3M™ MicroTouch™
Software MT 7
Gesture Application Programming Interface (API) Guide

Read and understand all safety information contained in this document before using this product.
The information in this document is subject to change without notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of 3M Touch Systems, Inc. 3M may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you license to these patents, trademarks, copyrights, or other intellectual property except as expressly provided in any written license agreement from 3M Touch Systems, Inc.

The information provided in this document is intended as a guide only. For the latest detailed engineering specifications, please contact your 3M Touch Systems, Inc. Application Engineer. 3M Touch Systems, Inc. is committed to continually improving product designs, as a result, product specifications may be subject to change without notification.

"RoHS compliant 2005/95/EC" means that the product or part does not contain any of the following substances in excess of the following maximum concentration values in any homogeneous material, unless the substance is in an application that is exempt under RoHS: (a) 0.1% (by weight) for lead, mercury, hexavalent chromium, polybrominated biphenyls or polybrominated diphenyl ethers; or (b) 0.01% (by weight) for cadmium. This information represents 3M’s knowledge and belief, which may be based in whole or in part on information provided by third party suppliers to 3M.

NOTICE: Given the variety of factors that can affect the use and performance of a 3M Touch Systems, Inc. Product (the “Product”), including that solid state equipment has operation characteristics different from electromechanical equipment, some of which factors are uniquely within User’s knowledge and control, it is essential that User evaluate the 3M Touch Systems, Inc. Product and software to determine whether it is suitable for User’s particular purpose and suitable for User’s method of application. 3M Touch Systems, Inc. statements, engineering/technical information, and recommendations are provided for User’s convenience, but their accuracy or completeness is not warranted. 3M Touch Systems, Inc. products and software are not specifically designed for use in medical devices as defined by United States federal law. 3M Touch Systems, Inc. products and software should not be used in such applications without 3M Touch Systems, Inc. express written consent. User should contact its sales representative if User’s opportunity involves a medical device application.

IMPORTANT NOTICE TO PURCHASER: Specifications are subject to change without notice. These 3M Touch Systems, Inc. Products and software are warranted to meet their published specifications from the date of shipment and for the period stated in the specification. 3M Touch Systems, Inc. makes no additional warranties, express or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose. User is responsible for determining whether the 3M Touch Systems, Inc. Products and software are fit for User’s particular purpose and suitable for its method of production, including intellectual property liability for User's application. If the Product, software or software media is proven not to have met 3M Touch Systems, Inc. warranty, then 3M Touch Systems, Inc. sole obligation and User’s and Purchaser’s exclusive remedy, will be, at 3M Touch Systems, Inc. option, to repair or replace that Product quantity or software media or to refund its purchase price. 3M Touch Systems, Inc. has no obligation under 3M Touch Systems, Inc. warranty for any Product, software or software media that has been modified or damaged through misuse, accident, neglect, or subsequent manufacturing operations or assemblies by anyone other than 3M Touch Systems, Inc. 3M Touch Systems, Inc. shall not be liable in any action against it in any way related to the Products or software for any loss or damages, whether non-specified direct, indirect, special, incidental or consequential (including downtime, loss of profits or goodwill) regardless of the legal theory asserted.

Copyright 3M © 2009  All rights reserved.

Document Title: 3M™ MicroTouch™ Software MT 7 Gesture API Guide
Document Number: 36963, Version 02

3M, the 3M logo, MicroTouch, and the MicroTouch logo are either registered trademarks or trademarks of 3M in the United States and/or other countries.

Windows and/or other Microsoft products referenced herein are either registered trademarks or trademarks of Microsoft Corporation in the U.S. and/or other countries.

All other trademarks are the property of their respective owners.
Contents

Chapter 1  Introduction
   About This Manual .....................................................................................................9
   Sensor Care and Cleaning .................................................................9
   3M Touch Systems Support Services .....................................10
   Contact 3M Touch Systems .........................................................10

Chapter 2  Installing MT 7 Gesture API Software
   MT 7 Gesture Software Overview .................................................11
      Development Files ........................................................................11
      Runtime Libraries ........................................................................11
   Installing MT 7 Gesture Software (for Windows®) ......................12
   Installing MT 7 Gesture Software (for Linux®) ..............................12
      Preparation ................................................................................12
      General Instructions ................................................................13
      Normal Installation ..................................................................13
      Installing on Write-Protected Systems ........................................13
      Other Options ..........................................................................13
      Error Messages During Installation .......................................14
      Removing MT 7 For Linux ........................................................14
      Known Issues ..........................................................................14
   Performing a Silent Install (for Windows®) .................................14
   Customizing Your Touch Settings (for Windows®) .....................15
   Uninstalling MT 7 Gesture Software (for Windows®) ....................15

Chapter 3  Trace API Details
   Overview ...........................................................................................17
   Using Enumerators .................................................................18
   Using Sensors ..............................................................................18
   Using Traces ................................................................................19
   Using Events ..............................................................................19
   Polling, Monitoring, and Callbacks .......................................20
   General Notes on Handles ..........................................................22
   Enumeration API Routines .......................................................23
      MT7TraceCreateEnum ..............................................................23
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT7TraceGetEnumID</td>
<td>23</td>
</tr>
<tr>
<td>MT7TraceCloneEnum</td>
<td>23</td>
</tr>
<tr>
<td>MT7TraceGetNextSensor</td>
<td>24</td>
</tr>
<tr>
<td>MT7TraceReleaseEnum</td>
<td>24</td>
</tr>
<tr>
<td>Touch Sensor API Routines</td>
<td>24</td>
</tr>
<tr>
<td>MT7TraceGetSensorID</td>
<td>25</td>
</tr>
<tr>
<td>MT7TraceGetSensorName</td>
<td>25</td>
</tr>
<tr>
<td>MT7TraceCloneSensor</td>
<td>25</td>
</tr>
<tr>
<td>MT7TraceGetNextTrace</td>
<td>26</td>
</tr>
<tr>
<td>MT7TraceSetSensorCallback</td>
<td>26</td>
</tr>
<tr>
<td>MT7TraceReleaseSensor</td>
<td>26</td>
</tr>
<tr>
<td>Trace API Routines</td>
<td>27</td>
</tr>
<tr>
<td>MT7TraceGetTraceID</td>
<td>27</td>
</tr>
<tr>
<td>MT7TraceCloneTrace</td>
<td>28</td>
</tr>
<tr>
<td>MT7TraceGetNextEvent</td>
<td>28</td>
</tr>
<tr>
<td>MT7TraceGetPriorEvent</td>
<td>28</td>
</tr>
<tr>
<td>MT7TraceSetNextEvent</td>
<td>29</td>
</tr>
<tr>
<td>MT7TraceGetEventCount</td>
<td>29</td>
</tr>
<tr>
<td>MT7TraceGetEvent</td>
<td>29</td>
</tr>
<tr>
<td>MT7TraceGetTraceSensor</td>
<td>30</td>
</tr>
<tr>
<td>MT7TraceSetTraceCallback</td>
<td>30</td>
</tr>
<tr>
<td>MT7TraceReleaseTrace</td>
<td>31</td>
</tr>
<tr>
<td>Event API Routines</td>
<td>31</td>
</tr>
<tr>
<td>MT7TraceGetEventID</td>
<td>31</td>
</tr>
<tr>
<td>MT7TraceCloneEvent</td>
<td>32</td>
</tr>
<tr>
<td>MT7TraceGetEventType</td>
<td>32</td>
</tr>
<tr>
<td>MT7TraceGetEventX</td>
<td>32</td>
</tr>
<tr>
<td>MT7TraceGetEventY</td>
<td>32</td>
</tr>
<tr>
<td>MT7TraceGetEventTime</td>
<td>33</td>
</tr>
<tr>
<td>MT7TraceGetEventTrace</td>
<td>33</td>
</tr>
<tr>
<td>MT7TraceReleaseEvent</td>
<td>33</td>
</tr>
<tr>
<td>Poll and Monitor API Routines</td>
<td>34</td>
</tr>
<tr>
<td>MT7TracePoll</td>
<td>34</td>
</tr>
<tr>
<td>MT7TraceStartMonitor</td>
<td>34</td>
</tr>
<tr>
<td>MT7TraceStopMonitor</td>
<td>35</td>
</tr>
<tr>
<td>MT7TraceSetArrivalCallback</td>
<td>35</td>
</tr>
<tr>
<td>MT7TraceSetUniversalCallback</td>
<td>35</td>
</tr>
<tr>
<td>Callback Routines</td>
<td>36</td>
</tr>
<tr>
<td>MT7TraceCallback</td>
<td>36</td>
</tr>
<tr>
<td>Trace API Object Handles</td>
<td>36</td>
</tr>
<tr>
<td>MT7TraceEnum</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceSensor</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceTrace</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceEvent</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceReason</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceReason_Arrival</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceReason_Departure</td>
<td>37</td>
</tr>
<tr>
<td>MT7TraceReason_NewTrace</td>
<td>38</td>
</tr>
<tr>
<td>MT7TraceReason_NewEvent</td>
<td>38</td>
</tr>
<tr>
<td>MT7TraceEventType</td>
<td>39</td>
</tr>
</tbody>
</table>
## Chapter 4 Stroke API Details

### Overview .................................................. 41
### Defining Reference Strokes .......................... 41
### Using Recognition Engines ......................... 42
### Using Stroke Matches ................................. 43
### Polling, Monitoring, and Callbacks ............... 44
### General Notes on Handles ......................... 45
### Engine API Routines ................................. 45

- MT7StrokeCreateEngine .................................. 46
- MT7StrokeCloneEngine .................................. 46
- MT7StrokeGetEngineName ............................... 47
- MT7StrokeGetEngineFile .................................. 47
- MT7StrokeGetEngineID ................................... 48
- MT7StrokeAddAllSensors ............................... 48
- MT7StrokeRemoveAllSensors .......................... 48
- MT7StrokeAddSensor ....................................... 49
- MT7StrokeRemoveSensor ............................... 49
- MT7StrokeAddTrace ....................................... 49
- MT7StrokeRemoveTrace ................................. 50
- MT7StrokeRemoveAllTraces ............................ 50
- MT7StrokeReleaseEngine .............................. 50

### Match API Routines .................................... 51

- MT7StrokeCloneMatch ..................................... 51
- MT7StrokeGetMatchName ............................... 51
- MT7StrokeGetMatchID .................................... 51
- MT7StrokeGetMatchEngine ............................. 52
- MT7StrokeGetMatchTraceCount ..................... 52
- MT7StrokeGetMatchTrace .................. 52
- MT7StrokeReleaseMatch ............................... 53

### Callback Routines ...................................... 53

- MT7StrokeCallback ....................................... 53

### MT7Stroke API Object Handles ..................... 54

- MT7StrokeEngine .......................................... 54
- MT7StrokeMatch ........................................... 54

### MT7MatchReason ........................................ 54

- MT7MatchReason_NoMatch ............................. 54
- MT7MatchReason_Match ................................. 54

### Configuration File ..................................... 54

- File Format ................................................ 54
- ‘Strokes’ Element ......................................... 54
- ‘Stroke’ Element .......................................... 55
- Reference Stroke Definition .......................... 55
- Attribute List ............................................. 56
- Parser Errors ............................................. 57
- XML Format Errors ...................................... 57
- File Content Errors ................................. 58
## Chapter 5  Radial Menu Details

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>59</td>
</tr>
<tr>
<td>Using Libraries</td>
<td>60</td>
</tr>
<tr>
<td>Using Templates</td>
<td>60</td>
</tr>
<tr>
<td>Using Menus</td>
<td>61</td>
</tr>
<tr>
<td>Using Actions</td>
<td>62</td>
</tr>
<tr>
<td>Polling, Monitoring, and Callbacks</td>
<td>62</td>
</tr>
<tr>
<td>General Notes on Handles</td>
<td>63</td>
</tr>
<tr>
<td>Library API Routines</td>
<td>63</td>
</tr>
<tr>
<td>MT7RadialCreateLibrary</td>
<td>63</td>
</tr>
<tr>
<td>MT7RadialCloneLibrary</td>
<td>64</td>
</tr>
<tr>
<td>MT7RadialGetLibraryName</td>
<td>64</td>
</tr>
<tr>
<td>MT7RadialGetLibraryFile</td>
<td>65</td>
</tr>
<tr>
<td>MT7RadialGetLibraryID</td>
<td>65</td>
</tr>
<tr>
<td>MT7RadialGetTemplate</td>
<td>65</td>
</tr>
<tr>
<td>MT7RadialGetTemplateCount</td>
<td>66</td>
</tr>
<tr>
<td>MT7RadialGetTemplateByIndex</td>
<td>66</td>
</tr>
<tr>
<td>MT7RadialReleaseLibrary</td>
<td>66</td>
</tr>
<tr>
<td>Template API Routines</td>
<td>67</td>
</tr>
<tr>
<td>MT7RadialCloneTemplate</td>
<td>67</td>
</tr>
<tr>
<td>MT7RadialGetTemplateName</td>
<td>67</td>
</tr>
<tr>
<td>MT7RadialGetTemplateID</td>
<td>68</td>
</tr>
<tr>
<td>MT7RadialGetTemplateLibrary</td>
<td>68</td>
</tr>
<tr>
<td>MT7RadialStartMenu</td>
<td>68</td>
</tr>
<tr>
<td>MT7RadialReleaseTemplate</td>
<td>69</td>
</tr>
<tr>
<td>Menu API Routines</td>
<td>70</td>
</tr>
<tr>
<td>MT7RadialCloneMenu</td>
<td>70</td>
</tr>
<tr>
<td>MT7RadialGetMenuID</td>
<td>70</td>
</tr>
<tr>
<td>MT7RadialGetMenuName</td>
<td>70</td>
</tr>
<tr>
<td>MT7RadialGetMenuX</td>
<td>71</td>
</tr>
<tr>
<td>MT7RadialGetMenuY</td>
<td>71</td>
</tr>
<tr>
<td>MT7RadialGetMenuTemplate</td>
<td>71</td>
</tr>
<tr>
<td>MT7RadialAddTrace</td>
<td>72</td>
</tr>
<tr>
<td>MT7RadialRemoveTrace</td>
<td>72</td>
</tr>
<tr>
<td>MT7RadialReleaseMenu</td>
<td>72</td>
</tr>
<tr>
<td>Action API Routines</td>
<td>73</td>
</tr>
<tr>
<td>MT7RadialCloneAction</td>
<td>73</td>
</tr>
<tr>
<td>MT7RadialGetActionName</td>
<td>73</td>
</tr>
<tr>
<td>MT7RadialGetActionID</td>
<td>73</td>
</tr>
<tr>
<td>MT7RadialGetActionMenu</td>
<td>74</td>
</tr>
<tr>
<td>MT7RadialGetActionTrace</td>
<td>74</td>
</tr>
<tr>
<td>MT7RadialReleaseAction</td>
<td>74</td>
</tr>
<tr>
<td>Callback Routines</td>
<td>75</td>
</tr>
<tr>
<td>MT7RadialCallback</td>
<td>75</td>
</tr>
<tr>
<td>MT7Radial API Object Handles</td>
<td>75</td>
</tr>
<tr>
<td>MT7RadialLibrary</td>
<td>75</td>
</tr>
<tr>
<td>MT7RadialTemplate</td>
<td>75</td>
</tr>
<tr>
<td>MT7RadialMenu</td>
<td>76</td>
</tr>
<tr>
<td>MT7RadialAction</td>
<td>76</td>
</tr>
<tr>
<td>MT7RadialReason</td>
<td>76</td>
</tr>
<tr>
<td>MT7RadialReleaseReason</td>
<td>76</td>
</tr>
</tbody>
</table>
Chapter 6  Box Tool API Details

Overview .................................................................................................................. 89
Using Box Tools........................................................................................................ 89
Polling, Monitoring, and Callbacks ........................................................................... 90
General Notes on Handles ....................................................................................... 91
Box Tool API Routines ............................................................................................ 92
MT7BoxToolCreateFixedTool ................................................................................. 92
MT7BoxToolCreateFixed2Tool .............................................................................. 93
MT7BoxToolCreateTool ....................................................................................... 94
MT7BoxToolCloneTool ....................................................................................... 94
MT7BoxToolCloneToolWithCallback ................................................................... 95
MT7BoxToolGetToolID ......................................................................................... 95
MT7BoxToolGetInitialX ....................................................................................... 96
MT7BoxToolGetInitialY ....................................................................................... 96
MT7BoxToolGetCurrentX ...................................................................................... 96
MT7BoxToolGetCurrentY ...................................................................................... 96
MT7BoxToolGetInitialDistance ............................................................................ 97
MT7BoxToolGetInitialAngle ............................................................................... 97
MT7BoxToolGetCurrentDistance ........................................................................... 98
MT7BoxToolGetCurrentAngle ............................................................................. 98
MT7BoxToolGetDistanceChange ........................................................................... 98
MT7BoxToolGetDistanceRatio ............................................................................. 99
MT7BoxToolGetAngleChange ............................................................................. 99
MT7BoxToolReleaseTool ................................................................. 99
Callback Routines ........................................................................ 100
MT7BoxToolCallback ................................................................. 100
MT7 Box Tool API Object Handles .................................................. 100
MT7BoxTool ............................................................................ 101
MT7BoxToolReason .................................................................... 101
MT7BoxToolReason_Update ......................................................... 101
MT7BoxToolReason_End ............................................................. 101
MT7BoxToolCenter ..................................................................... 101
MT7BoxToolCenter_Touchdown ..................................................... 101
MT7BoxToolCenter_Current ......................................................... 101
MT7BoxToolInput ....................................................................... 101
MT7BoxToolInput_None ............................................................. 101
MT7BoxToolInput_Trace ............................................................. 101

Appendix A Troubleshooting
Overview ....................................................................................... 103
Controlling Diagnostic Outputs ...................................................... 103
Windows ..................................................................................... 103
Linux ......................................................................................... 104
General Purpose Diagnostics ........................................................ 104
MT7GestureMemoryDump ........................................................... 104
MT7Stroke Diagnostics ................................................................. 104
MT7StrokeParserErrors ............................................................... 104
MT7StrokeImageOutput ............................................................... 104
MT7StrokeCompareStrokes ......................................................... 104
MT7StrokeShowAllScores ........................................................... 105
MT7Radial Diagnostics ................................................................. 105
MT7RadialParserErrors .............................................................. 105
Programs Will Not Start .............................................................. 105

Appendix B 107Glossary

Appendix C Tuning Parameters

Appendix D Sample Code
Simple Trace Example ................................................................. 111
Trace with Enumeration Example .................................................. 112
Simple Stroke Example ............................................................... 114
XML file (sample.xml) ............................................................... 114
Code ........................................................................................ 114
Complex Stroke Example ............................................................ 115
Simple Radial Example .............................................................. 117
XML file (sample.xml) ............................................................... 117
Code ........................................................................................ 117
Simple Box Tool Example .......................................................... 118
Radial Menu with a Box Tool Example ........................................ 120
CHAPTER 1

Introduction

About This Manual

This manual describes how to:

- Install 3M™ MicroTouch™ Software MT 7 Gesture API
- Use the API to customize your work environment.
- Troubleshoot the API

Sensor Care and Cleaning

The sensor requires very little maintenance. 3M Touch Systems, Inc. recommends that you periodically clean the glass surface.

⚠️ CAUTION

To reduce the risk of the potentially hazardous situations associated with the use of isopropyl alcohol which may result in minor or moderate injury or property damage:

Follow all instructions and recommendations in the alcohol manufacturer's Material Safety Data Sheet and product label.

Typically, an isopropyl alcohol and water solution ratio of 50:50 is the best cleaning agent for your sensor. You can also use straight isopropyl alcohol. Be sure to follow solvent manufacturer's precautions and directions for use when using any solvents.

It is important to avoid using any caustic chemicals on the sensor.

Always dampen the cloth and then clean the sensor. Be sure to spray the cleaning liquid onto the cloth, not the sensor, so that drips do not seep inside the display or stain the bezel.

Apply the cleaner with a soft, lint-free cloth. Avoid using gritty cloths.

Always handle the sensor with care. Do not pull on or stress the flex tail.
3M Touch Systems Support Services

3M Touch Systems, Inc. provides extensive support services through our website and technical support organization. Visit the 3M Touch Systems website at http://www.3Mtouch.com/, where you can download touch system software and drivers, obtain regularly updated technical documentation on 3M Touch Systems products, and learn more about our company.

Whenever you contact Technical Support, please provide the following information:

- Touch display size, part number and serial number
- Current driver version
- Operating system used
- Information on additional peripherals

Technical Support is available Monday through Friday 8:30 a.m. to 5:30 p.m. with limited call back service after 5:30 p.m. until 8:00 p.m. US Eastern Standard Time – 9 a.m. to 5 p.m. throughout Europe.

You can contact 3M Touch Systems, Inc. Technical Support (US only -- Eastern Standard Time) by calling the hot line, sending email or a fax.

- Technical Support Hot Line: 978-659-9200
- Technical Support Fax: 978-659-9400
- Toll Free: 1-866-407-6666 (Option 3)
- Email: US-TS-techsupport@mmm.com

Contact 3M Touch Systems

Contact information for all offices can be found on our website at: http://www.3Mtouch.com/
CHAPTER 2
Installing MT 7 Gesture API Software

The 3M™ MicroTouch™ Software MT 7 Gesture Application Programming Interface Guide enables developers to add gesture support to their touch sensitive applications. MT 7 Gesture Software provides this support for MicroTouch™ sensors under the following operating systems:

- Microsoft® Windows® 2000
- Microsoft® Windows® Vista
- Microsoft® Windows® XP
- Linux® Kernel 2.6

MT 7 Gesture Software Overview

The MT 7 Gesture Software contains libraries for use by your program. These libraries allow you to track traces from the touch sensors. You can match these traces against a set of strokes and react to these gestures. You can also use these traces as input to radial menus to allow the user to select from a group of options. There is also a box tool that assists with scaling and rotation based on traces.

The MT 7 Gesture Software has two components:

- Development files
- Runtime libraries

Development Files

These files are needed for application development. This includes the necessary C language header files and, for Microsoft® Windows® development systems, library files (lib files) needed for linking. Also included is a copy of this manual.

This option automatically includes the runtime library files.

Runtime Libraries

These files, dynamic link libraries (dll files) for MS Windows systems or shared objects (so files) for Linux systems, are used by applications. Intended for system deployment, this option allows the installation of only the files needed for runtime systems.
Installing MT 7 Gesture Software (for Windows®)

When you install MT 7 Gesture software, you have two installation options:

- Development (installs all components)
- Runtime (installs only the runtime libraries)

To install MT 7 Gesture software:

1. Make sure you already have the latest MT 7 Software drivers installed. If needed, download this from the www.3Mtouch.com website and follow the instructions.
2. If you received this package as a compressed file, unzip it into a directory of your choice. Using Windows® Explorer, browse to that folder.
3. Double-click the Setup.exe file and follow the onscreen instructions to begin the installation.
4. When prompted to choose the installation type, select whether to perform a Development or Runtime install (or, if available, Upgrade).

This automatically installs the requested MT 7 Gesture software files and creates a MT 7 Gesture group in your Windows Start menu.

Installing MT 7 Gesture Software (for Linux®)

The MT 7 for Linux® software is proprietary and not open source. The distribution is an executable program. Run this program using the following command line:

```bash
chmod a+x mt7.*.bin
```

You will then be asked to either accept or decline the license agreement. If you accept the agreement, the program produces a 'tar.gz' file. Follow the remaining instructions to complete the installation process. For your convenience, the 'tar.gz' file contains a copy of the license in the file 'License.txt'.

Preparation

You must log in the Linux® operating system as the root user or use the 'su' command to get super user access. On some systems, you may need to run in single user mode (run level 1 on some systems). The installation process requires write access to various system directories. By default, the directories are:

- /etc/udev or /etc/hotplug
- /usr/lib/xorg/modules/input or /usr/X11R6/lib/modules/input
- /usr/lib
- /etc/init.d or /etc/rc.d
- /etc/X11/xinit/xinitrc.d or /etc/X11/xsessions.d

After installation completes, the MT 7 for Linux® software no longer requires write access to these directories.
General Instructions

The MT 7 for Linux® package arrives as a tar.gz file. By default, the expanded package resides in the subdirectory 'twscreen'. A script named 'Install' installs the package. Installation includes creating the necessary init script, creating symbolic links in /usr/lib, making a X input module available, enabling USB hotplug for 3M™ MicroTouch™ devices, and creating a Remove script.

Before installing the package, read below to see if any of these special situations apply to you. If they do, you may need to edit the Install script.

If you run the Install script and then discover errors, always run the Remove script to undo the installation. This removes possibly erroneous links into your system. Then edit the Install script, make the necessary changes, and install again.

Normal Installation

Copy the tar.gz file into the directory where you want the package to reside, such as /etc. Issue the command 'tar xzf twscreen.<version>.tar.gz' where <version> is a version number string of the tar.gz file. This creates the directory 'twscreen' in the current directory. From there, go into the twscreen directory and issue the command './Install'. This script installs links in the Linux® and X Windows system for your touch sensor to work with your system.

Installing on Write-Protected Systems

The MT 7 for Linux® software requires access to a writeable directory for runtime and configuration data. By default, it uses a 'data' directory in the twscreen directory. If you need to have the package in a read-only directory, then decide on a writeable directory for the package to use. Edit the Install script and search for the line 'DataDir=""'. Change the definition of DataDir to the writeable directory. Save the changes and run the script. This creates the data directory where desired and creates a link in the package directory to the data directory.

Other Options

The MT 7 for Linux® software should install without other changes on most Linux® systems. However, if you have a customized directory structure or the installation script fails, you may need to edit the Install script and change one or more of the following script variables.

LibDir

The MT 7 for Linux® software uses several shared objects. The Install script links these into /usr/lib. If your system's libraries are elsewhere, change this variable to point to the correct path. This directory is mandatory and the software cannot run without its libraries in a common library directory.

SEDir1 and SEDir2

Some of the shared objects in the MT 7 for Linux® software require access to objects secured by SELinux. If the Install script detects SELinux, it assigns a security type to some shared objects. The script inspects the directories /usr/selinux/booleans and /selinux/booleans to determine if SELinux is active. If your SELinux is in a directory other than these, change the definition of one of these variables.
SEGGivePermission
If the Install script detects SELinux, it assigns the security type ‘texrel_shlib_t’ to some shared objects. If this is not appropriate for your system, edit the script and change this variable.

If you want the configuration file copied elsewhere, change the XorgConf value to the desired file name.

Error Messages During Installation
This covers error messages generated by the Install script. Programs used by the script may generate their own messages. This section does not cover those messages.

ERROR: MT 7 for Linux® not installed - shared memory support not detected
The MT 7 for Linux® software uses shared memory. Some older Linux® systems do not provide this support and the package will not run. Contact 3M Touch Systems for other options.

Cannot find needed libstdc++.so in /usb/lib
Variations of the message may refer to other directories depending on the setting of the LibDir variable in the Install script. The installation script could not find the standard C++ shared objects. Confirm that the LibDir variable is set correctly. If so, you need to install the C++ shared objects or, if already present, create a symbolic link using the name libstdc++.so to the C++ shared object.

Removing MT 7 For Linux
In the installation directory, run the script Remove. This removes the data directories and disengages MT 7 for Linux® from the system. The files remain. You must manually remove the files from the twscreen directory.

Known Issues
Refer to the Readme.txt file for a list of known issues associated with this release.

Performing a Silent Install (for Windows®)
If you have to perform multiple system installations, you may want to take advantage of the MT 7 Gesture software silent install.

To begin, you must create a silent install script that contains the recorded installation instructions. You create this script by running the Setup program in record mode. During a silent install, the Setup program receives input from the recorded install script rather than from a user. Once you record an installation script, you can run it on as many similar systems as you like.

The first step in producing a silent installation is to record your standard installation. Open a command line window and set your working directory to the directory containing the MT 7 Gesture for Windows™ setup program. Issue the command "setup -r" and run the installation normally.
When the installation finishes, it creates a recording file, setup.iss, in your Windows™ system directory (such as C:\Windows). Copy this file to the directory containing the MT 7 Gesture for Windows™ setup program.

To run the silent installation, simply issue the command "setup -s". This replays the installation you recorded above.

Note that the silent installation fails if the installer reacts differently than when it was recorded. This could happen if the folder for the MT 7 Software driver differs between machines. You should test the silent installation on all expected configurations.

Since the recording process creates a file, the MT 7 Gesture software must be on a hard drive and not on a CD-ROM. Once the file is created, you can make a CD-ROM, or equivalent media, with the distribution files and the setup.iss file. Your use of this CD-ROM is subject to the license agreement of this software. This feature does not give you the right to distribute this software if that right is not already granted to you by the license agreement.

Customizing Your Touch Settings (for Windows®)

By default, the installer places its files in the existing MT 7 Software driver folders, usually ‘C:\Program Files\MicroTouch\MT 7’. The runtime libraries go in this folder. The development files, if installed, go into a subfolder called ‘Gesture’. If you chose a custom installation directory, the runtime libraries are copied to that directory and the development files go into a ‘Gesture’ subfolder there.

You need to configure your development system to find these files. Usually, this is a include file path to the header files and a library file path to the development library files. If you do not use a custom directory during installation, use the path ‘C:\Program Files\MicroTouch\MT 7\Gesture’ for both include files and library files.

The operating system will need to find the runtime libraries to run your program. You need to add the directory ‘C:\Program Files\MicroTouch\MT 7’ to your system PATH variable. This is set using the System properties page through the control panel. Note that if you used a custom installation directory, you may need to add two paths, one for the runtime libraries and one to the MT 7 Software driver directory.

Uninstalling MT 7 Gesture Software (for Windows®)

The Uninstall program removes all MT 7 Gesture software components from your computer. These components include MT 7 Gesture runtime and development files and folders.

To uninstall MT 7 Gesture software:
1. Close MT 7 Gesture software if it is open.
2. Click on Start → Programs → MicroTouch → MT 7 Gesture → Uninstall MT 7 Gesture Software
   The path may be different if you performed a custom installation and chose a custom name for the MT 7 Gesture software program group.
3. Follow the onscreen instructions to remove MT 7 Gesture software. 3M Touch Systems strongly suggests you reboot your system when complete.
CHAPTER 3

Trace API Details

Overview

The MT7Trace API allows applications to receive touch inputs from controllers. The API has four types of objects: enumerators, sensors, traces, and events. Most applications use only the trace and event objects. Applications that require finer control over the touch system may use the enumerator and sensor objects.

All applications use the trace and event objects. Any contact with a touch sensor produces a trace. A trace begins with a touchdown event, with or without a subsequent drag event, and ends with a liftoff event. The application is responsible for processing traces and their events.

Some applications may require finer control of the touch system, including how it handles each sensor. In this case, an application first creates an enumerator. It uses the enumerator to get handles for the sensors on the computer. It then polls or monitors the sensors for traces. In turn, the application may poll or monitor the traces for events.

The MT7Trace API uses a callback system to inform the application that an event occurred within the touch system. There are four reasons why the API invokes a callback:

- Device arrival
• Device departure
• New trace creation
• New event on an existing trace

An application can attach a callback to a specific sensor or trace object or use a universal callback. It can also use an arrival callback to detect if a sensor arrives or departs, i.e., when a user attaches a USB controller or removes it.

Finally, the application must enable touch acquisition for the MT7Trace API. There are two ways that the application can do this. The application can periodically poll the API. It can also start the MT7Trace monitor thread that runs separate from the application’s main thread. The API invokes the callbacks only when the application polls the API or from within the context of its monitor thread.

Using Enumerators

Usually a computer has only one touch sensor on it. However, some systems may have multiple sensors attached. To determine how many sensors are attached to a system and to obtain handles for them, an application needs to enumerate them.

The routine MT7TraceCreateEnum enumerates the sensors currently attached to the system and returns a handle to that information. The application takes the enumerator handle and repeatedly calls MT7TraceGetNextSensor to get the handles for the sensors. When MT7TraceGetNextSensor indicates there are no more sensors, the application calls MT7TraceReleaseEnum to release the enumerator.

An application can create enumerators whenever and wherever appropriate. Assuming the system has the same sensors attached, the enumerators return the same number of sensor handles. However, each handle is unique so an application cannot simply compare handle values to determine if they represent the same sensor. The application must use the routine MT7TraceGetSensorID for each handle. It then compares the returned numbers to determine if the handles represent the same sensor.

Using Sensors

When an application has a sensor handle, it can use the handle to get information about the sensor. As mentioned above, the routine MT7TraceGetSensorID retrieves the sensor identification number. The MT7TraceGetSensorName retrieves the name of the sensor. This name is also useful for identifying individual sensors on a computer with multiple touch sensors.

Note that it is possible to have two sensor handles with unique ID numbers but with the same name. This could happen if an application obtained a handle to a USB sensor, a user removed the controller from the system and then reattached it, and the application obtained a new handle. In this example, the two sensor handles represent the same controller but one before the departure and one afterward.

Most applications are interested in processing and responding to touch events. The MT7Trace API collects these events in traces. A trace begins with a touchdown event, with or without a subsequent drag event, and ends with a liftoff event.

Depending on its type, the sensor may support multiple concurrent traces or it may support only a single trace at a time.
The application uses the MT7TraceGetNextTrace routine to retrieve a handle to any new trace on a sensor. Before doing so, the application must obtain updates to the sensor. Refer to the section “Polling, Monitoring, and Callbacks” for details on how to obtain these updates.

Using Traces

A trace represents a series of events produced on a sensor by a single touch. All traces begin with a single touchdown event. This occurs when a finger or stylus first makes contact with the sensor. If the finger or stylus then moves across the sensor, the MT7Trace API adds drag events to the trace. Finally, when the user removes their finger or stylus, the trace receives a liftoff event.

When an application first retrieves a trace handle, it has at least one event in it, the touchdown event. It may have other events in it, even a terminating liftoff event if the API receives the liftoff event before the application can retrieve the trace’s events.

If a trace handle does not have the liftoff event, then the application must obtain updates to the trace. Refer to the section “Polling, Monitoring, and Callbacks” for details on how to obtain updates on the state of a trace.

The simplest method to get events from a trace handle is to use the MT7TraceGetNextEvent routine. This returns an event handle. When the application first gets the trace handle or when it receives notification there are new events pending, the application should repeatedly call MT7TraceGetNextEvent and process the returned event handles until the routine indicates there are no more pending events.

An application can access events using other routines. The MT7TraceGetEvent routine allows an application to retrieve an event handle based on an index number. The MT7TraceGetPriorEvent routine allows an application to access the event relative to the last event retrieved by MT7TraceGetNextEvent. The routine MT7TraceSetNextEvent sets the index of the next event returned by MT7TraceGetNextEvent. Finally, the routine MT7TraceGetEventCount returns the number of events in the trace.

Using Events

An event corresponds to some activity on the sensor. There are three types of events.

1. A touchdown event starts a trace. It represents where a finger or stylus first makes contact with the sensor.

2. There may be any number of drag events in a trace. Each drag event represents a change in the location of a finger or stylus on the sensor while still in contact with the sensor. It is possible for a trace not to have any drag events if the finger or stylus does not move before lifting off.

3. A liftoff event completes a trace. It represents the location on the touch sensor where the finger or stylus broke contact. An application should release a trace handle when it sees a liftoff event.
The location of an event has both X and Y axes components. These are normalized to a value between 0 and 1. To convert to display coordinates, simply multiple this normalized value by one less than the display dimension along that axis. The origin of the normalized event coordinate system is in the lower left corner of the touch sensor.

Several routines retrieve properties from an event handle. An application can get the X and Y location of the event by calling `MT7TraceGetEventX` and `MT7TraceGetEventY`. The routine `MT7TraceGetEventType` returns the event type. The routine `MT7TraceGetEventTime` returns the number of milliseconds that have passed since the start of the trace.

**Polling, Monitoring, and Callbacks**

The application needs to inspect the sensors for changes in their condition. Normally this is to detect the start of a new trace or to detect a new event in a trace. The MT7Trace API supports two ways to get this information, either by polling or by using a background monitor. By polling, an application controls when to receive updates from the touch system. Using a background monitor provides real-time, asynchronous updates from the touch system.

To poll, an application periodically calls the `MT7TracePoll` routine. To start the background monitor, an application calls `MT7TraceStartMonitor`. To stop the background monitor, call the routine `MT7TraceStopMonitor`.

Both polling and the background monitor work on objects in the touch system for which the application has supplied callback routines. When there is a change for an object, the API invokes the callback associated with the object. For sensor objects, use the routine `MT7TraceSetSensorCallback` to supply a callback routine. Likewise, use the `MT7TraceSetTraceCallback` to supply a callback for a specific trace. To detect controller arrival, use `MT7TraceSetArrivalCallback`. Finally, the routine `MT7TraceSetUniversalCallback` applies the same callback to all objects.

When invoked, the callback has a reason code and an object handle. These tables summarize what reason codes the API generates for a given callback. The “new” entries show where the application gets a new handle and the “old” entries show where the application retrieves an existing handle.
There are four reason codes: new trace, new event, departure, and arrival. Controller arrival and departure are important only if the application is managing sensors. For the new trace code, the application should call `MT7TraceGetNextTrace` repeatedly to obtain the new traces. Thereafter, traces may have new event reason codes. Here, the application should call `MT7TraceGetNextEvent` to get the new touch event for the trace.

Note that the application should call `MT7TraceGetNextEvent` for each trace returned by the `MT7TraceGetNextTrace` to process the events that created the trace. This is important for traces that contain only touchdown and liftoff events. In this case, it is possible that only a new trace event is generated for the trace.

<table>
<thead>
<tr>
<th>Reason</th>
<th>Callback</th>
<th>Arrival</th>
<th>Departure</th>
<th>NewTrace</th>
<th>NewEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal</td>
<td>New sensor</td>
<td>New sensor</td>
<td>New sensor</td>
<td>New trace</td>
<td></td>
</tr>
<tr>
<td>Sensor</td>
<td>Old sensor</td>
<td>Old sensor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace</td>
<td></td>
<td></td>
<td>Old trace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arrival</td>
<td>New sensor</td>
<td>New sensor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For universal and arrival callbacks, the handle is a new sensor or trace handle. If the application decides not to use the handle outside the callback routine, it must call the `MT7TraceRelease*` routine. For sensor and trace callbacks, the handle is the same as supplied to the `MT7TraceSet*Callback` routine and the application does not need to release it unless the application decides it no longer needs the handle. Refer to the section on handles for more information.

It is possible to supply the same function as all the `MT7TraceSet*Callback` routines. However, it is advisable to use a different routine for universal callback from the one used for sensor and trace callbacks. This is an easy way for the application to know if it should release the handle or not.

Calling `MT7TraceSet*Callback` with a NULL callback routine disables the callback.

Each callback has an application-supplied value associated with it. When the API invokes a callback, it provides the value to callback. These values are for the convenience of the application. The API does not use or interpret these values.

If an application sets a universal callback and then sets a callback for a sensor or trace, the API invokes both callbacks for the same change. For example, if an application has a universal and sensor callback set, a new trace causes the API to invoke the universal callback with a new sensor handle and invoke the sensor callback with the sensor handle supplied by the application.

The background monitor executes in its own thread and any callbacks invoked by it run in the context of the monitor thread. If needed, the application is responsible for synchronizing its threads and actions taken by the callback. For polling, the callbacks run in the same thread as the routine that called the `MT7TracePoll` routine.
General Notes on Handles

The MT7Trace API returns handles for enumerators, sensors, traces, and event objects.

It is important to remember that each handle has a unique value even though two or more handles may represent the same object. This can happen by the use of an MT7TraceClone* routine. A second enumeration can produce a second set of sensor handles that represent the same sensor objects as the first enumeration. In such a case, whenever a new trace occurs, both the first set and second set of sensor handles produce handles for the trace. These trace handles produce identical streams of touch events.

It may be important for an application to determine if two handle represent the same object. An application must use the routines MT7TraceGetSensorID, MT7TraceGetEnumeratorID, MT7TraceGetTraceID, and MT7TraceGetEventID routines to determine if two handles represent the same object.

If two sections of the application use the same handle, they can influence each other. For example, assume two sections of the application try to use the same trace handle. One section of the application calls MT7TraceGetNextEvent and retrieves an event. The other section of code never sees that event when it calls MT7TraceGetNextEvent. If an application uses the same handle in different sections, it must coordinate the activities of the two sections.

A better method is to use cloning. Each section can use a clone of the original handle. The cloned handles allow independent processing of an object without worrying about affecting other sections of the application.

The Clone routines, MT7TraceCloneEnum, MT7TraceCloneSensor, MT7TraceCloneTrace, and MT7TraceCloneEvent, create a new handle from an existing handle. The cloned handle has the same state information of the original handle at the time of the cloning. For example, the first call of MT7TraceGetNextEvent for two newly cloned trace handles returns handles for the same event object.

When an application no longer needs a handle, it should release it. Failure to release handles increases the memory usage of an application and may adversely affect performance. Generally, an application should release any handle it receives from the API when the application no longer needs the handle. The only exceptions to this are the sensor and trace callbacks where the API supplies a handle that the application already holds. Refer to the section above on “Polling, Monitors, and Callback” for details.

An application can release handles in any order. It is not necessary to release all the traces generated by a sensor before releasing the sensor handle. Likewise, releasing a handle that represents an object does not affect other handles that represent the same object. This allows applications to release handles when convenient without worrying about side effects. The handle release routines are MT7TraceReleaseEnum, MT7TraceReleaseSensor, MT7TraceReleaseTrace, and MT7TraceReleaseEvent.
Enumeration API Routines

Enumeration routines include the following:

- MT7TraceCreateEnum
- MT7TraceGetEnumID
- MT7TraceCloneEnum
- MT7TraceGetNextSensor
- MT7TraceReleaseEnum

MT7TraceCreateEnum

This routine creates a sensor enumerator. This allows an application to obtain handles for all the sensors attached to the system.

Usage: MT7TraceEnum MT7TraceCreateEnum (void);

Return Values: The return value is a handle to a new enumerator. If there is a failure to create the enumerator, MT7TraceCreateEnum returns NULL instead. The application should call MT7TraceReleaseEnum when it is finished with the enumerator handle.

Notes: Use the routine MT7TraceGetNextSensor to retrieve each sensor in the enumeration.

If invoked for the first time and there are arrival or universal callbacks installed, the API will generate sensor arrival events.

MT7TraceGetEnumID

This routine returns the unique identifier of the enumeration object represented by the enumeration handle.

Usage: unsigned int MT7TraceGetEnumID (MT7TraceEnum hEnumerator);

Arguments: hEnumerator Enumeration handle created by MT7TraceCreateEnum or MT7TraceCloneEnum

Return Values: The return value is the non-zero ID number of the underlying enumeration object.

Notes: Use this routine to determine if two enumeration handles represent the same enumeration object.

MT7TraceCloneEnum

This routine returns a new handle to the enumeration object represented by the supplied handle.

Usage: MT7TraceEnum MT7TraceCloneEnum (MT7TraceEnum hEnumerator);
Arguments: hEnumerator  Enumeration handle created by MT7TraceCreateEnum or MT7TraceCloneEnum

Return Values: The return value is a new handle to the enumeration object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call MT7TraceReleaseEnum when it is finished with the new handle.

Notes: Use this routine to create a new handle to an existing enumeration object.

**MT7TraceGetNextSensor**

This routine obtains a handle to a sensor object from an enumeration handle.

Usage: MT7TraceSensor MT7TraceGetNextSensor (MT7TraceEnum hEnumerator);

Arguments: hEnumerator  Enumeration handle created by MT7TraceCreateEnum or MT7TraceCloneEnum

Return Values: The return value is a handle to a sensor object from the enumeration handle. If there is a failure or there are no more sensors in the enumeration, the return value is NULL. The application should call MT7TraceReleaseSensor when it is finished with the sensor handle.

Notes: Use this routine to obtain sensor handles from an enumeration.

**MT7TraceReleaseEnum**

This routine releases an enumeration handle.

Usage: void MT7TraceReleaseEnum (MT7TraceEnum hEnumerator);

Arguments: hEnumerator  Enumeration handle created by MT7TraceCreateEnum or MT7TraceCloneEnum

Notes: Use this routine to release an enumeration handle when the application no longer needs it. Call this routine for each enumeration handle created by MT7TraceCreateEnum or MT7TraceCloneEnum.

**Touch Sensor API Routines**

Touch Sensor API routines include the following:

- MT7TraceGetSensorID
- MT7TraceGetSensorName
- MT7TraceCloneSensor
- MT7TraceGetNextTrace
- MT7TraceSetSensorCallback
- MT7TraceReleaseSensor
**MT7TraceGetSensorID**

This routine returns the unique identifier of the sensor object represented by the sensor handle.

**Usage:**

```c
unsigned int MT7TraceGetSensorID (MT7TraceSensor hSensor);
```

**Arguments:**

- `hSensor` Sensor handle created by `MT7TraceGetNextSensor` or `MT7TraceCloneSensor` or supplied by an arrival or universal callback.

**Return Values:**

The return value is the non-zero ID number of the underlying sensor object.

**Notes:**

Use this routine to determine if two sensor handles represent the same sensor object.

**MT7TraceGetSensorName**

This routine gets the name of the touch sensor.

**Usage:**

```c
const char * MT7TraceGetSensorName (MT7TraceSensor hSensor);
```

**Arguments:**

- `hSensor` Sensor handle created by `MT7TraceGetNextSensor` or `MT7TraceCloneSensor` or supplied by an arrival or universal callback.

**Return Values:**

The return value is a pointer to a NULL-terminated ASCII string that is the name of the sensor.

**Notes:**

Use this routine to compare 2 sensor handles to determine if they represent the same sensor.

The MT 7 driver determines the sensor name. Examples of valid sensor names are PORTCOM1, SER00000001 and USB00000002

**MT7TraceCloneSensor**

This routine returns a new handle to the sensor object represented by the supplied sensor handle.

**Usage:**

```c
MT7TraceSensor MT7TraceCloneSensor (MT7TraceSensor hSensor);
```

**Arguments:**

- `hSensor` Sensor handle created by `MT7TraceGetNextSensor` or `MT7TraceCloneSensor` or supplied by an arrival or universal callback.

**Return Values:**

The return value is a new handle to the sensor object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call `MT7TraceReleaseSensor` when it is finished with the new handle.

**Notes:**

Use this routine to create a new handle to an existing sensor object.
MT7TraceGetNextTrace

This routine obtains a handle to a new trace object from a sensor handle.

Usage:

```c
MT7TraceTrace MT7TraceGetNextTrace (MT7TraceSensor hSensor);
```

Arguments:

- `hSensor`  
  Sensor handle created by MT7TraceGetNextSensor or MT7TraceCloneSensor or supplied by an arrival or universal callback.

Return Values:

The return value is a handle to a new trace object from the sensor handle. If there is a failure or there are no more traces in the sensor, the return value is NULL. The application should call MT7TraceReleaseTrace when it is finished with the trace handle.

Notes:

Use this routine to obtain new trace handles from a sensor. When the application obtains handle to a trace from the sensor handle, it is not possible to query the sensor handle again to get the same trace.

MT7TraceSetSensorCallback

This routine sets a callback routine for a sensor handle.

Usage:

```c
void MT7TraceSetSensorCallback (MT7TraceSensor hSensor, MT7TraceCallback fnCallback, void * pvUserCode);
```

Arguments:

- `hSensor`  
  Sensor handle created by MT7TraceGetNextSensor or MT7TraceCloneSensor or supplied by an arrival or universal callback.

- `fnCallback`  
  Callback function to invoke if something happens on the sensor

- `pvUserCode`  
  Application-supplied value that is passed into the callback when invoked

Notes:

Use this routine to obtain notification that something happened with the sensor. The two possibilities for a sensor callback are controller departure and the start of a new trace.

When the API invokes the callback routine, the sensor handle passed to the callback routine is `hSensor`. The API invokes the callback within an MT7TracePoll call or within the monitor thread created by the MT7TraceStartMonitor call.

MT7TraceReleaseSensor

This routine releases a sensor handle.

Usage:

```c
void MT7TraceReleaseSensor (MT7TraceSensor hSensor);
```

Arguments:

- `hSensor`  
  Sensor handle created by MT7TraceGetNextSensor, MT7TraceCloneSensor, MT7TraceGetTraceSensor or supplied by an arrival or universal callback.
Notes: Use this routine to release a sensor handle when the application no longer needs it. Call this routine for each sensor handle created by MT7TraceGetNextSensor, MT7TraceCloneSensor, or MT7TraceGetTraceSensor.

Use this routine for any sensor handle received through a callback installed with the MT7TraceSetArrivalCallback or MT7TraceSetUniversalCallback. An application can release the sensor handle within the callback routine or save it to process later. If the application saves the handle, the application must release the handle when the application no longer needs it.

Regardless of the type of callback, an application should consider releasing the sensor handle when there is an MT7TraceReason_Departure reason in the callback. This is the last callback the API invokes for the sensor or any of its traces.

Trace API Routines

The Trace API routines include the following:

- MT7TraceGetTraceID
- MT7TraceCloneTrace
- MT7TraceGetNextEvent
- MT7TraceGetPriorEvent
- MT7TraceSetNextEvent
- MT7TraceGetEventCount
- MT7TraceGetEvent
- MT7TraceGetTraceSensor
- MT7TraceSetTraceCallback
- MT7TraceReleaseTrace

MT7TraceGetTraceID

This routine gets the identification number of a trace.

Usage: unsigned int MT7TraceGetTraceID (MT7TraceTrace hTrace);

Arguments: hTrace Trace handle created by MT7TraceGetNextTrace, MT7TraceCloneTrace or supplied by an arrival or universal callback.

Return Values: The return value is a non-zero identification number of the trace represented by the handle.

Notes: Use this routine to compare two trace handles to determine if they represent the same trace.
**MT7TraceCloneTrace**

This routine returns a new handle to the trace object represented by the supplied trace handle.

**Usage:**

```c
MT7TraceTrace MT7TraceCloneTrace (MT7TraceTrace hTrace);
```

**Arguments:**

- `hTrace` Trace handle created by `MT7TraceGetNextTrace`, `MT7TraceCloneTrace` or supplied by an arrival or universal callback.

**Return Values:**

The return value is a new handle to the trace object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call `MT7TraceReleaseTrace` when it is finished with the new handle.

**Notes:**

Use this routine to create a new handle to an existing trace object.

---

**MT7TraceGetNextEvent**

This routine obtains a handle to a next event object from a trace handle.

**Usage:**

```c
MT7TraceEvent MT7TraceGetNextEvent (MT7TraceTrace hTrace);
```

**Arguments:**

- `hTrace` Trace handle created by `MT7TraceGetNextTrace`, `MT7TraceCloneTrace` or supplied by an arrival or universal callback.

**Return Values:**

The return value is a handle to a next event object from the trace handle. If there is a failure or there are no more events in the trace, the return value is NULL. The application should call `MT7TraceReleaseEvent` when it is finished with the event handle.

**Notes:**

Use this routine to obtain event handles from a sensor. Normally, the events are the newest events in the trace. However, `MT7TraceGetNextEvent` restarts at the event set by the `MT7TraceSetNextEvent` routine.

---

**MT7TraceGetPriorEvent**

The routine retrieves events prior to the last obtained event.

**Usage:**

```c
MT7TraceEvent MT7TraceGetPriorEvent (MT7TraceTrace hTrace, unsigned int nOffset);
```

**Arguments:**

- `hTrace` Trace handle created by `MT7TraceGetNextTrace`, `MT7TraceCloneTrace` or supplied by an arrival or universal callback.
- `nOffset` Offset from the last event returned by `MT7TraceGetNextEvent`
Return Values: The return value is a handle to an event object that precedes the last event returned by MT7TraceGetNextEvent. If there is a failure or there are not enough prior events in the trace, the return value is NULL. The application should call MT7TraceReleaseEvent when it is finished with the event handle.

Notes: Use this routine to obtain event handles of events immediately preceding the last event returned by MT7TraceGetNextEvent. An offset of 1 indicates the event before the last event, 2 indicates the event two back, and 0 indicates the last event.

If the offset is too large, meaning that the application is attempting to access events before the touchdown event, the routine returns a NULL.

**MT7TraceSetNextEvent**

The routine sets the ‘next’ event index.

**Usage:**

```c
void MT7TraceSetNextEvent (MT7TraceTrace hTrace, unsigned int nIndex);
```

**Arguments:**

- `hTrace` Trace handle created by MT7TraceGetNextTrace, MT7TraceCloneTrace or supplied by an arrival or universal callback.
- `nIndex` Index number of the next event to retrieve by MT7TraceGetNextEvent.

**Notes:**

Use this routine to change which event MT7TraceGetNextEvent returns. An index of 0 means that the starting touchdown event is returned next. If the index is more than the number of events in the trace, the next event returned is the next new event from the sensor.

**MT7TraceGetEventCount**

The routine gets the number of events in the trace.

**Usage:**

```c
unsigned int MT7TraceGetEventCount (MT7TraceTrace hTrace);
```

**Arguments:**

- `hTrace` Trace handle created by MT7TraceGetNextTrace, MT7TraceCloneTrace or supplied by an arrival or universal callback.

**Return Values:** The return value is the number of events in the trace.

**MT7TraceGetEvent**

This routine gets a particular event from a trace.

**Usage:**

```c
MT7TraceEvent MT7TraceGetEvent (MT7TraceTrace hTrace, unsigned int nIndex);
```
Arguments: hTrace  Trace handle created by MT7TraceGetNextTrace, MT7TraceCloneTrace or supplied by an arrival or universal callback.

nIndex  Zero-based index of the event to retrieve

Return Values: The return value is a handle to an event object in a trace indicated by the index. If there is a failure or there are not enough events in the trace, the return value is NULL. The application should call MT7TraceReleaseEvent when it is finished with the event handle.

Notes: Use this routine to obtain event handles at a particular point in the trace. An index of 0 indicates the touchdown event in the trace. If the index is too large, meaning that the application is attempting to access events that have not occurred, the routine returns a NULL.

This routine does not affect the behavior of MT7TraceGetNextEvent.

MT7TraceGetTraceSensor
This routine returns a handle to the sensor that generated the trace.

Usage: MT7TraceSensor MT7TraceGetTraceSensor (MT7TraceTrace hTrace);

Arguments: hTrace  Trace handle created by MT7TraceGetNextTrace, MT7TraceCloneTrace or supplied by an arrival or universal callback.

Return Values: The return value is a handle to a sensor object that generated the trace object. If there is a failure, the return value is NULL. The application should call MT7TraceReleaseSensor when it is finished with the sensor handle.

MT7TraceSetTraceCallback
This routine sets a callback routine for a trace handle.

Usage: void MT7TraceSetTraceCallback (MT7TraceTrace hTrace, MT7TraceCallback fnCallback, void * pvUserCode);

Arguments: hTrace  Trace handle created by MT7TraceGetNextTrace, MT7TraceCloneTrace or supplied by an arrival or universal callback.

fnCallback  Callback function to invoke if something happens with the trace.

pvUserCode  Application-supplied value that is passed into the callback when invoked.

Notes: Use this routine to obtain notification that something happened with the trace. The only possible response for a trace callback is a new event.
When the API invokes the callback routine, the sensor handle passed to the callback routine is `htrace`. The API invokes the callback within an `MT7TracePoll` call or within the monitor thread created by the `MT7TraceStartMonitor` call.

**MT7TraceReleaseTrace**

This routine releases a trace handle.

**Usage:**

```c
void MT7TraceReleaseTrace (MT7TraceTrace hTrace);
```

**Arguments:**

- `hTrace` Trace handle created by `MT7TraceGetNextTrace`, `MT7TraceCloneTrace`, `MT7TraceGetEventTrace` or supplied by universal callback.

**Notes:**

Use this routine to release a trace handle when the application no longer needs it. Call this routine for each trace handle created by `MT7TraceGetNextTrace`, `MT7TraceCloneTrace`, or `MT7TraceGetEventTrace`. Also call this routine for each trace handle supplied by the universal callback.

**Event API Routines**

The Event API routines include the following:

- `MT7TraceGetEventID`
- `MT7TraceCloneEvent`
- `MT7TraceGetEventType`
- `MT7TraceGetEventX`
- `MT7TraceGetEventY`
- `MT7TraceGetEventTime`
- `MT7TraceGetEventTrace`
- `MT7TraceReleaseEvent`

**MT7TraceGetEventID**

This routine gets the identification number of an event.

**Usage:**

```c
unsigned int MT7TraceGetEventID (MT7TraceEvent hEvent);
```

**Arguments:**

- `hEvent` Event handle created by `MT7TraceGetNextEvent`, `MT7TraceGetPriorEvent`, `MT7TraceGetEvent`, or `MT7TraceCloneEvent`

**Return Values:**

The return value is a non-zero identification number of the event represented by the handle.

**Notes:**

Use this routine to compare two event handles to determine if they represent the same event.
MT7TraceCloneEvent
This routine returns a new handle to the event object represented by the supplied event handle.

Usage:  MT7TraceEvent MT7TraceCloneEvent (MT7TraceEvent hEvent);

Arguments:  hEvent  Event handle created by MT7TraceGetNextEvent, MT7TraceGetPriorEvent, MT7TraceGetEvent, or MT7TraceCloneEvent

Return Values:  The return value is a new handle to the event object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call MT7TraceReleaseEvent when it is finished with the new handle.

Notes:  Use this routine to create a new handle to an existing event object.

MT7TraceGetEventType
This routine returns the touch event.

Usage:  MT7TraceEventType MT7TraceGetEventType (MT7TraceEvent hEvent);

Arguments:  hEvent  Event handle created by MT7TraceGetNextEvent, MT7TraceGetPriorEvent, MT7TraceGetEvent, or MT7TraceCloneEvent

Return Values:  The return value is the event, either MT7TraceEvent_Touchdown, MT7TraceEvent_Drag, or MT7TraceEvent_Liftoff.

Notes:  Use this routine to determine the type of event that occurred. A touchdown event starts a trace, a drag event indicates motion within the trace, and a liftoff event terminates a trace.

MT7TraceGetEventX
This routine returns the X coordinate of a touch event.

Usage:  float MT7TraceGetEventX (MT7TraceEvent hEvent);

Arguments:  hEvent  Event handle created by MT7TraceGetNextEvent, MT7TraceGetPriorEvent, MT7TraceGetEvent, or MT7TraceCloneEvent

Return Values:  The return value is the X coordinate of the touch event, between 0 and 1 inclusive. A value of 0 indicates the user touched the sensor at the extreme left side while a 1 indicates the user touched the extreme right side.

MT7TraceGetEventY
This routine returns the Y coordinate of a touch event.
Usage: float MT7TraceGetEventY (MT7TraceEvent hEvent);
Arguments: hEvent Event handle created by MT7TraceGetNextEvent,
MT7TraceGetPriorEvent, MT7TraceGetEvent, or
MT7TraceCloneEvent
Return Values: The return value is the Y coordinate of the touch event, between 0 and 1 inclusive. A value of 0 indicates the user touched the sensor at the extreme bottom edge while a 1 indicates the user touched the extreme top edge.

MT7TraceGetEventTime
This routine returns the timestamp of a touch event.
Usage: unsigned int MT7TraceGetEventTime (MT7TraceEvent hEvent);
Arguments: hEvent Event handle created by MT7TraceGetNextEvent,
MT7TraceGetPriorEvent, MT7TraceGetEvent, or
MT7TraceCloneEvent
Return Values: The return value is the timestamp of the touch event in milliseconds.
Notes: The timestamp is in milliseconds since system startup. It is possible that, if a system remains on long enough, that this value will wrap around. Excluding the wraparound issue, all event timestamps in a trace increase and no two events have the same timestamp. It is possible for events from different traces to have the same timestamp.

MT7TraceGetEventTrace
This routine returns the trace owning a touch event.
Usage: MT7TraceTrace MT7TraceGetEventTrace (MT7TraceEvent hEvent);
Arguments: hEvent Event handle created by MT7TraceGetNextEvent,
MT7TraceGetPriorEvent, MT7TraceGetEvent, or
MT7TraceCloneEvent
Return Values: The return value is a handle to the trace object that contains the event object represented by the event handle. If there is an error, the return value is NULL. The application should call MT7TraceReleaseTrace when it no longer requires the trace handle.
Notes: Use this routine to get the trace that owns the touch event.

MT7TraceReleaseEvent
This routine releases an event handle.
Usage: void MT7TraceReleaseEvent (MT7TraceEvent hEvent);
Arguments:  hEvent  Event handle created by MT7TraceGetNextEvent, MT7TraceGetPriorEvent, MT7TraceGetEvent, or MT7TraceCloneEvent

Notes:  Use this routine to release an event handle when the application no longer needs it. Call this routine for each event handle created by MT7TraceGetNextEvent, MT7TraceGetPriorEvent, MT7TraceGetEvent, or MT7TraceCloneTrace.

Poll and Monitor API Routines

The Poll and Monitor API routines include the following:

• MT7TracePoll
• MT7TraceStartMonitor
• MT7TraceStopMonitor
• MT7TraceSetArrivalCallback
• MT7TraceSetUniversalCallback

MT7TracePoll

This routine polls the touch system for new traces and events.

Usage:  int MT7TracePoll (void);

Return Values:  The return value is 1 if the poll was successful. A successful poll does not mean there were any changes in the touch system. A return value of 0 indicates that the application does not have any callbacks registered or that the background monitor is running.

Notes:  Use this routine to acquire new traces and events. This routine also acquires arrival and departure events.

The routine calls the appropriate callback routine to process any changes in the touch system. The callbacks occur within the context of the program thread that called the routine.

MT7TraceStartMonitor

This routine starts a background monitor on the touch system.

Usage:  int MT7TraceStartMonitor (void);

Return Values:  The return value is 1 if the monitor has started or 0 if it has not.
Notes: Use this routine to acquire new traces and events. This routine also acquires arrival and departure events.

The monitor runs in a separate thread. The monitor calls the appropriate callback routine to process any changes in the touch system. The callbacks occur within the context of the monitor thread. The application’s callback is responsible for any arbitration between the monitor thread and any application threads.

Even though this routine does not allow the application to start a monitor in the absence of any callbacks, it is possible to remove the callbacks by calling the appropriate routines with a NULL value. If this happens, the monitor continues to run even though it will not send any notifications to the application.

MT7TraceStopMonitor
This routine stops the background monitor.

Usage: void MT7TraceStopMonitor (void);

Notes: Use this routine to stop the background monitor.

MT7TraceSetArrivalCallback
This routine assigns an arrival callback.

Usage: void MT7TraceSetArrivalCallback (MT7TraceCallback fnCallback, void * pvUserCode);

Arguments:

fnCallback Application’s arrival callback or NULL to disable the arrival callback

pvUserCode Application-supplied value that is passed into the callback when invoked

Notes: The MT7TracePoll and background monitor use the callback routine to notify the application that there is a new sensor. In this case, the callback’s reason code is MT7TraceReason_Arrival and the item is a handle to the new sensor.

The MT7TracePoll and background monitor also use the callback routine to notify the application when an existing sensor leaves the system. This can happen if a user disconnects a USB controller. In this case, the callback’s reason code is MT7TraceReason_Departure and the item is a handle to the existing sensor.

The application should call MT7TraceReleaseSensor when it no longer needs the handle.

Refer to the section on callback routines for details.

MT7TraceSetUniversalCallback
This routine assigns a universal callback.
Usage: void MT7TraceSetUniversalCallback (MT7TraceCallback fnCallback, void * pvUserCode);

Arguments: fnCallback Application’s universal callback or NULL to disable the universal callback
pvUserCode Application-supplied value that is passed into the callback when invoked

Notes: The MT7TracePoll and background monitor use the callback routine to notify the application whenever something changes in the touch system. Depending on the callback’s reason code, the supplied item is either a new sensor handle or a new trace handle. The application should call MT7TraceReleaseSensor or MT7TraceReleaseTrace when it no longer needs the handle.
Refer to the section on callback routines for details.

Callback Routines

There is only one type of callback routine used by the MT7Trace API, MT7TraceCallback. Remember that MT7TraceCallback is simply a function prototype. An application can have many routines that match this prototype and pass them into the MT7TraceSet*Callback routines.

MT7TraceCallback

This routine handles a notification from the API.

Usage: void MT7TraceCallback (MT7TraceReason eReason, void * hItem, void * pvUserCode);

Arguments: eReason The reason why the API invoked the callback; either MT7TraceReason_Arrival, MT7TraceReason_Departure, MT7TraceReason_NewTrace, or MT7TraceReason_NewEvent
hItem Handle to the item associated with the notification
pvUserCode Value supplied by the application through the MT7TraceSet*Callback routine

Notes: The application has a variety of responsibilities based on the reason why the API invoked the callback and the callback invoked. Refer to the section above on “Polling, Monitoring, and Callbacks” for details.

Trace API Object Handles

An application uses handles to access the various objects in the MT7Trace API. Each handle has a unique value. Multiple handles may represent the same object.
**MT7TraceEnum**

This handle represents an enumeration object. An application uses this handle to obtain handles for sensor objects.

**MT7TraceSensor**

This handle represents a sensor. An application uses this handle to obtain trace handles for traces on the sensor.

**MT7TraceTrace**

This handle represents a trace on a sensor. An application uses this handle to obtain event handles for events within the trace.

**MT7TraceEvent**

This handle represents an event in a trace. An application uses this handle to determine the location and other information about the event.

**MT7TraceReason**

The **MT7TraceReason** is a code that notes some activity in the system, sensor, or trace.

When the application obtains a reason code, there is a related object. The callback **MT7TraceCallback** has a reason code as an argument as well as a handle to the related object. The reason code determines if the related object is a touch sensor or trace.

**MT7TraceReason_Arrival**

This reason occurs when the API discovers a new sensor. This can occur on the first invocation of **MT7TracePoll** or **MT7TraceStartMonitor**. It also occurs with the connection of a new controller, such as with the connection of a USB controller. In some special cases, this can also occur with the first invocation of **MT7TraceCreateEnum**.

This reason code pertains to callbacks set with the **MT7TraceSetArrivalCallback** and **MT7TraceSetUniversalCallback** routines. The API supplies a new sensor handle to the callback routine.

The application must call **MT7TraceReleaseSensor** when it no longer needs a handle. Unless the application uses the sensor handle outside the callback, it is safe to release the handle within the callback routine.

**MT7TraceReason_Departure**

This reason occurs when API notices the removal of an existing sensor. This can occur with the removal of a controller, such as with the disconnection of a USB controller.

This reason code pertains to callbacks set with the **MT7TraceSetSensorCallback**, **MT7TraceSetArrivalCallback**, and **MT7TraceSetUniversalCallback** routines. If the application uses a sensor callback, the application receives the same sensor handle it supplied to **MT7TraceSetSensorCallback**. If the application uses an arrival or universal callback, the sensor handle is a new handle.
The application must call `MT7TraceReleaseSensor` when it no longer needs a handle. Unless the application uses the sensor handle outside the callback, it is safe to release the handle within an arrival or universal callback. It may be useful to release the handle within a sensor callback as well since the API will not issue any more callbacks on that sensor.

**MT7TraceReason_NewTrace**

This reason occurs when a user first touches a sensor. On sensors that allow multiple touches, this occurs with each new contact with the sensor.

This `MT7TraceSetSensorCallback` and `MT7TraceSetUniversalCallback` reason code pertains to callbacks set with the routines. If the application uses a sensor callback, the application receives the same sensor handle it supplied to `MT7TraceSetSensorCallback`. If the application uses a universal callback, the sensor handle is a new handle.

To enumerate the new traces on the sensor, the application must repeatedly call `MT7TraceGetNextTrace`. For each trace the application enumerates, it should repeatedly call `MT7TraceGetNextEvent` on the trace handle to obtain the individual touch events.

The application must call `MT7TraceReleaseSensor` when it no longer needs a handle. Unless the application uses the sensor handle outside the callback, it is safe to release the handle within a callback.

The application must call also `MT7TraceReleaseTrace` and `MT7TraceReleaseEvent` when it enumerates the new trace and event handles. As with the sensor handles, it is safe to release the trace and event handles inside the callback unless the application uses them outside the callback.

**MT7TraceReason_NewEvent**

This reason occurs for existing traces as long as the user continues to touch the sensor. It also occurs when the user breaks contact and terminates the trace. Note that the API does not issue a `NewEvent` reason code for a trace if a trace contains the liftoff event when it invokes a callback with the `NewTrace` reason.

This reason code pertains to callbacks set with the `MT7TraceSetTraceCallback` and `MT7TraceSetUniversalCallback` routines. If the application uses a trace callback, the application receives the same trace handle it supplied to `MT7TraceSetTraceCallback`. If the application uses a universal callback, the trace handle is a new handle.

To enumerate the new events in the trace, the application must repeatedly call `MT7TraceGetNextEvent`.

The application must call `MT7TraceReleaseTrace` when it no longer needs a handle. Unless the application uses the trace handle outside the callback, it is safe to release the handle within a callback. It may be useful to release the handle within a trace callback as if the application receives the liftoff event since the API will not issue any more callbacks on that trace.
The application must call also `MT7TraceReleaseEvent` when it enumerates the new event handles. As with the trace handles, it is safe to release the event handles inside the callback unless the application uses them outside the callback.

**MT7TraceEventType**

Each trace starts with a single touchdown event, with or without subsequent drag events, and terminates with a single liftoff event.

**MT7TraceEvent_Touchdown**

This event starts a trace. It represents where a finger or stylus first made contact with the sensor.

**MT7TraceEvent_Drag**

Any number of drag events may occur in a trace. Each drag event represents a movement in the location of a finger or stylus while is still in contact with the sensor.

**MT7TraceEvent_Liftoff**

This event completes a trace. It represents the location on the sensor where the finger or stylus broke contact. An application should release the trace handle when it receives the liftoff event.
CHAPTER 4

Stroke API Details

Overview

The MT7Stroke API provides the means to recognize patterns of touch traces. The API refers to these patterns as reference strokes.

Defining Reference Strokes

The first thing to do is decide what strokes to recognize. These strokes are reference strokes. Each reference stroke has a shape. Other options control whether a reference stroke includes a liftoff event, if it must complete within a certain amount of time, or how well a candidate trace must match. A configuration file contains these reference strokes. Refer to the section “Configuration Files” for additional details on this file.

A trace must match the general shape of a reference stroke. This shape is a series of directions. Each direction is a compass direction, N, S, E, W, NE, NW, SE, and SW, where each direction is normally equidistant. Each direction can have a multiplier to change its distance. For example, the shape “N,E” matches a trace that moves toward the top of the display and then to the right whereas the shape “2S,W” matches a trace that moves twice as far to the bottom of the display as the next movement to the left.
The API normalizes these shapes and traces. This has two effects. First, the size of a trace is irrelevant. A shape like “N,E” matches both a small movement north then east as well as a trace that travels along the left side of the display and then across the top. Second, two apparently distinct shapes may be the same. For example, the two reference strokes “2S E .5N” and “4S 2E N” are the same.

The MT7Stroke API uses a closeness of fit algorithm to score traces against reference strokes. Even though a reference stroke may suggest a perpendicular turn, a straight trace can still match. For example, a trace that runs NNE can match the reference stroke “N,E”.

Other options control whether a trace matches a reference stroke. The option “Threshold” controls how well a trace must score against a stroke to become a match. The option “Liftoff” controls if a trace must have a liftoff event to match a reference stroke. The option “Timeout” requires that a stroke must match within a certain amount of time. Refer to the section on “Configuration Files” for details.

Note that allowing a match without a liftoff can cause matches to occur early in the trace. The developer should experiment with this to make sure the API does not match an undesired trace.

**Using Recognition Engines**

The API uses a recognition engine to load and process the configuration file. The application must load a configuration file in order to use it. The engine also accepts traces from the MT7Trace API and matches these against the reference strokes. When the engine matches a trace against a reference stroke or decides that a trace will not match a reference stroke, the engine sends a match event to the application through a callback routine.
The routine `MT7StrokeCreateEngine` creates an engine using a configuration file. An application can have many engines and each engine can have its own configuration file.

It is possible that the configuration file has an error in it. If so, `MT7StrokeCreateEngine` fails. The developer can then enable output from the API to determine how to change the configuration file. Refer to the section “Configuration Files” for details.

The `MT7StrokeCreateEngine` routine returns a handle to the new engine. If an application wants to use the same engine to control multiple features, such as different touch screens or possibly for multiple threads, the application should clone the engine handle. Use the routine `MT7StrokeCloneEngine` to get a copy of a handle.

There are routines that provide information about the engine. The routine `MT7StrokeGetEngineName` returns the name of the configuration file as defined in the file. `MT7StrokeGetEngineFile` returns the file name of the configuration file. The routine `MT7StrokeGetEngineID` returns an identification number of the engine represented by the handle.

Once created, the application needs to provide traces to the engine. The routine `MT7StrokeAddTrace` adds a single trace to the engine, the routine `MT7StrokeAddSensor` adds a single sensor, and `MT7StrokeAddAllSensors` adds all sensors to the engine. The engine then processes the provided traces, including any new traces on the provided sensors.

An application can also remove traces from the engine. The routines that do this are `MT7StrokeRemoveTrace`, `MT7StrokeRemoveSensor`, and `MT7StrokeRemoveAllSensors`.

To receive updates when an engine matches a trace to a reference stroke, the application must register a callback routine with the engine. The application can do this with either the `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine` routine. Whenever the engine invokes a callback, it supplies a new match handle.

Finally, when the application no longer needs an engine handle, it should call the routine `MT7StrokeReleaseEngine`. This stops callbacks from the engine handle and releases any resources allocated to it. An application can release engines in any order it desires.

### Using Stroke Matches

When the recognition engine invokes a callback, it provides a match handle detailing information about the match, or non-match, event. The application then reacts to the match as needed.

The routine `MT7StrokeGetMatchName` returns the name of the matched reference stroke as defined in the configuration file. The routine `MT7StrokeGetMatchID` returns the unique identification number of the match. The routine `MT7StrokeGetEngine` returns a handle to the engine that generated the match.

Each match contains one or more traces that matched, or failed to match, a reference stroke. The routine `MT7StrokeGetMatchTraceCount` returns the number of traces while `MT7StrokeGetMatchTrace` returns handles to the traces.
Under some circumstances, an application may want a copy of the match handle. Use the routine `MT7StrokeCloneMatch` to make a copy. When finished with the copy, the application should call `MT7StrokeReleaseMatch`. Do not use `MT7StrokeReleaseMatch` to release the match handle passed into the callback routine; the API automatically releases this handle.

**Polling, Monitoring, and Callbacks**

The MT7Stroke API uses the MT7Trace API to generate new traces or events. To support this, the application must either poll the MT7Trace API or start a background monitor for the MT7Trace API. By polling, an application controls when to receive updates from the touch system. Using a background monitor provides real-time, asynchronous updates from the touch system.

To poll, an application periodically calls the `MT7TracePoll` routine. To start the background monitor, an application calls `MT7TraceStartMonitor`. To stop the background monitor, call the routine `MT7TraceStopMonitor`.

A recognition engine automatically installs callbacks on the necessary elements of the MT7Trace API. This enables the MT7Trace API to acquire data from the touch system.

As touch events arrive, the engine tries to match the trace against its reference strokes. When a match occurs, or when the engine decides that a trace will not match a reference stroke, it invokes the callback with the reason and match information. The reason is either `MT7StrokeReason_NoMatch` or `MT7StrokeReason_Match`.
The background monitor executes in its own thread and any callbacks invoked by it run in the context of the monitor thread. If needed, the application is responsible for synchronizing its threads and actions taken by the callback. For polling, the callbacks run in the same thread as the routine that called the MT7TracePoll routine.

**General Notes on Handles**

The MT7Stroke API returns handles for recognition engines and stroke matches.

It is important to remember that each handle has a unique value even though two or more handles may represent the same object. This can happen by the use of a Clone routine.

It may be important for an application to determine if two handles represent the same object. To determine if two handles of the same type refer to the same object, call MT7StrokeGetEngineID or MT7StrokeGetMatchID for each handle. If the return values match, the handles represent the same object.

If two sections of the application used the same handle, they can influence each other. If an application uses the same handle in different sections, it must coordinate the activities of the two sections.

A better method is to use cloning. Each section can use a clone of the original handle. The cloned handles allow independent processing of an object without worrying about affecting other sections of the application.

The clone routines, MT7StrokeCloneEngine and MT7StrokeCloneMatch, create a new handle from an existing handle. The cloned handle has the same state information as the original handle at the time of the cloning. For example, each section of code can set its own callback for the engine.

When an application no longer needs a handle, it should release it. Failure to release handles increases the memory usage of an application and may adversely affect performance.

An application can release handles in any order. It is not necessary to release all the matches generated by an engine before releasing the engine handle. Likewise, releasing a handle that represents an object does not affect other handles that represent the same object. This allows applications to release handles when convenient without worrying about side effects. The handle release routines are MT7StrokeReleaseEngine and MT7StrokeReleaseMatch.

The match handles contain references to trace handles. Refer to the MT7Trace API for more information regarding trace handles.

**Engine API Routines**

The Engine API routines include the following:

- MT7StrokeCreateEngine
- MT7StrokeCloneEngine
- MT7StrokeGetEngineName
- MT7StrokeGetEngineFile
- MT7StrokeGetEngineID
• MT7StrokeAddAllSensors
• MT7StrokeRemoveAllSensors
• MT7StrokeAddSensor
• MT7StrokeRemoveSensor
• MT7StrokeAddTrace
• MT7StrokeRemoveTrace
• MT7StrokeRemoveAllTraces
• MT7StrokeReleaseEngine

**MT7StrokeCreateEngine**

This routine creates a new engine object.

**Usage:**

```c
MT7StrokeEngine MT7StrokeCreateEngine (const char * pszFile, MT7StrokeCallback fnCallback, void * pvUserCode);
```

**Arguments:**

- `pszFile`: Pointer to a NULL-terminated ASCII string that contains the name of an engine configuration file.
- `fnCallback`: Callback routine to invoke when the engine gets a match or NULL to disable the callback.
- `pvUserCode`: Application-supplied value that is passed into the callback when invoked.

**Return Values:**

The return value is an engine handle to the new engine object. If there is an error, the return value is NULL.

**Notes:**

Use this routine to create a new engine.

The application is responsible for releasing the handle when the application no longer needs it. Use the routine `MT7StrokeReleaseEngine` to release the handle.

If the return value is NULL, it is likely there is an error in the configuration file. Refer to the Configuration Files section for more information.

Refer to the Polling, Monitoring and Callbacks section for details on callbacks and user codes.

**MT7StrokeCloneEngine**

This routine creates a new engine handle from an existing one.

**Usage:**

```c
MT7StrokeEngine MT7StrokeCloneEngine (MT7StrokeEngine hEngine, MT7StrokeCallback fnCallback, void * pvUserCode);
```

**Arguments:**

- `hEngine`: Engine handle created by `MT7StrokeCreateEngine`, `MT7StrokeCloneEngine`, or `MT7StrokeGetMatchEngine`. 
fnCallback Callback routine to invoke when the engine gets a match or NULL to disable the callback
pvUserCode Application-supplied value that is passed into the callback when invoked

Return Values: The return value is a new engine handle to the engine object represented by the supplied engine handle. If there is an error, the return value is NULL.

Notes: Use this routine to clone an existing engine handle. This may be beneficial if different sections of code need to provide their own callback routines.
The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7StrokeReleaseEngine to release the handle.

Refer to the Polling, Monitoring and Callbacks section for details on callbacks and user codes.

MT7StrokeGetEngineName
This routine gets the internal configuration name of the engine.

Usage: const char * MT7StrokeGetEngineName (MT7StrokeEngine hEngine);

Arguments: hEngine Engine handle created by MT7StrokeCreateEngine or MT7StrokeCloneEngine

Return Values: The return value is the NULL-terminated ASCII name of the name assigned inside the file used to create the engine.

Notes: The file used to create the engine contains an optional internal name. If the file has no such name, the routine returns the configuration file name instead.

Do not use this routine to determine if two engine handles represent the same engine object. It is possible to create two distinct engines with the same configuration file. Instead, use the MT7StrokeGetEngineID routine.

MT7StrokeGetEngineFile
This routine gets the configuration file name of the engine.

Usage: const char * MT7StrokeGetEngineFile (MT7StrokeEngine hEngine);

Arguments: hEngine Engine handle created by MT7StrokeCreateEngine or MT7StrokeCloneEngine

Return Values: The return value is the NULL-terminated ASCII name of the file used to create the engine.
Notes: The name returned is identical to that supplied to the MT7StrokeCreateEngine routine when the application first created the engine.

Do not use this routine to determine if two engine handles represent the same engine object. It is possible to create two distinct engines with the same configuration file. Instead, use the MT7StrokeGetEngineID routine.

MT7StrokeGetEngineID
This routine gets the engine identifier.

Usage: unsigned int MT7StrokeGetEngineID (MT7StrokeEngine hEngine);

Arguments: hEngine Engine handle created by MT7StrokeCreateEngine or MT7StrokeCloneEngine

Return Values: The return value is the identifier of the engine object represented by the engine handle.

Notes: Use this routine to compare two engine handles to see if they represent the same engine.

MT7StrokeAddAllSensors
This routine adds all sensors to the engine.

Usage: void MT7StrokeAddAllSensors (MT7StrokeEngine hEngine);

Arguments: hEngine Engine handle created by MT7StrokeCreateEngine or MT7StrokeCloneEngine

Notes: Use this routine to add all sensors to an engine. The engine also monitors the Trace API for arrival events. This supersedes any arrival callback set by the application.

Refer to the section on “Polling, Monitoring, and Callbacks” for details about using this routine in conjunction with MT7TraceSetArrivalCallback.

MT7StrokeRemoveAllSensors
This routine removes all sensors from the engine.

Usage: void MT7StrokeRemoveAllSensors (MT7StrokeEngine hEngine);

Arguments: hEngine Engine handle created by MT7StrokeCreateEngine or MT7StrokeCloneEngine

Notes: Use this routine to purge all sensors from an engine. This effectively halts all touch processing by the engine. The engine also ignores any sensor arrival events from the Trace API.
**MT7StrokeAddSensor**

This routine adds a specific sensor to the engine.

**Usage:**

```c
void MT7StrokeAddSensor (MT7StrokeEngine hEngine, MT7TraceSensor hSensor);
```

**Arguments:**

- `hEngine` Engine handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`
- `hSensor` Handle to the sensor to be added

**Notes:**

Use this routine to add a sensor to an engine. If the engine already is managing the sensor represented by the sensor handle, the routine ignores the request.

This routine clones the supplied sensor handle. Hence, the application is still responsible for managing the supplied sensor handle and must release the handle at an appropriate time.

**MT7StrokeRemoveSensor**

This routine removes a specific sensor from the engine.

**Usage:**

```c
void MT7StrokeRemoveSensor (MT7StrokeEngine hEngine, MT7TraceSensor hSensor);
```

**Arguments:**

- `hEngine` Engine handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`
- `hSensor` Handle to the sensor to be removed

**Notes:**

Use this routine to remove a sensor from an engine. This routine uses the sensor handle to get the identifier of the sensor object. The engine then releases the sensor handle it owns that has the same identifier.

The application is still responsible for managing the supplied sensor handle. It must release the handle at an appropriate time.

**MT7StrokeAddTrace**

This routine adds a specific trace to the engine.

**Usage:**

```c
void MT7StrokeAddTrace (MT7StrokeEngine hEngine, MT7TraceTrace hTrace);
```

**Arguments:**

- `hEngine` Engine handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`
- `hTrace` Handle to the trace to be added

**Notes:**

Use this routine to add the trace represented by the trace handle to an engine. If the engine already is managing the trace represented by the trace handle, the routine ignores the request.
This routine clones the supplied trace handle. Hence, the application is still responsible for managing the supplied trace handle and must release the handle at an appropriate time.

**MT7StrokeRemoveTrace**

This routine removes a specific trace from the engine.

**Usage:**

```c
void MT7StrokeRemoveTrace (MT7StrokeEngine hEngine, MT7TraceTrace hTrace);
```

**Arguments:**

- `hEngine`  
  Engine handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`

- `hTrace`  
  Handle to the trace to be removed

**Notes:**

- Use this routine to remove the trace represented by the trace handle from an engine.
- This routine uses the trace handle to get the identifier of the trace object. It then releases the trace handle it owns that has the same identifier.
- The application is still responsible for managing the supplied trace handle. It must release the handle at an appropriate time.

**MT7StrokeRemoveAllTraces**

This routine removes all traces from the engine.

**Usage:**

```c
void MT7StrokeRemoveAllTraces (MT7StrokeEngine hEngine);
```

**Arguments:**

- `hEngine`  
  Engine handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`

**Notes:**

- Use this routine to purge all traces from an engine.

**MT7StrokeReleaseEngine**

This routine releases an engine handle.

**Usage:**

```c
void MT7StrokeReleaseEngine (MT7StrokeEngine hEngine);
```

**Arguments:**

- `hEngine`  
  Engine handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`

**Notes:**

- Use this routine to release an engine handle when the application no longer needs it. Call this routine for each match handle created by `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`.

- If the engine is processing sensors or traces, it automatically releases these handles. The engine does not invoke any callbacks to notify the application of the engine closing or its effects on its sensors or traces.
Match API Routines

The Match API routines include the following:

- **MT7StrokeCloneMatch**
- **MT7StrokeGetMatchName**
- **MT7StrokeGetMatchID**
- **MT7StrokeGetMatchEngine**
- **MT7StrokeGetMatchTraceCount**
- **MT7StrokeGetMatchTrace**
- **MT7StrokeReleaseMatch**

**MT7StrokeCloneMatch**

This routine returns a new handle to the match object represented by the supplied match handle.

**Usage:**

```
MT7StrokeMatch MT7StrokeCloneMatch (MT7StrokeMatch hMatch);
```

**Arguments:**

- `hMatch` Match handle created by `MT7StrokeCloneMatch` or passed into a `MT7StrokeCallback`

**Return Values:**

The return value is a new handle to the match object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call `MT7StrokeReleaseMatch` when it is finished with the new handle.

**Notes:**

Use this routine to create a new handle to an existing match object.

**MT7StrokeGetMatchName**

This routine returns the name of the matching reference stroke.

**Usage:**

```
const char * MT7StrokeGetMatchName (MT7StrokeMatch hMatch);
```

**Arguments:**

- `hMatch` Match handle created by `MT7StrokeCloneMatch` or passed into a `MT7StrokeCallback`

**Return Values:**

The return value is a pointer to a NULL-terminated ASCII string that is the name of the matching reference stroke. If the match handle represents a non-matching object, the return value is NULL.

**Notes:**

Use this routine to see which reference stroke, if any, corresponds to the match’s traces.

**MT7StrokeGetMatchID**

This routine returns the identification number of the match.

**Usage:**

```
unsigned int MT7StrokeGetMatchID (MT7StrokeMatch hMatch);
```
Arguments: hMatch  Match handle created by MT7StrokeCloneMatch or passed into a MT7StrokeCallback

Return Values: The return value is the identification number of the match object which the handle represents.

Notes: Use this routine to see if two match handles represent the same match.

**MT7StrokeGetMatchEngine**

This routine returns a handle to the engine that generated the match.

Usage: 

```c
MT7StrokeEngine MT7StrokeGetMatchEngine(MT7StrokeMatch hMatch);
```

Arguments: hMatch  Match handle created by MT7StrokeCloneMatch or passed into a MT7StrokeCallback

Return Values: The return value is a handle to the engine that generated the match object associated with the given handle.

Notes: The application should call the MT7StrokeReleaseEngine routine when it no longer needs the returned engine handle.

**MT7StrokeGetMatchTraceCount**

This routine returns the number of traces contained in a match handle.

Usage: 

```c
unsigned int MT7StrokeGetMatchTraceCount(MT7StrokeMatch hMatch);
```

Arguments: hMatch  Match handle created by MT7StrokeCloneMatch or passed into a MT7StrokeCallback

Return Values: The return value is the number of traces associated with a match handle. There is always at least one trace associated with a match.

**MT7StrokeGetMatchTrace**

This routine gets a trace handle associated with a match handle.

Usage: 

```c
MT7TraceTrace MT7StrokeGetMatchTrace(MT7StrokeMatch hMatch, unsigned int nIndex);
```

Arguments: hMatch  Match handle created by MT7StrokeCloneMatch or passed into a MT7StrokeCallback

nIndex  Zero-based index of the trace to retrieve

Return Values: The return value is a trace handle to the trace at the given position. If the index is greater than or equal to the number of traces indicated by the MT7StrokeGetTraceCount routine, the return value is NULL.

Notes: Use this routine to access traces associated with a match. Use the MT7TraceReleaseTrace routine to release a trace handle when the application no longer needs it.
**MT7StrokeReleaseMatch**

This routine releases a match handle.

**Usage:**

```c
void MT7StrokeReleaseMatch (MT7StrokeMatch hMatch);
```

**Arguments:**

- `hMatch` Match handle created by `MT7StrokeCloneMatch`

**Notes:**

Use this routine to release a match handle when the application no longer needs it. Call this routine for each match handle created by `MT7StrokeCloneMatch`. This routine automatically releases the trace handles associated with the match.

Do not use this routine on a handle passed into an `MT7StrokeCallback` routine. The API is responsible for any match handle passed into a callback routine.

### Callback Routines

There is only one type of callback routine used by the MT7Stroke API, `MT7StrokeCallback`. Remember that `MT7StrokeCallback` is simply a function prototype. An application can have many routines that match this prototype and pass them into the `MT7StrokeStartEngineMonitor` routine.

**MT7StrokeCallback**

The engine invokes this routine when it matches a reference stroke or determines that a trace will not match any reference strokes.

**Usage:**

```c
void MT7StrokeCallback (MT7MatchReason eReason, MT7StrokeMatch hMatch, void * pvUserCode);
```

**Arguments:**

- `eReason` Reason why the API invoked the callback
- `hMatch` New match handle
- `pvUserCode` Value supplied by the application through `MT7StrokeCreateEngine` or `MT7StrokeCloneEngine`.

**Notes:**

Refer to the Match API for details on how to handle a match. The match handle contains at least one trace handle as well as information on the type of match recognized.

The API automatically releases the match handle along with the trace handles associated with the match handle. If an application needs to process the match or any traces outside the callback, the application must clone the match handle or its traces.

Currently, there are two reason codes. However, to handle possible future enhancement, an application should not assume that there are only two reason codes. The application should avoid code that uses the ‘default’ case in a ‘switch’ statement or a terminating ‘else’ statement that is not part of an ‘else if’ block. Likewise, the callback should not assume a reason based on the return value of the `MT7StrokeGetMatchName` routine.
MT7Stroke API Object Handles

An application uses handles to access the various objects in the MT7Stroke API. Each handle has a unique value. Multiple handles may represent the same object.

MT7StrokeEngine
This handle represents a recognition engine object. An application uses this handle to process touch screens and traces to produce stroke matches.

MT7StrokeMatch
This handle represents a match made by a recognition engine. An application uses this handle to determine which stroke, if any, the recognition engine matched and the traces used to make the match.

MT7MatchReason
The engine supplies this series of codes to the engine callback routines to denote what type of match occurred.

MT7MatchReason_NoMatch
A reason code of “NoMatch” indicates that the engine decided that the traces in the accompanying match handle would never match any reference strokes. The reference stroke in the match handle is NULL.

MT7MatchReason_Match
A reason code of “Match” indicates that the engine matched the traces in the accompanying match handle to a reference stroke.

Configuration File
An MT7Stroke configuration file contains a list of reference strokes. An engine loads this file and hence has a list of strokes.

File Format
The MT7Stroke configuration file uses an XML format. All characters in the file must be single-byte ASCII characters.

‘Strokes’ Element
In general, an XML file contains a single element called the root element. The root element for an MT7Stroke configuration file has the name ‘Strokes’. This element must be present. A proper configuration file then appears like this, where the first line starts the file and the last line ends the file.

<Strokes attribute list>
  List of strokes
</Strokes>
The ‘attribute list’ is an optional list of XML attributes that modify the meaning of the file. The ‘list of strokes’ portion of the file is a series of ‘Stroke’ elements.

‘Stroke’ Element


```xml
<Stroke attribute list>
  Reference stroke definition
</Stroke>
```

The ‘attribute list’ is a list of XML attributes that modify the meaning of the file. The ‘name’ attribute is mandatory. The ‘Reference stroke definition’ portion defines the shape of the reference stroke to compare against traces.

Reference Stroke Definition

A reference stroke is a string of directions as shown below.

<table>
<thead>
<tr>
<th>Code</th>
<th>Direction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>North</td>
<td>A trace increases its Y coordinate value</td>
</tr>
<tr>
<td>NE</td>
<td>Northeast</td>
<td>A trace increases both its X and Y coordinate values</td>
</tr>
<tr>
<td>E</td>
<td>East</td>
<td>A trace increases its X coordinate value</td>
</tr>
<tr>
<td>SE</td>
<td>Southeast</td>
<td>A trace increases its X coordinate value and decreases its Y coordinate value</td>
</tr>
<tr>
<td>S</td>
<td>South</td>
<td>A trace decreases its Y coordinate value</td>
</tr>
<tr>
<td>SW</td>
<td>Southwest</td>
<td>A trace decreases both its X and Y coordinate values</td>
</tr>
<tr>
<td>W</td>
<td>West</td>
<td>A trace decreases its X coordinate value</td>
</tr>
<tr>
<td>NW</td>
<td>Northwest</td>
<td>A trace decreases its X coordinate value and increases its Y coordinate value</td>
</tr>
</tbody>
</table>

Spaces or commas separate a string of directions with multiple directions. For example, the strings “N E” and “N, E” represent a trace that first moves “north” and then “east” whereas a string “NE” represents a trace that moves directly “northeast”.

An integer multiplier can precede a direction. This increases the relative distance a trace must travel in a direction relative to the other directions in the reference stroke. A reference stroke of “N, 2E” matches a trace that moves twice as far “east” as it first moved “north”.

Spaces or commas separate a string of directions with multiple directions. For example, the strings “N E” and “N, E” represent a trace that first moves “north” and then “east” whereas a string “NE” represents a trace that moves directly “northeast”.

An integer multiplier can precede a direction. This increases the relative distance a trace must travel in a direction relative to the other directions in the reference stroke. A reference stroke of “N, 2E” matches a trace that moves twice as far “east” as it first moved “north”.
This example shows valid reference stroke definitions.

```xml
<Strokes>
  <Stroke>N E</Stroke>
  <Stroke>S,W</Stroke>
  <Stroke>2NE S</Stroke>
</Strokes>
```

### Attribute List

Attributes can modify how the engine processes the reference strokes and traces. These attributes apply to both the ‘Strokes’ root element and the ‘Stroke’ elements. Attributes in the ‘Strokes’ root element affect all other elements. Attributes in a ‘Stroke’ element affect only that reference stroke. If not set, the engine uses a default value.

The ‘Name’ attribute is mandatory for all ‘Stroke’ elements. All others attributes are optional.

The attributes may appear in any order. Each attribute consists of the name of the attribute, an equal sign (‘=’), and a value enclosed by double quotes (“ and ”). A space precedes the attribute to separate it from the element name or the previous attribute.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>None; required for all ‘Stroke’ elements</td>
<td>For the ‘Strokes’ element, this is the configuration file name returned by MT7StrokeGetEngineName. For the ‘Stroke’ element, this is the match name returned by MT7StrokeGetMatchName.</td>
</tr>
<tr>
<td>Threshold</td>
<td>Float</td>
<td>.70</td>
<td>This is the minimum score that a trace can have compared to the reference stroke before the engine declares a match. Suggested values are between .5 and .8, where .8 is the more stringent test</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>0</td>
<td>If the number of milliseconds that a trace has existed exceeds this number, the engine does not match the trace to the reference stroke. A value of 0 does not place a time limit on the trace.</td>
</tr>
<tr>
<td>Liftoff</td>
<td>0 or 1</td>
<td>1</td>
<td>If set to 1, the engine requires a trace to have a liftoff event to match to the reference stroke. If set to 0, the engine may match a trace to the reference stroke before there is a liftoff event.</td>
</tr>
<tr>
<td>Enable</td>
<td>0 or 1</td>
<td>1</td>
<td>If set to 1, the engine uses the reference stroke. If set to 0, the engine ignores the reference stroke.</td>
</tr>
</tbody>
</table>

This sample file shows valid attribute usage.

```xml
<Stroke Name="Test" Threshold=".8">
  <Stroke Name="Up & Right">N E</Stroke>
  <Stroke Name="Down & Left">S,W</Stroke>
  <Stroke Name="Saw tooth">2NE S</Stroke>
</Strokes>
```
Parser Errors

During development of a stroke program, you also need to develop the configuration file. During this process, it is possible to make coding errors in the file.

You may first notice this when `MT7StrokeCreateEngine` returns a NULL. If this happens, enable the environment variable `MT7StrokeParserErrors`. If enabled, the API writes error messages to standard output or a debug logger. You can check this data to see what errors you need to correct.

Refer to the section on using environment variables with the MT7Gesture APIs for more details.

XML Format Errors

These errors occur with the format of the configuration file, prohibiting parsing of the file. Correct the file format errors and try again.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning and Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>The parser could not determine file encoding. Make sure that the configuration file contains only ASCII characters or uses UTF-8 encoding.</td>
</tr>
<tr>
<td>Failed to open file</td>
<td>The parser could not find the specified file or did not have the proper permissions. Check the file name, its location, and make sure the program has read access to the file.</td>
</tr>
<tr>
<td>Error parsing Element</td>
<td>The parser could not find a ‘&lt;’ or ‘&gt;’ when expecting an element tag.</td>
</tr>
<tr>
<td>Failed to read Element name</td>
<td>The name of the element tag is not in proper form.</td>
</tr>
<tr>
<td>Error reading Element value</td>
<td>The parser could not find the end tag for an element.</td>
</tr>
<tr>
<td>Error reading Attributes</td>
<td>The parser could not process an attribute because the attribute name is not valid, there is no ‘=’ sign, or there is no quoted value.</td>
</tr>
<tr>
<td>Error: empty tag</td>
<td>A ‘/’ appears in the tag but does not have the expected ‘&gt;’ immediately after it.</td>
</tr>
<tr>
<td>Error reading end tag</td>
<td>The parser could not find the end tag or found a non-matching end tag.</td>
</tr>
<tr>
<td>Error parsing Unknown</td>
<td>The parser could not find the end of an unsupported tag.</td>
</tr>
<tr>
<td>Error parsing Comment</td>
<td>The parser could not find the terminating ‘---&gt;’ of a comment.</td>
</tr>
<tr>
<td>Error document empty</td>
<td>The file is either empty, contains nothing but white space, or has no root element.</td>
</tr>
<tr>
<td>Error parsing CDATA</td>
<td>The parser could not decode a CDATA section.</td>
</tr>
</tbody>
</table>
## File Content Errors

These errors deal with the content of the configuration file. These errors only appear when you address any errors with the XML file format.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal direction found in stroke “x” on line “y”</td>
<td>The stroke definition for the ‘Stroke’ element named “x” on line “y” has an invalid direction. Valid directions are ‘N’, ‘S’, ‘E’, ‘W’, ‘NE’, ‘NW’, ‘SE’, and ‘SW’.</td>
</tr>
<tr>
<td>Multiplier found without direction in stroke “x” on line “y”</td>
<td>The stroke definition for the ‘Stroke’ element named “x” on line “y” has a multiplier without a direction. For example, a stroke definition containing the string “2 3N” generates this error.</td>
</tr>
<tr>
<td>No root element found</td>
<td>The file does not contain the ‘Strokes’ root element.</td>
</tr>
<tr>
<td>No stroke defined for stroke “x” on line “y”</td>
<td>The ‘Stroke’ element named “x” on line “y” does not have a stroke definition.</td>
</tr>
<tr>
<td>Option “Enable” requires a value of 0 or 1 on line “y”</td>
<td>The “Enable” attribute on line “y” has an invalid value. The only valid values are 0 and 1.</td>
</tr>
<tr>
<td>Option “Liftoff” requires a value of 0 or 1 on line “y”</td>
<td>The “Liftoff” attribute on line “y” has an invalid value. The only valid values are 0 and 1.</td>
</tr>
<tr>
<td>Option “Name” requires at least one character on line “y”</td>
<td>The “Name” attribute on line “y” has an invalid value. The name must contain at least one character.</td>
</tr>
<tr>
<td>Option “Threshold” requires a value of at least .50 and no more than .90 on line “y”</td>
<td>The “Threshold” attribute on line “y” has an invalid value. The only valid values are numbers between .50 and .90.</td>
</tr>
<tr>
<td>Option “Timeout” requires a value of 0 or more on line “y”</td>
<td>The “Timeout” attribute on line “y” has an invalid value. The only valid values are non-negative integers.</td>
</tr>
<tr>
<td>Poorly defined stroke “x” at line “y”</td>
<td>The stroke definition for the ‘Stroke’ element named “x” on line “y” contains an error that is not a syntactical error. Redo the stroke definition or contact 3M Touch Systems for assistance.</td>
</tr>
<tr>
<td>Root element must be “Strokes”, is “x” on line “y”</td>
<td>The file contains an illegal root element, named “x”, on line “y” in the file.</td>
</tr>
<tr>
<td>Stroke does not have a name on line “y”</td>
<td>The ‘Stroke’ element on line “y” in the file does not have a ‘Name’ attribute.</td>
</tr>
<tr>
<td>Too many errors; processing stopped</td>
<td>There are too many errors and the API will stop processing any more errors.</td>
</tr>
<tr>
<td>Unexpected element “x” at line “y”</td>
<td>An element, named “x”, was found on line “y” in the file. The parser only recognizes ‘Stroke’ elements.</td>
</tr>
<tr>
<td>Unexpected node type “x” at line “y”</td>
<td>The parser only recognizes XML element and comment tags. The parser found an unsupported tag at line “y” in the file. The type number “x” only has internal significance.</td>
</tr>
<tr>
<td>Unrecognized option “x” on line “y”</td>
<td>An attributed named “x” on line “y” is not one recognized by the parser.</td>
</tr>
</tbody>
</table>
CHAPTER 5

Radial Menu Details

Overview

The MT7Radial API enables you to manage a radial menu system for an application. This allows the application to present choices to the user in a touch-friendly manner.

The MT7Radial API requires the use of the MT7Trace API. The application uses the MT7Trace API to get traces from the touch system. Hence, the application must monitor the touch system through the MT7Trace API as a prerequisite for using the MT7Radial API.

An application must test if a trace should result in a radial menu. For example, a certain area of the display may have several operations associated with it. When a trace starts in this area, the application can pop up a radial menu where the trace starts.

The MT7Radial API has four types of objects: libraries, templates, menus, and actions. A menu object represents a radial menu. This manages the display of the radial menu and processes inputs to determine the user’s choice.

When a menu object determines the user made a choice, it generates an action object and sends it to the application. The application contains details on what choice the user made along with information about the trace that triggered the selection.
The library is a collection of menu templates. The application loads a library from a file that contains a number of templates. When the application decides it needs to present a menu, it uses one of the templates to generate the menu.

**Using Libraries**

A library file contains one or more menu templates. Refer to the section on defining a library file for details about its contents.

An application must load a library file to make the templates available to the API. The routine `MT7RadialCreateLibrary` does this and returns a handle to the library object. When the application no longer needs the library, it releases the handle by calling the routine `MT7RadialReleaseLibrary`.

An application may create multiple libraries as needed.

At some point, the application decides it needs to employ a radial menu. The application needs to know which library and template it should use. The routine `MT7RadialGetTemplate` takes a library handle and template name and returns a template handle that the application can then use.

There are several support routines for libraries. The routine `MT7RadialGetLibraryName` returns the name of the library while the routine `MT7RadialGetLibraryFile` returns the name of the library file. The routine `MT7RadialGetTemplateCount` returns the number of templates in the library and the routine `MT7RadialGetTemplateByIndex` returns a template based on an index number.

These routines use library handles instead of the library object itself. Refer to the section “General Notes on Handles” for details. Two routines, `MT7RadialCloneLibrary` and `MT7RadialGetLibraryID`, support duplicating handles and identifying sibling handles.

It is possible that a library file contains a syntax error that causes the `MT7RadialCreateLibrary` routine to fail. Refer to the Library Files section for details on how to diagnose errors.

**Using Templates**

A template describes the appearance, behavior, and selectable actions of a radial menu. A template is not a radial menu. Instead, it controls how the API creates the radial menu, how the radial menu appears, and how the radial menu behaves.

A library contains one or more templates. The application gets a template from the library by using the `MT7RadialGetTemplate` or `MT7RadialGetTemplateByIndex` routines. When the application no longer needs the template, it releases the handle by calling the routine `MT7RadialReleaseTemplate`.

An application may access multiple templates as needed.

At some point, the application decides it needs to employ a radial menu. This is in response to a trace. Assuming the application already has the appropriate template, the application calls the `MT7RadialStartMenu` routine to create a menu based on the template and the trace.
There are a couple of support routines for templates. The routine `MT7RadialGetTemplateName` returns the name of the template. The routine `MT7RadialGetTemplateLibrary` returns a handle to the library that contains the template.

These routines use template handles instead of the template object itself. Refer to the section “General Notes on Handles” for details. Two routines, `MT7RadialCloneTemplate` and `MT7RadialGetTemplateID`, support duplicating handles and identifying sibling handles.

### Using Menus

A menu is an active radial menu. It takes input from the touch system and eventually finishes by either triggering an action or cancelling itself. A template defines the menu’s behavior. Refer to the section “Defining Templates” on the options for a menu.

A template creates a menu with the `MT7RadialStartMenu` routine. When the application no longer needs the menu, it releases the handle by calling the routine `MT7RadialReleaseMenu`. Normally this occurs within the menu’s callback function.

The template defines where the menu graphically appears relative to the triggering trace. This can either be at the touchdown point of the trace or the trace’s current location. The `MT7RadialStartMenu` routine may override the template’s setting.

The template also defines what input from the touch system the menu uses. Likewise, the `MT7RadialStartMenu` can override this setting. The menu may react only to the initial trace or to any input from the trace’s sensor. Optionally, the application can add and remove traces with the `MT7RadialAddTrace` and `MT7RadialRemoveTrace` routines.

At some point, the touch system input may satisfy some condition to select an action. The template defines the appropriate conditions. When the API determines that the user has selected an action, it invokes the callback routine. The callback routine provides the application with a reason code, an action handle, and a supplied user code. The reason code denotes if the user selected an action or canceled the menu.
There are a couple of support routines for menus. The routine \texttt{MT7RadialGetMenuName} returns the name of the template that generated the menu. The routine \texttt{MT7RadialGetMenuTemplate} returns a handle to the template.

These routines use menu handles instead of the menu object itself. Refer to the section “General Notes on Handles” for details. Two routines, \texttt{MT7RadialCloneMenu} and \texttt{MT7RadialGetMenuID}, support duplicating handles and identifying sibling handles.

### Using Actions

When a radial menu invokes a callback, it provides an action handle detailing information about the user’s selection. The application then reacts to the user’s input as needed.

The routine \texttt{MT7RadialGetActionName} returns the name of the selected action as defined in the configuration file. The routine \texttt{MT7RadialGetActionID} returns the unique identification number of the action. The routine \texttt{MT7RadialGetActionMenu} returns a handle for the menu that generated the action.

Each action has a handle for the trace that triggered the action. The routine \texttt{MT7RadialGetActionTrace} returns the handle to this trace.

Under some circumstances, an application may want a copy of the action handle. Use the routine \texttt{MT7RadialCloneAction} to make a copy. When finished with the copy, the application should call \texttt{MT7RadialReleaseAction}. Do not use \texttt{MT7RadialReleaseAction} to release the action handle passed into the callback routine; the API automatically releases this handle.

### Polling, Monitoring, and Callbacks

The application needs to inspect the touch system for user input. The MT7Radial API requires touch input in the form of traces. Hence, an application must call either \texttt{MT7TracePoll} or \texttt{MT7TraceStartMonitor} to obtain traces. An application can periodically call \texttt{MT7TracePoll} to poll for touch input or use \texttt{MT7TraceStartMonitor} to start a background monitor. Refer to Chapter 1 MT7Trace API for details.

When the application decides to display a menu by calling the \texttt{MT7RadialStartMenu} routine, it must provide a callback routine along with an optional user code. When the user selects a menu option, the API invokes the callback routine and passes in a reason code, an action handle, and a supplied user code. The application can inspect the reason code and action handle to determine the user’s intent.

Each menu has an application-supplied value associated with it. When the API invokes a callback routine, it provides the value to callback. These values are for the convenience of the application. The API does not use or interpret these values.

The background monitor executes in its own thread and any callbacks invoked by it run in the context of the monitor thread. If needed, the application is responsible for synchronizing its threads and actions taken by the callback. For polling, the callbacks run in the same thread as the routine that called the \texttt{MT7TracePoll} routine.
General Notes on Handles

The MT7Radial API uses handles for libraries, templates, menus, and actions.

If two sections of an application use the same handle, they can influence each other. Instead, an application should clone handles so that each independent section of code can process an object without affecting other sections of code.

The clone routines, MT7RadialCloneLibrary, MT7RadialCloneTemplate, MT7RadialCloneMenu, and MT7RadialCloneAction, create a new handle from an existing handle. The cloned handle has the same state information of the original handle at the time of the cloning.

It may be important for an application to determine if two handles represent the same object. The MT7RadialGetLibraryID, MT7RadialGetTemplateID, and MT7RadialGetMenuID return the identifier of the handle’s object. If two handles have the same identifier number, they refer to the same object.

When an application no longer needs a handle, it should release it. Failure to release handles increases the memory usage of an application and may adversely affect performance. Generally, an application should release any handle it receives from the API when the application no longer needs the handle. The two exceptions are the menu handle returned by MT7RadialStartMenu and the action handle passed into a callback routine.

An application can release handles in any order. It is not necessary to release all the templates generated by a library before releasing the library handle. Likewise, releasing a handle that represents an object does not affect other handles that represent the same object. This allows applications to release handles when convenient without worrying about side effects. The handle release routines are MT7RadialReleaseLibrary, MT7RadialReleaseTemplate, MT7RadialReleaseMenu, and MT7RadialReleaseAction.

Library API Routines

The Library API routines include the following:

- MT7RadialCreateLibrary
- MT7RadialCloneLibrary
- MT7RadialGetLibraryName
- MT7RadialGetLibraryFile
- MT7RadialGetLibraryID
- MT7RadialGetTemplate
- MT7RadialGetTemplateCount
- MT7RadialGetTemplateByIndex
- MT7RadialReleaseLibrary

MT7RadialCreateLibrary

This routine creates a new library object.
Usage: MT7RadialLibrary MT7RadialCreateLibrary (const char * pszFile);

Arguments: pszFile Pointer to a NULL-terminated ASCII string that contains the name of a library file

Return Values: The return value is a library handle to the new library object. If there is an error, the return value is NULL.

Notes: Use this routine to create a new library.

The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7RadialReleaseLibrary to release the handle.

If this routine fails there may be an error in the library file. Refer to the Library Files section for details on how to diagnose errors.

MT7RadialCloneLibrary

This routine creates a new library handle from an existing one.

Usage: MT7RadialLibrary MT7RadialCloneLibrary (MT7RadialLibrary hLibrary);

Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary

Return Values: The return value is a new library handle to the library object represented by the supplied library handle. If there is an error, the return value is NULL.

Notes: Use this routine to clone an existing library handle. This may be beneficial if different sections of code need to use the same library.

The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7RadialReleaseLibrary to release the handle.

MT7RadialGetLibraryName

This routine gets the internal name of the library.

Usage: const char * MT7RadialGetLibraryName (MT7RadialLibrary hLibrary);

Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary

Return Values: The return value is the NULL-terminated ASCII name of the name assigned inside the file used to create the library.
Notes: The file used to create the library contains an optional internal name. If the file has no such name, the routine returns the library file name instead.

Do not use this routine to determine if two library handles represent the same library object. It is possible to create two distinct libraries with the same library file. Instead, use the MT7RadialGetLibraryID routine.

MT7RadialGetLibraryFile
This routine gets the library file name of the library.
Usage: const char * MT7RadialGetLibraryFile (MT7RadialLibrary hLibrary);
Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7LibraryCloneLibrary
Return Values: The return value is the NULL-terminated ASCII name of the file used to create the library.
Notes: The name returned is identical to that supplied to the MT7RadialCreateLibrary routine when the application first created the library.
Do not use this routine to determine if two library handles represent the same library object. It is possible to create two distinct libraries with the same library file. Instead, use the MT7RadialGetLibraryID routine.

MT7RadialGetLibraryID
This routine gets the library identifier.
Usage: unsigned long MT7RadialGetLibraryID (MT7RadialLibrary hLibrary);
Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary
Return Values: The return value is the identifier of the library object represented by the library handle.
Notes: Use this routine to compare two library handles to see if they represent the same library.

MT7RadialGetTemplate
This routine obtains a handle for a template from a library handle based on the template’s name.
Usage: MT7RadialTemplate MT7RadialGetTemplate (MT7RadialLibrary hLibrary, const char * pszName);
Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary
pszName Name of the template to retrieve
Return Values: The return value is a handle to the template object within a library that matches the given name. If there is a failure or no template that matches the name, the return value is NULL. The application should call MT7RadialReleaseTemplate when it is finished with the template handle.

Notes: Use this routine to obtain a template handle from a library.

MT7RadialGetTemplateCount
The routine gets the number of templates in the library.

Usage: unsigned int MT7RadialGetTemplateCount (MT7RadialLibrary hLibrary);

Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary

Return Values: The return value is the number of templates in the library.

MT7RadialGetTemplateByIndex
This routine gets a template from a library based on an index number.

Usage: MT7RadialTemplate MT7RadialGetTemplateByIndex (MT7RadialLibrary hLibrary, unsigned int nIndex);

Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary
nIndex Zero-based index of the template to retrieve

Return Values: The return value is a handle to a template object in a library indicated by the index. If there is a failure or the index is not valid, the return value is NULL. The application should call MT7RadialReleaseTemplate when it is finished with the template handle.

Notes: Use this routine to obtain the template handles at a particular index in the library. The first template in the library has an index of 0.

MT7RadialReleaseLibrary
This routine releases a library handle.

Usage: void MT7RadialReleaseLibrary (MT7RadialLibrary hLibrary);

Arguments: hLibrary Library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary

Notes: Use this routine to release a library handle when the application no longer needs it. Call this routine for each library handle created by MT7RadialCreateLibrary or MT7RadialCloneLibrary.
Template API Routines

The Template API routines include the following:

- MT7RadialCloneTemplate
- MT7RadialGetTemplateName
- MT7RadialGetTemplateID
- MT7RadialGetTemplateLibrary
- MT7RadialStartMenu
- MT7RadialReleaseTemplate

MT7RadialCloneTemplate

This routine creates a new template handle from an existing one.

Usage: MT7RadialTemplate MT7RadialCloneTemplate (MT7RadialTemplate hTemplate);

Arguments: hTemplate Template handle created by MT7RadialGetTemplate, MT7RadialGetTemplateByIndex, or MT7RadialCloneTemplate

Return Values: The return value is a new template handle for the template object represented by the supplied template handle. If there is an error, the return value is NULL.

Notes: Use this routine to clone an existing template handle. This may be beneficial if different sections of code need to use the template. The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7RadialReleaseTemplate to release the handle.

MT7RadialGetTemplateName

This routine gets the name of the template.

Usage: const char * MT7RadialGetTemplateName (MT7RadialTemplate hTemplate);

Arguments: hTemplate Template handle created by MT7RadialGetTemplate, MT7RadialGetTemplateByIndex, or MT7RadialCloneTemplate

Return Values: The return value is the NULL-terminated ASCII name of the template.

Notes: The library file defines the name of each template. Do not use this routine to determine if two template handles represent the same template object. It is possible to have identically named templates in different library files. Instead, use the MT7RadialGetTemplateID routine.
**MT7RadialGetTemplateID**
This routine gets the template identifier number.

**Usage:**

```c
unsigned long MT7RadialGetTemplateID(MT7RadialTemplate hTemplate);
```

**Arguments:**

- `hTemplate` Template handle created by `MT7RadialGetTemplate`, `MT7RadialGetTemplateByIndex`, or `MT7RadialCloneTemplate`.

**Return Values:**

The return value is the identifier of the template object represented by the template handle.

**Notes:**
Use this routine to compare two template handles to see if they represent the same template.

**MT7RadialGetTemplateLibrary**
This routine gets a handle to the library that generated the template.

**Usage:**

```c
MT7RadialLibrary MT7RadialGetTemplateLibrary(MT7RadialTemplate hTemplate);
```

**Arguments:**

- `hTemplate` Template handle created by `MT7RadialGetTemplate`, `MT7RadialGetTemplateByIndex`, or `MT7RadialCloneTemplate`.

**Return Values:**

The return value is a handle to a library object that generated the template object. If there is a failure, the return value is NULL. The application should call `MT7RadialReleaseLibrary` when it is finished with the library handle.

**Notes:**
Use this routine if the application needs to access the library that produced the template.

**MT7RadialStartMenu**
This routine starts a menu described by the template.

**Usage:**

```c
MT7RadialMenu MT7RadialStartMenu(MT7RadialTemplate hTemplate, MT7TraceTrace hTrace, MT7RadialCenter eCenter, MT7RadialInput eInput, MT7RadialCallback fnCallback, void * pvUserCode);
```

**Arguments:**

- `hTemplate` Template handle created by `MT7RadialGetTemplate`, `MT7RadialGetTemplateByIndex`, or `MT7RadialCloneTemplate`.
- `hTrace` Handle to the trace associated with the menu.
- `eCenter` Setting that determines the center of the menu; either `MT7RadialCenter_Default`, `MT7RadialCenter_Touchdown`, or `MT7RadialCenter_Current`.
eInput Setting that determines the input for the menu; either MT7RadialInput_Default, MT7RadialInput_Trace, or MT7RadialInput_Sensor

fnCallback Callback routine to invoke when the menu finishes

pvUserCode User code to passed back through the callback routine

**Return Values:**
The return value is a handle for a menu object that generated from the template object with the given trace. If there is a failure, the return value is NULL.

**Notes:**
Use this routine to start a radial menu based on the provided trace.
The center of the menu is relative to the trace. The template normally defines if the center is at the touchdown location of the trace or at the trace’s current position. An application can override this by passing MT7RadialCenter_Touchdown or MT7RadialCenter_Current in the eCenter argument.
The input to the menu may be either the trace itself or any trace from the sensor. The template normally defines what input to use. An application can override this by passing MT7RadialInput_Trace or MT7RadialInput_Sensor in the eInput argument. An application can further control the input traces with the routines MT7RadialAddTrace and MT7RadialRemoveTrace.
The routine clones the trace handle. It is the responsibility of the application to release the supplied trace handle when the application no longer needs it.
Unlike other handles, the API manages the returned menu handle. The application should not call MT7RadialReleaseMenu for this handle. If the application needs to process the menu, for example when adding or removing traces, it must call MT7RadialCloneMenu to obtain its own handle.

**MT7RadialReleaseTemplate**
This routine releases a template handle.

**Usage:**
```c
void MT7RadialReleaseTemplate (MT7RadialTemplate hTemplate);
```

**Arguments:**
- **hTemplate** Template handle created by MT7RadialGetTemplate, MT7RadialGetTemplateByIndex, or MT7RadialCloneTemplate

**Notes:**
Use this routine to release a template handle when the application no longer needs it. Call this routine for each template handle created by MT7RadialGetTemplate, MT7RadialGetTemplateByIndex, or MT7RadialCloneTemplate.
Menu API Routines

The Menu API routines include the following:

- MT7RadialCloneMenu
- MT7RadialGetMenuID
- MT7RadialGetMenuName
- MT7RadialGetMenuX
- MT7RadialGetMenuY
- MT7RadialGetMenuTemplate
- MT7RadialAddTrace
- MT7RadialRemoveTrace
- MT7RadialReleaseMenu

MT7RadialCloneMenu

This routine creates a new menu handle from an existing one.

**Usage:**

```c
MT7RadialMenu MT7RadialCloneMenu (MT7RadialMenu hMenu);
```

**Arguments:**

- `hMenu` Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu

**Return Values:**

The return value is a new menu handle to the menu object represented by the supplied menu handle. If there is an error, the return value is NULL.

**Notes:**

Use this routine to clone an existing menu handle. This may be beneficial if different sections of code need to use the menu. The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7RadialReleaseMenu to release the handle.

MT7RadialGetMenuID

This routine gets the menu identifier number.

**Usage:**

```c
unsigned long MT7RadialGetMenuID (MT7RadialMenu hMenu);
```

**Arguments:**

- `hMenu` Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu

**Return Values:**

The return value is the identifier of the menu object represented by the menu handle.

**Notes:**

Use this routine to compare two menu handles to see if they represent the same menu.
**MT7RadialGetMenuName**

This routine gets the name of the menu.

**Usage:**
```c
const char * MT7RadialGetMenuName (MT7RadialMenu hMenu);
```

**Arguments:**
- `hMenu`: Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu

**Return Values:**
The return value is the NULL-terminated ASCII name of the menu.

**Notes:**
A menu has the same name as its template.
Do not use this routine to determine if two menu handles represent the same menu object. It is possible to have identically named templates in different library files. Instead, use the MT7RadialGetMenuID routine.

**MT7RadialGetMenuX**

This routine gets the X coordinate of the center of the menu.

**Usage:**
```c
float MT7RadialGetMenuX (MT7RadialMenu hMenu);
```

**Arguments:**
- `hMenu`: Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu

**Return Values:**
The return value is the X coordinate of the center of the menu, ranging for 0 at the left side of the sensor to 1 at the right side of the sensor. If there is a failure, the return value is -1.

**MT7RadialGetMenuY**

This routine gets the Y coordinate of the center of the menu.

**Usage:**
```c
float MT7RadialGetMenuY (MT7RadialMenu hMenu);
```

**Arguments:**
- `hMenu`: Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu

**Return Values:**
The return value is the Y coordinate of the center of the menu, ranging for 0 at the bottom of the sensor to 1 at the top of the sensor. If there is a failure, the return value is -1.

**MT7RadialGetMenuTemplate**

This routine gets a handle for the template that generated the menu.

**Usage:**
```c
MT7RadialTemplate MT7RadialGetMenuTemplate (MT7RadialMenu hMenu);
```

**Arguments:**
- `hMenu`: Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu

**Return Values:**
The return value is a handle for a template object that generated the menu object. If there is a failure, the return value is NULL. The application should call MT7RadialReleaseTemplate when it is finished with the template handle.
Notes: Use this routine if the application needs to access the template that produced the menu.

MT7RadialAddTrace
This routine adds a trace to a menu for input.

Usage: 
```c
void MT7RadialAddTrace (MT7RadialMenu hMenu, MT7TraceTrace hTrace);
```

Arguments:
- `hMenu` Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu
- `hTrace` Handle to the trace to add to the menu for input

Notes: The routine checks the trace ID number of the new trace against its current list of traces. If the trace is not in the list, the routine clones the trace handle and adds it to the list.
Since the routine clones the trace handle, it is the responsibility of the application to release the trace handle when the application no longer needs it.

MT7RadialRemoveTrace
This routine removes a trace from a menu.

Usage: 
```c
void MT7RadialRemoveTrace (MT7RadialMenu hMenu, MT7TraceTrace hTrace);
```

Arguments:
- `hMenu` Menu handle created by MT7RadialStartMenu or MT7RadialCloneMenu
- `hTrace` Handle to the trace to remove the menu

Notes: The routine checks the trace ID number of the given trace against its current list of traces. If the trace is in the list, the routine removes the trace from consideration as an input.

MT7RadialReleaseMenu
This routine releases a menu handle.

Usage: 
```c
void MT7RadialReleaseMenu (MT7RadialMenu hMenu);
```

Arguments:
- `hMenu` Menu handle created by MT7RadialCloneMenu

Notes: Use this routine to release a menu handle when the application no longer needs it. Call this routine for each menu handle created by MT7RadialCloneMenu.
Action API Routines

The Action API routines include the following:

- MT7RadialCloneAction
- MT7RadialGetActionName
- MT7RadialGetActionID
- MT7RadialGetActionMenu
- MT7RadialGetActionTrace
- MT7RadialReleaseAction

MT7RadialCloneAction

This routine returns a new handle for the action object represented by the supplied action handle.

**Usage:**

```
MT7RadialAction MT7RadialCloneAction (MT7RadialMatch hAction);
```

**Arguments:**

- `hAction` Action handle created by MT7RadialCloneAction or passed into a MT7RadialCallback routine

**Return Values:**

The return value is a new handle for the action object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call MT7RadialReleaseAction when it is finished with the new handle.

**Notes:**

Use this routine to create a new handle to an existing action object.

MT7RadialGetActionName

This routine returns the name of the triggered action.

**Usage:**

```
const char * MT7RadialGetActionName (MT7RadialAction hAction);
```

**Arguments:**

- `hAction` Action handle created by MT7RadialCloneAction or passed into a MT7RadialCallback routine

**Return Values:**

The return value is a pointer to a NULL-terminated ASCII string that is the name of the triggered action. If the action handle represents a cancelled action, the return value is NULL.

**Notes:**

Use this routine to see which action, defined in the library file, if any, corresponds to the triggered action.

MT7RadialGetActionID

This routine returns the identification number of the action.

**Usage:**

```
unsigned long MT7RadialGetActionID (MT7RadialAction hAction);
```

**Arguments:**

- `hAction` Action handle created by MT7RadialCloneAction or passed into a MT7RadialCallback routine

**Return Values:**

The return value is a new handle for the action object represented by the supplied handle. If there is a failure, the return value is NULL. The application should call MT7RadialReleaseAction when it is finished with the new handle.

**Notes:**

Use this routine to create a new handle to an existing action object.
Return Values: The return value is the identification number of the action object represented by the action handle.
Notes: Use this routine to see if two action handles represent the same action.

**MT7RadialGetActionMenu**

This routine returns a handle for the menu that generated the action.

Usage: MT7RadialMenu MT7RadialGetActionMenu (MT7RadialAction hAction);
Arguments: hAction Action handle created by MT7RadialCloneAction or passed into a MT7RadialCallback routine
Return Values: The return value is a handle for the menu that generated the action object associated with the given handle.
Notes: The application should call the MT7RadialReleaseMenu routine when it no longer needs the returned menu handle.

**MT7RadialGetActionTrace**

This routine gets a trace handle to the trace that triggered the action.

Usage: MT7TraceTrace MT7RadialGetActionTrace (MT7RadialAction hAction);
Arguments: hAction Action handle created by MT7RadialCloneAction or passed into a MT7RadialCallback routine
Return Values: The return value is a trace handle to the trace that triggered the action. If no action was triggered, the return value is NULL.
Notes: Use this routine to access the trace associated with an action. Use the MT7TraceReleaseTrace routine to release a trace handle when the application no longer needs it.

**MT7RadialReleaseAction**

This routine releases an action handle.

Usage: void MT7RadialReleaseAction (MT7RadialAction hAction);
Arguments: hAction Action handle created by MT7RadialCloneAction
Notes: Call this routine for each match handle created by MT7RadialCloneAction.
Do not use this routine on a handle passed into an MT7RadialCallback routine. The API is responsible for any action handle passed into a callback routine.
Callback Routines

There is only one type of callback routine used by the MT7Radial API, MT7RadialCallback. Remember that MT7RadialCallback is simply a function prototype. An application can have many routines that match this prototype and pass them into the MT7RadialStartMenu routine.

MT7RadialCallback

This routine handles an action notification from the API.

Usage:

```c
void MT7RadialCallback (MT7RadialReason eReason, MT7RadialAction hAction, void * pvUserCode);
```

Arguments:

- `eReason` The reason why the API invoked the callback; either MT7RadialReason_NoAction or MT7RadialReason_Action
- `hAction` Handle to the action associated with the notification
- `pvUserCode` Value supplied by the application through the MT7RadialStartMenu routine

Notes:

The application has different responsibilities based on the reason why the API invoked the callback.

- If the reason is MT7RadialReason_NoAction, the user canceled the menu. Normally, an application does not need to do anything in this case. When the callback returns, the API simply releases the menu.
- If the reason is MT7RadialReason_Action, the action handle contains information about the action triggered. The application needs to query the action handle for the user’s selection and then respond accordingly. The routine MT7RadialGetActionName returns the name of the selected action.

Note that the API releases the action handle and associated menu handle, the one returned by MT7RadialStartMenu, when the callback returns. The application does not need to call either MT7RadialReleaseMenu or MT7RadialReleaseAction for either of these handles.

MT7Radial API Object Handles

An application uses handles to access the various objects in the MT7Radial API. Each handle has a unique value. Multiple handles may represent the same object.

MT7RadialLibrary

This handle represents a menu library. An application creates this handle when it loads a menu library file. It uses this handle to obtain handles for template objects.

MT7RadialTemplate

This handle represents a menu template. A menu library contains one or more menu templates. An application uses this handle to create a menu.
MT7RadialMenu
This handle represents a graphical radial menu. A menu responds to touch system inputs. Based on this input, the menu eventually generates an action handle to notify the application of the user’s choice.

MT7RadialAction
This handle represents a user’s choice from a menu. The API generates this and sends it to the application through a callback routine.

MT7RadialReason
The MT7RadialReason is a code that notes the user’s choice from a menu. The API passes a reason code to the application, along with an action handle, through a callback.

MT7RadialReason_NoAction
This reason code happens when the user cancels a radial menu. The associated action may contain a trace handle if the user canceled the menu with some action in the touch system. The trace handle is NULL if the API canceled the menu through a timeout.

MT7TraceReason_Action
This reason code happens when the user selects an option on a radial menu. The associated action has an action name and a trace handle.

MT7RadialCenter
The MT7Radial API may position a radial menu centered on one of two points, either at the beginning of a trace or at its current location. A menu’s template provides a default position. The application can override this when it calls the MT7RadialStartMenu routine.

MT7RadialCenter_Default
The API positions the menu according to the setting of the template in the library file.

MT7RadialCenter_Touchdown
The API positions the menu at the beginning of the trace.

MT7RadialCenter_Current
The API positions the menu at the current position of the trace.

MT7RadialInput
A menu reacts to input from the touch system. This can be either the trace provided through the MT7RadialStartMenu routine or it can be any trace on the trace’s sensor. A menu’s template defines the default inputs. The application can override this when it calls the MT7RadialStartMenu routine.
**MT7RadialInput_Default**

The menu uses the input mode defined by the template in the library file.

**MT7RadialInput_Trace**

The menu uses the provided trace as its sole input. An application can later use `MT7RadialAddTrace` to add more traces as input.

**MT7RadialInput_Sensor**

The menu uses the provided trace’s sensor for input. This includes the provided trace and any new traces on the sensor. This excludes any other existing traces on the sensor.

**Defining Library Files**

An MT7Radial library file contains a list of menu templates. The application loads the file into a library object and accesses the templates through that library object.

**File Format**

The MT7Radial library file uses an XML format. All characters in the file must be single-byte ASCII characters. The file has elements, each of which may have attributes specific to the type of element. The types of element include the following:

- Menus
- Menu
- Border
- Mask
- Terminate
- Visual
- Text Color
- Font
- Activation
- Action

**Menus Elements**

In general, an XML file contains a single element called the root element. The root element for an MT7Radial library file is called ‘Menus’. This element must be present. A proper library file then appears as shown below, where the first line starts the file and the last line ends the file.

```xml
<Menus attribute list>
   List of elements
</Menus>
```

The ‘attribute list’ is an optional list of XML attributes that modify the meaning of the file. The ‘list of elements’ portion is a series of other elements such as the ‘Menu’ element.
The attributes available to this element are the same as those for the ‘Menu’ element below.

‘Menu’ Element

The menu element represents a menu template. A library file must contain at least one of these elements. Only a ‘Menus’ element can contain a ‘Menu’ element. The element appears as shown below in the library file.

```
<Menu attribute list>
  List of elements
</Menu>
```

The ‘attribute list’ is a mandatory list of XML attributes that modify the menu. The ‘list of elements’ portion is a list of other elements that comprise a menu, such as ‘Action’ elements.

The attributes for the ‘Menus’ and ‘Menu’ elements are the same. The ‘Menus’ element attributes provide the default values for all menus. The attributes of a ‘Menu’ element override these defaults. An attribute omitted from both the elements have a program default value.

The ‘Inner’ and ‘Outer’ attributes define the inside and outside diameters of the ring around the menu. The menu arranges its actions’ text and icons in this area.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>None; required for all ‘Menu’ elements</td>
<td>For the ‘Menus’ element, this is the library file name returned by MT7RadialGetLibraryName. For the ‘Menu’ element, this is the action name returned by MT7RadialGetActionName.</td>
</tr>
<tr>
<td>Inner</td>
<td>Float</td>
<td>0.08</td>
<td>This is the inner diameter of the menu. The inside border of the menu, by default, takes 8% of the smallest span of the sensor.</td>
</tr>
<tr>
<td>Outer</td>
<td>Float</td>
<td>0.10</td>
<td>This is the outer diameter of the menu. The outside border of the menu, by default, takes 10% of the smallest span of the sensor.</td>
</tr>
<tr>
<td>Enable</td>
<td>0 or 1</td>
<td>1</td>
<td>If set to 1, the library uses the menu. If set to 0, the library ignores the menu.</td>
</tr>
</tbody>
</table>

‘Bevel’ Element

The ‘Bevel’ element controls the appearance of the bevel edge of the radial buttons of the menu’s options.

The ‘Bevel’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the bevel appearance for all subsequent menus. Inside a ‘Menu’ element, it defines the bevel appearance for that menu.

If there is no border color, specified by the ‘Border’ element, there are no visible menu buttons and this element has no effect on the menu’s appearance.
If visible, there are four edges to each menu button: the inside edge, the outside edge, the clockwise or right edge, and the counterclockwise or left edge.

The bevel width is a fraction of the depth of the border between a menu’s inner and outer diameters. The width applies each edge. Since the bevel surrounds the option button, a setting of .5 bevels the entire width of the button.

The light source shades the edges. Those edges the face the light source have a ‘light’ color and those that face away from the light source have a ‘dark’ color.

You can specify each of the light and dark colors in one of two ways. First, a ‘level’ attribute takes the border color a fraction of the way to black for the dark color and to white for the light color. Alternatively, each an ‘RGB’ attribute defines a specific color.

An RGB color is a mixture of red, green, and blue color levels. The color level may range from 0 to 1 or from 0 to 255. In both cases, 0 means no contribution from the color and the high value means the color is at full intensity. If any of the levels is above 1, the library assumes that all values are in the range of 0 to 255.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>Float</td>
<td>.1</td>
<td>This is the width of each bevel edge expressed as a fraction of the border depth. A value of 0 disables the bevel edges.</td>
</tr>
<tr>
<td>LightSource</td>
<td>Float</td>
<td>.375</td>
<td>This is the direction from where the light source for the bevel edges where 0 is from the right side of the display, .25 is from the top of the display, .5 is from the left side of the display, and .75 is from the bottom of the display.</td>
</tr>
<tr>
<td>DarkLevel</td>
<td>Float</td>
<td>.3</td>
<td>This is the color, expressed as a fraction of the way to black from the border color, for the bevel edges facing away from the light source.</td>
</tr>
<tr>
<td>DarkColor</td>
<td>RGB</td>
<td>none</td>
<td>This is the RGB value of the color for the bevel edges facing away from the light source.</td>
</tr>
<tr>
<td>LightLevel</td>
<td>Float</td>
<td>.3</td>
<td>This is the color, expressed as a fraction of the way to white from the border color, for the bevel edges facing toward the light source.</td>
</tr>
<tr>
<td>LightColor</td>
<td>RGB</td>
<td>none</td>
<td>This is the RGB value of the color for the bevel edges facing toward the light source.</td>
</tr>
</tbody>
</table>

‘Border’ Element

The ‘Border’ element defines the color of the menu between its inner and outer border.

The ‘Border’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the border color for all subsequent menus. Inside a ‘Menu’ element, it defines the border color for that menu.
The border color is a mixture of red, green, and blue color levels. The color level may range from 0 to 1 or from 0 to 255. In both cases, 0 means no color and the high value means the color is at full intensity. If any of the levels is above 1, the library assumes that all values are in the range of 0 to 255.

A ‘Border’ element must have the “RGB” attribute set to be valid.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>String</td>
<td>none</td>
<td>This string contains, in order and delimited by commas, the red, green, and blue color levels. For example, a string “.25,.5,1” is a color that is at 25% red, 50% green, and 100% blue level.</td>
</tr>
</tbody>
</table>

‘Mask’ Element

The ‘Mask’ element defines the color used to provide a ‘see through’ mask. This provides the color used with icons to make portions of the icon, usually the edges, transparent.

The ‘Mask’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the mask color for all subsequent menus. Inside a ‘Menu’ element, it defines the mask color for that menu.

The mask color is a mixture of red, green, and blue color levels. The color level may range from 0 to 1 or from 0 to 255. In both cases, 0 means no color and the high value means the color is at full intensity. If any of the levels is above 1, the library assumes that all values are in the range of 0 to 255.

If there is no ‘Mask’ element in the XML file, the API uses the color 50% red, 100% green, and 100% blue.

A ‘Mask’ element must have the “RGB” attribute set to be valid.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>String</td>
<td>none</td>
<td>This string contains, in order and delimited by commas, the red, green, and blue color levels. For example, a string “.25,.5,1” is a color that is at 25% red, 50% green, and 100% blue level.</td>
</tr>
</tbody>
</table>

‘Terminate’ Element

The ‘Terminate’ element defines how the user can cancel a menu. The options are any combination of these four user actions: timeout, liftoff, touchdown inside, and touchdown outside the menu.

The ‘Terminate’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the termination criteria for all subsequent menus. Inside a ‘Menu’ element, it defines the termination criteria for that menu.

By default, the user has thirty seconds to use the menu. The timer starts when the user lifts off the sensor. Any subsequent touchdown stops the timer.
Normally disabled, the menu may also terminate upon a liftoff or subsequent touchdown. The touchdown must be either inside the menu’s inner border or outside its outer border.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>30</td>
<td>This is the number of seconds the menu waits for user input before terminating. A value of 0 means the menu does not terminate. A negative value is invalid.</td>
</tr>
<tr>
<td>Liftoff</td>
<td>0 or 1</td>
<td>0</td>
<td>If set to 1, the menu terminates when the user lifts off the sensor. If set to 0, the user may lift off the sensor without terminating the menu.</td>
</tr>
<tr>
<td>Inside</td>
<td>0 or 1</td>
<td>0</td>
<td>If set to 1, the menu terminates if the user starts a trace inside the menu’s inner border. If set to 0, the user may touch the sensor inside the menu’s inner border without terminating the menu.</td>
</tr>
<tr>
<td>Outside</td>
<td>0 or 1</td>
<td>0</td>
<td>If set to 1, the menu terminates if the user starts a trace outside the menu’s outer border. If set to 0, the user may touch the sensor outside the menu’s outer border without terminating the menu.</td>
</tr>
</tbody>
</table>

‘Visual’ Element

The ‘Visual’ element controls the appearance of the menu. This includes when the menu appears, where it appears relative to its triggering trace, where the first element appears on the menu, and if the menu arranges the action in a clockwise or counterclockwise order.

The ‘Visual’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the visual characteristics for all subsequent menus. Inside a ‘Menu’ element, it defines the visual characteristics for that menu.

The ‘Angle’ attribute defines where the center of the first action appears. Hence, the action’s arc extends equally to each side of this angle. Visually, the action’s text or icon appears at this angle.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>Integer</td>
<td>0</td>
<td>This is the number of seconds that must pass before the menu displays. This allows the user to select actions with the menu appearing.</td>
</tr>
<tr>
<td>Angle</td>
<td>Float</td>
<td>0</td>
<td>This is the starting angle of the first action. 0 represents the right side of the menu, .25 the top of the menu, .5 the left side of the menu, and .75 the bottom of the menu. A value less than 0 or greater than 1 is invalid.</td>
</tr>
<tr>
<td>Center</td>
<td>0 or 1</td>
<td>0</td>
<td>A value of 0 causes the menu to center itself at a trace’s touchdown position. A value of 1 causes the menu to center itself at the trace’s current position. See MT7RadialStartMenu and the ‘cCenter’ argument for details.</td>
</tr>
<tr>
<td>Input</td>
<td>0 or 1</td>
<td>0</td>
<td>A value of 0 causes the menu to accept input only from the trace specified in MT7RadialStartMenu or 1 if it accepts input from any new trace on the trace’s sensor. See MT7RadialStartMenu’s ‘eInput’ argument for details.</td>
</tr>
<tr>
<td>Alpha</td>
<td>Float</td>
<td>1</td>
<td>This value controls the transparency of the radial menu. The value ranges from 0, when the menu is invisible, to 1, when the menu is solid thus hiding the image beneath.</td>
</tr>
<tr>
<td>Gap</td>
<td>Float</td>
<td>0</td>
<td>This value forces a gap between sectors of the radial menu. This value has the same scale as the ‘Weight’ attributes of the ‘Action’ element and applies to each enabled action. A value 0 results in no gap.</td>
</tr>
</tbody>
</table>

**‘TextColor’ Element**

The ‘TextColor’ element defines the color of the text displayed for a menu’s actions.

The ‘TextColor’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the text color for all subsequent menus. Inside a ‘Menu’ element, it defines the text color for that menu.

The text color is a mixture of red, green, and blue color levels. The color level may range from 0 to 1 or from 0 to 255. In both cases, 0 means no color and the high value means the color is at full intensity. If any of the levels is above 1, the library assumes that all values are in the range of 0 to 255.

A ‘TextColor’ element must have the “RGB” attribute set to be valid.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>String</td>
<td>None</td>
<td>This string contains, in order and delimited by commas, the red, green, and blue color levels. For example, a string “.25,.5,1” is a color that is at 25% red, 50% green, and 100% blue level.</td>
</tr>
</tbody>
</table>
‘Font’ Element

The ‘Font’ element defines the font used to display an action’s text.

The ‘Font’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the font for all subsequent menus. Inside a ‘Menu’ element, it defines the font for that menu.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Family</td>
<td>String</td>
<td>Arial for Windows Helvetica for Linux</td>
<td>This is the font family to use. If the library cannot load the given font, it tries to use the default font family.</td>
</tr>
<tr>
<td>Size</td>
<td>Integer</td>
<td>8</td>
<td>This is the font point size. It must be a positive number.</td>
</tr>
<tr>
<td>Bold</td>
<td>0 or 1</td>
<td>0</td>
<td>If set to 0, use a regular font. If set to 1, use a bold font.</td>
</tr>
<tr>
<td>Italic</td>
<td>0 or 1</td>
<td>0</td>
<td>If set to 0, use a regular font. If set to 1, use an italic font.</td>
</tr>
</tbody>
</table>

‘Activation’ Element

The ‘Activation’ element defines how a user selects an action. A menu may use one or more activation modes.

The ‘Activation’ element is valid inside a ‘Menus’ element or a ‘Menu’ element. Inside a ‘Menus’ element, it defines the activation criteria for all subsequent menus. Inside a ‘Menu’ element, it defines the activation criteria for that menu.

There are three types of activation regions: on the border, inside the border, and outside the border. The border is between the inner and outer dimensions of the menu, even when the border is not visible.

Three user actions can trigger an action.
1. Liftoff means the user must lift their finger off the sensor in the region to trigger an action.
2. Touchdown means the user must first touch the sensor in the region to trigger an action.
3. Linger means the user holds their touch steady over an option. The user must hold their touch steady within an area defined by the attribute “LingerArea” for a period of time defined by the attribute “LingerDelay”.

Except as noted, the user action determines which action to use dependent on the action’s arc. An action’s relative weight, the order of actions, and the starting angle of the menu determine the width and location of the action’s arc. The arc radiates from the center of the menu to the edge of the sensor.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Border</td>
<td>String</td>
<td>Liftoff</td>
<td>Determines what user actions on the border trigger menu activation. This can be a comma delimited list of the conditions “Liftoff”, “Touchdown”, and “Linger”. The absence of a value disables activation in the border.</td>
</tr>
<tr>
<td>Inside</td>
<td>String</td>
<td>None</td>
<td>Determines what user actions inside the border trigger menu activation. This can be a comma delimited list of the conditions “Liftoff”, “Touchdown”, and “Linger”. The absence of a value disables activation inside the border.</td>
</tr>
<tr>
<td>Outside</td>
<td>String</td>
<td>none</td>
<td>Determines what user actions outside the border trigger menu activation. This can be a comma delimited list of the conditions “Liftoff”, “Touchdown”, and “Linger”. The absence of a value disables activation outside the border.</td>
</tr>
<tr>
<td>LingerArea</td>
<td>Float</td>
<td>.1</td>
<td>Determines the area within a user’s touch must remain steady to trigger an action by lingering. This value is in screen dimensions, where .5 is half a screen dimension.</td>
</tr>
<tr>
<td>LingerDelay</td>
<td>Integer</td>
<td>500</td>
<td>Determines how long a user’s touch must remain steady to trigger an action by lingering. The value is in milliseconds.</td>
</tr>
</tbody>
</table>

‘Action’ Element

The ‘Action’ element represents an option on the menu that the user may select. The menu shows each action arranged in a circle along its border. The action appears as either text or an icon. Each action has a name associated with it.

The ‘Action’ element may only be a part of a ‘Menu’ element. Each ‘Menu’ element must contain at least two ‘Action’ elements.

Each action has a name. The routine MT7RadialGetActionName returns this value.

Each ‘Action’ element has a weight associated with it. The action’s weight determines how large of an arc it has on the menu’s border relative to the other actions. If one action has a weight twice as large as another action, the first action’s arc is twice as wide as the second action’s arc.

The action may have only one ‘Label’ or ‘Icon’ attribute. An error occurs if both or neither are set.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>None; required for all elements</td>
<td>This is the action name returned by MT7RadialGetActionName.</td>
</tr>
<tr>
<td>Label</td>
<td>Float</td>
<td>none</td>
<td>If provided, this text appears on the menu.</td>
</tr>
<tr>
<td>Icon</td>
<td>String</td>
<td>none</td>
<td>This is the name of a PNG file. The PNG file contains the action’s icon.</td>
</tr>
<tr>
<td>Weight</td>
<td>Float</td>
<td>1</td>
<td>This is the action’s relative weight. It must be a positive number.</td>
</tr>
<tr>
<td>Enable</td>
<td>0 or 1</td>
<td>1</td>
<td>If set to 1, the menu uses the action. If set to 0, the menu ignores the action.</td>
</tr>
</tbody>
</table>

**Parser Errors**

**XML Format Errors**

These errors occur with the format of the library file, prohibiting parsing of the file. Correct the file format errors and try again.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning and Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>The parser could not determine file encoding. Make sure that the configuration file contains only ASCII characters or uses UTF-8 encoding.</td>
</tr>
<tr>
<td>Failed to open file</td>
<td>The parser could not find the specified file or did not have the proper permissions. Check the file name, its location, and make sure the program has read access to the file.</td>
</tr>
<tr>
<td>Error parsing element</td>
<td>The parser could not find a ‘&lt;’ or ‘&gt;’ when expecting an element tag.</td>
</tr>
<tr>
<td>Failed to read element name</td>
<td>The name of the element tag is not in a proper form.</td>
</tr>
<tr>
<td>Error reading element value</td>
<td>The parser could not find the end tag for an element.</td>
</tr>
<tr>
<td>Error reading attributes</td>
<td>The parser could not process an attribute because the attribute name is not valid, there is no ‘=’ sign, or there is no quoted value.</td>
</tr>
<tr>
<td>Error: empty tag</td>
<td>A ‘/’ appears in the tag but does not have the expected ‘&gt;’ immediately after it.</td>
</tr>
<tr>
<td>Error reading end tag</td>
<td>The parser could not find the end tag or found a non-matching end tag.</td>
</tr>
<tr>
<td>Error parsing unknown</td>
<td>The parser could not find the end of an unsupported tag.</td>
</tr>
<tr>
<td>Error parsing comment</td>
<td>The parser could not find the terminating ‘--&gt;' of a comment.</td>
</tr>
<tr>
<td>Error parsing declaration</td>
<td>The declaration does not start with ‘&lt;?xml’</td>
</tr>
</tbody>
</table>
### Message | Meaning and Resolution
--- | ---
Error document empty | The file is either empty, contains nothing but white space, or has no root element.
Error parsing CDATA | The parser could not decode a CDATA section.

**File Content Errors**

These errors deal with the content of the library file. These errors only appear when you address any errors with the XML file format.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning and Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option “z” requires a 0 or 1 value on line “y”</td>
<td>The attribute “z” must have a value of 0 or 1 on line “y” in the file.</td>
</tr>
<tr>
<td>“x” element missing a “Name” attribute at line “y”</td>
<td>The element “x” requires a “Name” attribute at line “y” in the file.</td>
</tr>
<tr>
<td>Element “x” must have a color attribute on line “y”</td>
<td>The color element “x” at line “y” in the file has no valid color attributes.</td>
</tr>
<tr>
<td>Inner border (w) must be at least 0 and less than the outer border (z) for element “x” at line “y”</td>
<td>The “Inner” border has a value of “w” that must be between 0 and the value “z”. The attribute appears with the element named “x” at line “y” in the file.</td>
</tr>
<tr>
<td>Invalid RGB attribute value on line “y”</td>
<td>The “RGB” attribute at line “y” in the file does not have the form “red,green,blue”.</td>
</tr>
<tr>
<td>Invalid value for “z” for element “x” on line “y”</td>
<td>The color component “z” (red, green, or blue) of an RGB value has an invalid value, less than 0 or greater than 255, for element “x” at line “y” in the file.</td>
</tr>
<tr>
<td>Must have two or more “Action” elements for element “Menu” on line “y”</td>
<td>The “Menu” element at line “y” in the file requires at least two “Action” elements.</td>
</tr>
<tr>
<td>No root element found</td>
<td>The file does not contain the ‘Menus’ root element.</td>
</tr>
<tr>
<td>Outer border (z) must be between 0 and 1 element “x” at line “y”</td>
<td>The “Outer” border has a value of “z” that must be between 0 and 1. The attribute appears with the element named “x” at line “y” in the file.</td>
</tr>
<tr>
<td>Root element must be “Menus”, is “x” on line “y”</td>
<td>The file contains an illegal root element, named “x”, on line “y” in the file.</td>
</tr>
<tr>
<td>“Timeout” must be 0 or positive for element “x” on line “y”</td>
<td>The attribute “Timeout” must have a non-negative value for the element “x” at line “y” in the file.</td>
</tr>
<tr>
<td>Unexpected element “x” at line “y”</td>
<td>An element, named “x”, was found on line “y” in the file and the parser does not recognize the element.</td>
</tr>
<tr>
<td>Unexpected node type “x” at line “y”</td>
<td>The parser only recognizes XML element and comment tags. The parser found an unsupported tag at line “y” in the file. The type number “x” only has internal significance.</td>
</tr>
<tr>
<td>Message</td>
<td>Meaning and Resolution</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>“Name” must be set for element “x” on line “y”</td>
<td>The element “x” at line “y” requires a “Name” attribute but one was not found.</td>
</tr>
<tr>
<td>Either “Label” or “Icon” must be set for element “Action” on line “y”</td>
<td>The “Action” element on line “y” has neither a “Label” nor “Icon” attribute and one is required.</td>
</tr>
<tr>
<td>Only one “Label” or “Icon” must be set for element “Action” on line “y”</td>
<td>The “Action” element on line “y” has both a “Label” and “Icon” attribute and only one is required.</td>
</tr>
<tr>
<td>“Weight” must be greater than 0 for element “x” on line “y”</td>
<td>The “Weight” attribute has a 0 or negative value for the element “x” on line “y”.</td>
</tr>
<tr>
<td>“Cannot find icon file “z” for the “x” element on line “y”</td>
<td>The icon (PNG) file “z” could not be found or opened for the element “x” on line “y”.</td>
</tr>
<tr>
<td>Option “z” requires a floating point value on line “y”</td>
<td>The attribute “z” on line “y” must be a floating point value.</td>
</tr>
<tr>
<td>Option “z” requires an integer value on line “y”</td>
<td>The attribute “z” on line “y” must be an integer value.</td>
</tr>
<tr>
<td>Unknown function</td>
<td>Internal error</td>
</tr>
<tr>
<td>Unknown attribute “z” found on line “y”</td>
<td>The attribute “z” on line “y” is not valid for the element.</td>
</tr>
<tr>
<td>No activation criteria defined for element “x” on line “y”</td>
<td>There is no defined user activation rendering the menu incapable of reacting to user input.</td>
</tr>
<tr>
<td>“Delay” must be less than “Terminate” (z) for element “x” on line y</td>
<td>The “Delay” attribute must be less than the given “Terminate” value which defaults to 30 milliseconds.</td>
</tr>
<tr>
<td>“LingerArea” (z) must between 0 and 1 for element “x” on line y</td>
<td>The attribute “LingerArea” value “z” is not between 0 and 1.</td>
</tr>
<tr>
<td>“LingerDelay” must be 100 or greater for element “x” on line y</td>
<td>The provided linger delay is less than 100 milliseconds which is the smallest allowable delay.</td>
</tr>
</tbody>
</table>
CHAPTER 6
Box Tool API Details

Overview

The MT7BoxTool API provides the means to measure the changes in a trace relative to either a fixed point or another trace. This allows the application to perform actions based on the geometry of the traces and a fixed point.

This API requires the use of the MT7Trace API. An application uses the MT7Trace API to get traces from the touch system. Hence, the application must monitor the touch system through the MT7Trace API as a prerequisite for using the MT7BoxTool API. It is possible to use the MT7Trace API indirectly through other APIs such as MT7Stroke and MT7Radial.

The application is responsible for determining if a trace or pair of traces warrant the use of a box tool. For instance, it may be convenient to use a box tool in certain areas of a display and not in others. When appropriate, the application creates a box tool to interpret touch input.

A box tool recognizes two sets of points, the two initial points and the two current points. The two initial points may be from fixed coordinates or from fixed events in one or two traces. The current points consist of either the current locations of two traces or the current location of a trace and the location of one of the initial points.

The distance and angle of the two initial points establishes the initial geometry of the box tool. The two current points likewise establishes the current geometry of the box tool. Since traces determine the current points, the current geometry may change throughout the lifetime of the box tool. The box tool tracks its current geometry and allows an application to determine easily the changes between the initial and current geometries.

Using Box Tools

An application uses the routines MT7BoxToolCreateTool, MT7BoxToolCreateFixedTool, and MT7BoxToolCreateFixed2Tool to create a box tool. These routines return a handle to the box tool. When the application no longer needs the tool, it releases the handle by calling the routine MT7BoxToolReleaseTool. This usually occurs in the tool’s callback routine.

An application may create multiple box tools as needed.
The box tool must determine the initial location of its points. The routine MT7BoxToolCreateTool uses either the touchdown event or current event of two traces for the tool’s initial points. The routine MT7BoxToolCreateFixedTool uses a coordinate pair and a trace for its initial points. The routine MT7BoxToolCreateFixed2Tool uses two coordinate pairs for its initial points.

All routines use at least one trace to track for their current points. The routines MT7BoxToolCreateTool and MT7BoxToolCreateFixed2Tool allow the option of using a second trace. If there is no second trace, one of the initial points becomes a fixed current point.

At some point, there may be more events on the box tool’s traces. When this occurs, the API invokes the callback routine. The callback routine provides the application with a reason code, the tool’s handle, and an application-supplied code. The reason code denotes if the box tool has a change or if the user terminated the box tool. The user can terminate the box tool by lifting off the active traces.

The application can obtain information about the box tool and perform actions accordingly. The routines MT7BoxToolGetInitialX, MT7BoxToolGetInitialY, MT7BoxToolGetCurrentX, and MT7BoxToolGetCurrentY return the appropriate coordinates of the tool’s traces. The initial values never change for a box tool. The current values can change.

The API can also determine the distance between the two positions as well as the angle of a line between the two positions relative to the x-axis. The routines MT7BoxToolGetInitialDistance, MT7BoxToolGetInitialAngle, MT7BoxToolGetCurrentDistance, and MT7BoxToolGetCurrentAngle get these values for initial and current positions.

The routines MT7BoxToolGetDistanceChange, MT7BoxToolGetDistanceRatio, and MT7BoxToolGetAngleChange measure changes between the initial and current geometries. The routine can return both the absolute and proportional change in distance. The change in angle is the difference between the initial and current angles.

These routines use handles instead of the box tool object itself. Refer to the section “General Notes on Handles” for details. The routines MT7BoxToolCloneTool and MT7BoxToolCloneToolWithCallback support duplicating handles. The routine MT7BoxToolGetToolID assists with identifying sibling handles.

Polling, Monitoring, and Callbacks

The application needs to inspect the touch system for user input. The MT7BoxTool API requires the user touch input in the form of traces. Hence, an application must call either MT7TracePoll or MT7TraceStartMonitor to obtain traces. An application can periodically call MT7TracePoll to poll for touch input or use MT7TraceStartMonitor to start a background monitor. Review the MT7Trace API for details.
When the application decides to employ a box tool by calling the MT7BoxToolCreateTool, MT7BoxToolCreateFixedTool, or MT7BoxToolCreateFixed2Tool routines, it must provide a callback routine along with an optional user code. When the user continues the traces, the API invokes the callback routine and passes in a reason code, the tool's handle, and the supplied user value. The application can inspect the reason code and tool handle to determine the user's action.

Each tool has an application-supplied value associated with it. When the API invokes a callback routine, it provides the value to callback. These values are for the convenience of the application. The API does not use or interpret these values.

The background monitor executes in its own thread and any callbacks invoked by it run in the context of the monitor thread. If needed, the application is responsible for synchronizing its threads and actions taken by the callback. For polling, the callbacks run in the same thread as the routine that called the MT7TracePoll routine.

### General Notes on Handles

The MT7BoxTool API uses handles for the box tools.

It is important to remember that each handle has a unique value even though two or more handles may represent the same object. This can happen by the use of the MT7BoxToolCloneTool and MT7BoxToolCloneToolwithCallback routines.

It may be important for an application to determine if two handles represent the same object. Use the routine MT7BoxToolGetToolID to determine if two tool handles represent the same object.

If two sections of the application used the same handle, they can influence each other. Instead each additional section of code should use a clone of the original handle. The cloned handles allow independent processing of an object without worrying about affecting other sections of the application.

The MT7BoxToolCloneTool and MT7BoxToolCloneToolwithCallback routines create a new handle from an existing handle. The cloned handle has the same state information as the original handle at the time of the cloning.

When an application no longer needs a handle, it should release it. Failure to release handles increases the memory usage of an application and may adversely affect performance. Generally, an application should release any handle it receives from the API when the application no longer needs the handle.

An application can release handles in any order. It is permissible to release a trace handle using the MT7Trace API after creating a box tool with it. The box tool has its own copy of the trace. This allows applications to release handles when convenient without worrying about side effects. The handle release routine for box tools is MT7BoxToolReleaseTool.
Box Tool API Routines

Box Tool API routines include the following:

- MT7BoxToolCreateFixedTool
- MT7BoxToolCreateFixed2Tool
- MT7BoxToolCreateTool
- MT7BoxToolCloneTool
- MT7BoxToolCloneToolwithCallback
- MT7BoxToolGetToolID
- MT7BoxToolGetInitialX
- MT7BoxToolGetInitialY
- MT7BoxToolGetCurrentX
- MT7BoxToolGetCurrentY
- MT7BoxToolGetInitialDistance
- MT7BoxToolGetInitialAngle
- MT7BoxToolGetCurrentDistance
- MT7BoxToolGetCurrentAngle
- MT7BoxToolGetDistanceChange
- MT7BoxToolGetDistanceRatio
- MT7BoxToolGetAngleChange
- MT7BoxToolReleaseTool

MT7BoxToolCreateFixedTool

This routine creates a new box tool using a fixed point and a trace.

**Usage:**

```
MT7BoxTool MT7BoxToolCreateFixedTool (float fX, float fY, MT7TraceTrace hTrace, MT7BoxToolCenter ePosition, MT7BoxToolCallback fnCallback, void * pvUserCode);
```

**Arguments:**

- `fX` X coordinate of the fixed point, ranging from 0, the left side of the sensor, to 1, the right side of the sensor
- `fY` Y coordinate of the fixed point, ranging from 0, the bottom side of the sensor, to 1, the top side of the sensor
- `hTrace` Handle to the trace to track as the second point
ePosition Trace point to use as the initial second point, either
MT7BoxToolCenter_Touchdown to use the trace’s
touchdown location or
MT7BoxToolCenter_Current to use the trace’s
current location

fnCallback Callback routine to invoke when events occur in the
box tool

pvUserCode User code passed back through the callback routine

Return Values: The return value is a tool handle for the new box tool object. If there is
an error, the return value is NULL.

Notes: Use this routine to create a new box tool.

The application is responsible for releasing the handle when the
application no longer needs it. Use the routine
MT7BoxToolReleaseTool to release the handle.

MT7BoxToolCreateFixed2Tool

This routine creates a box tool from two fixed initial points and tracks two traces.

Usage: MT7BoxTool MT7BoxToolCreateFixed2Tool (float fX1,
float fY1, float fX2, float fY2, MT7TraceTrace hTrace1, MT7TraceTrace hTrace2, MT7BoxToolCallback fnCallback, void * pvUserCode);

Arguments: fX1 X coordinate of the first initial point
fY1 Y coordinate of the first initial point
fX2 X coordinate of the second initial point
fY2 Y coordinate of the second initial point
hTrace1 Handle to the trace that represents the first current
point or NULL if to use the first initial point for the
first current point
hTrace2 Handle to the trace that represents the second current
point or NULL if to use the second initial point for the
second current point
fnCallback Callback routine to invoke when the tool’s state
changes
pvUserCode User code passed back through the callback routine

Return Values: The return value is a tool handle to the new box tool object. If there is
an error, the return value is NULL.

Notes: Use this routine to create a new box tool.

At least one of the arguments hTrace1 and hTrace2 must be a
valid trace handle. If both are NULL, the return value is NULL
indicating an error.
The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7BoxToolReleaseTool to release the handle.

**MT7BoxToolCreateTool**

This routine creates a new box tool using two traces.

**Usage:**
```
MT7BoxTool MT7BoxToolCreateTool (MT7TraceTrace hTrace1, MT7BoxToolCenter ePosition1, MT7BoxToolInput eInput1, MT7TraceTrace hTrace2, MT7BoxToolCenter ePosition2, MT7BoxToolCallback fnCallback, void * pvUserCode);
```

**Arguments:**
- `hTrace1`: Handle to the trace that represents the first point
- `ePosition1`: Trace point to use as the initial first point, either MT7BoxToolCenter_Touchdown to use the first trace’s touchdown location or MT7BoxToolCenter_Current to use the first trace’s current location
- `eInput1`: Flag to determine if the box tool should monitor the first trace; either MT7BoxToolInput_None to not process any changes in the first trace or MT7BoxToolInput_Trace to process changes in the first trace
- `hTrace2`: Handle to the trace to track as the second point
- `ePosition2`: Trace point to use as the initial second point, either MT7BoxToolCenter_Touchdown to use the second trace’s touchdown location or MT7BoxToolCenter_Current to use the second trace’s current location
- `fnCallback`: Callback routine to invoke when events occur in the box tool
- `pvUserCode`: User code passed back through the callback routine

**Return Values:**
The return value is a tool handle to the new box tool object. If there is an error, the return value is NULL.

**Notes:**
Use this routine to create a new box tool.

The application is responsible for releasing the handle when the application no longer needs it. Use the routine MT7BoxToolReleaseTool to release the handle.

**MT7BoxToolCloneTool**

This routine creates a new tool handle from an existing one.

**Usage:**
```
MT7BoxTool MT7BoxToolCloneTool (MT7BoxTool hTool);
```
Arguments:  

- **hTool**: Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`

Return Values: The return value is a new handle to the tool object represented by the supplied tool handle. If there is an error, the return value is NULL.

Notes: Use this routine to clone an existing tool handle. This may be beneficial if different sections of code need to use the same tool. The new handle uses the same callback and user code as the supplied tool handle.

**MT7BoxToolCloneToolWithCallback**

This routine creates a new tool handle from an existing one.

Usage:

```
MT7BoxTool MT7BoxToolCloneToolWithCallback
    (MT7BoxTool hTool, MT7BoxToolCallback fnCallback,
     void * pvUserCode);
```

Arguments:

- **hTool**: Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`
- **fnCallback**: Callback routine to invoke when events occur in the box tool
- **pvUserCode**: User code passed back through the callback routine.

Return Values: The return value is a new handle to the tool object represented by the supplied tool handle. If there is an error, the return value is NULL.

Notes: Use this routine to clone an existing tool handle. This may be beneficial if different sections of code need to use the same tool and a different callback routine or user code is needed. The application is responsible for releasing the handle when the application no longer needs it. Use the routine `MT7BoxToolReleaseTool` to release the handle.

**MT7BoxToolGetToolID**

This routine gets the tool identifier.

Usage:

```
unsigned long MT7BoxToolGetToolID (MT7BoxTool hTool);
```

Arguments:

- **hTool**: Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`

Return Values: The return value is the identifier of the object represented by the tool handle.

Notes: Use this routine to compare two tool handles to see if they represent the same tool.
**MT7BoxToolGetInitialX**

This routine retrieves the initial X coordinate of one of the two points of the box tool.

**Usage:**

```c
float MT7BoxToolGetInitialX (MT7BoxTool hTool, unsigned int nPoint);
```

**Arguments:**

- `hTool` Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`
- `nPoint` Point number, 0 for the first point and 1 for the second point

**Return Values:**

The return value is the X coordinate of one of the two initial points. This value ranges between 0 and 1

**Notes:** These values never change

**MT7BoxToolGetInitialY**

This routine retrieves the initial Y coordinate of one of the two points of the box tool.

**Usage:**

```c
float MT7BoxToolGetInitialY (MT7BoxTool hTool, unsigned int nPoint);
```

**Arguments:**

- `hTool` Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`
- `nPoint` Point number, 0 for the first point and 1 for the second point

**Return Values:**

The return value is the Y coordinate of one of the two initial points. This value ranges between 0 and 1

**Notes:** These values never change

**MT7BoxToolGetCurrentX**

This routine retrieves the current X coordinate of one of the two points of the box tool.

**Usage:**

```c
float MT7BoxToolGetCurrentX (MT7BoxTool hTool, unsigned int nPoint);
```

**Arguments:**

- `hTool` Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`
- `nPoint` Point number, 0 for the first point and 1 for the second point

**Return Values:**

The return value is the X coordinate of one of the two current points. This value ranges between 0 and 1

**Notes:** These values change as the traces associated with the points gain events. If the first point is fixed or the application ignores changes to the first trace, the current coordinates of the first point are the same as the initial coordinates of the first point.

**MT7BoxToolGetCurrentY**

This routine retrieves the current Y coordinate of one of the two points of the box tool.
Usage: float MT7BoxToolGetCurrentY (MT7BoxTool hTool, unsigned int nPoint);
Arguments: hTool Tool handle created by MT7BoxToolCreateTool, MT7BoxToolCreateFixedTool, or MT7BoxToolCloneTool
nPoint Point number, 0 for the first point and 1 for the second point
Return Values: The return value is the Y coordinate of one of the two current points.
Notes: These values change as the traces associated with the points gain events. If the first point is fixed or the application ignores changes to the first trace, the current coordinates of the first point are the same as the initial coordinates of the first point.

MT7BoxToolGetInitialDistance
This routine retrieves the initial distance of the two points of the box tool.
Usage: float MT7BoxToolGetInitialDistance (MT7BoxTool hTool);
Arguments: hTool Tool handle created by MT7BoxToolCreateTool, MT7BoxToolCreateFixedTool, or MT7BoxToolCloneTool
Return Values: The return value is the distance between the initial points of the box tool. The value ranges between 0 and the square root of 2.
Notes: This value never changes.

MT7BoxToolGetInitialAngle
This routine retrieves the initial angle of the two points of the box tool.
Usage: float MT7BoxToolGetInitialAngle (MT7BoxTool hTool);
Arguments: hTool Tool handle created by MT7BoxToolCreateTool, MT7BoxToolCreateFixedTool, or MT7BoxToolCloneTool
Return Values: The return value is the angle of the line through the initial points of the box tool. This angle is relative to the X axis. The value of the angle ranges from 0 to 1. This table shows how to interpret these values.

<table>
<thead>
<tr>
<th>Angle</th>
<th>X coordinate</th>
<th>Y coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>.00</td>
<td>Second point coordinate is greater than the first</td>
<td>Both points have the same value</td>
</tr>
<tr>
<td>.25</td>
<td>Both points have the same value</td>
<td>Second point coordinate is greater than the first</td>
</tr>
<tr>
<td>.50</td>
<td>Second point coordinate is less than the first</td>
<td>Both points have the same value</td>
</tr>
<tr>
<td>.75</td>
<td>Both points have the same value</td>
<td>Second point coordinate is less than the first</td>
</tr>
</tbody>
</table>
Multiply the value by 360 to convert to degrees. Multiply the value by $2\pi$ to convert to radians.

**Notes:** This value never changes.

### MT7BoxToolGetCurrentDistance

This routine retrieves the current distance of the two points of the box tool.

**Usage:**
```c
float MT7BoxToolGetCurrentDistance (MT7BoxTool hTool);
```

**Arguments:**
- `hTool` Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`

**Return Values:**
The return value is the distance between the current points of the box tool. The value ranges between 0 and the square root of 2.

**Notes:** This value changes as the traces associated with the points gain events.

### MT7BoxToolGetCurrentAngle

This routine retrieves the current angle of the two points of the box tool.

**Usage:**
```c
float MT7BoxToolGetCurrentAngle (MT7BoxTool hTool);
```

**Arguments:**
- `hTool` Tool handle created by `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or `MT7BoxToolCloneTool`

**Return Values:**
The return value is the angle of the line through the current points of the box tool. This angle is relative to the X axis. The value of the angle ranges from 0 to 1. This table shows how to interpret these values.

<table>
<thead>
<tr>
<th>Angle</th>
<th>X coordinate</th>
<th>Y coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>.00</td>
<td>Second point coordinate is greater than the first</td>
<td>Both points have the same value</td>
</tr>
<tr>
<td>.25</td>
<td>Both points have the same value</td>
<td>Second point coordinate is greater than the first</td>
</tr>
<tr>
<td>.50</td>
<td>Second point coordinate is less than the first</td>
<td>Both points have the same value</td>
</tr>
<tr>
<td>.75</td>
<td>Both points have the same value</td>
<td>Second point coordinate is less than the first</td>
</tr>
</tbody>
</table>

Multiply the value by 360 to convert to degrees. Multiply the value by $2\pi$ to convert to radians.

**Notes:** This value changes as the traces associated with the points gain events.

### MT7BoxToolGetDistanceChange

This routine retrieves the change in distance between the initial points and current points.
Usage: float MT7BoxToolGetDistanceChange (MT7BoxTool hTool);
Arguments: hTool Tool handle created by MT7BoxToolCreateTool,
            MT7BoxToolCreateFixedTool, or
            MT7BoxToolCloneTool
Return Values: The return value is the change in distance between the initial and
              current line of the box tool. Specifically, this is the distance of the
              current line less the distance of the initial line. The value ranges
              between 0 and the square root of 2.
Notes: This value changes as the traces associated with the points gain events.

MT7BoxToolGetDistanceRatio
This routine retrieves the ratio of the current distance to the initial distance.
Usage: float MT7BoxToolGetDistanceRatio (MT7BoxTool hTool);
Arguments: hTool Tool handle created by MT7BoxToolCreateTool,
            MT7BoxToolCreateFixedTool, or
            MT7BoxToolCloneTool
Return Values: The return value is the ratio of the box tool’s initial distance to its
              current distance. If the initial distance is 0, the return value is the IEEE
              NAN (not a number) value.
Notes: This value changes as the traces associated with the points gain events.

MT7BoxToolGetAngleChange
This routine retrieves the change in the angle between the initial points and current
points.
Usage: float MT7BoxToolGetAngleChange (MT7BoxTool hTool);
Arguments: hTool Tool handle created by MT7BoxToolCreateTool,
            MT7BoxToolCreateFixedTool, or
            MT7BoxToolCloneTool
Return Values: The return value is the difference between the box tool’s initial and
              current angles, specifically the current angle less the initial angle.
              The API normalizes angle difference so that the value is between 0 and
              1. If the user rotates the trace clockwise by a small amount, it appears
              as a large counterclockwise rotation.
              Multiply the value by 360 to convert to degrees. Multiply the value by
              2\pi to convert to radians.
Notes: This value changes as the traces associated with the points gain events.

MT7BoxToolReleaseTool
This routine releases a tool handle.
Usage: void MT7BoxToolReleaseTool (MT7BoxTool hTool);
**Arguments:**

- `hTool`  Tool handle created by `MT7BoxToolCreateTool`,  
  `MT7BoxToolCreateFixedTool`, or  
  `MT7BoxToolCloneTool`  

**Notes:**

Use this routine to release a tool handle when the application no longer needs it. Call this routine for each tool handle created by  
`MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`, or  
`MT7BoxToolCloneTool`.

---

**Callback Routines**

There is only one type of callback routine used by the MT7BoxTool API,  
`MT7BoxToolCallback`. Remember that `MT7BoxToolCallback` is simply a function prototype. An application can have many routines that match this prototype and pass them into the `MT7BoxToolCreateTool`, `MT7BoxToolCreateFixedTool`,  
`MT7BoxToolCreateFixed2Tool`, and `MT7BoxToolCloneToolWithCallback`  
routines.

**MT7BoxToolCallback**

This routine handles an update notification from the API.

**Usage:**

```c
void MT7BoxToolCallback (MT7BoxToolReason eReason,  
MT7BoxTool hTool, void * pvUserCode);
```

**Arguments:**

- `eReason`  The reason why the API invoked the callback; either  
  `MT7BoxToolReason_Update` or  
  `MT7BoxToolReason_End`  
- `hTool`  Tool handle associated with the notification  
- `pvUserCode`  Value supplied by the application through the  
  `MT7BoxToolCreateTool` or  
  `MT7BoxToolCreateFixedTool` routines

**Notes:**

The application has different responsibilities based on the reason why  
the API invoked the callback.  

If the reason is `MT7BoxToolReason_Update`, there is a change in the  
monitored traces and hence a change in the current point positions and  
graphure.  

If the reason is `MT7BoxToolReason_End`, there was a liftoff on one of  
the traces. The application can get the final point positions and  
geometry. However, the application should release the tool handle.  
There will be no further updates to the tool.

---

**MT7 Box Tool API Object Handles**

An application uses handles to access the box tool objects in the MT7BoxTool API. Each  
handle has a unique value. Multiple handles may represent the same object.
MT7BoxTool

This handle represents a box tool. An application creates this handle when it has a trace it wishes to measure relative to a fixed point or another trace. The application uses this handle to provide information about the positions of the traces and their geometry.

MT7BoxToolReason

The MT7BoxToolReason is a code that notes what changed with a box tool’s traces. The API passes a reason code to the application, along with the tool’s handle, through a callback.

MT7BoxToolReason_Update

This reason code happens when there is a change in position of one or both traces of the tool.

MT7BoxToolReason_End

This reason code happens when one of the traces receives a liftoff event. The application should release the tool handle in this case.

MT7BoxToolCenter

The MT7BoxToolCreateTool may position the first point on one of two locations in the first trace. This location is either at the beginning of a trace or at its current location. The application specifies which location to use with one of these values.

MT7BoxToolCenter_Touchdown

The API positions the first point at the beginning of the trace.

MT7BoxToolCenter_Current

The API positions the first point at the current position of the trace.

MT7BoxToolInput

A box tool created with the MT7BoxToolCreateTool routine may process events from the first trace or may ignore the first trace. The application specifies which to do with one of these values.

MT7BoxToolInput_None

The tool ignores events from the first trace.

MT7BoxToolInput_Trace

The tool processes events from the first trace. Any event on this trace causes the API to invoke the box tool’s callback routine.
APPENDIX A

Troubleshooting

Overview

The MT7 Gesture Software family of APIs supports runtime diagnostics. These diagnostics may be helpful to see if there is any undesirable behavior, such as memory leaks, or to see what errors are in a configuration file. Normally these diagnostics do not appear. If desired, you define environment variables to view these diagnostics.

Controlling Diagnostic Outputs

Normally the MT7Gesture APIs do not produce diagnostics. To enable diagnostic outputs for a particular feature, define the appropriate environment variable.

Each environment variable may take one of three values. A “0” disables the diagnostic output. A value of “1” sends the output to standard output, which is normally a terminal window. A value of “2” sends the output to a debug logger, such as the output window in Microsoft’s Visual Studio. The API treats any other value as a “0”.

Windows

To temporarily enable a diagnostic, open a terminal window, set the appropriate environment variable, and run your program. The diagnostic messages should appear in the terminal window.

For example, if you want to enable memory diagnostics for a program called ‘MyProgram’, use the following commands:

> set MT7GestureMemoryDump=1
> MyProgram >output.txt

Note that normal “Windows” programs do not send data for standard output. You must redirect standard output to a file or ‘pipe’ it to another program like ‘more’. Programs built as “terminal” programs do not have this issue.

If you want to issue diagnostics over several sessions, you can set the environment variable in the system properties. Through the Window’s Control Panel, open the ‘System’ applet, go to the ‘Advanced’ tab, and press the ‘Environment Variable’ button. This presents a dialog where you can define the desired variables. The MT7Gesture API then shows the desired diagnostic messages any time you run your program.
Linux

To temporarily enable a diagnostic, open a terminal window, set the appropriate environment variable, and run your program. This means for setting environment variables differ between the various command shells. Consult the appropriate documentation for your shell.

When enabled, the diagnostic messages should appear in the terminal window.

For example, if you want to enable memory diagnostics for a program called ‘MyProgram’, use the following commands:

```
> MT7GestureMemoryDump=1
> export MT7GestureMemoryDump
> ./MyProgram >output.txt
```

If you want to issue diagnostics over several sessions, you can set the environment variable in login scripts.

General Purpose Diagnostics

**MT7GestureMemoryDump**

This variable controls the output of memory diagnostics. The APIs output these diagnostics when a program terminates. For each API, there is a start message and an end message. The end message details how many bytes of memory leaked and the number of allocations leaked.

If there were any leaks, the diagnostics also list the individual leaks between the start and end messages. This information may be useful to 3M Touch Systems’ technical support staff in diagnosing any memory leaks.

**MT7Stroke Diagnostics**

**MT7StrokeParserErrors**

When a program calls the MT7StrokeCreateEngine routine, the API parses a configuration file. If there is an error, the API does not create the engine. Enable this variable to view any errors in the file. If there are no errors, there is no output.

**MT7StrokeImageOutput**

It may be useful to see a visual representation of the reference strokes in a configuration file. This option produces a rough, character-based representation of the stroke. Any two strokes that appear similar may produce match conflicts.

**MT7StrokeCompareStrokes**

This variable enables another means to see if two reference strokes are close enough to produce match conflicts. When enabled, the API compares each reference stroke with each other and checks their score. If one reference stroke matches another, the API produces a diagnostic message stating with strokes matched and their score.
**MT7StrokeShowAllScores**

During testing, it may be useful to see how a trace scored against all the reference strokes. Set this variable and the API outputs the final scoring of a trace against each reference stroke.

**MT7Radial Diagnostics**

**MT7RadialParserErrors**

When a program calls the MT7RadialCreateLibrary routine, the API parses a configuration file. If there is an error, the API does not create the library. Enable this variable to view any errors in the file. If there are no errors, there is no output.

**Programs Will Not Start**

The MT 7 Gesture libraries use the Microsoft C++ runtime library, version 9. This package may not be present on older systems. If not present, it must be installed manually.

A symptom of this problem is an error message displayed when starting a program that uses the MT 7 Gesture libraries. This error message may be "Application Error - The application failed to initialize properly (0xc01500020)". Other error messages may mention the file MSVCR90.DLL.

If you encounter this problem, visit the Microsoft download center at [www.microsoft.com/downloads](http://www.microsoft.com/downloads). Search for 'Visual C++ 2008 Redistributable Package'. Find the package with the highest SP level (SP1 or better) and has 'x86' in the file name. Download the package and install it. If requested, reboot your system.

After installation of this package, your program using the MT 7 Gesture libraries should run.
# APPENDIX B

## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box tool</td>
<td>A tool that tracks the changes from the initial positions of two points to their current positions. At least one of the points is an active trace. This tool primarily tracks changes in the points’ distance and angle.</td>
</tr>
<tr>
<td>Controller</td>
<td>An electronic device that detects touches on the sensor and transmits the interpreted touches to the computer.</td>
</tr>
<tr>
<td>Display</td>
<td>Video monitor or display that produces an image</td>
</tr>
<tr>
<td>Drag</td>
<td>Corresponds to the movement of the touch along the sensor.</td>
</tr>
<tr>
<td>Driver</td>
<td>A software component that acquires touch events from controllers. Usually a driver translates these events into mouse actions.</td>
</tr>
<tr>
<td>Liftoff</td>
<td>Corresponds to the breaking of contact with the sensor.</td>
</tr>
<tr>
<td>Radial menu</td>
<td>A menu that presents its options around a point instead of as a linear list.</td>
</tr>
<tr>
<td>Reference Stroke</td>
<td>An abstract trace against which the engine attempts to match traces. A reference stroke can be closed or open, defined as either ending with a liftoff or not.</td>
</tr>
<tr>
<td>Sensor</td>
<td>A touch sensitive device that fits over a video display</td>
</tr>
<tr>
<td>Stroke Engine</td>
<td>A software object that is responsible for matching traces against reference strokes.</td>
</tr>
<tr>
<td>Touch system</td>
<td>The collection of the driver and the controllers, with their sensors.</td>
</tr>
<tr>
<td>Touch Controller</td>
<td>Electronic device that detects touches on the sensor and transmits the interpreted touches to the computer.</td>
</tr>
<tr>
<td>Touch Driver</td>
<td>Software that communicates with a controller and produces effects in the computer.</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Touch event.</td>
<td>Corresponds to touch activity on the sensor. The controller generates events. There are three types of events</td>
</tr>
<tr>
<td>Touch Sensor</td>
<td>Component that fits over the display that is sensitive to touches</td>
</tr>
<tr>
<td>Touch System</td>
<td>A general term referring to the sensor, controller, and driver.</td>
</tr>
<tr>
<td>Touchdown</td>
<td>corresponds to the initial contact on the sensor</td>
</tr>
<tr>
<td>Trace</td>
<td>series of touch events that begins with a touchdown event, with or without a subsequent drag event, and ends with a liftoff event</td>
</tr>
</tbody>
</table>
APPENDIX C

Tuning Parameters

The MT7 Gesture package allows a developer to tune some behaviors of its component APIs. The file MT7Gesture.xml contains these values. This file may exist in the current working directory when an application runs or the directory that holds the application itself.

The root element is “MT7Gesture” so the tags “<MT7Gesture>” and “</MT7Gesture>” must enclose the contents of the file.

The name of the API, “MT7Trace”, “MT7Stroke”, “MT7Radial”, and “MT7BoxTool” are the only elements of the root element. At this time, only “MT7Trace” has tuning parameters. The tag “<MT7Trace />” contains the tuning parameters pertinent to this API. The tuning parameters are attributes within the tag.

The table below shows the tuning parameters for the MT7Trace API. Again, these parameters are attributes of the MT7Trace element.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiftoffJoinDistance</td>
<td>Float</td>
<td>0</td>
<td>This is the maximum distance, between 0 and 1, that a new trace must be from the liftoff of a delayed trace to be joined to that trace. This aids with hiding any inadvertent liftoffs from an application. The units are a fraction of a screen dimension, so a .5 is half the screen dimension.</td>
</tr>
<tr>
<td>LongLiftoffDelay</td>
<td>Integer</td>
<td>0</td>
<td>This is the number of milliseconds that the API delays the liftoff event on a ‘long’ trace. If 0, there is no delay. This aids with hiding any inadvertent liftoffs from an application. See ‘LongTraceDistance’ for more details.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data Type</td>
<td>Default Value</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LongTraceDistance</td>
<td>Float</td>
<td>0</td>
<td>This is the distance, between 0 and 1 that a trace must travel to use the employ the ‘LongLiftoffDelay’ to suppress inadvertent liftoffs. If the trace never exceeds this distance, the API uses the ‘ShortLiftoffDelay’ instead. The units are a fraction of a screen dimension, so a .5 is half the screen dimension.</td>
</tr>
<tr>
<td>ShortLiftoffDelay</td>
<td>Integer</td>
<td>0</td>
<td>This is the number of milliseconds that the API delays the liftoff event on a ‘short’ trace. If 0, there is no delay. This aids with hiding any inadvertent liftoffs from an application. See ‘LongTraceDistance’ for more details.</td>
</tr>
<tr>
<td>LiftoffJoinDistance</td>
<td>Float</td>
<td>0</td>
<td>This is the maximum distance, between 0 and 1 that a new trace must be from the liftoff of a delayed trace to be joined to that trace. This aids with hiding any inadvertent liftoffs from an application. The units are a fraction of a screen dimension, so a .5 is half the screen dimension.</td>
</tr>
</tbody>
</table>

The following is a sample of how a file may look.

```xml
<MT7Gesture>
  <MT7Trace
    LiftoffJoinDistance=".1"
    LongTraceDistance=".2"
    LongLiftoffDelay="200" />
</MT7Gesture>
```

The effect here is that for any long trace, with distance traveled greater that 20% of the screen dimension, the API does not immediately announce the trace’s liftoff. Instead, the API waits 200 milliseconds to expire before generating the liftoff event. If a second trace starts within the 200 milliseconds and within 10% of the screen dimension from the liftoff event, the API treats the second trace as a continuation of the first trace. The application never sees the second trace. By contrast, the API immediately announces the liftoff event of a short trace since its delay is 0 milliseconds.
APPENDIX D

Sample Code

The following are code snippets and sample XML files that illustrate how to use the MT7 Gesture APIs. Note that these are examples only and a developer must write any production code to address the specific needs of an application.

Note that these examples do not check for possible error conditions.

Simple Trace Example

The snippet here uses a universal callback and polling to access traces.

```c
/* Our callback routine. */
void MyCallback (MT7TraceReason eReason, void * hItem,
                void * pvUserCode)
{
    MT7TraceTrace hTrace;
    MT7TraceEvent hEvent;

    switch (eReason)
    {
        /* We don't care about sensor arrival or departure */
        case MT7TraceReason_Arrival:
        case MT7TraceReason_Departure:
            MT7TraceReleaseSensor (hItem);
            break;

        /* Have new traces on a sensor; call GetNextTrace and GetNextEvent repeatedly */
        case MT7TraceReason_NewTrace:
            while ((hTrace = MT7TraceGetNextTrace (hItem)) != NULL)
            {
                while ((hEvent = MT7TraceGetNextEvent (hTrace)) != NULL)
                {
                    DoSomething (hTrace, hEvent);
                    MT7TraceReleaseEvent (hEvent);
                }
            }
    }
```
MT7TraceReleaseEvent (hTrace);
} MT7TraceReleaseSensor (hItem);
break;

/* Have new events on a trace; call GetNextEvent repeatedly */
case MT7TraceReason_NewEvent:
while ((hEvent = MT7TraceGetNextEvent (hItem)) != NULL) {
    DoSomething (hItem, hEvent);
    MT7TraceReleaseEvent (hEvent);
} MT7TraceReleaseEvent (hItem);
break;
}

int main (int argc, char * argv [])
{
    /* Set the universal callback and poll forever */
    MT7TraceSetUniversalCallback (MyCallback, NULL);
    for (;;) {
        MT7TracePoll ();
    }
}

**Trace with Enumeration Example**

The snippet here enumerates sensors, tests their names, and installs callbacks on the sensors.

typedef enum
{
    ScreenLeft, ScreenRight,
} ScreenPosition;

/* Our callback routines. */
void TraceCallback (MT7TraceReason eReason, void * hItem, 
    void * pvUserCode)
{
    MT7TraceEvent hEvent;
    MT7TraceEventType eType;

    /* A trace callback receives only NewEvent reasons */
    while ((hEvent = MT7TraceGetNextEvent (hItem)) != NULL) {
        DoSomething ((ScreenPosition) pvUserCode, hEvent);
        eType = MT7TraceGetEventType (hEvent);
        MT7TraceReleaseEvent (hEvent);
    }
}
/* If this was a liftoff, terminate the trace */
if (eType == MT7TraceEvent_Liftoff)
{
    MT7TraceReleaseTrace (hItem);
    break;
}
}
}

void SensorCallback (MT7TraceReason eReason, void * hItem,
void * pvUserCode)
{
    MT7TraceTrace hTrace;
    switch (eReason)
    {
    /* Someone disconnected the sensor so release it */
    case MT7TraceReason_Departure:
        MT7TraceReleaseSensor (hItem);
        break;

    /* Have new traces on a sensor; call GetNextTrace, assign a
    callback to them, and process their events */
    case MT7TraceReason_NewTrace:
        while ((hTrace = MT7TraceGetNextTrace (hItem)) != NULL)
        {
            MT7TraceSetTraceCallback (hTrace, TraceCallback,
                                        pvUserCode);
            TraceCallback (MT7TraceReason_NewEvent, hTrace,
                           pvUserCode);
        }
        break;
    }
}

int main (int argc, char * argv [])
{
    MT7TraceEnum hEnum;
    MT7TraceSensor hSensor;

    /* Enumerate the sensors */
    hEnum = MT7TraceCreateEnum ();
    while ((hSensor = MT7TraceGetNextSensor (hEnum)) != NULL)
    {
        ScreenPosition ePosition;
        if (strcmp (MT7TraceGetSensorName (hSensor),
                    "USB00000001") == 0)
        {
            ePosition = ScreenLeft;
        }
else ePosition = ScreenRight;
MT7TraceSetSensorCallback (hSensor, SensorCallback,
   (void *) ePosition);
}
MT7TraceReleaseEnum (hEnum);

/* Poll forever */
for (;;)
{
    MT7TracePoll ();
}

Simple Stroke Example

The snippet here captures a user’s gestural actions to determine how they want to move an underlying image. There are two parts, an XML definition file and C source code.

XML file (sample.xml)

```xml
<Strokes>
  <Stroke Name="NDrag">N</Stroke>
  <Stroke Name="SDrag">S</Stroke>
  <Stroke Name="EDrag">E</Stroke>
  <Stroke Name="WDrag">W</Stroke>
  <Stroke Name="NFlick" Timeout="250">N</Stroke>
  <Stroke Name="SFlick" Timeout="250">S</Stroke>
  <Stroke Name="EFlick" Timeout="250">E</Stroke>
  <Stroke Name="WFlick" Timeout="250">W</Stroke>
</Strokes>
```

Code

```c
/* Our callback routine. */
void MyCallback (MT7MatchReason eReason, MT7StrokeMatch hMatch, 
   void * pvUserCode)
{
    /* Process only matches */
    if (eReason == MT7MatchReason_Match)
    {
        const char * pszName = MT7StrokeGetMatchName (hMatch);
        if (pszName [1] == 'F') Page (pszName [0]);
        else Drag (pszName [0]);
    }
}
```
```c
int main (int argc, char * argv [])
{
    MT7StrokeEngine hEngine;

    /* Set up the stroke engine */
    hEngine = MT7StrokeCreateEngine ("sample.xml",
        MyCallback, NULL);
    MT7StrokeAddAllSensors (hEngine);

    /* Poll for touch events */
    for (;;)
    {
        MT7TracePoll ();
    }

    /* Release the engine */
    MT7StrokeReleaseEngine (hEngine);
}

Complex Stroke Example

The snippet here differs from the previous example in that it uses gestures only in the upper left corner of a sensor. It uses the same XML file from above.

/* Declare the engine handle to be global so that the sensor callback can access it. */
MT7StrokeEngine hEngine;

/* Our callback routines. */
void UniversalCallback (MT7TraceReason eReason, void * hItem, void * pvUserCode)
{
    MT7TraceTrace hTrace;
    MT7TraceTrace hEvent;

    switch (eReason)
    {
        case MT7TraceReason_Arrival:
        case MT7TraceReason_Departure:
            MT7TraceReleaseSensor (hItem);
            break;
    }
```
case MT7TraceReason_NewTrace:
    while ((hTrace = MT7TraceGetNextTrace (hItem)) != NULL)
    {
        hEvent = MT7TraceGetEvent (hTrace, 0);
        if (MT7TraceGetEventX (hEvent) > .5 &&
            MT7TraceGetEventY (hEvent) > .5)
        {
            MT7StrokeAddTrace (hEngine, hTrace);
        }
        MT7TraceReleaseEvent (hEvent);
        MT7TraceReleaseTrace (hTrace);
    }
    MT7TraceReleaseSensor (hSensor);
    break;

    case MT7TraceReason_NewEvent:
        MT7TraceReleaseTrace (hItem);
        break;
    }
}

void MatchCallback (MT7MatchReason eReason, MT7StrokeMatch hMatch, 
    void * pvUserCode)
{
    /* Process only matches */
    if (eReason == MT7MatchReason_Match)
    {
        const char * pszName = MT7StrokeGetMatchName (hMatch);
        if (pszName [1] == 'F') Page (pszName [0]);
        else Drag (pszName [0]);
    }
}

int main (int argc, char * argv [])
{
    /* Set up the stroke engine */
    hEngine = MT7StrokeCreateEngine ("sample.xml",
        MatchCallback, NULL);
    MT7SetUniversalCallback (UniversalCallback, NULL);

    /* Poll for touch events */
    for (;;)
    {
        MT7TracePoll ();
    }

    /* Release the engine */
    MT7StrokeReleaseEngine (hEngine);
}
Simple Radial Example

This snippet demonstrates the use of a simple radial menu. It monitors the MT7Trace API to obtain traces to launch the menu.

**XML file (sample.xml)**

```xml
<Menus>
  <Menu Name="MyMenu" Inner=".2" Outer=".5">
    <Border RGB="1,.5,0" />
    <Visual Input="1" />
    <Activation Border="Touchdown,Liftoff,Linger"
               Outside="Touchdown,Liftoff,Linger" />
    <Action Name="1" Label="This" />
    <Action Name="2" Label="That" />
    <Action Name="3" Label="The Other" />
  </Menu>
</Menus>
```

**Code**

```c
MT7RadialTemplate hTemplate;

/* Our callback routines. */
void MenuCallback (MT7RadialReason eReason,
                  MT7RadialAction hAction,
                  void * pvUserCode)
{
  if (eReason == MT7RadialReason_Action)
  {
    DoAction (MT7RadialGetName (hAction));
  }
}

void UniversalCallback (MT7TraceReason eReason, void * hItem,
                        void * pvUserCode)
{
  MT7TraceTrace hTrace;

  switch (eReason)
  {
    case MT7TraceReason_Arrival:
    case MT7TraceReason_Departure:
      MT7TraceReleaseSensor (hItem);
      break;
    case MT7TraceReason_NewTrace:
      while ((hTrace = MT7TraceGetNextTrace (hItem)) != NULL) {
```
Simple Box Tool Example

This snippet demonstrates the use of a simple box tool. It monitors the MT7Trace API to obtain traces to launch the box tool.

This code makes the simplistic assumption that the user touches the sensor with only two contacts at a time. Production code should consider all possible user actions, including the possibility that the user may terminate the first trace.

/* First trace */
MT7TraceTrace hTrace1 = NULL;
/* Our callback routines. */
void ToolCallback (MT7BoxToolReason eReason, MT7BoxTool hTool, void * pvUserCode)
{
/ * Process only matches */
if (eReason == MT7BoxToolReason_End)
{
    MT7BoxToolReleaseTool (hTool);
}
else
{

    UpdateImage (MT7BoxToolGetDistanceRatio (hTool),
                   MT7BoxToolGetAngleChange (hTool));
}

void UniversalCallback (MT7TraceReason eReason, void * hItem,
                        void * pvUserCode)
{
    MT7TraceTrace hTrace;

    switch (eReason)
    {
        case MT7TraceReason_Arrival:
        case MT7TraceReason_Departure:
            MT7TraceReleaseSensor (hItem);
            break;
        case MT7TraceReason_NewTrace:
            while ((hTrace = MT7TraceGetNextTrace (hItem)) != NULL)
            {
                if (hTrace1 == NULL) hTrace1 = hTrace;
                else
                {
                    MT7BoxToolCreateTool (hTrace1,
                                          MT7BoxToolCenter_Current,
                                          MT7BoxToolInput_Trace, hTrace,
                                          MT7BoxToolCenter_Touchdown, ToolCallback, NULL);
                    MT7TraceReleaseTrace (hTrace1);
                    MT7TraceReleaseTrace (hTrace);
                    hTrace1 = NULL;
                }
            }
            MT7TraceReleaseSensor (hSensor);
            break;
        case MT7TraceReason_NewEvent:
            MT7TraceReleaseTrace (hItem);
            break;
    }
}

int main (int argc, char * argv [])
{
    /* Poll for touch events */
    MT7SetUniversalCallback (UniversalCallback, hTemplate);
Radial Menu with a Box Tool Example

This snippet demonstrates the use of a radial menu with a box tool. It monitors the MT7Trace API to obtain traces to launch the menu.

This code makes the simplistic assumption that the user touches the sensor in an orderly fashion. Production code should consider all possible user actions.

```c
MT7RadialMenu hMenu = NULL;
MT7RadialTemplate hTemplate;

/* Our callback routines. */
void ToolCallback (MT7BoxToolReason eReason, MT7BoxTool hTool, void * pvUserCode)
{
    /* Process only matches */
    if (eReason == MT7BoxToolReason_End)
    {
        MT7BoxToolReleaseTool (hTool);
    }
    else
    {
        UpdateImage (MT7BoxToolGetDistanceRatio (hTool),
                     MT7BoxToolGetAngleChange (hTool));
    }
}

void MenuCallback (MT7RadialReason eReason,
                   MT7RadialAction hAction,
                   void * pvUserCode)
{
    if (eReason == MT7RadialReason_Action)
    {
        if (strcmp (MT7RadialGetActionName (hAction), "1") == 0)
        {
            MT7TraceTrace hTrace = MT7RadialGetActionTrace (hAction);
            MT7BoxToolCreateFixedTool (MT7RadialGetMenuX (hMenu),
                                        MT7RadialGetMenuY (hMenu), hTrace,
                                        MT7BoxToolCenter_Current, ToolCallback, NULL);
            MT7RadialReleaseMenu (hMenu);
            MT7TraceReleaseTrace (hTrace);
            MT7TraceReleaseTrace (hTrace);
            hMenu = NULL;
        }
    }
}```
void UniversalCallback (MT7TraceReason eReason, void * hItem, 
    void * pvUserCode)
{
    MT7TraceTrace hTrace;
    switch (eReason)
    {
    case MT7TraceReason_Arrival:
    case MT7TraceReason_Departure:
        MT7TraceReleaseSensor (hItem);
        break;

    case MT7TraceReason_NewTrace:
        while (hMenu == NULL && 
            (hTrace = MT7TraceGetNextTrace (hItem)) != NULL)
        {
            hMenu = MT7RadialStartMenu (hTemplate, hTrace, 
                MT7RadialCenter_Default, MT7RadialInput_Sensor, 
                MenuCallback, NULL);
            MT7TraceReleaseTrace (hTrace);
        }
        MT7TraceReleaseSensor (hSensor);
        break;

    case MT7TraceReason_NewEvent:
        MT7TraceReleaseTrace (hItem);
        break;
    }
}

int main (int argc, char * argv [])
{
    MT7RadialLibrary hLibrary;

    /* Get up the menu template */
    hLibrary = MT7RadialCreateLibrary ("sample.xml");
    hTemplate = MT7RadialGetTemplate (hLibrary, "MyMenu");
    MT7RadialReleaseLibrary (hLibrary);

    /* Poll for touch events */
    MT7SetUniversalCallback (UniversalCallback, hTemplate);
    for (;;)
    {
        MT7TracePoll ();
    }

    /* Release the template */
    MT7RadialReleaseTemplate (hTemplate);
}