applied to the individual points in the `N`-point signal frame. It is specified by a transfer function of the form `h(n) = f(n)` where *f* is a function of *n*. For a speech signal of a given window length (that is, `n`) it is usually desirable to have a wide passband and a large attenuation outside the passband. The windows listed in Table 4-1 are supported:

Table 4-1 Window Transfer Functions

Window Name	Window Transfer Function	
Hamming	h(n) = 0.54 - 0.46 * cos(2*pi*n/(N-1)) = 0	0 <= n <= N-1 otherwise
Hann	h(n) = 0.5 - 0.5*cos(2*pi*n/(N-1)) = 0	0 <= n <= N-1 otherwise
Bartlett (triangle)	h(n) = 2n/(N-1) = 2 - 2n/(N-1) = 0	0 <= n <= (N-1)/2 (N-1)/2 < n <= N-1 otherwise
Blackman	h(n) = 0.42 - 0.5*cos(2*pi*n/(N-1)) + 0.08*cos(4*pi*n/(N-1)) = 0	0 <= n <= N-1 otherwise

Example 4-1 shows the code for windowing a signal and taking its FFT.

Example 4-1 Window and FFT a Single Frame of a Signal

```
/* Window and FFT a single frame of a signal.
 * Note that output size = N/2 + 2 where N is the input size
 * The output of the FFT is in conjugate-symmetric format (see under
 */ “Fast Fourier Transform”.

float  xTime[256];
SCplx  xFreq[128];

/* Insert code here to put time-domain samples in xTime */


RLsWinHamming(xTime, 128);
RLsRealFftNip(xTime, xFreq, 7, RL_FORWARD);
```

The windowing functions save the window coefficients internally. Thus, the window coefficients do not have to be computed every time the function is called. However, the coefficients depend on the input size so they need to be recomputed whenver the input size changes. If the same windowing function needs to be called for different sized inputs, then it is better to first calculate the window by calling one of the windowing

functions (RL?WinHamming(), for example) on a vector with all elements set to 1.0. This vector can then be multiplied with the input signal vector to get the windowed signal vector.

WinBartlett

Multiplies a vector by a Bartlett windowing function.

```
void RLwWinBartlett(short int *vect, int n, int
doScaleOutput, int *scaleFactor);
    /* 16-bit integer vector */
void RLsWinBartlett(float *vect, int n);
    /* single precision; real vector */
```

vect	Pointer to the vector to be multiplied by the chosen windowing function.
n	The length of the vector *vect[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?WinBartlett() multiplies a vector by the Bartlett (triangle) window. To obtain the window samples themselves, set all of the elements of the vector *vect[n]* to unity.

WinBlackman

Multiplies a vector by a Blackman windowing function.

```
void RLwWinBlackman(short int *vect, int n, int
doScaleOutput, int *scaleFactor);
    /* 16-bit integer vector */

void RLsWinBlackman(float *vect, int n);
    /* single precision; real vector */
```

vect	Pointer to the vector to be multiplied by the chosen windowing function.
n	The length of the vector *vect[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?WinBlackman() multiplies a vector by the Blackman window. To obtain the window samples themselves, set all of the elements of the vector *vect[n]* to unity.

WinHamming

Multiplies a vector by a Hamming windowing function.

```
void RLwWinHamming(short int *vect, int n, int
doScaleOutput, int *scaleFactor);
    /* 16-bit integer vector */

void RLsWinHamming(float *vect, int n);
    /* single precision; real vector */
```

vect	Pointer to the vector to be multiplied by the chosen windowing function.
n	The length of the vector *vect[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?WinHamming() multiplies a vector by the Hamming window. To obtain the window samples themselves, set all of the elements of the vector *vect[n]* to unity.

WinHann

Multiplies a vector by a Hann windowing function.

```
void RLwWinHann(short int *vect, int n, int
doScaleOutput, int *scaleFactor);
   /* 16-bit integer vector */

   void RLsWinHann(float *vect, int n);
   /* single precision; real vector */
```

vect	Pointer to the vector to be multiplied by the chosen windowing function.
n	The length of the vector *vect[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?WinHann() multiplies a vector by the Hann window. To obtain the window samples themselves, set all of the elements of the vector *vect[n]* to unity.

FreeWinTbls

Frees all internal memory allocated by windowing functions.

```
void RLwFreeWinHammingTbls (void);
void RLsFreeWinHammingTbls (void);
void RLwFreeWinHanningTbls (void);
void RLsFreeWinHammingTbls (void);
void RLwFreeWinBlackmanTbls (void);
void RLsFreeWinBlackmanTbls (void);
void RLwFreeWinBartlettTbls (void);
void RLsFreeWinBartlettTbls (void);
```

Discussion

The function `RL?FreeWin..Tbls()` frees all internal memory that was allocated for windowing transfer function tables during windowing evaluations.

Fast Fourier Transforms

This section describes the functions which compute the Fast Fourier Transform (FFT). The minimal complete set of FFT functions for float and 16-bit integer data is provided. FFT functions allocate internal memory for twiddle factors and bit-reversed indices. The function for deallocating internal memory dedicated to FFTs is provided.

Format Descriptions

The input and output of the complex-valued FFT are formatted as a vector of type `WCplx` (for 16-bit integer-valued inputs) and `SCplx` (for floating-point valued inputs). Functions are provided in the library to obtain the magnitude, power spectrum, log magnitude or log-power spectrum of the complex output.

The C type definitions for `WCplx` and `SCplx` are as follows:

```
typedef struct _WCplx {
    short int re;
    short int im;
} WCplx;

typedef struct _SCplx {
    float re;
    float im;
} SCplx;
```

The input of the real-valued FFT is a vector of type `short int` (for 16-bit integer-valued inputs) and `float` (for floating-point valued inputs). If complex values `y[i]`, `i=0,...,n, n=`2^{order} are the output of a real-valued FFT, then `y[0]`, `y[n/2]` are real, and `y[i]` and `y[n-i], i=0,.,n/2-1` are complex-conjugate. The result of a real-valued FFT is stored in a real vector of size `n` in the following order (complex-conjugate format):

0	1	2	3	...	n-2	n-1
y[0]	y[n/2]	y[1].re	y[1].im	...	y[n/2-1].re	y[n/2-1].im

For `order=0` a vector in complex-conjugate format contains one element. This format coincides with the PERM format of the NSP library. If the

`float` vector `samps` contains the real-valued FFT output in complex-conjugate format, other outputs can be computed as follows:

```
Csamps = (SCplx*)samps;
Csamps[n/2].re = Csamps[0].im;
Csamps[n/2].im = Csamps[0].im = (float)0.0;
for (i=1; i<n/2; i++) {
    Csamps[n-i].re = Csamps[i].re;
    Csamps[n-i].im = -Csamps[i].im;
} /* for */
```

The input of the complex-conjugate FFT is a vector of type `short int` (for 16-bit integer-valued inputs) and `float` (for floating-point valued inputs) in complex-conjugate format. The output is a vector of the same length.

Fft

Computes the forward or inverse Fast Fourier Transform (FFT) of a complex signal in-place.

```
void RLvFft (WCplx *samps, int order, int flags, int
doScaleOutput, int *scaleFactor);
   /* In-place transform for 16-bit integer complex
      vector. Computes the 16-bit integer complex
      output */

void RLcFft (SCplx *samps, int order, int flags);
   /* In-place transform for single-precision complex
      vector. Computes the single-precision complex
      output */
```

samps	Input vector for in-place transform. The output is written to the same vector.
order	The size of the transform expressed as a power of 2. The length of *samps* is expected to be 2^{order}.
flags	Options for the transform. The following options are currently supported: RL_FORWARD — Forward transform RL_INVERSE — Inverse transform RL_INVERSE_NOSCALE — Inverse transform without scaling (that is, the transform output is not multiplied by 1/N where N is the length of transform). RL_FAST — Call the faster but lower accuracy FFT code. Only for integer FFT functions.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?Fft()` performs a Fast Fourier Transform (FFT) on the complex input vector *`samps`*. The computation is done in-place and the complex output vector is written back to *`samps`*. Internally, a radix-4 algorithm is used.

FftNip

Computes the forward or inverse Fast Fourier Transform (FFT) of a complex signal not-in-place.

```
void RLvFftNip (const WCplx *inSamps, WCplx *outSamps,
int order, int flags, int doScaleOutput, int
*scaleFactor);
   /* Not-in-place transform for 16-bit integer complex
      vector. Computes the 16-bit integer complex
      output */
void RLcFftNip (const SCplx *inSamps, SCplx *outSamps,
int order, int flags);
   /* Not-in-place transform for single-precision complex
       vector. Computes the single-precision complex
       output */
```

inSamps	Input vector for not-in-place transform.
outSamps	Output vector for not-in-place transform.
order	The size of the transform expressed as a power of 2. The length of the input and output vectors is expected to be 2^{order}.
flags	Options for the transform. The following options are currently supported:

RL_FORWARD	Forward transform
RL_INVERSE	Inverse transform
RL_INVERSE_NOSCALE	Inverse transform without scaling (that is, the transform output is not multiplied by 1/N where N is the length of transform).

	`RL_FAST` Call the faster but lower accuracy FFT code. Only for integer FFT functions.
`doScaleOutput`, *`scaleFactor`*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?FftNip()` performs a Fast Fourier Transform (FFT) on the complex input vector *`inSamps`*. The complex output vector is written to *`outSamps`*. Internally, a radix-4 algorithm is used.

RealFft

Computes the forward or inverse Fast Fourier Transform (FFT) of an integer- or real-valued signal in-place.

```
void RLwRealFft (short int *samps, int order, int flags,
int doScaleOutput, int *scaleFactor);
    /* In-place transform for 16-bit integer vector.
       Computes the 16-bit integer output vector in
       complex-conjugate format */

void RLsRealFft (float *samps, int order, int flags);
    /* In-place transform for single-precision real
       vector. Computes the single-precision output
       vector in complex-conjugate format */
```

samps — Input vector for in-place transform. The output is written to the same vector.

order — The size of the transform expressed as a power of 2. The length of *samps* is expected to be 2^{order}.

flags — Options for the transform. The following options are currently supported:

RL_FORWARD	Forward transform
RL_INVERSE	Inverse transform
RL_INVERSE_NOSCALE	Inverse transform without scaling (that is, the transform output is not multiplied by 1/N, where N is the length of the transform).
RL_FAST	Call the faster but lower accuracy FFT code. Only for integer FFT functions.

doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?RealFft()` performs a Fast Fourier Transform (FFT) on the real input vector *samps*. The computation is done in-place and the output vector in complex-conjugated format is written back to *samps*. Internally, a radix-4 algorithm is used.

RealFftNip

Computes the forward or inverse Fast Fourier Transform (FFT) of an integer- or real-valued signal not-in-place.

```
void RLwRealFftNip (const short int *inSamps, short int
*outSamps, int order, int flags, int doScaleOutput, int
*scaleFactor);
   /* Not-in-place transform for 16-bit integer vector.
      Computes the 16 bit integer output vector in
      complex-conjugated format */
void RLsRealFftNip (const float *inSamps, float
*outSamps, int order, int flags);
   /* Not-in-place transform for single-precision real
      vector. Computes the single-precision output
      vector in complex-conjugated format */
```

inSamps	Input vector for not-in-place transform.
outSamps	Output vector for not-in-place transform.
order	The size of the transform expressed as a power of 2. The length of the input and output vectors is expected to be 2^{order}.
flags	Options for the transform. The following options are currently supported:

RL_FORWARD	Forward transform
RL_INVERSE	Inverse transform
RL_INVERSE_NOSCALE	Inverse transform without scaling (that is, the transform output is not multiplied by 1/N, where N is the length of the transform).

`RL_FAST`	Call the faster but lower accuracy FFT code. Only for integer FFT functions.
`doScaleOutput`, *`scaleFactor`*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?RealFftNip()` performs a Fast Fourier Transform (FFT) on the real input vector *`inSamps`*. The output vector in complex-conjugate format is written to *`outSamps`*. Internally, a radix-4 algorithm is used.

In Example 4-2 the `RLsRealFft()` function is used to calculate the FFT of a 128-point real input signal.

Example 4-2 Using RLsRealFft() to Perform the FFT

```
/* Calculate the FFT of a 128-point real input signal
 * Input signal is in x, output is also in x
 * Order of the FFT is 7 (log-base-2 of 128).
 * Output size is N/2 float values, because only half
 * the FFT is generated (the FFT of a real signal being
 * conjugate-symmetric). The FFT is done in-place.
 */

float  x[128];

/* Insert code here to put the 128 samples in x     */
RLsRealFft(x, 7, RL_FORWARD);
```

CcsFft

Computes the forward or inverse Fast Fourier Transform (FFT) of an integer- or real-valued complex-conjugated signal in-place.

```
void RLwCcsFft (short int *samps, int order, int flags,
int doScaleOutput, int *scaleFactor);
    /* In-place transform for 16-bit integer vector in
       complex-conjugated format. Computes the 16-bit
       integer output vector */

void RLsCcsFft (float *samps, int order, int flags);
    /* In-place transform for single-precision real vector
       in complex-conjugated format. Computes the single-
       precision output vector */
```

samps — Input vector for in-place transform. The output is written to the same vector.

order — The size of the transform expressed as a power of 2. The length of *samps* is expected to be 2^{order}.

flags — Options for the transform. The following options are currently supported:

RL_FORWARD	Forward transform
RL_INVERSE	Inverse transform
RL_INVERSE_NOSCALE	Inverse transform without scaling (that is, the transform output is not multiplied by 1/N, where N is the length of the transform).
RL_FAST	Call the faster but lower accuracy FFT code. Only for integer FFT functions.

doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?CcsFft()` performs a Fast Fourier Transform (FFT) on *samps*, the input vector in complex-conjugate format. The computation is done in-place and the output real vector is written back to *samps*. Internally, a radix-4 algorithm is used.

CcsFftNip

Computes the forward or inverse Fast Fourier Transform (FFT) of an integer- or real-valued complex-conjugated signal not-in-place.

```
void RLwCcsFftNip (const short int *inSamps, short int
*outSamps, int order, int flags, int doScaleOutput, int
*scaleFactor);
    /* Not-in-place transform for 16-bit integer vector in
       complex-conjugated format. Computes the 16-bit
       integer output vector */

void RLsCcsFftNip (const float *inSamps, float *outSamps,
int order, int flags);
    /* Not-in-place transform for single-precision real
       vector in complex-conjugated format. Computes the
       single-precision output vector */
```

inSamps	Input vector for not-in-place transform.
outSamps	Output vector for not-in-place transform.
order	The size of the transform expressed as a power of 2. The length of the input and output vectors is expected to be 2^{order}.
flags	Options for the transform. The following options are currently supported:

RL_FORWARD	Forward transform
RL_INVERSE	Inverse transform
RL_INVERSE_NOSCALE	Inverse transform without scaling (that is, the transform output is not multiplied by 1/N, where N is the length of the transform).

	`RL_FAST` Call the faster but lower accuracy FFT code. Only for integer FFT functions.
`doScaleOutput`, *`scaleFactor`*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function `RL?CcsFftNip()` performs a Fast Fourier Transform (FFT) on *`inSamps`*, the input vector in complex-conjugate format. The output real vector is written to *`outSamps`*. Internally, a radix-4 algorithm is used.

FreeFftTbls

Frees all internal memory allocated for FFT computation.

```
void RLFreeFftTbls (void);
```

Discussion

The function `RLFreeFftTbls()` frees all internal memory that was allocated for twiddle factors and bit-reversed indices tables during FFT evaluation.

Speech Specific Signal Processing

The functions described in this section perform signal processing and feature extraction operations that are commonly done on speech signals. These functions are

- Signal pre-emphasis
- Cepstral analysis

Signal Pre-emphasis

The spectrum of voiced speech normally exhibits an overall -6dB/octave roll-off due to the effects of lip radiation and the spectral trend of the voiced excitation source. Pre-emphasis refers to the compensation for this roll-off by preprocessing the signal to give it a +6dB/octave boost in the appropriate spectral range.

Preemphasize

Pre-emphasizes the signal using a first order filter with a transfer function H(z) = 1 - az^{-1} .

```
void RLwPreemphasize(short int *vect, float a, int n, int
doScaleOutput, int *scaleFactor);
   /* 16-bit integer vectors */

   void RLsPreemphasize(float *vect, float a, int n);
   /* single precision; real vectors */
```

vect	The input vector that needs pre-emphasis.
a	The coefficient used for the first order transfer function. Usually a value of 0.95 is chosen for speech signals.
n	The length of the input vector *vect[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RLb?Preemphasize() pre-emphasizes the input signal contained in the vector *vect[n]*. The resulting values are stored back into the same vector. A high-pass filtering for pre-emphasis is implemented by the difference equation:

$y(n) = x(n) - a * x(n-1)$

where x and y are the input and output respectively.

In z-transform notation, the transform function can be written as:

$H(z) = Y(z)/X(z) = 1 - az^{-1}$

Usually a value of 0.95 is used for the value of *a*.

FreePreemphasizeTbls

Frees all internal memory allocated by the RLb?Preemphasize functions.

```
void RLFreePreemphasizeTbls (void);
```

Discussion

The function `RLFreePreemphasizeTbls()` frees all internal memory that was allocated by `RLb?Preemphasize` functions.

Cepstral Analysis

Cepstral analysis (also known as cepstral truncation) is a technique for removing the pitch ripple from the high resolution speech spectra. Voiced speech can be modeled as the convolution of the vocal tract response with the excitation source. Let `x(n)` represent the voiced speech signal, `h(n)` represent the vocal tract response, and `p(n)` represent the excitation signal. Then `x(n)` can be written as follows:

```
x(n) = p(n) * h(n)
```

where "`*`" is the convolution operator.

NOTE. *Note that this equation does not take lip radiation effects into account.*

The goal of cepstral analysis for speech feature extraction is to obtain the vocal tract response after removing the pitch ripple. One straightforward way of doing this is to filter the log-magnitude (or log-energy) of the signal with an inverse FFT. This is followed by truncation of the coefficients beyond the pitch frequency and then a forward FFT. However, a more common variation is cepstral smoothing using a Discrete Cosine Transform (DCT) with the coefficients placed on a mel-scale. In this library, the Mel-frequency Cepstral Coefficients (MFCC) are implemented using the DCT of filter-banked FFT spectra.

CepstralMFCC

Compute mel-scaled cepstral coefficients by cepstral smoothing with a DCT (Discrete Cosine Transform) on triangular bandpass filter bank outputs.

```
void RLwCepstralMFCC(BOOL doFft, short int *vect,
wMelFilters_t *filters, short int *ceps, int nceps, int
doScaleOutput, int *scaleFactor);
       /* 16-bit integer vectors */

void RLsCepstralMFCC(BOOL doFft, float *vect,
sMelFilters_t *filters, float *ceps, int nceps);
    /* single precision; real vectors */
```

doFft	A boolean. If `true`, an FFT followed by a log-magnitude operation is performed on the input vector. If `false`, it is assumed that the input vector is the log-magnitude (or log-power-spectrum) of the FFT of some signal and at least `n/2 + 1` (where `n` is the size of the FFT) elements are expected in the input vector.
vect	The input vector.
filters	Pointer to the `?MelFilters_t` data structure (created by a call to `RL?MFCCInit()`) containing the filter specifications.
ceps	The output vector in which the extracted MFCC cepstral coefficients will be stored.
nceps	The number of cepstral MFCC cepstral coefficients to be extracted.

doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?CepstralMFCC() computes the Mel-Frequency Cepstral Coefficients (MFCC). An FFT of the input signal is taken and the logarithm (base 10) of its magnitude is then filtered by a filter bank of mel-spaced triangular bandpass filters. The filters are equally spaced on a mel-scale defined by:

$$Mel(f) = 2595 \log_{10}(1 + f/700)$$

where f is the frequency on the linear scale.

The MFCC, using a Discrete Cosine Transform (DCT) of the filter outputs, are then computed as

$$MFCC_i = \sum_k X_k \cos(i(k - 0.5)\pi/20), \quad i=1,2,..M, \quad k = 1,..K$$

where X_k is the log-energy output of the kth filter, M = *nceps* and K = the number of bandpass filters.

MFCCInit

Creates and initializes the data structure containing the triangular band pass filter specifications and weights.

```
wMelFilters_t *RLwMFCCInit(int order, int nBandPass, int
startFreq, int endFreq, int samplingFreq, BOOL
doLinearInitially, int nLinearFilters, int
maxLinearFreq);
   /* 16-bit integer vectors */

sMelFilters_t *RLsMFCCInit(int order, int nBandPass, int
startFreq, int endFreq, int samplingFreq, BOOL
doLinearInitially, int nLinearFilters, int
maxLinearFreq);
   /* single precision; real vectors */
```

order	The order (expressed as a power of 2) of the input vector (and also the order of the FFT) that will be passed to the function `RLsCepstralMFCC()`.
nBandPass	The number of triangular bandpass filters to be created.
startFreq	The start frequency of the first band pass filter.
endFreq	The end frequency of the last band pass filter.
samplingFreq	The sampling frequency of the input (speech signal) vector that will be passed to the function `RLsCepstralMFCC()` along with the filter data structure.

doLinearInitially	A boolean. If `true`, the function creates the first few band pass filters uniformly placed on a linear scale. If `false`, all filters are uniformly placed on a mel scale.
nLinearFilters	The number of filters to be placed on a linear scale if argument *doLinearInitially* is `true`.
maxLinearFreq	The end frequency of the last linearly placed filter if argument *doLinearInitially* is `true`.

Discussion

The function `RL?MFCCInit()` is called once to construct and initialize the data structure `?MelFilters_t`. The `?MelFilters_t` structure is required to extract the MFCC cepstral coefficients by calls to the function `RL?CepstralMFCC()`. A filter bank of *nBandPass* mel-spaced triangular bandpass filters is created. The filters are equally spaced on a mel-scale (starting from the frequency *startFreq* and ending at the frequency *endFreq*) defined by:

$$Mel(f) = 2595\log_{10}(1 + f/700)$$

where *f* is the frequency on the linear scale.

When *doLinearInitially* is `true`, the first *nLinearFilters* are placed uniformly on a linear frequency scale starting from *startFreq* and ending at *maxLinearFreq*. All of the other filters are placed uniformly on a mel-scale.

The `?MelFilters_t` data structure contains precomputed weights and data corresponding to the filter/FFT-coefficient combinations, thus reducing the computational load of the function `RL?CepstralMFCC()`.

The function returns a pointer to the `?MelFilters_t structure.`

FreeMFCCFilters

Destroys and reclaims the storage associated with the filters data structure.

```
void RLwFreeMFCCFilters(wMelFilters_t *filters);
    /* 16-bit integer vectors */

void RLsFreeMFCCFilters(sMelFilters_t *filters);
    /* single precision; real vectors */
```

filters A pointer to the data structure `?MelFilters_t`.

Discussion

The function `RL?FreeMFCCFilters()` is called to destroy and reclaim the storage space associated with the data structure `?MelFilters_t`.

Example 4-3 shows the code for extracting the MFCC coefficients from a single input signal frame.

Example 4-3 Extraction of MFCC from a single input signal frame

```
/* Compute the MFCC from a 128-point signal using 20 bandpass
 * filters in the frequency range 0 to 8Khz. Ten filters are
 * placed linearly up to a frequency of 1Khz and the remaining
 * filters are placed on a mel-scale. The sampling frequency of
 * the input signal is 16Khz.
 */

float        x[128];           // imput signal
float        x[10];            // the MFCC coefficients
sMelFilters_t *filters;        // the filters structure


/* Insert code here to put the 128 samples in x     */


/* first initialize the filters */
filters = RLsMFCCInit(7, 20, 0, 8000, 16000, TRUE, 10, 1000);

/* Now extract 10 MFCC coefficients using the filter structure.
RLsCepstralMFCC(TRUE, x, filters, ceps, 10);

/* Delete the filters structure to reclaim memory */
RLsFreeMFCCFilters(filters);
```

Recognition Basics 5

The functions described in this chapter compute similarity measures and probability density functions. Sometimes these basic functions comprise an entire recognition system, but more often they are components of a larger recognition system. The similarity measures include dot products and Euclidean distances. Gaussian mixtures and multi-layer perceptrons are non-linear functions of these similarity measures which are used to estimate probability density functions (pdfs). One particular type of probability density which is used commonly with hidden Markov models in speech recognition is referred to as an observation likelihood.

Similarity Measures

This section describes the functions in the Intel Recognition Primitives Library which compute similarity measures (also known as distance metrics).

DotP

Computes the dot product (inner product) of two vectors.

```
int RLwDotP(short int *vect1, short int *vect2, int n,
int doScaleOutput, int *scaleFactor);
    /* 16-bit integer vectors */

double RLsDotP(float *vect1, float *vect2, int n);
    /* single precision; real vectors */
```

vect1, *vect2*	The vectors for which the dot product is computed.
n	The length of the two vectors *vect1[n]* and *vect2[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?DotP() returns the dot product of the two vectors *vect1[n]* and *vect2[n]*. The dot product is defined as:

$$DotP = \Sigma_i \ (vect1_i * vect2_i)$$

where the summation is carried out over each component of the vectors.

L1Norm

Computes the L1Norm (city-block or Manhattan Distance) between two vectors.

```
int RLwL1Norm(short int *vect1, short int *vect2, int n,
int doScaleOutput, int *scaleFactor);
    /* 16-bit integer vectors */

double RLsL1Norm(float *vect1, float *vect2, int n);
    /* single precision; real vectors */
```

vect1, vect2	The vectors for which the L1Norm is computed.
n	The length of the two vectors *vect1[n]* and *vect2[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?L1Norm() returns the L1Norm (city-block distance) between the two vectors *vect1[n]* and *vect2[n]*. The L1Norm is defined as:

$$L1Norm = \Sigma_i \, abs(vect1_i - vect2_i)$$

where the summation is carried out over each component of the vectors.

L2Norm

Computes the L2Norm (Euclidean distance) between two vectors.

```
int RLwL2Norm(short int *vect1, short int *vect2, int n,
Bool doSquareRoot, int doScaleOutput, int *scaleFactor);
    /* 16-bit integer vectors */

double RLsL2Norm(float *vect1, float *vect2, int n, Bool
doSquareRoot);
    /* single precision real vectors */
```

vect1, vect2	The two vectors for which the L2Norm is computed.
n	The length of the vectors *vect1[n]* and *vect2[n]*.
doSquareRoot	A boolean. If true, the square root of the sum is of squares is calculated and returned.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?L2Norm() returns the L2Norm (Euclidean distance) between the two vectors *vect1[n]* and *vect2[n]*. The L2Norm is defined as:

$$L2Norm = \sqrt{\Sigma_i (vect1_i - vect2_i)^2}$$

where the summation is carried out over each component of the vectors.

Mahalanobis

Computes the Mahalanobis distance (covariance weighted distance) between two vectors.

```
int RLwMahalanobis(short int *vect1, short int *vect2,
short int **inverseCovarianceMatrix, int n, Bool
doSquareRoot, int doScaleOutput, int *scaleFactor);
    /* 16-bit integer vectors; full covariance matrix */

int RLwMahalanobisD(short int *vect1, short int *vect2,
short int *reciprocalVariance, int n, Bool doSquareRoot,
int doScaleOutput, int *scaleFactor);
    /* 16-bit integer vectors; diagonal covariance matrix
       (uses a vector) */

double RLsMahalanobis(float *vect1, float *vect2, float
**inverseCovarianceMatrix, int n, Bool doSquareRoot,);
    /* Single precision real vectors, full covariance
       matrix */

double RLsMahalanobisD(float *vect1, float *vect2, float
*reciprocalVariance, int n, Bool doSquareRoot,);
    /* Single precision real vectors, diagonal covariance
       matrix (uses a vector) */
```

vect1, *vect2*	The two vectors for which the Mahalanobis distance is computed.
inverseCovarianceMatrix	The inverted covariance matrix for the domain of the vectors *vect1[n]* and *vect2[n]*.
reciprocalVariance	A vector representing the reciprocals of the leading diagonal of the covariance matrix of the domain of the vectors *vect1[n]* and *vect2[n]*.

n	The length of the two vectors *vect1[n]* and *vect2[n]*. The matrix *Cov[n,n]* is a square matrix of size *n * n* and *dCov[n]* is a vector of length *n*.
doSquareRoot	Tell the function to take square root of the sum squares if true.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?Mahalanobis?() returns the Mahalanobis (covariance weighted distance) between the two vectors *vect1[n]* and *vect2[n]*.

The Mahalanobis distance is defined as:

$$\text{MahalanobisDist} = \sqrt{\Sigma_i \Sigma_j (\text{vect1}_i - \text{vect2}_i) * \text{inverseCovarianceMatrix}_{ij} * (\text{vect1}_j - \text{vect2}_j)}$$

where the summation is carried out over each component of the vectors.

When a diagonal covariance matrix is used, the Mahalanobis distance can be defined as:

$$\text{MahalanobisDist} = \sqrt{\Sigma_i (\text{vect1}_i - \text{vect2}_i)^2 * \text{reciprocalVariance}_i}$$

where the summation is carried out over each component of the vectors. The vector *reciprocalVariance[n]* now represents the reciprocals of the leading diagonal of the covariance matrix.

Observation Likelihood Estimates

This section describes functions that compute observation likelihood estimates. Multi-layer perceptron (MLP) evaluation and Gaussian mixtures are supported.

Gaussian Mixtures

Gaussian mixtures are implemented in the framework of a server model. The application initially sets up a Gaussian mixture server by calling the function `RL?InitGaussMixServer()`. The application should then pass all of the relevant information for setting up the mixtures to `RL?InitGaussMixServer()`. Subsequently, for each input vector, the mixture is evaluated by calling the function `RL?EvalGaussMix()`. This scheme eliminates the overhead of passing all the required arguments whenever a mixture needs to be evaluated. The following sections describe the mathematics involved in the computation of the Gaussian mixtures.

InitGaussMixServer

Initializes a Gaussian mixture server.

```
wGaussMixServer_t RLwInitGaussMixServer(short int
*weightVect, short int **meanVect, short int
***inverseCov, short int **inverseCovD, int n, int
nGauss, BOOL useExpTable, int distScale);
   /* 16-bit integer vectors */

sGaussMixServer_t RLsInitGaussMixServer(float
*weightVect, float **meanVect, float ***inverseCov, float
**inverseCovD, int n, int nGauss, BOOL useExpTable);
   /* Single precision; floating point vectors */
```

weightVect	The vector (of length *nGauss*) defining the weight applied to each Gaussian.
meanVect	A table containing *nGauss* rows where each row is the mean vector for the corresponding Gaussian. Implemented as a vector (length *nGauss*) of pointers, one for each mean vector. Each component of the mean vector contains the mean for that element position of the vector.
inverseCov	A vector of tables, one for each inverse (full) covariance matrix. The vector of tables is implemented as a vector (of length *nGauss*) of pointers, one for each inverse covariance matrix corresponding to each Gaussian. Each inverse covariance matrix is implemented as a table. Either *inverseCov* or *inverseCovD* should be NULL.
inverseCovD	A table containing *nGauss* rows where each row is the leading diagonal of the inverse covariance matrix for the corresponding Gaussian. Implemented as a vector (length *nGauss*) of

pointers, one for each vector. Either *inverseCov* or *inverseCovD* should be NULL.

n — The length of the mean vector. Each matrix in *inverseCov* is a square matrix of size *n* * *n* and each vector in *inverseCovD* is of length *n*.

nGauss — The number of Gaussians in the mixture.

useExpTable — A boolean. If true, lookup tables are used to approximate the exponential. If false, the exponential is computed accurately using the math library. The approximate exponential is computed as

```
exp(x) = exp(i) * exp(f)
```

where i is the integral part of x and f is the fractional part of x rounded to the third decimal place. Lookup tables are used for exp(i) and exp(f).

While using the lookup table option improves overall performance, it should be used with caution because of the loss of precision (resulting from the rounding of the fractional part to the third decimal place).

distScale — Scaling factor (expressed as a power of 2) that is used to normalize the covariance weighted distance for each Gaussian. The distance is multiplied by $2^{distScale}$ before being exponentiated. This is done for the integer version of the function because the mean and covariances might have been scaled up when represented as integers. A negative value should be used for normalization. However, a positive value will also work and will result in scaling up the computed distance. Use a value of 0 if no scaling is to be performed.

Discussion

The function RL?InitGaussMixServer?() creates a new Gaussian mixture server and returns a pointer to a structure ?GaussMixServer representing the server. The server architecture eliminates the need for passing a large number of arguments when evaluating a Gaussian mixture. Only the input vector and the server structure (obtained by calling this function) need be passed when evaluating a Gaussian mixture with the function RL?EvalGaussMix(). All the arguments that were used to initialize the mixture can be destroyed to reclaim space (if needed) because the mixture parameters are stored in an internal format.

The Gaussian mixture computed by this library is a weighted sum of Gaussian distributions $f(x)$, which can be written as:

$$f(x) = \Sigma_i W_i * \exp((x - \mu_i) * E_i * (x - \mu_i))$$

where w_i, μ_i, and E_i, are the weighting coefficient, mean vector, and inverse covariance matrix respectively of the ith multi-dimensional Gaussian distribution. The argument x is the multidimensional input vector.

The Gaussian mixture can be evaluated using the function RL?EvalGaussMix().

EvalGaussMix

Evaluates a Gaussian mixture.

```
int RLwEvalGaussMix(wGaussMixServer_t *server, short int
*vect, int n, int doScaleOutput, int *scaleFactor);
   /* 16-bit integer vector */

double RLsEvalGaussMix(sGaussMixServer_t *server, float
*vect, int n);
   /* single precision; real vector */
```

server	The pointer to the structure ?GaussMixServer_t representing the Gaussian mixture server. The pointer is obtained by an initial call to RL?InitGaussMixServer.
vect	The input vector for which the Gaussian mixture is evaluated.
n	The length of the input vector *vect[n]*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1. Used only by the RLwEvalGaussMix() function.

Discussion

The function RL?EvalGaussMix() evaluates the Gaussian mixture identified by *server* for the input vector *vect[n]*. The evaluated probability is then returned.

FreeGaussMixServer

Destroys and releases the storage space for one or more Gaussian mixture servers.

```
void RLwFreeGaussMixServer(wGaussMixServer_t server);
   /* 16-bit integer vector */

void RLsFreeGaussMixServer(sGaussMixServer_t server);
   /* single precision; real vector */
```

server	The pointer to the structure ?GaussMixServer_t representing the Gaussian mixture server. The pointer is obtained by an initial call to RL?InitGaussMixServer.

Discussion

The function RL?FreeGaussMixServer() destroys and releases the storage space for the Gaussian mixture server pointed to by *server*.

Example 5-1 shows the code to set up and use a Gaussian Mixture server.

Example 5-1 Setting Up and Using Gaussian Mixtures

```
/* This code sets up a Gaussian mixture with 5 mixture
 * components for inputs with 128 elements.
 */

float        x[128];           // input vector
float        weights[5];       // mixture weights
float        mean[5][128];     // mean vectors for each Gaussian
float        diag[5][128];     // diagonals of inverse covariance
                               // matrices for each Gaussian
sGaussMixServer_t *server;     // Gaussian mixture server struct
double prob;     // mixture probability for input x

float        *t_mean[5];       // vector of pointers to mean rows
float        *t_diag[5];       // vector of pointers to diag rows
int          i;

/* Insert code here to initialize x, weights, mean and diag */
/* first initialize the Gaussian mixture server     */
for (i=0;i<5;i++) {
      t_mean[i] = &mean[i][0];
      t_diag[i] = &diag[i][0];
}
server = RLsInitGaussMixServer(weights, t_mean, NULL, t_diag,
               128, 5, TRUE);

/* Now compute the mixture probability for the input x     */
prob = RLsEvalGaussMix(server, x, 128);

/* Delete the server to reclaim memory*/
RLsFreeGaussMixServer(server);
```

Multi-Layer Perceptron

The function described in this section performs feed forward processing for a multi-layer perceptron (MLP) neural network, sometimes referred to as a back-propagation network. The MLP is used extensively in OCR applications and is beginning to see application in speech recognition applications. Facilities for learning are not provided as part of the Recognition Primitives Library because learning is typically a one-time event performed during development of the weights for an MLP.

MLPerceptron

Performs multi-layer feed forward neural network processing on an input vector.

```
void RLwMLPerceptron(int numberOfInputs, int
inputsExponent, short int* input, int numberOfLayers,
int* layerNeuronCounts, int weightsExponent, short int*
weights, short int* output, int doScaleOutput, int*
scaleFactor);
    /* 16-bit integer vector */

void RLsMLPerceptron(int numberOfInputs, float* input,
int numberOfLayers, int* layerNeuronCounts, float*
weights, float* output);
    /* single precision; real vector */
```

numberOfInputs — Number of elements in the input vector.

inputsExponent — A single `int` value which is the exponent for all of the inputs. This argument is used only by the `RLwMLPerceptron()` function. The `RLwMLPerceptron()` function treats each input as a fixed point number where the *short int* data passed to it corresponds to the mantissas of the input values. The *inputsExponent*

	parameter corresponds to the common exponent of the input values. For example, if the desired input values are 3.1415 and -26.454, to maintain the best overall precision *inputsExponent* should be set to -10 and the short integers 3217 (0C91H) and -27088 (9630H) stored in the memory pointed to by *input*.
input	Pointer to the input vector to be processed which can be made up of `short ints` or `floats` depending on which of the two functions is used.
numberOfLayers	The number of layers of neurons in the Multi-Layer Perceptron.
layerNeuronCounts	Pointer to an array of integers which contains the number of neurons in each layer of the network. The first value in the array corresponds to the number of neurons in the layer connected to the inputs. The last value corresponds to the number of outputs the network produces.
weightsExponent	A single `int` value which is the exponent for all of the weights. This argument is used only by the `RLwMLPerceptron()` function. The `RLwMLPerceptron()` function treats each weight as a fixed point number where the `short int` data passed to it corresponds to the mantissas of the weight values. The *weightsExponent* parameter corresponds to the common exponent of the weights. For example, if the desired weights are 3.1415 and -26.454, to maintain the best overall precision *weightsExponent* should be set to -10 and the short integers 3217 (0C91H) and -27088 (9630H) stored in the memory pointed to by *input*.

weights	Pointer to the weights to be used which can be made up of short ints or floats depending on which of the two functions is used. The weights are ordered in sequence from first input to last, from first neuron to last, and from first layer to last. Thus, the first *numberOfInputs* weights will correspond to the connections between the inputs and the first neuron in the first hidden layer. The next *numberOfInputs* weights will be the weights for the second neuron, and so on.
output	Pointer to where the output vector will be written. The output representation for the RLwMLPerceptron() function is a fixed-point representation. The outputs are short ints scaled as specified by the arguments *doScaleOutput* and *scaleFactor*.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1. Used only by the RLwMLPerceptron() function.

Discussion

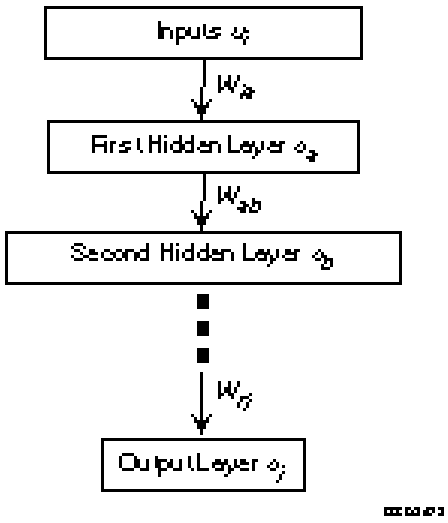
The RL?MLPerceptron() function performs multi-layer feed forward neural network processing on an input vector to produce a vector of neuron outputs given a set of weights. The processing is performed for each neuron in the specified network. The analytical expression for the computations done by the RL?MLPerceptron() function is as follows:

$$o_j = \frac{1}{1 + e^{-\left(\sum_i w_{ij} * u_i\right)}}$$

Multi-Layer Perceptron Architecture

Where u_i is a component of the input vector, W_{ij} is the ith weight for neuron j and o_j is the output of the jth neuron. The outputs of one layer of neurons are the inputs to the next layer of neurons. This is true for all layers except the output layer.

Figure 5-1 Multi-layer Perceptron Architecture



Vector Quantization and Kohonen Network

In the process of pattern recognition it is common to map a pattern to a lower dimension space to reduce the computation required for classification or to improve the generalization capability of a recognition system. Vector quantization and the Kohonen network are two techniques commonly used.

The same computations are done for both vector quantization and a Kohonen network. The only difference is in how the weights or codebook vectors are derived from a set of training patterns. Since the Recognition Primitives Library does not support training, only one set of functions is provided which can be used for both vector quantization and the Kohonen network. The Euclidean distances (L2Norms) are computed for an input vector and a set of weight vectors. By sorting the Euclidean distances, the closest vector to the input vector is identified. This is useful in vector quantization for finding the codebook vector which best matches the input vector. The closest matching vector to the input vector is also useful in training a Kohonen network. The functions optionally provide a ranked list of the indices of the closest matching vectors.

Facilities for learning are not provided as part of the Recognition Primitives Library because learning is typically a one-time event performed during development of the weights for a Kohonen network or during derivation of a set of codebook vectors for vector quantization.

VQKohonen

Calculates and ranks the Euclidean distances between an input vector and an array of weight vectors.

```
void RLwVQKohonen(short int* input, int numberOfInputs,
int numberOfOutputs, short int* weights, int
numberToRank, short int* rankList, int doSquareRoot,
 unsigned int* output, int doScaleOutput, int*
scaleFactor);
    /* 16-bit integer vector */

void RLsVQKohonen(float* input, int numberOfInputs, int
numberOfOutputs, float* weights, int numberToRank, short
int* rankList, int doSquareRoot, float* output);
    /* single precision; real vector */
```

input	Pointer to the input vector to be processed.
numberOfInputs	Number of elements in the input vector.
numberOfOutputs	Number of outputs or weight vectors for which distances are to be calculated.
weights	Pointer to the weight matrix to be used. The weights are ordered in sequence: first by input and second by output or weight vector.
numberToRank	The number of smallest distances to be ranked. Zero indicates that no sorting should be done. Note that the larger this number is, the more compute time is required. If *numberToRank* is set to zero, no memory needs to be allocated for *rankList*.

rankList	Pointer to a list of *integer* indices, indicating which *numberToRank* of the weight vectors are closest to the input vector. The index of the weight vector with the smallest distance appears first. For sorting, memory allocation for a full *numberOfOutputs* integers must be provided.
doSquareRoot	Tells the function to take the square root of the sum of the differences. The result is that Euclidean distances are calculated rather than the squares of the Euclidean distances. Substantial compute time can be saved by not calculating the square root without impacting the ability to classify. Setting *doSquareRoot*=0 will eliminate the taking of square roots. Setting *doSquareRoot*=1 corresponds to the calculation of standard Euclidean distances.
output	Pointer to where the output vector of Euclidean distances should be written. It is assumed that enough memory has been allocated by the user. If *doSquareRoot*=1 the vector pointed to will consist of Euclidean distances. The outputs are short `int`s scaled as specified by the arguments *doScaleOutput* and *scaleFactor*.
doScaleOutput	Tells the function how to scale the output to fit into the output data type. If the inputs utilize the full range of short `int`s and no square root is taken, the outputs will exceed the largest value that an `int` can store. This option flag can be set to zero to reduce compute time if the user knows that the output will always fit within an `int`. If turned off, an overflow will occur and no warning will be given.

scaleFactor	Pointer to an int scale factor which indicates the power of two the output should be multiplied by to recover the actual distances or distances squared. For example, if the integer value pointed to is 3 then the outputs should be multiplied by 2^3.

Discussion

The function RL?VQKohonen() computes a vector of distances, *o*, and stores it in the memory location pointed to by *output*. It also sorts the distances and returns a list of ranked indices in order from smallest to largest in the memory location pointed to by *rankList*.

The Euclidean distance computed by the RL?VQKohonen() function can be described as follows:

$$o_j = \sqrt{\sum_i \left(u_i - w_{ij}\right)^2}$$

where u_i is the *i*th component of the input vector, *input[n]*, and w_{ij} is the *i*th component of the *j*th weight vector. The array of weight vectors is pointed to by *weights*.

Image Processing 6

The algorithms described in this chapter are image processing routines used in optical character recognition (OCR). The functions supported are:

- Bit manipulation
- Image rotation
- Mirror image reflection
- Image copying
- Mask convolution

Pixel Arithmetic and Logical Operations

The functions described in the chapter "Vector Operations" can be used to manipulate pixel data in images. These functions implement logical and arithmetic operations on bit, nibble, and byte vectors.

Image Geometric Transformations

Functions described in this section perform transformations such as rotation on the image, mirror image reflection, and image copying.

RotateImage

Rotates an image by a specified angle.

```
void RLbRotateImage(const unsigned char **image, unsigned
char **outImage, int angle, int nInRows, int nInCols, int
nOutRows, int nOutCols)
    /* binary images */

void RLnRotateImage(const unsigned char **image, unsigned
char **outImage, int angle, int nInRows, int nInCols, int
nOutRows, int nOutCols)
    /* 4-bit nibble images */

void RLyRotateImage(const unsigned char **image, unsigned
char **outImage, int angle, int nInRows, int nInCols, int
nOutRows, int nOutCols)
    /* unsigned byte images */

void RLtRotateImage(const signed char **image, signed
char **outImage, int angle, int nInRows, int nInCols, int
nOutRows, int nOutCols)
    /* signed byte images */
```

image, *outImage*	The input image and output images respectively. An image is implemented as a pointer to a vector of pointers to vectors (one for each row of pixels).
angle	The angle by which the image should be rotated. The angle should be -90, 90, or 180 degrees.
nInRows, *nInCols*	The number of rows and columns in the input image.
nOutRows, *nOutCols*	The number of rows and columns in the output image.

Discussion

The function `RL?RotateImage()` rotates the input image *`image`* by *`angle`* degrees using the center of the input image as a pivot. The resulting rotated image is stored in *`outImage`*. This function could also be used (by specifying *`angle`* = *90)* for converting a row-based image (that is, where the rows are stored as arrays) to a column-based image (that is, where the columns are stored as arrays). Horizontal pixel scans are faster on row-based images and vertical pixel scans are faster on column-based images.

Example 6-1 shows the code for rotating a binary image by 90 degrees.

Example 6-1 Using RLbRotateImage() to Rotate a Binary Image

```
/* Rotate a 64 by 128 binary image by 90 degrees */

unsigned char   x[64][16];      // input image uses 16 bytes
                                // per image row for 128 pixels
unsigned char   y[128][8];      // output image uses 8 bytes
                                // per image row for 64 pixels
unsigned char *tx[64];          // vector of pointers to x rows
unsigned char *ty[128];         // vector of pointers to y rows

/* Insert code here for creating the images   */


for (i = 0; i < 64; i++)
    tx[i] = &x[i][0];
for (i = 0; i < 128; i++)
    ty[i] = &y[i][0];

/* Rotate the 64 by 128 input binary image to produce the
 * 128 by 64 output binary image
 */

RLbRotateImage(tx, ty, 90, 64, 128, 128, 64);
```

MirrorImage

Mirror reflects an image relative to a horizontal or vertical line.

```
void RLbMirrorImage(const unsigned char **image, unsigned
char **outImage, int nInRows, int nInCols, int orient);
   /* binary images */

void RLnMirrorImage(const unsigned char **image, unsigned
char **outImage, int nInRows, int nInCols, int orient);
   /* 4-bit nibble images */

void RLyMirrorImage(const unsigned char **image, unsigned
char **outImage, int nInRows, int nInCols, int orient);
   /* unsigned byte images */

void RLtMirrorImage(const signed char **image, signed
char **outImage, int nInRows, int nInCols, int orient);
   /* signed byte images */
```

image	The data structures for the input image. An image is implemented as a pointer to a vector of pointers to vectors (one for each row of pixels).
outImage	The data structures for the output image. An image is implemented as a pointer to a vector of pointers to vectors (one for each row of pixels).
nInRows	The number of rows in the input image.
nInCols	The number of columns in the input image.
orient	Indicates whether to reflect an image vertical or horizontal. The following values are allowed:

RL_ORIENT_HOR	Reflect horizontal
RL_ORIENT_VER	Reflect vertical

Discussion

The function `RL?MirrorImage()` mirror reflects an image relative to a horizontal or vertical line.

CopyImage

Copies an image or a part of an image to another image.

```
void RLbCopyImage(const unsigned char **image, unsigned
char **outImage, rect *copyRgn, int orgRow, int orgCol);
   /* binary images */

void RLnCopyImage(const unsigned char **image, unsigned
char **outImage, rect *copyRgn, int orgRow, int orgCol);
   /* 4-bit nibble images */

void RLyCopyImage(const unsigned char **image, unsigned
char **outImage, rect *copyRgn, int orgRow, int orgCol);
   /* unsigned byte images */

void RLtCopyImage(const signed char **image, signed char
**outImage, rect *copyRgn, int orgRow, int orgCol);
   /* signed byte images */
```

image	The data structures for the input image. An image is implemented as a pointer to a vector of pointers to vectors (one for each row of pixels).
outImage	The data structures for the output image. An image is implemented as a pointer to a vector of pointers to vectors (one for each row of pixels).
copyRgn	Specifies the rectangular region to be copied.
orgRow, *orgCol*	Indicates the left upper vertex of the output image.

Discussion

The function `RL?CopyImage()` copies an image or a part of an image to another image.

Mask Convolution

This is one of the most important image processing primitives and a variety of image processing functions can be implemented with this capability (for example, edge detection, blurring, noise removal, feature detection, and so on). A square or rectangular mask can be used. For a mask of size `m` by `n` (assuming `m` and `n` are odd) with pixels c_{kl}, a new pixel output value o_{ij} (output image size is `x` by `y`) is computed for pixels i_{qr} in the input image (size `x` by `y`) as

$$o_{ij} = \sum_{\substack{k=0 \\ l=0}}^{\substack{k=m \\ l=n}} i_{qr}\, c_{kl}$$

where `i = 0,1,..,x` and `j = 0,1,..y` and `q = i + k-floor(m/2)` and `r = j + l - floor(n/2)`.

MaskConvolve

Convolves a mask with an image.

```
void RLMaskConvolve(const char **image, const char
**mask, char **outImage, int imageDataType, int
maskDatatype, int outDataType, int nImageRows, int
nImageCols, int nMaskRows, int nMaskCols, rect
*convolveRegion, int doScale, int *scaleFactor)
```

image, *mask*, *outImage*

The data structures for the input image, mask, and output images respectively. An image (or mask) is implemented as a pointer to a vector of pointers to vectors (one for each row of pixels). Although each vector is declared to be an array of bytes (`char`), the actual interpretation of the bytes is determined from the value of the argument *?DataType* described below.

imageDataType, *maskDataType*, *outDataType*

The data types of the input image, mask, and output images respectively. The following predefined constants are used to indicate the types which are allowed for the input and output images:

`BIT`	Binary image (that is, single bit pixels). 8 pixels per byte.
`UNIBBLE`	Unsigned 4-bit pixels. Two pixels per byte.
`UBYTE`	Unsigned 8-bit byte pixels.
`SBYTE`	Signed 8-bit byte pixels.

The following types are allowed for the mask:

`SBYTE`	Signed 8-bit byte pixels.
`SWORD`	Signed 16-bit word pixels.

nImageRows, *nImageCols*	The number of rows and columns in the images.
nMaskRows, *nMaskCols*	The number of rows and columns in the mask.

convovleRegion	Specifies the rectangular region over which the convolution is to be performed. The type declaration for *rect* is:
	`typedef struct _rect {`
	`  int leftTopRow; int leftTopCol;`
	`  int rightBotRow; rightBotCol;} rect;`

doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1. However, in this case, the option `RL_AUTO_SCALE` is not allowed.

Discussion

The function `RLMaskConvolve()` convolves the mask *mask* with the input image *image* over the region defined by *convolveRegion* and creates the output image *outImage* which contains the result of the convolution. All of the other arguments listed above describe the data structures and types of the input image, output image, and mask.

Example 6-2 shows how mask convolution can be used to implement a simple image blurring operation. Each pixel in the output is set to the average of its immediate eight neighbours.

Example 6-2 Blurring an Image Using Mask Convolution

```
/* Blur an image using a simple mask containing all ones except
 * the centre pixel.
 */

char   x[32][32];   // Input 32 by 32 8-bit signed pixel image
char   y[32][32];   // Output 32 by 32 8-bit signed pixel image
char   mask[3][3];  // 3 by 3 mask, 8-bit signed pixels
rect   region = {0, 0, 31, 31}; // convolve entire image
int    i, j;
int    *scaleFactor;

unsigned char *ti[32];  // vector of pointers to input image rows
unsigned char *to[32];  // vector of pointers to output image rows
unsigned char *tm[3];   // vector of pointers to mask rows

/* insert code here for creating the input image */


for (i = 0; i < 32; i++)
    ti[i] = &x[i][0];
for (i = 0; i < 32; i++)
    to[i] = &y[i][0];
for (i = 0; i < 3; i++)
    tm[i] = &mask[i][0];

/* Initialize the mask          */
for (i = 0; i < 3; i++)
   for (j = 0; j < 3; j++)
      mask[i][j] = 1;
mask[1][1] = 0; // set the centre pixel to 0

/* Perform the convolution. Use a scaleFactor of 3 so
 * that each output pixel is divided by 8. Effectively
 * each output pixel is computed as the sum of its eight
 * neighbours divided by 8.
 */

*scaleFactor = 3;
RLMaskConvolve(ti, tm, to, SBYTE, SBYTE, SBYTE, 32, 32,
            3, 3, region, RL_FIXED_SCALE, scaleFactor);
```

Dynamic Programming 7

This chapter describes two Dynamic Programming techniques: Dynamic Time Warping (DTW) and Hidden Markov Models (HMM). These are pattern matching techniques, which are used when the natural data has a dimension in which the data is distorted. For example, this dimension includes time for continuous speech and the left-to-right direction for cursive handwriting.

Dynamic Time Warping

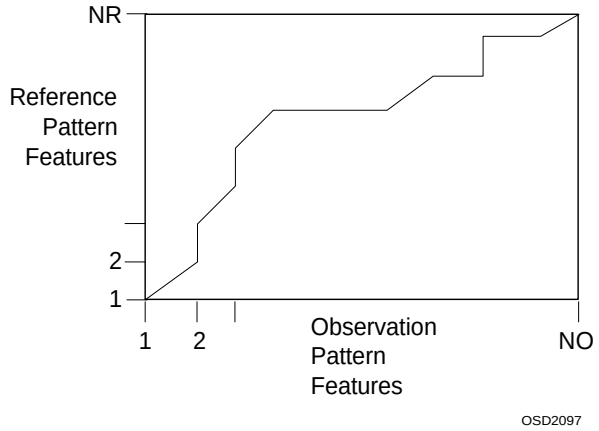
Dynamic Time Warping is a technique for elastically matching a sequence of observations to a reference pattern (template). The observation and reference features are laid out in sequence along the x and y axes respectively of a grid. For each point in the grid, the distance (according to some distance metric) between corresponding features in the observation and reference pattern features is computed. This distance is then added to the minimum distance found in one of a specified set of precursor grid points.

The optimum alignment between an observation sequence and a reference pattern is represented by a path through the m indices of the reference pattern to the n indices of the observation sequence, of the form

m = w(*n*), that minimizes the accumulated distance.

The diagram below shows one possible alignment path between the observation sequence and reference patterns.

Figure 7-1 An Alignment Path Between an Observation Sequence and Reference Patterns



The DTW can be fully specified by

- Endpoint constraints.
- Local constraints.
- The distance metric.

These are described below.

Endpoint Constraints

The simplest endpoint constraint is of the form

- **w**(1) = 1 and
- **w**(*NO*) = *NR*

(where *NO* and *NR* are the last points of the observation and reference patterns) that is, perfect alignment of the endpoints. This is called *constrained endpoints, 2-to-1 slope range (RL_CE21).*

A second variant called *unconstrained endpoints 2-to-1 slope range (RL_UE21)* slightly relaxes the endpoint constraints to the set

$1 <= \mathbf{w}(1) <= 1+\delta$

$NR - \delta <= \mathbf{w}(NO) <= NR$

Local Constraints

Local constraints describe the possible types of motion like directions, slopes, and so on. One constraint that is always assumed for speech is that the time index can only increase monotonically. Under this condition the minimum accumulated distance **D**(*n*, *m*) from the initial point *n* = 1, *m* = 1 to the grid point (*n*, *m*) can be computed recursively as

$$\mathbf{D}(n, m) = \mathbf{d}(O_n, R_m) + \mathbf{min}[\mathbf{D}(n-1, q)], \quad q <= m$$

where O_n and R_m are the *n*th observation and *m*th reference pattern features respectively and *q* is a set of *m* values such that a path exists between (*n*-1, *q*) and (*n*, *m*).

One type of local constraint is known as the Itakura local constraint. This constraint dictates that three or more consecutive points in the path cannot be flat (that is, horizontal). The Itakura local constraint can be expressed as:

$$\mathbf{w}(n) - \mathbf{w}(n-1) = 0, 1, 2 \quad \text{if } \mathbf{w}(n-1) \neq \mathbf{w}(n-2)$$

$$= 1, 2 \quad \text{if } \mathbf{w}(n-1) = \mathbf{w}(n-2) \text{ otherwise}$$

Effectively, due to this constraint, three or more consecutive points in the path cannot be flat (that is, horizontal). With this constraint, the accumulated distance can be written in a simpler form recursively as

$$\mathbf{D}(n,m) = \mathbf{d}(O, R) + \mathbf{min}[\mathbf{D}(n-1, m)\mathbf{g}(n-1, m), \mathbf{D}(n-1, m-1), \mathbf{D}(n-1, m-2)]$$

where

$$\mathbf{g}(n-1, m) = 1 \quad \text{if } \mathbf{w}(n-1) \neq \mathbf{w}(n-2)$$

$$= \alpha \text{ (infinity)} \quad \text{if } \mathbf{w}(n-1) = \mathbf{w}(n-2) \text{ otherwise}$$

The final desired solution (that is, **D**(*NO*, *NR*), the minimum accumulated distance over the entire path) is computed iteratively using the above equation.

Distance Metrics

The supported distance metrics are City-block and Euclidean.

EvalDTW

Computes the minimum distance between an observation sequence and a set of reference (or template) sequences (patterns) using the Dynamic Time Warping algorithm.

```
int RLwEvalDTW(int *idPattern, int nPattern, short int
**oSeq, int nSeq, int len, int oFactor, int flags, int
thrsh, int thrshFactor, int delta, int *dist, int
doScale, int scaleFactor);
    /* 16-bit integer vectors */

int RLsEvalDTW(int idPattern, int nPattern, float **oSeq,
int nSeq, int len, int flags, double thrsh, int delta,
double *dist);
    /* single precision; real vectors */
```

idPattern	The vector of pattern IDs of the reference patterns (templates) that the observation sequence will be matched to. Each pattern is created by a call to the function `RL?PatternIni()`.
nPattern	The number of reference patterns to evaluate. This is the length of the *idPattern* vector.
oSeq	A pointer to a vector of pointers (corresponding to the observation sequence) to vectors. Each vector corresponds to one point or feature in the sequence.
nSeq	The length of the observation sequence.
len	The length of each observation or reference feature vector.

oFactor The int scale factor. This is the power of two by which the *oSeq* should be multiplied to recover the actual values of sequence elements.

flags An ORed bit mask of options for the DTW. These options are predefined and can be logically ORed to turn on the appropriate options. The following is a list of available:

Endpoint Constraint Options

RL_CE21 Constrained endpoints 2-to-1 slope range.

RL_UE21 Unconstrained endpoints 2-to-1 slope range.

Local Constraint Options

RL_ITAKURA Itakura local constraint. This is the only local constraint currently supported.

Distance Metric Options

RL_L1NORM City-block distance metric.

RL_L2NORM Euclidean distance metric.

Threshold Option

RL_TRHESHOLD Option for abandoning the match when accumulated distance is more than the *thrsh* value.

thrsh The distance threshold value for abandoning matches. This value should be used only when the RL_THRESHOLD option is specified.

thrshFactor The power of two by which the *thrsh* should be multiplied to recover the actual *thrsh* value.

delta The delta value to be used only when the endpoint constraint is specified as RL_UE21.

dist	Pointer to where the accumulated distances between the observation sequence and the *nPattern* corresponding patterns should be saved. It is assumed that enough memory has been allocated by the user.
doScale	Indicates how the *dist* vector should be scaled. Refer to "Integer Scaling" in Chapter 1.
scaletFactor	The pointer to an `int` scale factor for the *dist* vector.

Discussion

The function `RL?EvalDTW()` evaluates the minimum distance (corresponding to the best elastic match) between the observation and reference patterns *oSeq* and *idPattern* respectively using the dynamic programming DTW algorithm. The index of the reference pattern in the *idPattern* vector with the minimum distance to the observation sequence is returned. The distances to all of the reference patterns are returned in the vector *dist*. The reference patterns can be created by calling the function `RL?PatternIni()`.

PatternIni

Creates and initializes a reference pattern for use in the DTW algorithm.

```
int RLwPatternIni(short int **pSeq, int nSeq, int len,
int pFactor);
/* 16-bit integer vectors */

int RLsPatternIni(float **pSeq, int nSeq, int len);
/* single precision; real vectors */
```

pSeq	A vector of pointers (corresponding to the pattern sequence) to vectors. Each vector corresponds to one point (or feature) in the sequence.
nSeq	The length of the pattern sequence.
len	The length of each feature vector.
pFactor	The `int` scale factor which is the power of two by which the *pSeq* should be multiplied to recover the actual values of sequence elements.

Discussion

The function `RL?PatternIni()` creates and initializes a reference pattern (that is, template) for use when calling the `RL?EvalDTW()` function. The ID of the newly created pattern is returned.

PatternFree

Destroys and frees the memory of reference patterns.

```
void RLwPatternFree(int idPattern);
/* 16-bit integer vectors */

void RLsPatternFree(int idPattern);
/* single precision; real vectors */

void RLwPatternFreeAll();
/* 16-bit integer vectors */

void RLsPatternFreeAll();
/* single precision; real vectors */
```

idPattern	The ID of the corresponding pattern to be destroyed.

Discussion

The function `RL?PatternFree()` destroys and frees the memory associated with a reference pattern. The function `RL?PatternFreeAll()` destroys all existing reference patterns.

Example 7-1 shows the code to evaluate the minimum distance between the observation sequence and reference patterns.

Example 7-1 Using DTW Evaluation

```
/* Evaluate the minimum distance between the observation sequence
 and reference patterns  */

int     nPttrn = 3;          /* number of patterns to evaluate */
int    *idPttrn;             /* id's patterns vector */

int     bestPttrn;           /* the best pattern index */
double *dist;               /* vector for accumulated distances
saving */

int    flags = RL_CE21|RL_ITAKURA|RL_L1NORM; /* flags for DTW
evaluation */

float  a[32][8], b[30][8], c[34][8], d[30][8]; /* patterns sequences
and observation sequence*/

float  *t_a[32], *t_b[30], *t_c[34], *t_d[30]; /* vectors of pointers
to sequences "points" */
int    i;

/* Insert code here to initialize the sequences     */

....

/* Allocate memory for pattern manipulation */
idPttrn = (int*)malloc(nPttrn*sizeof(int));
dist    = (double*)malloc(nPttrn*sizeof(double));

/* Iinitialize the vectors of pointers to sequences "points" */
for(i=0;i<32;i++) t_a[i] = &a[i][0];
for(i=0;i<30;i++) t_b[i] = &b[i][0];
for(i=0;i<34;i++) t_c[i] = &c[i][0];
for(i=0;i<30;i++) t_d[i] = &d[i][0];

/* Patterns initialization */
idPttrn[0] = RLsPatternIni(t_a, 32, 8);
idPttrn[1] = RLsPatternIni(t_b, 30, 8);
idPttrn[2] = RLsPatternIni(t_c, 34, 8);

/* DTW evaluation */
bestPttrn = RLsEvalDTW(idPttrn, nPttrn, t_d, 30, 8, flags,
                    (float)0.0, 0, dist);

/* Delete the patterns to reclaim memory */
RLsPatternFreeAll();
```

Hidden Markov Models

HMMs are used to model data that have statistical properties that vary with time. Speech is the most common data type modeled by HMMs. In speech recognition, features such as Cepstral coefficients are extracted from frames of speech to reduce the dimension of the speech signal. A sequence of feature vectors, also referred to as observations $O_t(t=1,2\ldots T)$, are and then classified into linguistic components such as words, subwords, or phonemes. A sequence of such components is referred to as an utterance. An HMM can be characterized by its probability of producing such a given sequence of observations. The functions in this section provide for the rapid evaluation of these HMM probabilities given a set of previously derived HMMs and a sequence of observations.

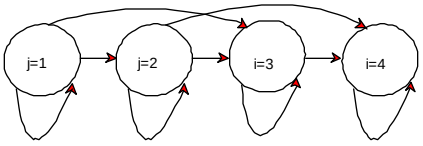
A Markov model consists of *N* states numbered *j* = 1,2,... N. Transitions can occur from one state to another.

Three types of HMMs can be defined based on the type of transitions that are allowed between states:

- Ergodic or "fully connected" HMMs. Any state of the HMM can be reached (in a single step) from any other state.
- Left-right or "Bakis" model. As time increases the state index increases (or stays the same); that is, the state transitions proceed only from left to right.
- Constrained jump model. Transitions can occur from state *j* to state *j*, *j*+1 or *j*+2. See Figure 7-1.

This library supports all three of the models in the preceding list.

Figure 7-1. Constrained Jump HMM with Four States



An HMM is fully specified by the parameter set **{**A, B, π**}** where:

- A is the matrix $\{a_{ij}\}$ where a_{ij} is the probability of transition from state i to state j.
- B is the vector $\{b_j(o_t)\}$ where $b_j(o_t)$ is the observation likelihood of observation o_t when in state j.
- π is the vector $\{\pi_j\}$ where π_j is the probability of starting in state j.

A complete description of the notation used here and how it relates to the parameters used in the library functions is shown in Table 7-1.

HMMs can be further classified into three types based on how the observation likelihoods $b_j(o_t)$ are modeled. These types are:

- Discrete HMMs: In discrete HMMs, the observation vector o_t is quantized. See the VQKohonen function in this library. The index, k, of a reference vector that most closely matches the observation vector is determined. The index is then used to look up a corresponding value b_{jk}, which represents the observation likelihood $b_j(o_t)$.[3] See Figure 7.2.
- Semi-Continuous HMMs: Here the $b_j(o_t)$ values are obtained as the output of a continuous conditional Gaussian mixture probability density function given an observation vector, (o_t), as input. The Gaussian components are not associated with the state or HMM and can be computed once for the given continuous acoustic observation vector. The Gaussian components pkd are common to all the HMMs in the system. While the mixture weights, W_{kj} are specific to each state of each HMM[4]. See Figure 7-3.
- Continuous HMMs: Each $b_j(o_t)$ here requires the computation of a continuous Gaussian mixture probability density function This type of HMM is therefore the most computation intensive because there is one Gaussian mixture per state per HMM. See Figure 7-4.

All three types of HMMs are supported by this library.

The key computational tasks that make up a recognition system are the following::

- Feature Extraction
- Training
- Recognition

These computational components are described in the following section.

Feature Extraction

Feature extraction provides a vector of smaller dimension than the original data, which contains only the essential information for the task at hand. The Cepstral MFCC function in the library extracts the most commonly used speech features from a vector of speech audio samples. Feature extraction is performed periodically on a time window of audio samples. A vector of speech acoustic features is referred to as an observation vector, and is typically extracted once every 10 milliseconds from a 20 millisecond window of the speech audio signal, which is sampled at 8 to 16 kHz. Feature extraction from the speech signal corresponding to an utterance produces a sequence of continuous observation vectors o_t (t = 1,2...,T). For the discrete HMM (described earlier), these continuous observation vectors are vector quantized using a codebook containing a set of reference vectors R_k, (k = 1,2...,k). The codebook index, k, is used to look up likelihoods, b_{jk}, for each state of the HMM.

Training

The training task consists of estimating the probabilities {**A**, **B**, π} (described in the preceding section) given a training set of known utterances. The training of HMMs is a complex issue with many tradeoffs, and as a result cannot be easily supported by a general purpose library. The user must do the training prior to using the library.

Recognition

The recognition task uses the observation likelihoods $b_j(o_t)$ and the transition probabilities a_{ij} to calculate the probability that each HMM produced the utterance. The Forward algorithm [1] estimates the probability for all possible paths through an HMM while the Viterbi algorithm, which is faster, estimates only the probability of the best path through an HMM. The word recognized is the word associated with the

HMM that produced the highest probability. Both algorithms use dynamic programming but perform different calculations at each point within the lattice.

The Forward algorithm uses both “multiply” and “add” operations while the Viterbi algorithm uses only the “add” operation. The Viterbi algorithm is the preferred method to estimate HMM probabilities because it avoids costly multiply operations. In this library, the Viterbi algorithm is supported through the function `RL?EvalHMMViterbi()`

HMM Implementation and Class Concept

The HMM implementation follows a server model, where a server is initialized for each HMM using an initialization function, `RL?InitHMMServer()`. When many HMM-servers are involved, which is typical for recognition tasks, all HMM-servers can be separated into classes. However, these need not be disjoint classes; that is, more than one class can contain the same HMM-server. This feature is useful when linguistic processing is used to determine the next set of HMMs to evaluate; the next set is represented by a class, which contains HMM-servers of the same type (discrete, semi-continuous, or continuous). The functions `RLCreateHMMClass()`, `RLFreeHMMClass()`, `RLAddHMMToClass()`, `RLRemoveHMMFromClass()`, and `RLHMMFreeAll()` support the creation, deletion, and management of classes.

You can pass a single HMM or an entire class of HMMs to `RL?EvalHMMViterbi()` for evaluation.

Three different schemes for recognition using the Recognition Primitives Library functions are presented in Figures 7-2, 7-3, and 7-4. Table 7-1 describes the notational conventions used in the figures. Figures 7-5 and 7-6 illustrate the data structures used in the semi-continuous and continuous HMMs.

For the discrete HMM case, each codeword of the codebook is an observation vector (a center of corresponding acoustic space region).

For the semi-continuous HMM case, each codeword is used as the mean vector of a simple Gaussian probability density function. Gaussian mixture servers for these pdfs must be initialized before the initialization of any semi-continuous HMM server and must not be freed while the HMM servers are in use.

For the continuous HMM case, a codebook is not used. A Gaussian mixture probability density function is defined for each state of every HMM. Gaussian mixture servers for all states of a continuous HMM server must be initialized before the initialization of the HMM server and must not be freed while it is in use.

Table 7-1. Notation Conventions for Figures 7-2, 7-3, and 7-4

Description	Notation	Library Parameter	Data Structure
Number of HMM States	N	nStates	scalar
Number of Observation symbols	K	nSymbols	scalar
Number of Dimensions in Observation and Reference Vectors	D	nQuant	scalar
Initial Probabilities	π_j	iprob	$\{ \pi_j , j = 1 \ldots N \}$ pointer to vector
Transition Probability from state i to state j	a_{ij}	tprob	$\{ a_{ij} , \begin{matrix} i = 1 \ldots N, \\ j = 1 \ldots N \end{matrix} \}$ pointer to two dimensional array
Observation Sequence	O_t	nqvect	$\{ O_t , t = 1 \ldots T \}$
Length of Observation Sequence	T	nObserv	scalar
Reference Vectors used in VQ	R_k	lookUp	$\{ R_k , k = 1 \ldots K \}$

Probability for symbol k in state j (discrete HMM)	b_{jk}	symbprob	$\{b_{jk}, \begin{matrix} j = 1...N, \\ k = 1...K \end{matrix}\}$ pointer to two dimensional lookup table
Vector of Codebook Indices (discrete HMM)	V	vect	$\{v_t, t = 1...T\}$
CCGPDFs for symbol k (semi continuous HMM)	$p_k(O_t)$	ccgPdfs	$\{p_k, k = 1...K\}$ pointer to a set of Gaussian Mixture Servers
Mean Vector for the i-th Gaussian	μ_i	meanVect	see meanVect in data structure diagrams
Inverse Covariance Matrix for the i-th Gaussian	E_i	cov	See cov in data structure diagrams
Number of Gaussians in the mixture	U	nGauss	scalar (could vary for each state j)
Observation likelihood for O_t in state j	$b_j(O_t)$	(internal)	scalar
Gaussian Mixture Weight for symbol k in state j (semi continuous HMM)	W_{jk}	weightVect	$\{W_{jk}, \begin{matrix} j = 1...N \\ k = 1...K \end{matrix}\}$ see weightVect in data structure diagrams
Gaussian Mixture Weight for u-th Gaussian in state j (continuous HMM)	W_{ju}	weightVect	$\{W_{ju}, \begin{matrix} j = 1...N \\ u = 1...U_j \end{matrix}\}$ see weightVect in data structure diagrams
Viterbi Probability	P^v	pvit	scalar

Figure 7-2. Recognition Scheme Using Discrete Servers

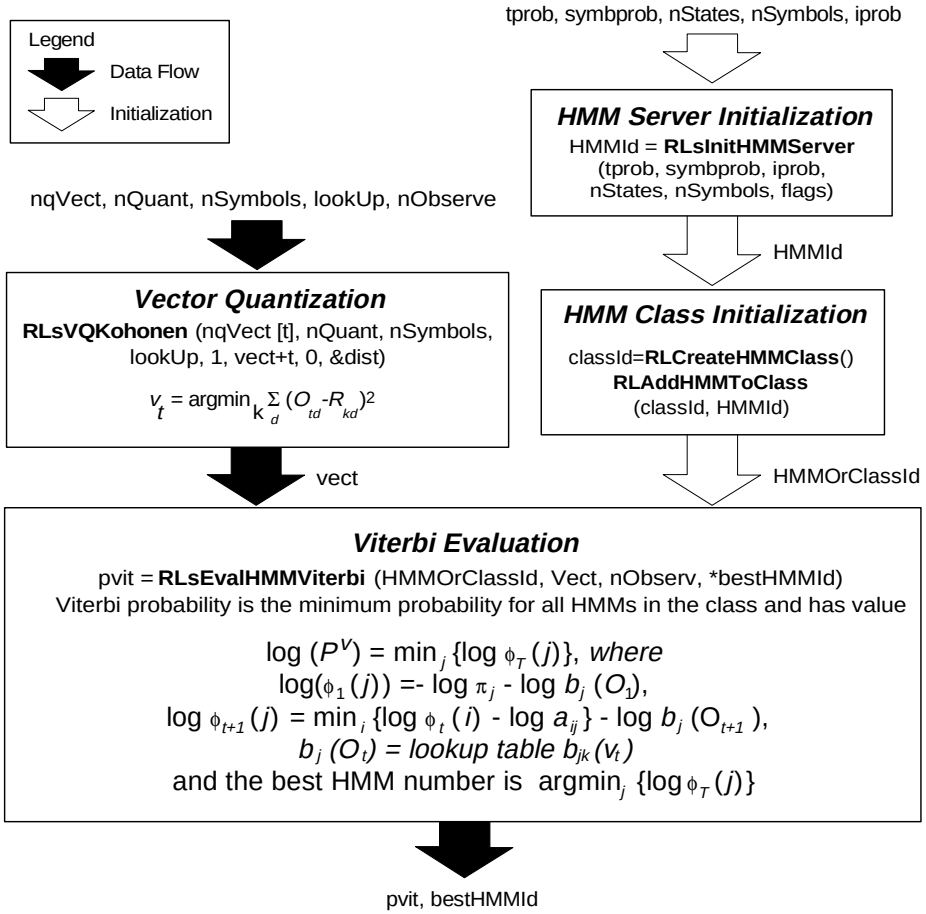


Figure 7-3. Recognition Scheme Using Semi-Continuous HMM Servers

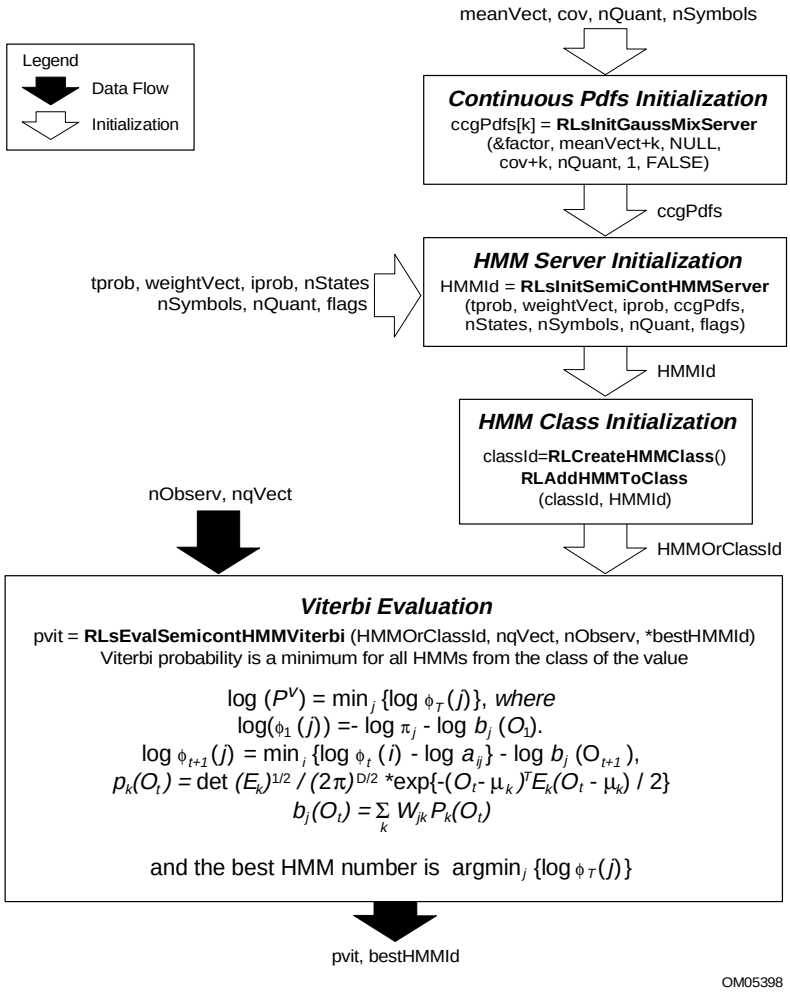


Figure 7-4. Recognition Scheme Using Continuous HMM Servers

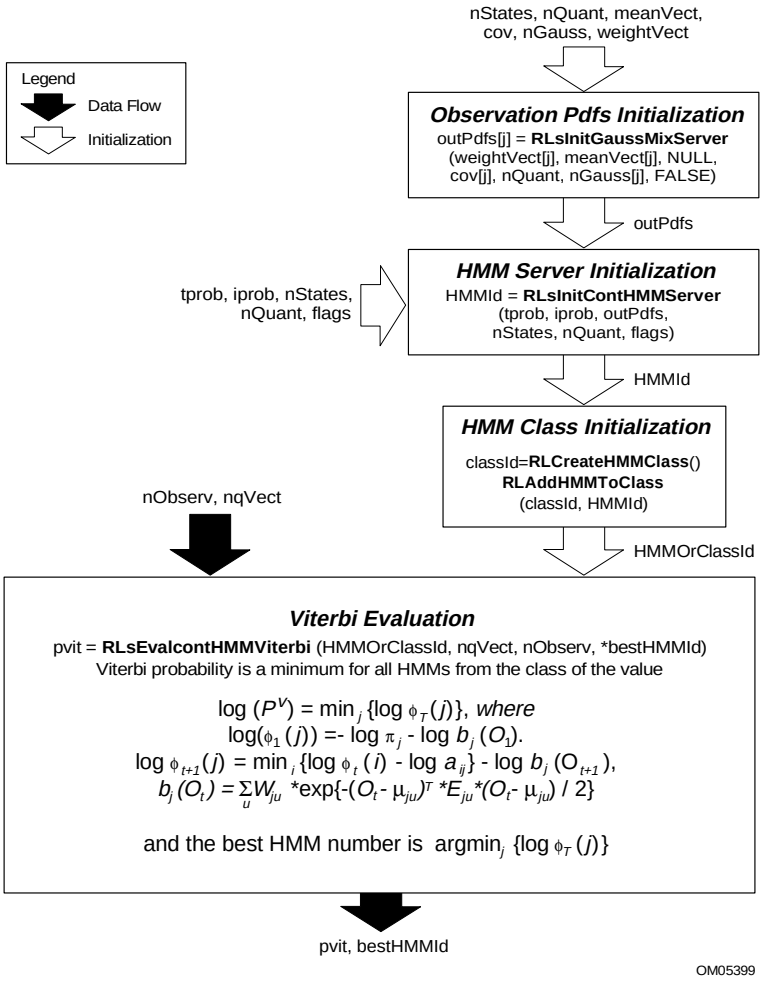


Figure 7-5. Data Structures Used in Semi-Continuous HMMs

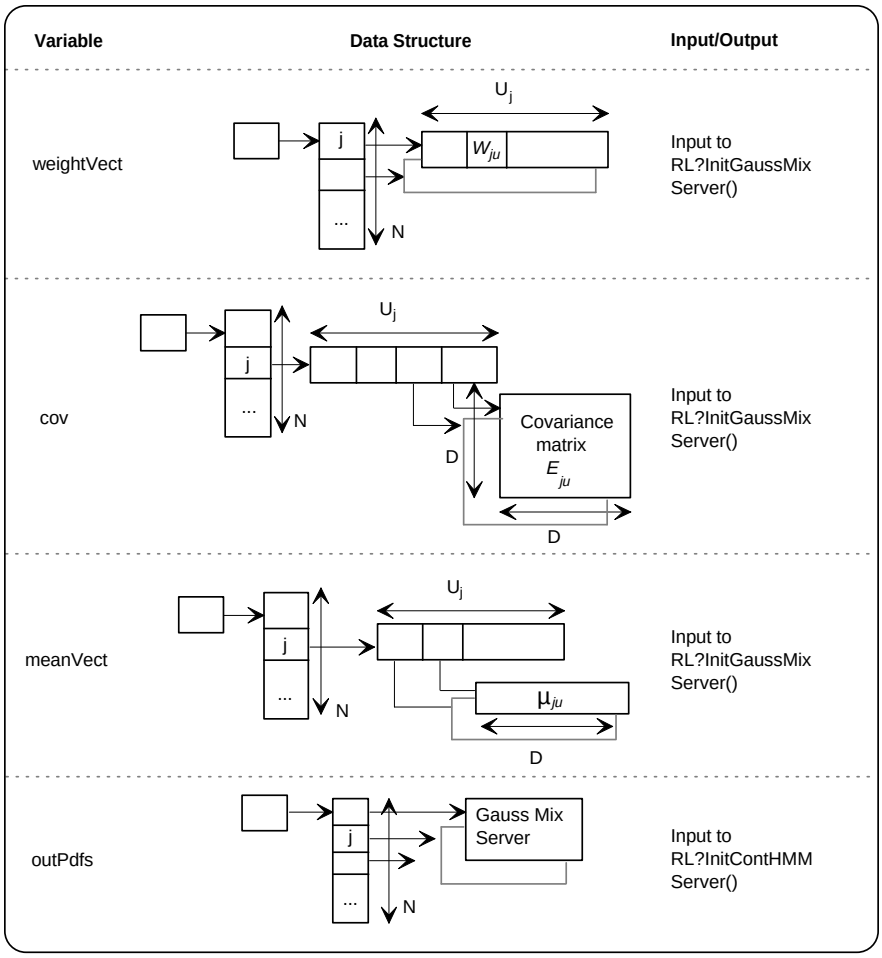
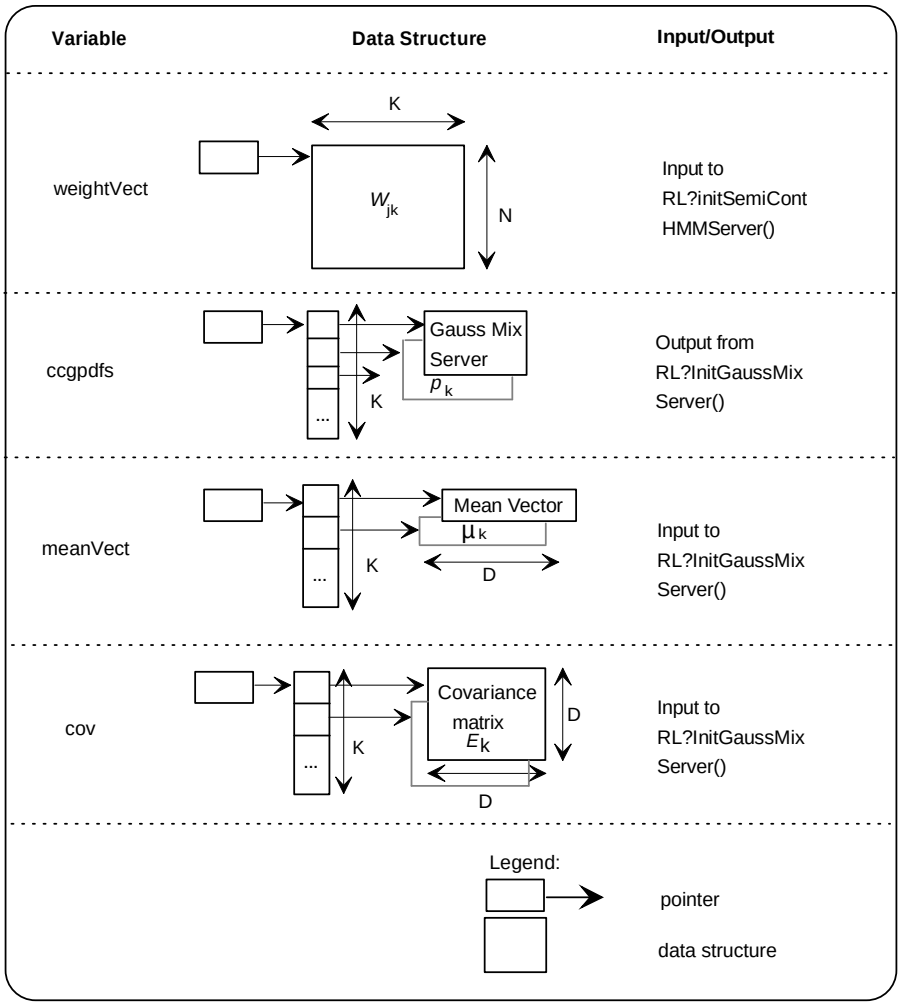


Figure 7-6. Data Structures Used in Continuous HMMs



om05394

HMM References

[1] Rabiner, L. R., “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” Proceedings of the IEEE 1989.

[2] A. Waibel and K. F. Lee, “Readings in Speech Recognition,” Morgan Kaufmann Publishers, Inc. San Mateo, CA, copyright 1990.

[3] J. R. Deller, J. G. Proakis, and J. H. L. Hansen, “Discrete-Time Processing of Speech Signals,” Macmillan Publishing Company, copyright 1993

[4] Huang, X.D. and Jack, M.A., “Semi-Continuous Hidden Markov Models for Speech Recognition,” Computer Speech and Language 3(3):239-252, July 1989

InitHMMServer

Creates and initializes an HMM server.

```
int RLwInitHMMServer(short **tprob, short **symbProb,
short *iprob, int nStates, int nSymbols, int flags);
   /* discrete HMM for 16-bit integer vectors */

int RLsInitHMMServer(float **tprob, float **symbProb,
float *iprob, int nStates, int nSymbols,  int flags);
   /* discrete HMM for single precision; real vectors */

int RLwInitSemicontHMMServer(short **tprob, short
**WeightVect, short *iprob, wGaussMixServer_t **ccgPdfs,
int nStates, int nSymbols, int nQuant, int flags, int
scaleFactor);
   /* semi-continuous HMM for 16-bit integer vectors */

int RLsInitSemicontHMMServer(float **tprob, float
**wieghtVect, float *iprob, sGaussMixServer_t **ccgPdfs ,
int nStates, int nSymbols, int nQuant, int flags);
   /* semi-continuous HMM for single precision; real
vectors */

int RLwInitContHMMServer(short **tprob, short *iprob,
wGaussMixServer_t **outPdfs, int nStates, int nQuant, int
flags, int scaleFactor);
   /* continuous HMM for 16-bit integer vectors */

int RLsInitContHMMServer(float **tprob, float *iprob,
sGaussMixServer_t **outPdfs, int nStates, int nQuant, int
flags);
   /* continuous HMM for single precision; real vectors*/
```

tprob

Pointer to a vector of pointers to the rows of the transition probability matrix. Rows and columns both correspond to states. Each value within the matrix represents the probability of transition from the state associated with its row to the state

	associated with its column. The structure of the HMM (constrained jump, Bakis or ergodic) is defined by the values of the transition probabilities.
symbProb	*symbProb* is a pointer to a vector of pointers to the rows of the observation symbol probability matrix used in discrete HMMs. Row vectors correspond to states. The number of columns is equal to the number of symbols. Each value in the matrix corresponds to the probability of emitting the associated symbol while in the state associated with its row.
weightVect	*weightVect* is a pointer to a vector of pointers to the rows of a weight matrix. The number of columns is equal to the number of nSymbols. The values are not logarithmic even if the RL_HMM_LOGARITHMIC flag is set. For the short integer form, RLwInitSemiContHMMServer, the *weightVect* matrix elements are assumed to be in fixed-point notation where the short ineteger value passed is to be multiplied by 2^{-14} before being used. The observation likelihood for state *j* is the dot product of the *j*th row of *weightVect* and the vector of continuous conditional Gaussian pdfs. This data structure is used only by semi-continuous HMMs.
iprob	Pointer to the initial probability vector of length *nStates*.
ccgPdfs	Pointer to a Gaussian mixture server pointer vector of length *nSymbols*. This vector of pointers specifies the Gaussian Mixture servers that provide the continuous conditional Gaussian pdfs to the semi-continuous HMM evaluation function.

scaleFactor	*scaleFactor* is not currently used.
outPdfs	Pointer to a Gaussian mixture server pointer vector of length *nStates*. This vector of pointers specifies the Gaussian mixture servers that provide observation likelihoods to the continuous HMM evaluation function.
nStates	Number of states in the HMM.
nSymbols	Number of discrete symbols used in the discrete HMM. Symbols refers to the number of unique codebook entries to which the observations are quantized. For semi-continuous HMMs, nSymbols represents the number of ccgpdfs used.
nQuant	Dimension of quantization space. The dimension of the observation vectors, the reference vectors used in VQ, and the mean vectors of the Gaussians used in the semi-continuous and continuous models..
flags	RL_HMM_LOGARITHMIC Currently the only flag supported for InitHMMServer , which indicates that *symProb*, *weightVect*, and *iprob* are logarithmic. The values for these parameters should be negative log probabilities. The logarithm (base 10) of the probability (in the range 0 to 1) should be taken and the sign changed to positive. For short integers, the numbers might need to be scaled to fit them in the 16-bit range and to prevent overflow. The same scaling factor should be used for *tprob*, *symbProb*, and *iprob* values. Currently, logarithmic data is the only type supported so this flag must be used.

Discussion

The functions `RL?InitHMMServer()`, `RL?InitSemicontHMMServer()`and `RL?InitContHMMServer()`, create and initialize an HMM by passing all of the required arguments to it. The function then returns an ID for the newly created HMM or -1 if an error occured. This ID can then be used when calling the functions `RL?EvalHMMViterbi()`, `RL?EvalSemiContHMMViterbi()`, and `RL?EvalContHMMViterbi()` to evaluate an HMM or a group of HMMs. See Figures 7-2, 7-3, and 7-4.

FreeHMMServer

Deletes and frees the storage associated with an HMM server.

```
void RLFreeHMMServer(int HMMId);
```

HMMId	The ID of the HMM server to delete.

Discussion

The function `RLFreeHMMServer()` destroys and frees the storage space for the HMM server corresponding to *HMMId*. The HMM server also is removed from all the HMM classes. The value returned is 1 if the function ended successfully and -1 otherwise. An error status is set if the HMM server does not exist. (See "Error Functions" in Chapter 2.)

EvalHMMViterbi

Evaluates a set of HMMs for a given observation sequence using the Viterbi Algorithm.

```
int RLwEvalHMMViterbi(int HMMOrClassId, int *vect, int
nObserv, int *bestHMMId, int doScale, int *scaleFactor);
    /* discrete HMM for short integer vectors  */

float  RLsEvalHMMViterbi(int HMMOrClassId, int *vect, int
nObserv, int *bestHMMId);
    /* discrete HMM for single precision real vectors  */

int RLwEvalSemicontHMMViterbi(int HMMOrClassId, short
*nqVect, int nObserv, int *bestHMMId, int doScale, int
*scaleFactor);
    /* semi-continuous HMM for short integer vectors  */

float  RLsEvalSemicontHMMViterbi(int HMMOrClassId, float
*nqVect, int nObserv, int *bestHMMId);
    /* semi-continuous HMM for single precision real
vectors  */

int RLwEvalContHMMViterbi(int HMMOrClassId, short
*nqVect, int nObserv, int *bestHMMId, int doScale, int
*scaleFactor);
    /* continuous HMM for short integer vectors  */

float  RLsEvalContHMMViterbi(int HMMOrClassId, float
*nqVect, int nObserv, int *bestHMMId);
    /* continuous HMM for single precision real vectors */
```

HMMOrClassId The ID of the HMM server or HMM class that is to be used, which is created originally by a call to `RL?Init?HMMServer()` or `RLCreateHMMClass()`. If *HMMOrClassId* is less then zero, ID is considered to be a class ID, if greater then zero, it is considered to be the ID of

	an HMM server. All HMM servers from class must be initialized with same values of *nSymbols* and *nQuant* parameters.
vect	A vector of codebook indices corresponding to the sequence of codebook vectors, which most closely match the sequence of observations. The dimension of *vect* is *nObserv*.
nqVect	The observation sequence non-quantized vector of length *nObserv*. Each element of *nqVect* is a pointer to an observation vector of length *nQuant*. .
nObserv	The number of symbols in the observation sequence.
bestHMMId	The ID of the HMM with the best (minimum) computed score. The minimum negative log probability score corresponds to the maximum probability score. This argument is useful only when the ID of a class is passed to the function in the argument *HMMORClassID*. The value of *bestHMMId* is returned by the function.
doScaleOutput, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RL?EvalHMMViterbi() evaluates a single HMM or a group of HMMs (that is, Class) for the given observation vector *vect* using the Viterbi algorithm. If *HMMOrClassId* is positive, it is considered to be an HMM server ID, and only one model is evaluated. If *HMMOrClassId* is negative, it is considered to be an HMM class ID, and a set of models are evaluated. The value returned is the best score (or probability in the log domain).

The Viterbi algorithm used for recognition computes the probability (that is, the likelihood) that the most likely state sequence in the HMM produces the observation sequence O_t (t = 1,2...,T).

$$P^v = \max_{j=1,N}\{ \phi_t \} = \text{Viterbi Probability}$$

where ϕ_t is computed using the recursive pair

$$\phi_{t+1} = \max_{j=1,N}\{ \phi_t * a_{ij}\} * b_j(O_{t+1}),\ t = 1,2...,T\text{-}1$$

and $\phi_1(j) = \pi_j * b_j(O_1)$

In this implementation, negative log probabilities are used, so the above equations reduce to :

$$\log(P^V) = -\min_{j=1,N}\{\log(\phi_t)\}$$

where ϕ_t is computed using the recursive pair

$$\log(\phi_{t+1}) = \min_{j=1,N}\{\log(\phi_t) - \log(a_{ij})\} - \log(b_j(O_{t+1})),\ t = 1,2...,T\text{-}1$$

and

$$\log(\phi_1(j)) = -\log(\pi_j) - \log(b_j(O_1))$$

Note that in this equation, for a discrete HMM, $b_j(o_t) = b_{jk}$ is the probability of producing observation o_t when in state j and the observation o_t is vector quantized to reference pattern R_k. For a semi-continuous or continuous HMM the $b_j(O_t)$ for all states j at time t are computed by Gaussian mixture probability evaluation. See figures 7.3 and 7.4 respectively. The quantity a_{ij} is the transition probability going from state i to state j and π_j is the initial probability of state j.

CreateHMMClass

Creates an HMM class (a collection of HMM servers).

```
int RLCreateHMMClass();
```

Discussion

The function `RLCreateHMMClass()` creates and returns the ID of a new HMM class. If an error occurs -1 is returned. An HMM class is a collection of HMM servers that can be passed as a group to the `RL?EvalHMMViterbi()` function. HMM servers can be added to a class using the function `RLAddHMMToClass()` and removed from a class using the function `RLRemoveHMMFromClass()`. An HMM class contains HMM servers of one type (discrete, semi-continuous or continuous). Class type is defined with the addition of the first HMM server.

FreeHMMClass

Deletes and frees the storage associated with an HMM class.

```
int RLFreeHMMClass(int classId);
```

classId The ID of the HMM class to delete.

Discussion

The function `RLFreeHMMClass()` deletes and frees the storage associated with an HMM class. The value returned is 1 if the function ended successfully and -1 otherwise. An error status is set if the class does not exist. (See "Error Functions" in Chapter 2.)

AddHMMToClass

Adds an HMM server to a class.

```
int RLAddHMMToClass(int classId, int HMMId);
```

classId	The ID of the HMM class.
HMMId	The ID of the HMM server to add to the HMM class.

Discussion

The function `RLAddHMMToClass()` adds an HMM server to a class. The value returned is 1 if the function ends successfully (the HMM server is added to the class or already belongs to it) and -1 otherwise. An error status is set if the class or the HMM server does not exist or if the server is added to class of another type. (See "Error Functions" in Chapter 2.)

RemoveHMMFromClass

Removes an HMM server from a class.

```
int RLRemoveHMMFromClass(int classId, int HMMId);
```

classId	The ID of the HMM class.
HMMId	The ID of the HMM server to remove.

Discussion

The function `RLRemoveHMMFromClass()` removes an HMM server from a class. The value returned is 1 if the function ends successfully (the HMM server is removed from the class or does not belong to it) and -1 otherwise. An error status is set if the class or the HMM server does not exist. (See "Error Functions" in Chapter 2.)

HMMFreeAll

Clean-up function to free storage space and delete all HMM servers and classes.

```
void RLHMMFreeAll();
```

Discussion

`RLHMMFreeAll()` deletes all of the current HMM servers and classes. The function also frees the storage associated with them.

Miscellaneous Functions 8

This chapter includes functions for data conversion, complex vector support and processor type detection.

Data Conversion

The data conversion functions convert scalar or vector data of one type to another data type. When data is converted from a higher-precision representation to lower-precision representation it is referred to as “scaling down.” When converting from a lower-precision representation to a higher-precision representation it is referred to as “scaling up.” Functions are also provided to convert complex data types to real data types.

Scaling Down

The following functions convert scalars or vectors from a higher-precision representation to a lower-precision representation. The scaling options (described in Chapter 1) determine how the data is scaled.

ConvertDown

Scales down a scalar from one data type to another.

```
short int RLswConvertDown(float value, int doScale, int
*scaleFactor);
   /* float-to-short */

short int RLdwConvertDown(double value, int doScale, int
*scaleFactor);
   /* double-to-short */

long RLsiConvertDown(float value, int doScale, int
*scaleFactor);
   /* float-to-long  */

long RLdiConvertDown(double value, int doScale, int
*scaleFactor);
   /* double-to-long */
```

value	The data value to be converted.
doScale, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RLxyConvertDown()scales down a value of type *x* to a value of type *y* using the scaling options provided. The converted value is returned.

Example 8-1 show the code to transform different input values with various scaling options.

Example 8-1 Using the Function RLswConvertDown()

```
/* This example shows the use of the function RLswConvertDown() in
 * transforming different input values with various scaling
 * options.*/

#include  <stdio.h>
#include “rl_cnvrt.h”

main() {
  float      value1 = (float)-3.5;
  float      value2 = (float)6.7;
  float      value3 = (float)32770.0;
  float      value4 = (float)0.2;
  float      value5 = (float)-32777.3;
  short int  outValue1, outValue2, outValue3, outValue4, outValue5,
outValue6;
  int        scaleFactor;

scaleFactor = 0;

  outValue1 = RLswConvertDown(value1, RL_NO_SCALE, &scaleFactor);

  scaleFactor = 2;
  outValue2 = RLswConvertDown(value2, RL_FIXED_SCALE, &scaleFactor);

  scaleFactor = -2;
  outValue3 = RLswConvertDown(value2, RL_FIXED_SCALE, &scaleFactor);

  outValue4 = RLswConvertDown(value3, RL_AUTO_SCALE, &scaleFactor);

  outValue5 = RLswConvertDown(value4, RL_AUTO_SCALE, &scaleFactor);

  outValue6 = RLswConvertDown(value5, RL_SATURATE, &scaleFactor);

  /*  insert code here to print  */
}
```

Output

```
Scaling: RL_NO_SCALE  scaleFactor = 0
    inValue1(float) = -3.5  outValue1(short int) = -3
```

```
Scaling: RL_FIXED_SCALE  scaleFactor = 2
    inValue2(float) = 6.7  outValue2(short int) = 1

Scaling: RL_FIXED_SCALE  scaleFactor = -2
   inValue2(float) = 6.7  outValue3(short int) = 26

Scaling: RL_AUTO_SCALE  scaleFactor = 1
    inValue3(float) = 32770.0  outValue4(short int) = 16385

Scaling: RL_AUTO_SCALE  scaleFactor = -17
    inValue4(float) = 0.2  outValue5(short int) = 26214

Scaling: RL_SATURATE  scaleFactor = 0
    inValue5(float) = -32777.3  outValue6(short int) = -32768
```

bConvertDown

Scales down a vector from one data type to another.

```
void RLswbConvertDown(float *inVect, short int *outVect,
int n, int doScale, int *scaleFactor);
    /* float-to-short */

void RLdwbConvertDown(double *inVect, short int *outVect,
int n, int doScale, int *scaleFactor);
    /* double-to-short */

void RLsibConvertDown(float *inVect, long *outVect, int
n, int doScale, int *scaleFactor);
    /* float-to-long */

void RLdibConvertDown(double *inVect, long *outVect, int
n, int doScale, int *scaleFactor);
    /*double-to-long */
```

inVect	Pointer to the input vector to be converted.
outVect	Pointer to the scaled down output vector.
n	Length of the input (and output) vector.
doScale, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RLxybConvertDown()scales down a vector of type *x* to a vector of type *y* using the scaling options *doScale* and *scaleFactor*.

Example 8-2 show the code to transform different input vectors with various scaling options.

Example 8-2 Using the Function RLdwbConvertDown()

```
/* This example shows the use of the function RLdwbConvertDown() in
 * transforming different input vectors with various scaling
 * options. */

#include  <stdio.h>
#include “rl_cnvrt.h”

main() {
    double     inVect[5] = {-3.5, 32768.0, 0.2, 65000.0, 9.0};
    short int  outVect1[5], outVect2[5];
    int        n = 5, scaleFactor, i;

    RLdwbConvertDown(inVect, outVect1, n, RL_AUTO_SCALE,
&scaleFactor);

    RLdwbConvertDown(inVect, outVect2, n, RL_SATURATE, &scaleFactor);

    /*  insert code here to print  */
}
```

Output

```
inVect: -3.5  32768.0  0.2  65000.0  9.0

Scaling:  RL_AUTO_SCALE  scaleFactor = 1
   outVect1:  -1  16384  0  32500  4

Scaling:  RL_SATURATE  scaleFactor = 0
   outVect2:  -3  32767  0  32768  9
```

Scaling Up

The following functions convert scalars or vectors from a lower-precision representation to a higher-precision representation.

ConvertUp

Scales up a scalar from one data type to another.

```
float RLwsConvertUp(short value, int scaleFactor);
    /* short-to-float */

double RLwdConvertUp(short value, int scaleFactor);
    /* short-to-double */

float RLisConvertUp(long value, int scaleFactor);
    /* long-to-float */

double RLidConvertUp(long value, int scaleFactor);
    /* long-to-double */
```

value	The data value to be converted.
scaleFactor	A scale factor used in scaling up the data. The data is multiplied by $2^{(-scaleFactor)}$ during conversion.

Discussion

The function `RLxyConvertUp()` scales up a value of type *x* to a value of type *y* using the *scaleFactor* argument. The converted value is returned.

Example 8-3 show the code to transform an input value with various scaling options.

Example 8-3 Using the Function RLwsConvertUp()

```
/* This example shows the use of the function RLwsConvertUp() in
 * transforming an input value with various scaling options. */

#include  <stdio.h>
#include "rl_cnvrt.h"

main() {
    short int    inValue = -3;
    float        outValue1, outValue2;
    int          scaleFactor = -2;

    outValue1 = RLwsConvertUp(inValue, scaleFactor);

    outValue2 = RLwsConvertUp(inValue, 1);

    /*  insert code here to print  */
}
```

Output

```
  scaleFactor = -2
      inValue(short int) = -3   outValue1(float) = -12.0

  scaleFactor = 1
      inValue(short int) = -3   outValue2(float) = -1.5
```

bConvertUp

Scales up a vector from one data type to another.

```
void RLwsbConvertUp(short *inVect, float *outVect, int n,
int scaleFactor);
    /* short-to-float */

void RLwdbConvertUp(short *inVect, double *outVect, int
n, int scaleFactor);
    /* short-to-double */

void RLisbConvertUp(long *inVect, float *outVect, int n,
int scaleFactor);
    /* long-to-float */

void RLidbConvertUp(long *inVect, double *outVect, int n,
int scaleFactor);
    /* long-to-double */
```

inVect	Pointer to the input vector to be converted.
outVect	Pointer to the scaled-up output vector.
n	Length of the input (and output) vector.
scaleFactor	A scale factor used in scaling up the data. The data is multiplied by $2^{(-scaleFactor)}$ during conversion.

Discussion

The function `RLxybConvertUp()` scales up a vector of type *x* to a vector of type *y* using the *scaleFactor* argument.

Example 8-4 show the code to transform an input value with various scaling options.

Example 8-4 Using the Function RLwsbConvertUp()

```
/* This example shows the use of the function RLwsbConvertUp()
 * in transforming an input vector with various scaling options.*/

#include  <stdio.h>
#include “rl_cnvrt.h”

main() {
    short int inVect[5] = {0, 2, -9, 13, -20};
    float     outVect1[5], outVect2[5];
    int       n = 5, scaleFactor = -1, i;

    RLwsbConvertUp(inVect, outVect1, n, scaleFactor);

    RLwsbConvertUp(inVect, outVect2, n, 4);

    /*  insert code here to print  */
}
```

Output

```
inVect:  0  2  -9  13  -20

scaleFactor = -1
    outVect1:  0.000000  4.000000  -18.000000  26.000000  -40.000000

scaleFactor = 4
    outVect2:  0.000000  0.125000  -0.562500  0.812500  -1.250000
```

Complex Vector Support

The following functions convert complex vectors to real vectors.

bConvert

Converts a complex vector to a real vector.

```
void RLcsbConvert(SCplx *inVect, float *outVect, int n,
int flags);
    /* float complex-to-float */

void RLvwbConvert(WCplx *inVect, short *outVect, int n,
int flags, int doScale, int *scaleFactor);
    /* short complex-to-short */
```

inVect	Pointer to the input complex vector to be converted.
outVect	Pointer to the output vector. Since the output is no longer complex, its length is half the input, that is, *n*/2. The data in the vector is contiguous.
n	Length of the input vector.
flags	Option that determines how the output is computed. These flags can be ORed when used (for example, `RL_MAG | RL_LOG10`). However `RL_MAG` and `RL_SQMAG` cannot be used together. If you specify `RL_LOG10` with `RL_MAG` or `RL_SQMAG`, the logarithm is taken as the final operation (resulting in log magnitude or log squared magnitude). The supported options are:

`RL_MAG`	Magnitude output
`RL_SQMAG`	Squared magnitude (this is also known as the power spectrum).

RL_LOG10	Logarithm (this is log base 10)
doScale, *scaleFactor*	Refer to "Integer Scaling" in Chapter 1.

Discussion

The function RLxybConvert()converts a complex vector of type *x* to a real vector of type *y* using the *flags* argument. The output, being real, is half the size of the complex input.

Processor Information

This section describes a function that can be used to query for the processor family (for example, 486, Pentium, and so on). The function uses a structure ProcessorInfo to return the information. This structure is defined as follows:

```
typedef struct {
        int family;
        int model;
        int stepping;
        char name[100];
    } ProcessorInfo;
```

The *family* field assumes the following values for each processor family:

Table 8-1 Family Field Values and Descriptions

Value	Description	Processor Name
-1	An error occured in the query function.	
0	8086/88 processor	i086™/i088™
2	80286 processor	i286™
3	80386 processor	i386
4	80486 processor	i486
5	Pentium processor	Pentium
55	Pentium processor with MMX™ technology	P55C
6	Pentium Pro processor	Pentium Pro

GetProcessorInfo

Returns information about the X86 Processor.

```
void RLGetProcessorInfo(ProcessorInfo *info);
```

info	A pointer to a structure of type `ProcessorInfo` that will contain all the details of the processor. This structure is described above.

Discussion

The function `RLGetProcessorInfo()` gets information about the processor that the application is currently running on. Currently, information about the X86 family that the processor belongs to is returned in the structure pointed to by *info*.

Library Information

This section describes a function that can be used to query the version number of the current version of the Recognition Primitives Library. The function uses a structure `LibraryInfo` to return the information. This structure is defined as follows:

```
typedef struct {
        int  major;
        int  minor;
        int  build;
    } LibraryInfo;
```

The *major*, *minor*, and *build* fields are the major, minor, and build numbers respectively of the current version of the library. For example if the library revision is 2.0 build 27, then the fields in this structure are set as

```
major = 2, minor = 0, build = 27
```

These fields are set to -1 in case there is an error in retrieving the information.

GetLibraryInfo

Returns information about the current version of the Recognition Primitives Library.

```
void RLGetLibraryInfo(LibraryInfo *info);
```

info	A pointer to a structure of type `LibraryInfo` that will contain all the details about the version of the library. This structure is described earlier.

Discussion

The function `RLGetLibraryInfo()` gets information concerning the version number of the Recognition Primitives Library that the application is currently using. Currently, the major, minor, and build numbers of the library are returned in the structure pointed to by *`info`*.

Bibliography

Labrosse, J.J., *Using Scaled Arithmetic,* Embedded Systems Programming, March 1995

Kohonen, T., *Self Organization and Associative Memory,* (3rd Ed) New York, ©1989, Springer Verlag.

Gray, R.M., *Vector Quantization*, IEEE ASSP Magazine 1(2):4-29, April 1984.

Hertz, J., Krogh, A., Palmer, R., *Introduction to the Theory of Neural Computation*,
Addison-Wesley, Reading, Ma., ©1991.

Rao, V., Rao, H., *C++ Neural Networks and Fuzzy Logic*,
MIS: Press, New York, New York, ©1993.

Index

L

M

O

P

R

S

V

W