

# **PadCom**

*Point of Sale Terminal Control Library Version 4.3*

## **User's Manual**

*For use with @pos.com POS terminals including PenWare™ 100, 1100, 1500, 2000, 3000, 3100, and the iPOS TC running posClassic*

*Supports the following target operating systems:*

*Microsoft 16 bit Windows 3.x and 32 bit Windows 95/NT;*

*IBM OS/2; DOS version 3.3 or higher*



Part No. PC17401  
Revision F

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of @pos.com. @pos.com and its suppliers provides this product "as is" and make no representation or warranty regarding the content of this product and its documentation. All information in the product and documentation is subject to change without notice. @pos.com disclaims all warranties, either express or implied, including the warranties of merchantability and fitness for a particular purpose. In no event shall @pos.com or its suppliers be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages.

Copyright © 1994-99 @pos.com. All rights reserved.

@pos.com and the @pos.com logo are registered trademarks and PenWare100, PenWare1100, PenWare1500, PenWare2000, PenWare3000, PenWare3100, iPOS TC, posClassic, PadCom, SigKit and SigBox are trademarks of @pos.com.

Microsoft is a registered trademark and Windows and Visual C/C++, Visual Basic are trademarks of Microsoft Corporation.

Borland is a registered trademark and Borland C/C++ is a trademark of Borland International, Inc.

IBM, OS/2 and Visual Age C++ are trademarks or registered trademarks of International Business Machines Corporation.

# Table of contents

<b>INTRODUCTION.....</b>	<b>7</b>
<i>Introducing @pos.com PadCom library .....</i>	<i>7</i>
<i>Features of PadCom library.....</i>	<i>7</i>
<i>@pos.com POS terminals supported by PadCom.....</i>	<i>7</i>
<i>PadCom Interface .....</i>	<i>7</i>
<b>INSTALLATION.....</b>	<b>9</b>
<b>OVERVIEW.....</b>	<b>11</b>
<i>@pos.com Device Overview.....</i>	<i>11</i>
<i>Aspect ratio considerations.....</i>	<i>11</i>
<i>Representing pen strokes.....</i>	<i>12</i>
<i>Illustration of two pen strokes: .....</i>	<i>12</i>
<i>Using PadCom.....</i>	<i>12</i>
<b>LIBRARY REFERENCE .....</b>	<b>14</b>
<i>Library functions list by @pos.com pad compatibility.....</i>	<i>14</i>
<i>Library functions list by category.....</i>	<i>18</i>
<b>ALPHABETIC REFERENCE OF FUNCTIONS .....</b>	<b>23</b>
<i>padBinaryGetTable .....</i>	<i>23</i>
<i>padBinaryGetVar.....</i>	<i>23</i>
<i>padBox.....</i>	<i>24</i>
<i>padClear.....</i>	<i>24</i>
<i>padClearPixel.....</i>	<i>24</i>
<i>padComDate .....</i>	<i>25</i>
<i>padComVersion.....</i>	<i>25</i>
<i>padConnect.....</i>	<i>25</i>
<i>padConnectClearScreen.....</i>	<i>26</i>
<i>padDisplayObject .....</i>	<i>26</i>
<i>padDisplayTime .....</i>	<i>27</i>
<i>padEchoComm.....</i>	<i>27</i>
<i>padEraseTable .....</i>	<i>28</i>
<i>padFieldButton .....</i>	<i>28</i>
<i>padFieldSignature.....</i>	<i>29</i>
<i>padFlush.....</i>	<i>29</i>
<i>padFormDeleteFld.....</i>	<i>30</i>
<i>padFormSaveFld.....</i>	<i>30</i>
<i>padFrame.....</i>	<i>30</i>
<i>padGet .....</i>	<i>31</i>
<i>padGetAllMagCardTracks .....</i>	<i>32</i>
<i>padGetArea.....</i>	<i>32</i>
<i>padGetBaudRate .....</i>	<i>33</i>
<i>padGetBkColor .....</i>	<i>33</i>
<i>padGetCmdSetID .....</i>	<i>33</i>
<i>padGetColor .....</i>	<i>34</i>
<i>padGetColors.....</i>	<i>34</i>
<i>padGetConnectTimeout.....</i>	<i>34</i>

<i>padGetDefaultBaudRate</i> .....	35
<i>padGetDUKPTbinaryPIN</i> .....	35
<i>padGetDUKPTtextPIN</i> .....	36
<i>padGetFont</i> .....	37
<i>padGetFontSize</i> .....	37
<i>padGetInTimeout</i> .....	38
<i>padGetMagTrack</i> .....	38
<i>padGetMasterSessionBinaryPIN</i> .....	39
<i>padGetMasterSessionTextPIN</i> .....	40
<i>padGetMaxCardTracks</i> .....	41
<i>padGetMaxCardTrackSize</i> .....	41
<i>padGetModel</i> .....	41
<i>padGetNumVar</i> .....	42
<i>padGetOutTimeout</i> .....	42
<i>padGetPage</i> .....	42
<i>padGetPIN</i> .....	43
<i>padGetPort</i> .....	43
<i>padGetPortAddr</i> .....	44
<i>padGetPortIrq</i> .....	44
<i>padGetPorts</i> .....	44
<i>padGetScanRate</i> .....	45
<i>padGetStdPIN</i> .....	45
<i>padGetTableItem</i> .....	46
<i>padGetTime</i> .....	46
<i>padGetVersion</i> .....	47
<i>padHardwareButton</i> .....	47
<i>padHardwareButtonIEnable</i> .....	48
<i>padHeight</i> .....	48
<i>padHideTime</i> .....	49
<i>padHorzDPI</i> .....	49
<i>padInkExport</i> .....	49
<i>padInvert</i> .....	50
<i>padIsaReset</i> .....	50
<i>padIsKey</i> .....	51
<i>padIsLcd</i> .....	51
<i>padIsNewStroke</i> .....	51
<i>padIsOn</i> .....	52
<i>padIsPenDown</i> .....	52
<i>padIsRecord</i> .....	52
<i>padLcdHeight</i> .....	52
<i>padLcdHorzDPI</i> .....	53
<i>padLcdVertDPI</i> .....	53
<i>padLcdWidth</i> .....	53
<i>padLeftButton</i> .....	53
<i>padLightOff</i> .....	54
<i>padLightOn</i> .....	54
<i>padLine</i> .....	54
<i>padMemClear</i> .....	55
<i>padMemDelete</i> .....	55
<i>padMemDeleteVar</i> .....	56
<i>padMemFind</i> .....	56
<i>padMemGetChecksum</i> .....	56
<i>padMemGetFree</i> .....	57
<i>padMemGetVar</i> .....	57

<i>padMemLoadBitmap</i> .....	58
<i>padMemLoadBitmapFile</i> .....	58
<i>padMemLoadText</i> .....	59
<i>padMemReset</i> .....	59
<i>padMemSetVar</i> .....	59
<i>padMiddleButton</i> .....	60
<i>padName</i> .....	60
<i>padNewX</i> .....	61
<i>padNewY</i> .....	61
<i>padOff</i> .....	61
<i>padOldX</i> .....	61
<i>padOldY</i> .....	62
<i>padOn</i> .....	62
<i>padPassThroughHandshaking</i> .....	62
<i>padPassThroughOff</i> .....	63
<i>padPassThroughOn</i> .....	63
<i>padPassThroughResetCodes</i> .....	64
<i>padPassThroughSetOffCode</i> .....	65
<i>padPassThroughSetOnCode</i> .....	65
<i>padPortReclaim</i> .....	66
<i>padPortRelease</i> .....	66
<i>padPromptHexNumber</i> .....	66
<i>padPromptNum</i> .....	67
<i>padPromptNumber</i> .....	68
<i>padPromptReset</i> .....	69
<i>padPromptSignature</i> .....	70
<i>padPromptString</i> .....	70
<i>padPromptTimeout</i> .....	71
<i>padPutBits</i> .....	71
<i>padPutBmpFile</i> .....	72
<i>padPutLogo</i> .....	72
<i>padPutText</i> .....	73
<i>padReadByte</i> .....	73
<i>padRecord</i> .....	74
<i>padReset</i> .....	74
<i>padResetArea</i> .....	75
<i>padResetBaudRate</i> .....	75
<i>padResetConnectTimeout</i> .....	75
<i>padResetDefaultBaudRate</i> .....	76
<i>padResetInTimeout</i> .....	76
<i>padResetMagCard</i> .....	76
<i>padResetOutTimeout</i> .....	77
<i>padRightButton</i> .....	77
<i>padScale</i> .....	78
<i>padScaleDPI</i> .....	78
<i>padScaleTo</i> .....	78
<i>padScaleX</i> .....	79
<i>padScaleY</i> .....	79
<i>padSendByte</i> .....	80
<i>padSetArea</i> .....	80
<i>padSetAutoInking</i> .....	81
<i>padSetBaudRate</i> .....	81
<i>padSetBkColor</i> .....	82
<i>padSetClearButton</i> .....	83

<i>padSetColor</i> .....	83
<i>padSetCompress</i> .....	84
<i>padSetConnectTimeout</i> .....	84
<i>padSetDebug</i> .....	84
<i>padSetDefaultBaudRate</i> .....	85
<i>padSetFlowControl</i> .....	85
<i>padSetFont</i> .....	86
<i>padSetInkingArea</i> .....	86
<i>padSetInTimeout</i> .....	87
<i>padSetOutTimeout</i> .....	87
<i>padSetLcdClearTimeout</i> .....	88
<i>padSetLogo</i> .....	88
<i>padSetLogoBmpFile</i> .....	89
<i>padSetNumVar</i> .....	89
<i>padSetPadMode</i> .....	90
<i>padSetPadOffset</i> .....	90
<i>padSetPixel</i> .....	91
<i>padSetPort</i> .....	92
<i>padSetPortAddr</i> .....	92
<i>padSetPortHandle</i> .....	93
<i>padSetPortIrq</i> .....	93
<i>padSetPorts</i> .....	94
<i>padSetScanRate</i> .....	94
<i>padSetTime</i> .....	94
<i>padSetType</i> .....	95
<i>padSoundBell</i> .....	95
<i>padSoundEnable</i> .....	96
<i>padSoundSetFreq</i> .....	96
<i>padSoundTone</i> .....	97
<i>padStop</i> .....	98
<i>padToHIENGLISH</i> .....	98
<i>padToLOENGLISH</i> .....	98
<i>padToHIMETRIC</i> .....	99
<i>padToLOMETRIC</i> .....	99
<i>padType</i> .....	100
<i>padUpdate</i> .....	100
<i>padVertDPI</i> .....	100
<i>padWidth</i> .....	101
<b>APPENDIX A</b> .....	<b>102</b>
<b>APPENDIX B</b> .....	<b>104</b>
PADCOM SIGNATURE CAPTURE SAMPLE FOR DOS: .....	104
PADCOM SIGNATURE CAPTURE SAMPLE FOR WINDOWS 3.X: .....	105
PADCOM SIGNATURE CAPTURE SAMPLE FOR WINDOWS95/NT .....	108
<b>APPENDIX C</b> .....	<b>111</b>
PADCOM MSR SAMPLE CODE FOR DOS: .....	111
PADCOM MSR SAMPLE CODE FOR WIN 3.X.....	113
PADCOM MSR SAMPLE CODE FOR WIN 95/NT: .....	119
<b>INDEX</b> .....	<b>126</b>

# Introduction

## Introducing @pos.com PadCom library

PadCom library by @pos.com is a tool that allows developers to interface to @pos.com units easily and quickly. Developers can use PadCom library to capture signatures, read MSR, perform PINpad transactions, display text or bitmaps and write innovative graphical interfaces. By leveraging on @pos.com's position as a leader in signature capture technology, you can rest assured that your application will provide accurate and quality results with minimal programming efforts.

## Features of PadCom library

@pos.com PadCom library includes the following key features:

- Full interface to all @pos.com devices.
- Flexible architecture to support a wide variety of development needs.
- Simple, easy to use command set for rapid applications development.
- Consistent cross-platform APIs for enhanced portability.
- Accurate representation of image points.
- Interrupt/message driven engine for quick response to hardware activities.
- Reliable implementation of RS232 serial communications protocol.
- Automatic detection of connected @pos.com unit and automatic configuration.

In addition, PadCom library has been designed to work seamlessly with @pos.com SigKit<sup>1</sup>, the signature processing library for a complete signature capture solution.

## @pos.com POS terminals supported by PadCom

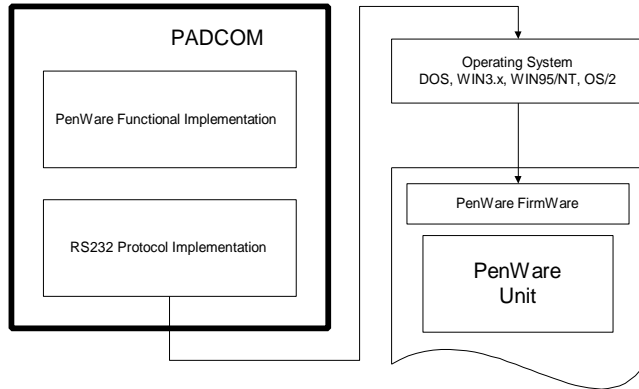
PadCom contains full support for the PenWare100, PenWare1100, PenWare1500, PenWare2000, PenWare3000, PenWare 3100, and the iPOS TC. PadCom's support for the iPOS TC is limited to posClassic mode only and does not support posBrowser or posPortal mode. The iPOS TC in posClassic mode is backwards compatible with the PenWare3000 and PenWare3100. PadCom treats the PenWare3100 and the iPOS TC both as PenWare 3000 POS terminals.

## PadCom Interface

The following diagram represents the PadCom interface to @pos.com units.

---

<sup>1</sup> For more information about the SigKit library and other development tools, please contact your @pos.com sales representative.



The diagram shows how PadCom communicates with the @pos.com unit. The functional implementation of @pos.com device is implemented on top of the RS232 protocol implementation. PadCom then communicates to the device over the supported operating systems platforms, which include DOS, WIN3.x, WIN95/NT and OS/2<sup>2</sup>. Although, it is possible to use other RS232 implementations, @pos.com recommends that the programmers use the built-in driver. It is important to note that the programmers do not have any direct access to internal firmware on the @pos.com units.

---

<sup>2</sup> For more information about other operating systems and supported drivers, please contact your @pos.com sales representative.

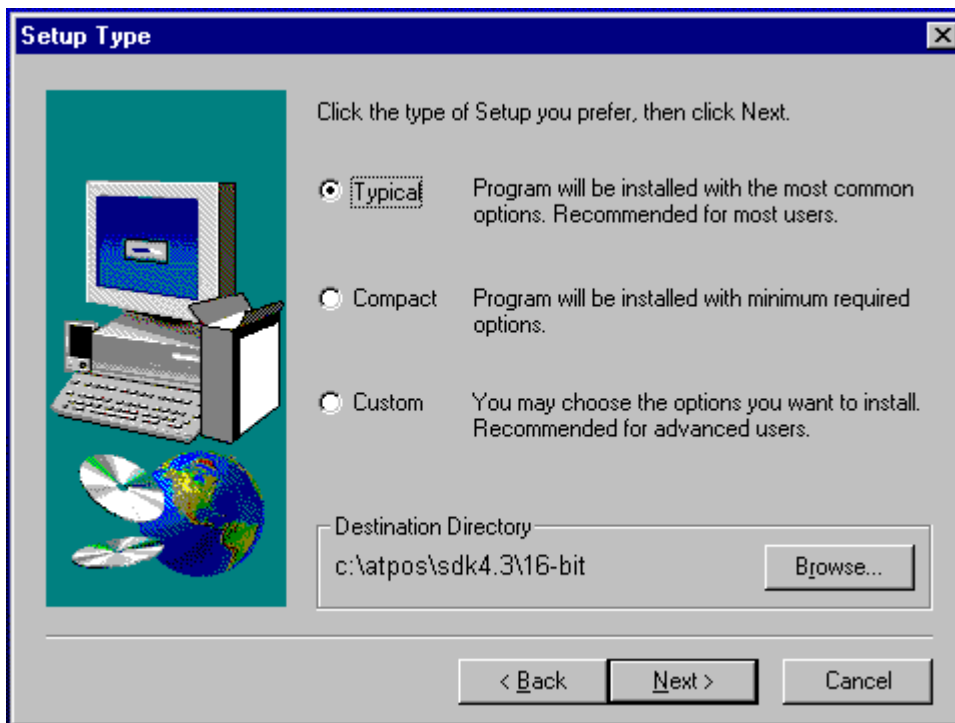
## Installation

@pos.com Library Installation Program is a simple to use InstallShield® application. It allows you to selectively install the desired components and the sample programs. Please note that the distribution diskettes allow you to install all of PadCom, SigKit and the Custom VBX control<sup>3</sup>.

Please follow the instructions given below:

1. Insert the diskette labeled “Library Installation Disk 1 of 2” in your 3 ½” disk drive.
2. From your Windows Manager, select the 3 ½” disk drive.
3. Find and execute setup.exe (You can execute the program by double-clicking on its icon).

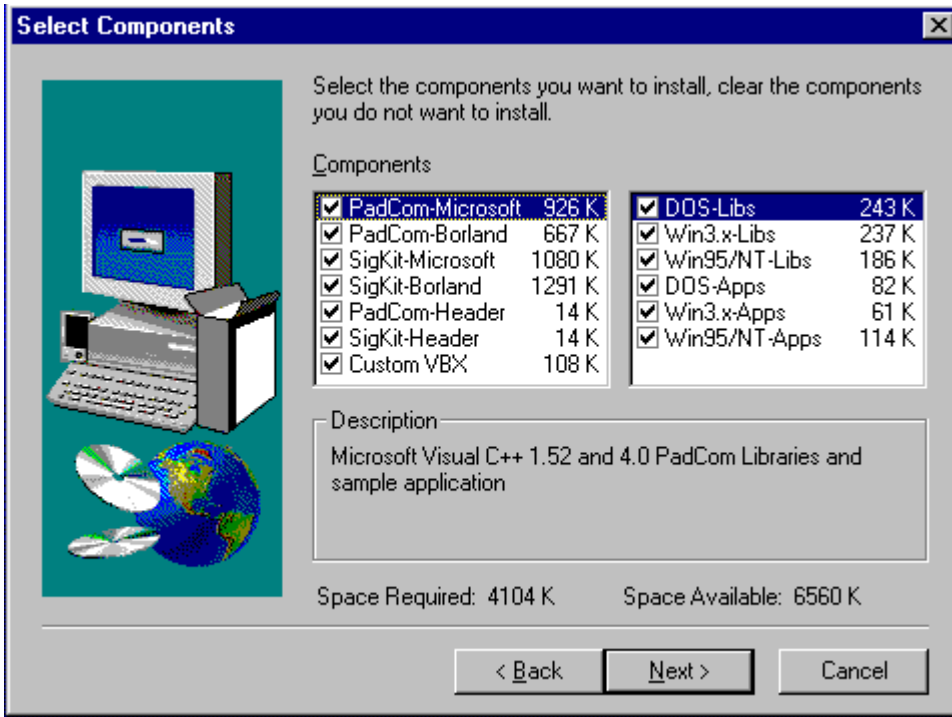
A series of simple screens are presented to the user with some relevant information and some selection options for customization. Please read the instructions on each of these screens carefully. The screen-shot shown below allows the user to customize the installation by selecting only the desired set of components.



Selecting the “Typical” option installs all of PadCom, SigKit and the custom VBX control. Selecting the “Compact” option installs all but the custom VBX control. Choosing the “Custom” option allows the user to selectively install components as desired. @pos.com recommends choosing the “Typical” option for beginners. The following screen shot shows the screen associated with the “Custom” option.

---

<sup>3</sup> For more information on SigKit and the custom VBX control, please contact your @pos.com sales representative.



The “Custom” option breaks down the installation into components. As shown, these components are grouped based on the type of compiler used. Recall that @pos.com supports Microsoft’s Visual C++ 1.52, Visual C++ 4.0 and Borland C++ 4.2 compilers.

The user has an option to select only the desired components. Each component (PadCom-Microsoft shown in the screen shot example), has an option to install the libraries and/or the samples applications. The library sub-components are post-fixed with “Lib” or “Libs” whereas the Application sub-components are post-fixed with “App” or “Apps.”

The rest of the installation is quite straightforward. Please follow the remaining instructions to complete the installation process. Please view the “Readme.txt” file supplied with the installation for details on installation. A brief description of some of the topics covered in the “Readme.txt” file is presented later on for convenience.

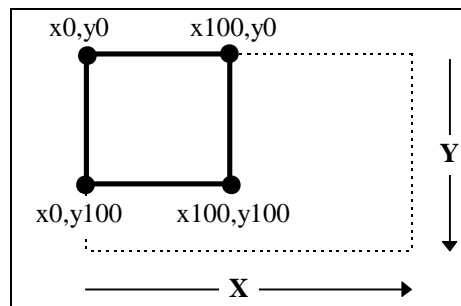
## Overview

This section provides the reader with an overview of the @pos.com pad technology and the terminology used in this manual. A simple example source code is listed to quickly write your first @pos.com application.

### @pos.com Device Overview

The pad surface is composed of a large number of individual points or “pixels” as are video monitors, printers and other devices. Because the library is designed to maintain the integrity of the input points, it does NOT distort the image by adjusting for squareness. Therefore, when using points obtained, it is important to understand both the mapping of the pad surface and the concept of aspect ratios. In addition, the methods used to represent pen strokes must be considered.

The points on the pad are specified by using a pair of horizontal and vertical coordinates (often referred to as “X” and “Y” coordinates, respectively). The “origin” of the coordinates is at the top/left corner and increases in a positive direction towards the bottom/right. Therefore, the coordinate pair “0,0” is at top/left corner; “0,100” is 100 points to the right, “100,0” is 100 points down; and “100,100” is both 100 points to the right and 100 points down. The actual coordinates of the bottom/right corner can be represented as “**padWidth()-1, padHeight()-1**”, or obtained by using the **padGetArea** function.



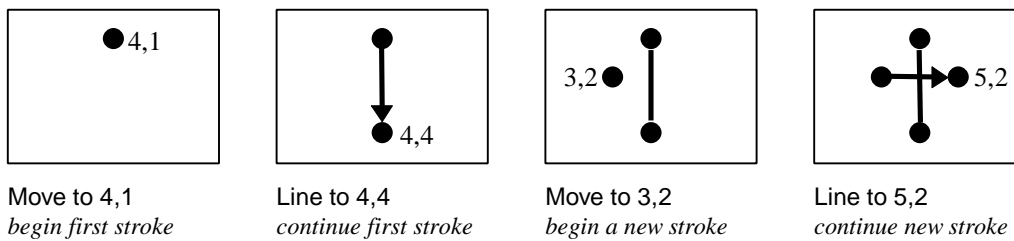
### Aspect ratio considerations

Since the points returned by the library represent the actual points on the physical pad device, it is necessary to understand the concept of “aspect ratios”. Basically, an aspect ratio refers to the difference between the width and height of the physical points. For example, a 1:1 ratio means that the width and height of a point are the same and that the point is perfectly square. Similarly, an aspect ratio of 2:1 means that a point is twice as wide as it is high. This is important because a shape that appears square when created at, for example, a 2:1 aspect ratio, will appear vertically stretched when viewed at an aspect ratio of 1:1. The aspect ratio of the pad surface is represented by the ratio between horizontal and vertical “dots-per-inch” and can be obtained by using the functions **padHorzDPI** and **padVertDPI** respectively. To maintain proper aspect ratios when “drawing” the points on the screen or other device, the library provides functions such as **padScaleDPI** to facilitate aspect ratio adjustment.

## Representing pen strokes

While a single point consists of a single horizontal and vertical coordinate, a pen stroke consists of all the points received from the time the pen comes into contact with the pad through the time it is removed. A typical stroke contains many points. The first point, referred to as a “moveto” point, indicates where the pen was first placed in contact with the pad. Remaining points, referred to as “lineto” points, indicate that the pen is being “dragged” across the pad surface. Consequently, the **padGet** function retrieves a “moveto/lineto” pen indicator as well as the corresponding coordinates.

### Illustration of two pen strokes:



## Using PadCom

As mentioned before, PadCom provides a fully customizable interface to @pos.com pads. An operating system dependent set of libraries are provided using the most popular compilers to support a wide variety of development need. For example, for Windows 3.x operating system, a separate set of three libraries is compiled using Borland C++ and Microsoft Visual C++ compilers. In addition to this, various memory models and thread support is provided wherever applicable.

PadCom consists of statically linked libraries. This means that the application development needs to be carried out in the following stages.

### Implementation

Before proceeding with the design specifications, it is recommended that you study the functionality supported by the chosen @pos.com device. A complete list of functionality exposed by PadCom is provided in Library reference on page 14.

A few sample source code listing is attached in Appendix B on page 104 as an example of how to use PadCom library and the exposed API functions. The same sample source code is also provided as a part of the sample application code installed with the SDK components. Please bear in mind that these samples are intended for demo purposes only and do not necessarily perform useful tasks. However, they provide a basis on which you can build complex and meaningful Point Of Sale applications.

### Compiling and Linking

Compiling and linking your application involves selecting the appropriate libraries and choosing the correct build environment on the compiler of your choice. As mentioned before, a complete operating system and compiler dependent solution is available for your use. The libraries are named using a standard naming convention.

**For DOS:**

padcomds.lib padcomdm.lib and padcomdl.lib are available to support the small, medium and the large memory models.

**For Windows 3.x:**

padcomws.lib padcomwm.lib and padcomwl.lib are available to support the small, medium and the large memory models.

**For Windows 95/NT:**

padcomw.lib and padcomwt.lib<sup>4</sup> are available.

The user should select the supported library and add it to the project. Please do not forget to add the common header file padcom.h or else the program will not compile correctly. Please note that @pos.com libraries are not differentiated based on the type of compiler used. For example, padcomds.lib is available for both the Borland C++ and Microsoft Visual C++ compilers. It is the responsibility of the programmer to use the correct library. The automatic installation program puts these libraries in clearly marked folders.

Please make sure that the compiler settings comply with the memory model (or the thread option wherever applicable). Failure to do so can result in unpredictable program failures.

**Troubleshooting**

Application development always involves debugging your application code. @pos.com has put together a highly competent Technical Support Team to help you troubleshoot your applications quickly. There is however some simple troubleshooting that you can do on your end before calling the Technical Support at @pos.com.

@pos.com strongly recommends that you always run the sample test programs that you installed while installing the library components. Please call @pos.com immediately if any of these sample programs do not work on the intended Operating Systems.

If the sample test programs work but your applications do not, please re-check the compiler settings, used libraries and the sample code. Before you call @pos.com Technical Support, please note down the operating system, environment, compilers, compiler settings, used libraries and other information that you think may be useful or relevant. This will help the technical support personnel diagnose and/or make recommendations quickly.

---

<sup>4</sup> Padcomwt.lib is available for the Microsoft compiler only.

## Library reference

### Library functions list by @pos.com pad compatibility.

In the following table the “Function” column contains the function’s name. The supported pad model columns (100, 1100, 1500, 2000 and 3000/3100/posClassic) contain an “X” if the function is supported when using that model pad, or are left blank if the function is not supported. “\*” indicates that the function is obsolete and should not be used.

Function	100	1100	1500	2000	3000/3100/posClassic
padBinaryGetTable	X	X	X	X	X
padBinaryGetVar	X	X	X	X	X
padBox			X		X
padClear			X	X	X
padClearPixel					X
padComDate	X	X	X	X	X
padComVersion	X	X	X	X	X
padConnect	X	X	X	X	X
padConnectClearScreen	X	X	X	X	X
padDisplayObject					X
padDisplayTime					X
padEchoComm					X
padEraseTable	X	X	X	X	X
padFieldButton					X
padFieldSignature	X	X	X	X	X
padFlush	X	X	X	X	X
padFormDeleteFld					X
padFormSaveFld					X
padFrame			X		X
padGet	X	X	X	X	X
padGetAllMagCardTracks					X
padGetBaudRate	X	X	X	X	X
padGetArea	X	X	X	X	X
padGetBkColor					X
padGetCmdSetID					X
padGetColor					X
padGetColors					X
padGetConnectTimeout	X	X	X	X	X
padGetDefaultBaudRate					X
padGetDUKPTbinaryPIN					X
padGetDUKPTtextPIN					X
padGetFont			X	X	X
padGetFontSize					X
padGetInTimeout	X	X	X	X	X
padGetMagTrack				X	X
padGetMasterSessionBinaryPIN					X

padGetMasterSessionTextPIN					X
padGetMaxCardTracks					X
padGetMaxCardTrackSize					X
padGetModel		X	X		X
padGetNumVar	X	X	X	X	X
padGetOutTimeout	X	X	X	X	X
padGetPage	X	X	X	X	X
padGetPIN				*	*
padGetPort	X	X	X	X	X
padGetPortAddr	X	X	X	X	X
padGetPortIrq	X	X	X	X	X
padGetPorts	X	X	X	X	X
padGetScanRate					X
padGetStdPIN				*	*
padGetTableItem	X	X	X	X	X
padGetVersion		X	X		X
padHardwareButton	X	X	X		
padHardwareButton1Enable	X	X	X		
padHeight	X	X	X	X	X
padHideTime					X
padHorzDPI	X	X	X	X	X
padInkExport	X	X	X	X	X
padInvert					X
padIsKey					X
padIsLcd	X	X	X	X	X
padIsNewStroke	X	X	X	X	X
padIsaReset	X	X	X	X	X
padIsOn	X	X	X	X	X
padIsPenDown	X	X	X	X	X
padIsRecord	X	X	X	X	X
padLcdHeight			X	X	X
padLcdHorzDPI			X	X	X
padLcdVertDPI			X	X	X
padLcdWidth			X	X	X
padLeftButton	X	X	X		
padLightOff	X	X	X		
padLightOn	X	X	X		
padLine			X		X
padMemClear					X
padMemDelete					X
padMemDeleteVar	X	X	X	X	X
padMemFind					X
padMemGetChecksum					X
padMemGetFree					X
padMemGetVar	X	X	X	X	X
padMemLoadBitmap					X
padMemLoadBitmapFile					X
padMemLoadText					X

padMemReset					X
padMemSetVar	X	X	X	X	X
padMiddleButton	X	X	X		
padName	X	X	X	X	X
padNewX	X	X	X	X	X
padNewY	X	X	X	X	X
padOff	X	X	X	X	X
padOldX	X	X	X	X	X
padOldY	X	X	X	X	X
padOn	X	X	X	X	X
padPassThroughHandshaking	X	X	X	X	X
padPassThroughOff					X
padPassThroughOn					X
padPassThroughResetCodes					X
padPassThroughSetOffCode					X
padPassThroughSetOnCode					X
padPortReclaim	X	X	X	X	X
padPortRelease	X	X	X	X	X
padPromptHexNumber					X
padPromptNum					X
padPromptNumber					X
padPromptReset					X
padPromptSignature					X
padPromptString					X
padPromptTimeout					X
padPutBits				X	X
padPutBmpFile				X	X
padPutLogo				X	X
padPutText			X	X	X
padReadByte	X	X	X	X	X
padRecord	X	X	X	X	X
padReset					X
padResetArea	X	X	X	X	X
padResetBaudRate					X
padResetConnectTimeout	X	X	X	X	X
padResetDefaultBaudRate					X
padResetInTimeout	X	X	X	X	X
padResetMagCard				X	X
padResetOutTimeout	X	X	X	X	X
padRightButton	X	X	X		
padScale	X	X	X	X	X
padScaleDPI	X	X	X	X	X
padScaleTo	X	X	X	X	X
padScaleX	X	X	X	X	X
padScaleY	X	X	X	X	X
padSendByte	X	X	X	X	X
padSetArea	X	X	X	X	X
padSetAutoInking					X

padSetBaudRate					X
padSetBkColor					X
padSetClearButton			X		X
padSetColor					X
padSetCompress					X
padSetConnectTimeout	X	X	X	X	X
padSetDebug					X
padSetDefaultBaudRate					X
padSetFlowControl		X	X		
padSetFont				X	X
padSetInTimeout	X	X	X	X	X
padSetInkingArea					X
padSetLcdClearTimeout			X		
padSetLogo				X	X
padSetLogoBmpFile				X	X
padSetNumVar	X	X	X	X	X
padSetOutTimeout	X	X	X	X	X
padSetPadOffset		X	X		X
padSetPixel					X
padSetPort	X	X	X	X	X
padSetPortAddr	X	X	X	X	X
padSetPortHandle	X	X	X	X	X
padSetPortIrq	X	X	X	X	X
padSetPorts	X	X	X	X	X
padSetScanRate					X
padSetTime					X
padSetType	X	X	X	X	X
padStop	X	X	X	X	X
padSoundBell					X
padSoundEnable					X
padSoundSetFreq					X
padSoundTone					X
padToHIENGLISH	X	X	X	X	X
padToLOENGLISH	X	X	X	X	X
padToHIMETRIC	X	X	X	X	X
padToLOMETRIC	X	X	X	X	X
padType	X	X	X	X	X
padUpdate	X	X	X	X	X
padVertDPI	X	X	X	X	X
padWidth	X	X	X	X	X

**Library functions list by category**

The following lists the functions in groups of distinct functional categories.

**Basic operations:**

padConnect	Attempts a connection to a pad (similar to padOn).
padConnectClearScreen	Connect and clear pad screen.
padGet	Retrieve the pen coordinate and status.
padIsaReset	Reset the extension card device
padOff	Turn the pad off.
padOn	Turn the pad on (similar to padConnect).
padRecord	Start recording pad data.
padStop	Stop recording pad data.
padUpdate	Update pad data.
padSetType	Sets a specific type of pad to be used.
padReset	Reset the pad.

**Clipping active area:**

padGetArea	Get the currently active clipping area.
padResetArea	Reset clipping to the full pad surface.
padSetArea	Sets the currently active clipping area.
padSetInkingArea	Sets the currently active inking area.

**Data Collection Operations:**

padBinaryGetTable	Get data from a table row.
padEraseTable	Erase all data in a binary table
padGetTableItem	Transfer data from a table row to a variable
padIsKey	Check if the DUPKT key is set.
padSetCompress	Set the compression number for storing the captured signature data.

**Error handling:**

padFlush	Flush any data waiting to be processed.
padSetDebug	Set the debug mode On or Off.

**LCD display operations:**

padBox	Draws a box onto the LCD display.
padClear	Clears the LCD display.
padClearPixel	Clears a pixel on the LCD display.
padFrame	Draws a frame onto the LCD display.
padGetBkColor	Gets the current background color.
padDisplayObject	Draw a stored memory object.
padGetColor	Gets the current foreground color.
padGetColors	Gets the number of colors available.
padInvert	Inverts an area on the LCD display.
padLine	Draws a line on the LCD display.
padPutText	Draws text on the LCD display.
padPutBits	Draws a bitmap on the LCD display.
padPutBmpFile	Draws a Windows BMP file on the LCD display.

padPutLogo	Draws the logo image on the LCD display.
padSetBkColor	Sets the current background color.
padSetClearButton	Sets which button clears the LCD.
padSetColor	Sets the current foreground color.
padSetLcdClearTimeout	Sets the automatic clearing time out for the LCD.
padSetLogo	Sets the logo image.
padSetLogoBmpFile	Sets the logo image to a Windows BMP file.
padSetPixel	Sets a pixel on the LCD display.
padSetFont	Set the current text font.
padGetFont	Get the current text font.
padSetAutoInking	Set the Auto-inking mode.
padGetFontSize	Get the dimensions of a font size.

**LCD display specifications:**

padIsLcd	Check if an LCD display is available.
padLcdHeight	Get height of LCD display area in pixels.
padLcdHorzDPI	Get horizontal resolution of the LCD display.
padLcdVertDPI	Get vertical resolution of the LCD display.
padLcdWidth	Get width of LCD display area in pixels.

**Memory Operations:**

padMemReset	Reset memory.
padMemGetFree	Get amount of free memory.
padMemGetChecksum	Perform a checksum of memory.
padMemClear	Clear memory content.
padMemDelete	Delete a memory item.
padMemFind	Find an item in memory.
padMemLoadText	Load a string into memory.
padMemLoadBitmap	Load bitmap into memory.
padMemLoadBitmapFile	Loads a bitmap file into memory.

**Miscellaneous:**

padLightOn	Turn on the light.
padLightOff	Turn off the light.
padLeftButton	Check if the left button is down.
padRightButton	Check if the right button is down.
padMiddleButton	Check if the middle button is down.
padHardwareButton	Check if a specific button is down.
padHardwareButton1Enable	Enables/disables hardware button 1.

**Model and Version Operations:**

padComDate	Get the build date of the PadCom library
padComVersion	Get the version number of the PadCom library
padGetCmdSetID	Get the current command set ID.
padGetModel	Get the model number of the attached pad
padGetVersion	Get the model revision number

**Pad specifications:**

padGetPage	Get pad dimension information.
padHeight	Get height of pad area in pixels.
padHorzDPI	Get horizontal resolution of the pad.

padVertDPI                   Get vertical resolution of the pad.  
padWidth                    Get width of pad area in pixels.

**Port configuration for DOS:**

padGetPortAddr            Gets a COM port's address.  
padGetPortIrq             Gets a COM port's interrupt number.  
padSetPortAddr            Sets a COM port's address.  
padSetPortIrq             Sets a COM port's interrupt number.

**Port Operations:**

padGetPort                 Gets the current communications port.  
padGetPorts                Gets the current number of ports available.  
padPassThroughHandshaking Enables/disables hardware handshaking (pass through).  
padSetPort                 Sets the preferred communications port.  
padSetPortHandle          Sets the port handle.  
padSetPorts                Sets the amount of ports available on the PC.

**POS Services:**

padGetMagTrack            Gets a track of data from a magnetic card.  
padGetMaxCardTracks       Get the amount of tracks supported.  
padGetMaxCardTrackSize    Get the maximum card track size in bytes.  
padGetAllMagCardTracks    Read all tracks on a magnetic card.  
padGetPIN                 Gets a PIN number from the user. (obsolete)  
padGetStdPIN               Gets a PIN number from the user. (obsolete)  
padGetDUKPTbinaryPIN      Gets a PIN number from the user.  
padGetDUKPTtextPIN        Gets a PIN number from the user.  
padGetMasterSessionBinaryPIN Gets a PIN number from the user.  
padGetMasterSessionTextPIN Gets a PIN number from the user.  
padResetMagCard            Resets the magnetic card reader and clears the track buffer

**Prompt Operations:**

padPromptReset            Reset all prompts.  
padPromptHexNumber        Display a hexadecimal number entry prompt.  
padPromptNum              Display an integer number entry prompt.  
padPromptNumber            Display a decimal number entry prompt.  
padPromptSignatruue       Display a signature capture prompt.  
padPromptString            Display an alphanumeric data entry prompt.  
padPromptTimeout          Sets the timeout value for the prompt commands.

**Scaling of Coordinates Received:**

padScale                  Scale an arbitrary value to a given fraction.  
padScaleDPI                Scale to a desired DPI resolution.  
padScaleTo                 Scale to a desired rectangle.  
padScaleX                  Scale a horizontal coordinate to given DPI.  
padScaleY                  Scale a vertical coordinate to given DPI.  
padToHIENGLISH            Scale to units based on 1000ths of inch.  
padToLOENGLISH            Scale to units based on 100ths of inch.  
padToHIMETRIC             Scale to units based on 100ths of a millimeter.  
padToLOMETRIC             Scale to units based on 10ths of a millimeter.

**Scan Rate Operations:**

padSetScanRate                   Set a scan rate.  
padGetScanRate                   Get the current scan rate.

**Sound Operations:**

padSoundBell                    Type of sound action to make.  
padSoundEnable                 Enable or disable the sound.  
padSoundSetFreq                The frequency for the sound.  
PadSoundTone                    Sound with a frequency and duration.

**State information:**

padIsOn                         Check if pad has been turned on.  
padIsNewStroke                 Check if beginning a new stroke.  
padIsPenDown                  Check if the pen is down.  
padIsRecord                    Check if currently recording pad data.  
padOldX                         Returns the previous horizontal pen coordinate.  
padOldY                         Returns the previous vertical pen coordinate.  
padNewX                         Returns the current horizontal pen.  
padNewY                         Returns the current vertical pen coordinate.  
padName                         Get the name of the attached pad.  
padType                         Get the type of the attached pad.

**System & Communication Operations:**

padEchoComm                    Echo data back to the host.  
padGetBaudRate                 Returns the current baud rate.  
padGetConnectTimeout          Gets the initial connection input/output timeout value.  
padGetDefaultBaudRate         Returns the current default baud rate.  
padGetInTimeout                Gets the timeout value for input.  
padGetOutTimeout               Gets the timeout value for output.  
padPassThroughOff              Turns off passthrough mode  
padPassThroughOn               Turns on passthrough mode  
padPassThroughResetCodes     Resets the passthrough codes  
padPassThroughSetOffCode     Sets the passthrough "ON" code  
padPassThroughSetOnCode      Sets the passthrough "OFF" code  
padReadByte                    Reads a single byte of from the pad.  
padResetBaudRate               Resets the baud rate to the default (9600).  
padResetConnectTimeout        Resets the initial connection input/output timeout value.  
padResetDefaultBaudRate       Resets the default baud rate to PadCom's initial value (9600).  
padResetInTimeout              Resets the timeout value for input.  
padResetOutTimeout             Resets the timeout value for output.  
padSendByte                    Sends a single byte to the pad.  
padSetBaudRate                 Sets the baud rate to a specified rate.  
padSetConnectTimeout          Sets the initial connection input/output timeout value.  
padSetDefaultBaudRate         Sets the default baud rate to open a connection at.  
padSetFlowControl              Enables/disables flow control.  
padSetInTimeout                Sets the timeout value for input.  
padSetOutTimeout               Sets the timeout value for output.  
padSetPadOffset                Set's the pad's touch surface offset.

**Time & Date Operations:**

padDisplayTime                 Displays the time.  
padHideTime                    Hide the time display.

padSetTime	Set the current time.
padGetTime	Get the current time.

For a detailed description of the functions listed, refer to the alphabetic reference of functions in the next section.

## Alphabetic reference of functions

This section provides detailed information about each library function arranged in alphabetic order. The above listing (“Library functions list by category”, page 18) presents a brief overview of these functions based on functional category.

---

### padBinaryGetTable

**WORD padBinaryGetTable( char FAR \* lpszDatabaseName, BYTE FAR \* lpuBuffer, WORD wBufSize, WORD wIndex )**

Parameter	Description
lpszDatabaseName	Pointer to a buffer containing raw signature data.
lpuBuffer	Pointer to the buffer
wBufSize	Size of the buffer
wIndex	The index of the row transferred

#### Description

Get data from a table row

#### Returns

Receive a row data from the binary table.

---

### padBinaryGetVar

**BOOL padBinaryGetVar( char FAR \* pcName, WORD wNameLength, char FAR \* pcBinData, WORD FAR \* pwDataLength)**

Parameter	Description
pcName	Variable name containing binary data.
wNameLength	Name length.
pcBinData	Pointer to a buffer to get binary data.
pwDataLength	Buffer length.

#### Description

Retrieve binary data from the specified memory location.

#### Returns

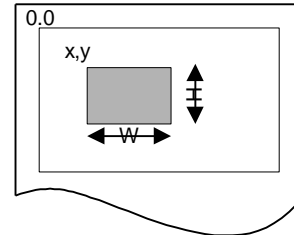
Returns TRUE if successful, FALSE otherwise.

#### See Also

**padBox****BOOL padBox( int X, int Y, int Width, int Height )**

Draws a box onto the LCD display.

Parameter	Description
X	Horizontal coordinate to draw the box from.
Y	Vertical coordinate to draw the box from.
Width	Horizontal size of the box in pixels.
Height	Vertical size of the box in pixels.

**Description**

Draws a solid box on the LCD display starting at the coordinates specified by **X** and **Y** using the size specified by **Width** and **Height**. The box is drawn using the current foreground color. The illustration depicts the placement of an arbitrary box at location **x,y** with width of **W** and height **H**.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padClearPixel, padFrame, padInvert, padLine, padSetPixel

---

**padClear****void padClear( void )****Description**

Clears the entire LCD display if one is attached.

**See Also**

padIsLcd, padPutText, padPutBits, padPutLogo

---

**padClearPixel****BOOL padClearPixel( int X, int Y )**

Clears a pixel on the LCD display.

Parameter	Description
X	Horizontal coordinate of the pixel to set.
Y	Vertical coordinate of the pixel to set.

**Description**

Sets a pixel on the LCD display at the location specified by **X** and **Y** to the current background color.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padBox, padFrame, padInvert, padLine, padSetPixel

---

**padComDate**

**char \* padComDate()**

**Description**

Returns a pointer to a string containing the date of the build of the PadCom library. The format of the returned string is in ASCII text format including the month, day, and year of the build. For example, "July 16, 1998".

**See Also**

padComVersion

---

**padComVersion**

**char \* padComVersion()**

**Description**

Returns a pointer to a string containing the version number of the build of the PadCom library. The format of the returned string is in ASCII text format and includes the major version, minor version, and revision number of the build. For example, "04.00.0008".

**See Also**

padComDate

---

**padConnect**

**BOOL padConnect ( void )**

Connects to the attached @pos.com unit.

**Description**

Checks for the existence of a pad and initializes communications. Normally this command searches all valid COM ports for the existence of all supported @pos.com pad types. This command must be executed before using any other commands in the library; the only exceptions to this are the commands **padSetPort**, **padSetPortAddr**, **padSetPortIrq**, and **padSetType**, which modify the behavior of **padConnect** and must be called prior to calling **padConnect**. This function is similar to

---

**padOn** except that it will always attempt to connect to a pad even if a connection was already established.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padConnectClearScreen, padOn, padOff, padRecord, padSetPort, padSetPortAddr, padSetPortIrq, padSetType, padSetBaudRate, padGetBaudRate, padSetDefaultBaudRate, padGetDefaultBaudRate, padResetDefaultBaudRate, padResetBaudRate

---

**padConnectClearScreen****BOOL padConnectClearScreen ( BOOL enable )**

Parameter	Description
enable	Flag to enable/disable screen clear on Connect.

**Description**

padConnectClearScreen allows you to specify if or not the screen is to be cleared upon connection. The default is for padConnectClearScreen to be set to TRUE. When set to TRUE the LCD is cleared whenever a connection is made to the @pos.com POS device using padConnect or padOn. If set to FALSE the LCD is not cleared when a connection is made to an @pos.com POD device. This command effects all subsequent calls to padOn and padConnect and does not itself make a connection to the @pos.com POS device. To make a connection you must call padConnect or padOn.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padOn, padOff, padConnect

---

**padDisplayObject****BOOL padDisplayObject( WORD ObjectId, padPOINT \*pptAt )**

Draws a memory object onto the position specified by pptAt on the screen, if applicable.

Parameter	Description
ObjectId	ID of the memory item to store
pptAt	Pointer to padPOINT specifying screen location

**Description**

This command displays an object at the given location on the display. An object can be stored in non-volatile memory using **padMemLoadText**, **padMemLoadBitmap** or similar commands. Every

memory object is associated with an **ObjectId** that identifies the object. The created memory object may be deleted or overwritten.

Non-volatile memory is typically used to store large and often used bitmaps and texts. Since, the data resides on the unit, the data transfer between the COM-port and the unit is reduced resulting in much faster display times.

NOTE: All memory objects may be displayed, even empty ones. To check if a specified memory object contains data use **padMemFind**.

#### **Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

#### **See Also**

padMemLoadText, padMemLoadBitmap, padMemDelete

---

### **padDisplayTime**

**BOOL padDisplayTime( WORD wHorz, WORD wVert, BYTE nFontId )**

<b>Parameter</b>	<b>Description</b>
wHorz	the horizontal position for displaying the time
wVert	the vertical position for displaying the time
nFontId	the font used for displaying the time

#### **Description**

This function displays the time on a PenWare 3000 or compatible terminal.

#### **Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

#### **See Also**

padHideTime, padSetTime, padGetTime, padSetFont

---

### **padEchoComm**

**int padEchoComm( char \*SendDataBlock, char \*RecvDataBlock, WORD DataLength)**

Echoes data back to the host ( the PC ).

<b>Parameter</b>	<b>Description</b>
SendDataBlock	Pointer to the block of data to send
RecvDataBlock	Pointer to the received data block
DataLength	Length of the sent data bytes

The user sends a block of data specified by **SendDataBlock** (with the length of the data specified by **DataLength**) to the pad and the pad sends the data back to the host which is then placed in **RecvDataBlock**. This command can be used to verify the communication link between the host and the pad.

This command can optionally be used to synchronize the software with the state of the unit by waiting for the echo to return before continuing.

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

---

**padEraseTable**

**BOOL padEraseTable( char FAR \* lpszDatabaseName )**

<b>Parameter</b>	<b>Description</b>
lpszDatabaseName	Name of the database variable that stores the transactions

**Description**

Erase all data in a binary table

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

---

**padFieldButton**

**BOOL padFieldButton( WORD wId, WORD wStyle, WORD wLeft, WORD wTop, WORD wWidth, WORD wHeight, BYTE uFont, char FAR \*pcText, WORD wTextLen, WORD wCmdPress, char FAR \* pcPressArgData, WORD wPressArgDataLen, WORD wCmdRelease, char FAR \* pcReleaseArgData, WORD wReleaseArgDataLen )**

<b>Parameter</b>	<b>Description</b>
wId	Field identification number.
wStyle	Field style attributes. 0 for defaults.
wLeft	X-coordinate of upper point
wTop	Y-coordinate of upper point
wWidth	Width of the signature field.
wHeight	Height of the signature field.
uFont	Font to be used.
pcText	Button text to be displayed.
wTextLen	Text length.
wCmdPress	Command to be executed when the button is pressed.

---

pcPressArgData	Command data for press command.
wPressArgDataLen	Data length for press command.
wCmdRelease	Command to be executed when the button is released.
pcReleaseArgData	Command data for release command.
wReleaseArgDataLen	Data length for release command.

**Description**

Creates a button and display it on the pad screen. Commands can be executed when button is pressed and released.

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

---

**padFieldSignature**

**BOOL padFieldSignature ( WORD wId, WORD wStyle, WORD wLeft, WORD wTop,  
WORD wWidth, WORD wHeight, WORD wMaxPoints,  
WORD wEnterTime, WORD wEnterCmd,  
char FAR \* pcData, int nDataLen)**

<b>Parameter</b>	<b>Description</b>
wId	Field identification number.
wStyle	Field style attributes. 0 for defaults.
wLeft	X-coordinate of upper point
wTop	Y-coordinate of upper point
wWidth	Width of the signature field.
wHeight	Height of the signature field.
wMaxPoints	Max points that are allowed to be entered to the signature field.
wEnterTime	Penup time out in seconds, which is used for AUTO ENTER.
wEnterCmd	The command to be executed when ENTER button is pressed.
pcData	Command data buffer.
nDataLen	Length of the command data.

**Description**

Creates a signature field.

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

---

**padFlush**

**void padFlush( void )**

Flush the input queue.

**Description**

Remove any pending pad data from the input queue. This is not normally needed however it may be useful in some situations.

---

**padFormDeleteFld****BOOL padFormDeleteFld(WORD wId)**

<b>Parameter</b>	<b>Description</b>
wId	Field identification number.

**Description**

Delete the field given by wId.

**See Also**

padFormSaveFld

---

**padFormSaveFld****BOOL padFormSaveFld(WORD wId, char FAR \* pcName, WORD wNameLen)**

<b>Parameter</b>	<b>Description</b>
wId	Field identification number.
pcName	Variable name
wNameLen	Name length.

**Description**

Save the current signature data in to the specified variable.

**See Also**

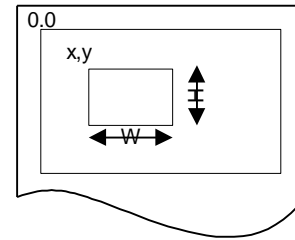
padBox, padClearPixel, padInvert, padLine, padSetPixel

---

**padFrame****BOOL padFrame( int X, int Y, int Width, int Height )**

Draws a frame onto the LCD display. This function draws an empty frame as opposed to **padBox** which draws a filled frame.

Parameter	Description
X	Horizontal coordinate to draw the frame from.
Y	Vertical coordinate to draw the frame from.
Width	Horizontal size of the frame in pixels.
Height	Vertical size of the frame in pixels.



### Description

Draws a frame (outline/empty box) on the LCD display starting at the coordinates specified by **X** and **Y** using the size specified by **Width** and **Height**. The frame is drawn using the current foreground color.

The illustration depicts the placement of an arbitrary frame at location **x,y** with width of **W** and height **H**.

### Returns

Returns TRUE if successful, FALSE otherwise.

### See Also

padBox, padClearPixel, padInvert, padLine, padSetPixel

---

## padGet

**BOOL padGet ( int \*XPosition, int \*YPosition, int \*Pen )**

Receives new data from the pad.

Parameter	Description
XPosition	Pointer to an integer to hold the new horizontal coordinate of the pen.
YPosition	Pointer to an integer to hold the new vertical coordinate of the pen.
Pen	Pointer to an integer to hold the new pen status (0=moveto, 1=lineto).

### Description

Checks for and returns the last horizontal/vertical coordinates and pen status. A pen status of 1 indicates the continuation of a line (LineTo(x,y)) whereas a 0 represents the beginning of a stroke (MoveTo(x,y)). NULL pointers may be passed as any of the parameters to suppress gathering of the associated data. Note that if no pen activity occurred since the last **padUpdate**, the functions returns FALSE and does not alter the values pointed to.

### Returns

Returns TRUE if new pen data was received, FALSE otherwise.

### See Also

padUpdate, padRecord, padStop

---

**padGetAllMagCardTracks**

```
BOOL padGetAllMagCardTracks( BYTE *Tracks,      char *Track1,  
                               WORD *Track1Size,  char *Track2,  
                               WORD *Track2Size,  char *Track3,  
                               WORD *Track3Size  )
```

Returns data captured from all tracks on a magnetic card, if available.

<b>Parameter</b>	<b>Description</b>
Tracks	total tracks
Track1	pointer to the track1
Track1Size	pointer to the track1 size
Track2	pointer to the track2
Track2Size	pointer to the track2 size
Track3	pointer to the track3
Track3Size	pointer to the track3 size

A card must have been swiped prior to executing this command otherwise it returns zero. The **TrackXSize** parameters should contain the maximum buffer sizes for each track before the call, and after will contain actual track sizes after the call. The card reader must be reset (using **padResetMagCard**) before calling this command. Please refer to Appendix C on page 111 for sample code that describes how to read the MSR.

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

**See Also**

padGetMaxCardTrackSize, padGetMaxCardTracks, padResetMagCard, padGetMagTrack

---

**padGetArea**

```
BOOL padGetArea( int *xLeft, int *yTop, int *xRight, int *yBottom )
```

Retrieves the minimum and maximum coordinates of the active pad area.

<b>Parameter</b>	<b>Description</b>
xLeft	Pointer to an integer to hold the minimum horizontal coordinate.
yTop	Pointer to an integer to hold the minimum vertical coordinate.
xRight	Pointer to an integer to hold the maximum horizontal coordinate.
yBottom	Pointer to an integer to hold the maximum vertical coordinate.

**Description**

Retrieves the current clipping area into the variables provided. NULL pointers may be passed as any or all parameters.

**Returns**

Returns TRUE if area retrieved represents the full pad surface, FALSE otherwise.

**See Also**

padSetArea, padWidth, padHeight

---

**padGetBaudRate**

**DWORD padGetBaudRate( void )**

**Description**

Used to find out the current communications baud rate.

**Returns**

Returns the current baud rate.

**See Also**

padResetBaudRate, padSetBaudRate

---

**padGetBkColor**

**int padGetBkColor( void )**

Gets the current background color.

**Description**

Gets the current background color.

**Returns**

Returns the current background.

**See Also**

padGetColor, padGetColors, padSetBkColor, padSetColor

---

**padGetCmdSetID**

**WORD padGetCmdSetID( void )**

Gets the command set ID number.

The command set ID number for a standard PenWare3000 is "1234". The command set ID number is a code that identifies the type of command set used and is not to be confused with the version of the command set. This command is for future purposes only.

---

**Returns**

Returns the ID of the command set upon success, FALSE otherwise.

**See Also**

padGetVersion

---

**padGetColor**

**int padGetColor( void )**

Gets the current foreground color.

**Description**

Gets the current foreground color.

**Returns**

Returns the current foreground color.

**See Also**

padGetBkColor, padGetColors, padSetBkColor, padSetColor

---

**padGetColors**

**int padGetColors( void )**

Gets the number of colors available.

**Description**

Gets the number of colors available.

**Returns**

Returns the number of colors available.

**See Also**

padGetBkColor, padGetColor, padSetBkColor, padSetColor

---

**padGetConnectTimeout**

**unsigned int padGetConnectTimeout( void )**

**Description**

This function returns the time-out value in milliseconds used for the COM port during the initial connection phase.

**See Also**

padSetInTimeout, padResetInTimeout, padGetOutTimeout, padResetOutTimeout,  
padSetConnectTimeout, padResetConnectTimeout

---

**padGetDefaultBaudRate**

**DWORD padGetDefaultBaudRate( void )**

Gets the current default baud rate. See **padSetDefaultBaudRate** for more information.

**Returns**

Returns the current default baud rate.

**See Also**

padSetBaudRate, padGetBaudRate, padResetBaudRate, padSetDefaultBaudRate,  
padResetDefaultBaudRate

---

**padGetDUKPTbinaryPIN**

**int padGetDUKPTbinaryPIN**  
(  
    **char FAR \* pTitle,**  
    **char FAR \* pAcctNum,**  
    **char FAR \* pKeySerialNumber,**  
    **int \* pKeySerialNumberLength,**  
    **char FAR \* pBinaryPIN,**  
    **int iTimeout**  
)

<b>Parameter</b>	<b>Description</b>
pTitle	specifies a string to display as the title of the PIN entry screen.
pAcctNum	user's account number used to generate the encrypted PIN number.
pKeySerialNumber	generated key serial number used to decode the encrypted PIN number.
pKeySerialNumberLength	length of the generated key serial number.
pBinaryPIN	the resulting binary PIN number returned as a byte array not an actual string.
iTimeout	specifies the maximum amount of seconds this function attempts to obtain PIN data from the PIN entry prompt. If the specified amount of seconds elapse without user input then the PIN entry prompt is canceled and the display is cleared.

**Description**

Displays a PIN entry prompt for collecting VISA standard encrypted pin numbers. This function is only for pads such as the 3000 and 3100 that support VISA standard DUKPT PIN entry prompts. A key must be injected into the unit before this function can be used. Please contact your hardware supplier or @pos.com for information on secure key injection.

**Returns**

- 0 - Success
- 1 - Cancel
- 2 - Bad Parameters
- 3 - Pad Not On
- 4 - Command Error
- 5 - Response Error
- 6 - No PIN
- 7 - Not Supported

**See Also**

padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN, padGetMasterSessionTextPIN

---

**padGetDUKPTtextPIN**

```
int padGetDUKPTtextPIN
(
    char FAR * pTitle,
    char FAR * pAcctNum,
    char FAR * pTextPIN,
    int      iTimeout
)
```

Parameter	Description
pTitle	specifies a string to display as the title of the PIN entry screen.
pAcctNum	the user's account number used to generate the encrypted PIN number.
pTextPIN	the resulting standard ASCII/VISA standard type "71" text PIN block which includes both the encrypted PIN and the key serial number used to create the encrypted PIN.
iTimeout	specifies the maximum amount of seconds this function attempts to obtain PIN data from the PIN entry prompt. If the specified amount of seconds elapse without user input then the PIN entry prompt is canceled and the display is cleared.

**Description**

Displays a PIN entry prompt for collecting VISA standard encrypted pin numbers. This function is only for pads such as the 3000 and 3100 that support standard DUKPT PIN entry prompts. One DUKPT key must be injected into the unit before this function can be used. Please contact your hardware supplier or @pos.com for information on secure key injection.

**Returns**

- 0 - Success
- 1 - Cancel
- 2 - Bad Parameters

- 3 - Pad Not On
- 4 - Command Error
- 5 - Response Error
- 6 - No PIN
- 7 - Not Supported

**See Also**

padGetDUKPTbinaryPIN, padGetMasterSessionBinaryPIN, padGetMasterSessionTextPIN

---

**padGetFont**

**int padGetFont( void )**

Gets the current text font.

**Returns**

Returns the current font number. The FontIDs corresponds to the sizes as specified in the table below:

Font Id	Horizontal Size	Vertical Size	Available for PenWare2000	Available for PenWare3000/3100
0	8	8	YES	YES
1	16	16	YES	YES
2	6	8	NO	YES
3	8	12	NO	YES
4	12	16	NO	YES
5	16	24	NO	YES

**See Also**

padSetFont, padPutText

---

**padGetFontSize**

**int padGetFontSize( BYTE Id, padPOINT \* pptPixels )**

Get the dimensions of a specified font.

Parameter	Description
Id	Id of the specified font.
pptPixels	Pointer to store the obtained font size.

This command returns the horizontal and vertical size of the character dimensions used in the specified font. For example, on a PenWare3000, in font number 0 each character is 8 pixels wide and 8 pixels high. Calling this command with the **Id** of 0 will return 8x8 (in **pptPixels**).

The following table describes the relationship between font ID and the font sizes.

Font Id	Horizontal	Vertical	Available for	Available for
---------	------------	----------	---------------	---------------

---

	Size	Size	PenWare2000	PenWare3000/3100
0	8	8	YES	YES
1	16	16	YES	YES
2	6	8	NO	YES
3	8	12	NO	YES
4	12	16	NO	YES
5	16	24	NO	YES

**Returns**

Returns TRUE if function call succeeds and FALSE otherwise

**See Also**

padSetFont, padGetFont

---

**padGetInTimeout**

**unsigned int padGetInTimeout( void )**

**Description**

This function returns the current time-out value in milliseconds used for the COM port's input from the connected pad. This is not related to the time-out used during the initial connection phase (see padSetConnectTimeout).

**See Also**

padSetInTimeout, padResetInTimeout, padGetOutTimeout, padSetOutTimeout, padResetOutTimeout, padSetConnectTimeout, padGetConnectTimeout, padResetConnectTimeout

---

**padGetMagTrack**

**unsigned padGetMagTrack( int Track, char \*Buf, unsigned Size )**

Retrieves a track from a magnetic card.

Parameter	Description
Track	The number of the track to be read.
Buf	Pointer to a character buffer to receive the track data.
Size	Maximum number of characters to read into <i>Buf</i> .

**Description**

This functions provides low level access to the magnetic card reader, if attached. When reading track 1, 2, or 3, the function returns the number of bytes actually read and the **Buf** parameter contains the data read, if any.

Before reading a track, you must reset the magnetic card reader. To reset the magnetic card reader you can call **padResetMagCard** or optionally you can call this function using 0 for all of the

parameters. The return value will be nonzero if a reader is attached, otherwise it will be zero. Please refer to **padResetMagCard** for more information.

Please refer to Appendix C on page 111 for a sample code that describes how to read an MSR.

### Returns

When *Track* is 0, returns nonzero if a reader is attached, zero otherwise.

Other *Track* values return the number of bytes read from the desired track, if any.

### See Also

padResetMagCard, padGetMaxCardTrackSize, padGetMaxCardTracks, padGetAllMagCardTracks  
padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN,  
padGetMasterSessionTextPIN

---

## padGetMasterSessionBinaryPIN

### int padGetMasterSessionBinaryPIN

```
(  
    char FAR * pTitle,  
    char FAR * pAcctNum,  
    char FAR * pSessionKey,  
    int      pMasterKeyID,  
    char FAR * pBinaryPIN,  
    int      iTimeout  
)
```

Parameter	Description
pTitle	specifies a string to display as the title of the PIN entry screen.
pAcctNum	the user's account number used to generate the encrypted PIN number.
pSessionKey	the session key used to generate the encrypted PIN number.
pMasterKeyID	used to identify which of the 10 (0-9) injected master keys to use to generate the encrypted PIN number.
pBinaryPIN	is the resulting binary PIN number returned as a byte array not an actual string.
iTimeout	specifies the maximum amount of seconds this function attempts to obtain PIN data from the PIN entry prompt. If the specified amount of seconds elapse without user input then the PIN entry prompt is canceled and the display is cleared.

### Description

Displays a PIN entry prompt for collecting VISA standard encrypted pin numbers. This function is only for pads such as the 3000 and 3100 that support standard Master/Session PIN entry prompts. 10 master keys must be injected into the unit before this function can be used. Please contact your hardware supplier or @pos.com for information on secure key injection.

### Returns

- 0 - Success
- 1 - Cancel
- 2 - Bad Parameters

- 3 - Pad Not On
- 4 - Command Error
- 5 - Response Error
- 6 - No PIN
- 7 - Not Supported

**See Also**

padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionTextPIN

---

**padGetMasterSessionTextPIN**

```
int padGetMasterSessionTextPIN
(
    char FAR * pTitle,
    char FAR * pAcctNum,
    char FAR * pSessionKey,
    int      pMasterKeyID,
    char FAR * pTextPIN,
    int      iTimeout
)
```

**Parameter**      **Description**

pTitle	specifies a string to display as the title of the PIN entry screen.
pAcctNum	the user's account number used to generate the encrypted PIN number.
pSessionKey	the session key used to generate the encrypted PIN number.
pMasterKeyID	used to identify which of the 10 (0-9) injected master keys to use to generate the encrypted PIN number.
pTextPIN	the resulting standard ASCII/VISA standard type "71" text PIN block.
iTimeout	specifies the maximum amount of seconds this function attempts to obtain PIN data from the PIN entry prompt. If the specified amount of seconds elapse without user input then the PIN entry prompt is canceled and the display is cleared.

**Description**

Displays a PIN entry prompt for collecting VISA standard encrypted pin numbers. This function is only for pads such as the 3000 and 3100 that support standard Master/Session PIN entry prompts. 10 master keys must be injected into the unit before this function can be used. Please contact your hardware supplier or @pos.com for information on secure key injection.

**Returns**

- 0 - Success
- 1 - Cancel
- 2 - Bad Parameters
- 3 - Pad Not On
- 4 - Command Error
- 5 - Response Error
- 6 - No PIN
- 7 - Not Supported

**See Also**

padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN

---

**padGetMaxCardTracks****BOOL padGetMaxCardTracks( BYTE \*Tracks )**

Retrieves the number of tracks supported by the magnetic card reader in **Tracks**. Please refer to Appendix C on page 111 for a sample code that describes how to read an MSR.

<b>Parameter</b>	<b>Description</b>
Tracks	Pointer to a BYTE data type that returns the number of supported tracks.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padGetMaxCardTrackSize, padGetAllMagCardTracks

---

**padGetMaxCardTrackSize****BOOL padGetMaxCardTrackSize( WORD \*TrackSize )**

Retrieves the maximum readable card track size supported by the magnetic card reader. Please refer to Appendix C on page 111 for a sample code that describes how to read an MSR.

<b>Parameter</b>	<b>Description</b>
TrackSize	Pointer to a WORD data type that returns the maximum readable track size.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padGetMaxCardTracks, padGetAllMagCardTracks

---

**padGetModel****BOOL padGetModel( WORD \*Model )**

Retrieves the model ID.

**Description**

The ID is returned as a 16-bit integer in the WORD pointer **Model**. For a PenWare3000 the model ID returned is 3000. This is for future use only.

---

<b>Parameter</b>	<b>Description</b>
Model	Pointer to a WORD data type that returns the model ID.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padGetVersion

---

**padGetNumVar**

**BOOL padGetNumVar( char FAR \*lpszVarName, WORD FAR \*pwValue )**

**Description**

Get the contents of a variable in num format

<b>Parameter</b>	<b>Description</b>
lpszVarName	Name of the variable to assign the numeric value to
pwValue	Value

**Returns**

Returns TRUE upon success, FALSE otherwise.

---

**padGetOutTimeout**

**unsigned int padGetOutTimeout( void )**

**Description**

This function returns the current time-out value in milliseconds used for the COM port's output to the connected pad. This is not related to the time-out used during the initial connection phase (see padSetConnectTimeout).

**See Also**

padSetOutTimout, padResetInTimeout, padGetOutTimeout, padSetInTimeout, padResetOutTimeout, padSetConnectTimeout, padGetConnectTimeout, padResetConnectTimeout

---

**padGetPage**

**void padGetPage( int \*Width, int \*Height, int \*HorzDPI, int \*VertDPI )**

Retrieves the size and resolution of the @pos.com POS device's touch sensitive surface.

<b>Parameter</b>	<b>Description</b>
------------------	--------------------

---

Width	Pointer to hold the total amount of horizontal touch sensitive points.
Height	Pointer to hold the total amount of vertical touch sensitive points.
HorzDPI	Pointer to hold the horizontal dots-per-inch of the touch sensitive surface.
VertDPI	Pointer to hold the vertical dots-per-inch of the touch sensitive surface.

**Description**

This function retrieves the total amount of touch points available on the @pos.com POS device's touch surface area in the locations pointed to by **Width** and **Height**. It retrieves the dots per inch resolution of the @pos.com POS device's touch surface area in the locations pointed to by **HorzDPI** and **VertDPI**. NULL pointers may be passed for any parameter to suppress gathering the associated data. The coordinates retrieved are not display coordinates, they are touch surface coordinates. In general, touch surface coordinates are much higher in resolution than display coordinates. For example the 3100's touch surface contains 4096x4096 touch points while the 3100's display contains only 320x240 pixels, even though they are about the same size in inches.

**See Also**

padHeight, padWidth, padHorzDPI, padVertDPI

---

**padGetPIN**

**BOOL padGetPIN( padPINDATA \*PIN, int Timeout )**

This function is obsolete and does nothing. Use padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN, or padGetMasterSessionTextPIN.

**Description**

DO NOT USE THIS COMMAND. This command is obsolete. Instead use padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN, or padGetMasterSessionTextPIN.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padGetMagTrack, padGetDUKPTbinaryPIN, padGetDUKPTtextPIN,  
padGetMasterSessionBinaryPIN, padGetMasterSessionTextPIN

---

**padGetPort**

**int padGetPort( void )**

Returns the active port.

**Description**

If the pad is in “off” state, the port number returned represents the “default port” as mentioned in **padSetPort**. If the pad is currently “on” (after a successful call to **padOn** or **padConnect** functions), the port number is the actual port in use.

**Returns**

Returns an integer representing one of the following ports:

0 for none, 1 for COM1, 2 for COM2, 3 for COM3 or 4 for COM4.

**See Also**

padSetPort, padSetPorts, padGetPorts

---

**padGetPortAddr**

**DOS: int padGetPortAddr( int Port )**

Returns the address of a port.

NOTE: This function is available in the DOS version only!

<b>Parameter</b>	<b>Description</b>
Port	Should be 1 for COM1, 2 for COM2, 3 for COM3 or 4 for COM4.

**Returns**

Returns the address of the specified communications port.

**See Also**

padSetPortAddr, padGetPortIrq

---

**padGetPortIrq**

**DOS: int padGetPortIrq( int Port )**

Returns the interrupt number associated with a port.

NOTE: This function is available in the DOS version only!

<b>Parameter</b>	<b>Description</b>
Port	Should be 1 for COM1, 2 for COM2, 3 for COM3 or 4 for COM4

**Returns**

Returns the interrupt number used by the specified communications port.

**See Also**

padSetPortIrq, padGetPortAddr

---

**padGetPorts**

**int padGetPorts( void )**

Returns the current number of available ports.

**Description**

Use padSetPorts to change the number of ports available.

**Returns**

The return values are 1 for 1 port, 2 for 2 ports, etc.

**See Also**

padSetPort, padSetPorts

---

**padGetScanRate****int padGetScanRate( void )**

Get the current scan rate. Scan rate is used to vary the number of points captured per second.

**Returns**

Returns the current scan rate if succeeds, FALSE if function fails

**See Also**

padSetScanRate

---

**padGetStdPIN**

**BOOL padGetStdPIN( padSTDPINDATA \*PinData, char \*Title, char \*AcctNum,  
int MaxSecs )**

<b>Parameter</b>	<b>Description</b>
PinData	Pointer to the result.
Title	Pointer to the title prompt.
AcctNum	Pointer to the NULL-terminated account number
MaxSecs	Maximum seconds before timeout

**Description**

DO NOT USE THIS COMMAND. This command is obsolete. Instead use padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN, or padGetMasterSessionTextPIN.

**Returns**

Return conditions are interpreted as follows:

<b>Return Code</b>	<b>Value</b>	<b>Description</b>
STDPIN_SUCCESS	1	Success.

---

STDPIN_CANCEL	0	Pin prompt canceled
STDPIN_ERR_BADPTR	-1	One of the pointers is bad
STDPIN_ERR_PADNOTON	-2	Pad is not on
STDPIN_ERR_CMD	-3	Command format error
STDPIN_ERR_RESULT	-4	Error in obtaining result
STDPIN_ERR_NOPIN	-5	Timed out.

**See also**

padGetDUKPTbinaryPIN, padGetDUKPTtextPIN, padGetMasterSessionBinaryPIN,  
padGetMasterSessionTextPIN

---

**padGetTableItem**

**WORD padGetTableItem( char FAR \* lpszDatabaseName, BYTE FAR \* lpuBuffer,  
WORD wBufSize, WORD wIndex )**

<b>Parameter</b>	<b>Description</b>
lpszDatabaseName	Name of the database variable that stores the transaction
lpuBuffer	Pointer to the buffer
wBufSize	Size of the buffer
wIndex	Id of the index into the table used as a binary

**Description**

Transfer data from a table row to a variable

**Returns**

Total bytes read.

---

**padGetTime**

**BOOL padGetTime( BYTE FAR \* pHour, BYTE FAR \* pMin, BYTE FAR \* pSec )**

<b>Parameter</b>	<b>Description</b>
pHour	a pointer to a byte for receiving the current hour
pMin	a pointer to a byte for receiving the current minute
pSec	a pointer to a byte for receiving the current second

**Description**

This function gets the current time on the PenWare3000 or compatible POS terminal.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padDisplayTime, padSetTime, padHideTime

---

---

**padGetVersion****BOOL padGetVersion( WORD \*Version )**

Retrieves the model revision number of the firmware.

<b>Parameter</b>	<b>Description</b>
Version	Pointer to WORD data type to store the result

**Description**

The revision number is returned in the following format:

HIBYTE	Major revision number as binary value
LOBYTE	Minor revision number as binary value

This command allows a user to check the version number of the firmware (ROM chips in the pad). For example, version 1.0 would be returned as 0100h, 2.15 would be returned as 0215h, 3.1 would be returned as 0310h. This command is useful if you are writing a program that makes use of commands not found on earlier versions of the firmware. With this command your program can check the version number of the firmware in the pad and only run if it's equal to or greater than the version number required.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**padGetModel

---

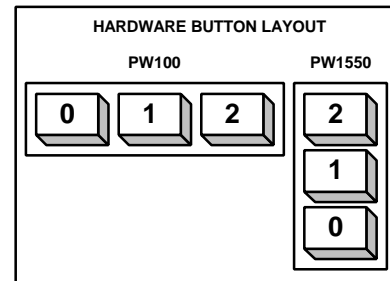
**padHardwareButton****BOOL padHardwareButton( WORD ButtonID )**

Checks if a hardware button is pressed.

<b>Parameter</b>	<b>Description</b>
ButtonID	specifies which hardware button to retrieve that state of.

**Description**

Uses the most recently received pad data to determine if the specified button is pressed. Note that the status is only updated when **padUpdate** returns TRUE to indicate new data received. On a PenWare100 button 0 corresponds to the left button, button 1 corresponds to the middle button and button 2 corresponds to the right button. On a PenWare1500 with the optional buttons button 0 is the bottom button, button 1 is the middle button and button 2 is the top button. The PenWare100 normally does not allow button 1 to be read. To enable the reading of button 1 use **padHardwareButton1Enable**.



### Returns

Returns TRUE if the specified button is pressed, FALSE otherwise.

### See Also

**padUpdate**, **padRecord**, **padLeftButton**, **padRightButton**, **padMiddleButton**, **padHardwareButton1Enable**

---

## padHardwareButton1Enable

### BOOL padHardwareButton1Enable( BOOL Enable )

Enables/disables hardware button 1.

Parameter	Description
Enable	specifies whether to enable button 1 or not.

### Description

Normally the PenWare100 does not allow you to read button 1 (the middle button). Some of the more recent PenWare100's allow button 1 to be read. To enable button 1 to be read set **Enable** to TRUE. To disable button 1 from being read set **Enable** to FALSE. If you are using a PenWare100 that does not allow button 1 to be read then this function will return FALSE if you try to enable button 1. The default is for button 1 to be disabled for a PenWare100. For all other @pos.com devices the default is for button 1 to be enabled.

### Returns

Returns TRUE upon success, FALSE otherwise.

### See Also

**padUpdate**, **padRecord**, **padMiddleButton**, **padHardwareButton**

---

## padHeight

### int padHeight( void )

**Returns**

Returns the total number of vertical points on the pad surface (i.e. 1024).

**See Also**

padWidth, padVertDPI

---

**padHideTime****BOOL padHideTime( void )**

Stops the time from being displayed on a PenWare3000 or compatible terminal.

**Returns**

Returns the total number of vertical points on the pad surface (i.e. 1024).

**See Also**

PadDisplayTime, padGetTime, padSetTime

---

**padHorzDPI****int padHorzDPI( void )****Returns**

Returns the number of horizontal points per inch.

**See Also**

padWidth, padVertDPI

---

**padInkExport****BOOL padInkExport( har FAR \* pcName, WORD wNameLen, WORD wFormat,  
char FAR \* pcFmtData, WORD FAR \* pwFmtLen)**

<b>Parameter</b>	<b>Description</b>
pcName	Variable name containing raw signature data.
wNameLen	Name length.
wFormat	Compression format.
pcFmtData	Pointer to buffer to receive compressed data.
pcFmtLen	Buffer length.

**Description**

Covert the raw signature data into given format.

01 - INK\_POINTS  
02 - INK\_TOKEN  
05 - INK\_PACKET

---

07 - INK\_COTF  
08 - INK\_NOLOSS

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padFieldsignature, padFormSaveFld

---

**padInvert**

**BOOL padInvert( int X, int Y, int Width, int Height )**

Inverts an area on the LCD display.

<b>Parameter</b>	<b>Description</b>
X	Horizontal coordinate to start inverting from.
Y	Vertical coordinate to start inverting from.
Width	Horizontal size in pixels of the area to invert.
Height	Vertical size in pixels of the area to invert.

**Description**

Inverts an area on the LCD display by turning all black pixels white and all white pixels black. The area to be inverted is defined by the coordinates specified by **X** and **Y** using the size specified by **Width** and **Height**.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padBox, padClearPixel, padFrame, padLine, padSetPixel

---

**padIsaReset**

**BOOL padIsaReset ( WORD FAR \* status )**

<b>Parameter</b>	<b>Description</b>
status	The address to contain the ISA status

**Description**

Check if an extension card device is attached

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padWidth, padVertDPI

---

**padIsKey**

**BOOL padIsKey( BYTE \*IsKey )**

**Description**

The function checks if the DUKPT key has been set. The DUKPT key is used to secure the current transactions.

**Parameter**

IsKey

**Description**

Pointer to a BYTE type data that returns TRUE if DUKPT is set and FALSE if it is not.

**Returns**

Returns TRUE upon success, FALSE otherwise.

---

**padIsLcd**

**BOOL padIsLcd( void )**

**Returns**

Returns TRUE if an LCD display is available, FALSE otherwise.

**See Also**

padClear, padPutText, padPutBits

---

**padIsNewStroke**

**BOOL padIsNewStroke( void )**

Checks if the last X,Y coordinate received is the start of a new pen stroke.

**Description**

Uses the most recently received pad data to determine if the most recent point received is the beginning of a new pen stroke. Note that the status is only updated when **padUpdate** returns TRUE to indicate new data received.

**Returns**

Returns TRUE if the last point represents a new stroke, FALSE otherwise.

---

**See Also**

padUpdate, padRecord, padGet, padNewX, padNewY

---

**padIsOn**

**BOOL padIsOn( void )**

**Returns**

Returns TRUE if the pad is on, FALSE otherwise.

**See Also**

padOn, padOff, padIsRecord

---

**padIsPenDown**

**BOOL padIsPenDown( void )**

Checks if the pen is down or “in contact with” the pad.

**Description**

Uses the most recently received pad data to determine if the pen is in contact with the pad. Note that the status is only updated when **padUpdate** returns TRUE to indicate new data received.

**Returns**

Returns TRUE if the pen is down, FALSE otherwise.

**See Also**

padUpdate, padRecord, padGet, padIsNewStroke

---

**padIsRecord**

**BOOL padIsRecord( void )**

**Returns**

Returns TRUE if the pad is currently sending data, FALSE otherwise.

**See Also**

padRecord, padStop, padIsOn

---

**padLcdHeight**

**int padLcdHeight( void )**

---

**Returns**

Returns the total number of vertical points on the LCD display surface.

**See Also**

padIsLcd, padLcdWidth, padLcdHorzDPI, padLcdVertDPI

---

**padLcdHorzDPI**

**int padLcdHorzDPI( void )**

**Returns**

Returns the number of horizontal points per inch of the LCD display.

**See Also**

padIsLcd, padLcdWidth, padLcdVertDPI

---

**padLcdVertDPI**

**int padLcdVertDPI( void )**

**Returns**

Returns the number of vertical points per inch of the LCD display.

**See Also**

padIsLcd, padLcdHeight, padLcdHorzDPI

---

**padLcdWidth**

**int padLcdWidth( void )**

**Returns**

Returns the total number of horizontal points on the LCD display.

**See Also**

padIsLcd, padLcdHeight, padLcdHorzDPI

---

**padLeftButton**

**BOOL padLeftButton( void )**

Checks if the left button is pressed.

**Description**

Uses the most recently received pad data to determine if the left button is pressed. Note that the status is only updated when **padUpdate** returns TRUE to indicate new data received.

**Returns**

Returns TRUE if the left button is pressed, FALSE otherwise.

**See Also**

padUpdate, padRecord, padRightButton, padMiddleButton, padHardwareButton

---

**padLightOff**

**void padLightOff( void )**

Turns off the light.

**Description**

Sends commands to the pad to turn off the red light.

**See Also**

padLightOn

---

**padLightOn**

**void padLightOn ( void )**

Turns on the light.

**Description**

Sends commands to the pad to turn on the red light..

**See Also**

padLightOff

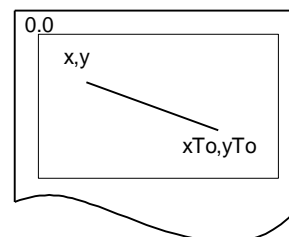
---

**padLine**

**BOOL padLine( int X, int Y, int Xto, int Yto )**

Draws a line on the LCD display.

Parameter	Description
X	Horizontal coordinate to draw the line from.
Y	Vertical coordinate to draw the line from.



Xto                      Horizontal coordinate to draw the line to.  
Yto                      Vertical coordinate to draw the line to.

**Description**

Draws a straight line on the LCD display starting at the location specified by **X** and **Y**, and ending at the location specified by **Xto** and **Yto**. The line is drawn using the current foreground color. The illustration depicts an arbitrary line drawn from a starting point **x,y** to **xTo,yTo**.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padBox, padClearPixel, padFrame, padInvert, padSetPixel

---

**padMemClear****BOOL padMemClear( void )**

Clears the entire contents of the @pos.com POS device's on-board non-volatile memory. Non-volatile memory is typically used to store large and often used bitmaps and text objects. Since, the data resides on the unit, the data transfer between the COM-port and the unit is reduced resulting in much faster display times.

**Returns**

Returns TRUE if the non-volatile memory is cleared , FALSE otherwise.

**See Also**

padMemGetFree, padMemReset, padMemDelete, padMemLoadText, padMemLoadBitmap, padMemFind, padMemGetChecksum

---

**padMemDelete****BOOL padMemDelete( WORD Id )**

Delete the specified item from the non-volatile memory if it exists.

<b>Parameter</b>	<b>Description</b>
Id	Id of the memory item to delete.

This command will delete the specified memory object.

NOTE: All memory objects may be deleted, even empty ones. To check if a specified memory object contains data use **padMemFind**.

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

**See Also**

padMemGetFree, padMemReset, padMemClear, padMemLoadText, padMemLoadBitmap, padMemFind, padMemGetChecksum

---

**padMemDeleteVar**

**BOOL padMemDeleteVar( char FAR \*lpszVarName )**

<b>Parameter</b>	<b>Description</b>
lpszVarName	Name of the memory variable.

**Returns**

Returns TRUE if the function call succeeds, FALSE otherwise.

**See Also**

---

**padMemFind**

**BOOL padMemFind( WORD Id, BYTE \*Stored )**

Checks if an item is stored in memory.

<b>Parameter</b>	<b>Description</b>
Id	ID of the stored memory item
Stored	Pointer to a BYTE that returns TRUE if the memory item is found, FALSE otherwise

This command allows you to check if a memory item, specified by **Id**, is stored in the unit's memory. Non-volatile memory is typically used to store large and often used bitmaps and texts. Since, the data resides on the unit, the data transfer between the COM-port and the unit is reduced resulting in much faster display times. The items are referred to in memory based on their Ids.

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemReset, padMemClear, padMemDelete, padMemLoadText, padMemLoadBitmap, padMemFind, padMemGetChecksum.

---

**padMemGetChecksum**

**BOOL padMemGetChecksum( WORD \*Checksum )**

---

**Checksum** is a pointer to a WORD data type that stores the checksum of all stored items. This is used to verify that memory contents have not been changed since the last call to **padMemGetChecksum**.

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemReset, padMemGetFree, padMemClear, padMemDelete, padMemLoadText, padMemLoadBitmap, padMemFind,

---

**padMemGetFree**

**BOOL padMemGetFree( WORD \*FreeBytes )**

**FreeBytes** is a pointer to WORD type data that returns the number of free non-volatile memory bytes available in the unit. Non-volatile memory is typically used to store large and often used bitmaps and texts. Since, the data resides on the unit, the data transfer between the COM-port and the unit is reduced resulting in much faster display times.

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemReset, padMemClear, padMemDelete,, padMemLoadText, padMemLoadBitmap, padMemFind, padMemGetChecksum

---

**padMemGetVar**

**BOOL padMemGetVar( char FAR \*pcName, WORD wNameLen, char FAR \*pcData, WORD FAR \*pwDataLen)**

<b>Parameter</b>	<b>Description</b>
pcName	Variable name containing the data.
wNameLen	Name length.
pcData	Pointer to a buffer to retrieve data to.
pwDataLen	Buffer length.

**Description**

Retrieve the data stored at specified memory location.

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemSetVar, padFormSaveFld

---

**padMemLoadBitmap**

**BOOL padMemLoadBitmap( WORD Id, padPOINT \*pptSize, BYTE \*Bits )**

Loads a raw unformatted bitmap into the non-volatile memory with the given **Id**. This function will overwrite any item already assigned to **Id**. (Use **padMemFind** to determine if the given **Id** is being used already.) This command does not accept BMP style bitmaps with header data, it only accepts raw bitmap data with no header. The maximum amount of bitmaps that can be loaded depends on how much non-volatile memory is available (see **padMemGetFree**) and how large the bitmaps are.

Parameter	Description
Id	ID of the memory item.
pptSize	horizontal and vertical size of the bitmap in pixels. The maximum horizontal and vertical values allowable for the bitmap are identical to the values returned by <b>padLcdWidth</b> (equivalent to 320 for the PenWare3000) and <b>padLcdHeight</b> (equivalent to 240 for the PenWare3000).
Bits	pointer to the raw bitmap data to load. On the PenWare3000 the bitmap data should be stored using the standard 1 bit per pixel black and white scheme
since	the PenWare3000 has a black and white display only. See Appendix A on page 102.

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemReset, padMemClear, padMemDelete, padMemFind, padMemLoadText, padMemGetFree, padMemGetChecksum, padPutBits, padPutBmpFile, padDisplayObject, padMemLoadBitmapFile

---

**padMemLoadBitmapFile**

**BOOL padMemLoadBitmapFile( WORD Id, char \*FileName )**

Loads a BMP style bitmap file into the non-volatile memory with the given **Id**. This function will overwrite any item already assigned to **Id**. (Use **padMemFind** to determine if the given **Id** is being used already.) Only 1 bit per pixel black and white BMP formats are accepted. The maximum amount of bitmaps that can be loaded depends on how much non-volatile memory is available (see **padMemGetFree**) and how large the bitmaps are.

Parameter	Description
Id	ID of the memory item.
FileName	The name of the black and white BMP file to load.

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemReset, padMemClear, padMemDelete, padMemFind, padMemLoadText, padMemGetFree, padMemGetChecksum, padPutBits, padPutBmpFile, padDisplayObject, padMemLoadBitmap

---

**padMemLoadText**

**BOOL padMemLoadText( WORD Id, char \*Text, WORD TextLength )**

Loads an ASCII text string into the non-volatile memory with the given **Id**. The text can be arbitrarily long. However, @pos.com units do not wrap around the texts. It is the responsibility of the programmer to ensure that the text fits the screen and perform necessary wrap around.

<b>Parameter</b>	<b>Description</b>
Id	ID of the memory item to store
Text	Pointer to the text string to store
TextLength	Length of the text string

**Returns**

Returns TRUE if the function succeeds , FALSE otherwise.

**See Also**

padMemReset, padMemClear, padMemDelete, padMemFind, padMemLoadBitmap, padMemGetFree, padMemGetChecksum

---

**padMemReset**

**BOOL padMemReset( void )**

Used to reset the memory. This command doesn't clear the used memory as opposed to **padMemClear** which clears all user memory.

**Returns**

Returns TRUE if the non-volatile memory is reset , FALSE otherwise.

**See Also**

padMemGetFree, padMemClear, padMemDelete, padMemLoadText, padMemLoadBitmap, padMemFind, padMemGetChecksum

---

**padMemSetVar**

**BOOL padMemSetVar( char FAR \* pcVarName, WORD wNameLen, WORD wCmd, char FAR \*pcCmdData, WORD wDataLen)**

<b>Parameter</b>	<b>Description</b>
pcVarName	Variable name to save the data into.
wNameLen	Name length.
wCmd	Command to be executed to get the data.
pcCmdData	Command data.
wDataLen	Length of the command data.

**Description**

Sets the specified variable with the data retrieved by executing the given command.

**Returns**

Returns TRUE if the non-volatile memory is reset , FALSE otherwise.

**See Also**

padMemGetFree, padMemClear, padMemDelete, padMemLoadText, padMemLoadBitmap, padMemFind, padMemGetChecksum

---

**padMiddleButton****BOOL padMiddleButton( void )**

Checks if the right hardware button is pressed.

**Description**

Uses the most recently received pad data to determine if the middle button is pressed. Note that the status is only updated when **padUpdate** returns TRUE to indicate new data received. The PenWare100 normally does not allow the middle button (button 1) to be read. To enable the reading of the middle button use padHardwareButton1Enable.

**Returns**

Returns TRUE if the right button is pressed, FALSE otherwise.

**See Also**

padUpdate, padRecord, padLeftButton, padRightButton, padHardwareButton, padHardwareButton1Enable

---

**padName****const char \*padName( void )**

Returns the name of the pad attached.

**Description**

Returns a pointer to a string constant containing the name of the pad attached. For example, "PenWare3000" is returned if the attached pad is a PenWare3000, "PENWARE2000" is returned if

the pad attached is a PenWare2000. This differs from **padType** in that **padType** returns a numeric value rather than a string.

**See Also**  
padType

---

### **padNewX**

**int padNewX( void )**

**Returns**

Returns the most recent horizontal coordinate received.

**See Also**  
padGet, padOldX, padNewY

---

### **padNewY**

**int padNewY( void )**

**Returns**

Returns the most recent vertical coordinate received.

**See Also**  
padGet, padOldY, padNewX

---

### **padOff**

**void padOff( void )**

Turns the pad off.

**Description**

Stops any recording, turns off the pad and closes communication channels. The port is closed at the baud rate of the initial connection. The user must close @pos.com devices used by calling this API. Failure to do so, may result in failure to re-connect to the attached unit.

**See Also**  
padOn, padStop, padConnect, padSetBaudRate, padGetBaudRate, padSetDefaultBaudRate, padGetDefaultBaudRate, padResetDefaultBaudRate, padResetBaudRate

---

### **padOldX**

**int padOldX( void )**

---

**Returns**

Returns the previous horizontal coordinate received.

**See Also**

padOldY, padNewX

---

**padOldY**

**int padOldY( void )**

**Returns**

Returns the previous vertical coordinate received.

**See Also**

padOldX, padNewY

---

**padOn**

**BOOL padOn( void )**

Turns the pad on.

**Description**

Checks for the existence of a pad and initializes communications. Normally this command searches all valid COM-ports for the existence of all supported @pos.com pad types. This command must be executed before using the library; the only exceptions to this are the commands **padSetPort**, **padSetPortAddr**, **padSetPortIrq**, and **padSetType**, which modify the behavior of **padOn** and must be called prior to calling **padOn**. This function is similar to **padConnect** except that it will never attempt to connect to a pad if a connection is was already established. The **padOn** command can only be issued once if it succeeds until a **padOff** command is issued. If you wish to reconnect to a pad use the **padConnect** command instead.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padConnect, padConnectClearScreen, padOff, padRecord, padSetPort, padSetPortAddr, padSetPortIrq, padSetType, padSetBaudRate, padGetBaudRate, padSetDefaultBaudRate, padGetDefaultBaudRate, padResetDefaultBaudRate, padResetBaudRate

---

**padPassThroughHandshaking**

**WORD padPassThroughHandshaking ( WORD aHandshaking )**

<b>Parameter</b>	<b>Description</b>
aHandshaking	Yes / No

**Description**

Enables/disables hardware handshaking when in pass through mode.

**Returns**

Previous handshaking mode.

**See Also**

padPassThroughOn, padPassThroughSetOffCode, padPassThroughSetOnCode,  
padPassThroughResetCodes

---

**padPassThroughOff****WORD padPassThroughOff( char \*code, WORD codeLength )**

Turns off pass through mode. While in pass through mode all pads in pass through mode connected together in a chain will monitor the data passing through them. If any of the pads see the padPassThroughOff command and the correct code passed along with it then that pad will disable pass through mode. If the code is not correct for a pad seeing the command then that pad will not process it. It will simply pass the padPassThroughOff command through to its pass through port as if it was normal binary data. Whatever device is connected to its pass through port then will receive the padPassThroughOff command. If it is another pad in pass through mode that is connected to its pass through port and if the code is correct for that pad then that pad will disable its pass through mode.

See padPassThroughSetOffCode and padPassThroughResetCodes for more information on using the correct code for turning off pass through mode. To enable pass through mode use padPassThroughOn.

<b>Parameter</b>	<b>Description</b>
<b>code</b>	a buffer containing the binary code used to turn off pass through mode.
<b>codeLength</b>	the length of the binary pass through code in bytes

**Returns**

Returns nonzero if successful, zero otherwise.

**See Also**

padPassThroughOn, padPassThroughSetOffCode, padPassThroughSetOnCode,  
padPassThroughResetCodes

---

**padPassThroughOn****WORD padPassThroughOn( char \*code, WORD codeLength )**

Turns on pass through mode. To enter pass through mode for a specific pad you must use the correct pass through code for that pad. If the code is not correct then that pad will not enter pass through mode. After correctly enabling pass through mode PadCom can send and receive data to and from a device connected to the pass through port of that pad but cannot send and receive data to or from that pad until the pass through mode of that pad is disabled (see padPassThroughOff).

The device connected to the pass through port of a pad must operate at the same baud rate as the pad or you will not be able to communicate with it. You can use padSetBaudRate before entering pass through mode to set the pad to the same baud rate as the device connected to its pass through port. When connection a series of pads together using the pass through ports it becomes a complex task to manage if all of the pads initially have different baud rates. PadCom does not readily support multiple devices with multiple baud rates...it is best to avoid this situation.

NOTE: Before entering pass through mode it is advised that you set the pass through “OFF” code for that pad. Failure to do so may cause that pad to get stuck in pass through mode (until powered down). If you send the padPassThroughOff command with a “OFF” code that doesn’t match that pad’s “OFF” code then that pad will not process the command and will simply send it to any device connected to its pass through port.

See padPassThroughSetOnCode and padPassThroughResetCodes for more information on using the correct code for turning on pass through mode. To disable pass through mode use padPassThroughOff.

<b>Parameter</b>	<b>Description</b>
<b>code</b>	a buffer containing the binary code used to turn on pass through mode.
<b>codeLength</b>	the length of the binary pass through code in bytes

#### **Returns**

Returns nonzero if successful, zero otherwise.

#### **See Also**

padPassThroughOff, padPassThroughSetOffCode, padPassThroughSetOnCode,  
padPassThroughResetCodes

---

## **padPassThroughResetCodes**

### **WORD padPassThroughResetCodes( void )**

This command resets the pass through “ON” and “OFF” codes of a pad to its factory defaults. The default codes are two bytes in length. The default “ON” code is “AA 55” in hexadecimal (“170 85” in decimal). The default “OFF” code is “55 AA” in hexadecimal (“85 170” in decimal). After using this command you must use the default values given above when enabling or disabling pass through for that pad.

NOTE: You cannot use this command to reset the codes of a pad that is currently in pass through mode. If a pad is in pass through mode it will ignore this command and send it to its pass through port.

See `padPassThroughSetOnCode` and `padPassThroughSetOffCode` for more information on using the correct code for turning on pass through mode.

**Returns**

Returns nonzero if successful, zero otherwise.

**See Also**

`padPassThroughOff`, `padPassThroughSetOffCode`, `padPassThroughSetOnCode`, `padPassThroughOn`

---

**padPassThroughSetOffCode**

**WORD** `padPassThroughSetOffCode( char *code, WORD codeLength )`

This command sets the pass through “OFF” code of a pad. The code can be from 1 to 4 bytes in length. After setting the “OFF” code you must use the same “OFF” code when disabling pass through mode for that pad or you will not be able to disable pass through mode for that pad (see `padPassThroughOff`). You can have a chain of pads connected in series using the pass through ports of each pad. Giving each pad a unique “OFF” code allows you to selectively disable pass through mode for a specific pad. If any two or more pads have the same “OFF” code then `padPassThroughOff` will disable the first pad with that “OFF” code in the chain.

NOTE: You cannot use this command to set the “OFF” code of a pad that is currently in pass through mode. If a pad is in pass through mode it will ignore this command and send it to its pass through port.

<b>Parameter</b>	<b>Description</b>
<b>code</b>	a buffer containing the binary code used to turn off pass through mode.
<b>codeLength</b>	the length of the binary pass through code in bytes

**Returns**

Returns nonzero if successful, zero otherwise.

**See Also**

`padPassThroughOff`, `padPassThroughResetCodes`, `padPassThroughSetOnCode`, `padPassThroughOn`

---

**padPassThroughSetOnCode**

**WORD** `padPassThroughSetOnCode( char *code, WORD codeLength )`

This command sets the pass through “ON” code of a pad. The code can be from 1 to 4 bytes in length. After setting the “ON” code you must use the same “ON” code when enabling pass through mode for that pad or you will not be able to enable pass through mode for that pad (see `padPassThroughOn`). If you have more than two pads connected together in a series using the pass through ports it is not necessary for each pad to have a unique “ON” code, it is however very useful for each pad to have a unique “OFF” code (see `padPassThroughSetOffCode`).

NOTE: You cannot use this command to set the “ON” code of a pad that is currently in pass through mode. If a pad is in pass through mode it will ignore this command and send it to its pass through port.

<b>Parameter</b>	<b>Description</b>
<b>code</b>	a buffer containing the binary code used to turn on pass through mode.
<b>codeLength</b>	the length of the binary pass through code in bytes

**Returns**

Returns nonzero if successful, zero otherwise.

**See Also**

padPassThroughOff, padPassThroughResetCodes, padPassThroughSetOffCode, padPassThroughOn

---

**padPortReclaim**

**BOOL padPortReclaim ()**

**Description**

Reclaims the port

**See Also**

padPortRelease

---

**padPortRelease**

**void padPortRelease ()**

**Description**

Releases the pad

**See Also**

padPortReclaim

---

**padPromptHexNumber**

**BOOL padPromptHexNumber( char \*Title, WORD MaxDigits, char \*Result )**

Displays a hexadecimal keypad and returns the entered hexadecimal number.

Parameter	Description
Title	Specifies the title of the displayed hexadecimal number pad. Its a pointer to a null terminated character string.
MaxDigits	Maximum digits to be displayed.
Result	Returns a variable length text string containing the number entered.

This command prompts the user to enter a hexadecimal value. The prompt includes “ENTER”, “CLEAR”, “C”, “UNDO”, and “CANCEL” buttons. “ENTER” accepts the numeric entry. “CLEAR” and “C” both clear the numeric entry. “UNDO” removes the last digit entered. “CANCEL” exits the prompt. The numeric value can have up to 17 digits. The result is returned as a text string containing the digits entered. The illustration depicts the layout of a HEX number pad.

Title Text				
0	1	2	3	Enter
4	5	6	7	Clear
8	9	A	B	Undo
C	D	E	F	Cancel

### Returns

Returns TRUE upon success, FALSE otherwise.

### See Also

padPromptNum, padPromptNumber, padPromptReset, padPromptSignature, padPromptString, padPromptTimeout

---

## padPromptNum

**BOOL padPromptNum( char \*Title, WORD \*Result )**

Displays a numeric keypad and returns the entered number.

Parameter	Description
Title	Specify the title of the displayed number pad. Its a pointer to a null terminated character string.
Result	Returns a 16-bit value representing the entered number.

This command prompts the user to enter a number whose value can range from 0 to 65,535. The prompt includes “ENTER”, “CLEAR”, “C”, “UNDO”, and “CANCEL” buttons. “ENTER” accepts the numeric entry. “CLEAR” and “C” both clear the numeric entry. “UNDO” removes the last numeric digit entered. “CANCEL” exits the prompt. The illustration depicts the layout of a number pad.

Title Text

1	2	3	Enter
4	5	6	Clear
7	8	9	Undo
C	0		Cancel

### Returns

Returns TRUE upon success, FALSE otherwise.

### See Also

padPromptHexNumber, padPromptNumber, padPromptReset, padPromptSignature, padPromptString, padPromptTimeout

---

## padPromptNumber

**BOOL padPromptNumber ( char \*Title, WORD MaxDigits, WORD Decimal, char \*Result )**

Displays a numeric keypad and returns the entered number.

Parameter	Description
Title	Specify the title of the displayed number pad. Its a pointer to a null terminated character string.
MaxDigits	Maximum digits to be displayed (including decimal point).
Decimal	The amount of digits allowed to the right of the decimal point.
Result	Returns a variable length text string containing the number entered.

This command prompts the user to enter a numeric value. The prompt includes “ENTER”, “CLEAR”, “C”, “UNDO”, and “CANCEL” buttons. “ENTER” accepts the numeric entry. “CLEAR” and “C” both clear the numeric entry. “UNDO” removes the last numeric digit entered. “CANCEL” exits the prompt. Unlike **padPromptNum**, the numeric value can be a decimal value and can have up to 17 digits, including the optional decimal point and is not returned as a 16-bit value, instead the result is returned as a text string containing the digits entered and a decimal point (if any). If **Decimal** equals 0, then no decimal point is allowed. If **Decimal** equals 2, then a total of two digits to the right of the decimal point are allowed, a value of 3 allows three digits to the right of the decimal point and so on. The illustration depicts the layout of a number pad with the decimal point.

Title Text			
<input type="text"/>			
1	2	3	Enter
4	5	6	Clear
7	8	9	Undo
C	0	.	Cancel

#### Returns

Returns TRUE upon success, FALSE otherwise.

#### See Also

padPromptHexNumber, padPromptNum, padPromptReset, padPromptSignature, padPromptString, padPromptTimeout

---

## padPromptReset

### BOOL padPromptReset( void )

This command resets/cancels all prompts. This command causes the prompt to act as though the “CANCEL” button on the prompt has been pressed. If this command is called, and a prompt is currently being displayed (such as **padPromptNumber**), the prompt will immediately be removed from the screen. This is used in cases where the prompt needs to be removed if the user has not acted on the prompt within a given amount of time. For example, suppose that **padPromptNumber** is called and a number prompt is now on the display, 3 minutes pass and the user has not responded to the prompt. You can use **padPromptReset** to stop the current prompt and go onto something else.

#### Returns

Returns TRUE upon success, FALSE otherwise.

#### See Also

padPromptHexNumber, padPromptNum, padPromptNumber, padPromptSignature, padPromptString, padPromptTimeout

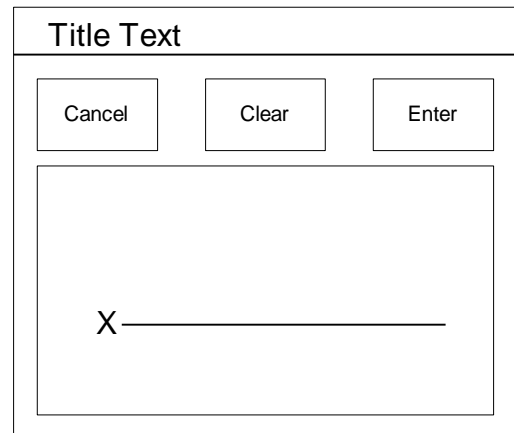
---

**padPromptSignature**

**BOOL padPromptSignature( char \*Title, WORD MaxPoints, char \*SigData,  
WORD SigBufSize, WORD \*SigDataSize )**

Display a Signature capture prompt on the pad. The user can set the compressed capture mode using **padSetCompress** command.

Parameter	Description
Title	Specifies the title of the displayed prompt pad. Its a pointer to a null terminated character string.
MaxPoints	Maximum signature capture points. (Default = 1024.)
SigData	A pointer to the buffer to hold the signature data.
SigBufSize	Specifies the maximum size of the signature data buffer.
SigDataSize representing	Returns a 16-bit value the size of the captured signature data.

**Description**

The command puts up a pre-determined signature prompt on the screen. The illustration depicts the layout of the signature prompt. The data is returned in a special @pos.com signature data packet. This packets needs to be post-processed to extract the signature data from the buffer. Please contact @pos.com for more information on extracting the signature data returned from this command.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padPromptHexNumber, padPromptNum, padPromptNumber, padPromptReset, padPromptString, padPromptTimeout

---

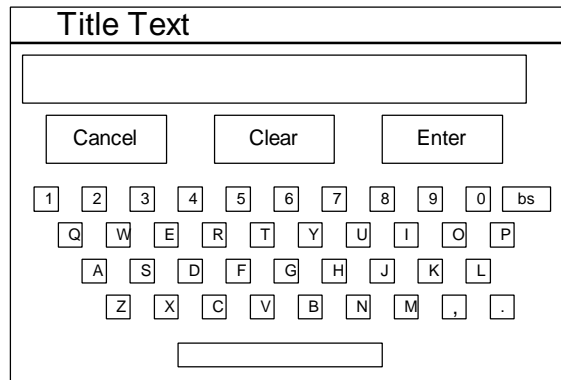
**padPromptString**

**BOOL padPromptString( char \*Title, WORD MaxDigits, char \*Result )**

Displays an alphanumeric keypad and returns the entered alphanumeric data.

Parameter	Description
Title	Specifies the title of the displayed alphanumeric entry pad. Its a pointer to a null terminated character string.
MaxDigits	Maximum alphanumeric characters to be displayed.
Result	Returns a variable length text string containing the alphanumeric data entered.

This command prompts the user to enter alphanumeric data. The prompt includes “ENTER”, “CLEAR”, ”C”, “UNDO”, and “CANCEL” buttons. “ENTER” accepts the alphanumeric entry. “CLEAR” and “C” both clear the alphanumeric entry. “UNDO” removes the last character entered. “CANCEL” exits the prompt. The alphanumeric data can have up to 17 characters. The result is returned as a text string containing the characters entered. The illustration depicts the string input keypad.



### Returns

Returns TRUE upon success, FALSE otherwise.

### See Also

padPromptHexNumber, padPromptNum, padPromptNumber, padPromptReset, padPromptSignature, padPromptTimeout

---

## padPromptTimeout

**void padPromptTimeout ( unsigned long Seconds );**

Set's the time-out value to be used in prompts.

Parameter	Description
Seconds	Specifies the amount of time in seconds to keep the prompt on the display.

### Description

This command is used to set the maximum amount of time a prompt (such as **padPromptSignature**) will remain on the display. The default is 600 seconds (10 minutes). After the specified amount of time elapses the prompt will be removed from the screen.

### See Also

padPromptHexNumber, padPromptNum, padPromptNumber, padPromptReset, padPromptSignature, padPromptString

---

## padPutBits

**BOOL padPutBits( int X, int Y, int Width, int Height, const void \*Bits )**

Draws a bitmap on the LCD display.

Parameter	Description
X	Horizontal coordinate to place the bitmap.
Y	Vertical coordinate to place the bitmap.
Width	Horizontal size of the bitmap in pixels.
Height	Vertical size of the bitmap in pixels.
Bits	Pointer to the bitmap data.

**Description**

Transfers the contents of a bitmap buffer to the LCD display, if available.

NOTE: This command does not display a BMP type bitmap with header data. It only accepts raw bitmap data without a header. When using this command with a PenWare2000 the values for **X** and **Y** must both be zero, the value for **Width** must be 240 and the value for **Height** must be 128. See the Appendix A on page 102 for more information on using raw bitmaps.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padIsLcd, padClear, padPutLogo, padMemLoadBitmap, padPutBmpFile, padPutLogo, padSetLogoBmpFile

---

**padPutBmpFile**

**BOOL padPutBmpFile( int X, int Y, const char \*FileName)**

Draws a Windows Bitmap on the LCD display.

<b>Parameter</b>	<b>Description</b>
X	Horizontal coordinate to place the bitmap.
Y	Vertical coordinate to place the bitmap.
FileName	Name of Windows bitmap file.

**Description**

Transfers the contents of the Windows bitmap file to the LCD display, if available.

NOTE: This command accepts a Windows bitmap (BMP) file. It does not accept raw unformatted bitmap data. When using this command with a PenWare2000 the values for **X** and **Y** must both be zero, the value for **Width** must be 240 and the value for **Height** must be 128.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padIsLcd, padClear, padSetLogoBmpFile, padMemLoadBitmap, padPutBits

---

**padPutLogo**

**BOOL padPutLogo( void )**

Draws the logo on the LCD display.

**Description**

Displays the current logo image stored in the pad's non-volatile memory on the LCD display. The logo image can be set to any bitmap image the user desires (see **padSetLogo** and **padSetLogoBmpFile**).

**See Also**

padIsLcd, padSetLogo, padSetLogoBmpFile

---

**padPutText**

**BOOL padPutText( int X, int Y, const char \*String )**

Draws text on the LCD display.

<b>Parameter</b>	<b>Description</b>
X	Horizontal coordinate to draw text.
Y	Vertical coordinate to draw text.
String	NULL terminated string of text to be displayed.

**Description**

Draws the text pointed to by *String* onto the display if available. The upper left corner of the text will be positioned at the coordinate specified by the giving *X* and *Y* parameters. Text will be drawn using the current font. The parameter *X* specifies the horizontal and the parameter *Y* specifies the vertical positions on the LCD display.

**See Also**

padIsLcd, padClear, padSetFont

---

**padReadByte**

**BOOL padReadByte( BYTE \*b )**

Reads a single byte from the pad.

<b>Parameter</b>	<b>Description</b>
b	a pointer to a memory location that will receive the byte.

**Description**

This command retrieves a single byte from the pad. This command can be used when the pad is in pass through mode to retrieve data from a device connected to the pass through port of the pad (if available). Using **padSetInTimeout** you can control how long this command will wait for the byte to be retrieved.

**Returns**

If successful it returns TRUE, otherwise it returns FALSE.

**See Also**

padSendByte, padPassThroughOn, padSetInTimeout

---

**padRecord**

**DOS:**        **BOOL padRecord( void )**  
**WIN16:**    **BOOL padRecord( HWND hWnd )**  
**WIN32:**    **BOOL padRecord**  
              **( HWND hWnd, UINT aMsg, WPARAM p1, LPARAM p2 )**  
**OS2:**       **BOOL padRecord**  
              **( HWND hWnd, ULONG aMsg, MPARAM p1, MPARAM p2 )**

Begin receiving data from the pad.

<b>Parameter</b>	<b>Description</b>
hWnd	Handle to a window to receive notification messages.
aMsg	A message to use for notification.
p1	An optional message parameter.
p2	An optional message parameter.

**Description**

Sends commands to the pad to start recording pad activities. When running DOS, this function does not perform any notifications and it is up to the application to use **padUpdate** repeatedly to collect incoming data. When using Windows or OS/2, a notification message is sent to the Window specified by *hWnd*. Upon receiving the message, the window should use **padUpdate** to check for and receive new data. When using 16 bit Windows 3.x, the notification message is WM\_COMMNOTIFY. When using either 32 bit Windows 95/NT or OS/2, you may specify any message with optional message parameters.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padStop, padUpdate, padGet

---

**padReset**

**BOOL padReset( void )**

**Description**

Used to reset the pad. This command should be used to reset all default values on the pad. At times, it can take a few seconds for the pad to be completely reset.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padResetArea, padStop, padOff

---

**padResetArea**

**void padResetArea( void )**

Resets the minimum and maximum coordinates of the active pad area.

**Description**

Used to undo **padSetArea**. Restores usable area of the pad to the full pad surface.

**See Also**

padGetArea, padSetArea

---

**padResetBaudRate**

**BOOL padResetBaudRate( void )**

Sets the communications baud rate back to the default.

**Description**

Set's the communications baud rate to the default baud rate of 9600. Note that the command **padOff** resets the baud rate back to the baud rate it initially connected at.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padSetBaudRate, padGetBaudRate, padSetDefaultBaudRate, padGetDefaultBaudRate, padResetDefaultBaudRate

---

**padResetConnectTimeout**

**unsigned int padResetConnectTimeout( void )**

**Description**

This function resets the time-out value used for the COM port (both input and output) during the initially connecting phase. After a connection is established the time-out values can be adjusted using padSetInTimeout and padSetOutTimout.

---

**Returns**

Returns the current connection time-out setting

**See Also**

padSetOutTimeout, padGetInTimeout, padGetOutTimeout, padSetInTimeout, padResetOutTimeout, padSetConnectTimeout, padGetConnectTimeout

---

**padResetDefaultBaudRate**

**BOOL padResetDefaultBaudRate( void )**

**Description**

Set's the default communications baud rate to the initial baud rate of 9600. See padSetDefaultBaudRate for more information.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padSetBaudRate, padGetBaudRate, padSetDefaultBaudRate, padGetDefaultBaudRate, padResetBaudRate

---

**padResetInTimeout**

**unsigned int padResetInTimeout( void )**

**Description**

This function resets the time-out value used for the COM port's input from the connected pad. This is not related to the time-out used during the initial connection phase (see padSetConnectTimeout).

**Returns**

Returns the default timeout setting

**See Also**

padSetOutTimeout, padGetInTimeout, padGetOutTimeout, padSetInTimeout, padResetOutTimeout, padSetConnectTimeout, padGetConnectTimeout, padResetConnectTimeout

---

**padResetMagCard**

**BOOL padResetMagCard( void )**

**Description**

This function resets the magnetic card reader available on some @pos.com hardware such as the iPOS TC in posClassic mode and the PenWare 3100. Commands such as **padGetMagTrack** and **padGetAllMagCardTracks** require that the magnetic card reader be reset before use. Resetting the magnetic card reader clears the card track data from the magnetic card reader's buffer and activates the card reader to allow capture of track data from another card. Only one card may be swiped at a time. Each time a new card is to be swiped a call to **padResetMagCard** must be made before swiping the card. After that you can use **padGetMagTrack** or **padGetAllMagCardTracks** to read the card's captured track data.

**Returns**

Returns TRUE if a card reader is available, FALSE if no card reader is available.

**See Also**

padGetMagTrack, padGetMaxCardTrackSize, padGetMaxCardTracks, padGetAllMagCardTracks

---

**padResetOutTimeout**

**unsigned int padResetOutTimeout( void )**

**Description**

This function resets the time-out value used for the COM port's output to the connected pad. This is not related to the time-out used during the initial connection phase (see padSetConnectTimeout).

**Returns**

Returns the default timeout setting

**See Also**

padSetOutTimeout, padGetInTimeout, padGetOutTimeout, padSetInTimeout, padResetInTimeout, padSetConnectTimeout, padGetConnectTimeout, padResetConnectTimeout

---

**padRightButton**

**BOOL padRightButton( void )**

Checks if the right hardware button is pressed.

**Description**

Uses the most recently received pad data to determine if the right button is pressed. Note that the status is only updated when **padUpdate** returns TRUE to indicate new data received.

**Returns**

Returns TRUE if the right button is pressed, FALSE otherwise.

---

**See Also**padUpdate, padRecord, padLeftButton, padMiddleButton, padHardwareButton

---

**padScale****int padScale( int Value, int Numerator, int Denominator )**

Scales an arbitrary value by a given fraction.

<b>Parameter</b>	<b>Description</b>
Value	The value to be scaled.
Numerator	The numerator of the fraction to scale by.
Denominator	The denominator of the fraction to scale by.

**Description**

This function uses integer math to perform scaling by a given fraction.

**Returns**

Returns the result of the scaling.

**See Also**padScaleX, padScaleY, padScaleDPI, padScaleTo

---

**padScaleDPI****void padScaleDPI( int \*X, int \*Y, int HorzDPI, int VertDPI )**

Scales horizontal and vertical pad coordinates to a desired DPI resolution.

<b>Parameter</b>	<b>Description</b>
X	Pointer to an integer to hold the horizontal coordinate to be scaled.
Y	Pointer to an integer to hold the vertical coordinate to be scaled.
HorzDPI	The horizontal dots-per-inch resolution desired.
VertDPI	The vertical dots-per-inch resolution desired.

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output. Converts the values pointed to by X and Y pointers. NULL pointers may be passed for either X or Y parameters to suppress scaling of the associated coordinate.

**See Also**padScaleTo, padScaleX, padScaleY

---

**padScaleTo**

**void padScaleTo( int \*X, int \*Y, int Width, int Height )**

Scales horizontal and vertical pad coordinates based on a desired frame size.

<b>Parameter</b>	<b>Description</b>
X	Pointer to an integer to hold the horizontal coordinate to be scaled.
Y	Pointer to an integer to hold the vertical coordinate to be scaled.
Width	The horizontal size of the desired frame area.
Height	The vertical size of the desired frame area.

**Description**

This function is designed to scale coordinates to a desired frame size. It does not maintain the proper aspect ratio unless the desired output frame is proportional to the pad surface. Converts the values pointed to by *X* and *Y* pointers. NULL pointers may be passed for either *X* or *Y* parameters to suppress scaling of the associated coordinate.

**See Also**

padScaleDPI, padScaleX, padScaleY

---

**padScaleX****int padScaleX( int X, int HorzDPI )**

Scales a horizontal pad coordinate to a desired DPI resolution.

<b>Parameter</b>	<b>Description</b>
X	An integer specifying the horizontal coordinate to be scaled.
HorzDPI	The horizontal dots-per-inch resolution of the horizontal coordinate.

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output. The parameter **X** **must**

**Returns**

Returns the horizontal position scaled to the desired DPI.

**See Also**

padScaleY, padScaleDPI

---

**padScaleY****int padScaleY( int Y, int VertDPI )**

Scales a vertical pad coordinate to a desired DPI resolution.

<b>Parameter</b>	<b>Description</b>
------------------	--------------------

---

**Y** An integer specifying the vertical coordinate to be scaled.  
**VertDPI** The vertical dots-per-inch resolution of the horizontal coordinate.

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output.

**Returns**

Returns the vertical position scaled to the desired DPI.

**See Also**

padScaleX, padScaleDPI

---

**padSendByte**

**BOOL padSendByte( BYTE b )**

Sends a stream of data to the pad.

<b>Parameter</b>	<b>Description</b>
b	the byte to send to the pad.

**Description**

This command sends a byte to the pad. This command can be used when the pad is in pass through mode to send data to a device connected to the pass through port of the pad (if available). Using padSetOutTimeout you can control how long this command will wait for the byte to be sent.

**Returns**

If successful it returns TRUE, otherwise it returns FALSE.

**See Also**

padReadByte, padPassThroughOn, padSetOutTimeout

---

**padSetArea**

**void padSetArea( int xLeft, int yTop, int xRight, int yBottom )**

Sets the minimum and maximum coordinates of the active pad area.

<b>Parameter</b>	<b>Description</b>
xLeft	Minimum horizontal coordinate.
yTop	Minimum vertical coordinate.
xRight	Maximum horizontal coordinate.
yBottom	Maximum vertical coordinate.

**Description**

This function causes clipping of points received from the @pos.com POS device's touch sensitive surface. All points that are outside of the specified range are ignored. The coordinates given are not display coordinates, they are touch surface coordinates. In general, touch surface coordinates are much higher in resolution than display coordinates. For example the 3100's touch surface contains 4096x4096 touch points while the 3100's display contains only 320x240 pixels, even though they are about the same size in inches. You can use **padGetPage** to obtain the size and resolution of the touch sensitive surface.

NOTE: The **padRecord** function must be called before points will be retrieved from the @pos.com POS device's touch sensitive surface.

**See Also**

padGetArea, padGetPage, padResetArea, padRecord

---

**padSetAutoInking****BOOL padSetAutoInking( BYTE AutoInking )**

Sets auto inking mode as specified by **AutoInking (1 or 0)**.

**Description**

Auto-inking causes the @pos.com POS device's LCD display to automatically display a pixel where the @pos.com POS device's overlaying touch surface area is touched. The @pos.com POS device's overlaying touch surface area must be in record mode (see **padRecord**) for this to occur. If auto-inking is disabled, the LCD display will not automatically display a pixel when the pad's surface is touched even when in record mode. If the value in **AutoInking** is set to zero then auto-inking is disabled, if set to one then auto-inking is enabled.

NOTE: This function interacts with **padSetInkingArea**.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padGetBkColor, padGetColor, padGetColors, padSetColor, padSetInkingArea

---

**padSetBaudRate****DWORD padSetBaudRate( DWORD BaudRate )**

Sets the communications baud rate.

<b>Parameter</b>	<b>Description</b>
BaudRate	the baud rate to attempt a communications link at.

**Description**

This command attempts to set the communications baud rate to the amount specified by **BaudRate**. The valid values for **BaudRate** are: 0, 1200, 2400, 4800, 9600, 19200, 38400. Setting the **BaudRate** to 0 causes the command to automatically try to connect at the highest possible baud rate (NOTE: using 0, may cause problems in DOS applications run under windows). The default baud rate is 9600 for PadCom. This command should be called only after **padOn** or **padConnect** is called and a valid connection is already established, otherwise it will have no effect. Note that the command **padOff** resets the baud rate to the default baud rate of 9600.

**Returns**

Returns the new baud rate if successful, FALSE otherwise.

**See Also**

padConnect, , padGetBaudRate, padResetBaudRate, padOn

---

**padSetBkColor****BOOL padSetBkColor( int Color )**

Sets the current background color.

<b>Parameter</b>	<b>Description</b>
Color	the color to set the current background color to.

**Description**

Sets the current background color. Currently, this value can be either 1 or 0.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padGetBkColor, padGetColor, padGetColors, padSetColor

---

**padSetClearButton****BOOL padSetClearButton( BYTE Button )**

Sets which hardware button(s) clears the LCD.

<b>Parameter</b>	<b>Description</b>
Button	specifies which button(s) clear the LCD.

**Description**

This function is used to allow the LCD to be cleared by pressing one or more of the @pos.com device's hardware buttons. Any combination of hardware buttons 0, 1, and 2 can be used. Using a value of zero disables button clearing of the LCD.

	0	1	2	3	4	5	6	7
Button 0		X		X		X		X
Button 1			X	X			X	X
Button 2					X	X	X	X

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**  
padClear, padHardwareButton

---

**padSetColor****BOOL padSetColor( int Color )**

Sets the current foreground color.

<b>Parameter</b>	<b>Description</b>
Color	the color to set the current foreground color to.

**Description**

Sets the current foreground color. Currently, this value can be 1 or 0.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**  
padGetBkColor, padGetColor, padGetColors, padSetBkColor

---

**padSetCompress****BOOL padSetCompress( BOOL Compress )**

The command can be used to capture compressed data points directly from the pad. This reduces the number of collected data points and results in faster capture speeds. However, the signature obtained is in the compressed form. The **Compress** flag must be set (TRUE) to enable this mode. The default capture mode is uncompressed and remains so till this command is invoked.

<b>Parameter</b>	<b>Description</b>
Compress	TRUE to enable compressed capture mode. FALSE to collect uncompressed data points.

**Returns**

Returns TRUE if successful, FALSE otherwise.

---

**padSetConnectTimeout****unsigned int padSetConnectTimeout( unsigned int newTimeout )****Description**

This function sets the time-out value in milliseconds used for the COM port during the initial connection phase. After establishing a connection use padSetInTimeout and padSetOutTimeout to control the time-out settings of the COM port.

**Returns**

Returns the old timeout setting

**See Also**

padSetInTimeout, padResetInTimeout, padGetOutTimeout, padResetOutTimeout,  
padGetConnectTimeout, padResetConnectTimeout

---

**padSetDebug****BOOL padSetDebug( WORD Options )**

Set the system debug option on or off. If the system debug option is on, the LCD will display an error message when an error occurs.

<b>Parameter</b>	<b>Description</b>
Options	Specifies the debug options: 0000h => no debug 0001h => display error messages

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padFlush

---

**padSetDefaultBaudRate****DWORD padSetDefaultBaudRate( DWORD BaudRate )**

<b>Parameter</b>	<b>Description</b>
BaudRate	specifies the baud rate to use as the default baud rate. Valid values are 9600, 19200, 38400 and 57600

Used to set the default baud rate. Ordinarily PadCom will initially open the COM port at 9600 baud. This is the default baud rate. The PenWare3000/3100 and the iPOS TC can communicate at baud rates higher than 9600. If for example, a PenWare3000 is connected and it is configured to operate at 57600 baud it will take PadCom longer to connect to it then it would if the PenWare3000 was configured to operate at 9600 baud. This is because ordinarily PadCom searches all COM ports for all possible @pos.com devices using all possible baud rates starting with the default of 9600 baud. If a PenWare3000 is connected to COM port 2 and it is configured to communicate at 57600 baud, first PadCom will search port 1 at the default baud rate which normally is 9600 baud for a PenWare100, PenWare1500, PenWare3000, followed by a PenWare2000. If it fails it will search for a PenWare3000 at 57600, 19200, and then at 38400. PadCom will then repeat this course of action for port 2, port 3, port 4, and so on, until an @pos.com device is found. If the default baud rate is set to 57600 then it will be the first baud rate the port is opened at, if a connection fails at 57600 then next it will try 9600, followed by 19200, and finally it will try 38400. For the fastest possible connection use this command in conjunction with **padSetPort** and **padSetType**.

**Returns**

Returns the old default baud rate or zero if it fails.

**See Also**

padConnect, padGetDefaultBaudRate, padGetBaudRate, padSetBaudRate, padResetBaudRate, padResetDefaultBaudRate

---

**padSetFlowControl****BOOL padSetFlowControl( BOOL FlowControl )**

Enables/disables flow control.

<b>Parameter</b>	<b>Description</b>
FlowControl	TRUE enables flow control, FALSE disables flow control

**Description**

Enables/disables FlowControl for a connected @pos.com device. This command only works when a connection is established with the @pos.com device. Once this command has been successfully issued to the @pos.com device you must call **padConnect** (or **padOff** followed by **padOn**) in order for the changes to take place. The default is flow control enabled.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padOn, padConnect, padOff

---

**padSetFont****int padSetFont( int Font )**

Sets the font used to display text.

<b>Parameter</b>	<b>Description</b>
Font	Id of the desired font.

**Description**

Sets the font used for text output to value specified in **Font**.

Available fonts:

<b>Font Id</b>	<b>Horizontal Size</b>	<b>Vertical Size</b>	<b>Available for PenWare2000</b>	<b>Available for PenWare3000/310 0</b>
0	8	8	YES	YES
1	16	16	YES	YES
2	6	8	NO	YES
3	8	12	NO	YES
4	12	16	NO	YES
5	16	24	NO	YES

**Returns**

Returns the previously selected font.

**See Also**

padGetFont, padPutText

---

**padSetInkingArea****BOOL padSetInkingArea( int x, int y, int w, int h )**

If auto inking is disabled for the entire surface of the pad (see **padSetAutoInking**) this function allows you to specify an area of the LCD where auto inking will occur. Pen stroke data is not altered so that you will still receive pen strokes for strokes outside of the inking area. To limit inking and pen stroke data use **padSetArea**. To reset the inking area to the entire surface of the LCD pass 0,0,0,0 to this function as its **x**, **y**, **w**, and **h** parameter values.

The coordinates given are display coordinates, they are not touch surface coordinates. In general, touch surface coordinates are much higher in resolution than display coordinates. For example the 3100's touch surface contains 4096x4096 touch points while the 3100's display contains only 320x240 pixels, even though they are about the same size in inches. Use **padLcdHeight**, **padLcdWidth**, **padLcdHorzDPI**, and **padLcdVertDPI** to find the size and resolution of the LCD display.

NOTE: This function interacts with **padSetAutoInking**. If **padSetAutoInking** is set to true then the entire @pos.com POS device's overlaying touch surface area will cause inking to occur on the entire LCD display area which will cause **padSetInkingArea** to have no effect at all. In order for **padSetInkingArea** to have an effect **padSetAutoInking** must be set to false and **padRecord** must be enabled.

Parameter	Description
x	horizontal starting position of the inking area.
y	vertical starting position of the inking area.
w	width of the inking area.
h	height of the inking area.

**See Also**

padGetArea, padLcdHeight, padLcdWidth, padLcdHorzDPI, padLcdVertDPI, padRecord, padResetArea, padSetArea, padSetAutoInking

---

**padSetInTimeout**

**unsigned int padSetInTimeout( unsigned int newTimeout )**

**Description**

This function sets the time-out value in milliseconds used for the COM port's input from the connected pad. This is not related to the time-out used during the initial connection phase (see padSetConnectTimeout).

**Returns**

Returns the old timeout setting

**See Also**

padSetOutTimeout, padResetInTimeout, padGetOutTimeout, padSetInTimeout, padResetOutTimeout, padSetConnectTimeout, padGetConnectTimeout, padResetConnectTimeout

---

**padSetOutTimeout**

**unsigned int padSetOutTimeout( unsigned int newTimeout )****Description**

This function sets the time-out value in milliseconds used for the COM port's output to the connected pad. This is not related to the time-out used during the initial connection phase (see padSetConnectTimeout).

**Returns**

Returns the old timeout setting

**See Also**

padSetInTimeout, padResetInTimeout, padGetOutTimeout, padResetOutTimeout,  
padSetConnectTimeout, padGetConnectTimeout, padResetConnectTimeout

---

**padSetLcdClearTimeout****BOOL padSetLcdClearTimeout( BYTE Timeout )**

Sets the LCD automatic clearing Timeout value.

<b>Parameter</b>	<b>Description</b>
<b>Timeout</b>	The number of seconds to allow to elapse before timeout is reached. A value of zero disables the automatic LCD clearing feature.

**Description**

If disabled the LCD will never automatically clear itself. If set to 200 for example and the device's touch sensitive surface is not touched for a total of 200 seconds then the device's LCD will automatically clear itself.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padIsLcd, padClear

---

**padSetLogo****BOOL padSetLogo( const void \*Bits )**

Sets the logo bitmap for devices with displays.

<b>Parameter</b>	<b>Description</b>
<b>Bits</b>	Pointer to the bitmap bits.

**Description**

Set the LCD display logo to the contents of the bitmap image **Bits**. The logo is not actually displayed until **padPutLogo** is called. To reset the logo to the default image, use NULL as the **Bits** parameter. The logo is required to have 240 pixels across and 128 pixels down.

NOTE: this command does not accept a Windows bitmap (BMP) with a header. It only accepts raw bitmap data without a header. See the Appendix A as explained on page 102.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padIsLcd, padPutLogo, padSetLogoBmpFile

---

**padSetLogoBmpFile**

**BOOL padSetLogoBmpFile( const char \*FileName )**

Sets the logo image to a Windows bitmap file.

<b>Parameter</b>	<b>Description</b>
FileName	Name of Windows bitmap file.

**Description**

Sets the LCD display logo to the contents of the Windows Bitmap file named **FileName**. The logo is not actually displayed until **padPutLogo** is called. The logo is required to have 240 pixels across and 128 pixels down.

NOTE: this command only accepts a Windows Bitmap file (BMP), it does not accept unformatted raw bitmap data without a header. For more information on acceptable bitmap format, please refer to Appendix A on page number 102.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padIsLcd, padClear, padSetLogo

---

**padSetNumVar**

**BOOL padSetNumVar( char FAR \*lpszVarName, WORD wValue )**

<b>Parameter</b>	<b>Description</b>
------------------	--------------------

---

lpszVarName      Name of the variable to assign the numeric value to.  
wValue            Value

**Description**

Set a variable with given num data

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padGetNumVar

---

**padSetPadMode****BOOL padSetPadMode( BYTE Mode )**

Sets the @pos.com devices operational mode.

<b>Parameter</b>	<b>Description</b>
Mode	specifies which mode to set the @pos.com device to.

**Description**

<b>Mode</b>	<b>Description</b>
0	native mode
1	PenWare100 emulation mode

When in native mode the device is not emulating another device. When in PenWare100 emulation mode the device's touch surface will behave as if it had the DPI resolution of the PenWare100.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padSetPadOffset, padSetClearButton

---

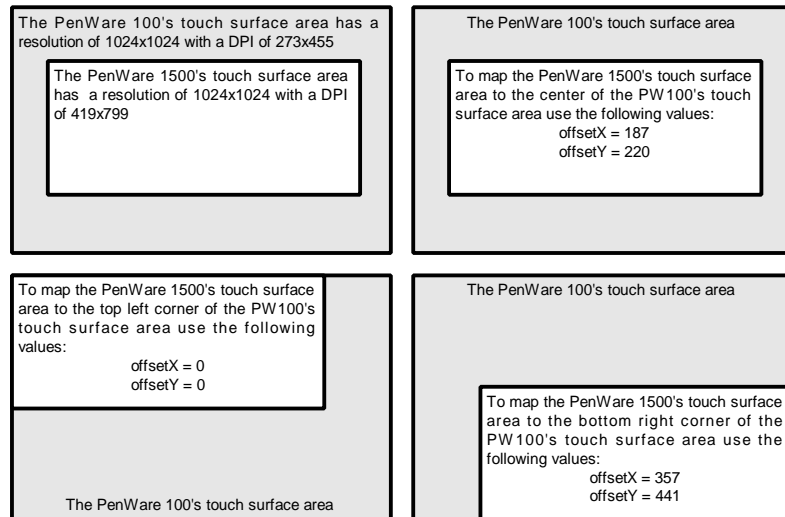
**padSetPadOffset****BOOL padSetPadOffset( int OffsetX, int OffsetY )**

Sets the pad's touch surface point offset when in PenWare100 emulation mode.

<b>Parameter</b>	<b>Description</b>
OffsetX	Horizontal offset measure in the PenWare100's resolution.
OffsetY	Vertical offset measure in the PenWare100's resolution.

## Description

This command is used to set the offset value of the horizontal and vertical points retrieved from the @pos.com device's touch surface area when mapping the device's smaller surface area to the PenWare100's larger surface area when in PenWare100 emulation mode. For example, the PenWare100's touch surface area is approximately 9.5x5.5 centimeters in size and the PenWare1500's touch



surface area is approximately 7x3 centimeters in size. To center the smaller PenWare1500's touch surface area within the PenWare100's touch surface area you would use an offset of 187x220. To place the PenWare1500's touch surface area at the bottom right corner of the PenWare100's touch surface area you would use an offset of 357x441. To place the PenWare1500's touch surface area at the top left corner of the PenWare100's touch surface area you would use an offset of 0x0 meaning no offset is used. This command only works if the @pos.com device is in PenWare100 emulation mode. If the device is not in PenWare100 emulation mode this command has no affect. To put the device into PenWare100 emulation mode use the **padSetPadMode** command.

## Returns

Returns TRUE if successful, FALSE otherwise.

## See Also

padSetPadMode, padSetClearButton

---

## padSetPixel

### BOOL padSetPixel( int X, int Y )

Sets a pixel on the LCD display.

Parameter	Description
X	Horizontal coordinate of the pixel to set.
Y	Vertical coordinate of the pixel to set.

## Description

Sets a pixel on the LCD display at the location specified by **X** and **Y** to the current foreground color.

**Returns**

Returns TRUE if successful, FALSE otherwise.

**See Also**

padBox, padClearPixel, padFrame, padInvert, padLine

---

**padSetPort****BOOL padSetPort( int Port )**

Sets the communications port to use.

**Parameter**

Port

**Description**

Should be 1 for COM1, 2 for COM2, 3 for COM3 or 4 for COM4.

0 causes all the ports to be sequentially checked for a @pos.com device.

**Description**

The next time **padOn** is executed, only the COM port specified by **Port** will be used. This function works only when the library is in an “off” state (i.e. before **padOn()** or after **padOff()**). This is not normally needed however it may be useful in some situations.

**Returns**

Returns TRUE if the port specified is legal and library is “off”, otherwise FALSE.

**See Also**

padGetPort, padIsOn, padSetPortAddr, padSetPortIrq, padSetType

---

**padSetPortAddr****DOS: BOOL padSetPortAddr( int Port, int Address )**

Sets the address of a port.

NOTE: This function is available in the DOS version only!

**Parameter**

Port

Address

**Description**

Should be 1 for COM1, 2 for COM2, 3 for COM3 or 4 for COM4

The value to be used as the port address

**Description**

Sometimes it may be necessary to manually specify the address used when referring to a communications port. **padSetPortAddr** provides this ability. This function works only when the

---

library is in an “off” state (i.e. before **padOn()** or after **padOff()**). This is not normally needed however it may be useful in some situations.

**Returns**

Returns TRUE if the port specified is legal and library is “off”, otherwise FALSE.

**See Also**

padSetPort, padSetPortIrq, padSetType

---

**padSetPortHandle**

**Win32:** void padSetPortHandle( HANDLE newPortHandle )

**Win16:** void padSetPortHandle( int newPortHandle )

Parameter	Description
newPortHandle	Handle to the port

**Description**

Set a handle to the port

**See Also**

padSetPort, padSetPortIrq, padSetType

---

**padSetPortIrq**

**DOS:** int padSetPortIrq( int Port, int Irq )

Sets the interrupt number associated with a port.

NOTE: This function is available in the DOS version only!

Parameter	Description
Port	Should be 1 for COM1, 2 for COM2, 3 for COM3 or 4 for COM4
Irq	The value to be used as the port interrupt number

**Description**

Sometimes it may be necessary to manually specify the interrupt request numbers used when referring to a communications port. **padSetPortIrq** provides this ability. This function works only when the library is in an “off” state (i.e. before **padOn()** or after **padOff()**). This is not normally needed however it may be useful in some situations.

**Returns**

Returns TRUE if the port specified is legal and library is “off”, otherwise FALSE.

---

**See Also**padSetPort, padSetPortAddr, padSetType

---

**padSetPorts****int padSetPorts( int MaxPorts )**

Sets the maximum number of communications ports available.

<b>Parameter</b>	<b>Description</b>
MaxPorts	valid values start at 1 for 1 COM port. Note: The maximum allowed for Windows 3.1 is 9.

**Description**

The next time **padOn** or **padConnect** is executed, only the COM ports specified by **MaxPorts** will be available for use. This function works only when the library is in an "off" state (i.e. before **padOn/padConnect** or after **padOff**).

**Returns**

Returns the previous number of ports used.

**See Also**padGetPort, padGetPorts, padIsOn, padSetPortAddr, padSetPortIrq, padSetType

---

**padSetScanRate****BOOL padSetScanRate( int ScanRate )**

Sets the current scan rate as specified by ScanRate. Values from 26 to 199 can be set. Scan rate changes the number of signature points scanned per second.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**padGetScanRate

---

**padSetTime****BOOL padSetTime( BYTE cHour, BYTE cMin, BYTE cSec )**

<b>Parameter</b>	<b>Description</b>
cHour	sets the hour
cMin	sets the minute
cSec	sets the second

**Description**

This function sets the current time on the PenWare3000 or compatible POS terminal.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padDisplayTime, padGetTime, padHideTime

---

**padSetType****BOOL padSetType ( int Type )**

Sets the pad type to search the COM ports for.

Parameter	Description
Type	used to specify the type of @pos.com pad.

**Description**

Sets the type of pad to search the COM ports for when using **padOn**. Normally **padOn** searches the COM ports for all supported @pos.com pad types. Setting **padSetType** to a valid @pos.com pad type causes **padOn** to only search the COM-ports for that specific type of pad. The valid arguments are:

For a PenWare100:	MP100, PW100, PENWARE100
For a PenWare1500/1100:	PENWARE1500
For a PenWare2000:	IFC2000, PW2000, PENWARE2000
For a PenWare3000/3100:	PENWARE3000, PW3000

For example “padSetType ( PW2000 )” causes **padOn** to only search for a PenWare2000.

Note: the 3000 and 3100 are both treated as the same pad types when using this function in PadCom and the 1100 and 1500 are also both treated as the same pad types when using this function in PadCom. Also, you must call **padSetType** before calling **padOn** or it will have no effect.

**Returns**

Returns TRUE if the type of pad specified is valid and FALSE otherwise.

**See Also**

padOn, padSetPort, padSetPortAddr, padSetPortIrq

---

**padSoundBell****BOOL padSoundBell( WORD BellType )**

Plays a sound or cancels a sound being played on the internal speaker (if available).

---

<b>Parameter</b>	<b>Description</b>
BellType	Type of sound action to make

**Description**

This command allows you to play various different types of preset sounds stored in the PenWare3000/3100 and compatible devices. The parameter **BellType** has the following valid decimal setting (hexadecimal values are in parenthesis):

- 0 (0x00) - cancels any sound being played
- 1 (0x01) - plays the standard "bell" sound
- 16 (0x10) - plays an "alarm" sound
- 20 (0x14) - plays a "success" sound
- 24 (0x18) - plays a "fail" sound

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padSoundTone, padSoundSetFreq, padSoundEnable

---

**padSoundEnable****BOOL padSoundEnable( BYTE Enable )**

Enables or disables sound from the internal speaker (if available).

<b>Parameter</b>	<b>Description</b>
Enable	if zero sound is disabled, if non-zero sound is enabled

**Description**

This command enables or disables the speaker on a PenWare3000 or compatible device.

**Returns**

Returns TRUE upon success, FALSE otherwise.

**See Also**

padSoundTone, padSoundBell, padSoundSetFreq

---

**padSoundSetFreq****BOOL padSoundSetFreq( WORD Freq )**

Plays a constant sound frequency from the internal speaker (if available).

<b>Parameter</b>	<b>Description</b>
------------------	--------------------

---

**Freq**                      the frequency value to set the speaker to

### Description

This command allows the user to set the internal speaker to play a frequency specified by the value in **Freq**. Setting the frequency to zero turns off the sound.

### Returns

Returns TRUE upon success, FALSE otherwise.

### See Also

padSoundTone, padSoundBell, padSoundEnable

---

## padSoundTone

**BOOL padSoundTone( WORD Freq, WORD Duration )**

Plays a constant sound frequency from the internal speaker (if available) for a specified amount of time.

### Parameter

Freq

Duration

### Description

the frequency value to set the speaker to

the amount of time to play the sound frequency

### Description

This command allows the user to set the internal speaker to play a frequency specified by the value in **Freq** for the amount of “tempo beats” specified by **Duration**. 156 “temp beats” are equivalent to approximately 1000 milliseconds (1 second).

The following table shows the frequency values used to play musical notes:

	<b>Octave 1</b>	<b>Octave 2</b>	<b>Octave 3</b>	<b>Octave 4</b>
<b>C</b>	131	262	523	1047
<b>C#</b>	139	278	555	1111
<b>D</b>	147	294	587	1175
<b>D#</b>	156	312	623	1247
<b>E</b>	165	330	659	1319
<b>F</b>	175	349	698	1397
<b>F#</b>	186	371	741	1483
<b>G</b>	196	392	784	1568
<b>G#</b>	208	416	832	1664
<b>A</b>	220	440	880	1760
<b>A#</b>	234	467	934	1868
<b>B</b>	247	494	988	1976

### Returns

Returns TRUE upon success, FALSE otherwise.

**See Also**

padSoundBell, padSoundSetFreq, padSoundEnable

---

**padStop**

**void padStop( void )**

Stop receiving data from the pad.

**Description**

Turns off real-time recording (initiated with **padRecord** command) and sends commands to the pad to stop transmitting pad activities.

**See Also**

padOff, padRecord

---

**padToHIENGLISH**

**void padToHIENGLISH( int \*X, int \*Y )**

Scales horizontal and vertical pad coordinates to units representing 0.001 inches.

<b>Parameter</b>	<b>Description</b>
X	Pointer to an integer to hold the horizontal coordinate to be scaled.
Y	Pointer to an integer to hold the vertical coordinate to be scaled.

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output. Converts the values pointed to by *X* and *Y* pointers. Corresponds to the Windows mapping mode MM\_HIENGLISH, and OS/2 PU\_HIENGLISH units. NULL pointers may be passed for either *X* or *Y* parameters to suppress scaling of the associated coordinate.

**See Also**

padToLOENGLISH, padToLOMETRIC, padToHIMETRIC

---

**padToLOENGLISH**

**void padToLOENGLISH( int \*X, int \*Y )**

Scales horizontal and vertical pad coordinates to units representing 0.01 inches.

<b>Parameter</b>	<b>Description</b>
X	Pointer to an integer to hold the horizontal coordinate to be scaled.
Y	Pointer to an integer to hold the vertical coordinate to be scaled.

---

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output. Converts the values pointed to by *X* and *Y* pointers. Corresponds to the Windows mapping mode MM\_LOENGLISH, and OS/2 PU\_LOENGLISH units. NULL pointers may be passed for either *X* or *Y* parameters to suppress scaling of the associated coordinate.

**See Also**

padToHIENGLISH, padToLOMETRIC, padToHIMETRIC

---

**padToHIMETRIC**

**void padToHIMETRIC( int \*X, int \*Y )**

Scales horizontal and vertical pad coordinates to units representing 0.01 millimeters.

<b>Parameter</b>	<b>Description</b>
X	Pointer to an integer to hold the horizontal coordinate to be scaled.
Y	Pointer to an integer to hold the vertical coordinate to be scaled.

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output. Converts the values pointed to by *X* and *Y* pointers. Corresponds to the Windows mapping mode MM\_HIMETRIC, and OS/2 PU\_HIMETRIC units. NULL pointers may be passed for either *X* or *Y* parameters to suppress scaling of the associated coordinate.

**See Also**

padToLOMETRIC, padToLOENGLISH, padToHIENGLISH

---

**padToLOMETRIC**

**void padToLOMETRIC( int \*X, int \*Y )**

Scales horizontal and vertical pad coordinates to units representing 0.1 millimeters.

<b>Parameter</b>	<b>Description</b>
X	Pointer to an integer to hold the horizontal coordinate to be scaled.
Y	Pointer to an integer to hold the vertical coordinate to be scaled.

**Description**

This function is designed to provide proper aspect ratio scaling for accurate output. Converts the values pointed to by *X* and *Y* pointers. Corresponds to the Windows mapping mode MM\_LOMETRIC, and OS/2 PU\_HIMETRIC units. NULL pointers may be passed for either *X* or *Y* parameters to suppress scaling of the associated coordinate.

**See Also**

padToHIMETRIC, padToLOENGLISH, padToHIENGLISH

---

## **padType**

**enum padTypes padType( void )**

Returns the type of pad attached.

### **Description**

Returns a value that identifies the type of pad attached. The numeric value returned by **padType** may be one of the following constant values defined in the file "PADCOM.H":

UNKNOWN	No recognizable device has been found.
PENWARE100	A PenWare100 pad has been found.
PENWARE1500	A PenWare1500/1100 pad has been found.
PENWARE2000	A PenWare2000 pad has been found.
PENWARE3000	A PenWare3000/3100 pad has been found.

Note: to distinguish between the 3000 and 3100 use padGetModel. The 1100 and 1500 are both treated as 1500's in PadCom.

This command differs from padName in that it returns a numeric value rather than a string.

### **See Also**

padName

---

## **padUpdate**

**BOOL padUpdate( void )**

Checks for and receives new data from the pad.

### **Description**

Receives data packets from the pad and decodes them into meaningful information. All information received is stored until the next time this function is executed. Use functions such as padGet, padIsPenDown, and padHardwareButton to access the information received. This function is generally used in response to a notification message as discussed in the padRecord reference.

### **Returns**

Returns TRUE if new data was received, otherwise FALSE.

### **See Also**

padRecord, padGet, padHardwareButton, padLeftButton, padRightButton, padMiddleButton

---

## **padVertDPI**

**int padVertDPI( void )**

**Returns**

Returns the number of vertical points per inch.

**See Also**

padHeight, padHorzDPI

---

**padWidth**

**int padWidth( void )**

**Returns**

Returns the total number of horizontal points on the pad surface (i.e. 1024).

**See Also**

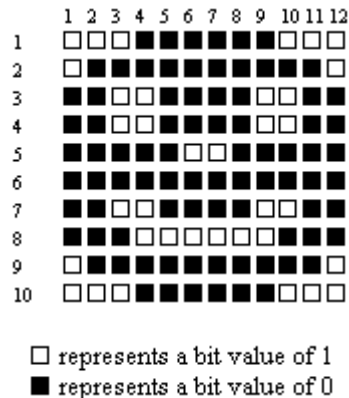
padHeight, padHorzDPI

## Appendix A

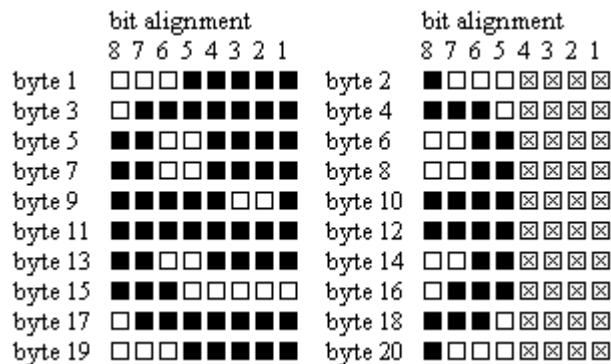
### Supported Bitmap Format

Many PadCom commands such as **padPutBmpFile** and **padSetLogoBmpFile** accept Windows style bitmap files. Other command such as **padPutBits** and **padMemLoadBitmap** do not. These commands accept raw black and white bitmap data. The raw black and white bitmap format used throughout PadCom is the standard raw bitmap format used for black and white bit mapped images. The bitmap data is byte aligned. The first byte represents the top left most portion of the image. The last byte represents the bottom right most portion of the image. The black and white format uses one bit per pixel. Bit 1 is the rightmost pixel of each byte and bit 8 is the leftmost pixel of each byte. A bit value of zero is black and a bit value of one is white.

The following illustrates a bit mapped image as it would appear on a black and white LCD display. The image has a horizontal size of 12 pixels and a vertical size of 10 pixels.



The following illustrates the raw bit map data of the above image. Notice that the image data is two bytes wide and 10 bytes tall even though the size of the actual image is 12 pixels (12 bits) wide and 10 pixels tall. Notice also that the right most pixels are not used. These unused bits are required for the data to be byte aligned.



- represents a bit value of 1
- represents a bit value of 0
- represents an unused bit

NOTE: unlike the PenWare3000, the PenWare2000 only allows bitmaps with a resolution of exactly 240x128 and they must be placed at location 0x0.

## Appendix B

### PadCom Sample Source Code

The sample source code is supplied for Microsoft compilers only. A copy of the sample code can be found on the installation diskettes along with the necessary make files. Please bear in mind that these samples are intended for demo purposes only and do not necessarily perform useful tasks. However, they provide a basis on which you can build complex and meaningful Point Of Sale applications.

#### *PadCom Signature Capture Sample for DOS:*

```
//
//-----
// Test.c
//
// A simple DOS Demo program using DOS Graphics Functions.
//
// Compiler:
//   Microsoft Visual C++ 1.52
//   Target      : DOS Application
//   Memory Model : Medium
//   Include     : ..\..\..\padcom.h
//   Library     : ..\libs\padcomdm.lib
// Environment:
//   DOS 3.3 or better
//
// Copyright (c) 1994-1999, @pos.com. All rights reserved.
//
//-----
//

#include <graph.h>
#include <conio.h>

#include "PadCom.h"

int GraphOn( void );
void GraphOff( void );

void main()
{
    int x, y, p;

    //=====
    // TURN PAD ON-INITIALIZE
    //=====
    if( !padOn() )
    {
        cputs( "Can't find writing pad!" );
        return;
    }

    //=====
    // SWITCH TO GRAPHICS MODE
    //=====
    if( !GraphOn() )
    {
        cputs( "Unable to use graphics" );
        padOff();
        return;
    }

    //=====
    // START REAL TIME RECORDING
```

```
//=====
padRecord();

//=====
// COLLECT DATA
//=====
while( !kbhit( ) )
{
    // CHECK IF THERE IS MORE DATA FROM THE PAD
    if( padUpdate() && padGet( &x, &y, &p ) )
    {
        // SCALE THE POINTS TO ABOUT DOUBLE In SIZE
        padScaleDPI( &x, &y, 160, 160 );

        // DRAW THEM ON THE SCREEN
        if( p )
            _lineto( x, y );
        else
            _moveto( x, y );
    }
}

//=====
// RESTORE VIDEO MODE
//=====
GraphOff();

//=====
// TURN PAD OFF-CLEANUP
//=====
padOff();
}

int GraphOn( void )
{
    char Msg[] = "PLEASE SIGN ON THE PAD, PRESS ANY KEY WHEN DONE";

    if( !_setvideomode( _MAXRESMODE ) )
        return 0;
    _setviewport( 0,0,639,479 );
    _settextposition( 25,15 );
    _outtext( Msg );

    return 1;
}

void GraphOff( void )
{
    _setvideomode( _DEFAULTMODE );
}

```

### ***PadCom Signature Capture Sample for Windows 3.x:***

```
//
//-----
// Test.c
//
// A simple WIN16 Demo program
//
// Compiler:
//   Microsoft Visual C++ 1.52
//   Target      :   Windows Application
//   Memory Model :   Medium
//   Include     :   ..\..\..\padcom.h
//   Library     :   ..\libs\padcomwm.lib
// Environment:
//   Windows 3.x
//
// Copyright (c) 1994-1999, @pos.com. All rights reserved.

```

```
//
//-----
//

#include "windows.h"

#include "PadCom.h"

int PASCAL WinMain( HINSTANCE, HINSTANCE, LPSTR, int );
LONG __export CALLBACK WndProc( HWND, UINT, WPARAM, LPARAM );

HINSTANCE theInstance;
HWND theWnd;

// -----
// WinMain
// -----
int PASCAL WinMain(
    HINSTANCE hInst, HINSTANCE hInstPrev, LPSTR lpstrCmdLine, int cmdShow
)
{
    MSG msg;
    WNDCLASS wc;

    theInstance = hInst;

    // Register the window class if this is the first instance.
    if( !hInstPrev )
    {
        wc.lpszMenuName = NULL;
        wc.lpszClassName = "TESTAPP";
        wc.hInstance = hInst;
        wc.hIcon = NULL;
        wc.hCursor = NULL;
        wc.hbrBackground = (HBRUSH)COLOR_WINDOW + 1;
        wc.style = 0;
        wc.lpfnWndProc = WndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;

        if( !RegisterClass( &wc ) )
            return 0;
    }

    // Create the main window
    if( !(theWnd = CreateWindowEx(
        WS_EX_TOPMOST, "TESTAPP", "SIGN YOUR NAME THEN PRESS ANY BUTTON",
        WS_OVERLAPPED | WS_SYSMENU, CW_USEDEFAULT, CW_USEDEFAULT, 375, 225,
        NULL, NULL, hInst, NULL ) )
    )
        return 0;

    // Show main window
    ShowWindow( theWnd, cmdShow );
    UpdateWindow( theWnd );

    // Main message loop
    while( GetMessage( LPMSG&msg, NULL, 0, 0 ) )
    {
        TranslateMessage( LPMSG&msg );
        DispatchMessage( LPMSG&msg );
    }

    return 0;
}

// -----
// WndProc
// -----
LONG __export CALLBACK WndProc(
```

```
HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam
)
{
    switch( Msg )
    {
        case WM_CREATE:
        {
            // Turn on the writing pad
            if( padOn() )
            {
                // Start recording...
                padRecord( hWnd );
            }
            else
            {
                // Error
                MessageBox( hWnd, "Unable to find writing pad", "TESTAPP", MB_OK );
                PostQuitMessage( 1 );
            }
        }
        break;

        case WM_COMMNOTIFY:
        {
            // Check for any new data received
            while( padUpdate() )
            {
                int x, y, p;

                // If pen is down, get pen position info and draw it
                if( padGet( &x, &y, &p ) )
                {
                    RECT Rect;
                    HDC hDC;

                    // Setup DC to perform scaling based on client rect
                    hDC = GetDC( hWnd );
                    GetClientRect( hWnd, &Rect );
                    SetMapMode( hDC, MM_ANISOTROPIC );
                    SetWindowExt( hDC, padWidth(), padHeight() );
                    SetViewportExt( hDC, Rect.right, Rect.bottom );

                    // If not first point of new stroke, draw stroke
                    if( p )
                    {
                        MoveTo( hDC, padOldX(), padOldY() );
                        LineTo( hDC, x, y );
                    }

                    ReleaseDC( hWnd, hDC );
                }
            }
        }
        return 1;

        case WM_DESTROY:
        {
            // Always turn things off!
            padStop();
            padOff();

            PostQuitMessage( 0 );

            break;
        }

        case WM_CHAR:
```

```
    {
        DestroyWindow( hWnd );
        break;
    }

    default:
    {
        return DefWindowProc( hWnd, Msg, wParam, lParam );
    }
}

return 0;
}
```

### ***PadCom Signature Capture Sample for Windows95/NT***

```
//
//-----
// Test.c
//
// A simple WIN32 Demo program
//
// Compiler:
//   Microsoft Visual C++ 4.0
//   Target      : Windows Application
//   Include     : ..\..\..\padcom.h
//   Library     : ..\libs\padcomw.lib
// Environment:
//   Windows 95/NT
//
// Copyright (c) 1994-1999, @pos.com. All rights reserved.
//
//-----
//
#include <windows.h>
#include "PadCom.h"

// USER DEFINED PADCOM MESSAGE
#define WM_PADNOTIFY WM_USER + 100

int CALLBACK WinMain( HINSTANCE, HINSTANCE, LPSTR, int );
LONG CALLBACK WndProc( HWND, UINT, WPARAM, LPARAM );

HINSTANCE theInstance;
HWND theWnd;

// -----
// WinMain
// -----
int CALLBACK WinMain(
    HINSTANCE hInst, HINSTANCE hInstPrev, LPSTR lpstrCmdLine, int cmdShow
)
{
    MSG msg;
    WNDCLASS wc;

    theInstance = hInst;

    // Register the window class if this is the first instance.
    if( !hInstPrev )
    {
        wc.lpszMenuName = NULL;
        wc.lpszClassName = "TEST";
        wc.hInstance = hInst;
        wc.hIcon = NULL;
        wc.hCursor = NULL;
        wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
        wc.style = 0;
        wc.lpfnWndProc = WndProc;
    }
}
```

```
        wc.cbClsExtra      = 0;
        wc.cbWndExtra     = 0;

        if( !RegisterClass( &wc ) )
            return 0;
    }

    // Create the main window
    if( !(theWnd = CreateWindowEx(
        WS_EX_TOPMOST, "TEST", "Sign on the pad and hit any key when done",
        WS_OVERLAPPED | WS_SYSMENU, CW_USEDEFAULT, CW_USEDEFAULT, 375, 225,
        NULL, NULL, hInst, NULL ) )
    )
        return 0;

    // Show main window
    ShowWindow( theWnd, cmdShow );
    UpdateWindow( theWnd );

    // Main message loop
    while( GetMessage( (LPMSG)&msg, NULL, 0, 0 ) )
    {
        TranslateMessage( (LPMSG)&msg );
        DispatchMessage( (LPMSG)&msg );
    }

    return 0;
}

// -----
// WndProc
// -----
LONG CALLBACK WndProc(
    HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam
)
{
    switch( Msg )
    {
        case WM_CREATE:
        {
            // Turn on the writing pad
            if( padOn() )
            {
                // Start recording...
                padRecord( hWnd, WM_PADNOTIFY, 0,0 );
            }
            else
            {
                // Error
                MessageBox( hWnd, "Unable to find writing pad", "TESTAPP", MB_OK );
                PostQuitMessage( 1 );
            }
        }
        break;

        case WM_PADNOTIFY:
        {
            // Check for any new data received
            while( padUpdate() )
            {
                int x, y, p;

                // If pen is down, get pen position info and draw it
                if( padGet( &x, &y, &p ) )
                {
                    RECT Rect;
                    HDC hDC;
                }
            }
        }
    }
}
```

```
        // Setup DC to perform scaling based on client rect
        hDC = GetDC( hWnd );
        GetClientRect( hWnd, &Rect );
        SetMapMode( hDC, MM_ANISOTROPIC );
        SetWindowExtEx( hDC, padWidth(), padHeight(), NULL );
        SetViewportExtEx( hDC, Rect.right, Rect.bottom, NULL );

        // If not first point of new stroke, draw stroke
        if( p )
        {
            MoveToEx( hDC, padOldX(), padOldY(), NULL );
            LineTo( hDC, x, y );
        }

        ReleaseDC( hWnd, hDC );
    }

}

return 1;
}

case WM_DESTROY:
{
    // Always turn things off!
    padStop();
    padOff();

    PostQuitMessage( 0 );

    break;
}

case WM_CHAR:
{
    DestroyWindow( hWnd );
    break;
}

default:
{
    return DefWindowProc( hWnd, Msg, wParam, lParam );
}
}
return 0;
}
```

## Appendix C

### PadCom Sample Source Code for the Magnetic Card Reader (MSR)

The sample code for reading an MSR attached to a @pos.com device is provided for various platforms below. The source code is only supplied for demo purposes and is NOT a part of the installation diskettes. There are numerous ways to capture MSR from @pos.com devices and the demo code simply shows one of these. @pos.com does not necessarily claim that this is the best way to read MSR data.

#### *PadCom MSR Sample Code for DOS:*

```
//
// *****
// *****
// **
// ** MSR.C
// **
// ** This is a sample program showing how to use the Magnetic Stripe Reader
// ** hardware available on some @pos.com device's such as the PenWare 3000
// **
// ** Platform: DOS
// **
// ** Compiler: Microsoft Visual C++ 1.52
// **
// **
// ** (C) Copyright 1999 @pos.com.
// **
// *****
// *****
//
//
// *****
// *
// * Required header files
// *
// *****
//
#include <stdio.h>
#include <conio.h>
#include "PadCom.H"

//
// *****
// *
// * Constant values
// *
// *****
//
//
// @pos.com's magnetic stripe reader follows the MEGTEK standard.
// This standard has the following track sizes defined.
// We add 1 to include the appended NULL character at the end of each track.
//
#define TRACK1_MAX (74+1)
#define TRACK2_MAX (39+1)
#define TRACK3_MAX (106+1)

//
// *****
// *
```

```
// * main - the application entry point *
// *
// *****
//
void main ( void )
{
    char
        track1 [TRACK1_MAX],
        track2 [TRACK2_MAX],
        track3 [TRACK3_MAX];
    int
        dataReadFromTrack1 = 0,
        dataReadFromTrack2 = 0,
        dataReadFromTrack3 = 0;

    printf ( "\nExecuting @pos.com Sample program MSR.EXE\n\n" );

    //
    // *****
    // * Step 1: Turn on the @pos.com device using padConnect.*
    // *****
    //
    if ( ! padConnect() )
    {
        printf ( "ERROR: No @pos.com device found\n\n" );
        printf ( "\n\nPress any key\n" );
        while ( !kbhit () );
        return;
    }

    printf ( "Please swipe your Magnetic Card\n\n" );

    printf ( "(Press ESC to cancel)\n\n" );

    //
    // *****
    // * Step 2: prepare the MSR to read data. *
    // *****
    //
    if ( !padGetMagTrack ( 0,0,0 ) )
    {
        printf ( "ERROR: No Magnetic Stripe Reader found\n\n" );
        printf ( "\n\nPress any key\n" );
        while ( !kbhit () );
        padOff ();
        return;
    }

    //
    // Initialize the contents of the buffers.
    //
    strcpy ( track1, "NO DATA READ" );
    strcpy ( track2, "NO DATA READ" );
    strcpy ( track3, "NO DATA READ" );

    while ( 1 )
    {
        //
        // If the ESC key is pressed then return.
        //
        if ( kbhit () )
        {
            if ( getch () == 27 )
            {
                padOff ();
                return;
            }
        }
    }

    //

```

```

// *****
// * Step 3: Check if any of the tracks were read. *
// *****
//
dataReadFromTrack1 = padGetMagTrack ( 1, track1, TRACK1_MAX );
dataReadFromTrack2 = padGetMagTrack ( 2, track2, TRACK2_MAX );
dataReadFromTrack3 = padGetMagTrack ( 3, track3, TRACK3_MAX );

if
(
    dataReadFromTrack1 ||
    dataReadFromTrack2 ||
    dataReadFromTrack3
)
{
    //
    // At this point we know that at least 1 track was read
    // successfully (the last track read, i.e., track 3).
    //
    // It is possible one or more of the other tracks were
    // checked before any data was retrieved from the card.
    //
    // As soon as one track contains data all of the other
    // tracks will as well.
    //
    // To make sure we get all of the data from all of the tracks
    // we will read all of them again.
    //
    //
    // *****
    // * Step 4: Read all of the tracks *
    // *****
    // Note: step 4 is not needed if only one track is being read
    //
    dataReadFromTrack1 = padGetMagTrack ( 1, track1, TRACK1_MAX );
    dataReadFromTrack2 = padGetMagTrack ( 2, track2, TRACK2_MAX );
    dataReadFromTrack3 = padGetMagTrack ( 3, track3, TRACK3_MAX );

    //
    // print out the results
    //
    printf ( "Data received from Magnetic Card:\n\n");
    printf ( "Track 1 size: %d\n", dataReadFromTrack1 );
    printf ( "Track 1 data: %s\n\n", track1 );
    printf ( "Track 2 size: %d\n", dataReadFromTrack2 );
    printf ( "Track 2 data: %s\n\n", track2 );
    printf ( "Track 3 size: %d\n", dataReadFromTrack3 );
    printf ( "Track 3 data: %s\n\n", track3 );
    printf ( "\n\nPress any key\n" );
    while ( !kbhit ( ) );

    //
    // Turn of the @pos.com device and quit the test application
    //
    padOff ( );
    return;
}
}
}

```

### ***PadCom MSR Sample Code for Win 3.x***

```

//
// *****
// *****
// **
// ** MSR.C **
// **
// ** This is a sample program showing how to use the Magnetic Stripe Reader **

```

```
// ** hardware available on some @pos.com device's such as the PenWare 3000      **
// **                                                                            **
// ** Platform: Windows 3.1                                                    **
// **                                                                            **
// ** Compiler: Microsoft Visual C++ 1.52                                     **
// **                                                                            **
// **                                                                            **
// ** (C) Copyright 1999 @pos.com.                                             **
// **                                                                            **
// *****
// *****
//
//
// *****
// *                                                                           *
// * Required header files                                                   *
// *                                                                           *
// *****
//
#include <windows.h>

#include <string.h>
#include <stdio.h>
#include "PadCom.H"

//
// *****
// *                                                                           *
// * Constant values                                                         *
// *                                                                           *
// *****
//
//
// @pos.com's magnetic stripe reader follows the MEGTEK standard.
// This standard has the following track sizes defined.
// We add 1 to include the appended NULL character at the end of each track.
//
#define TRACK1_MAX (74+1)
#define TRACK2_MAX (39+1)
#define TRACK3_MAX (106+1)

#define MSR_TIMER_ID 1
#define MSR_DATA_READ 1

//
// *****
// *                                                                           *
// * Function prototypes                                                     *
// *                                                                           *
// *****
//
int PASCAL          WinMain      ( HINSTANCE,HINSTANCE, LPSTR, int );
LONG __export CALLBACK WndProc   ( HWND, UINT, WPARAM, LPARAM );
void CALLBACK      TimerProc    ( HWND, UINT, UINT, DWORD );
int                ReadMSR      ( void );
int                ResetMSR     ( HWND );

//
// *****
// *                                                                           *
// * Global variables                                                       *
// *                                                                           *
// *****
//
HINSTANCE theInstance;
```

```

HWND         theWnd;
TIMERPROC    lpfnMyTimerProc;

char *appName = "MSR";
char *appTitle = "MSR.EXE - Please swipe your Magnetic Card";

char
    track1 [TRACK1_MAX],
    track2 [TRACK2_MAX],
    track3 [TRACK3_MAX];
int
    dataReadFromTrack1 = 0,
    dataReadFromTrack2 = 0,
    dataReadFromTrack3 = 0;

//
// *****
// *
// * WinMain - the application entry point
// *
// *****
//
int PASCAL WinMain
(
    HINSTANCE  hInst,
    HINSTANCE  hInstPrev,
    LPSTR      lpstrCmdLine,
    int        cmdShow
)
{
    MSG        msg;
    WNDCLASS   wc;

    theInstance = hInst;

    //
    // Register the window class if this is the first instance.
    //
    if( !hInstPrev )
    {
        wc.lpszMenuName = NULL;
        wc.lpszClassName = appName;
        wc.hInstance = hInst;
        wc.hIcon = NULL;
        wc.hCursor = NULL;
        wc.hbrBackground = (HBRUSH)COLOR_WINDOW + 1;
        wc.style = 0;
        wc.lpfnWndProc = WndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;

        if( !RegisterClass( &wc ) )
            return 0;
    }

    //
    // Attempt to create the main window
    //
    theWnd = CreateWindowEx
    (
        WS_EX_TOPMOST, appName, appTitle, WS_OVERLAPPED | WS_SYSMENU,
        CW_USEDEFAULT, CW_USEDEFAULT, 375, 225, NULL, NULL, hInst, NULL
    );

    //
    // If the window was not created then quit
    //
    if ( !theWnd )
    {
        return 0;
    }
}

```

```
    }

    //
    // Show the main window
    //
    ShowWindow( theWnd, cmdShow );
    UpdateWindow( theWnd );

    //
    // Process the main message loop
    //
    while( GetMessage( (LPMSG) &msg, NULL, 0, 0 ) )
    {
        TranslateMessage ( (LPMSG) &msg );
        DispatchMessage ( (LPMSG) &msg );
    }

    return 0;
}

//
// *****
// *
// * WndProc - Message handler for the application
// *
// *****
//
LONG __export CALLBACK WndProc(
    HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam
)
{
    char msrData [1024];

    switch( Msg )
    {
        case WM_CREATE:
        {
            //
            // *****
            // * Step 1: Turn on the @pos.com device using padConnect.*
            // *****
            //
            if( padConnect() )
            {
                if ( !ResetMSR ( hWnd ) )
                {
                    PostQuitMessage( 1 );
                }
            }
            else
            {
                //
                // Error
                //
                MessageBox( hWnd, "ERROR: No @pos.com device found!", appName, MB_OK );
                PostQuitMessage( 1 );
            }
        }

        break;

        case WM_COMMAND:
        {
            switch ( wParam )
            {
                case MSR_DATA_READ:
                {
                    //

```

```

// Display the data read from the Magnetic Stripe Reader
//
sprintf
(
    msrData,
    "Track 1 size: %d\n"
    "Track 1 data: %s\n\n"
    "Track 2 size: %d\n"
    "Track 2 data: %s\n\n"
    "Track 3 size: %d\n"
    "Track 3 data: %s\n\n",

    dataReadFromTrack1,
    track1,
    dataReadFromTrack2,
    track2,
    dataReadFromTrack3,
    track3
);

MessageBox( hWnd, msrData, "MSR.EXE - Data received from the Magnetic
Card", MB_OK );

//
// Reset
//
if ( !ResetMSR ( hWnd ) )
{
    PostQuitMessage( 1 );
}
}
break;
}

case WM_DESTROY:
{
    //
    // Turn of the @pos.com device
    //
    padOff();

    PostQuitMessage( 0 );

    break;
}

case WM_CHAR:
{
    DestroyWindow( hWnd );
    break;
}

default:
{
    return DefWindowProc( hWnd, Msg, wParam, lParam );
}
}

return 0;
}

//
// *****
// *
// * TimerProc - Message handler for timer used to read the MSR
// *
// *****
//

```

```

void CALLBACK TimerProc( HWND hWnd, UINT Msg, UINT idTime, DWORD dwTime )
{
    KillTimer ( hWnd, idTime );

    switch( Msg )
    {
        case WM_TIMER:
        {
            switch ( idTime )
            {
                case MSR_TIMER_ID:
                {
                    if ( ReadMSR ( ) )
                    {
                        PostMessage ( hWnd, WM_COMMAND, MSR_DATA_READ, 0 );
                        return;
                    }
                }
            }
        }
    }

    SetTimer(hWnd, MSR_TIMER_ID, 1000, lpfnMyTimerProc);
}

//
// *****
// *
// * ResetMSR - initializes the MSR and sets up a timer to pole for MSR data      *
// *
// * *****
//
int ResetMSR ( HWND hWnd )
{
    //
    // *****
    // * Step 2: prepare the MSR to read data.                                     *
    // * *****
    //
    if ( padGetMagTrack ( 0,0,0 ) )
    {
        //
        // Initialize the contents of the buffers.
        //
        strcpy ( track1, "NO DATA READ" );
        strcpy ( track2, "NO DATA READ" );
        strcpy ( track3, "NO DATA READ" );

        //
        // In Windows it is not good to use "do" or "while" loops.
        // Instead we will use a timer to pole the MSR for data
        //
        lpfnMyTimerProc = (TIMERPROC) MakeProcInstance( (FARPROC) TimerProc,
                                                         theInstance );
        SetTimer(hWnd, MSR_TIMER_ID, 1000, lpfnMyTimerProc);

        return 1;
    }
    {
        MessageBox( hWnd, "ERROR: Unable to initialize Magnetic Stripe Reader!",
                    appName, MB_OK );

        return 0;
    }
}

//
// *****
// *

```

```

// * ReadMSR - reads the MSR data if any is available *
// *
// *****
//
int ReadMSR ( void )
{
    //
    // *****
    // * Step 3: Check if any of the tracks were read. *
    // *****
    //
    dataReadFromTrack1 = padGetMagTrack ( 1, track1, TRACK1_MAX );
    dataReadFromTrack2 = padGetMagTrack ( 2, track2, TRACK2_MAX );
    dataReadFromTrack3 = padGetMagTrack ( 3, track3, TRACK3_MAX );

    if
    (
        dataReadFromTrack1 ||
        dataReadFromTrack2 ||
        dataReadFromTrack3
    )
    {
        //
        // At this point we know that at least 1 track was read
        // successfully (the last track read, i.e., track 3).
        //
        // It is possible one or more of the other tracks were
        // checked before any data was retrieved from the card.
        //
        // As soon as one track contains data all of the other
        // tracks will as well.
        //
        // To make sure we get all of the data from all of the tracks
        // we will read all of them again.
        //
        //
        // *****
        // * Step 4: Read all of the tracks *
        // *****
        // Note: step 4 is not needed if only one track is being read
        //
        dataReadFromTrack1 = padGetMagTrack ( 1, track1, TRACK1_MAX );
        dataReadFromTrack2 = padGetMagTrack ( 2, track2, TRACK2_MAX );
        dataReadFromTrack3 = padGetMagTrack ( 3, track3, TRACK3_MAX );

        return 1;
    }
    return 0;
}

```

### ***PadCom MSR Sample Code for Win 95/NT:***

```

//
// *****
// *****
// **
// ** MSR.C **
// **
// ** This is a sample program showing how to use the Magnetic Stripe Reader **
// ** hardware available on some @pos.com device's such as the PenWare 3000 **
// **
// ** Platform: Windows 95/NT **
// **
// ** Compiler: Microsoft Visual C++ 4.0 **
// **
// **
// ** (C) Copyright 1999 @pos.com. **
// **

```

```
// *****
// *****
//

//
// *****
// *
// * Required header files
// *
// *****
//
#include <windows.h>

#include <string.h>
#include <stdio.h>
#include "PadCom.H"

//
// *****
// *
// * Constant values
// *
// *****
//

//
// @pos.com's magnetic stripe reader follows the MEGTEK standard.
// This standard has the following track sizes defined.
// We add 1 to include the appended NULL character at the end of each track.
//
#define TRACK1_MAX (74+1)
#define TRACK2_MAX (39+1)
#define TRACK3_MAX (106+1)

#define MSR_TIMER_ID 1
#define MSR_DATA_READ 1

//
// *****
// *
// * Function prototypes
// *
// *****
//
int WINAPI WinMain ( HINSTANCE, HINSTANCE, LPSTR, int );
LRESULT WndProc ( HWND, UINT, WPARAM, LPARAM );
void CALLBACK TimerProc ( HWND, UINT, UINT, DWORD );
int ReadMSR ( void );
int ResetMSR ( HWND );

//
// *****
// *
// * Global variables
// *
// *****
//
HINSTANCE theInstance;
HWND theWnd;
TIMERPROC lpfnMyTimerProc;

char *appName = "MSR";
char *appTitle = "MSR.EXE - Please swipe your Magnetic Card";

char
track1 [TRACK1_MAX],
track2 [TRACK2_MAX],
track3 [TRACK3_MAX];
```

```
int
    dataReadFromTrack1 = 0,
    dataReadFromTrack2 = 0,
    dataReadFromTrack3 = 0;

//
// *****
// *
// * WinMain - the application entry point
// *
// *****
//
int WINAPI WinMain
(
    HINSTANCE hInst,
    HINSTANCE hInstPrev,
    LPSTR lpstrCmdLine,
    int cmdShow
)
{
    MSG msg;
    WNDCLASS wc;

    theInstance = hInst;

    //
    // Register the window class if this is the first instance.
    //
    if( !hInstPrev )
    {
        wc.lpszMenuName = NULL;
        wc.lpszClassName = appName;
        wc.hInstance = hInst;
        wc.hIcon = NULL;
        wc.hCursor = NULL;
        wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
        wc.style = 0;
        wc.lpfnWndProc = (WNDPROC) WndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;

        if( !RegisterClass( &wc ) )
            return 0;
    }

    //
    // Attempt to create the main window
    //
    theWnd = CreateWindowEx
    (
        WS_EX_TOPMOST, appName, appTitle, WS_OVERLAPPED | WS_SYSMENU,
        CW_USEDEFAULT, CW_USEDEFAULT, 375, 225, NULL, NULL, hInst, NULL
    );

    //
    // If the window was not created then quit
    //
    if ( !theWnd )
    {
        return 0;
    }

    //
    // Show the main window
    //
    ShowWindow( theWnd, cmdShow );
    UpdateWindow( theWnd );

    //
    // Process the main message loop
```

```
//
while( GetMessage( LPMSG) &msg, NULL, 0, 0 )
{
    TranslateMessage ( (LPMSG) &msg );
    DispatchMessage ( (LPMSG) &msg );
}

return 0;
}

//
// *****
// *
// * WndProc - Message handler for the application
// *
// * *****
//
LRESULT WndProc(
    HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam
)
{
    char msrData [1024];

    switch( Msg )
    {
        case WM_CREATE:
        {
            //
            // *****
            // * Step 1: Turn on the @pos.com device using padConnect.*
            // *****
            //
            if( padConnect() )
            {
                if ( !ResetMSR ( hWnd ) )
                {
                    PostQuitMessage( 1 );
                }
            }
            else
            {
                //
                // Error
                //
                MessageBox( hWnd, "ERROR: No @pos.com device found!", appName, MB_OK );
                PostQuitMessage( 1 );
            }

            break;
        }

        case WM_COMMAND:
        {
            switch ( wParam )
            {
                case MSR_DATA_READ:
                {
                    //
                    // Display the data read from the Magnetic Stripe Reader
                    //
                    sprintf
                    (
                        msrData,
                        "Track 1 size: %d\n"
                        "Track 1 data: %s\n\n"
                        "Track 2 size: %d\n"
                        "Track 2 data: %s\n\n"
                        "Track 3 size: %d\n"
                    )
                }
            }
        }
    }
}
```

```
        "Track 3 data: %s\n\n",
        dataReadFromTrack1,
        track1,
        dataReadFromTrack2,
        track2,
        dataReadFromTrack3,
        track3
    );

    MessageBox( hWnd, msrData, "MSR.EXE - Data received from the Magnetic
        Card", MB_OK );

    //
    // Reset
    //
    if ( !ResetMSR ( hWnd ) )
    {
        PostQuitMessage( 1 );
    }
    }
    break;
}

case WM_DESTROY:
{
    //
    // Turn of the @pos.com device
    //
    padOff();

    PostQuitMessage( 0 );

    break;
}

case WM_CHAR:
{
    DestroyWindow( hWnd );
    break;
}

default:
{
    return DefWindowProc( hWnd, Msg, wParam, lParam );
}
}

return 0;
}

//
// *****
// *
// * TimerProc - Message handler for timer used to read the MSR *
// *
// * *****
//
void CALLBACK TimerProc( HWND hWnd, UINT Msg, UINT idTime, DWORD dwTime )
{
    KillTimer ( hWnd, idTime );

    switch( Msg )
    {
        case WM_TIMER:
        {
            switch ( idTime )
            {
```

```
        case MSR_TIMER_ID:
        {
            if ( ReadMSR () )
            {
                PostMessage ( hWnd, WM_COMMAND, MSR_DATA_READ, 0 );
                return;
            }
        }
    }
}

SetTimer(hWnd, MSR_TIMER_ID, 1000, lpfnMyTimerProc);
}

//
// *****
// *
// * ResetMSR - initializes the MSR and sets up a timer to pole for MSR data *
// *
// *****
//
int ResetMSR ( HWND hWnd )
{
    //
    // *****
    // * Step 2: prepare the MSR to read data. *
    // *****
    //
    if ( padGetMagTrack ( 0,0,0 ) )
    {
        //
        // Initialize the contents of the buffers.
        //
        strcpy ( track1, "NO DATA READ" );
        strcpy ( track2, "NO DATA READ" );
        strcpy ( track3, "NO DATA READ" );

        //
        // In Windows it is not good to use "do" or "while" loops.
        // Instead we will use a timer to pole the MSR for data
        //
        lpfnMyTimerProc = (TIMERPROC) MakeProcInstance( (FARPROC) TimerProc,
                                                    theInstance );
        SetTimer(hWnd, MSR_TIMER_ID, 1000, lpfnMyTimerProc);

        return 1;
    }
    {
        MessageBox( hWnd, "ERROR: Unable to initialize Magnetic Stripe Reader!",
                    appName, MB_OK );

        return 0;
    }
}

int ReadMSR ( void )
{
    //
    // *****
    // * Step 3: Check if any of the tracks were read. *
    // *****
    //
    dataReadFromTrack1 = padGetMagTrack ( 1, track1, TRACK1_MAX );
    dataReadFromTrack2 = padGetMagTrack ( 2, track2, TRACK2_MAX );
    dataReadFromTrack3 = padGetMagTrack ( 3, track3, TRACK3_MAX );

    if
    (
```

```
    dataReadFromTrack1 ||
    dataReadFromTrack2 ||
    dataReadFromTrack3
)
{
    //
    // At this point we know that at least 1 track was read
    // successfully (the last track read, i.e., track 3).
    //
    // It is possible one or more of the other tracks were
    // checked before any data was retrieved from the card.
    //
    // As soon as one track contains data all of the other
    // tracks will as well.
    //
    // To make sure we get all of the data from all of the tracks
    // we will read all of them again.
    //
    //
    // *****
    // * Step 4: Read all of the tracks *
    // *****
    // Note: step 4 is not needed if only one track is being read
    //
    dataReadFromTrack1 = padGetMagTrack ( 1, track1, TRACK1_MAX );
    dataReadFromTrack2 = padGetMagTrack ( 2, track2, TRACK2_MAX );
    dataReadFromTrack3 = padGetMagTrack ( 3, track3, TRACK3_MAX );

    return 1;
}
return 0;
}
```

# INDEX

## A

aspect ratio, 11, 78, 79, 97, 98

## B

baud rate, 25, 28, 96  
buttons, 19, 21, 48, 54, 60, 77

## C

clipping, 18, 33, 80  
communications port, 20, 44, 45, 91, 92, 93  
coordinates, 11, 20, 32, 51, 61, 75, 96

## D

debug, 83  
dots-per-inch, 20, 78, 79

## E

encryption key, 51, 87  
error, 18, 19, 28, 30, 43, 51, 52, 55, 74

## F

font, 38, 51, 73, 85

## H

height, 11, 19, 20  
horizontal, 11, 20, 32, 43, 49, 53, 61, 78, 96, 99

## I

IBM OS/2, 74, 97, 98  
inches, 11, 20, 44, 49, 53, 78, 99

## L

**LCD display**, 19, 25, 50, 71, 87, 90  
light, 19, 21, 54  
line to, 12, 21, 32, 52

## M

magnetic card reader, 25, 28, 39, 42  
memory, 55, 56, 57, 58, 59  
memory object, 28  
Microsoft Windows, 74, 97, 98  
millimeter, 20, 21  
move to, 12, 21, 32, 51, 52

## N

notification of activity, 74, 99

## P

pad, 11, 43, 78, 79, 98  
pad surface, 11, 18, 33, 49, 53, 74, 75, 97, 99  
pen, 32, 51, 52

## R

recording, 18, 21, 32, 62, 73, 74, 79, 96, 99  
resolution, 11, 20, 44, 78, 79, 97, 98  
revision number, 35, 47

## S

scaling, 11, 20, 44, 49, 53, 78, 97, 99  
scan rate, 46  
SigKit library, 2, 7  
stop recording, 96  
stroke, 12, 21, 32, 51, 52

## T

time, 22, 96

## V

vertical, 11, 20, 32, 43, 53, 61, 78, 96

## W

width, 11, 19, 20  
**window handle**, 74