AllMyPapers

AMPLIB™

Software Development Tool Kit

API Reference Guide

Version 8.0
3/2020

040-100034

All My Papers
1840 Snake River Road
Suite C, Katy, TX 77449
(408) 366-6400
www.allmypapers.com

Contents

Introduction

The AMPLIB SDK and MICR

Before the 1950s, all checks were hand sorted. As more people opened checking accounts, banks found it difficult to keep up with the volume. In 1958 Stanford University and the Bank of America developed Magnetic Ink Character Recognition (MICR), which enabled checks to be sorted by computers. MICR characters were designed to have distinctive shapes that could be easily recognized by machines capable of sensing the magnetic particles in the ink.

Now, magnetized ink is not necessary because of the power of desktop computers, low-cost optical scanning technologies, and new advanced Optical Character Recognition (OCR) algorithms. Developers using the AMPLIB high-level programming system can recognize MICR characters printed with any standard ink., and recognize them reliably even in situations where signatures and other handwritten comments on the check partially obscure the MICR characters.

AMPLIB consists of Microsoft Windows compatible Dynamic Link Libraries, associated import libraries, header files, and other support files that can be used by a wide variety of programming languages and systems. For brevity, the product will be referred to as AMPLIB throughout this document.

Goals

Specific goals of AMPLIB include:

- Accurate reading of MICR characters even if obscured by noise or by descenders from the signature on the check.
- Accurate reading of MICR characters on grayscale and color check images.
- Automatic detection of the MICR character line on a check image.
- A clear and understandable API that addresses the desired operation and masks the underlying complexity.
- Automatic memory management of image workspaces.

.

- Automatic verification of captured MICR data from X9 Image Cash letter files or check processing system databases.

.

- Automatic image usability verification of the MICR Codeline contained on the item’s image.

- Transportability to different operating system platforms as well as hardware platforms. The library supports Windows 95/98/ME and Windows NT/2000.

Optional Goals

The API includes options to assist the developer in providing desired high level results.

- Voting between an existing result (hardware or software generated) and the AMPLIB result.
- Modifying the source image in the same manner as was used to achieve accurate MICR read result.
- Field parsing for standard fields such as route, account, amount and check number.

The AMPLIB SDK and Bar Code Recognition

Reading bar codes from binary images has become relatively common. The AmpLib bar code binary engine has tested better than any of its competitors on a wide range of images. The latest version of AmpLib adds bar code reading from **color** and **grayscale** images.

The use of **color** and **grayscale** images can be used to improve read rate, read more data from the same size bar code or read the same amount of data from a smaller bar code.

Installation

System Requirements

AMPLIB requires Microsoft Windows to be installed on your system to run correctly and supports:

- Windows 95/98/ME (TERMINATED)
- Windows 2000/NT (TERMINATED)
- Windows XP/Vista (TERMINATED)
- Windows 8 (TERMINATED)
- Windows 7/8.1/10
- Windows Server 2003 (TERMINATED)
- Windows Server 2008, 2012

For best performance, AllMyPapers recommends a minimum of 2 GB.

A compiler or other language product that can call DLL-resident functions is required to use the toolkit.
COM support has terminated and .NET is recommended.

Software Installation Process

The AMPLIB - MICR/Barcode Developer's Toolkit is only available on the Internet. After downloading the software refer to the README installation notes for further instructions.

DLL Based API

AMPLIB is built as a set of Dynamic Link Libraries (DLLs) using the standard Microsoft convention of **_stdcall** calling sequences. This provides a degree of language independence between AMPLIB and a user-written application. Any language that can call a DLL-resident function can access AMPLIB. Care has been taken in the function call arguments to require a minimum of special types so function prototypes can be written in almost any language. As a result, AMPLIB can be used with C/C++, Delphi, Visual Basic and other languages.

Single Threaded/Multi Threaded DLLs

The standard AMPLIB DLL is single threaded. This is what is available for download from the allmypapers website. Multi thread versions of the DLL are available under special license agreement for the MICR OCR, MICR Verify and Image Processing engines. Additional engines are being added to the multi thread DLLs so please check with sales for the latest set. See the ampGetFileVersion for the method to determine which DLL is installed.

Customer Support

The AllMyPapers Support Policy is defined in a package insert.

Description of AMPLIB Components

Components

- AMPLIB.HLP
- AMPLIB.INI
- AMPLIB.OUT
- AMPLIBAPI.H
- AMPLIBAPIC.H
- AMPBARAPI.H
- AMPBARAPIC.H
- AMPLIB.BAS
- AMPLIB.PAS
- AMPLIB.LIB
- AMPLIB.DLL
- AMPLM.DLL
- AMPPX.DLL
- LICMGR.EXE
- LOGOSDM.DLL
- LVSNUM.FNT
- LVSOCRA.FNT
- LVSOCRB.FNT
- LVSMICR.FNT
- LVSFSB.FNT
- LVSCAR.FNT
- STORAGE.DAT

The set of DLL files implement the AMPLIB system.

AMPLIB.HLP is the AMPLIB help file used with the standard Windows Help Engine. AMPLIB.INI describes configuration parameters and must be in the same directory as the AMPLIB.DLL.

All of the executable files should be in your PATH if you wish to run them from another location. By default, all the files are placed in Program Files\All My Papers\Amplib\Bin. If there are not found there, the current program directory will be searched and finally the users PATH.

The *.H and *.BAS, and *.PAS files are headers for the Microsoft C, Visual BASIC, and Delphi 4.0 languages respectively. The files “amplibapi.h” and "ampbarapi.h" should be considered the master versions.

The files “storage.dat”, "lvsmicr.fnt", "lvsocra.fnt", "lvsocrb.fnt", "lvsnum.fnt", lvsfsb.fnt,” and “lvscar.fnt.” contain shape definitions for the MICR, OCR-A, OCR-B, generic numeric characters, four-state bar codes, and check courtesy amount characters. The file "logosdm.dll" is used for recognizing data matrix 2D barcodes. All the files should be placed in the same directory as AMPLIB.DLL.

Distribution of Components

The header files (*.h,*.pas,*bas) and library file (*.lib) are **NOT** to be distributed unless given permission by a specific contract.
The files “storage.dat”, "lvsmicr.fnt", "lvsocra", "lvsocrb", "lvsnum.fnt" are only needed to be included in MICR distributions.
There a number of files that are not listed above and that are used to support the demo applications included in AmpLib. These files should not be distributed.

Initialization File: AMPLIB.INI

The AMPLIB.INI file provides startup time control parameters. The Trace facility is the most commonly used. By invoking Trace at start time, issues such as licensing can more easily be resolved. The user can control tracing separately with the **`ampTraceEnable`** function. In general, running with Trace on will slow down processing by a large amount.

The license control process will issue warnings when temporary licenses are about to expire. The Quiet parameter in the ini file will prevent this from happening. This is particularly important in server environments.

Example INI file that will turn on trace, name the trace file and turn off license expiration messages.

```
[Trace]
Enable=yes
File=h:\amplib.log
Quiet=1
```

Introduction to AMPLIB Programming

Overview

AMPLIB gives you a powerful yet simple paradigm for adding MICR character and barcode recognition to your imaging applications under Microsoft Windows. Central to AMPLIB are a few basic terms and concepts, which are used throughout the AMPLIB system. Master these and you have mastered AMPLIB.

C/C++ Programming

If you are not already familiar with C/C++ programming, you should first familiarize yourself with the language, because AMPLIB functions are described using C-language syntax. The AMPLIBAPI.H header file is written to Microsoft standards. C compiles may require minor modification to the header file to avoid name mangling.

Visual BASIC Programming

AMPLIB ships with a Microsoft Visual BASIC compatible declaration module, which can be included in your Visual BASIC applications. Your Visual BASIC Programming Guide contains reference information on converting C data declarations to Visual BASIC declarations. For flexibility, you may want to change some of the declaration argument types to "**As Any**" to allow passing NULL pointer arguments, or other variants.

Delphi Programming

AMPLIB ships with a Borland Delphi 4.0 compatible declarations module which can be included in your Delphi applications. Many of the example programs are written in Delphi.

.NET Programming

AMPLIB ships a separate module for .NET programming. This includes applications written in VB and CSharp along with their source code.

Quick Start

The purpose of this section is to provide source code examples that show how easy it is to use AMPLIB to read MICR characters from image files. The two examples provided here are written in C/C++, but the sequence and names of the AMPLIB calls would be the same for Visual Basic, Delphi, or any other programming environment. The first code example is shown below.

```
  //  Reading MICR from checks scanned on a check scanner.
  //  Attempt to read any MICR symbols on the check image.
  //      Rotate 180 degrees if cannot find on bottom of page and try top.
  //   Return 0 if successful
int nEasyMICRRead (LPSTR szFilename, LPSTR szResult)
{
 int nStat;
 PWORKIMAGE pWFile;         // Source MICR image
 ampPREPINFO piInfo;         // Preprocessing parameters
 ampMICRINFO ampMI;          // MICR parameters/results data structure

   // Create AMP image for the source image file
 nStat = ampCreateGrayWorkImage (&pWFile, 0, 0);
 if  (nStat)   return nStat;

   // Load the AMP image from the specified file
 nStat = ampLoadImage(pWFile, szFilename, "", NULL, 1);
 if (nStat)
 {
      ampFreeImage(pWFile);
      return nStat;
 }

 strcpy (ampMI.infile,  "");   // Work files normally found on users path
 strcpy (ampMI.outfile, "");  // Not Used

 ampMI.Dorepair = 1;         // Repair degraded characters
 ampMI.Do180 = 1;            // Look for upside down MICR codes
 ampMI.Resolution = - 1 ;  // Use image resolution
 ampMI.Code = 0 ;            // Default to E13B
 ampMI.NoBlanks = 0 ;      // Report blanks
 ampMI.UseTranslator = 0 ;       // Use default character translation
 ampMI.NoRules = 0;         // Use rules
 ampMI.Rules = 0 ;            // Type of rules: ABA
 ampMI.MinCon = 80 ;              // Minimum confidence
 ampMI.DoImageRepair = 0 ;    // No output of repaired image
 ampMI.MaxTime = 0 ;       // No limit on time to process
 ampMI.StartTime = 0 ;      // 0 for first entry

   // Read the MICR characters
 nStat = ampReadMicr (pWFile, &ampMI);
 if (nStat)
 {
      ampFreeImage(pWFile);
      return nStat;
 }

   // The results are now in ampMI.resultchar so add
   // a null character to make it a "c" compatible string
   // and copy it so the calling routine can use it
  ampMI.resultchar[ampMI.lastchar] = 0;
  strcpy (szResult, &ampMI.resultchar[0]);

   //  All done, release image memory
 ampFreeImage(pWFile);
 return 0;
}
```

This example is designed to use an image file that contains only a check. It assumes that there is no substantial skew to the image, and that there are no margins around the check. The image would be typical of what a check scanner generates.

AMPLIB has the capability to automatically rotate, deskew, and crop check images that are in an irregular format. The following example uses the ampPrepMicr function to prepare a check image that has been scanned on a traditional page scanner.

```
  //
  //  Reading MICR from checks scanned on letter size page scanner.
  //
  //  Pre-processing the images from a letter size page paper scanner
  //  with black edges. The same process, with a parameter change, works on
  //  white edge scanners as well.
  //          Black edges will be removed.
  //          Check image will be rotated 90 degrees if taller than wider.
  //          Check image will be deskewed.
  //
  //  Attempt to read any MICR symbols at the bottom of the specified check
  //  image  and if none are found, rotate the image 180 degrees and try again.
  //
  //  Return 0 if successful

int nEasyMICRRead_Prep (LPSTR szFilename, LPSTR szResult)

{
  int nStat;

  PWORKIMAGE pWFile;           // Source MICR image
  PWORKIMAGE pWMicr;           // Image after preprocessing
  ampPREPINFO piInfo;          // Preprocessing parameters
  ampMICRINFO ampMI;           // MICR parameters/results data structure

    // Create AMP image for the source image file
  nStat = ampCreateWorkImage (&pWFile, 0, 0);

  if (nStat)
      return nStat;

    // Create AMP image for the preprocessed image
  nStat = ampCreateWorkImage (&pWMicr, 0, 0);

  if (nStat)
  {
      ampFreeImage(pWFile);
      return nStat;
  }

    // Load the AMP image from the specified file
  nStat = ampLoadImage(pWFile, szFilename, "", NULL, 1);

  if (nStat)
  {
      ampFreeImage(pWFile);
      ampFreeImage(pWMicr);
      return nStat;
  }
```

```
   // Initialize MICR prep structure to deskew with black edges
  piInfo.BlackEdges = 1;

   // Copy MICR image doing any scaling, rotation, and deskewing needed
  nStat = ampPrepMicr(pWFile, pWMicr, &piInfo);

  if (nStat)
  {
       ampFreeImage(pWFile);
       ampFreeImage(pWMicr);
       return nStat;
  }

  strcpy (ampMI.infile,  "");   // Work files normally found on users path
  strcpy (ampMI.outfile, "");  // Not used

  ampMI.Dorepair = 1;         // Repair degraded characters
  ampMI.Do180 = 1;            // Look for upside down characters
  ampMI.Resolution = - 1 ;  // Use image resolution
  ampMI.Code = 0 ;            // Default to E13B
  ampMI.NoBlanks = 0 ;       // Report blanks
  ampMI.UseTranslator = 0 ;       // Use default character translation
  ampMI.NoRules = 0;          // Use rules
  ampMI.Rules = 0 ;            // Type of rules: ABA
  ampMI.MinCon = 80 ;              // Minimum confidence
  ampMI.DoImageRepair = 0 ; // No output of repaired image
  ampMI.MaxTime = 0 ;       // No Limit on time to process
  ampMI.StartTime = 0 ;      // 0 for first entry

   // Read the MICR characters
  nStat = ampReadMicr (pWMicr, &ampMI);

  if (nStat)
  {
       ampFreeImage(pWFile);
       ampFreeImage(pWMicr);
       return nStat;
  }

   // The results are now in ampMI.resultchar so add
   // a null character to make it a "c" compatible string
   // and copy it so the calling routine can use it
  ampMI.resultchar[ampMI.lastchar] = 0;
  strcpy (szResult, &ampMI.resultchar[0])

  //  All done, release image memory
  ampFreeImage(pWFile);
  ampFreeImage(pWMicr);
  return 0;
}
```

Using these two examples as an overview to the general flow of MICR reading with AMPLIB, let’s continue by discussing some of the fundamentals of AMPLIB that are needed to construct more advanced applications.

Work Image

A *work image*, in the AMPLIB sense, is just a block of storage used to hold an array of pixels to form an image. AMPLIB takes care of managing the storage for your work images. Work images are identified by a 32-bit variable which contains a pointer to the work image. The calling application provides the space (4 bytes) and name for this pointer. Calling ampCreateWorkImage allocates the memory needed for the work image itself.

Work images are a fundamental data structure of AMPLIB. All image operations are directed toward a work image.

A work image may be either *fixed size* or *variable size*. For most applications, variable size work images will be the most convenient. Variable size work images will automatically receive as much storage as needed; they may shrink and grow dynamically during use. Fixed size work images receive an initial storage allocation when they are created and that amount never changes. Both types of work images may be destroyed (and their storage freed) at any time under program control.

To create a variable size work image, create the work image with a width and height of 0. Non-zero width and height values will create a fixed size image.

A work image is rectangular in nature and defined as having a *pitch* and *height.* Pitch is the number of pixels across a line of the work image, and height is the number of lines in the work image. For example, a letter sized image, 8.5" x 11" at 300 dpi would typically have a pitch of 2560 and a height of 3296 pixels. Pitch is an important concept, as will be discussed with sub-images below. Remember that pitch is a distance measurement, the number of pixels between adjacent lines.

Color and Grayscale Images

Color and grayscale image files are processed at load or scan time to be stored internally as color or grayscale image data. MICR and Barcode OCR can be much more effective with grayscale data but rarely uses color for processing. The color information is usually converted to grayscale at load time and optionally designated colors may be dropped out.

Sub-Images

AMPLIB always operates on the *active sub-image area* of a given work image. A sub-image is a rectangular region of a work image that is equal to or smaller than the work image pitch and height. Figure 1 shows the relationship of a sub-image to an image.

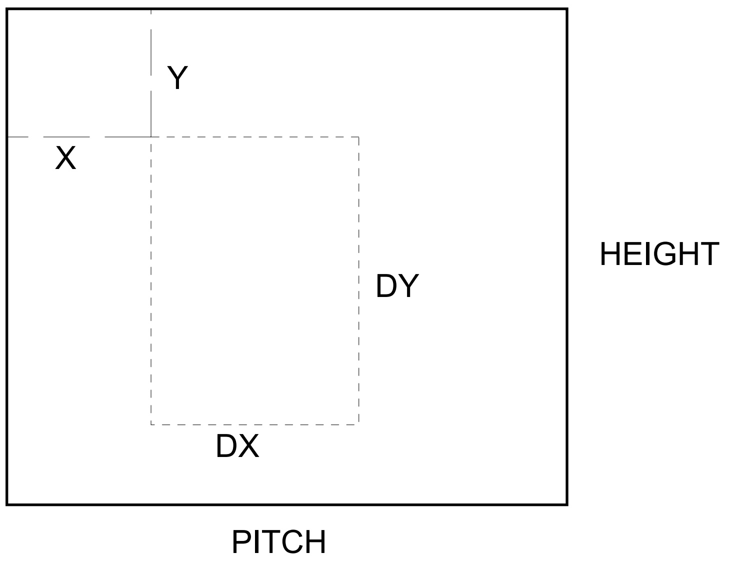


Figure 1 - Work Image / Sub-image relation

Narrowing the sub-image can speed up processing for some operations.

A sub-image area is a rectangular region offset from the upper left corner (ULC) of the work image. The upper left corner of the sub-image is offset (X,Y) pixels from the ULC of the work image, and its size is DX pixels wide by DY pixels high.

Even though the sub-image is only DX pixels wide, pixels on adjacent lines are still PITCH pixels apart. This must be true in order for the sub-image to form a sliding window across the work image.

Sub-images allow for region-of-interest selection of image data. You may only be concerned with a small area of the whole image, and narrowing down the active sub-image area can speed processing time for some applications. You can vary the sub-image definition by calling **ampSetImageMetrics**.

Alias Images

Alias images can simplify region-of-interest processing of image data.

An *alias image* is just a special type of sub-image. An alias image has no storage of its own; it is derived from a base work image, shares storage with the work image, but possesses its own values for X, Y, DX, and DY. A work image may have any number of alias images, each with its own unique name.

Alias images may be used as either a source or destination image with most AMPLIB API's.

Alias images offer an advantage over modifying the sub-image definition for a work image. For example, you could have a work image named 'Original' and an alias of 'Original' named 'Sub'. You can always refer to the whole image with the name 'Original' regardless of the definition of 'Sub'. You needn't clutter your code by continually changing the sub-image definition back and forth.

NOTE: When working with variable size work images, it is important to remember that the work image may be reallocated to completely different dimensions each time it is the destination image of some toolkit operation. So you should always create the alias image after the work image is loaded and/or processed.

In general, wherever a work image can be used, an alias image can be used also. In the API descriptions that follow, the term *image*, when unqualified, means either a work image or an alias image.

In general, local alias images may not be used as a destination, only as an image source. There are some exceptions to this rule, such as the ***ampCopyImage*** *function.*

Compiling and Linking

Overview

This section describes the process for compiling and linking an AMPLIB program written in Microsoft C.

C programs should include the header file "amplibapi.h". The header file has complete prototype descriptions of the API functions that are available.

Your program should be linked with "amplib.lib", which is an import library that contains the DLL linkage information.

To shut down your application, you should intercept the WM_CLOSE message and call **ampFreeAllImages** to perform AMPLIB memory cleanup. Other exit paths in your application should post or send a WM_CLOSE to your application's window, rather than calling PostQuitMessage() directly, to ensure that the WM_CLOSE code is performed.

```
case WM_CLOSE:
    ampFreeAllImages();
    DestroyWindow(hWnd);
    return 0;
```

Other API's are called as needed to implement your solution. See the source code example for details.

Build Process

Build a project with your Visual C++ Development System, and include the appropriate AMPLIB library file in your **Settings|Linker** options, and the ..\amplib\include directory in the **Settings|Preprocessor** options. Compile and link as with any other C program.

C++ Compiles

When compiling in C++ mode, you must be sure that the compiler interprets the names in standard “C” mode, rather than using C++ decoration. Use the following format to include the header files:

```
extern “C” {
      #include “amplibapi.h”
}
```

The AMPLIB API

Overview

This chapter describes the AMPLIB API (Applications Programming Interface) available to C/C++ programs and other DLL compatible languages.

Summary of AMPLIB APIs

Image Management Functions

ampCreateWorkImage

ampCreateGrayWorkImage

ampCreateColorWorkImage

ampCreateImageAlias

ampFreeImage

ampFreeAllImages

ampGetImageMetrics

ampGetGrayImageMetrics

ampGetImageResolution

ampSetImageMetrics

ampSetImageResolution

MICR Functions

ampFieldVerify

ampFieldVerifyEx

ampParseMicr

ampPrepMicr

ampReadMicr

ampReadCamera

ampReadMicrCamera

ampReadScannerForChecks

ampReadMicrRepair

ampVoteIRDRepair

ampVoteIRDRetry

ampVoteMicrRepair

ampVoteMicrRetry

Bar Code Functions

ampReadBarCodes

ampGetBarCodeData

OCR Functions

ampReadOcr

File Functions

ampCreateDIB
ampCreateDIBSection
ampLoadClipboard
ampLoadDIB
ampLoadDIBHandle
ampLoadDIBSectionHandle
ampLoadImage
ampLoadImageHnd
ampLoadImageBuffer
ampSaveClipboard
ampSaveImage
ampSaveImageHnd
ampSetImageMargins
ampSetInputImageMetrics

Image Manipulation Functions

ampBitBltImage
ampClearImage
ampConvertImage
ampCopyImage
ampDeSkew
ampDitherImage
ampFillImage
ampGrayMirrorImage
ampGrayProcesses
ampGrayScaleResolution
ampInvertImage
ampMirrorImage
ampOutsideFillImage
ampRotateImage
ampScaleImage
ampThresholdImage

Image Filtering Functions

ampDeBorder
ampDeLine
ampDeShade

ampDeSpec

ampDeStreak

ampFilterImage

Miscellaneous Functions

ampAssembleMICR

ampCheckImageQuality

ampGetImageAddress

ampGetImageBlock

ampGetImageInfo

ampGetLicenseInfo

ampGetMessageText

ampGrayGetImageBlock

ampGrayPutImageBlock

ampPutImageBlock

ampTrace

ampTraceEnable

API Description Format

The following sections describe the actual AMPLIB API's. They are grouped by category and presented alphabetically in each group.

```
ampSampleAPI( arg1 ,arg2 ,arg3 )
```

```
type arg1;
type arg2;
type arg3;
```

The descriptive text will have details on the arguments and purpose of the call.

The first argument is typically a Pointer/Handle to a PWORKIMAGE structure that defines all of the attributes and contents of the image.

Text arguments that select function options or modes are not case sensitive. For example, "noheader" and "Noheader" and "NOHEADER" are all equivalent.

Some functions use a **ampRECT** structure to pass a sub-image definition. This structure is analogous to the Windows RECT struct, except that ampRECT is defined using long ints. This ampRECT structure will always be used as follows:

```
rect.right = DX
rect.bottom = DY
rect.left = X
rect.top = Y
```

Most functions return 0 for successful completion, non-zero if an error was detected in the call. The error returns are defined in AMPLIBAPI.H as ampERR_xxxx.

Note that API's are also listed in the index sans the 'amp' prefix to make topical lookup easier.

Image Management Function

This set of functions is concerned with managing the work image space. Work images are referenced by pointer only; the actual contents of the work image structure is irrelevant to the user. Work images are created in response to an API function call, and all storage management is automatic. Your program can release the image space by calling **ampFreeAllImages**.

ampCreateWorkImage

```
int ampCreateWorkImage( &pW, maxwidth,
            maxheight )
```

```
    PWORKIMAGE *pW;
    long maxwidth;
    long maxheight;
```

Warning: loading a color or grayscale file into a binary will cause the data to be thresholded and stored as a binary image.

Creates a binary work image with the identifier *pW* which is a pointer to the work image structure created by AMPLIB. AMPLIB will allocate all memory needed for the workimage structure and update the value of pW with the pointer to that memory.

Images may be created as either *fixed size* or *variable size* images. For almost all applications, variable size images are your best choice. Variable size images are reallocated on demand to adapt to the needed memory requirements. Fixed size images retain a fixed amount of memory during their lifetime; they are primarily used when it is desired to "tile" output into a mosaic of sub-images.

You create a variable size image by setting both *maxwidth* and *maxheight* to zero. If a fixed size work image is desired, both *maxwidth* and *maxheight* must be non-zero. The work image metrics will be initialized as follows:

```
    X = Y = 0
    DX = PITCH = maxwidth
    DY = HEIGHT = maxheight
```

Note that the PITCH value for work images will always be rounded up to the nearest multiple of 32. The image metrics can be modified with the **ampSetImageMetrics** API.

This API always returns 0 if the function was successful, non-zero otherwise.

Normally, variable size images will be the best choice and the image space will grow or shrink as needed. In some cases, though, a fixed size image will be the better choice. When a variable size image is given as the destination result of some image processing operation, the destination image will be re-sized in accordance with the result. When a fixed size image is given as the destination image, the resulting image will be placed in the active sub-image. If the sub-image area is too small, an error will be generated. Using fixed size images allows you to control the "tiling" of image data from various image operations. If a fixed size image needs to be made bigger or smaller, it must first be destroyed and then re-created.

Example:

```
    ampCreateWorkImage( pWMyimage, 0, 0 );
    ampCreateWorkImage( pWOther, 2528, 3296);
```

ampCreateGrayWorkImage

```
int ampCreateGrayWorkImage( &pW, maxwidth,
            maxheight )
```

```
PWORKIMAGE *pW;
long maxwidth;
long maxheight;
```

Warning: Loading a binary file into a grayscale work image will cause the work image to become binary. The user should verify the "bits per pixel" value for a work image before loading and "free and create" the image before loading a new file. This restriction is necessary for support of fixed size work images.

Creates an 8-bit grayscale work image with the identifier pW which is a pointer to the work image structure created by AMPLIB. AMPLIB will allocate all memory needed for the workimage structure and update the value of pW with the pointer to that memory.

Images may be created as either *fixed size* or *variable size* images. For almost all applications, variable size images are your best choice. Variable size images are reallocated on demand to adapt to the needed memory requirements. Fixed size images retain a fixed amount of memory during their lifetime; they are primarily used when it is desired to "tile" output into a mosaic of sub-images.

You create a variable size image by setting both *maxwidth* and *maxheight* to zero. If a fixed size work image is desired, both *maxwidth* and *maxheight* must be non-zero. The work image metrics will be initialized as follows:

```
X = Y = 0
DX = PITCH = maxwidth
DY = HEIGHT = maxheight
```

Note that the PITCH value for work images will always be rounded up to the nearest multiple of 32. The image metrics can be modified with the **ampSetImageMetrics** API.

This API always returns 0 if the function was successful, non-zero otherwise.

Normally, variable size images will be the best choice and the image space will grow or shrink as needed. In some cases, though, a fixed size image will be the better choice. When a variable size image is given as the destination result of some image processing operation, the destination image will be re-sized in accordance with the result. When a fixed size image is given as the destination image, the resulting image will be placed in the active sub-image. If the sub-image area is too small, an error will be generated. Using fixed size images allows you to control the "tiling" of image data from various image operations. If a fixed size image needs to be made bigger or smaller, it must first be destroyed and then re-created.

Example 1:

```
ampCreateGrayWorkImage( pWMyimage, 0, 0 );
ampCreateGrayWorkImage( pWOther,2528,3296);
```

Example 2:

```
//  Mixed binary and gray files
// hGlobalGray already exits
```

```
rc = ampGetGrayImageMetrics (hGlobalGray,
dx, dy, ix, iy, pitch, height, bpp ) ;
if (rc=0)
{
      if (bpp = 1) // bpp is bits per pixel
      {     // Previous load was binary
            // release and create a gray
            ampFreeImage(hGlobalGray);
            ampCreateGrayWorkImage(hGlobalGr
            ay, 0, 0);
      }
}
```

ampCreateColorWorkImage

```
int ampCreateColorWorkImage( &pW, maxwidth,
            maxheight, params )
```

```
PWORKIMAGE *pW;
long maxwidth;
long maxheight;
int params;
```

Warning: Loading a TIFF binary or grayscale file into a color work image will demote the work image to either binary or grayscale respectively. The user should verify the "bits per pixel" value for a work image before loading and "free and create" the image before loading a new file. This restriction is necessary for support of fixed size work images.

Creates a 32-bit color work image with the identifier pW which is a pointer to the work image structure created by AMPLIB. AMPLIB will allocate all memory needed for the workimage structure and update the value of pW with the pointer to that memory.

Images may be created as either *fixed size* or *variable size* images. For almost all applications, variable size images are your best choice. Variable size images are reallocated on demand to adapt to the needed memory requirements. Fixed size images retain a fixed amount of memory during their lifetime; they are primarily used when it is desired to "tile" output into a mosaic of sub-images.

You create a variable size image by setting both *maxwidth* and *maxheight* to zero. If a fixed size work image is desired, both *maxwidth* and *maxheight* must be non-zero. The work image metrics will be initialized as follows:

X = Y = 0
DX = PITCH = *maxwidth*
DY = HEIGHT = *maxheight*

Note that the PITCH value for work images will always be rounded up to the nearest multiple of 32. The image metrics can be modified with the **ampSetImageMetrics** API.

This API always returns 0 if the function was successful, non-zero otherwise.

Normally, variable size images will be the best choice and the image space will grow or shrink as needed. In some cases, though, a fixed size image will be the better choice. When a variable size image is given as the destination result of some image processing operation, the destination image will be re-sized in accordance with the result. When a fixed size image is given as the destination image, the resulting image will be placed in the active sub-image. If the sub-image area is too small, an error will be generated. Using fixed size images allows you to control the "tiling" of image data from various image operations. If a fixed size image needs to be made bigger or smaller, it must first be destroyed and then re-created.

Example 1:

```
ampCreateColorWorkImage(pWMyimage,0,0,0);
ampCreateColorWorkImage(pWNext,2528,3296,0);
```

Example 2:

```
//  Mixed binary, gray, and color files
// pWColor already exists
rc = ampGetGrayImageMetrics (pWColor, dx,
dy, ix, iy, pitch, height, bpp ) ;
if (rc=0)
{
      if (bpp < 32) // bpp is bits per pixel
      {     // Previous load was not color
            // release and create a color
            ampFreeImage(pWColor);
            ampCreateColorWorkImage(pWColor,
            0, 0, 0);
      }
}
```

ampCreateImageAlias

```
int ampCreateImageAlias( pW, pWAlias )
```

```
    PWWORKIMAGE pW;
    PWWORKIMAGE * pWAlias;
```

This function creates an alias image for the work image *pW*. The alias work image may be referenced with *pWAlias*. Alias images are particularly useful in working with sub-images of a given work image, also known as a region-of-interest. As many alias images as desired may be created for any work image. This allows you to create multiple sub-images and reference them by name, while still having the entire work image definition available at all times.

The alias image receives all the attributes of its base work image, but retains its own definitions for X, Y, DX, and DY (the *sub-image metrics*). The initial values for the alias image metrics are the same as its base work image *at the time of creation of the alias*. The alias image remains in existence until it is destroyed explicitly, or its base image is destroyed.

Note the importance of the qualifier, "*at the time of creation*", in the preceding paragraph. If an alias is created from a variable size work image prior to storing anything in that work image, the alias will have a sub-image definition of X=Y=DX=DY=0. You must define the alias image metrics first before using the alias image by calling **ampSetImageMetrics.** Do this *after* its base image has been loaded with image data, or the alias may produce undefined results.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampLoadImage( pWMyimage, fname, opts,
                ctype,imgnum);
    ampCreateImageAlias ( pWMyimage, &pWAlias);
    ampSetImageMetrics( pWAlias, 512, 256,
                1024,1024);
```

ampFreeAllImages

```
int ampFreeAllImages()
```

This API removes all images and frees up all the workspace allocated for them.

This API always returns 0 if the function was successful, non-zero otherwise.

ampFreeImage

```
int ampFreeImage( pW )
```

```
    PWORKIMAGE pW;
```

This API removes the image *pW* from the list of known images. Any alias images derived from it will also be destroyed. Any storage allocated for the image will be freed. If the image *pW* does not exist, the function will do nothing and return a normal success code.

This API always returns 0 if the function was successful, non-zero otherwise.

ampGetGrayImageMetrics

```
int ampGetGrayImageMetrics( pW, Dx, Dy, X, Y,
            Pitch, Height, BitsPerPixel)
```

```
    PWORKIMAGE pW;
    LPLONG Dx, Dy;
    LPLONG X, Y;
    LPLONG Pitch, Height;
    LPLONG BitsPerPixel ;
```

This API returns the present values of the image metrics for image *pW*. NULL can be used for arguments that are not needed by your program. It can be used with bilevel, grayscale, or color work images.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampGetGrayImageMetrics (pWMain, &dx, &dy,
                &x, &y, NULL, NULL,
                &BitsPerPixel);
```

ampGetImageMetrics

```
int ampGetImageMetrics( pW, Dx, Dy, X, Y,
            Pitch, Height)
```

```
    PWORKIMAGE pW;
    LPLONG Dx, Dy;
    LPLONG X, Y;
    LPLONG Pitch, Height;
```

This API returns the present values of the image metrics for image *pW*. NULL can be used for arguments that are not needed by your program.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampGetImageMetrics (pWMain, &dx, &dy,
                &x, &y, NULL, NULL);
```

ampGetImageResolution

```
int ampGetImageResolution( pW, Xres, Yres)
```

```
    PWORKIMAGE pW;
    LPLONG Xres, Yres;
```

This API returns the present resolution values for binary, grayscale, or color image *pW*. These values are set when an image is loaded or scanned, or by **ampSetImageResolution.** When an image is processed from one image to another, resolution values are propagated to the new image. Images must have a valid resolution in order to be processed correctly by the ampReadMicr function. Check images loaded from the clipboard with the ampLoadClipboard function should be verified for valid resolution values.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampGetImageResolution(pWMain, &Xres, &Yres);
```

ampGetRunsInfo

```
int ampGetRunsInfo( pW, params )
```

```
PWORKIMAGE pW;
ampRUNSINFO *params;
```

This function analyzes the binary image pW and returns information that can be used to gauge the level of change that has occurred, or to perform blank page detection. The information is returned in the ampRUNSINFO block.

The fields of ampRUNSINFO are defined as follows (see interface reference file for latest definition).

```
typedef struct of_AMPRUNSINFO {
      int   MinWidth;
      int   MaxWidth;
      // results output
      long  TotalPixels ;
      long  TotalRuns ;
} ampRUNSINFO;
```

MinWidth is the minimum horizontal pixel width of black runs to use in measurements.

MaxWidth is the maximum pixel width of black runs to use in measurements.

TotalPixels is the total number of black pixels found in the image.

TotalRuns is the total number of black runs found in the image.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
ampGetRunsInfo(pWMain, &runsInfo);
```

ampSetImageMetrics

```
int ampSetImageMetrics( pW, dx, dy, x, y )
```

```
PWORKIMAGE pW;
long  dx, dy;
long  x, Y;
```

This API changes one or more of the image metric values for the binary, grayscale, or color image *pW*. Only the sub-image metrics are modified; the storage allocation for the image will **not** be changed. The values will be checked for arithmetic correctness, e.g.,

x + dx must be <= pitch

y + dy must be <= height

If the value ARGSKIP is used for any argument other than *pW* , the corresponding image metric will be unchanged.

This API always returns 0 if the function was successful, non-zero otherwise.

ampSetImageResolution

```
int ampSetImageResolution( pW, Xres, Yres )
```

```
    PWORKIMAGE pW;
    long  Xres;
    long  Yres;
```

This API sets the resolution information for the binary, grayscale, or color work image *pW*. This function can be used when AMPLIB cannot determine the resolution in some other way, such as a clipboard image, TIFF header, etc. The image resolution information is propagated from source image to destination image during AMPLIB execution. Images must have a valid resolution in order to processed correctly by the ampReadMicr function. Check images loaded from the clipboard with the ampLoadClipboard function should be verified for valid resolution values.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampSetImageResolution(pWApp, 200, 200);
```

MICR Functions

This group of functions provides for reading the MICR shapes from an image and placing the resulting ASCII codes into an appropriate data structure. There are 2 basic types of functions provided. MICR OCR and MICR Verify.

MICR OCR functions will read the MICR line data from an image. These functions use multiple OCR engines and voting technology to produce an accurate result. Produces best and secondary read choices with confidence levels. Application developers have options to invoke image repair functions. Additional image processing functions from the SDK can be used to preprocess the image before OCR recognition to correct for skew and image border affects.

The API also has a set of super functions for processing checks captured by non traditional check scanners (page scanners, low cost scanners, cameras). These functions use image preprocessing functions to crop, deskew, threshold, and correct for image distortions caused by the non traditional check image capture process. These functions also provide the ability to process the image of a mulipart remittance form(E.g. 8.5X11 page with a check attached top or bottom). These functions will locate the check, read the MICR line and extract the check image.

MICR Verify functions will read the MICR from an image but will use previously captured MICR data to assist in the process. There are different applications that can use MICR Verify technology so there are a family of functions each using a different set of inputs, outputs and processing rules. All MICR Verify functions use a fast OCR engine to verify the easy to read items that usually represent the majority of the cases. Image repair, additional OCR engines, and OCR retry operations are used on the harder to read items as required to verify the most difficult to read items. The combined result is higher throughput with higher accuracy.

Camera Feature License

The Camera license bit can be used with any ReadMicr function and it is not restricted to ampReadMicrCamera/ampReadCamera function calls. The Camera license enable character by character voting amongst three different AmpLib engines. One engine has been heavy trained using camera based images. The result is higher read rates and lower substitution rates on the most difficult images. The Camera license is required with the ampReadMicrCamera/ampReadCamera function calls. In their cases, the low level voting within ampReadMicr is combined with high level image processing on the gray/color images to improve read rates and lower substitution rates even more.

MICR Verify Functions

MICR Verify – Check

Can be used in capture systems to detect and correct for errors in hardware captured MICR data. Uses the hardware captured MICR data with the image to detect and correct for rejects, substitutions and missed data.

Refer to functions:

- ampVoteMicrRepair
- ampVoteMicrRetry

MICR Verify – IRD

Regulatory policy requires that the MICR line encoded on an IRD must match the MICR line of the original item. This function will input the MICR field data from a X9.37 Image Cash Letter file with the check image to produce an accurate MICR line that can be applied to the IRD. X9.37 MICR field data is missing dashes, special symbols, and often contains errors requiring a verify OCR process to produce accurate IRDs. This same function can be used to detect and correct for errors in X9.37 MICR field data.

Refer to functions:

- ampVoteIRDRepair
- ampVoteIRDRetry

MICR Verify – Error Detection and Usability Analysis

Will detect images whose source document was different than the item the associated captured MICR data came from. This function will input the image and the MICR field data associated to the image and return results that can be used to validate if the MICR and image came from the same source document. The function will return the parsed OCR field data read from the image with a match confidence level. Two different versions of this function are provided to accommodate MICR field data from a X9.37 file (RT, Aux on-us, On-us, Amount) or from a database/capture file (RT, Account Number, Serial Number, Amount). Verify accuracy is very high as these function will ignore dashes, special symbols and leading zeros in the OCR processing.

This function will also verify the image usability of the MICR codeline on the image by ensuring the MICR field data is both present and legible.

Refer to functions:

- ampFieldVerify
- ampFieldVerifyEx

MICR Parameters

The parameters are contained in the structure ampMICRINFO as shown below.

```
typedef struct of_ampMICRINFO {
            int  lastchar;
             char resultchar[100];
             char bestchar[100];
             int  percent[100];
             char infile[256];
             char outfile[256];
             char translater[256]
             int Dorepair;
             int Do180;
             int DoneRepair ;
             int Done180 ;
             int Filter ;
             int Resolution
             int Code ;
             int NoBlanks ;
             int UseTranslator ;
             int NoRules ;
             int Rules ;
             int MinCon ;
             int char_x[100];
             char route[20];
             char account[20];
             char check[20];
             char amount[20];
             char epc[20];
             int DoImageRepair;
             long MaxTime ;
             long StartTime;
             double Skew;

     } ampMICRINFO;
```

lastchar

The number of characters in the result array. It can also be considered a pointer to the new line character at the end of a sting.

resultchar

The resulting character string from the read. The valid length is determined by lastchar. The resulting character array will include "misread" characters as determined by the user supplied controls on minimum confidence and rules.

bestchar

The best character string from the read. This are the best characters found even though they may not pass the tests needed to be a resultchar.

percent

The array containing the confidence percentage associated with the best characters found.

infile

The fully delimited file name of the character shape definition file "storage.dat"

outfile

This file name is always set to a null string.

translator

This table translates the internal codes to those desired by the user. For example, the default translation table uses an '*' for a character misread. By loading this table (and setting the UseTranslator parameter) the user will have their own character presentation. The table contents will also vary depending on the code being translated, i.e. E13B will have fewer characters than CMC7.

Translation Table

Location	Meaning	MICR	Default
0	'0'	'0'	'0'
1	'1'	'1'	'1'
2	'2'	'2'	'2'
3	'3'	'3'	'3'
4	'4'	'4'	'4'
5	'5'	'5'	'5'
6	'6'	'6'	'6'
7	'7'	'7'	'7'
8	'8'	'8'	'8'
9	'9'	'9'	'9'
10	Routing	A	'A'
11	Amount	B	'B'
12	On Us	X	'C'
13	Dash	Δ	'D'
14	Blank	' '	' '
15	Misread	NA	'*'
16	Best is same as Selected	NA	'N'

Dorepair

When this parameter is set and if any read errors are detected in the result data, then a temporary copy of the input image is repaired based on the results of the first read. The repaired image is then used for a second MICR read. The two MICR read results are then voted upon and the result of the vote is reported. When doing image repair and reprocess, the execution times will be about twice as long for images which have read errors in the first pass.

Do180

If a MICR line cannot be detected on the bottom of the input image, then a temporary copy will be created and rotated 180 degrees before attempting to perform another MICR read. The read of the original line will be considered unsuccessful if it has less than 4 valid characters and any unreadable characters.

DoneRepair

This is an output parameter that indicates if image repair was performed to get the result.

Done180

This is an output parameter that indicates that the image was turned 180 to get the result. The result reflects what was found at the top of the input image.

Filter

This is an output parameter that reports the type of image and implies the filter that was used to repair the image when the DoneRepair parameter is set. The results are AmpNeutral, AmpLight and AmpDark. A Neutral image is repaired with a noise removal filter. A Light image is repaired with a morphological darkening filter. A Dark image is repaired with a morphological lightening filter.

Resolution

The image must have a good approximation to its actual resolution. AmpLib is not simply a MICR OCR engine but it is really a "check reader". The resolution information is needed to determine where the bottom approximately 5/8" of a check is located. Forcing resolution with this parameter is normally not an issue since the image file will generally contain this information. In some situations the resolution is not included in the file data and in those cases a non-zero value for the Resolution parameter will be used. NOTE: The input image (pW) will have the resolution set if the input value is >= 0.

Input values:

< 0	: Use image resolution
==0	: Try to estimate image resolution
> 0	: Use this parameter for resolution.

Code

The type of MICR code being read:

E13B (AmpE13B)

CMC7 (AmpCMC7) (NA)

E13B non US (AmpE13BNonUS). The default value of 0 is E13B.

NoBlanks

An input parameter set to a 1 when the output will have blanks removed.

UseTranslator

An input parameter set to a 1 when the translator table is to be used.

NoRules

An input parameter set to a 1 when the internal banking rules are to be ignored. This is normally used when the input image is only a portion of a MICR line.

Rules

The national banking association rules to apply. Currently this is only the ABA rule set from the US.

MinCon

An input parameter describing the Minimum Confidence value that should be used to accept or reject a character. The value range is between 0 and 99 but the only reasonable values are between 80 and 90. Setting the value too high will reject characters that are read correctly. Setting the value too low will cause the acceptance of characters which are misreads or substitution errors.

The user must decide the best parameter value based on testing with their data set and with their set of needs. In general, good images do not cause substitution errors. It is corrupted images that cause problems. In all cases, a substitution error rate over a large data set is still expected to be a fraction of one percent. The following information is based on testing a wide range of images with the toolkit.

A MinCon of 80 is recommended for doing verification. It will generate some substitution errors on corrupted images but since it is being compared to another result, this effect is minimized.

A MinCon of 85 is recommended for general usage. This will reduce the substitution rate and only nominally reduce the read rate.

A MinCon of 89 is recommend for the lowest substitution error rate without dramatically reducing the read rate.

char_x

The x pixel location (from left edge of image) for each result character. This is often used in voting between/amongst different OCR engines.

route

The Route/Transit field as a separate string.

account

The Account field as a separate string.

check

The Check number field as a separate string.

amount

The Amount field as a separate string.

epc

The External Processing Code (EPC) field as a separate sting.

DoImage Repair

Output a repaired image

MaxTime

Limit the time to process an image. Input of 0 for no limits. Limits are input in units of hundreds of a second. A value of 50 is one half a second.

StartTime

This must always be a 0 upon input. Upon return, the output value is the start time using the timeb structure value in hundreds of a second.

Skew

The slope(dy/dx) of the MICR text line. The origin of the image is the upper left corner so sign of the slope will be reversed from lower left origin.

How final character selection is determined

The MinCon value is the primary determining factor in "selecting" or not selecting a character when no other factor is involved, however, many times other factors are involved. These factors include user-controlled parameters such as "repair image voting" and "banking rules enforcement". In addition, internal limits are enforced. For example, a character, which is too dark relative to its neighbors, will often be treated as indeterminate. Another internal limit is "minimum distance" between two confidence values. The OCR engine has preset values, which indicate that two results are "too close to call" even though they may meet MinCon requirements.

The "Dorepair" parameter causes the creation of a second check image which will then be repaired. It is possible to have two different characters with high confidence values from the two images. The Dorepair voter will report an indeterminate in this case even though the MinCon values for both may be very high. (Note: The typical cause for such an event is very different character segmentation between the two images.)

The use of banking "Rules" may both select or deselect a character in a fashion that would be counter indicated by its confidence value. A character may be selected if it is the "best" character, but not at the minimum confidence level, when a banking rule says that the best should occur at this point. For example, in the ABA rules an amount field is 10 digits and 2 control codes. If a character for one of the control codes is in the proper position and it is the "best" but not selected, then the use of a rule will promote it to "select".

A banking rule may also deselect a character or even delete one that meets MinCon conditions. For example, in the ABA rules, there are no valid characters beyond the amount field. Stray characters at this point may be deleted from the data set when a valid amount field can be determined.

ampAnalyzeResolutionEx

```
int ampAnalyzeResolutionEx (pInfo,
          pnHorizResolution, pnVertResolution,
          pnRCCRating)
```

```
ampMICRINFO *pInfo;
int *pnHorizResolution;
int *pnVertResolution;
int *pnRCCRating;
```

The ampAnalyzeResolutionEx extracts information from the pInfo structure following a MICR read operation such as ampReadMicr. In addition to performing MICR recognition ampReadMicr analyzes the relative horizontal positions of characters that were read with high confidence and establishes the average horizontal resolution of the check image. It also uses the average height of these characters to determine the vertical resolution. Note that check images captured from handheld cameras my have different horizontal and vertical resolutions if the camera angle was not directly perpendicular to the surface of the check. In addition, the shapes and OCR scores of identical numbers are compared to see how similar the morphology is for each of the numbers from 0-9. This figure is called the RCCRating and is a measure of whether the check was likely scanned or computer generated. A value 90 percent or higher indicates the check image was probably computer generated. A value lower than 90 indicates the image originally came from a paper check.

Example:

```
nStat = ampAnalyzeResolutionEx(pInfo,
&nHorizResolution, &nVertResolution,
&nRCCRating);
```

ampFormatMICRFields

```
int ampFormatMICRFields(szInput, nFilter,
            szTranslator, szOutput, szAuxOnUs,
            szEPC, szRoute, szOnUs, szAmount)
```

```
char  *szInput;
int   nFilter;
char  *szTranslation;
char  *szOutput;
char  *szAuxOnUs;
char  *szEPC;
char  *szRoute;
char  *szOnUs;
char  *szAmount;
```

The ampFormatMICRFields API accepts as input raw MICR ASCII string data pointed to by szInput and then formats that data into several output strings. The input data is passed through the translation table szTranslator (if the pointer is non-null) in order to make the characters compatible with AmpLib. The formatting algorithm uses the OnUs, Route, and Amount symbols to parse through the input data marking the beginning and ending of fields. Parsing problems caused by OCR errors are counted and returned to the calling program in the return code. If nFilter is nonzero, then any characters that precede a legitimate AuxOnUs field will be erased prior to formatting. If there were no parsing problems, the API will return 0.

If non-null, szOutput points to the formatted 62 byte MICR data. The traditional field byte boundary assignments after formatting are:

0-16	AuxOnUs field - length: 17 bytes
18	EPC Code - length: 1 byte
19-29	Route field - length 11 bytes
30-49	OnUs field - length 20 bytes
50-61	Amount field - length 12 bytes

Field data that is not present in the original input will be filled with space characters. Each output field is copied to its appropriate string (szAuxOnUs, szEPC, szRoute, etc.) provided the pointer to that string is non-null. All output strings are passed through the translation table szTranslator (if the pointer is non-null) in order to provide the calling program compatible character data.

Example:

```
nStat = ampFormatMICRFields(&szInputMICR[0],
0, NULL, &szOutputMICR[0], NULL, NULL, NULL,
NULL, NULL);
```

ampFieldVerifyEx

```
int ampFieldVerifyEx (pW, pInfo, phInfo,
          prInfo,ConfidenceMin,
          ConfidenceResult,
        InputFieldPresent,engineLevel,
        allVerify )
```

```
PWORKIMAGE pW;
ampMICRINFO *pInfo;
ampMICRINFO *phInfo;
ampMICRINFO *prInfo;
ampFIELDCONFIDENCE ConfidenceMin;
ampFIELDCONFIDENCE *ConfidenceResult;
ampFIELDCONFIDENCE *InputFieldPresent;
int *engineLevel;
int *allVerify;
```

The ampFieldVerifyEx API will verify that input MICR field data matches the MICR codeline on the image. It ensures the usability of the MICR codeline on the image and will detect items whose captured MICR data and image data came from different source documents.

This function is called by “ampFieldVerify” that should support most users. If you need to use this function for special cases please contact AllMyPapers support for specific instructions and source code sample.

This function accepts as input the items image and the captured MICR field elements (Routing number, account number, check serial number, and check amount.). Returned will be the field data as read by the OCR process. A confidence factor for each field is also returned indicating the degree of match between the input and returned field data. Input minimum confidence threshold values are also input for each field. The function uses this confidence threshold to invoke “Try harder” processing to be able to return a result with the required confidence level.

Workimage “pW” will contain the source image. Currently only black and white images are supported. “PInfo” will contain the parameters to use for the OCR read process.

Structure “phInfo” is used to provide the input MICR field values. Structure variables “route”, “account”, “check”, and “amount” are used for this purpose. The field contents must only contain numeric characters (no dashes or special symbols). The account, check number and amount field strings can contain leading zeros or have them removed. Leading zeros will be ignored in the verification process except for routing numbers. The routing number only need to be 8 digits in length (no check digit) but will also accept the 9 digit format.

Structure “prInfo” returns the MICR field contents as read from the check image. In this case the entire field is returned with leading zeros. No special symbols or dashes are returned with the exception of the routing number where a dash will be returned for a “4-4” format. “prInfo->results” will return the entire codeline for the OCR read of the last OCR engine process used. This does not combine the results of the different OCR reads and should not be relied on as the best read.

Confidence Values

See “ampFieldVerify” for the table of field confidence values returned by this function.

Confidence Values

Confidence Score	Description
99	All characters match at above min confidence (“MinCon”) specified in “piInfo” structure.
98 to 94	All characters match with best choice. (99 minus number of best choice matches)
93-86	All characters match but some unreadable by OCR. (93 minus # unreadable characters)
85-81	characters match except for some missing on OCR Result. (85 minus number of missing characters)
80-71	Field Mismatch. OCR and reference do not agree. (81minus number of characters that mismatch in reference field,)
50	Field does not exist on codeline image. Indicates missing codeline data (E.g. Missing check number or account number)
40	Account reference field matches, but additional characters returned in account number result. (Indicates presence of transit field data, or short account number mapping)

Structure members of ampFIELDCONFIDENCE are:”route”, “account”, “check”, and “amount”.

Structure “ConfidenceMin” will allow users to specify their required minimum threshold per field. Valid settings for these fields are 0 or 81 to 99 as per the confidence table. Generally the lower the confidence the faster the throughput as the OCR verification process will terminate early once the minimum confidence threshold is achieved.

Structure “ConfidenceResult” will contain the returned confidence values for each field.

Structure “InputFieldPresent” is used to indicate if you there is reference field data provided for each field. For example you could set InputFieldPresent->amount = 0, if the application did not need to verify the amount field contents.

The OCR process can uses several OCR engine processes to verify the field with the desired confidence level. “engineLevel” will return from 1-4 indicating the number of OCR engine processes it used to achieve the result.

This function will also set “allVerify” if it successfully verifies all fields at the min confidence levels.

ampFormatMICRFields

```
int ampFormatMICRFields(szInput, nFilter,
            szTranslator, szOutput, szAuxOnUs,
            szEPC, szRoute, szOnUs, szAmount)
```

```
char  *szInput;
int   nFilter;
char  *szTranslation;
char  *szOutput;
char  *szAuxOnUs;
char  *szEPC;
char  *szRoute;
char  *szOnUs;
char  *szAmount;
```

The ampFormatMICRFields API accepts as input raw MICR ASCII string data pointed to by szInput and then formats that data into several output strings. The input data is passed through the translation table szTranslator (if the pointer is non-null) in order to make the characters compatible with AmpLib. The formatting algorithm uses the OnUs, Route, and Amount symbols to parse through the input data marking the beginning and ending of fields. Parsing problems caused by OCR errors are counted and returned to the calling program in the return code. If nFilter is nonzero, then any characters that precede a legitimate AuxOnUs field will be erased prior to formatting. If there were no parsing problems, the API will return 0.

If non-null, szOutput points to the formatted 62 byte MICR data. The traditional field byte boundary assignments after formatting are:

0-16	AuxOnUs field - length: 17 bytes
18	EPC Code - length: 1 byte
19-29	Route field - length 11 bytes
30-49	OnUs field - length 20 bytes
50-61	Amount field - length 12 bytes

Field data that is not present in the original input will be filled with space characters. Each output field is copied to its appropriate string (szAuxOnUs, szEPC, szRoute, etc.) provided the pointer to that string is non-null. All output strings are passed through the translation table szTranslator (if the pointer is non-null) in order to provide the calling program compatible character data.

Example:

```
nStat = ampFormatMICRFields(&szInputMICR[0],
0, NULL, &szOutputMICR[0], NULL, NULL, NULL,
NULL, NULL);
```

ampParseMicr

```
int ampParseMicr( pInfo, prInfo)
```

```
ampMICRINFO  *pInfo;
ampMICRINFO  *prInfo;
```

The ampParseMicr API accepts as input the MICR data from a previous ampReadMicr call or MICR data from another source.

The input MICR data is in the pInfo->resultchar data structure. If the check number is captured externally, it is entered in the pInfo->checknumber field. The parse uses general rules for field parsing. Inputting the check number when known may improve the results.

The MICR data must have a valid ABA Routing number for the parse to occur. If the data does not have a valid Routing number, it may be a deposit entry or check from another country and the parse cannot occur.

Example:

```
pInfo->resultchar =
"a121000301a1121d95593c0625c" ;
pInfo->checknumber = "1121" ;
ampParseMicr( pInfo, prInfo ) ;
```

The result Info block will have an account number of "95593c0625" and a check number of "1121". Special codes are generally not removed from the account number.

ampPrepMicr

```
int ampPrepMicr( pWs, pWd, pPrepInfo )
```

```
    PWORKIMAGE pWs;
    PWORKIMAGE pWd;
    ampPREPINFO *pPrepInfo;
```

This API copies one workimage to another and while doing the transfer performs image manipulations that prepare it for MICR reading. The first step is to deskew the image in forms mode and crop any white or black margin. The BlackEdges input variable must be set to activate black margin removal. On completion, the SkewDetected variable will be set the amount of skew AMPLIB saw in the image.

The process assumes that the result image should be "check" size and makes decision based on that.

When the Rotate parameter is -1 or +1 then if after black/white edge removal, the image has the short edge across the image, the image will be rotated 90 degrees left or right. If Rotate is set to 0, no rotation occurs and if set to +90 or -90 then right or left 90 degree rotation always occurs. This API always returns 0 if the function was successful, non-zero otherwise.

The ampPREPINFO structure is shown below.

```
    typedef struct of_ampPREPINFO {
          BOOL   BlackEdges;      // input 0,1
          double SkewDetected;    // output
          int    Rotate ;
                 //   0 --no rotation
                 //  -1 --auto detect rotate left
                 //  +1 --auto detect rotate right
                 // -90 -- rotate left always
                 // +90 -- rotate right always
          int    Resolution ;
                 //  0 -- use image resolution
                 // >0 -- use this as image
                 //  resolution
          } ampPREPINFO ;
```

A short example is shown below.

```
    // Prep the image for MICR reading which may
    // include deskew, scaling and rotation

    if (bMICRBlackEdge)
          piInfo.BlackEdges = 1;
    else
          piInfo.BlackEdges = 0;
    piInfor.Rotation = 0 ;
    piInfor.Resolution = 0 ;
    nStat = ampPrepMicr(pW, pWMicr,
    (PAMPPREPINFO) &piInfo);
```

ampPrepPage

```
int ampPrepPage( pWs, pWd, pPrepInfo )
```

```
    PWORKIMAGE pWs;
    PWORKIMAGE pWd;
    ampPREPINFO *pPrepInfo;
```

This API copies one workimage to another and while doing the transfer performs image manipulations that prepare it for page processing. The first step is to deskew the image in forms mode and crop any white or black margin. The BlackEdges input variable must be set to activate black margin removal. On completion, the SkewDetected variable will be set the amount of skew AMPLIB saw in the image.

The process is similar to ampPrepMicr but it does NOT assume that the result image should be "check" size.

When the Rotate parameter is -1 or +1 then if after black/white edge removal, the image has the short edge across the image, the image will be rotated 90 degrees left or right. If Rotate is set to 0, no rotation occurs and if set to +90 or -90 then right or left 90 degree rotation always occurs. This API always returns 0 if the function was successful, non-zero otherwise.

ampReadMicr

```
int ampReadMicr( pW, pInfo)
```

```
    PWORKIMAGE pW;
    ampMICRINFO *pInfo;
```

This API is called to locate MICR shapes on the check image, convert those shapes to corresponding ASCII codes, and then place those codes in the output data structure. The ampMICRINFO structure is used to initialize parameters for ampReadMICR as well as communicate output results back to the calling program. This API always returns 0 if the function was successful, non-zero otherwise.

In order to parse the MICR data into fields, the data line must have a valid ABA Routing number for the parse to occur. If the data does not have a valid Routing number, it may be a deposit entry or check from another country and the parse cannot occur.

The ampMICRINFO structure is shown below.

```
typedef struct of_ampMICRINFO {
            int  lastchar;
             char resultchar[100];
             char bestchar[100];
             int  percent[100];
             char infile[256];
             char outfile[256];
             char translater[256]
             int Dorepair;
             int Do180;
             int DoneRepair ;
             int Done180 ;
             int Filter ;
             int Resolution
             int Code ;
             int NoBlanks ;
             int UseTranslator ;
             int NoRules ;
             int Rules ;
             int MinCon ;
             int char_x[100];
             char route[20];
             char account[20];
             char check[20];
             char amount[20];
             char epc[20];
             int DoImageRepair ;
             long MaxTime ;
             long StartTime ;
     } ampMICRINFO;
```

Before calling ampReadMicr, the infile variable must be filled with the pathname of the file storage.dat that contains the definitions of the character shapes that AMPLIB can read. Setting Dorepair to the value 1 tells AMPLIB to use morphological filters to process the input image in an attempt to remove stray pixels and repair gaps in the MICR characters. Setting Do180 to 1 enables AMPLIB to read MICR characters that are upside down (rotated 180 degrees) if normal left-to-right reading failed. Setting Dorepair and Do180 to zero will disable these read options.

After calling ampReadMicr, the variable lastchar contains the number of MICR characters read. Those characters plus additional preceding spaces are located in the resultchar field. If a particular MICR shape was not read with sufficient accuracy, an '*' character is used to mark its position. The fields bestchar and percent contain information representing the confidence of the recognition process for a particular character.

If the Resolution parameter is zero, meaning that it is requesting a resolution estimate, it will return the estimate in the Resolution parameter.

A short example program is shown below.

```
strcpy (ampMI.infile, szAppDirectory);
strcat (ampMI.infile, "\\storage.dat");
strcpy (ampMI.outfile,"");
ampMI.Dorepair = 0;
ampMI.Do180 = 1;
 ampMI.Resolution = - 1 ;   // Use image resolution
  ampMI.Code = 0 ;          // Default to E13B
  ampMI.NoBlanks = 0 ;      // Report blanks
  ampMI.UseTranslator = 0 ;      // Use default character translation
  ampMI.NoRules = 0;        // Use ABA rules
  ampMI.Rules = 0 ;         // ABA rules
  ampMinCon = 80 ;                  // Minimum confidence
ampMI.DoImageRepair = 0 ;      // Do not output repaired image
ampMI.MaxTime = 0 ;  // No limit on process time
ampMI.StartTime = 0 ;  // Start must be zero
nStat = ampReadMicr (pW, (PAMPMICRINFO) &ampMI);

if (nStat == 0)
{
     i = ampMI.lastchar;
     ampMI.resultchar[i] = 0;
     strcpy (szResults,"Results: ");
     strcat (szResults, ampMI.resultchar);
     MessageBox(hWnd, szResults, "MICR Characters Read",
     MB_OK);
}
else
{
     wsprintf(szResults,"Error %d occurred during MICR
            processing.",nStat);
     MessageBox(hWnd, szResults, "MICR Read Error",
              MB_OK|MB_ICONEXCLAMATION);
}
```

ampReadMicrPage

```
int ampReadMicrPage( pW, pInfo, x,y,dx,dy)
```

```
PWORKIMAGE pW;
ampMICRINFO *pInfo;
int *x ;
int *y ;
int * dx ;
int * dy ;
```

This API is called to locate MICR shapes on a full page image, report the location and convert those shapes to corresponding ASCII codes, and then place those codes in the output data structure. If the resulting dy is zero, no MICR codeline was found. Note: only operates with a Code value for US version of E13B. Functionally it is the same as ampReadMicr (see above) except for the output of the location information.

Minimun Licenses Required: amplib, E13B, Image Repair

This function is used to locate a check by its MICR line on a full page remittance document. It can also be used to find multiple strips on checks returned with strips attached. In the lattes case the user would make repeated calls with each subsequent call using the previous result to offset the image location to search.

```
ampLoadImage( pWMyimage, fname, opts,
            ctype,imgnum);
ampCreateImageAlias ( pWMyimage, &pWAlias);

nStat = ampReadMicrPage (pWAlias, &ampMI,
&x,&y,&dx,&dy);

ampGetImageMetrics( pWAlias, &DXorg, &DYorg,
            &Xorg, &Yorg, NULL, NULL);

if ( dy > 0 )// read again to see if more
       ampSetImageMetrics( pWAlias, DXorg,
       DYorg - y -dy  ,
       Xorg , Yorg + y + dy );

       nStat = ampReadMicrPage  (pWAlias,
       &ampMI, &x,&y,&dx,&dy);
```

ampReadMicrDouble

```
int ampReadMicrDouble( pW,
            pInfoA,xA,yA,dxA,dyA, pInfoB,
            xB,yB,dxB,dyB)
```

```
PWORKIMAGE pW;
ampMICRINFO *pInfoA;
int *xA ;
int *yA ;
int * dxA ;
int * dyA ;
ampMICRINFO *pInfoB;
int *xB ;
int *yB ;
int * dxB ;
int * dyB ;
```

This API is called to read two MICR lines from a check image, report the locations and convert the code lines to corresponding ASCII codes, and then place those codes in the output data structure. If the resulting dy is zero, no MICR codeline was found. Note: only operates with a Code value for US version of E13B. Functionally it is similar to ampReadMicr (see above) except for the output of the location information.

Minimun Licenses Required: amplib, E13B, Image Repair

This function is used to locate two MICR line. The second code line may be caused by a correction strip or be the check image in an IRD. In the case where the first (bottom) code line has an EPC code of 4, the top code line is assumed to be the check image in an IRD. Special processing is invoked to read the check image MICR line.

```
ampLoadImage( pWMyimage, fname, opts,
            ctype,imgnum);
ampCreateImageAlias ( pWMyimage, &pWAlias);

nStat = ampReadMicrDouble (pWAlias,
&ampMIA, &xA,&yA,&dxA,&dyA,
&ampMIB, &xB,&yB,&dxB,&dyB);
```

ampReadMicrRepair

```
int ampReadMicrRepair( pWs, pWd, pInfo)
```

```
PWORKIMAGE pWs;
PWORKIMAGE pWd;
ampMICRINFO *pInfo;
```

This API performs exactly as the ampReadMicr but will create an output image if image processing was necessary to read the MICR line. In this case, the same image processing is performed on the full source image and the result is output in the destination image pWd. The destination will be a modified source image if Dorepair was set upon input and DoneRepair was set upon exit. The image will be rotated 180 if Done180 is set upon exit. The API will create the pWd image unless the result of the read is an error. The caller must manage the resulting image.

If the Resolution parameter is zero, meaning that it is requesting a resolution estimate, it will return the estimate in the Resolution parameter.

The modifications to the resulting image may not improve the result but in general the modifications will improve the result. Input images under 200 dpi are the least likely to be improved by the repair process. In these cases the AmpLib engine performs additional steps which may be inappropriate for the resulting image and hence are not included in the repair image process.

ampReadMicrCamera

```
int ampReadMicrCamera(pwFront, pMICRINFO *)
```

```
PWORKIMAGE pwFront;
ampMICRINFO *pMICRINFO
```

This function takes as input color, grayscale or binary images captured by mobile devices, page scanners, cameras or traditional check scanners. It will process these images and return:

- The MICR codeline extracted from the image

The ampReadCamera functions are composed of a series of calls to ampReadMicr, amprMicrPrep, ampFilterImage and ampDynamicThreshold. It is assumed that calls to ampReadCamera are used because the source image does not have the same quality as traditional check scanners. It will perform multiple steps, as needed, depending on results of the early steps. Some of the parameters used internally are different than when the image is assumed to be from a check scanner. For example, the Minimun Confidence value for ampReadMicr is higher than the standard default. This may result in multiple steps in order to get a higher quality image by use of different settings for filters and threshold. The result is higher read rates and lower substitution rates.

ampReadCamera

```
int ampReadCamera(pwFront, pwRear,
            szInputOptions, szMicrOutput,
            pwFrontTIFF, pwRearTIFF,
            pCameraResult*, pMICRINFO *)
```

```
    PWORKIMAGE pwFront;
    PWORKIMAGE pwRear;
    PSTR szInputOptions;
    PSTR szMicrOutput;
    PWORKIMAGE pwFrontTIFF;
    PWORKIMAGE pwRearTIFF;
    ampCameraRes *pCameraResult;
    ampMICRINFO *pMICRINFO
```

This function takes as input color or grayscale images captured by mobile devices, page scanners, cameras, and check scanners. It will process these images and return:

- The MICR codeline extracted from the front image
- A front and rear Black and White TIFF image compliant to image exchange regulatory requirements.

Note: this function operates on gray or color images. If the user wishes to use this process with a black and white image, it must first be loaded into a GrayWorkImage before the call to ampReadCamera.

The ampReadCamera function is composed of a series of calls to ampReadMicr, amprMicrPrep, ampFilterImage and ampDynamicThreshold. It is assumed that calls to ampReadCamera are used because the source image does not have the same quality as traditional check scanners. It will perform multiple steps, as needed, depending on results of the early steps. Some of the parameters used internally are different than when the image is from a check scanner. For example, the Minimun Confidence value for ampReadMicr is higher than the standard default. This may result in multiple steps in order to get a higher quality image by use of different settings for filters and threshold. The result will be higher read rates and lower substitution rates. In addition to reading the MICR line, fully qualified black and white images for the front and back are produced as needed for image exchange. If the goal is simply to read the MICR, the ampReadMicrCamera function will perform all of the same MICR read steps but it will not produce qualified images. The ampReadMicrCamera also uses the traditional parameter blocks of all ampReadMicr function calls.

ampReadCamera will even process multipart remittance payment documents (E.g 8.5X11 page with check top or bottom. In this case the MICR line will be found and read and the image of the check will be located and cropped from the page image.

Use ampReadCamera with ampReadCameraRear to create a two step process that will first process the front image and then later the rear image. ampReadCamera will extract the MICR and generate the front TIFF image. ampReadCameraRear will then generate the rear TIFF image as a second step.

The input images are provided in work images <pwFront> and <pwRear>. The output images are placed in <pwFrontTIFF> and <pwRearTIFF>. The <pwFrontTIFF> and <pwRearTIFF> can be NULL and no output images will be generated. The <pwRear> parameter can be NULL and no rear image processing will occur.

<szInputOptions> is a string containing the processing options as follows:

- “C” instructs the function to crop the check images from the captured image
- “T” instructs the function to test and correct for trapezoidal shape errors in the check images
- “M=nn” Sets the MICR OCR character confidence level to “nn”. “nn” is a value set from 01-99. Characters with a read confidence level below this value will be output as a reject symbol (“*”). The recommended value is “81”.
- “N=n” Set the detection logic to assume an iNternational format. For example, N=4 is for India.
- “R” instructs the function to rotate 180 and reread the check if a MICR codeline is not found on the first attempt. In addition, the image will be rotated 90 degrees based on the image dimensions. If the height is greater than the width, it will be rotated 90 degrees. The actual direction will be based on the little "l"(left) or little "r"(right).
- “B” instructs the function to leave blanks in data.
- “Q=#” Sets the Resolution detection method.
    - "#=0" detects the resolution
    - "#=nnn" uses resolution value of nnn.
    - "#=-n" uses resolution value in image header
- “I” instructs the function to apply image repair filters to output images if used during recognition
- "A" Image prep/cropping has been using a general edge detection process. The A or Alternate cropping method uses a Sobel filter to detect edges. For full page scanners, the traditional method is superior. For camera based images, the Alternate or Sobel edge is superior. For those rare cases where the images are mixed, see the U or United option.
- "U" United will use the traditional image prep function first and then follow up, if needed, with the "A" alternate or Sobel prep. This is advised for mixed images
- "P=#" will Preprocess andimage to lighten the contents based on the # value relation to average darkness. Internally the darkest images are low numbers and the lightest are high numbers. Since all images are processed as gray scale, the range of values is from 0 to 255. The preprocess is used in the Crop or Prep process. When the "P" is used alone, the default value of 192 is used and will cause darkening.
- "E=#" will Enhance an image to lighten the contents based on the # value relation to average darkness. Internally the darkest images are low numbers and the lightest are high numbers. Since all images are processed as gray scale, the range of values is from 0 to 255. The Enhance process is used after the Crop or Prep process and before MICR OCR. When the "E" is used alone, the default value of 40 is used and will cause lightening.

If the "A" value is used and no other parameter is included, it will have the following defaults.

    "C";"P=40","E=192","O=80","l"

The parameters "CAR" are actually the recommended settings for camera based input.

Use the “C” option if the input image has not been previously cropped and corrected. Use the “T” option for camera images or if the source is unknown. The “T” option is not required for page scanner images.

With a successful result the function will return the MICR codeline in <szMicrOutput> that was extracted from the front image. <pwFrontTIFF> and <pwRearTIFF> will contain the processed images. These will be cropped, corrected, scaled, and converted to Black and White TIFF compliant images. The <pCameraResult> structure will contain flags to indicate success of the operations:

The CameraResult flags are:

int MICROK	Indicates the MICR was read successfully
int MICRCONF	0-100 level indicating confidence of reading MICR line. Over 50 indicates a successful read of the MICR line routing number.
int IMAGESOK	Indicates the success of generating compliant TIFF images
int IMG2DARK	Indicates one or both of the images are too dark
int IMG2LIGHT	Indicates one or both of the images are too light
int IMGNOTSIZE	Indicates one or both of the images are the wrong size
int REARMISSING	No output image available. Indicates a rear image was not received or failed to process.
int FRONTMISSING	No output image available. Indicates a front image failed to process.
int ORIGINX	For remittance vouchers and checks captured on check scanners this will contain the upper left corner of the located check on the original front image.
int ORIGINY	Upper left Y coordinate of the located check
int CHECKWIDTH	Width of the located check
int CHECKHEIGHT	Height of the located check

This API always returns 0 if the function was successful, non-zero otherwise.

ampReadCameraRear

```
int ampReadCameraRear(pwFrontTIFF, pwRearGray,
            szInputOptions, pwRearTIFF,
            pCameraResult*)
```

```
PWORKIMAGE pwFrontTIFF;
PWORKIMAGE pwRearGray;
PSTR szInputOptions;
```

```
PWORKIMAGE pwRearTIFF;
ampCameraRes *pCameraResult;
```

This function is called after ampReadCamea to process the rear image as a second step process. Use ampReadCamera to capture the MICR line and generate the front TIFF image. The ampReadCameraRear function then takes as input the front TIFF image and rear grayscale or color image. Will threshold and scale the rear image to match the fornt image and perform IQA on both images.

The front TIFF input image is provided in work image <pwFrontTIFF>. The rear grayscale/color input image is provided in work image <pwFrontGray. The output rear TIFF image is placed in <pwRearTIFF>.

<szInputOptions> is a string containing the processing options as described in ampReadCamera. You can and should pass the same option string as used in ampReadCamera.

The <pCameraResult> structure will contain flags to indicate success of the operations. Pass the same <pCameraResult> structure that was composed by ampReadCamera to combine the flags produced by the two steps.

This API always returns 0 if the function was successful, non-zero otherwise.

ampReadScannerForChecks

```
int ampReadScannerForChecks(pwFront, pwRear,
            szInputOptions, szMicrOutput,
            pwFrontTIFF, pwRearTIFF,
            pCameraResult*, pMICRINFO *)
```

```
    PWORKIMAGE pwFront;
    PWORKIMAGE pwRear;
    PSTR szInputOptions;
    PSTR szMicrOutput;
    PWORKIMAGE pwFrontTIFF;
    PWORKIMAGE pwRearTIFF;
    ampCameraRes *pCameraResult;
    ampMICRINFO *pMICRINFO
```

This function takes as input black and white, color or grayscale images captured by page and check scanners. It will process these images and return:

- The MICR codeline extracted from the front image
- A front and rear Black and White TIFF image compliant to image exchange regulatory requirements.

The ampReadScannerForChecks function is composed of a series of calls to ampReadMicr, amprMicrPrep, ampFilterImage and ampDynamicThreshold. It will perform multiple steps, as needed, depending on results of the early steps. Some of the parameters used internally are different than when the image is from a check scanner. For example, the Minimun Confidence value for ampReadMicr is higher than the standard default. This may result in multiple steps in order to get a higher quality image by use of different settings for filters and threshold. The result will be higher read rates and lower substitution rates. In addition to reading the MICR line, fully qualified black and white images for the front and back are produced as needed for image exchange. If the goal is simply to read the MICR, the ampReadMicrScannerForChecks function will perform all of the same MICR read steps but it will not produce qualified images. The ampReadMicrScannerForChecks also uses the traditional parameter blocks of all ampReadMicr function calls.

ampReadScannerForChecks will also process multipart remittance payment documents (E.g 8.5X11 page with check top or bottom. In this case the MICR line will be found and read and the image of the check will be located and cropped from the page image.

Use ampReadScannerForChecks with ampReadCameraRear to create a two step process that will first process the front image and then later the rear image. ampReadCamera will extract the MICR and generate the front TIFF image. ampReadCameraRear will then generate the rear TIFF image as a second step.

The input images are provided in work images <pwFront> and <pwRear>. The output images are placed in <pwFrontTIFF> and <pwRearTIFF>. The <pwFrontTIFF> and <pwRearTIFF> can be NULL and no output images will be generated. The <pwRear> parameter can be NULL and no rear image processing will occur.

When there is no rear image input and there is a rear image output, a blank image of the same size as the front output image will be created.

<szInputOptions> is a string containing the processing options as follows:

- “C” instructs the function to crop the check images from the captured image
- “M=nn” Sets the MICR OCR character confidence level to “nn”. “nn” is a value set from 01-99. Characters with a read confidence level below this value will be output as a reject symbol (“*”). The recommended value is “85”.
- “R” instructs the function to rotate 180 and reread the check if a MICR codeline is not found on the first attempt.
- “B” instructs the function to leave blanks in data.
- “Q=#” Sets the Resolution detection method.
  - "#=0" detects the resolution
  - "#=nnn" uses resolution value of nnn.
  - "#=-n" uses resolution value in image header
- “I” instructs the function to apply image repair filters to output images if used during recognition

Use the “C” option if the input image has not been previously cropped and corrected.

With a successful result the function will return the MICR codeline in <szMicrOutput> that was extracted from the front image. <pwFrontTIFF> and <pwRearTIFF> will contain the processed images. These will be cropped, corrected, scaled, and converted to Black and White TIFF compliant images. The <pCameraResult> structure will contain flags to indicate success of the operations:

The CameraResult flags are:

int MICROK	Indicates the MICR was read successfully
int MICRCONF	0-100 level indicating confidence of reading MICR line. Over 50 indicates a successful read of the MICR line routing number.
int IMAGESOK	Indicates the success of generating compliant TIFF images
int IMG2DARK	Indicates one or both of the images are too dark
int IMG2LIGHT	Indicates one or both of the images are too light
int IMGNOTSIZE	Indicates one or both of the images are the wrong size
int REARMISSING	No output image available. Indicates a rear image was not received or failed to process.
int FRONTMISSING	No output image available. Indicates a front image failed to process.
int ORIGINX	For remittance vouchers and checks captured on check scanners this will contain the upper left corner of the located check on the original front image.

int ORIGINY	Upper left Y coordinate of the located check
int CHECKWIDTH	Width of the located check
int CHECKHEIGHT	Height of the located check

This API always returns 0 if the function was successful, non-zero otherwise.

ampReadMicrScannerForChecks

```
int ampReadMicrCamera(pwFront, pMICRINFO *)
```

```
PWORKIMAGE pwFront;
ampMICRINFO *pMICRINFO
```

This function takes as input color, grayscale or binary images captured by mobile devices, page scanners, cameras or traditional check scanners. It will process these images and return:

- The MICR codeline extracted from the image

The ampReadCamera functions are composed of a series of calls to ampReadMicr, amprMicrPrep, ampFilterImage and ampDynamicThreshold. It is assumed that calls to ampReadCamera are used because the source image does not have the same quality as traditional check scanners. It will perform multiple steps, as needed, depending on results of the early steps. Some of the parameters used internally are different than when the image is assumed to be from a check scanner. For example, the Minimun Confidence value for ampReadMicr is higher than the standard default. This may result in multiple steps in order to get a higher quality image by use of different settings for filters and threshold. The result is higher read rates and lower substitution rates.

ampVoteIRDRepair

```
int ampVoteIRDRepair( pWs, pWd, pInfo, phInfo,
            prInfo )
```

```
PWORKIMAGE pWs;
PWORKIMAGE pWd;
ampMICRINFO *pInfo;
ampMICRINFO *phInfo;
ampMICRINFO *prInfo;
```

This API will read a MICR line from the source image and compare it with the data provided in phInfor. The "h" implies the results of a hardware reader. The vote or correctly the results of the vote are placed in prInfo. As with ampMicrRepair, the pWd image will be modified based on the filtering needed to improve the accuracy of the read. The software read results will appear in pInfo as they do in ampReadMicr.

ampVoteIRDRepair does a different style of voting on the EPC and Amount fields in the MICR line than ampVoteMicrRepair. ampVoteMicrRepair requires an exact match in both the EPC and Amount fields in order for the voted upon results to indicate an overall match. ampVoteIRDRepair is not as strict and allows a match to occur even if the EPC field in the check image to be completely missing. Similarly, a match may be declared if the Amount field is missing from the check image but is present in the hardware data. In addition, ampVoteIRD repair does not try to exactly match dash characters, but will skip over dashes looking for correspondence on the more significant characters in a given field.

The settings for the software result in pInfo are assumed to be the same as the data input in phInfo. For example, if a custom translation table is used to output the software result, the same table must be used to describe the input of the hardware data. The result data in prInfo will always have blanks removed.

The data returned in prInfo->percent contains information on how the vote results were obtained for each character as opposed to a confidence percentage. The data represents the following:

0--indeterminate

1--hardware and software were the same, hardware selected

2--hardware and software best are same, hardware selected

3--hardware and software did NOT match, software selected

4--hardware and software best do NOT match, hardware selected

5--hardware misreads and software reads, software selected

6--hardware misread and no software selection, misread selected

The IRD voting process changes all percent 2 values to 1 values. Consequently, if there is a match, prInfo->percent will have all 1s.

ampVoteIRDRetry

```
int ampVoteIRDRetry( pWs, pWd, pInfo, phInfo,
            prInfo )
```

```
PWORKIMAGE pWs;
PWORKIMAGE pWd;
ampMICRINFO *pInfo;
ampMICRINFO *phInfo;
ampMICRINFO *prInfo;
```

This API has the same parameters as ampVoteIRDRepair and in fact performs many different ampVoteIRDRepair operations looking for a good match between the hardware data located in phInfor and the MICR line as read from the source image pWs. Both of AMPLIB's recognition engines are used in the voting process as well as the speckle, line, and border image filters. Many different combinations of the recognition engines and filters are used on the image before the API terminates. ampVoteIRDRetry will return as soon as it has found a match. Usually, this happens very rapidly, but the process may take a significant amount of time if no match is possible because the MICR line image is degraded or perhaps the hardware input was wrong,

ampVoteMicrRepair

```
int ampVoteMicrRepair( pWs, pWd, pInfo,
            phInfo, prInfo )
```

```
PWORKIMAGE pWs;
PWORKIMAGE pWd;
ampMICRINFO *pInfo;
ampMICRINFO *phInfo;
ampMICRINFO *prInfo;
```

This API will read a MICR line from the source image and compare it with the data provided in phInfor. The "h" implies the results of a hardware reader. The vote or correctly the results of the vote are placed in prInfo. As with ampMicrRepair, the pWd image will be modified based on the filtering needed to improve the accuracy of the read. The software read results will appear in pInfo as they do in ampReadMicr.

The settings for the software result in pInfo are assumed to be the same as the data input in phInfo. For example, if a custom translation table is used to output the software result, the same table must be used to describe the input of the hardware data. The result data in prInfo will always have blanks removed.

The data returned in prInfo->percent contains information on how the vote results were obtained for each character as opposed to a confidence percentage. The data represents the following:

0--indeterminate

1--hardware and software were the same, hardware selected

2--hardware and software best are same, hardware selected

3--hardware and software did NOT match, software selected

4--hardware and software best do NOT match, hardware selected

5--hardware misreads and software reads, software selected

6--hardware misread and no software selection, misread selected

If the user wishes to improve the read rate, data result 5 will be the most important. If the user wishes to reduce substitution errors in the hardware results, then the user should raise the value of Minimum Confidence and pay special attention to data result 4.

ampVoteMicrRetry

```
int ampVoteMicrRetry( pWs, pWd, pInfo, phInfo,
            prInfo )
```

```
PWORKIMAGE pWs;
PWORKIMAGE pWd;
ampMICRINFO *pInfo;
ampMICRINFO *phInfo;
ampMICRINFO *prInfo;
```

This API has the same parameters as ampVoteMicrRepair and in fact performs many different ampVoteMicrRepair operations looking for a good match between the hardware data located in phInfor and the MICR line as read from the source image pWs. Both of AMPLIB's recognition engines are used in the voting process as well as the speckle, line, and border image filters. Many different combinations of the recognition engines and filters are used on the image before the API terminates. ampVoteMicrRetry will return as soon as it has found a match. Usually, this happens very rapidly, but the process may take a significant amount of time if no match is possible because the MICR line image is degraded or perhaps the hardware input was in error.

The MICR retry voting process changes all percent 2 values to 1 values. Consequently, if there is a match, prInfo->percent will have all 1s.

Bar Code Reading Functions

This group of functions provides for reading of bar codes in a variety of symbologies. You must have the appropriate license bit set in your AmpLib license in order to use these functions.

Bar Code Creation Guidelines

Generally the people who create the software performing the barcode read do not get to input suggestions on the print process. If you can influence the printing process, here are some guidelines.

Print the bar codes big enough to be read.

The key parameter in bar code printing is the size of the "x-width" after the bar code has been scanned. This means the user must account for the printer resolution and the scanner resolution in determining the best print size to use. The recommended minimum x-width after scanning is 3 pixels. It is quite possible to have successful reads with an occasional x_width of less than 3 pixels but in general, more scanned pixels per symbol element is better.

Print the bar code with a fixed width.

Pad the data with leading zeros or blanks so that the length is always the same. This speeds up the search process and reduces the chances of unwanted objects being read as a bar code.

Use a checksum character if it is an option.

Most of the newer bar codes include a checksum in all cases. Older codes such as Code 39 have it as an option. It is strongly recommended that a check sum be used for all printing. This almost absolutely assures that a substitution error in the data will not occur.

Observe the bar code white space limits.

Bar code need a clear space on either end called the "white space". The amount of white space varies per symbology but a good rule is to leave 1/4 inch clear all around the bar code.

Bar Code Reading Overview

The AmpLib bar code reading process consists of two basic operations: searching for a bar code placed anywhere at any angle on a page and then decoding the barcode found. This overview will associate the parameters with the process and hopefully expedite the users understanding.

Factors That Effect Read Performance

On todays processors, a full page image will be searched and read in a fraction of a second (less than a tenth of a second) with reasonably well formed barcode images. However this speed can be affected negatively based on numerous parameters.

Factors That Will Cause Images to Read Faster

Using the input image to specify a Region Of Interest(ROI)

When the location of a barcode is generally known, e.g. lower right corner, setting the input image to that corner using the ImageAlias function will make the speed 4 times faster since only 1/4 of the image must be processed.

Factors That Will Cause Images to Read Slower

As might be expected, many things will make the speed run slower.

Reading an unknown number of barcodes per page
Reading multiple barcode types per single read
Reading multiple sizes and at multiple orientations
Reading low quality images with a low Quality setting
Reading with noise filters set
Reading with especially small bar height
Reading with few characters in a bar.

When any or all of these choices are made, each may cause another pass of the image adding fractions of a second. These small delays added over thousands of images, can become significant hindrances to throughput. Testing different parameters on different images is a critical step to maximizing recognition read rates for a production run.

Factors That Will Cause Confusing Results

When adding multiple image processing steps, the effect of the timeout value will make some symbols fail that previously had read. For example, some filters add an extra pass on the image in order to fill-in gaps and holes Conversely, when using a Quality level of 0 on some symbol types, the image will be processed with increased resolution through interpolation. This increased resolution will also take more time and may induce a timeout.

Barcode Size

The search for a possible barcode object or component is based on the size of the barcodes being read. Generally this is not known in advance and a common set of sizes is used as the default. In all but the rarest cases the default size set will be sufficient to find a bar code object. The parameters effecting search are the search order parameters of PrSmall, PrMedium and PrLarge and the actual size parameters of Height and Width. There is an additional implicit parameter, which is often the cause of a misread. This is the image resolution value set in the input file or image structure. Since the search is based on size in inches, the resolution value must be reasonably accurate. The ForceResolution parameter can be used when the image data does not contain accurate resolution information. Finally, the output parameter TestedBarComponents tells us if a barcode object has been found. A zero result means that the search for a barcode object failed and not the read decode process itself. The most common initial error is the failure to set up the barcode engine in a manner to assure that all possible barcode objects are found on a page.

Decode Parameters

Once barcode objects are found, the parameters for the decoding process must be set correctly. The most important parameter is the SymbologyMask, which identifies which type of barcode should be read. Many barcode symbologies can be "auto discriminated" i.e. there is enough information in the symbology to determine the type of barcode that they are. The symbologies can be auto discriminated are Code 3 of 9, CODABAR, Interleaved 2 of 5, Code 128 and Code 93. Additional parameters required in the decode process include Orientation, DoChecksum, Fixed, Quality, Filter and Length. They will be covered in the detail section of the parameters.

The decode parameters are often used beyond simple control of barcode reading. They can be used to prevent unwanted reads of barcode like objects. They can be used to discriminate between multiple barcodes on a page and to recover from errors in the image itself. A description of their usage in these functions is important but not obvious.

Unwanted Reads

There are cases where text characters in a particular font (e.g. Helvetica) will match a valid barcode. This will happen most often when the minimum character length of the barcode is small. The longer the minimum length of a barcode, the less likely this type of error will occur. The recommended Length parameter is 6 characters even if the data requires fewer. If the barcode is always the same length, the Fixed parameter will also help prevent inadvertent reading. Finally, if the barcode printing can be controlled , always print a checksum and set the DoChecksum parameter.

Multiple Barcodes per Page

The barcode read engine is designed to handle any number of barcodes that will fit on a page. Sometimes, the user only wishes to find a specific code. The proper combinations of Length and Fixed will often accommodate this.

Poor Image Quality

The barcode engine does a good job of dealing with poor image quality especially in the color/grayscale domain. Some problems can only be fixed by user settings. The Filter parameter is designed to deal with column drop out on fax machines and scanners. The Quality parameter identifies low quality images, which will receive additional passes to attempt to read the barcode. These additional passes will often slow down the speed at which the barcodes are read. If the Quality parameter is set to zero(0), the image itself will be scaled to a higher "pseudo" resolution. This often improves read rates on poor quality images but at the expense of speed (See "**Factors That Will Cause Confusing Results").**

The Partial parameter can be used when debugging a page with poor image quality. It will report "partial" reads. In general it is not a good idea to use this for production processing since the amount of data generated can be large and the processing time long.

General Management Parameters

The user can control the maximum number of barcodes to be read on a page and how much total data they will have using MaxBars and MaxAscii parameters. These parameters must account for Partial barcode read data as well. The Number parameter identifies the actual number of barcodes to read per page. Setting this to the actual number to read (if known) will potentially speed up the process.

Bar Code Parameters

The input parameters for bar code reading are contained in the structure ampBARARGS as shown below. The interface definition file should be used as final reference.

```
typedef struct of_ampbarargs {
     double Width;         // (inches)
     double Height;        // (inches)
     long   SymbologyMask; // Symbologies
     long   Orientation;   //
     long   Quality;       //     0--9 (low--high)
     long   DoChecksum;    // True/False
     long   Filter;        // size of 0,1,2,3
     long   PrLarge;       // priority for Large size
     long   PrMedium;      // priority for Medium size
     long   PrSmall;       // priority for Small size
     long   Number;        // Number of symbols to find
     long   Length;        // Min char length
     long   Fixed;         // fixed length codes
     long   Partial;       // Min chars in a partial read
     long   MaxBars ;      // Number of data sets to report
     long   MaxAscii ;     // Amount of character space
     long   ForceResolution ;
            // (0 Use image resolution as is)
            // (>0 Use this value as resolution)
            // (<0 Estimate resolution)
     long   StartTime ;    // enter with 0, reports internal start time
     long   MaxTime ;      // max time to run in hundreds of sec
     } ampBARARGS;
```

Parameters

SymbologyMask

This parameter specifies which bar code symbologies will be recognized. One or more symbology names may be specified with this call. The SymbologyMask parameter is formed as a binary bit mask, which selects one or more of the symbologies to recognize.

Note that some symbologies are subsets of others, and cannot be positively identified in the bar code data record if the superset is also selected. For example, if both EAN 13 and UPC 12 are selected, the result will be EAN 13 regardless. While common barcodes will auto-descriminate, many must be used alone without any other barcodes.   In the following list, codes identiefed as "standalone" can not be used in conjunction with other codes. In addition, the I25, A25 and 25 codes can not be used together only one of the three can be used at once. In the following table, they have a "25 limited" comment.

The allowable barcode symbologies are:

BC_3of9	Code 3 of 9	
BC_CODABAR	CODABAR	
BC_I2of5	Interleaved 2 of 5	25 limited
BC_A2of5	Airline 2 of 5	25 limited
BC_128	Code 128	
BC_UCC128	UCC Code 128	
BC_2of5	Code 2 of 5	25 limited
BC_93	Code 93	
BC_UPC_A	UPC-A	
BC_UPC_E	UPC-E	
BC_EAN_13	EAN-13	
BC_EAN_8	EAN-8	
BC_POSTNET	Postnet	Standalone
BC_PDF417	PDF-417 2D code	Standalone
BC_PATCH	Patch Code	Standalone
BC-PLANET	US Post Office code	Standalone
BC_39_NOSS	Code 3 of 9 without start/stop code	Standalone
BC_BCC32	Bar Code 32 (Pharmacy)	Standalone
BC_DMATRIX	Data Matrix	Standalone
BC_4STATE	4-State (US Intelligent Mail / One Code, UK Royal Post, Austrialian Post, Royal Dutch Post, Singapore Post)	Standalone
BC_39_EXT	Code 39 Extension	Standalone
BC_QR	Quick Response Code	Standalone

Example:

To select Code 39 and Code 25 at the same time:

```
pBarArg->SymbologyMask = BC_3of9 | BC_2of5 ;
```

Width

The *Width* argument specifies the width, in inches, of the physical size of the bar code. (Width is defined as the direction perpendicular to the vertical bars of the bar code.) The value does not have to be exact, but a close approximation can speed bar code reading. Only use this argument when the actual size of the bar code is known and consistent.

It can be used in conjunction with the Search Order Parameters to provide a variety of sizes to search for.

Height

The *Height* argument specifies the height, in inches, of the physical size of the bar code. The same rules apply to height as apply to width, above.

Orientation

Orientation specifies the general orientation for bar codes in the input image. Note that horizontal orientation means that the vertical bars of the bar code are perpendicular to the x-axis of the image. This is sometimes called the "picket fence" orientation. Orientation is given as:

0	Horizontal only
1	Vertical only
2	Horizontal and vertical
3	Horizontal with significant skew
4	Vertical with significant skew
5	Vertical and horizontal with significant skew

Choosing only the actual orientation found on an image can reduce execution times. The bar code reading function assumes a normal amount of skew that would be expected from most auto-feeders. "Normal" in this case can be as much as 10 to 15 degrees of skew. For fastest throughput, try using the non-skewed orientation modes first to determine the correct choice for your processing needs.

Quality

The *Quality* argument may be used to condition the bar code reading process to assume different quality in the images. The quality value ranges from 0 to 9, with 9 representing the best quality input images. With lower values, the bar code reading function will "try harder" to read the bar code; higher numbers will allow the reading process to give up sooner. Generally, use low numbers for poor quality images, and high numbers for high quality images. Some guidelines for quality values:

FAX input	1
First generation print	9
2nd generation copy	7
Microfilm scan	4

The *Quality* parameter controls how much effort is expended by setting a minimum confidence value for an "acceptable" read. Specifically the confidence must be 10 times the value of the Quality parameter. For example, if the Quality value is 4, then the confidence must be 40 or above or else NO Read will be reported. The minimum Quality parameter value used to be 1 but is now 0. Now very low confidence values can still be reported as a read. These low confidence values will likely have some miss-reads but it is up to the user to sort out the good from the bad and adjust the Quality as needed. A problem is that every engine has its own way of calculating the confidence value used in Quality. In addition, some engines have more internal redundancy than others. UPCA and EAN 13 have a parity code embedded without being requested. This allows more confidence in these results. In the case of UPCA and EAN 13, the confidence number number used in Quality is related to the number of rasters with the same result. However in this case, the rasters have an embedded parity and hence they have a higher relative confidence.

Many systems will know what the general result of a barcode should be. In those cases, very low Quality values can be used to get results that would be suspect using only information available to the bar code engine.

DoChecksum

Enables checksum verification of bar code data when such verification is optional for the symbology. When this option is selected, the AIM standard checksum algorithm is performed on the bar code data. If the checksum does not match the expected value, the bar code in question will **not** be reported as a successful read.

Filter

Selects a spot filter to be applied to the bar code before reading. A filter value of F=0 selects no filtering, F=1 selects a 1x1 spot filter, F=2 selects a 2x1 spot filter.

Example:

```
Filter = 2;
```

enables a 2x1 spot filter.

Search Order Parameters

PrSmall

PrMedium

PrLarge

The search order parameters allow the user to control, which of the predefined sizes should be searched, for first. With the current speed of the CPU, the order of processing is less important. The parameters also allow disabling searching for bar codes of a specific size. Finally when working with a single, custom size bar code, disabling the search for the 3 standard sizes will improve performance.

Note that if a non-zero value is given for the *Width* and *Height* arguments , that size of bar code will be searched for first, regardless of the priorities given by these parameters.

The three size classes are defined as follows: (*Sizes shown are nominal and vary somewhat depending upon the symbology being searched for.)*

Small refers to a 0.9 inch wide by 0.25-inch high bar code.

Medium refers to a 1.7 inch by 0.5-inch high bar code.

Large refers to a 4.0 inch by 0.9 inch high bar code.

Each argument should be given a unique priority code, from 1 (highest) to 3 (lowest), or 0 if that size class should not be searched for.

Example 1:

```
    PrSmall = 1 ;
    PrMedium = 2 ;
    PrLarge = 3 ;
```

This example sets the search order to *Small*, *Large*, *and Medium.*

Example 2:

```
    PrSmall = 0 ;
    PrMedium = 2 ;
    PrLarge = 1 ;
```

This example requests that the *Small* size not be searched for, and searches for *Large* and *Medium*, in that order.

Number

The **ampReadBarCodes** function will search for at most *Number* bar codes. The function will continue searching until *Number* bar codes have been found, or until it cannot find any more. A bar code must be valid (pass all required definitions and restrictions) before it will satisfy this *Number* count. A N*umber* value of zero will search for all possible bar codes

Length

The *Length* defines the minimum number of characters that must be read in a bar code for that bar code to be considered valid. If fewer than this number of characters is found in a bar code, the bar code will not be reported in the count of valid bar codes found. (See *PartialLength* description below.)

Fixed

Fixed is a flag that indicates whether the bar codes may have a variable length or not. If *Fixed* is set to 1, each bar code must have exactly *Length* characters to be valid. If *Fixed* is set to 0, bar codes may be variable length, but must still pass the minimum length requirement given by *Length*. Be careful when specifying fixed length reads. When checksum verification is enabled, the check digit(s) will not be counted as part of the fixed length. Likewise, when no checksum is performed, any check digits in the bar code data will be reported back as data; thus the specified length must include them. Fixed length only applies to the codes 39,128, I25 and Codabar.

Partial

The *Partial* value is used in special cases when you want to see partial reads. If this argument is non-zero, it defines the minimum number of characters that must be valid in order to report a partial read. Partial reads are not counted as part of the *Number* of valid bar codes and do not serve to satisfy the bar code search count. They are read into a separate list, which can be accessed via a special argument setting in **ampGetBarCodeData.** Bar codes read as partials do count against the total *MaxBars* parameter.

MaxBars

The *MaxBars* parameter defines the maximum space allocated in AmpLib for bar code results on a single page. At a minimum, it must be larger than *Number*. The recommended value is 100, which will report up to 100 bar codes on a single page.

MaxAscii

The *MaxAscii* parameter defines the maximum space allocated in AmpLib for the ASCII character data on a single page. At a minimum, it must be larger than *Number * Length*. The recommended value is 100, which will report up to 100 characters total for all bar codes on a single page. When using PDF 417 numbers as high as 20,000 are not unreasonable.

ForceResolution

The input image will usually have the resolution of the data embedded within it. In some instances, the image generation software fails to include this information or it cannot include this information. The *ForceResolution* parameter lets the user control the information. A value of 0 uses the image resolution as is, a value of >0 will be the "forced" resolution and a value <0 will have AmpLib estimate the resolution.

MaxTime

Limit the time to process an image. Input of 0 for no limits. Limits are input in units of hundreds of a second. A value of 50 is one half a second.

StartTime

This must always be a 0 upon input. Upon return, the output value is the start time using the timeb structure value in hundreds of a second.

ampGetBarCodeData

```
int ampGetBarCodeData( pBase, pPage, index,
            partial, text, data )
```

```
ampBAROUT *pBase;
ampBARPAGEINFOR *pPage ;
int index;
BOOL partial;
LPSTR text;
BARCODEINFO *data;
```

This function returns the specific information for each bar code on a page as read by **ampReadBarCodes**.

The data for all barcodes read is found in pBase which was set by the **ampReadBarCodes** call.

Index is given as a number from 1 to *n*, where *n* must not exceed the number of valid bar codes reported by **ampReadBarCodes** in the **BarsRead** parameter of pPage.

If *Partial* is TRUE, the bar code data to be returned should be taken from the list of partial reads as returned by ***ampReadBarCodes*** in the **PartialsRead** parameter of pPage. Normally you will set this argument to FALSE, to retrieve only the valid bar codes read. In some special cases, you may want to turn on partial reads, in which case you select those data records instead of the valid bar code records by setting this TRUE.

Text is a pointer to a character array that will be filled with the text of the bar code. The *Text* data space must be large enough to accommodate the largest bar code data set. If checksum verification has been enabled, the check digit will have been stripped from this text string. If checksum verification has **not** been enabled, this text may include checksum digits, which might need to be removed before an application uses the data. The *Text* data may contain any binary character depending upon what is in the barcode itself.

Data is a pointer to a BARCODEINFO block that will be filled with related information on the bar code that was read.

Return Values

The function returns a 0 if there are no errors in the call and it fills a BARCODEINFO block with information describing the bar code found.

The actual bar code character information will be returned via binary character array *Text*. This is NOT a standard null-terminated string. The user must allocate a buffer large enough to hold the expected string data. The fields of BARCODEINFO are defined as follows (see interface reference file for latest definition).

```
typedef struct of_barcodeinfo {
      int TextLen;      // length of text
      DWORD Symbology;  // code symbology
      BOOL  Reversed;   // TRUE if reversed
      BOOL  Rotated;    // TRUE if rotated
      int ConfValue;    // 0 to 100 %
      int Check1;       // check digit
      int Check2;       // check digit
      int dx;           // width of bar
      int dy;           // height of bar
      int x;            // x origin of bar
      int y;            // y origin of bar
} BARCODEINFO;
```

TextLen is the length of the character data for this bar code and does not include a terminal NULL character.

Symbology identifies the symbology of this bar code. It is given as a value identical to that defined in the **BarCodeTypes** .

Reversed gives the direction of reading used to read this code. 0 indicates forward reading, 1 indicates reverse reading. This value indicates an upside-down bar code (180-degree rotation).

Rotated indicates a 90 degree rotation from horizontal. The so-called ladder presentation.

ConfValue gives the confidence level of the read. This is a simple percentage (0 to 100) of the region containing the bar code which was used to recognize the code. Higher values indicate greater confidence, but many codes read successfully with low values.

Check1 gives the numeric checksum.

Check2 gives the numeric checksum (Some symbologies have two checksums).

dx, *dy, x, y* give the sub-image metrics of the region enclosing the bar code in the input image. This data can be used to identify the meaning of the bar code when multiple codes on a page are expected.

Example:

```
    results = GetBarCodeData( pBase, pPage,1, 0,
                       lpText, &data );
```

ampReadBarCodes

```
int ampReadBarCodes( pW, pArg, pOut, pAscii,
            pPage )
```

```
PWORKIMAGE pW ;
PampBARARGS pArg;
ampBAROUT *pOut ;
char *pAscii ;
PampBARPAGEINFO pPage;
```

This function will read bar codes from the image pW. The active sub-image of *pW* will be examined for bar codes as defined by the *pArg* parameters. Processing will be more efficient if the sub-image zones in on the bar codes, but you do not have to be precise. Keep in mind that many types of image data can look like bar codes during initial passes over the data, so the less extraneous data that must be examined, the faster the function will run.

Bar code data that is read from the image will be stored in the *pOut* and *pAscii* structures. Since this will contain the data for all bar codes on a page, subsequent calls to **ampGetBarCodeData** will extract the data one bar code at a time. The general information regarding the result data will be in *pPage*.

Return Values

If the function call is not in error, the function loads the *ampBAROUT* and *CHAR* with the data for all the barcodes on a page. The *ampBARPAGEINFO* contains values giving the results of the bar code read (See interface files for latest definition).

```
typedef struct of_ampbarpageinfo {
        long    BarsRead ;          // Number of valid bar codes
        long    PartialsRead ;      // Number of partial reads
        long    TestedBarComponents ;   // could be bar codes
} ampBARPAGEINFO;
```

BarsRead returns the number of valid bar codes read.

PartialsRead returns the number of partial bar codes read.

TestedBarComponents returns the number of objects on a page which were examined to see if they were a bar code.

You can read the specific bar code information for each bar code found using the **ampGetBarCodeData** function.

Example:

```
char text[40];
char allspace [200] ;
BARCODEINFO data;
ampBARARGS  Arg;
ampBARARGS *pArg = &Arg ;
ampBAROUT Out[100]  ;
ampBAROUT *pOut = &Out ;
ampBARPAGEINFO Page ;
ampBARPAGEINFO *pPage = &Page ;

pArg->SymbologyMask = BC_3of9 ;
pArg->Number = 3 ;
pArg->Fixed = 1 ;
pArg->Length = 8 ;
pArg->MaxAscii = 200 ;
pArg->MaxBars = 100 ;
pArg->Partial = 0 ;
pArg->Height = pArg->Width = 0.0 ;
pArg->Quality = 0 ;
pArg->Orientation = 0 ; // Horizontal
pArg->PrSmall = 1 ;
pArg->PrMedium = 2 ;
pArg->PrLarge = 3 ;
pArg->Filter = 0 ;
pArg->DoChecksum = 0 ;
pArg->ForceResolution = 0 ;
pArg->StartTime = 0 ;
pArg->MaxTime = 0 ;

// Read Page
rc = ampReadBarCodes(pW, pArg,
pOut,allspace, pPage );

if (rc == 0)
   for (i=1; i <= pPage->BarsRead; i++)
{          // Fetch data for each bar code
      rc = ampGetBarCodeData(pOut, pPage, i,
                FALSE, text, &data);
                // do something with data
}
```

The example reads up to 3 Code 39 bar codes from pW image, each bar code must be exactly 8 characters long, and no partials are allowed.. After the page is processed, the data for each bar code is fetched.

OCR Functions

This group of functions provides for reading the OCR shapes from an image and placing the resulting ASCII codes into an appropriate data structure.

OCR Parameters

The parameters are contained in the structure ampOCRINFO as shown below.

```
typedef struct of_ampOCRINFO {
            int  lastchar;
             char resultchar[256];
             char bestchar[256];
             int  percent[256];
             char translater[256]
             int Singleline;
             int PassCount
             int Dorepair;
             int Do180;
             int DoneRepair ;
             int Done180 ;
             int Filter ;
             int Resolution
             int Code ;
             int NoBlanks ;
             int UseTranslator ;
             int NoRules ;
             int Rules ;
             int MinCon ;
             int char_x[256];
             int char_y[256];
             int char_dx[256];
             int char_dy[256];
             char route[20];
             char account[20];
             char check[20];
             char amount[20];
             char epc[20];
             long MaxTime ;
             long StartTime;
             long Roi ;
             long RoiTopOffset ;
             long RoiLeftOffset ;
             Long RoiX ;
             Long RoiY ;
             Long RoiDX ;
             Long RoiDY ;

     } ampOCRINFO;
```

lastchar

The number of characters in the result array. It can also be considered a pointer to the new line character at the end of a sting.

resultchar

The resulting character string from the read. The valid length is determined by lastchar. The resulting character array will include "misread" characters as determined by the user supplied controls on minimum confidence and rules.

bestchar

The best character string from the read. This are the best characters found even though they may not pass the tests needed to be a resultchar.

percent

The array containing the confidence percentage associated with the best characters found.

translator

This table translates the internal codes to those desired by the user. For example, the default translation table uses an '*' for a character misread. By loading this table (and setting the UseTranslator parameter) the user will have their own character presentation.

The codes E13B and CMC7 use the lower 31 locations for their codes. OCR A and B use the traditional ASCII code locations.

The table contents will also vary depending on the code being translated, i.e. E13B will have fewer characters than CMC7.

Translation Table

Location	Meaning	MICR	Default
0	'0'	'0'	'0'
1	'1'	'1'	'1'
2	'2'	'2'	'2'
3	'3'	'3'	'3'
4	'4'	'4'	'4'
5	'5'	'5'	'5'
6	'6'	'6'	'6'
7	'7'	'7'	'7'
8	'8'	'8'	'8'
9	'9'	'9'	'9'
10	Routing	A	'A'
11	Amount	B	'B'
12	On Us	X	'C'
13	Dash	Δ	'D'
14	Blank	' '	' '
15	Misread	NA	'*'
16	Best is same as Selected	NA	'N'

Singleline

When Singleline is set to 1, only a single line of the image is processed. When set to 0, all of the lines in the input image are processed. The recommended setting is zero(0) when the caller externally zones the image to a single line.

PassCount

The PassCount value determines how many attempts should be made to OCR an image. The results of which are voted upon and then returned. Zero(0) is the same as one. and // 0- single pass/ n= number of passes to attempt

Dorepair

When this parameter is set and if any read errors are detected in the result data, then a temporary copy of the input image is repaired based on the results of the first read. The repaired image is then used for a second MICR read. The two MICR read results are then voted upon and the result of the vote is reported. When doing image repair and reprocess, the execution times will be about twice as long for images which have read errors in the first pass.

Do180

If a MICR line cannot be detected on the bottom of the input image, then a temporary copy will be created and rotated 180 degrees before attempting to perform another MICR read. The read of the original line will be considered unsuccessful if it has less than 4 valid characters and any unreadable characters.

DoneRepair

This is an output parameter that indicates if image repair was performed to get the result.

Done180

This is an output parameter that indicates that the image was turned 180 to get the result. The result reflects what was found at the top of the input image.

Filter

This is an output parameter that reports the type of image and implies the filter that was used to repair the image when the DoneRepair parameter is set. The results are AmpNeutral, AmpLight and AmpDark. A Neutral image is repaired with a noise removal filter. A Light image is repaired with a morphological darkening filter. A Dark image is repaired with a morphological lightening filter.

Resolution

The Resolution parameter must be a good approximation to its actual image resolution. When reading MICR, the OCR engine is really a "check reader". The resolution information is needed to determine where the bottom approximately 5/8" of a check is located. Forcing resolution with this parameter is normally not an issue since the image file will generally contain this information. In some situations the resolution is not included in the file data and in those cases a non-zero value for the Resolution parameter will be used. NOTE: The input image (pW) will have the resolution set if the input value is >= 0.

Input values:

< 0	: Use image resolution
==0	: Try to estimate image resolution based on image size
> 0	: Use this parameter for resolution.

Code

The type of MICR code being read. The default value of 0 is E13B.

E13B	1
CMC7	2
OCRA-Numeric	4
OCRA-Numeric+Special	8
OCRA-AlphaNumeric	12
OCRA-AlphaNumeric+Special	16
OCRA-EuroBanking	20
OCRB-Numeric	32
OCRB-Numeric+Special	64
OCRB-AlphaNumeric	96
OCRB-AlphaNumeric+Special	128
OCRB-EuroBanking	160

NoBlanks

An input parameter set to a 1 when the output will have blanks removed.

UseTranslator

An input parameter set to a 1 when the translator table is to be used.

NoRules

An input parameter set to a 1 when the internal banking rules are to be ignored. This is normally used when the input image is only a portion of a MICR line.

Rules

The national banking association rules to apply. Currently this is only the ABA rule set from the US.

MinCon

An input parameter describing the Minimum Confidence value that should be used to accept or reject a character. The value range is between 0 and 99 but the only reasonable values are between 80 and 90. Setting the value too high will reject characters that are read correctly. Setting the value too low will cause the acceptance of characters which are misreads or substitution errors.

The user must decide the best parameter value based on testing with their data set and with their set of needs. In general, good images do not cause substitution errors. It is corrupted images that cause problems. In all cases, a substitution error rate over a large data set is still expected to be a fraction of one percent. The following information is based on testing a wide range of images with the toolkit.

A MinCon of 80 is recommended for doing verification. It will generate some substitution errors on corrupted images but since it is being compared to another result, this effect is minimized.

A MinCon of 85 is recommended for general usage. This will reduce the substitution rate and only nominally reduce the read rate.

A MinCon of 89 is recommend for the lowest substitution error rate without dramatically reducing the read rate.

char_x

The x pixel location (from left edge of image) for each result character. This is often used in voting between/amongst different OCR engines.

char_y

The y pixel location (from left edge of image) for each result character. This is often used in voting between/amongst different OCR engines.

char_dx

The x width in pixels for each result character.

char_dy

The y height in pixels for each result character.

route

The Route/Transit field as a separate string.

account

The Account field as a separate string.

check

The Check number field as a separate string.

amount

The Amount field as a separate string.

epc

The External Processing Code (EPC) field as a separate string.

DoImage Repair

Output a repaired image

MaxTime

Limit the time to process an image. Input of 0 for no limits. Limits are input in units of hundreds of a second. A value of 50 is one half a second.

StartTime

This must always be a 0 upon input. Upon return, the output value is the start time using the timeb structure value in hundreds of a second.

Roi (Region of Interest)

Roi	non 0 to use Region of interest parameters
RoiTopOffset	non 0 -- reference top edge else bottom edge
RoiLeftOffset	non 0 -- reference left edge else right edge
RoiX	distance from Left/Right reference
RoiY	distance from Top/Bottom reference
RoiDX	distance along Left/Right reference
RoiDY	distance along Top/Bottom reference

A non zero value indicates that the Region of Interest(ROI) parameters should be used to define the region to be processed. The ROI parameters are offsets to the Active Sub Image defined by the input **workimage**. The ROI parameters themselves are defined as offsets to the Top or Bottom, Left or Right of the subimage. For example, a check image may be of differing height but the MICR line is always offset from the bottom edge. When the user wishes to process the image both in a 180 degree fashion, the active subimage is rotated 180 degrees and the RoiTopOffset and the RotLeftOffset values are temporarily complemented. The RoiX, RoiY, RoiDx and RoiDy parameters are now valid for the 180 degree case. To process a standard check MICR line would use a RoiY value of 0 and RoiTopOffset of 0. The ROI can also be used to process different strips on the bottom of a check by setting the RoiY value to the height of a strip.

It is sometimes desired to process the full width of an image or the full height without really knowing what those values are. One of the other of the RoiDX or RoiDY parameter can negative, in this case the input sub-image DX or DY is used as the respective ROI parameter.

ampReadOCR

```
int ampReadOCR( pW, pInfo)
```

```
PWORKIMAGE pW;
ampOCRINFO *pInfo;
```

This API is called to locate shapes on the image, convert those shapes to corresponding ASCII codes, and then place those codes in the output data structure. The ampOCRINFO structure is used to initialize parameters for ampReadOCR as well as communicate output results back to the calling program. This API always returns 0 if the function was successful, non-zero otherwise.

The ampReadOCR will perform the same functions as ampReadMICR but has been expanded to cover the standard OCR A and B fonts.

The ampOCRINFO structure is shown below.

```
typedef struct of_ampOCRINFO {
            int  lastchar;
             char resultchar[256];
             char bestchar[256];
             int  percent[256];
             char translater[256];
             int Singleline;
             int PassCount
             int Dorepair;
             int Do180;
             int DoneRepair ;
             int Done180 ;
             int Filter ;
             int Resolution
             int Code ;
             int NoBlanks ;
             int UseTranslator ;
             int NoRules ;
             int Rules ;
             int MinCon ;
             int char_x[256];
             int char_y[256];
             int char_dx[256];
             int char_dy[256];
             char route[20];
             char account[20];
             char check[20];
             char amount[20];
             char epc[20];
             long MaxTime ;
             long StartTime ;
             long Roi ;
             long RoiTopOffset ;
             long RoiLeftOffset ;
             Long RoiX ;
             Long RoiY ;
             Long RoiDX ;
             Long RoiDY ;
     } ampOCRINFO;
```

Setting Dorepair to the value 1 tells AMPLIB to use morphological filters to process the input image in an attempt to remove stray pixels and repair gaps in the characters. Setting Do180 to 1 enables AMPLIB to read characters that are upside down (rotated 180 degrees) if normal left-to-right reading failed. Setting Dorepair and Do180 to zero will disable these read options.

After calling ampReadOCR, the variable lastchar contains the number of characters read. Those characters plus additional preceding spaces are located in the resultchar field. If a particular shape was not read with sufficient accuracy, an '*' character is used to mark its position.

The fields bestchar and percent contain information representing the confidence of the recognition process for a particular character.

A short example program is shown below.

```
ampMI.PassCount = 1 ;
ampMI.SingleLine = 0 ;
ampMI.Dorepair = 0;
ampMI.Do180 = 1;
ampMI.Resolution = - 1 ;// Use image resolution
ampMI.Code = 16 ;   // OCRA alphanumeric+ special
ampMI.NoBlanks = 0 ;// Report blanks
ampMI.UseTranslator = 0;// Use default translation
ampMI.NoRules = 1;      // No  ABA Rules
ampMI.Rules = 0 ;       // ABA rules
ampMinCon = 80 ;            // Minimum confidence
ampMI.MaxTime = 0 ; // No limit on process time
ampMI.StartTime = 0 ;       // Start must be zero
ampMI.ROI = 0 ;     // NO Region of Interest
                    // Use the whole subimage

nStat = ampReadOCR (pW, (PAMPOCRINFO) &ampMI);
if (nStat == 0)
{
     i = ampMI.lastchar;
     ampMI.resultchar[i] = 0;
     strcpy (szResults,"Results: ");
     strcat (szResults, ampMI.resultchar);
     MessageBox(hWnd, szResults, "OCR Characters Read",
     MB_OK);
}
else
{
     wsprintf(szResults,"Error %d occured during
     OCRprocessing.",nStat);
     MessageBox(hWnd, szResults, "MICR Read Error",
     MB_OK|MB_ICONEXCLAMATION);
}
```

Coupon Functions

This group of functions provides for rapid detecting and reading the contents of Remitannce Coupons especially as found in Retail Lockbox processing. The two main functions are for Detection and for Reading. When processing a large number of intermixed coupons and checks, rapid detection of a coupon can greatly improve the throughput of the process. For example a coupone with multiple barcodes can be detected in a fraction of a second per core in use. Whereas at the same time, processing the associated check can take much longer.

ampDetectCoupon

```
int ampDetectCoupon(  pWfront, pWback,
            nForceResolution, nField Count,
            nDetected, nRotated180, nSide,
            nFieldType1, SymbologyMask1,
            CheckSum1, MinLength1, Maxlength1,
            nFieldType2, SymbologyMask2,
            CheckSum2, MinLength2, Maxlength2,
            nLengthField1, nLengthField2,
            cFieldResults1, cFieldResults2,
            dx1, dy1, x1, y2,
            dx1, dy1, x1, y2 )
```

```
ampDetectCoupon  (
PWORKIMAGE pWfront,
PWORKIMAGE pWback ,
int nForceResolution,
int nFieldsCount,
int *nDetected,
int *nRotated180,
int *nSide,

int  nFieldType1 ,
DWORD SymbologyMask1,
BOOL  Checksum1,
int   MinLength1,
int   MaxLength1,

int   nFieldType2 ,
DWORD SymbologyMask2,
BOOL  Checksum2,
int   MinLength2,
int   MaxLength2,

int *nLengthField1,
int *nLengthField2,
char cFieldResults1[200],
char cFieldResults2[200],
int *dx1,   int *dy1, int *x1, int *y1,
int *dx2,   int *dy2, int *x2, int *y2
```

)

The ampDetectCoupon function will examine the Front image and alternatively the Back image for the conditions that define a Coupon. It will then report what, in anything, was found. In many cases, the detection function is sufficient to provide all the needed information about the coupon and no additional read is necessary.

The ampDetectCoupon function will examine one or two fields for the conditions tha define a Coupon. Those conditions are the presence of the specific symbology (barcode or OCR), the specific length of the result and additionally in future releases the specific location and content.

The number of fields that will be used to detect a coupon is one or two and this is input by the nFieldCount parameter.

The resolution of the image is usually contained in the image structure but if this is wrong for any reason, the nForceResolution value can be used to input the current information.
WARNING: The speed and quality of the coupon detection is often directly related to how accurate the resolution is in the image. It is not uncommon for scanner control software to place a 96 in the dpi setting when the actual resolution is unknown. This can effect the quality of OCR and some barcode reading such as US Postal Codes which are defined to be a specific physical size.

A typical condition for defining a coupon is the presence of the USPS One Code. Most (but not all) addresses on retail coupons will have a USPS One Code. By the same token, most checks will not have such a code. Hence a coupon detector can have the following settings:

```
nFieldType1 =  1 ;            // barcode
SymbologyMask1 =  BC_4STATEUSPS ;    // usps one code only
MinLength1 = 40 ;
MaxLength1 = 80 ;
```

The ampDetectCoupon can be used to detect and read coupons when the fields are general purpose barcodes. For example, assume there are two barcodes, one has the amount due and is 6 digits and the other is the account and it is 8 digits. The following settings will detect a two barcode image as a coupon:

```
nFieldType1 =  1 ;            // barcode
SymbologyMask1 =  BC_3of9;       // Code 39
MinLength1 = 6 ;
MaxLength1 = 6 ;

nFieldType1 = 1 ;             // barcode
SymbologyMask1 =  BC_3of9;       // Code 39
MinLength1 = 8 ;
MaxLength1 = 8 ;
```

In this case the result fields for length, value and location (if output parameters present) will be set. A single call will provide the detection and the values needed for processing this coupon.

Another common detect and read case is for OCR characters in the bottom or top clear band on the coupon. The amount of data in a field tend to be very long in this case. Again, the length of the result will determine if it is the Coupon. An example of this is:

```
nFieldType1 = 2 ;            // OCR
SymbologyMask1 = ampOCRANUM         ; //OCR A numeric
only
MinLength1 = 50 ;
MaxLength1 = 80 ;
```

In addition to reporting the value of the field, important additional information is present.

nRotated180	if true implies that the image was upside-down . This is important to correct the image for future usage. This information could also be used to speed up check processing because it too is likely upside down.
nSide	if true, the field was found on the second image and future processing needs to account for this as well.

Future Efforts

The ampDetectCoupon function will have additional parameters and conditions in the future:

Checksum Type	There are many different checksums calculations.

Extraction Technology Functions

In general, AMPLIB functions will find and read the contents of a class of images with examples being barcodes, MICR characters and OCR fonts. The Extraction Technology Functions will find(extract) images based on the edges of the image within a frame. This may be a page of paper, a check, a coupon or just about any rectilinear object shape within an image frame. The existing AMPLIB functions all do this but most of it is embedded within a special class. The Extraction Technology Functions generalize this capability.

The Extraction Functions are heavily based on the use of Sobel filters, that is heavily enhanced Sobel filters. A web search will provide lots of basic information on the general Sobel filter. Using a Sobel filter on an image can enhance the appearance of edges to improve their detection and improve the performance as well. It is also true that not all images will have an edge and the software must adjust accordingly.

Once the image is determined, another critical component is the resolution of the image. If the input image came from a standard scanner, then this information is known. However often the interface between the scanner and the host does not convey this information and it has to be auto determined. This is especially true on any camera input such as a web cam, cell phone or WIFI connected camera.

ampSobelEdgePrepEx

```
int ampSobelEdgePrepEx(PWORKIMAGE
            pWsIn,PWORKIMAGE pWd,
            ampPREPINFO *pinfo,int
            nImageTypeIn, PAMPQUAD
            virtualCoord,
            int minDx, int minDy,
            int nMinGlyphs, int nTypeGlyphs,
            int *nFoundGlyphs,
            int *nGlyphX, int *nGlyphY,int
            *nGlyphDX, int *nGlyphDY,
            int *nStrengthLeftOut, int
            *nStrengthRightOut, int
            *nStrengthTopOut, int
            *nStrengthBottomOut   )
```

```
ampSobelEdgePrepEx(
PWORKIMAGE pWsIn,
PWORKIMAGE pWd,
ampPREPINFO *pinfo,
int nImageTypeIn,
PAMPQUAD  virtualCoord,
int minDx,
int minDy,
int nMinGlyphs,
int nTypeGlyphs,
int *nFoundGlyphs,
int *nGlyphX,
int *nGlyphY,
int *nGlyphDX,
int *nGlyphDY,
int *nStrengthLeftOut,
int *nStrengthRightOut,
int *nStrengthTopOut,
int *nStrengthBottomOut )
```

PWORKIMAGE pWd

The destination image will be the cropped result. The resolution of the image will be set to the found resolution (in x and y) if resolution detection was requested and was found.

In the case of a coupon and when resolution was detected, the image will be scaled so that there is only a single resolution value for both x and y.

The location of the result in the original image is reported in the virtualCoord values. The location of the gyphs will be reported relative to the result image.

nImageTypeIn

= 0	page
= 1	check
= 5	coupon
= 6	coupon A (no glyph detection)
= 7	back side(no glyph detection)

Virtual Coord

The Virtual coord are the locations of the sub image in the original input image with Corners of Upper Left, Lower Left, Upper Right and Lower Right (x,y in all cases).

minDx,minDy

The size of the back side of a check or coupon should be the same as the front side. These parameters provide that information. In the future this could also be used to input expected size of a front size in a normalized fashion. For example it could describe the size expected if at 200 dpi. This would then be adjusted based on the actual dpi found.

minGlyphs

If this value is zero, a default is used based on the ImageType. For example, the default length of glyphs for a check is 10. There are numerous cases where this value should be modified. If a MICR font is used on a coupon but it only has a routing number, then the minGlyphs should be one less than the number of full size MICR fonts. Special characters should not be considered in the minGlyphs in this case. Hence the minGlyphs for a coupon with only a MICR routing number should be 8.

In the case of coupons, the typical glyph being found is an OCRA or OCRB font. In this case a minGlyph count should be at least half of the length of gyphs expected. By having a larger value for the min, random strings of characters on the image are less likely to have an effect or be detected as the glyph of interest.

nTypeGlyphs

TBD

When implemented, this will allow the caller to identify the glyph and hence the GPI. For example:

Glyph	GPI
MICR	8
OCRA/B	10(typical)
Postal Barcode	22
General Barcode	random

***nFoundGlyphs**

The number of glyphs found on the image.

***nGlyphX, *nGlyphY, *nGlyphDX, *nGlyphDY**

The location and dimensions of the located glyph. This can be used to confirm that the resolution value is valid base on the location of the glyph on the image. For example, a glyph on a check at the top probably means it is upside down.

The DY value will be the height of the glyph region which means it will be taller because of any unadjusted skew, which should be minimal. A DY much taller than the expected glyph height usually means that multi lines of gyphs were found and the resulting resolution will be decreased.

***nStrengthLeftOut, *nStrengthRightOut, *nStrengthTopOut, *nStrengthBottomOut**

The strength of the edge detection is reported as values from 0 to 4. When most of the edges are a zero, it means that the object is detected by its simple extent and not an edge. This means that skew correction probably did not occur.

File and Image Transfer Functions

This group of functions provides for analyzing, loading and saving images. Images may be loaded from disk, DIBs, or the Windows clipboard. Images may be saved to disk or used to create a DIB.

Functions for loading (with decompression) and saving (with compression) image files have some common features, described here.

The file organization type, given as *ftype* in the API calls, specifies one of several possible file formats, such as TIFF, PCX, etc. The file format defines how the image data is structured within the file and how it is to be accessed, but not necessarily the data compression type.

Certain file formats uniquely identify a compression type, and others support a choice of different compression types. The compression code will be specified using a *ctype* argument in the API calls.

An *ftype* argument may be augmented by a compression type when you need to explicitly define the compression code.

*[**Note on bit ordering**: AMPLIB software follows the CCITT fax convention for bit ordering, so the default bit order is LSB first. This corresponds to TIFF Fill Order 2. If you need Fill Order 1, select the Bit Byte Reversal option in the appropriate file function.]*

The following file organizations (*ftype*) are supported by AMPLIB:

TIFF	TIFF file, includes ViewStar TIFF format
PCX	PCX
DCX	Multi-page PCX
PDF	Adobe-compatible PDF(output only)
NOHEADER	Data only, no header
BMP	Bitmap
JPG	JPEG

The following compression types (*ctype*) are supported by AMPLIB:

G4	CCITT Group 4
G32	CCITT Group 3 2d
G31	CCITT Group 3 1d
TF2	Modified 1-dim. (TIFF Type 2)
NONE	Uncompressed image data

As indicated above, you may sometimes augment a *ftype* argument with a compression type, as might be needed for creating a specific kind of TIFF file. The format for doing so is ***ctype/ftype***.

For example, saving a file with a ftype of "**TIFF**" would create a TIFF file, with the compression type defaulting to CCITT Group 4. To create a Group 3 TIFF file, you would give a *ftype* of "**G31/TIFF**".

ampAnalyzeTagBuffer

```
int ampAnalyzeTagBuffer( pW, nImgnum,
            pbyBuffer, nBufferSize)
```

```
    PWORKIMAGE pW;
    int nImgnum;
    PBYTE pbyBuffer;
    int nBufferSize;
```

This function accepts a TIFF file located in memory and performs an analysis function to determine if the header tags conform to a UCD/187 minimally compliant file. These minimums are: G4 compression, resolution of 200 or 240 dpi, Intel byte order, and width tag presence. Files will not decompress if the width tag is missing.

This function requires the presence of the Amplib feature license bit.

pW is the standard Amplib workimage structure which contains the Workfile substructure used to access and advance through the TIFF tags.

nImgnum is the image number and should always be set to 1.

pbyBuffer is the starting memory location of the buffer that contains the TIFF image.

nBufferSize is the length of the TIFF image data in the memory buffer.

This API always returns 0 if the function was successful, non-zero otherwise. A non-zero result implies non-conformance. This is not intended to be an interpreter of failure, but just reporting failure.

ampCreateDIB

```
int ampCreateDIB( pW, lphDIBS, dwDir)
```

```
    PWORKIMAGE pW;
    LPHANDLE lphDIB;
    DWORD dwDir;
```

Creates a bilevel DIB in memory from image data in work image *pW*. This function returns a handle back to the calling routine. That routine must do a GlobalLock on the handle in order to obtain a pointer to the DIB memory block. It is up to the calling routine to eventually release the memory used in the DIB back to Windows by doing a DeleteObject (hDIB).

dwDir is used to control the orientation of the DIB. dwDir = 0 selects the normal bottom-up DIB. *dwDir* != 0 selects a top-down DIB orientation.

This API always returns 0 if the function was successful, non-zero otherwise.

ampCreateDIBSection

```
int ampCreateDIBSection( pW, lphDIBSection,
            dwDir)
```

```
PWORKIMAGE pW;
LPHANDLE lphDIBSection;
DWORD dwDir;
```

Creates a bilevel DIB in memory from image data in work image *pW*. This function returns a handle back to the calling routine. That routine must do a GlobalLock on the handle in order to obtain a pointer to the DIB memory block. It is up to the calling routine to eventually release the memory used in the DIB back to Windows by doing a DeleteObject (hDIBSection).

dwDir is used to control the orientation of the DIB. dwDir = 0 selects the normal bottom-up DIB. *dwDir* != 0 selects a top-down DIB orientation.

This API always returns 0 if the function was successful, non-zero otherwise.

ampLoadClipboard

```
int ampLoadClipboard ( pW, hWndMain )
```

```
    PWORKIMAGE pW;
    HWND hWndMain;
```

This function will load a work image *pW* from the contents of the Windows clipboard provided that the clipboard can be obtained in a DIB format.

This API always returns 0 if the function was successful, non-zero otherwise.

ampLoadDIB

```
int ampLoadDIB ( pW, lpbmi, lpbits, style,
              threshold )
```

```
    PWORKIMAGE pW;
    LPBITMAPINFO lpbmi;
    LPBYTE lpbits;
    long style;
    long threshold;
```

This function will load a work image *pW* from a Windows DIB (bitmap). The DIB must have been loaded into memory already. Both *style* and *threshold* should be set to 0.

Bitmap (.BMP) files may be loaded directly into AMPLIB via the **ampLoadImage** function.

This API always returns 0 if the function was successful, non-zero otherwise.

ampLoadDIBHandle

```
int ampLoadDIBHandle ( pW, hDIB, style,
            threshold )
```

```
PWORKIMAGE pW;
HANDLE hDIB;
long style;
long threshold;
```

This function will load a work image *pW* from a Windows DIB using the DIB handle. The DIB must have been loaded into memory already. Both *style* and *threshold* should be set to 0.

Bitmap (.BMP) files may be loaded directly into AMPLIB via the **ampLoadImage** function.

This API always returns 0 if the function was successful, non-zero otherwise.

ampLoadDIBSectionHandle

```
int ampLoadDIBSectionHandle ( pW, hDIBSection,
            style, threshold )
```

```
    PWORKIMAGE pW;
    HANDLE hDIBection;
    long style;
    long threshold;
```

This function is similar to ampLoadDIBHandle, but takes a handle to a DIB section instead of a handle to a DIB.

This API always returns 0 if the function was successful, non-zero otherwise.

ampLoadImage

```
int ampLoadImage ( pW, filepathname, options,
            ftype, imgnum )
```

```
PWORKIMAGE pW;
LPSTR filepathname;
LPSTR options;
LPSTR ftype;
int imgnum;
```

This function will read an image file into image *pW*. The file will be decompressed as needed. If the *ftype* argument is NULL, the file will be analyzed to determine its type.

Filepathname gives the pathname of the file to load. If the drive and path are not given, the current drive and directory will be used.

Some common file name extensions are recognized:

TIF	File is a TIFF file
OPT	File is Optika format
PCX	File is a Paintbrush PCX file
DCX	File is a multi-image PCX
IMG	File is assumed to ViewStar TIFF
BMP	File is a Windows bitmap file
JPG	File is a JPEG file

All of the G3 and G4 bilevel variations of the TIFF standard are supported. Uncompressed grayscale and color TIFF files are also supported. Loading a grayscale or color TIFF file into a bilevel workimage will cause the image to be thresholded into black and white. Loading a color image into a grayscale workimage will cause the red, green, and blue components of each pixel to be converted to gray. The packbits, LZW, and JPEG sub formats of TIFF are **not** supported in the current release.

Options is a string of characters that describe optional processing for the image data during load:

B	Bit-byte-reverses the image data stream. (See File and Transfer Function Introduction).
T=-1	Drop out read
T=-2	Drop out green
T=-3	Drop out blue

The T parameters (-1,-2, and -3) allow colored components to be suppressed when loading color TIFF and JPEG files.

Ftype identifies the file organization, as described at the beginning of this manual section. It is only needed when the type cannot be inferred from the file header or filename extension.

Imgnum identifies which image in a multi-image TIFF file to load, from 1 to n.

When loading files with header blocks (such as TIFF), using a variable size image will be most convenient because the image size will be adjusted automatically for you. When loading images with no header information (and thus no size information) you should either:

- set the image metrics for image *pW* to the correct size for the image, or
- set the input image size metrics with **ampSetInputImageMetrics**

This API always returns 0 if the function was successful, non-zero otherwise.

Examples:

```
ampLoadImage( pWMine, "002343.TIF", NULL,
            NULL, 1);
ampLoadImage( pWYours,"XYZ001.DAT", "R",
            "G31/NOHEADER", 0);
ampLoadImage( pWTemp, "J7201.002", "B",
            "RLC", 0);
ampLoadImage( pWTemp, "J7201.jpg", "t=-1",
            NULL, 0);
```

As a more advanced example, consider the following code fragment which sequentially processes all the images in a multipage TIFF file by using the *Imgnum* parameter. The routine nMICROutput is not shown, but simply appends the passed text in szText to a binary file that has already been opened with the file handle nFhndle. nMICRFCount is a global variable used to count the number of files processed and nMICRICount is another global used to count the number of images found in those files.

Warning: There are many different sub formats within the various file types listed and not all will load.

```
   int nStat,  nPage;
   int nCount;
   PWORKIMAGE pW;
   PWORKIMAGE pWMicr;
   ampMICRINFO ampMI;
   ampPREPINFO piInfo;
   HANDLE hFind;

   // Initialize workimage pointers
   pWMicr = NULL;
   pW = NULL;
   nPage = 1;
   nStat = 0;
   while ((nPage != 0) && (nStat == 0))
   {
          nStat = ampCreateWorkImage(&pW, 0, 0);
          if (nStat != 0)
          {
                 // Check if there was an amplib problem
                 wsprintf(szText,"Error %d occured while creating image structure.\r\n",
                            nStat);
                 nMICROutput(nFhndle, &szText[0]);
                 return (0);
          }
          nStat = ampLoadImage(pW, szFile, "G=1", NULL, nPage);
          if (nStat != 0)
          {
                 // Problem loading image
                 if (nPage == 1)
                 {
                        // Problem loading the first image - this is not good
                        nMICRFCount++;
                        wsprintf(szText,"Error %d occured while loading image.\r\n",
                                   nStat);
                        nMICROutput(nFhndle, &szText[0]);
                 }
                 else
                 {
                        // At least the first image was loaded correctly
                        // Assume it was multipage, and all the images have been
                         // processed
                        nPage = 0;
                 }
                 break;
          }
          else
          {
                 // Count the files processed
                 if (nPage == 1)
                        nMICRFCount++;
                 // Count the images processed
                 nMICRICount++;
                 // Create a new bilevel AMP image
                 nStat = ampCreateWorkImage (&pWMicr, 0, 0);
                 if (nStat != 0)
                 {
                        wsprintf(szText,"Error %d when creating image.\r\n",
                                  nStat);
                        nMICROutput(nFhndle, &szText[0]);
                        ampFreeImage (pW);
                        pW = NULL;
                        return (0);
                 }

                 // Prep the image for MICR reading which may
                 // include deskew, scaling and rotation
                 piInfo.BlackEdges = 1;
                 nStat = ampPrepMicr(pW, pWMicr, (PAMPPREPINFO) &piInfo);
                 if (nStat)
                 {
                        wsprintf(szText,"Error %d occured during  image
prep.\r\n",nStat);
                        nMICROutput(nFhndle, &szText[0]);
                        ampFreeImage (pWMicr);
                        pWMicr = NULL;
                        ampFreeImage(pW);
```

```
                pW = NULL;
                return (0);
            }
            ampMI.Dorepair = 1;
            ampMI.Do180 = 1;
            ampMI.UseTranslator = 0 ;
            ampMI.NoBlanks = 0 ;
            ampMI.NoRules = 0 ;
            ampMI.Code = 0 ;
            ampMI.Resolution = -1 ;
            ampMI.MinCon = 80;
            nStat = ampReadMicr (pWMicr, (PAMPMICRINFO) &ampMI);
            if (nStat == 0)
            {
                i = ampMI.lastchar;
                ampMI.resultchar[i] = 0;
                wsprintf(szText,"%s \r\n", ampMI.resultchar);
                nMICROutput(nFhndle, &szText[0]);
            }
            else
            {
                wsprintf(szText,"Error %d occured during MICR processing.\r\n",
                        nStat);
                nMICROutput(nFhndle, &szText[0]);
            }
            nPage++;
            ampFreeImage (pWMicr);
            pWMicr = NULL;
            ampFreeImage(pW);
            pW = NULL;
        }
    }
    if (pWMicr != NULL)
    {
        ampFreeImage (pWMicr);
        pWMicr = NULL;
    }
    if (pW != NULL)
    {
        ampFreeImage (pW);
        pW = NULL;
    }
```

ampLoadImageHnd

```
int ampLoadImageHnd ( pW, fid, options, ftype,
            imgnum )
```

```
PWORKIMAGE pW;
HFILE fid;
LPSTR options;
LPSTR ftype;
int imgnum;
```

This function is the same as **ampLoadImage** except that this function takes an open file handle (such as returned by _lopen or OpenFile) as an argument instead of a file path name. The file is assumed to be opened, readable, and positioned to the beginning of the file. The *ftype* argument is only required if there is no recognizable header.

This API always returns 0 if the function was successful, non-zero otherwise.

ampLoadImageBuffer

```
int ampLoadImageHnd ( pW, options, ftype,
            imgnum, bufferaddr , buffersize )
```

```
    PWORKIMAGE pW;
    LPSTR options;
    LPSTR ftype;
    int imgnum;
    LPBYTE bufferaddr;
    int buffersize;
```

This function is the same as **ampLoadImage** except that this function takes a memory buffer that has been preloaded with buffersize number of data bytes read from any of the file types supported by AMPLIB. The *ftype* argument is recommended because there is no filename extension that can be used to help ascertain the file compression format.

This API always returns 0 if the function was successful, non-zero otherwise.

ampSaveClipboard

```
int ampSaveClipboard ( pW, hWndMain )
```

```
    PWORKIMAGE pW;
    HWND hWndMain;
```

This function will save a work image *pW* to the Windows clipboard in a DIB format.

This API always returns 0 if the function was successful, non-zero otherwise.

ampSaveImage

```
int ampSaveImage (name, filepathname, options,
            ftype, kdy)
```

```
PWORKIMAGE  name;
LPSTR filepathname;
LPSTR  options;
LPSTR  ftype;
int    kdy;
```

The ampSaveImage API writes the image *name* to the disk file specified by *filepathname*.

Some extensions are recognized:

TIF	File is a TIFF file
PCX	File is a Paintbrush PCX file
BMP	File is a Bitmap
JPG	File is a JPEG

Options is a string of characters that describe optional processing for the image data during save:

A	Appends image data to file; used for creating multi-image format files.
B	Bit-byte-reverses the image data stream. See note above. Without this option, TIFF files will be created with Fill Order 2. With the **B** option, the Fill Order is set to 1.
Q	Format is "Q=n", which sets the quality level for JPEG files.
R	Format is "R=*n*", which sets the X and Y resolution tags in the TIFF file to *n*; used when the resolution information is not already known internally or you need to override the value.
X	Format is "X=*n*", which sets the X-resolution tag in the TIFF file to *n*.
Y	Format is "Y=*n*", which sets the Y-resolution tag in the TIFF file to *n*.
U	Format is "U=*n*", which sets the ResolutionUnits tag in the TIFF file to *n*; the default value is 2.

Ftype identifies the file organization to use, as described at the top of this manual section. It is only needed when the file type cannot be inferred from the filename extension, or you wish to override the value implied by the file name extension.

Kdy is used only with Group 3-2d, and specifies the duplication factor.

This API always returns 0 if the function was successful, non-zero otherwise.

Examples:

```
ampSaveImage( mine,"FD0023.TIF",
"R=200",NULL, 0);

ampSaveImage( mine,"STRANGE.XYZ",
"TB","G32/NOHEADER",4);

ampSaveImage( mine,"TEMP.PCX", NULL, NULL,
0);
```

The first example writes a TIFF file with default parameters of Group 4 compression, forcing the resolution tag to be 200 dpi. The second creates a headerless file with a non-standard extension, selects bit-byte-reversal of the image data, and compresses the file using CCITT Group 3 -2d format with *kdy*=4. The third example creates a PCX file.

ampSaveImageBuffer

```
int ampSaveImage (name, options, ftype, kdy,
            pbyBuffer, nBufferSize,
            pnBufferUsed)
```

```
    PWORKIMAGE  name;
    PSTR filepathname;
    PSTR  options;
    PSTR  ftype;
    int    kdy;
    PBYTE pbyBuffer;
    int nBufferSize;
    PINT pnBufferUsed;
```

The ampSaveImage API compresses the image *name* and writes the compressed data to the memory buffer pointed to by pbyBuffer. The memory buffer has a size of nBufferSize. When the operation completes, the number of compressed image bytes is reported back through pnBufferUsed. Only TIFF bilevel data formats are output to the buffer.

Options is a string of characters that describe optional processing for the image data during save:

B Bit-byte-reverses the image data stream. See note above. Without this option, TIFF files will be created with Fill Order 2. With the **B** option, the Fill Order is set to 1.

R Format is "R=*n*", which sets the X and Y resolution tags in the TIFF file to *n*; used when the resolution information is not already known internally or you need to override the value.

X Format is "X=*n*", which sets the X-resolution tag in the TIFF file to *n*.

Y Format is "Y=*n*", which sets the Y-resolution tag in the TIFF file to *n*.

U Format is "U=*n*", which sets the ResolutionUnits tag in the TIFF file to *n*; the default value is 2.

Ftype identifies the file organization to use, as described at the top of this manual section. It is only needed when the file type cannot be inferred from the filename extension, or you wish to override the value implied by the file name extension.

Kdy is used only with Group 3-2d, and specifies the duplication factor.

This API always returns 0 if the function was successful, non-zero otherwise.

Examples:

```
    ampSaveImageBuffer( mine,"R=200",NULL, 0,
    &byBuffer, sizeof(byBuffer), &nByteCount);

    ampSaveImageBuffer( mine,
    "TB","G32/NOHEADER",4, pbyBuffer, 100000,
    nBufferUsed);
```

The first example writes an equivalent TIFF file to memory with default parameters of Group 4 compression, forcing the resolution tag to be 200 dpi. The second creates a headerless file in the buffer with a non-standard extension, selects bit-byte-reversal of the image data, and compresses the file using CCITT Group 3 -2d format with *kdy*=4.

ampSaveImageHnd

```
int ampSaveImageHnd (name, fid, options,
            ftype, kdy)
```

```
    PWORKIMAGE  name;
    HFILE  fid;
    LPSTR  options;
    LPSTR  ftype;
    int    kdy;
```

This function is the same as **ampSaveImage** except that this function takes an open file handle (such as returned by _lopen or OpenFile) as an argument instead of a file path name. The file is assumed to be opened, have both read and write permission, and be positioned to where you want to start writing the image data. The *ftype* argument is required to specify the type of file to create.

This API always returns 0 if the function was successful, non-zero otherwise.

ampSetImageMargins

```
int ampSetImageMargins( pW, top, left, bottom,
            right)
```

```
PWORKIMAGE   Pw;
long top;
long left;
long bottom;
long right;
```

This function is used with variable size work images and the LoadImage API. If the image margins are set to some non-zero value, additional space will be allocated to form a margin, or frame around the image data. The margin area will have undefined data, and after loading the image, a subsequent call to **ampOutsideFillImage** should be made to set it to all zeros or all ones.

This API always returns 0 if the function was successful, non-zero otherwise.

ampSetInputImageMetrics

```
int ampSetInputImageMetrics ( Pw, dx, dy,
            filepos )
```

```
    PWORKIMAGE pW;
    long  dx, dy;
    long  filepos;
```

This function is used to specify the input image dimensions of a file when the information cannot be determined by the file header. If this function has not been called, the current sub-image metrics for the work image will be used.

A second use for this function is to force decompression to terminate early. For example, you may want to decompress only the top few inches of a document to extract a bar code or other information; the rest of the document isn't needed. You can specify a non-zero value for *dy* to decompress only *dy* lines of the image. (When working with structured image files, always set *dx* to 0, to ensure the correct dx value is used from the header.)

The *filepos* parameter gives the byte offset from the start of the file where the actual image data begins. This is normally 0, but can be overridden by this function for files loaded into image *pW*.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampSetInputImageMetrics(pWMine,2560,3296,0);
    ampLoadImage( pWMine, "TEST1.G4","R=90",
                      "G4/NOHEADER", 0);
```

Image Manipulation Functions

This group of functions provides capabilities for manipulating the image data stored in work images. A variety of general-purpose functions are included and are standard on all AMPLIB systems.

NOTE: Some Image Manipulation Functions may only accept binary images as inputs and outputs. Functions that begin with *ampGray* generally accept grayscale or color images as inputs and outputs. There are functions that can accept either binary, grayscale, or color images (e.g. ampClearImage, ampCopyImage, ampFillImage, ampInvertImage, ampMirrorImage, ampOutsideFillImage and ampRotateImage) and are so described in the text. Functions that use source and destination images should be given work images that have the same pixel depth. Use ampConvertImage to convert pixel depths between binary, grayscale, and color work image types

ampAnnotateImage

```
int ampAnnotateImage (pWDestname, szText,
            szFont, szOptions, rectRegion)
```

```
    PWORKIMAGE pWDestname;
    PSTR szText;
    PSTR szFont;
    PSTR szOptions;
    RECT *rectRegion;
```

The ampAnnotateImage API provides a general means of writing a text annotation onto the image given by pWDestname.

szText is a string of characters (in the ANSI character set) that is to be written onto the image. They will be written to the image area specified by the rectRegion argument.

szFont gives the ASCII name of the typeface to be used when writing the annotation. The font must be installed in your Windows system. For best scaling performance choose a TrueType or ATM font.

szOptions is a string of characters that describe optional processing for the image annotation prior to the save:

B Writes annotation with reversed background, e.g. white characters on a black background.
F Draws a frame around the annotation region.
I Writes the annotation with italicized text.
J Specifies the justification desired for the text within the given region. Format is "J=c" where c is as follows:
L: Left justify
C: Center
R: Right justify

O Defines the orientation or slant of the annotation text baseline and and is in the format "P=n" where n can be a positive or negative number in tenths of degrees. Positive numbers rotate the text counter-clockwise and negative numbers clockwise. The default value is 0 which is traditional horizontal left-to-right text. A negative value of -900 would create top-down vertical text. Text starts at the X,Y offset as specified below.

P Defines the pitch of the text in the annotation and is in the format "P=n" where n is 0-2 as follows:
0: Default
1: Variable
2: Fixed
Q Defines which corner to anchor the annotation region to. Format is "Q=n", where n is 0 through 4 as follows:
0: Anywhere on page
1: Upper right hand corner
2: Upper left hand corner
3: Lower left hand corner
4: Lower right hand corner
S Specifies the point size of the annotation text and is in the format "S=n". A 72 point font has uppercase letters one inch tall.
T Writes annotation with transparent background.
U Annotation text is drawn with an underline.
W Specifies the weight of the strokes used in the annotation text and is in the format "W=n". A value of 400 is normal with smaller values producing text with lighter/thinner strokes and larger higher making text with bolder/thicker strokes. A value of 0 selects the current value which is typically normal.
X Specifies the starting horizontal pixel location in the annotation region where text will be begun in left justify mode. The default value for left justified text is 8. The center justify and right justify options automatically change this value as required.
Y Specifies the starting vertical pixel location in the annotation region where text will be begun. The default value is 0 which is the top of the region.

rectRegion, along with the **Q** option, specifies where to write the annotation stamp onto the image. rectRegion is a pointer to a RECT structure and its use here is defined as follows (all values are long integers in units of 1/100 inch):

right	width of stamp region
bottom	height of stamp region
left	horizontal offset inward from corner
top	vertical offset inward from corner

The text will be written onto the image within the region specified, using the requested options and font. If a frame has been requested, it will be drawn with a single pixel line rectangle around the region. If the text will not fit within the region, it will be clipped. No notification will be given if clipping occurs.

Note that the requested annotation region size and position may be adjusted by the program to maintain pixel alignment rules.

This API always returns 0 if the function was successful, non-zero otherwise.

Examples:

```
RECT rectRegion;
rectRegion.top = 0;
rectRegion.left = 0;
rectRegion.right = 150;
rectRegion.bottom = 25;
ampAnnotateImage(pWDest, "000123", "Times New Roman",
        "TFQ=4", &rectRegion);
```

ampBitBltImage

```
int ampBitBltImage ( pWSrcname, pWDestname,
            opcode)
```

```
PWORKIMAGE pWSrcname;
PWORKIMAGE pWDestname;
int opcode;
```

This function performs a generalized Bit Block Transfer (BitBlt) of image data from image pWS*rcname* to image pWD*estname* using BitBlt function *opcode*.

BitBlt operations are useful for performing a variety of bitwise-logical functions on the image data. Both pWS*rcname* and pWD*estname* are required arguments, but may be the same so that data may be transformed in place. *Opcode* is given as a number and is defined as follows:

0: source →dest
1: source AND dest → dest
2: source AND NOT dest → dest
3: 0's → dest
4: source OR NOT dest → dest
5: source XNOR dest → dest
6: NOT dest → dest
7: source NOR dest → dest
8: source OR dest → dest
9: dest → dest
10: source XOR dest → dest
11: NOT source AND dest → dest
12: 1's → dest
13: NOT source OR dest → dest
14: source NAND dest → dest
15: NOT source → dest

This API always returns 0 if the function was successful, non-zero otherwise.

ampClearImage

```
int ampClearImage( pW )
```

```
    PWORKIMAGE  pW;
```

Clears the image pW to all zero pixels, which is the CCITT convention for an all white page. Grayscale and color images are also set to white. The grayscale white pixel value is 255. White color images have the red, green, and blue component of each pixel set to 255.

This API always returns 0 if the function was successful, non-zero otherwise.

ampConvertImage

```
int ampConvertImage ( pWSrcname, pWDestname,
            lStyle, lMode )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    long lStyle;
    long lMode;
```

This function copies the image data from image pWS*rcname* to image pWD*estname* converting pixel depths as needed. Consequently, source and destination images may be any mixture of binary, grayscale, or color. When converting from color to grayscale, the ratios of the red, green, and blue color components to the final grayscale value are 30%, 59%, and 11%. When converting from color to binary or grayscale to binary, a sophisticated edge-sensitive thresholding algorithm is used to preserve as much fine detail as possible. This is especially significant when the image has a MICR line or barcode to be read.

This function supports copying the subimage of the source to the subimage of the destination. If the source subimage is larger than the destination's then only the upper left portion of the data that fits will be transferred.

Currently the parameters lStyle and lMode are reserved for features that will be implemented sometime in the future. For now, simply set these parameters to zero. This API always returns 0 if the function was successful, non-zero otherwise.

ampCopyImage

```
int ampCopyImage ( pWSrcname, pWDestname )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
```

This function copies the image data from image pWS*rcname* to image pWD*estname*. Source and destination images may be both binary, grayscale, or color. Do not mix binary, grayscale, or color images in the same call.

This function supports copying the subimage of the source to the subimage of the destination. If the source subimage is larger than the destination's then only the upper left portion of the data that fits will be transferred.

This API always returns 0 if the function was successful, non-zero otherwise.

ampDeSkew

```
int ampDeSkew( pWSrcname, pWDestname, params )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    ampDESKEWINFO *params;
```

The binary or grayscale image *pWSrcname* will be copied into image *pWDestname* rotating the image clockwise or counter-clockwise as needed to remove any detected skew. How skew is detected, corrected, and reported is controlled by the dereferenced *params* structure.

The fields of ampDESKEWINFO are defined as follows (see interface reference file for latest definition).

```
    typedef struct of_deskewinfo {
          double MinSkew;
          double MaxSkew;;
          BOOL BlackEdges;
          BOOL DetectOnly;
          int RegType;
          int Confidence;   // input/output
          double SkewDetected; // out
          } ampDESKEWINFO;
```

MinSkew is the minimum amount of skew the process will work with and is specified in degrees.

MaxSkew is the maximum amount of skew. Images with skews outside the range of *MinSkew-MaxSkew* will not be modified.

BlackEdges causes the skew detection logic to look for black borders on the source image when TRUE and white borders when FALSE.

DetectOnly detects the skew without modifying the image when set TRUE. Results are reported back through the ampDESKEWINFO structure.

RegType controls how the resultant image will be registered after the deskew operation. Allowed values are: 0=Default, 1=Average, and 2=Forms.

With *RegType*=0, a white border will be applied to the frame as determined by the greatest extents of the image border widths before deskew.

With *RegType*=1, a white border will be applied to the frame as determined by the average border widths before the deskew operation.

With *RegType*=2, the deskewed image data is repositioned so that the top and left white border is removed.

Confidence is a value from 0 to 100 that limits deskew operations on images of low quality. A typical *Confidence* value of 50 works fine for most images. Lower numbers allow for deskew to take place with less information, hence less reliable determination of the skew angle. Higher numbers make the process more reliable, at the expense of not performing deskew on some images with lower confidence numbers. This value will be loaded with a computed value on return from the call, so be sure to set it to a meaningful value before issuing the ampDeSkew call.

SkewDetected contains the return value of the detected skew angle.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
ampDESKEWINFO deskewinfo;

deskewinfo.DetectOnly = FALSE;
deskewinfo.BlackEdges = TRUE;
deskewinfo.Confidence = 50;
deskewinfo.MinSkew = .15,
deskewinfo.MaxSkew = 45;
deskewinfo.RegType = 2;
deskewinfo.SkewDetected = 0;
nStat = ampDeSkew (pWSrc, pWDest, &deskewinfo);
```

ampDitherImage

```
int ampDitherImage ( pWSrcname, pWDestname,
            lStyle )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    long lStyle;
```

The grayscale image *pWSrcname* will be copied into the bilevel image *pWDestname* dithering the destination image according to lStyle. If lStyle is 0, an 8x8 grid dot pattern will be used that renders an effective 64 shades of gray. If lStyle is non-zero, a 4x4 pattern on a diagonal grid is used that gives 16 shades of gray. After the operation, pWDestname will have the same horizontal and vertical dimensions of pWSrcname.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    lStyle = 0;
    nStat = ampDitherImage (pWSrc, pWDest, lStyle);
```

ampFillImage

```
int ampFillImage( pW, color )
```

```
    PWORKIMAGE pW;
    WORD color;
```

This function will load either white or black into the work image *pW* which may be either binary, grayscale, or color. *Color* is given as a 16-bit unsigned integer value, which if zero will load white space into pW. Any nonzero value will load black into a binary image. For grayscale and color images, the complement of the *Color* value will be used for the grayscale or color components. For example, a *Color* value of 1 will set a grayscale pixel to 254 (near white). A *Color* value of 253 will set the red, green, and blue components of a color work image pixel to 2 (dark black).

This API always returns 0 if the function was successful, non-zero otherwise.

ampGrayMirrorImage

```
int ampGrayMirrorImage( pW, axis )
```

```
PWORKIMAGE pW;
char axis;
```

This function mirrors the grayscale or color data in the image *pW* about the selected axis. Rows or columns of image data are interchanged in such a way as to produce a mirror copy of the original, as would be needed if a sheet of microfilm were scanned upside down. *Axis* is given as 'X' or 'Y'. Choosing 'X' mirrors top-to-bottom, 'Y' mirrors left-to-right.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
ampGrayMirrorImage (pW, 'Y');
```

ampGrayProcesses

```
int ampGrayProcesses(pWs, Process)
```

```
PWORKIMAGE  pWs;
int    Process;
```

The ampGrayProcesses API performs one of many functions on the grayscale data located within the subimage portion of the image. Grayscale value range from 0 (black) to 255 (white).

Process may be one of the following operations:

GRAYLIGHTEN Lightens the grayscale image 8 steps. Values do not exceed 255.

GRAYDARKEN Darkens the grayscale image 8 steps. Values are truncated at 0.

GRAYREVERSE Grayscale values are switched from black-to-white and from white-to-black.

GRAYSTRETCH Grayscale values from 32 - 224 are contrast enhanced to range from 0 - 255. Values less than 16 become black (0). Values greater than 232 become white (255).

GRAYCOMPRESS Image contrast is decreased by remapping values from 0-255 to 32-224. This effectively reverses the GRAYSTRETCH operation.

GRAYNORMALIZE The image is analyzed to find the minimum and maximum grayscale values. These values then become the new endpoints for a GRAYSTRETCH contrast enhancement.

GRAYTHRESHOLD The image will be thresholded to black and white. All values below 128 will become black. All values above and including 128 will become white.

GRAYMILDSHARPEN Enhances the edges of image features by using a 3x3 Laplacian filter.

GRAYSTRONGSHARPEN Exaggerates the edges of image features by using a 3x3 Laplacian filter.

GRAYMILDBLEND Softly blurs the edges of image features by using a 3x3 Laplacian filter.

GRAYSTRONGBLEND Strongly blurs the edges of image features by using a 3x3 Laplacian filter.

GRAYGAMMALIGHTEN1 Lightens the grayscale image non-linearly in the midtone region. Gamma correction is used primarily on grayscale images before dithering and then printing to make the final output clearer on laser printers.

GRAYGAMMALIGHTEN2 Stronger lighten of the midtones than GRAYGAMMALIGHTEN1. Strong blacks like those found in the MICR line are preserved.

GRAYGAMMALIGHTEN3 Stronger lighten of the midtones than GRAYGAMMALIGHTEN2.

GRAYGAMMAEQUAL Equalizes the grayscale values of the image which generally improves the contrast when viewed.

GRAYGAMMASQRT The image is lightened using a square root based non-linear algorithm.

GRAYGAMMASQUARE Darkens the image with a non-linear transfer function.

GRAYGAMMALOG The image is lightened using a logarithm based transfer function.

GRAYGAMMAINVLOG Darkens the image with a non-linear transfer function based on inverse logarithms.

GRAYGAMMAGAUSSIAN The image is lightened with a Gaussian based transfer function.

This API always returns 0 if the function was successful, non-zero otherwise.

Examples:

```
ampGrayProcesses(pWGray,
GRAYLIGHTEN);

ampGrayProcesses(pWGray,
GRAYREVERSE);

ampGrayProcesses(pWGray,
GRAYSTRONGSHARPEN);
```

ampGrayScaleResolution

```
int ampGrayScaleResolution( pWSrcname,
            pWDestname, Xres, Yres )
```

```
PWORKIMAGE pWSrcname;
PWORKIMAGE pWDestname;
int   Xres;
int   Yres;
```

The grayscale or color image *pWSrcname* will be scaled into image *pWDestname* using independent scaling factors in the horizontal and vertical directions. These scaling factors will be determined by taking Xres and Yres and dividing by their corresponding values in pWSrcname. Linear interpolation is used in the scaling process to improve destination image accuracy.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
ampGrayScaleResolution (pWFax, pWTemp, 300,
300);
```

ampInvertImage

```
int ampInvertImage( pW )
```

```
PWORKIMAGE pW;
```

Inverts the sense of the image data in the binary, grayscale, or color image *pW*. All white pixels are set to black and all black pixels are set to white. For a binary image that means all "0" pixels are set to "1" and all "1" pixels are set to "0." In a grayscale image, the values from 0 through 255 are mapped into the values from 255 through 0. In a color image, the red, green, and blue color components from 0-255 are individually converted to 255-0.

This API always returns 0 if the function was successful, non-zero otherwise.

ampMirrorImage

```
int ampMirrorImage( pW, axis )
```

```
    PWORKIMAGE pW;
    char axis;
```

This function mirrors the data in the binary, grayscale, or color image *pW* about the selected axis. Rows or columns of image data are interchanged in such a way as to produce a mirror copy of the original, as would be needed if a sheet of microfilm were scanned upside down. *Axis* is given as 'X' or 'Y'. Choosing 'X' mirrors top-to-bottom, 'Y' mirrors left-to-right.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    ampMirrorImage (pWMain, 'X' );
```

ampOutsideFillImage

```
int ampOutsideFillImage( pW, data )
```

```
PWORKIMAGE pW;
WORD data;
```

This function will load a constant value into the binary, grayscale, or color image *pW* in the perimeter area outside of the sub-image area. For example, if PITCH=HEIGHT=2048, X=Y=256, and DX=DY=1536, then this function would fill a 256 pixel wide border around the sub-image data.

Data is given as a 16-bit integer value, which is loaded into the image 16 pixels at a time for binary images. For grayscale and color images, the complement of the *Color* value will be used for the grayscale or color components. For example, a *Color* value of 1 will set a grayscale pixel to 254 (near white). A *Color* value of 253 will set the red, green, and blue components of a color work image pixel to 2 (dark black).

This function can be used as a simple crop or frame facility.

This API always returns 0 if the function was successful, non-zero otherwise.

ampRotateImage

```
int ampRotateImage( pWSrcname, pWDestname,
            angle )
```

```
PWORKIMAGE pWSrcname;
PWORKIMAGE pWDestname;
int   angle;
```

The image *pWSrcname* will be rotated into image *pWDestname* by *angle* degrees. *Angle* must be either 90, 180, or 270. Rotation is in the counter-clockwise direction. Source and destination images may be both binary, both grayscale or both color. Do not mix binary and grayscale images in the same call.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
ampRotateImage (pW, pWRotate, 90 );
```

ampScaleImage

```
int ampScaleImage ( pWSrcname, pWDestname,
            threshold,
            xpixels_in, xpixels_out,
            ypixels_in, ypixels_out )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    int  threshold;
    int  xpixels_in, xpixels_out;
    int  ypixels_in, ypixels_out;
```

This function scales the input binary image *pWSrcname* into the output image *pWDestname* using the scaling parameters given in the function call. The x*pixels_in* and x*pixels_out* give the scaling parameters for the X-axis; the *ypixels_in* and *ypixels_out* give the scaling parameters for the Y-axis.

Threshold is used for some special case scale factors. When the x and y scale factors are the same, the following scaling ratios are treated specially: 2:1, 3:1, 4:1, 8:1

When using one of these special scaling ratios, you vary the "darkness" of the result by adjusting the threshold between 1 and the scale factor squared. E.g., for 3:1 scaling, the allowable threshold values are from 1 to 9. Smaller values will tend to darken (or embolden) characters; larger values will lighten them. If you pass 0 for a threshold value, the API will compute a value for you.

Examples:

```
    ampScaleImage(pWOriginal, pWScaled, 5,
                3, 1, 3, 1);
    ampScaleImage( pWOriginal, pWScaled, 0,
                4,3,  3, 2);
```

The first example scales the image 3:1 with a threshold of 5; i.e., for every 3 input pixels, only one output pixel is produced. The second example scales the image 4:3 in the horizontal (X-axis) direction and 3:2 in the vertical (Y-axis) direction.

Scaling bi-tonal images is problematic and the user should not assume that the resulting image would be free of any distortion or breakup. Because pixels in a bi-tonal image can only take on two values, 0 or 1, fractional pixel scaling is not possible. A pixel is either present or not; it cannot be made into a half-pixel. (This can be approximated in the gray-scale domain, but not with a binary representation.) Thus, scaling is performed by either duplicating pixels, or removing pixels. At small scaling ratios, such as 2:3, pixel neighborhood operations can be performed to improve appearance, but at higher ratios this is not practical.

The requested scaling ratio will be adjusted to fit a calculated "sweet" scaling ratio. Thus, very small changes in the requested ratio may produce no difference in the output scaling.

The scaling accuracy is typically within 2 percent. As an example, consider a scaling ratio of 100:97, or a 3% reduction. Essentially, for every 100 pixels input, 3 will be discarded. Where these pixels appear in your image is unpredictable; they may result in "squeezing" some characters on a line by one pixel. This results in a distortion of characters that is visible to the eye. If the image width were 128, then 125 pixels would be output, for an effective ratio of 128/125 = 2.34% reduction.

In order to achieve the best possible results, AMPLIB will perform a greatest common denominator algorithm on the specified ratios to find the closest matching nearest-neighbor algorithm for the desired ratio. The algorithm guarantees the correct number of pixels in the output image, although the actual scaling ratio used may be slightly different than requested. The remainder of the output image will be padded with white space.

This API always returns 0 if the function was successful, non-zero otherwise.

ampThresholdImage

```
int ampThresholdImage ( pWSrcname, pWDestname,
            nDestResolution, &dblDestWidth,
            &dblDestHeight, nType )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    int nDestResolution;
    double * dblDestWidth;
    double * dblDestHeight;
    int nType;
```

The grayscale image *pWSrcname* will be dithered into the bilevel image *pWDestname.* The type of dithering pattern is determined by nType and may be one of the following values:

```
TIDITH64    0   - 64 shades of gray
TITHRESH128 1   - bilevel
TIDITH16    2   - 16 shades of gray
TICHECKEDGE 3   - check based
```

The final size of the destination image is determined by the nDestResolution, dblDestWidth, and dblDestHeight input variables according to the following conditions. If either dblDestWidth or dblDestHeight is zero, then the output dimensions will be equal to the input dimensions of *pWSrcname* multiplied by nDestResolution divided by the resolution of *pWSrcname.* Otherwise the destination image will have the size in inches specified by dblDestWidth and dblDestHeight at the resolution specified by nDestResolution. The aspect ratio of the original source image will be preserved so the actual final resolution may not exactly equal dblDestWidth and dblDestHeight. Usually a value of 600 for nDestResolution gives a good combination of execution speed and image clarity. A value of 300 dpi will cause the image transfer to happen faster with the final image having less detail.

The nType value of 3 selects a threshold process that assumes the result will be a check image. This process can then use a feedback mechanism to refine the image based on standard values for check images.

This API always returns 0 if the function was successful, non-zero otherwise. If successful, dblDestWidth and dblDestHeight will be filled with the final horizontal and vertical dimensions of pWDestname.

Example:

double dblWidth = 5.0;
double dblHeight = 4.0;
nStat = ampThresholdImage (pWSrc, pWDest, 600,
&dblWidth, &dblHeight, TIDITH64);

ampDynamicThreshold

```
int ampDynamicThreshold ( pWSrcname,
            pWDestname, PCS, ABSBLACKTHRESH,
            HISTO, LOWPASS)
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    double PCS;
    int ABSBLACKTHRESH;
    int HISTO;
    int LOWPASS;
```

The grayscale image pWSrcname will be converted into the bilevel image pWDestname using a dynamic thresholding technique very good for checks particularly with background patterns and noise. This system requires a high quality grayscale image (> 80 DPI and 256 levels of gray). The function will return error code <202> if quality of grayscale image is below these levels.

PCS defines the contrast ratio threshold with a range from 0-1. If PCS = 0.0 the system will use the default threshold of 0.15 which is known to produce optimal b/w images for CAR/LAR recognition. Use a lower setting to darken the image, a higher setting to lighten the image. ABSBLACKTHRESH is the absolute black threshold. Any grayscale pixel below this threshold will be converted to a black pixel in the binary image. Default is set to 55 if ABSBLACKTHRESH = 0.

If HISTO = 1, the system will determine the optimal threshold curve based on histogram analysis. Use HISTO = 0 only if you know the images have good dynamic range for contrast. The histogram analysis will require extra processing time. Good quality check scanners produce images with good dynamic range. Use HISTO for using page scanner or unknown scanning device.

If LOWPASS = 1, the system will filter out high frequency noise in the grayscale image producing very clean image with low compressed fiel size. Set LOWPASS = 0 if this operation is not desired.

The Pitch of the b/w image will be increased to ensure the image rows end on a byte boundary with white pixels added for padding.

Image Filtering Functions

This class of functions performs binary algorithmic and morphological filtering operations on images, as would be used for image enhancement, background removal, etc. You must have the Image Filter option set in your license file to use these functions.

ampDeBorder

```
int ampDeBorder( pWSrcname, pWDestname,
            NoiseWidth, TotalRemoved )
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    int   NoiseWidth;
    PINT  TotalRemoved;
```

The binary image *pWSrcname* will be copied into image *pWDestname* removing black border pixels. White noise within the black edge may be ignored as long it has an overall linear dimension less than NoiseWidth. The function will set the dereferenced TotalRemoved parameter to the number of pixels that were removed from all edges.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    nNoiseWidth = 10;
    nTotalRemoved = 0;
    // Deborder the source image into the destination image
    nStat = ampDeBorder(pWSrc, pWDest, nNoiseWidth,
            &nTotalRemoved);
```

ampDeLine

```
int ampDeLine( pWSrcname, pWDestname, params,
            results )
```

```
PWORKIMAGE pWSrcname;
PWORKIMAGE pWDestname;
ampDELINEINFO *params;
ampLINEOBJECT *results;
```

The binary image *pWSrcname* will be copied into image *pWDestname* removing lines in the process. The definition of what constitutes a line is controlled by the dereferenced *params* structure. The locations of the lines removed are returned in an array of ampDELINEINFO structures, pointed to by *results*.

The fields of ampDELINEINFO are defined as follows (see interface reference file for latest definition).

```
typedef struct of_delineinfo {
      int MinLineWidth;
      int MaxLineWidth;
      int MinLineHeight;
      int MaxLineHeight;
      int LineGap;
      int MaxExports;   // input/output
      BOOL DetectOnly;
      BOOL EdgeClean;
      int SmearGap;
} ampDELINEINFO;
```

MinLineWidth is the minimum pixel length of candidate line objects. Objects shorter than this will be retained in the destination image.

MaxLineWidth is the maximum pixel length of candidate line objects.

MinLineHeight is the minimum pixel thickness that candidate lines must have.

MaxLineHeight marks the maximum pixel thickness threshold for candidate lines.

LineGap specifies how many white pixels may separate adjoining line segments before those line segments cease to be identified as a line. Breaks in lines larger than LineGap are treated as separate lines.

MaxExports is the number of ampLINEOBJECT structures that have been allocated for result reporting. On output, it returns the number of line objects.

DetectOnly inhibits bitmap line removal if TRUE, but still reports back the operational results as if line removal had occurred

EdgeClean aggressively removes noise pixels from the top and bottom edges of the line while removing the line. Set to TRUE if lines are not being thoroughly removed.

SmearGap performs a deline operation, a smear, and then another deline if nonzero. This is often used with black lines containing reverse text. The first deline will remove the "lines" above and below the text. The smear will fill in the text and make a fully connected line. Finally the new line is removed.

The fields of ampLINEOBJECT are defined as follows (see interface reference file for latest definition).

```
typedef struct of_lineobject {
      long  x1;   // x,y of line beginning
      long  y1;
      long  x2;   // x,y of line end
      long  y2;
      int   width; // line thickness
} ampLINEOBJECT;
```

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
lineCount = 50;
lineWidth = 1000

lineResults = (PAMPLINEOBJECT) calloc (lineCount *
sizeof
      (ampLINEOBJECT), 1);

// Clear out input and output parameters
memset (&delineInfo, 0, sizeof (ampDELINEINFO));

delineInfo.MinLineWidth = lineWidth / 2;
delineInfo.MaxLineWidth = lineWidth;
delineInfo.MinLineHeight = 1;
delineInfo.MaxLineHeight = lineWidth / 100;
delineInfo.LineGap = 5;
delineInfo.SmearGap = 5;
delineInfo.MaxExports = lineCount;
delineInfo.DetectOnly = 0;
delineInfo.EdgeClean = 1;

// Deline the source image into the destination image
 nStat = ampDeLine(pWCheck, pWTemp, &dlInfo,
      lineResults);
```

ampDeShade

```
int ampDeShade( pWSrcname, pWDestname, params,
            results )
```

```
PWORKIMAGE pWSrcname;
PWORKIMAGE pWDestname;
ampDESHADEINFO    *params;
ampSHADEOBJECT    *results;
```

The binary image *pWSrcname* will be copied into image *pWDestname* removing regions of half-toning usually caused by watermarks or photographs. The definition of what constitutes a shade region is controlled by the dereferenced *params* structure. Locations of the erased regions are reported back through *results* which points to an array of ampDESHADEINFO structures.

The fields of ampDESHADEINFO are defined as follows (see interface reference file for latest definition).

```
typedef struct of_AMPDESHADEINFO {
      int MinRegionWidth;
      int MinRegionHeight;
      int MinSpecWidth;
      int MinSpecHeight;
      int SpecWidthAdj;
      int SpecHeightAdj;
      int MaxExports;   // in/out
      BOOL DetectOnly;
} ampDESHADEINFO;
```

MinRegionWidth is the minimum horizontal pixel width of candidate shade region objects. Objects narrower than this will be retained in the destination image.

MinRegionHeight is the minimum vertical pixel height of candidate regions.

MinSpecWidth is the expected horizontal pixel dimension of speckles within the shade region.

MinSpecHeight is the expected vertical pixel dimension of speckles within the shade region.

SpecWidthAdj is reserved for future use.

SpecHeightAdj is reserved for future use.

MaxExports defines the number of ampDESHADEINFO structures on input and returns how many shade objects were processed on output.

DetectOnly inhibits shade region removal if TRUE, but still locates and reports back the objects found via *results*.

The fields of ampSHADEOBJECT are defined as follows (see interface reference file for latest definition).

```
typedef struct of_shadedobject {
      long  x;
      long  y;
      long  dx;
      long  dy;
```

```
        long  ModifiedComponents;
        long  ModifiedRuns;
        long  ModifiedPixels;
    } ampSHADEOBJECT;
```

x is the horizontal location of the upper left corner of the region.

y is the vertical location of the upper left corner of the region.

dx is the region width.

dy is the region height.

ModifiedComponents is the total number of runs and pixels found in this shade object.

ModifiedRuns specifies how many black runs were modified and changed to white runs when erasing this particular shaded region.

ModifiedPixels specifies how many black pixels were erased while removing this shade region.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    shadeCount = 50;
    pshadeResults = (PAMPSHADEDOBJECT) calloc
            (shadeCount * sizeof (ampSHADEDOBJECT), 1);
    // Clear out input and output parameters
    memset (&deshadeInfo, 0, sizeof (ampDESHADEINFO));
    deshadeInfo.MinRegionHeight = 30;
    deshadeInfo.MinRegionWidth = 20;
    deshadeInfo.MinSpecWidth = 4;
    deshadeInfo.MinSpecHeight = 4;
    deshadeInfo.SpecWidthAdj = 4;
    deshadeInfo.SpecHeightAdj = 4;
    deshadeInfo.MaxExports = shadeCount;
    // Deline the source image into the destination image
    nStat = ampDeShade(pWS, pWD, &deshadeInfo,
            pshadeResults);
```

ampDeSpec

```
int ampDeSpec(pWSrcname, pWDestname, params)
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    ampDESPECINFO *params;
```

The binary image *pWSrcname* will be copied into image *pWDestname* removing small speckles that are usually caused by scanning noise. The definition of what constitutes a speckle is controlled by the dereferenced *params* structure.

The fields of ampDESPECINFO are defined as follows (see interface reference file for latest definition).

```
    typedef struct of_AMPDESPECINFO {
          int MinWidth;
          int MaxWidth;
          int MinHeight;
          int MaxHeight;
          int ConnectivityType;
          long TotalPixels;
          long TotalPixelsModified;
          long TotalComponents;
          long TotalComponentsModified;
    } ampDESPECINFO;
```

MinWidth is the minimum horizontal pixel width of candidate speckles objects. Objects narrower than this will be retained in the destination image.

MaxWidth is the maximum allowed horizontal pixel width for a speckle candidate object.

MinHeight is the minimum vertical pixel dimension for candidate speckle objects.

MaxHeight specifies the maximum allowed vertical pixel height for a speckle.

ConnectivityType specifies to the despeckle engine the kind of connectivity definition to use during the operation. 4-neighbor (or 4-connected) means the adjacent pixels to the North, East, South, and West. A value of 8 tells the software to use these four plus the 4 diagonal neighbors.

TotalPixels specifies how many black pixels were counted in the image.

TotalPixelsModified specifies how many black pixels were made white during the despeckle operation.

TotalComponents is the total number of connected shapes found in the image.

TotalComponents Modified is the number of connected shapes in the image that qualified by size as speckles and which were consequently removed from the image.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
// Clear out input and output parameters
memset (&despecInfo, 0, sizeof (ampDESPECINFO));
// Set input parameters to define speckle size
despecInfo.MinWidth = 1;
despecInfo.MaxWidth = 3;
despecInfo.MinHeight = 1;
despecInfo.MaxHeight = 3;
despecInfo.ConnectivityType = 8;

// Deline the source image into the destination image
nStat = ampDeSpec(pWSrc, pWDest, &despecInfo);
```

ampDeStreak

```
int ampDeStreak(pWSrcname, pWDestname, params)
```

```
    PWORKIMAGE pWSrcname;
    PWORKIMAGE pWDestname;
    ampDESTREAKINFO *params;
```

The binary image *pWSrcname* will be copied into image *pWDestname* removing streaks. The definition of a streak is controlled by the dereferenced *params* structure.

The fields of ampDESTREAKINFO are defined as follows (see interface reference file for latest definition).

```
    typedef struct of_AMPDESTREAKINFO {
          int MinWidth;
          int MaxWidth;
          int Color;
          // results output
          long TotalPixels;
          long TotalPixelsModified;
          long TotalRuns;
          long TotalRunsModified;
    } ampDESTREAKINFO;
```

MinWidth is the minimum horizontal pixel width of candidate streak objects. Objects narrower than this will be retained in the destination image.

MaxWidth is the maximum allowed pixel width for streaks.

Color specifies the color of the streak: black (1) or white (0).

TotalPixels reports back how many *Color* pixels are contained in the image.

TotalPixelsModified reports how many pixels were changed to erase the streak.

TotalRuns reports the number of *Color* runs contained in the image.

TotalRunsModified reports the total number of runs changed to remove the streak.

This API always returns 0 if the function was successful, non-zero otherwise.

Example:

```
    // Clear out input and output parameters
    memset (&destreakInfo, 0, sizeof (ampDESTREAKINFO));
    destreakInfo.MinWidth = 500;
    destreakInfo.MaxWidth = 1000;
        destreakInfo.Color = 1;
        // Destreak the source image into the destination image
        nStat = ampDeStreak(pWS, pWD, &destreakInfo);
```

ampFilterImage

```
int ampFilterImage ( pWSrcname, pWDestname,
            filter_type, sub_code, threshold )
```

```
PWORKIMAGE pWSrcname;
PWORKIMAGE pWDestname;
LPSTR filter_type;
LPSTR sub_code;
int threshold;
```

This API provides a wide set of powerful binary filter operators that can be used to enhance images, remove backgrounds, reduce noise, etc. Each filter type has a set of sub-codes that further define the filter operation. The image data in *srcname* will be filtered and written to image *destname*.

Filter_type defines the class of filter operation to perform. The following filters are available:

majority	A directional majority filter that can preserve certain structures within the image.
erode	An erosion filter useful for thinning image elements
dilate	A dilation filter useful for fattening image elements.
spot	A spot removal (de-speckel) filter.

For erode, dilate, and spot filters, the *sub-code* can be given as '**weak**' (4-neighbor) or '**strong**' (8-neighbor) to control the strength of the effect.

4-neighbor (or 4-connected) means the adjacent pixels to the North, East, South, and West. 8-neighbor, (or 8-connected), means any adjacent pixel in a surrounding 3x3 box.

For the majority filter, a variety of *sub-code* values can be used.

Note: the majority filters, which control preservation of lines, act only on single pixel width lines. These filters can be useful in removing fine line screens, as appear in some negotiable documents, but they are *not* generalized line removal functions.

Filter Type	Sub Code	Definition
ERODE	WEAK	4- neighbor erosion. Can be used to lighted lines. If any of 4-connected neighbors are white, the pixel will be set white. This filter can remove single pixel lines.
ERODE	STRONG	8- neighbor erosion. A more aggressive filter than 4-neighbor, this will set the pixel white if any of the eight-connected neighbors are white.
DILATE	WEAK	4- neighbor dilation. This filter darkens an object by setting a pixel black if any of its 4-connected neighbors are black.
DILATE	STRONG	8- neighbor dilation The filter darkens an object by setting a pixel black if any of its 8 neighbors are black.
SPOT	WEAK	4x4 neighbor spot removal. A more aggressive spot removal filter. It will remove any black pixels in a 2x2 region if all the pixels in the surrounding 4x4 region are white.
SPOT	STRONG	6x6- neighbor spot removal. An even more aggressive spot removal filter. It will remove any black pixels in a 2x2 region if all the pixels in the surrounding 6x6 region are white.
SPOT	SINGLE	Single pixel spot removal. If all 8 neighbors of a pixel are white, the pixel is set to white.
MAJORITY	NORMAL	Standard majority filter. The number of black pixels in a 3x3 region is compared to a threshold. If the count >= the threshold, sets the center pixel to black, else sets the center pixel to white. Preserves features in image.
MAJORITY	ERODE	A weighted erosion filter. The number of white pixels in a 3x3 region is compared to a threshold. If the count >= the threshold, sets the center pixel to white. Preserves features in image.
MAJORITY	DILATE	A weighted dilation filter. The number of black pixels in a 3x3 region is compared to a threshold. If the count >= the threshold, sets the center pixel to black. Preserves features in image.
MAJORITY	SPUR	Removes single-pixel growths from vertical line edges
MAJORITY	BUMP1	Removes one pixel growths from vertical line edges
MAJORITY	BUMP2	Removes two pixel growths from vertical line edges

MAJORITY	NORMAL_NP	A weighted filter that does not try to preserve single pixel lines in horizontal, vertical, and diagonal directions.
MAJORITY	NORMAL_NPH	Like NORMAL, except that it preserves all but horizontal lines.
MAJORITY	NORMAL_NPV	Like NORMAL, except that it preserves all but vertical lines.
MAJORITY	NORMAL_NPD	Like NORMAL, except that it preserves all but diagonal lines.
MAJORITY	NORMAL_NPNE	Like NORMAL, except that it preserves all but NorthEast to southwest diagonal lines.
MAJORITY	NORMAL_NPNW	Like NORMAL, except that it preserves all but Northwest to Southeast diagonal lines.
MAJORITY	ERODE_NPH	Like ERODE, except that it preserves all but horizontal lines.
MAJORITY	ERODE_NPV	Like ERODE, except that it preserves all but vertical lines.
MAJORITY	ERODE_NPD	Like ERODE, except that it preserves all but diagonal lines.
MAJORITY	ERODE_NPNE	Like ERODE, except that it preserves all but Northeast to Southwest diagonal lines.
MAJORITY	ERODE_NPNW	Like ERODE, except that it preserves all but Northwest to Southeast diagonal lines.
MAJORITY	DILATE_NPH	Like DILATE, except that it preserves all but horizontal lines.
MAJORITY	DILATE_NPV	Like DILATE, except that it preserves all but vertical lines.
MAJORITY	DILATE_NPD	Like DILATE, except that it preserves all but diagonal lines.
MAJORITY	DILATE_NPNE	Like DILATE, except that it preserves all but Northeast to Southwest diagonal lines.
MAJORITY	DILATE_NPNW	Like DILATE, except that it preserves all but Northwest to Southeast diagonal lines.

Threshold is used only for the majority filter, and ranges from 0 to 9. A threshold value of 5 is considered neutral. Lower values will give darker looking images; higher values will tend to lighten the image.

This API always returns 0 if the function was successful, non-zero otherwise.

Examples:

```
    ampFilterImage(pWSrc, pWDest, "majority",
          "erode_npd", 5);

    ampFilterImage(pWSrc, pWDest, "dilate",
          "weak", 0);
```

The first example filters image 'old' into image 'new', using a majority filter with an erosion algorithm that deletes diagonal lines. The second example runs a weak (4-neighbor) dilation filter.

Background Removal Filter

The FilterType Major has three special subfilters: 200, 201 and 202. These filters will remove the background from a large class of images and leave the forground (text) information. Each one will also accept a threshold value to increase the degree of removal. The subfilters 200 and 202 also perform a test to determine if background noise is present before running the removal filter. If not present no filter operation is performed. The subfilter 200 uses the lower 5/8" of the image to determine background (MICR line on a check). The subfilter 202 uses the middle 1/3rd of the image to determine background (CAR/LAR region on a check). The subfilter 201 will perform the removal filter regardless as no test is performed.

The strength or amount of background removal is determined by the threshold value from 1 to 9 with 9 being the most aggressive background removal and 1 the least. When used for preparing an image for OCR, the aggressive value of 8 is often used. When the same image is being prepared for printing, the value of 6 is often used. This will leave some noise on the image but will not impact the small text content of the image.

Example:

```
ampFilterImage(pWSrc, pWDest, "major",
      "200", 9);
```

Miscellaneous Functions

This group of functions provide additional capabilities and programming tools. Functions that begin with *ampGray* may only accept grayscale images as inputs and/or outputs.

ampAssembleMICR

```
int ampAssembleMICR (PSTR pszAuxOnUs, PSTR
            pszEPC, PSTR pszRoute, PSTR
            pszOnUs, PSTR pszAmount,
            PSTR pszTranslation, PSTR
            pszOutputMICR)
```

```
PWORKIMAGE  pW;
PSTR pszAuxOnUs ;
PSTR pszEPC ;
PSTR pszRoute;
PSTR pszOnUs ;
PSTR pszAmount;
PSTR pszTranslation ;
PSTR pszOutputMICR ;
```

This function provides a method to assemble a single MICR line from the fields found on a check image. If the pszTranslation parameter is not NULL, it will also translate the control to the users preference.

Accept strings representing the Aux On Us, EPC, Route, On Us, and Amount data and construct an output MICR string in a traditional format using the special characters A,B,C,D, and -. If a translation table is provided, the output characters will be translated. If input strings are not present or are some way in error, the assembly process will try to continue. Route data must be present and consist of 9 or 11 characters. An embedded dash in the route data is acceptable.

The MICR output will be in the following left-to-right format:

Aux On Us (max length 17 characters)
Space (1 character if Aux On Us data present)
EPC (max length 1 character)
Routing (fixed length 11 characters)
On Us (max length 20 characters)
Space (1 character if input Amount data present)
Amount field (fixed length 12 characters if present)

pszAuxOnUs - If non null, the pointer to the input Aux On Us string
Numeric data must be represented by the characters 0-9 and the On Us symbol by / or c or C.

pszEPC - If non null, the pointer to the input EPC string
Numeric data must be represented by the characters 0-9.

pszRoute - The pointer to the input Route string

Numeric data must be represented by the characters 0-9 and the optional Route character by a or A. The string length must be 9 (no route characters) or 11 (beginning and end route character present).

pszOnUs - If non null, the pointer to the input On Us string

Numeric data must be represented by the characters 0-9 and the On Us symbol by / or c or C.

pszAmount - If non null, the pointer to the input Amount string

Numeric data must be represented by the characters 0-9 and the optional amount character by b or B if the amount characters are present.

szTranslation - ASCII character translation string -

If present all assembled MICR output is translated through this table.

Output Parameters:

pszOutputMICR - If non null, the pointer to the output string that will contains all the MICR fields in Aux OnUs, EPC, Route, OnUs, Amount order.

Return Code - 0 if there were no problems parsing the input string. Otherwise the number returned is an AMPLIB error code - typically ampERR_INVARG (47)

ampCheckImageQuality

```
int ampCheckImageQuality( pW, pQInfo
```

`PWORKIMAGE pW;`
`ampQUALITYINFO *pQInfo;`

This API performs a series of image quality measurements on the input bilevel image in order to determine its suitability for incorporation as a check front or back image in a .937 format image cash letter file. Financial institutions have guidelines that establish acceptable limits for check image sizes, G4 compression size, overall check darkness, etc. The ampQUALITYINFO structure serves as a medium for passing input parameters to the function and for passing output parameters back out.

The ampQUALITYINFO structure is shown below.

```
typedef struct of_ampQUALITYINFO {
     // Input Parameters
     int nCornerSampleXSize;
     int nCornerSampleYSize;
     int nBlackBorders;
     int nStreakDensity;
     // Input/Output Parameters
     Int nXPixelSize;
     int nYPixelSize;
     // Output Parameters
     int nXRes;
     int nYRes;
     DWORD dwTotalG4Length;
     DWORD dw3x3Count;
     DWORD dwBlobSites;
     DWORD dwStreakCount;
     BLOBDATA blobLoc[MAXBLOBCOUNT
     BLOBDATA streakLoc[MAXSTREAKCOUNT];
     DWORD dwULPeakRun;
     DWORD dwURPeakRun;
     DWORD dwLLPeakRun;
     DWORD dwLRPeakRun;
     DWORD dwPeakRunLength;
     DWORD dwRunOneCount;
     DWORD dwRunTwoCount;
     DWORD dwSpare1;
     double dblAvgRunLength;
     double dblBlackDensity;
     double dblTotalSkew;
     double dblULAvgRun;
     double dblULBlack;
     double dblURAvgRun;
     double dblURBlack;
     double dblLLAvgRun;
     double dblLLBlack;
     double dblLRAvgRun;
     double dblLRBlack;
} ampQUALITYINFO;
```

The BLOBDATA structure used to define two arrays has the following format:

```
typedef struct blobdata {
     int x ;
     int y ;
     int dx ;
     int dy ;
     }      BLOBDATA;
```

The blobLoc array has 8 elements and the streakLoc array has 10 as defined by the following constants.

```
#define MAXBLOBCOUNT 8
```

```
#define MAXSTREAKCOUNT 10
```

INPUTS - Before calling ampCheckImageQuality, the following input variables need to be defined.

nCornerSampleXSize

Width of rectangle used to measure torn corners - in .001 of inch –

250 is 1/4 inch. A good typical value for front and back check images is 750 or 3/4 inch.

nCornerSampleYSize

Height of rectangle used to measure torn corners - in .001 of inch 250 is 1/4 inch. A good typical value for front and back check images is 750 or 3/4 inch.

nBlackBorders

Set to 1 if check borders are black

nStreakDensity

Percentage of black x10 needed on a raster to make it part of a streak 990 (99.0%) is a good value

nXPixelSize

As an input this value specifies the Speckle Width (negative or 0 defaults to 3) of image noise shapes in pixels. A good typical value for check images is 4.

nYPixelSize

As an input this value specifies Speckle Height (negative or 0 defaults to 3) of image noise shapes in pixels. A good typical value for check images is 5.

OUTPUTS – After a successful call to ampCheckImageQuality, these are the result output values. The valid value ranges listed are taken from ECCHO guidelines.

nXPixelSize

As an output this value specifies the pixel width of the image and can be used with nXRes to calculate the horizontal width of the check image in inches. Valid widths for front check images range from 5.5 to 9.4 inches. The maximum acceptable width difference between a check front and back side is 0.5 inches.

nYPixelSize

As an output this value specifies the height of the document in pixels and can be used with nYRes to calculate the vertical height of the check image in inches. Valid heights for front check images range from 2.2 to 4.8 inches. The maximum acceptable height difference between a check front and back side is 0.6 inches.

nXRes

Horizontal resolution in Dots per Inch (DPI). Dots are the same as pixels in this definition.

nYRes

Vertical resolution in Dots per Inch (DPI). Dots are the same as pixels in this definition.

dwTotalG4Length

Length in bytes of image when compressed to G4. Check front side valid byte counts range from 414 to 130,000. Check back side byte counts range from 414 to 100,000.

dwSpeckleCount

Number of speckle objects - default size is 3x3. Speckle density is calculated using this number divided by the check pixel area (nXPixelSize * nYPixelSize) / (nXres * nYres). Good typical values of speckle density for a check front side are below 42 speckles per square inch. Back side speckle densities should be below 466.

dwBlobSites

Locations of blobs at edges in compass format. The lower 8 bits in this value are each assigned a compass ordinal. If a bit value is is nonzero, then the blob can be found in the blobloc array at an index that matches the bit location.
North is the LSB or 1, North East is 2, East is 4, South East is 8, South is 16, South West is 32, West is 64, and North West is 128.

dwStreakCount

Number of horizontal streaks. If this number is greater than 0, the first 10 streak locations and sizes are listed in the streakloc array.

blobloc()

This array of 8 structures contains the X,Y, DX, DY values of the blob locations in the image. Each one of these values is a 32-bit integer (4 bytes). The indices of this array (0-7) represent compass settings starting at North (0), North East (1), East (2), South East (3), South (4), South West (5), West (6), and ending at North West (7).

streakloc()

This array of 10 structures contains the X,Y, DX, DY values of the streak locations in the image. Each one of these values is a 32-bit integer (4 bytes)

dwULPeakRun

Upper left corner dominant runlength

dwURPeakRun

Upper right corner dominant runlength

dwLLPeakRun

Lower left corner dominant runlength

dwLRPeakRun

Lower right corner dominant runlength

dwPeakRunLength

Dominant runlength in image

dwRunOneCount

Number of runs of 1 pixel

dwRunTwoCount

Number of runs of 2 pixels

dwSpare1

Places double variables on 8 byte boundary

dblAvgRunLength
: Average runlength (total black pixels / total number of runs)

dblBlackDensity
: Average blackness of all pixels (total black / area) which must be less than 1.0. Acceptable darkness values for the front of a check image range from 2.1 to 39 percent. For check back sides the range is from 0.0 to 39 percent.

dblTotalSkew
: Skew of image as measured by the ampMicrPrep function in degrees – Positive is CCW – a typical unacceptable skew for a check image is 8.8 degrees or more.

dblULAvgRun
: Upper left corner average runlength (black pixels / number of runs).

dblULBlack
: Upper left corner black density (total black / area)

dblURAvgRun
: Upper right corner average runlength (black pixels / number of runs).

dblURBlack
: Upper right corner black density (total black / area)

dblLLAvgRun
: Lower left corner average runlength (black pixels / number of runs).

dblLLBlack
: Lower left corner black density (total black / area)

dblLRAvgRun
: Lower right corner average runlength (black pixels / number of runs).

dblLRBlack
: Lower right corner black density (total black / area)

The normal status return for successful completion of the ampCheckImageQuality function is the value 0.

ampGetImageAddress

```
int ampGetImageAddress( pW, lpAddr )
```

```
PWORKIMAGE  pW;
LPDWORD lpAddr;
```

This function is provided for special OEM use only. It returns the bitmap image address of the image *pW* via the argument *lpAddr*. Grayscale images are stored as an 8-bit grayscale DIB. Color images are stored as a 32-bit DIB with the color components for each pixel sequenced as bytes of blue, green, and red followed by an unused byte.

ampGetImageBlock

```
int ampGetImageBlock( pW, dx, dy, x, y, idata)
```

```
    PWORKIMAGE  pW;
    long   dx, dy;
    long   x, y;
    WORD  *idata
```

This function transfers a rectangular portion of image *pW* to the PC memory buffer *idata,* which must have been previously allocated and be large enough. Only image data from the currently defined sub-image will be transferred. DX and DY give the number of pixels and lines, respectively, to transfer. X and Y are relative to the image origin (0,0).

ampGetImageInfo

```
int ampGetImageInfo( pW, lpCtype, lpFtype,
            lpBBrev)
```

```
    PWORKIMAGE pW;
    LPSTR lpCtype;
    LPSTR lpFtype;
    LPWORD lpBBrev;
```

This API can be called to learn certain facts about an image that was loaded by **ampLoadImage** or **ampLoadImageHnd**.

lpCtype returns a string, as described at the top of this section, which identifies the compression type that was used to decompress this image. This string must be allocated by the caller, and must be large enough to hold the type description.

lpFtype returns a string, as described at the top of this section, which identifies the file organization type of the file that was loaded into this image. This string must be allocated by the caller, and must be large enough to hold the type description.

lpBBrev is a Boolean flag; TRUE means the bit-byte-reversal option was needed to load the image.

This API always returns 0 if the function was successful, non-zero otherwise.

ampGetLicenseInfo

```
void ampGetLicenseInfo( expdate, licdata)
```

```
time_t  *expdate;
BOOL    *licdata;
```

This function will return the expiration date for this machines AMPLIB license file, and an array of 32 Boolean values, which identify which features are licensed. A permanent(non expiring) license and no license will return **licdata** of -1. The license vector for unlicensed machines will be all False.

The current assignments match the format of the license file and are:

licdata[0]	:	E13B
licdata[1]	:	reserved
licdata[2]	:	Speed Class 1
licdata[3]	:	Speed Class 2
licdata[4]	:	Speed Class 3
licdata[5]	:	Speed Class 4
licdata[6]	:	Speed Class 5
licdata[7]	:	Image Repair
licdata[8]	:	MICR Verify
licdata[9]	:	MICR Batch
licdata[10]	:	MICR Parse
licdata[11]	:	Barcode 39
licdata[12]	:	Barcode 1D
licdata[13]	:	Barcode PDF 417
licdata[14]	:	AmpLib
licdata[15]	:	Barcode Data Matrix
licdata[16]	:	Mobile MICR
licdata[17]	:	Special bar codes (Airline, Bar Code 32)/Click
licdata[18]	:	XipPrint
licdata[19]	:	reserved
. . . .		
licdata[24]	:	reserved
licdata[25]	:	Sobel
licdata[26]	:	reserved
	. . . .	
licdata[31]	:	reserved

The licensed barcode symbologies allowed under the Barcode 1D are:

BC_3of9	Code 3 of 9	
BC_CODABAR	CODABAR	
BC_I2of5	Interleaved 2 of 5	25 limited
BC_A2of5	Airline 2 of 5	25 limited
BC_128	Code 128	
BC_UCC128	UCC Code 128	
BC_2of5	Code 2 of 5	25 limited

BC_MAT2of5	MAT 2 of 5	25 limited
BC_93	Code 93	
BC_UPC_A	UPC-A	
BC_UPC_E	UPC-E	
BC_EAN_13	EAN-13	
BC_EAN_8	EAN-8	
BC_POSTNET	Postnet	Standalone
BC_PATCH	Patch Code	Standalone
BC-PLANET	US Post Office code	Standalone
BC_39_NOSS	Code 3 of 9 without start/stop code	Standalone
BC_BCC32	Bar Code 32 (Pharmacy)	Standalone
BC_39_EXT	Code 39 Extension	Standalone

The licensed barcode symbologies allowed under the Barcode PDF 417 are:

BC_PDF417	PDF-417 2D code	Standalone
BC_4STATE	4-State (US Intelligent Mail, UK Royal Post)	Standalone
BC_PATCH	Patch Code	Standalone

The licensed barcode symbologies allowed under the Barcode Data Matrix are:

BC_DMATRIX	Data Matrix	Standalone
BC_QR	Quick Response Code	Standalone

ampGetMessageText

```
void ampGetMessageText( ecode, lpMsg)
```

```
    int   ecode;
    LPSTR  lpMsg;
```

This function will fill a string buffer with the message text string corresponding to the error code *ecode*. The string buffer should be pre-allocated by the caller and be at least 128 bytes long.

ampGetFileVersion

```
void ampGetFileVersion(
            szName,pnMajor1,pnMajor2,pnMinor1,
            pnMinor2)
```

```
LPSTR  szName;
PINT  pnMajor1;
PINT  pnMajor2;
PINT  pnMinor1;
PINT  pnMinor2;
```

Retrieve the amplib.dll file version information. The string will contain the text content from the file. The four parameters contain numeric values representing two major and two minor versioning levels. In releases after 6.1.2.0 the pMajor2 value will be odd for single threaded DLLS and even for multithreaded DLLs.

ampGrayGetImageBlock

```
int ampGrayGetImageBlock( pW, dx, dy, x, y,
            pbyBlock )
```

```
PWORKIMAGE  pW;
long   dx, dy;
long   x, y;
PBYTE  pbyData
```

This function transfers a rectangular portion of the grayscale image *pW* to the PC memory buffer pointed to by *pbyData*, which must have been previously allocated and be large enough. Only image data from the currently defined sub-image will be transferred. DX and DY give the number of pixels and lines, respectively, to transfer. X and Y are relative to the image origin (0,0). Color images are not supported.

ampGrayPutImageBlock

```
int ampGrayPutImageBlock( pW, dx, dy, x, y,
            pbyData)
```

```
    PWORKIMAGE  pW;
    long   dx, dy;
    long   x, y;
    PBYTE   pbyData
```

This transfers a rectangular block of image data from the buffer pointed to by *pbyData* on the PC to the image *pW*. Image data will be written to the currently defined sub-image. DX and DY give the number of pixels and lines, respectively, to transfer. X and Y are relative to the image origin (0,0). Color images are not supported.

ampPutImageBlock

```
int ampPutImageBlock( pW, dx, dy, x, y, idata)
```

```
    PWORKIMAGE  pW;
    long   dx, dy;
    long   x, y;
    WORD   *idata
```

This transfers a rectangular block of image data from buffer *idata* on the PC to the image *pW*. Image data will be written to the currently defined sub-image. DX and DY give the number of pixels and lines, respectively, to transfer. X and Y are relative to the image origin (0,0).

ampTrace

```
void ampTrace( formatstring, …)
```

```
    LPSTR   formatstring;
```

This function takes the same form as the C language printf function, and will write a line of information to the trace log. You should include newline character at the end of the line.

E.g.,

```
    ampTrace(“function call returned %d\n”, rc);
```

ampTraceEnable

```
void ampTraceEnable( enabled, szFileName )
```

```
    BOOL   enabled;
    LPSTR  szFileName;
```

Allows programmatic enabling and disabling of the AMPLIB trace log, which is useful in debugging. All My Papers Customer Support personnel may request this log from you. Use of this function around an area of code in question might make the file smaller and easier to examine. Trace logs are not erased within AMPLIB, so it may be useful to erase the file before debugging for the sake of clarity. There is no real limit to the number of unique trace files that may be used in one debugging session.

Example:

```
    ampTraceEnable( TRUE, &szAmpTraceFile[0]);
```

Appendix A

AMPLIB Based Application Programs

MICR BATCH

This application is available as an option. The application is written in C++ in a Visual C++ environment. MICR BATCH will process a complete directory of check image files and generate an ASCII text file as output. The application provides control of the AMPLIB MICR reader DLL in a ready to use application.

AMPTEST

This application is written in Delphi and demonstrates the use of image-processing and MICR functions. Amptest uses the AmpLib DLLs without the need for a license since it is an example of an OEM program, which will run on any machine without individual licensing.

Appendix B

AMPLIB Error Codes

1	Could not allocate PC memory space. A local or global allocation failed that was needed to complete the requested operation.
3	Specified work image does not exist. No image by the given name can be located.
4	Name already in use.
6	Not a primary image. An alias image may not be used in this instance.
10	AMPLIB cannot support any more tasks. The maximum number of callers has already been reached.
11	Internal error. A software error has been detected in the AMPLIB system. Please report this to AllMyPapers Technical Support.
12	Image bounds exceeded. The requested DX, DY, X, Y values exceed the values allowed for this image, as given by MaxHeight and MaxWidth, or the requested sub-image lies outside of the current image dimensions.
13	Image metrics error. The requested sub-image lies outside of the current image dimensions.
14	Internal error calling the Windows API.
15	Bad handle passed to function. The given handle is incorrect or inappropriate for the function in question.
16	User interrupt. A function terminated because of an improper call.
19	AMP function call error. There is an error in the arguments passed to the function in question.
20	No size information. The image has not yet been loaded with any image data and thus has no dimensions.
21	No cross-board operations are allowed. You may not perform an operation where the source and destination image operands reside on different co-processors.
22	Incompatible image sizes. When a destination image is fixed size, the result image must be less than or equal to the size of the destination.
23	Bad file name. The file path name given is incorrect or cannot be opened.
24	I/O error. The I/O system reported an error during execution of this function.
25	Cannot open trace file.
26	An invalid compression type was given.
27	An internal TIFF operation failed. In the processing of the IFD list or header, some critical operation failed.
28	Required TIFF tag missing. The TIFF 6.0 Specification defines those tags which at a minimum must be present in all baseline TIFF files. One of those tags is missing.

29	Image organization not supported. Only 1 bit per pixel bi-level images are supported.
30	This system is unable to run AMPLIB. Call AllMyPapers Technical Support.
31	Unable to open the requested TIFF file. It may not be a TIFF file, or has an invalid header.
32	The requested image within a multi-image TIFF file is not in the image file directory of that TIFF file.
33	An error occurred while reading the TIFF IFD.
34	The KDY value given for Group 3 2d compression is invalid.
35	Assertion logic error. Some internal software data or pointer consistency has occurred.
36	No region has been selected to support the requested operation.
37	The number passed to the function is out of range.
40	The resolution value given is not valid.
41	The page size value given is not valid.
42	The operation type is not valid.
43	The mode given is not valid.
46	The scale ratio given is not valid for this operation.
47	One of the arguments passed to the function is invalid.
48	AMPLIB is unable to create the requested file. This is most likely due to an invalid path or some I/O permission error.
49	The margins are not legal for the page size.
51	No file specification was given and is required for this operation.
52	No index string was found in the file path name string.
53	Huge objects not supported yet.
54	The clipboard is empty.
55	General error. No detail available.
56	Download failure.
60	General printer failure.
62	A bad tag was found in a TIFF file.
64	An invalid TIFF header was detected.
65	Scaling while printing requires buffered print mode.
66	Source and destination images must be different.
67	The function in question timed out.
68	A callback function returned an error.
69	Application lockout.
70	This version of AMPLIB is not correct for this application.
71	An invalid file type was specified.
72	The image IX value must be a multiple of 32 for this operation.
73	The margins specified are not legal for this operation.
74	The requested TIFF tag already exists in the IFD list.
75	An invalid Optika header was detected.
76	The requested file format is unsupported in this mode.
83	No ensigns defined or allowed.
84	Bad MODCA RECID parameter
85	IBM MMR format not supported

86	Unsupported compression type
87	Decompression error
88	Unsupported MODCA or IOCA file format
89	Compression error
90	Thread already attached to DLL
91	Disk is full
92	File Access error
93	Too many files open
94	File exists
95	Bad file handle
96	No such file or directory
98	Thread not attached
101	No object data in image block
102	Can't find a needed DLL
103	Can't find entry point in DLL
104	License file fails security check
105	License check detected date rollback
106	License expired
107	License required for this feature
108	Image degenerated to dx=0 or dy=0
113	Software implementation only
114	Not an AmpLib PDF file
115	Error parsing PDF file
116	Missing files/files not loaded
117	License computer id error
118	Problem opening license file
119	Problem opening TWAIN device
120	Problem reading paper sensor on TWAIN device
121	Scanner Timeout
122	Not supported scanner
123	No image acquired while scanning
124	Failure during image warp
125	Failure during Data Matrix read
126	Invalid JBIG header
127	JBIG decompression problem
128	Failure while rotating image
129	CLICK count exhausted
130	No image content(all black/white or nearly so)
161	Failure during Quick Response barcode read
201	Multi Thread error-- No pointer to the TLS available

Appendix C

AMPLIB License Manager

Overview

AMPLIB and all applications based on it are controlled via an Internet-connected license manager (LM). If a valid license is not detected by AMPLIB, ampReadMICR and ampPrepMICR calls will return with error code 107. The LM provides the tools necessary to create license files, and to register your software and contact information with AllMyPapers. For special needs, the AMPLIB license may also be coded in a parallel port security key (dongle) rather than in license files.

Modes of Operation

The LM user interface is comprised of 3 panels: On-line Registration, Off-line Registration, and Show Licenses. These 3 panels correspond to the three modes of operation, for Internet-connected registration, off-line registration for non-connected users or special circumstances, and displaying the currently installed licenses.

- On-line registration, either directly or via proxy file is the preferred method. This method uses an Internet Registration Server to register your software and generate your license keys.
- Off-line registration is used when only voice phone or fax communication is possible.
- Show Licenses is used to display the installed licenses on your workstation.

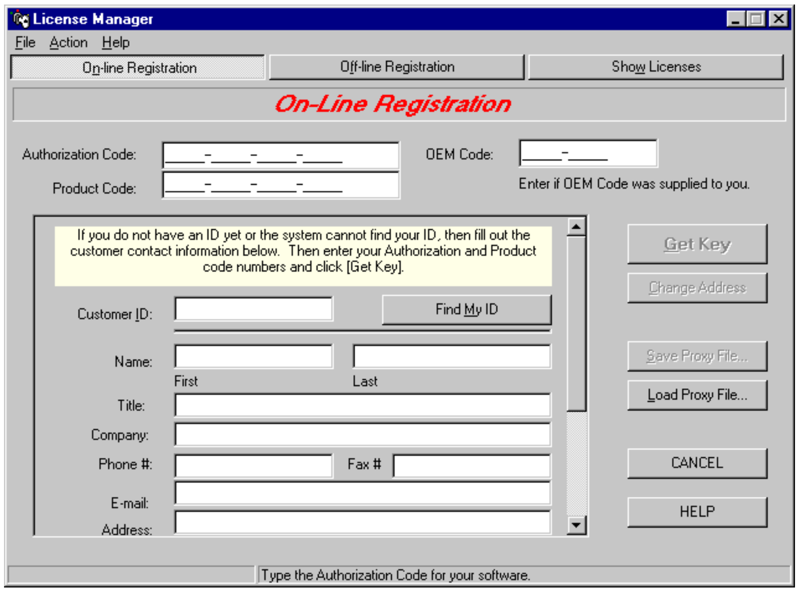
On-line Registration

This panel is used to perform user registration and license key activation through an active Internet connection on the user workstation. This can be performed 24 hours a day, 7 days a week. To use this capability, the user workstation must be connected to the Internet, and the user must have a set of license key codes to register. If the Internet connection is provided through a LAN connection, then there is no special setup. If the workstation is connected via dial-up networking and a modem, the connection should be made before starting the LM.

NOTE:

If you have no Internet connection, or you have a firewall or other device that obstructs communication with the License Server, then use the Proxy File method, discussed below.

The LM On-Line Registration panel is shown below.



If you do not have an ID yet or the system cannot find your ID, then fill out the customer contact information below. Then enter your Authorization and Product code numbers and click [Get Key].

Type the Authorization Code for your software.

There are a number of fields that need to be entered.

License Keys

When you purchase licensed software, you are issued a pair of numbers for each licensed product. (You may also receive numbers for evaluation, which generate temporary licenses.) These numbers may be printed on a report, shipper, or a stick on label. Below is an example of the report form.

```
Date: 5/11/99 2:52:43 PM


LICENSE TYPE : AMPLIB
TIME PERIOD  : Permanent
FEATURES:
   All 1-d Bar Codes
   PDF-417 2-d Bar Codes
   Image Transform
   Image Processing
   Image Filters
   LEVEL UNCHANGED


Authorization Code      Product Code      OEM Code
------------------  -------------------    ---------
332D-1234-1234-B629  AEB4-1234-1234-ACB2
```

These two numbers uniquely identify your purchase and what features are included with your purchase. They are entered in the corresponding fields at the top of the form. If you were not issued an OEM code with this number pair, leave that field blank.

Before you can retrieve your license key from the server, you must enter your contact information. The first time you run this program on a machine, the Customer ID field will come up blank.

If you have never registered before with this program, leave the Customer ID field blank. Enter all the name and address information requested of the program. When all fields are filled in, the [GetKey] button will be enabled. Click it to process your license.

Note that some field length checking is performed on the contact information fields. The following fields must have at least the given number of characters. Pad the field with blanks or other characters if the GetKey or SaveProxy buttons are not enabling.

FirstName > 0

LastName > 1

Company > 2

Email > 6

Phone > 9

Addr1 > 1

City > 1

State > 1

ZIP > 4

If you have previously used the program's on-line registration to activate a key, the Customer ID field will be filled in with your customer number. Click the [GetKey] button to process your license.

If you think you have previously registered, but the Customer ID field came up blank (this might be caused by re-installing Windows) , click the [Find My ID] button to locate your customer record on the server. If a record for your machine is found, the program will ask you to verify your identity. If you click "yes", your contact information and Customer ID fields will be filled in. Then click the [GetKey] button to process your license.

If the numbers are correctly processed, you will get a message box stating the license key has been processed.

If you get an error stating the program timed out waiting for the server, you may have a problem with your Internet connection. Try the operation again, in case the server was busy or the message was lost. If your site is protected by a firewall, ask your network administrator to make sure UDP packets are not screened out.

If you get an error message stating the keys could not be processed, double check they are entered correctly, and click [GetKey] again after correcting them. For problems you cannot solve, call Customer Support.

Changing Your Address

If you are already registered, but want to record a new address or phone number with the registration server, click the [Change Address] button.

This will enable your address information for editing. Make your changes and then click [Submit].

Proxy Files

If your workstation is not connected to the Internet, but you have access to another that is, you can “register by proxy.” If you have a firewall, proxy server or other packet filtering device installed that obstructs communication with the License Server, you can also use the Proxy File method to perform your registration.

(Note that the [Save Proxy File] button, just like the [GetKey] button, are not enabled until all the required fields have been entered.)

With Proxy files, you enter the registration data and license codes as you normally would, but instead of clicking [GetKey], you click the [Save Proxy File…] button. This command will save information needed for registering your software to a file, perhaps on a floppy disk.

Next, take that file to another workstation that *is* connected to the Internet, and has LM installed on it. Go to the On-line registration panel and click the [Load Proxy File…] button. This will cause the LM to communicate with the server to retrieve the activation key. If it is successful, the proxy file will be updated with this key information.

If you have no such workstation that can perform the Proxy file load, e-mail the proxy file to Customer Support. They will process the file and return it to you.

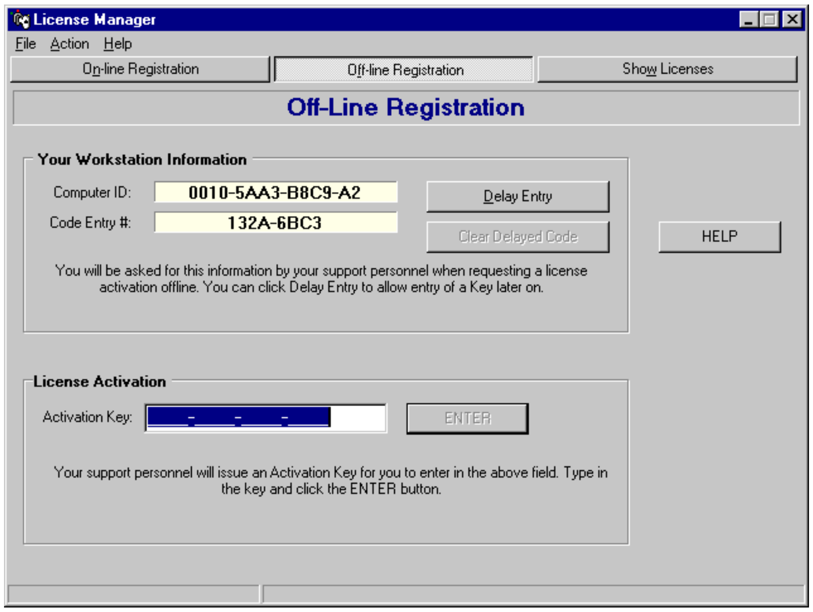
Now, return to the workstation to be licensed, and click [Load Proxy File…], this time using the file that was updated by the other machine. This action will then install the license on your workstation. Your registration information will have been sent to the server by the other machine.

Site License Administrator

If your site has more than one machine to license, you can speed up the process by entering the same Customer ID number for each machine. Once you have been issued an ID by the first registration process, remember that ID and use it for each other machine. That will establish one owner for all the machines recorded on the server, and reduce the typing needed for each machine.

Off-Line Registration

When no Internet connection is available, or when special circumstances dictate, you can contact Customer Support. They will direct you to the Off-line Registration panel, shown below.



Computer ID: 0010-5AA3-B8C9-A2

Code Entry #: 132A-6BC3

You will be asked for this information by your support personnel when requesting a license activation offline. You can click Delay Entry to allow entry of a Key later on.

Your support personnel will issue an Activation Key for you to enter in the above field. Type in the key and click the ENTER button.

You might be asked to use this form if:

- You set the PC's date/time backward beyond the time you last ran a licensed application.
- You change your machine configuration such that your Computer ID changes and your license becomes invalidated.
- You have requested an extension of a temporary license.

(Alternatively, Customer Support might issue you a new Authorization Code pair to accomplish the same thing.)

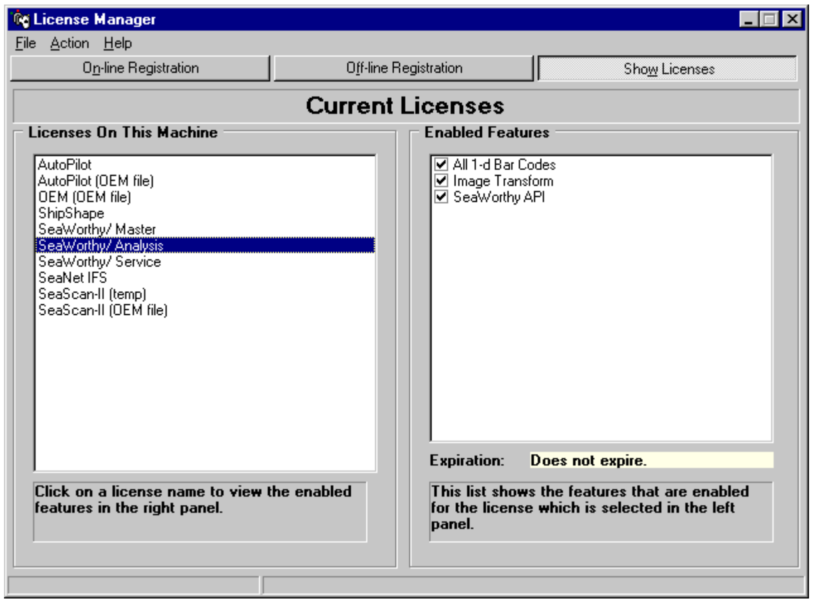
There is only one entry field on this form. You will be asked for your Workstation Information, shown at the top of the form. Customer Support will then give you an Activation Key to enter. After you type it, click the [ENTER] button to process it. A message box will indicate either success or failure.

If you cannot be in a real-time phone conversation with Customer Support, write down the Computer I.D. and Code Entry number, and click [Delay Entry]. This will record session information to be used again next time you run the program. Communicate these numbers via phone, fax, e-mail, or whatever to Customer Support along with your request. They will then respond with an Activation Key. Re-start the program, return to this panel, and enter the Activation Key. Click [ENTER] to process it.

If you decide later that you do not wish to proceed with this "delayed" action, click the [Clear Delayed Code] button to return to a normal state.

Show Licenses

This panel displays the licenses that are installed in the machine. A sample screen is shown below.



Current Licenses

Licenses On This Machine

- AutoPilot
- AutoPilot (OEM file)
- OEM (OEM file)
- ShipShape
- SeaWorthy/ Master
- SeaWorthy/ Analysis
- SeaWorthy/ Service
- SeaNet IFS
- SeaScan-II (temp)
- SeaScan-II (OEM file)

Click on a license name to view the enabled features in the right panel.

Enabled Features

- ✓ All 1-d Bar Codes
- ✓ Image Transform
- ✓ SeaWorthy API

Expiration: Does not expire.

This list shows the features that are enabled for the license which is selected in the left panel.

The left pane displays each of the licenses that are present on the machine. As the licenses in the left pane are selected with the mouse or keyboard, the right pane shows what features are coded in that license and the expiration date.

Note there are three types of licenses that may be shown in the left pane.

1. Permanent Licenses
2. Temporary Licenses, shown as (temp)
3. OEM licenses, shown as (OEM file)

It is possible to have multiple types for the same product. When this is the case, they OR together in function.

OEM Licenses

OEM license files are stored in the \Windows\AMP\Licenses folder. They generally have the form “SIPC00nn”. These are only issued by special contract.

See Tech Bulletin 004 for information on using OEM licenses.

Appendix D

What is MICR?

Overview

Originally developed in the 1950s to allow computers to sort checks, MICR (Magnetic Ink Character Recognition) is a special type font designed to be machine readable.

The font most commonly used in the United States, Canada, U.K., Australia, Turkey, India, Mexico, Venezuela, Columbia and the Far East is called E-13B.

The "E" in E-13B means that this was the fifth font considered. The "13" means that the height, width, and stroke widths of each character are either 0.013" or a multiple of 0.013". The "B" means that this was the second revision of the font.

The E-13B font includes the numbers 0 through 9, plus four special symbols: the Amount, the On Us, the Routing and Transit, and the Dash.

There is another standard, called CMC7, which is used in France, Italy, Spain, other Mediterranean countries, and South America (except Venezuela and Columbia). This font set has 41 characters and is distinguished by the vertical bars within each character.

Special magnetic MICR readers used to be necessary to decode the characters, which were printed with special toner containing iron oxide. This technology was built on the existing knowledge of the time regarding magnetic tape reading.

The drawbacks to the magnetic system included the expense of setting up special MICR printers to imprint the checks, and the expense of buying complicated MICR readers to decode the magnetic signals. The magnetic readers were also easily confused by extraneous information that might overlap the characters, such as a long descenders on a customer's signature. Variations in the amount of iron oxide in the ink, inconsistencies in how it was laid on the paper, and differences in paper quality all can lower the accuracy of magnetic MICR readers.

Now, using the OCR technology available with the AMPLIB System, MICR characters can be printed with ordinary ink, and be recognized quickly and reliably, even in circumstances that would confound a magnetic reader.

MICR Character Set

The MICR characters are always printed on the bottom portion of checks, drafts, and other negotiable documents, in an area 5/8” from the bottom edge, called the “Clear Band.”

Starting at the right, the first field is called the Amount. This field is filled in by the bank of first deposit, and is always bracketed by the Amount Symbol.

To the left of this is the On Us field, normally containing the account number at the drawee bank. It may also contain the sequential check number.

Next toward the left is the Routing and Transit field, containing the check digit number, the drawee bank number, and the bank routing number. This field is always bracketed by the Transit Symbol.

Last on the left is the Auxiliary On Us field. On business checks it may contain the check serial number, as well as accounting control information specific to the account. This field is always bracketed by the On Us Symbol.

Appendix E

Bar Code Symbology Examples

Code 39 (3 of 9)

0 1 2 3 4 5 6 7

Code 39, also referred to as Code 3 of 9, is an alphanumeric, self-checking, variable length code that uses five black bars and four spaces to define a character. Three of the elements are wide and six are narrow.

Code 39 supports the following characters:

- 26 uppercase letters
- Ten digits
- Seven special characters (- . $ / + % and a space)
- Start/stop character (*)
- A Code 39 bar code consists of the following elements:
- Leading quiet zone
- Start character
- Data characters, with characters separated by an intercharacter gap
- Optional check character
- Stop character
- Trailing quiet zone

Discrete 2 of 5

0 1 2 3 4 5 6 7

Discrete 2 of 5 is a variable length, high-density, self-checking, numeric code that uses five black bars to define a character. Two of the bars are wide and three are narrow.

Each Linear 2 of 5 bar code consists of the following elements:

- Leading quiet zone
- Start character
- Data characters
- Optional check character
- Stop character
- Trailing quiet zone

Note: Interleaved 2 of 5 and Linear 2 of 5 bar codes are mutually incompatible. You cannot select both of these bar types together when specifying multiple bar code types.

Interleaved 2 of 5

0 1 2 3 4 5 6 7

Interleaved 2 of 5 is a variable length (must be a multiple of two), high-density, self-checking, numeric code that uses five black bars and five white bars to define a character. Two digits are encoded in every character; one in the black bars and one in the white bars. Two of the black bars and two of the white bars are wide. The other bars are narrow.

Each Interleaved 2 of 5 bar code contains an even number of characters, and consists of the following elements:

- Leading quiet zone
- Start character
- Data characters
- Optional check character
- Stop character
- Trailing quiet zone

Note that in some cases, text or other random patterns on an image can inadvertently be detected as Interleaved 2 of 5 bar codes. To avoid this, you should always use a check digit and set the minimum number of characters as large as possible.

CODABAR

A 0 1 2 3 4 5 6 7 B

Codabar is a self-checking, variable length bar code that can encode 16 data characters. It is used primarily for numeric data, but also encodes six special characters. Codabar is useful for encoding dollar and mathematical figures because a decimal point, plus sign, and minus sign can be encoded.

Codabar supports the following characters:

- Ten digits
- Six special characters ($: / . + -)
- Four different start/stop codes (A - D)
- A Codabar bar code consists of the following elements:
- Leading quiet zone
- Start character
- Data characters
- Stop character
- Trailing quiet zone

Codabar supports three character encoding schemes:

Ten digits (0-9) and the special characters $ and - (minus) are printed with one wide bar and one wide space. All other elements are narrow.

Four special characters (: / . +) are encoded with three wide bars and no wide spaces.

Four start/stop characters (a b c d) are encoded with one wide bar and two wide spaces.

UPC-A

0 12345 67890 5

The Universal Product Code (UPC) bar code version A is a fixed length (12 characters) bar code scheme designed to uniquely identify a product and its manufacturer. The first digit in a UPC-A symbol is the number system digit, the next ten digits are data characters, and the last digit is the checksum. This is the standard bar code scheme for items of sale to the public. Note that supplementals are not supported.

A UPC-A bar code consists of the following elements:

- Left guard pattern
- Number system digit (encoded in odd parity)
- Manufacturer's code (encoded in odd parity)
- Center guard pattern
- Product code (encoded in even parity)
- Check digit (encoded in even parity)
- Right guard pattern

UPC-A uses a different coding scheme for the first half of a symbol (number system digit and manufacturer's code) and the second half of a symbol (product code and check digit). The left half uses the odd parity encodations of digits and the right half uses the even parity encodation digits.

Note: UPC-A and EAN bar codes are mutually incompatible. You cannot select both of these bar code types at the same time.

UPC-E

0 012345 7

The Universal Product Code (UPC) bar code version E is a zero-suppressed version of UPC-A. This version compresses the data characters and the checksum into six characters. Only tags with a number system character of zero can be encoded into UPC-E. In addition, the original ten data characters must have at least four zeros. Note that supplementals are not supported.

A UPC-E bar code consists of the following:

- Left guard pattern
- Data characters
- Compression type digit
- Right guard pattern

EAN-8 and EAN-13

0 123456 789012 0123 4565

The European Article Numbering (EAN) system is used for products that require a country origin. This is a fixed-length code used to encode either eight or thirteen characters. The first two characters identify the country of origin, the next characters are data characters, and the last character is the checksum. The EAN character set is a superset of the UPC-A character set. Note that supplementals are not supported.

An EAN bar code consists of the following:

- Left guard pattern
- Odd parity digits
- Center guard pattern
- Even parity digits
- Mandatory check digit
- Right guard pattern

Note: UPC-A and EAN bar codes are mutually incompatible. You cannot select both types of codes at the same time.

Code 93

0 1 2 3 4 5 6 7 8

Code 93 is a variable length bar code that encodes 47 characters. It is named Code 93 because every character is constructed from nine elements arranged into three bars with their adjacent spaces. Code 93 is a compressed version of Code 39 and was designed to complement Code 39.

Code 93 supports the following characters:

- 26 uppercase letters
- Ten digits
- Seven special characters (- . $ / + % and a space)
- Four special precedence characters ($, %, /, +)
- Start / stop character

A Code 93 bar code consists of the following elements:

- Leading quiet zone
- Start character
- Data characters
- First check character (referred to as C)
- Second check character (referred to as K)
- Stop character
- Termination bar
- Trailing quiet zone

Code 128

0 1 2 3 4 5 6 7 8

Code 128 is an alphanumeric, very high-density, compact, variable length bar code scheme that can encode the full 128 ASCII character set. Each character is represented by three bars and three spaces totaling 11 modules. Each bar or space is one, two, three, or four modules wide with the total number of modules representing bars an even number and the total number of modules representing a space an odd number. Three different start characters are used to select one of three character sets.

Code 128 supports 107 unique characters, including:

- Four function characters
- Four code set selection characters
- Three start characters

A Code 128 bar code consists of the following elements:

- Leading quiet zone
- Start character
- Data characters
- Mandatory check character
- Stop character
- Termination bar
- Trailing quiet zone

UCC 128

0 1 2 3 4 5 6 7 8

The EAN/UCC 128 symbology is a variation of the original Code 128 symbology designed primarily for use in product identification applications.

The EAN/UCC 128 specification uses the same code set as Code 128 except that it does not allow function codes FNC2-FNC4 to be used in a symbol and FNC1 is used as part of the start code in the symbol. The check digit in EAN/UCC128 symbols is also calculated differently than in Code 128.

Postnet

The Postnet (Postal Numeric Encoding Technique) is a fixed length symbology (5, 6, 9, or 11 characters) which uses constant bar and space width. Information is encoded by varying the bar height between the two values. Postnet codes are placed on the lower right of envelopes or postcards, and are used to expedite the processing of mail with automatic equipment and provide reduced postage rates.

A Postnet code consists of the following elements:

A tall start bar.

Data digits consisting of groups of five bars for each digit to be encoded. Each digit contains two tall bars and three short bars.

A five bar check character. The value of the check digit is chosen such that the sum of all data digits and the check character is an integral multiple of ten.

A tall stop bar.

Note: Postnet cannot be used in combination with any other bar code type when specifying multiple bar code types.

4-State Barcodes

The 4-State Barcodes are a family of symbologies which use 3-segment bars of fixed width and spacing. Two bits of data are encoded in each bar by varying the presence of the top and bottom segments, creating one of the 4 possible bar states. 4-State Barcodes are typically printed on mail to record address and internal postal routing information.

The different 4-State Barcode symbologies have differing lengths and data capacities. AMPLIB supports three 4-State Barcode formats:

- the US Postal Service 4-State Customer Barcode or 'ONECode Solution' barcode
- the Denmark Post Intelligent Barcode
- the Australia Post 4-State Customer Barcode

The US Postal Service 4-State Customer Barcode consists of 65 bars and can record up to 31 single-digit numbers. It includes a CRC code to allow error detection.

The Denmark Post Intelligent Barcode consists of 66 bars and can record up to 31 single-digit numbers. It includes Reed-Solomon error correction that allows complete data recovery from some damaged barcodes. Any 4 missing bars (and possibly up to 12 missing bars) can be recovered, and any 2 incorrect bars (and possibly up to 6 incorrect bars) can be recovered.

The Australia Post 4-State Customer Barcode is of variable length and can consist of 37, 52, or 67 bars. Each of these barcode lengths has a base payload of 8 single-digit numbers. In addition to this, codes with 52 bars can contain 8 more single-digit numbers or 5 alphanumeric characters, and codes with 67 bars can contain 15 more single-digit numbers or 10 alphanumeric characters. Regardless of length, each of these codes contain Reed-Solomon error correction symbols that allow complete data recovery despite some amounts of damage. Any 4 missing bars (and possibly up to 12 missing bars) can be recovered, and any 2 incorrect bars (and possibly up to 6 incorrect bars) can be recovered.

Reading Postal Codes

Four(4) State Barcodes (FSB) are atypical in that the information content is in the vertical height of the bars and not the horizontal width.

Each bar can have 4 different values and most of the codes will use a group of 4 bars to provide up to 64 different characters.

The All My Papers AmpLib engine with FSB barcode type will read Four State Barcodes and auto decode based on the number of bars in the code (see the table).

The national mail codes have bar height and spacing information provided as part of the definition. This is critical information and the resolution parameter in the image must be correct for accurate reads. Depending on what the country specification expects, some codes will not present all of the information available.

Royal Mail is a good example where there multiple versions that may or may not present the mail sub codes. In the case of Royal Mail, the check sum is not presented.

The US One Code is a case that will have the same number of bars in each code but different amounts of data.

Each country will have a range of skew that can be processed and the AMP FSB decoder follows those ranges. This value is usually far less than the general AMP barcode decodes.

Code	ECC	Check Sum	Bars	Start/Stop	Comments/Sub Codes
US One Code/ Intelligent Mail	Yes	No	65	Yes	Single code length with many different length results.
Royal Mail	No	Yes	38, 42	Yes	
Royal Dutch	No	Not Calcu-lated	32,44	No	
Singapore	No	Yes	30	Yes	
Australian Post	Yes	No	37, 52, 67		Sub codes -- default is C table. Consult AMP Support for other tables.
Canadian Post	Yes	No	52	Yes	Decode algorithm not in public domain--not decoded.

Most Bar Code Generators using Fonts will have an option to put annotation above or below the barcode. This may violate the spacing values on a mail piece. So test images should not have the value annotation if it does (see table for clear space value).

There are three basic levels of authentication in the codes (high, middle, low). An Error Correction Code (ECC) provides a high level. A check sum with character validation provides a middle level of authentication. Character validation without a checksum provides a low level of authentication. Neither the ECC characters nor the checksum characters are output for the reader.

Warning: The AMP FSB decoder is for the specific country mail codes and is not intended as a general purpose reader for arbitrary encoding of the code.

Code	Typical Full Bar Height	Typical Bars per Inch	Min Clear Space Above/ Below	Authentication
US One Code/ Intelligent Mail	0.18 – 0.25 inch	24	0.08 inch	High--ECC
Royal Mail				Middle--Check Sum
Royal Dutch				Low
Singapore				Middle—Check Sum

PDF-417

PDF417 is a high-density 2 dimensional bar code symbology that essentially consists of a stacked set of smaller bar codes. The symbology is capable of encoding the entire (255 character) ASCII set. PDF stands for "Portable Data File" because it can encode as many as 2725 data characters in a single bar code.

The complete specification for PDF417 provides many encoding options including data compression options, error detection and correction options, and variable size and aspect ratio symbols. The symbology was published by Symbol Technologies to fulfill the need for higher density bar codes.

The low level structure of a PDF417 symbol consists of an array of code words (small bar and space patterns) that are grouped together and stacked on top of each other to produce the complete printed symbol. An individual code word consists of a bar and space pattern 17 modules wide. The user may specify the module width, the module height, and the overall aspect ratio (overall height to width ratio) for the complete symbol. A complete PDF417 symbol consists of at least 3 rows of up to 30 code words and may contain up to 90 code word rows per symbol with a maximum of 928 code words per symbol.

The code words in a PDF417 symbol are generated using one of three data compression modes currently defined in the symbology specifications. This allows more than one character to be encoded into a single data code word. Because different data compression algorithms may be used, it is possible for different printed symbols to be created from same input data.

The symbology also allows for varying degrees of data security or error correction and detection. Nine different security levels are available with each higher level adding additional overhead to the printed symbol.

Data Matrix

Data Matrix is a high-density 2 dimensional bar code symbology which is made up of square modules arranged within a perimeter finder pattern. Although the formal specification allows dark symbols on light backgrounds as well as light symbols on dark background, AMPLIB supports only the former (as shown above). The symbology is capable of encoding the entire (255 character) ASCII set in a variety of internal formats. The largest symbol can encode as many as 3116 numeric characters.

The complete specification for Data Matrix provides many encoding options including data compression options, error detection and correction options, and variable size and aspect ratio symbols. AIM International, Inc. publishes a document entitled "International Symbology Specification-Data Matrix." AMPLIB supports only the symbols denoted as ECC 200 in this document.

Each Data Matrix symbol consists of data regions made up of an array of code words. Larger symbols consist of multiple data regions separated by alignment patterns. The finder pattern that surrounds a data region consists of an L shape in the lower left corner and alternating white-black squares on the top and right sides. The number of black and white squares on the top and right determine the overall size of the symbol. The symbol shown above has 18 rows and 18 columns with the single data region being 16x16. A quiet zone of white space surrounds each symbol. Square symbols can have 1, 4, 16, or 36 individual data regions. Square symbols with 1 data region range in size from 10x10 to 26x26. Square symbols with 4 data regions range in size from 32x32 to 52x52. AMPLIB supports square symbols with up to 52 elements on a side.

The code words in a Data Matrix symbol are generated using one of six data compression modes defined in the symbology specification. This allows more than one character to be encoded into a single data code word. Because different data compression algorithms may be used, it is possible for different printed symbols to be created from same input data.

The symbology uses error correction and detection to allow for the efficient recovery of data from degraded symbols. AMPLIB will only report back data from a symbol if the error correction codes have been correctly processed. This corresponds to a Confidence of 30-100%. A Confidence of 20% means there were too many errors to be corrected. 10% Confidence means that a potential symbol's alignment pattern was found, but that the error correction codes within the data region were heavily degraded and unusable.

Grayscale or bilevel representations of Data Matrix are supported by AMPLIB. If the data and alignment elements are small (3-5 pixels), grayscale images will give superior recognition performance. Grayscale images should always be of good contrast with a median value of about 128. The AMPLIB ampGrayProcesses routine can be used to optimize grayscale images prior to recognition. Symbols may be upside down and still be recognized correctly (although at a somewhat reduced rate). Symbols with an overall skew less than 20 degrees will read with greater accuracy.

Quick Response (QR)

QR is a 2-dimensional barcode symbology consisting of light and dark square modules arranged in a square grid pattern of varying size. The modules of a QR code represent payload data, barcode format information, Reed-Solomon error correction data, and patterns to facilitate acquisition and orientation by scanners.

Supported barcode sizes range from 21x21 modules to 177x177 modules. These sizes are evenly spaced in 4 module steps across this range (21x21, 25x25, 29x29, etc).

QR has a subformat called MicroQR which allows four small barcode sizes (11x11, 13x13, 15x15, 17x17). These have a single finder square symbol in their top-left corner and are encoded and decoded differently from normal QR codes.

QR codes contain finder square symbols in their top-left, top-right, and bottom-left corners. Additionally, they contain smaller orientation square symbols arranged on a grid throughout the pattern to facilitate grid determination by scanners.

QR codes can contain numeric data, alphanumeric data, raw data bytes, and kanji characters. QR codes are able to switch between different data types in the same code.

QR codes use Reed-Solomon error correction with user-selectable redundancy levels. Low, Medium, Quartile, and High levels are available, providing recovery from the loss of approximately 7%, 15%, 25%, and 30% of the data area of the code, respectively.

The largest QR code (177x177) with the Low Reed-Solomon level can contain up to 7089 numeric characters, 4296 alphanumeric characters, 2953 bytes of raw data, or 1817 kanji characters.

Glossary

Glossary of Terms

ABA

American Bankers Association; an association that provides, among many things, the check format specification for the United States. Each country will have its own organization and format specification.

AIM

Automatic Identification Manufacturers; an association that develops bar code specification standards.

Alias Image

An alias image is a named image structure that describes a sub-image area of a work image, but does not have any storage allocated; it simply points into the base workimage it is derived from.

API

API stands for Applications Programming Interface, and refers to a package of procedure and function calls to implement a set of related functions. AMPLIB includes an API, callable from REXX, C, Visual Basic, and any other DLL-compatible language.

DLL

Dynamic Link Library. A type of executable module that can be linked with an application at run time. AMPLIB is implemented as a DLL.

Download

The process of transferring files or data from another computer into your own.

DX

The term used to describe the width in pixels of a sub-image. All image operations act on the active sub-image, rather than the entire work image.

DY

The term used to describe the height in lines of a sub-image. All image operations act on the active sub-image, rather than the entire work image.

Fixed Size Work Image

A work image with a fixed allocation of memory; the amount of storage does not shrink or grow with use. Primarily used for tiled image operations. Scanner images and printer images are also in this category.

Height

Height of an image refers to the number of lines in the work image. This is not necessarily the same as the active sub-image height (see DY).

Job

In scanning terminology, a job refers to a group of documents delimited by job separator sheets.

MICR

Magnetic Ink Character Recognition. See Appendix D (p.80) for more information.

Microsoft Windows

The Microsoft Windows Operating System, which is required to run AMPLIB. Windows 95 or higher is needed.

OCR

Optical Character Recognition detects the amount of light reflected from a printed character to identify that character.

Page orientation

Documents may be scanned either in portrait mode, where the height is greater than the width, or landscape mode, where the width is greater than the height.

Page size

Term used to identify the physical size of a document to be scanned or printed. Page size is given as a string identifier, such as "letter", "legal", "a4", etc.

Pitch

Term used for width of a work image, in pixels. This is a distance between pixels on adjacent lines, and not necessarily the active sub-image area width (see DX).

Sub-Image

Every work image has an active sub-image area, defined by the image metrics X, Y, DX, and DY. (X,Y) give the pixel offset from the upper left corner of the work image; (DX, DY) give the width and height of the sub-image area.

Variable Size Work Image

A work image that has a variable amount of storage allocated for it. Variable size work images shrink and grow as they are used to match the size of the image data written to them.

Width

The width of an image defines the number of pixels across a line of the image. The width of an image will always be less than or equal to the image pitch.

Work Image

A rectangular block of storage maintained by AMPLIB to hold image data.

Index