Part 2 – MTC Express Developer's Guide

Introduction

What's an API?

Tactex provides an Application Program Interface (API) for development of custom applications and MAX external objects utilising the MTC Express. This API is provided for both Windows 95/98 and Mac OS 8 platforms. The MTC Express API consists of a set of software tools that a software developer can use to build applications which connect to the MTC Express and control the data stream from it.

Who Should Read This?

This Developer's Guide is required reading if you want to do one of the following:

- create a MAX external object in C
- create a software application that uses input from the MTC Express
- create a plug-in for an existing application that uses input from the MTC Express.

Overview of the API

The API consists of a set of C-language libraries to which the developers can link to their custom code.

The API provides features for accessing raw data from the individual pressure sensors, calibrated data, or resolved pointer locations and intensities.

The API initiates a serial communication link with the MTC Express. It decodes the serial data stream, and determines the x-y coordinates of each data element by means of configuration information held in a "mapping" file. The API also provides an automatic means of calibrating, or "normalizing" the data. The API handles the serial connection to the touch-pad, as well as the disk file I/O required for calibration and configuration. Several C-source files (for both Windows 95/98 and Mac OS 8) are included which demonstrate the use of the API within applications.

Requirements

In order to use the API, you must:

- know how to use the C programming language
- know how to program for Win32 applications or Mac OS applications
- have a computer with one of the following development systems:
  - Macintosh Metrowerks CodeWarrior release 5
  - Windows 9x Metrowerks CodeWarrior release 3
  - Microsoft Visual C++ version 5

Which Files Do I Need?

The MTC Express API is a collection of functions that have been compiled into a C-language library. Although the functional interface is identical (with a few small exceptions) between Macintosh OS applications, Win32 applications, and Macintosh MAX applications, different libraries and header files are required for each.

Macintosh applications:

mtc_ppc.lib
mtc.h
mtc_mac.h

MAX external objects:

mtc_max.lib
mtc.h
mtc_max.h

Win32 applications:

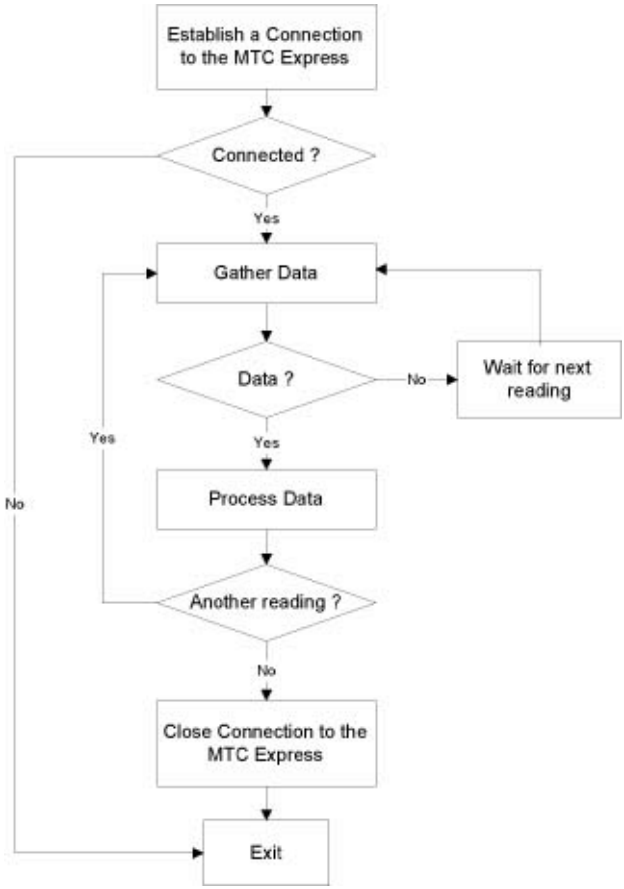
mtcwin32.lib
mtc.h
mtc_win32.h

Basics

This section provides the basic information you will need to use the API. Some examples which describe the use of the API are shown. More detailed descriptions of the functions can be found in the API Reference section.

Gathering data from the MTC Express

Applications accessing MTC Express first need to establish a connection to the MTC Express through the API. Before exiting your application, the connection to the MTC Express must be released. The figure below illustrates the typical functional flow when gathering MTC Express data. The following sections provide the instructions how to implement the flow chart.



Establishing a Connection to the MTC Express

Initiating a connection to the MTC Express is straightforward. You need to know the following things:

- The name and location of the mapping file (more on that later).
- The name and location of the normalization file (more on that later, too).
- The name of the serial port to which the MTC Express is connected.

The following snippet of code demonstrates a function that establishes a connection to an MTC Express on serial port A of a Macintosh.

```
#include "mtc_mac.h"
#include "mtc.h"

static MTCHandle              myMTC;
static MTCCompType            myUseCompression = MTCCompNone;
const  int                    kWaitTime = 500;

BOOL connect_to_mtc(void)
{
        MTCCreate      myMtcCreate;

        // myMtcCreate specifies the serial port
        // and file paths&names:
        // Mac OS serial input driver
        myMtcCreate.inputPort = (char *)"\p.AIn";

        // Mac OS serial output driver
        myMtcCreate.outputPort = (char *)"\p.AOut";

        // path and filenames
        myMtcCreate.mappingFile = "mapping.txt";
        myMtcCreate.normalFile = "normal.txt";

        myMTC = MTC_New(myMtcCreate);

        if (!myMTC)
                return(FALSE);

        // Start the data stream
        MTC_StartSendingData(myMTC,
        myUseCompression, kWaitTime);

        return(TRUE);
}
```

The function MTC_New takes as its only parameter an MTCcreate structure. Filling the MTCCreate structure is the key to successfully connecting to the MTC Express. That structure has four fields:

inputPort and outputPort –

The MTC Express API is a software layer that lies in-between the operating system serial driver and your application. When you call MTC_New, the API takes the names of the serial driver that you specify in the MTCCreate structure and passes them through to the operating system.

On the Macintosh, serial connections have names such as "Modem" for the benefit of the user, as well as two driver names for each serial port, such as ".AIn" and ".AOut". The API requires you to use the driver names. Furthermore, since the Macintosh Device Manager expects these to be pascal strings (i.e. unsigned char arrays with the first character being the string length) rather than null-terminated strings, you must provide pascal strings. MetroWerks CodeWarrior allows you to use the escape code '\p' as the first character to cause the compiler to generate a pascal string. Hence the two lines of code above which declare pascal strings, "\p.AIn", and cast them as character arrays for the purposes of assigning the fields of the MTCCreate structure. To make your job somewhat easier, the Macintosh OS provides a set of tools in the Communication Toolbox for determining which serial drivers are installed in your system, and what their serial driver names are.

On Win32 systems, the situation is much easier. There is only one driver name for both input and output. The serial port names are typically "com1" , "com2" , etc. The Win32 OS expects a null-terminated string, so the simple assignment such as
`myMtcCreate.inputPort = myMtcCreate.outputPort = "com1";`
is all that is required.

mappingFile–

This is a pointer to a null-terminated string specifying the path and filename of the mapping file (described subsequently). A valid mapping file must be present in the specified location or MTC_New will fail to complete a connection and it will return NULL.

normalFile –

This is a pointer to a null-terminated string specifying the path and filename of the normalization file (also described subsequently). If a normalization file with the specified name does not exist, then the API will create one with default calibration parameters.

The MTC_New function returns a "handle to the MTC", myMTC. Nearly every other function in the API takes that handle as its first parameter. That's why myMTC is declared as a global variable in the code above – if you lose its value, subsequent calls to the API functions will fail. Note that for cross-platform compatibility, the API uses the type definition BOOL and the defined values TRUE and FALSE. These are defined in mtc_mac.h, mtc_max.h, and mtc_win32.h.

The function MTC_StartSendingData is called to start the stream of data from the MTC Express to the computer. (Without this call, the MTC Express would sit there idly.) You provide three parameters:

myMtc –
: this is the handle to the MTC which you were just given.

myUseCompression –
: this specifies which method of data compression to use for the serial communication. For most purposes, you can simply use the value MTCCompNone, as shown.

kWaitTime –
: several of the API functions send a message to the MTC Express over the serial port and expect a particular response. In this example, MTC_StartSendingData sends a message to initiate the data streaming. You can specify to the API how long it should wait until the it receives the anticipated response – here we will wait up to 500 milliseconds. MTC_StartSendingData will not return until the first packet of data has been received from the MTC Express or kWaitTime has elapsed. MTC_StartSendingData will return TRUE if the data stream was initiated successfully, or FALSE if the API failed to receive data from the MTC within the specified time.

Getting the Touch-Pad Configuration Data

As you will see in subsequent sections, the API sends your application taxel data by filling a WORD array (WORD is a 16-bit unsigned integer, which has been defined in mtc_mac.h, mtc_max.h and mtc_win32.h). It is your responsibility to allocate the memory required for that array. How much memory should you allocate? Well, you probably already know that the MTC Express has 72 taxels, so you could simply declare an array like this:

```
WORD            myTaxelPressures[72];
```

But that is not the recommended way. At present, the MTC Express is the only product which uses this API, but Tactex is sure to introduce more products in future. Those future products may have a different number of taxels and a different physical configuration. (Most of those properties are defined in the mapping file, which we will discuss later.) If you want your application to remain compatible with future products, then you can't make assumptions about the number of taxels, the individual taxel locations, or the size and shape of the touch pad. The API provides means of obtaining this information. The following snippet of code shows how to determine the number of taxels and then allocates an appropriate amount of memory for the taxel data.

```
WORD    *allocate_mtc_data_mem(void)
{
        MTCConfig      myConfig;

        // Get the number of taxels in
        // the MTC Express unit.
        MTC_GetConfig(myMTC, &myConfig);

        // Allocate the memory needed for data storage.
        return(malloc(myConfig.nTaxels * sizeof(WORD)));
}
```

The function above calls the API function `MTC_GetConfig` to fill the `MTCConfig` structure. The `MTCConfig` structure holds information related to the size of the pad and the general arrangement of taxels, among other things. In this example, we want to know how many taxels the pad has, and that value is held in the `nTaxels` field.

Getting Data from the MTC Express

Once a connection to the MTC Express has been established, you can access the data steam. The data is available in two flavours: (1) "raw" data is delivered just as the MTC Express spits it out, or (2) "normalized" data has been calibrated to provide similar outputs for similar pressure applied to each taxel. Raw data directly from the MTC Express can be accessed by using the `MTC_GetRawData` function. Normalized data from the MTC Express can be accessed by using the `MTC_GetNormalizedData` function. The following example demonstrates a function which fills an array with normalized data.

```
long    gather_mtc_data(WORD *pPress)
{
        long    sampleNum;

        // Loop here until we get some data.
        do
        {
                // Request normalized data.
                sampleNum = MTC_GetNormalizedData(
                            myMTC, pPress);

                // We could have used the following
                // to request raw data:
                // sampleNum = MTC_GetRawData(
                //             myMTC, pPress);
        }
        while (sampleNum == 0);

        // Record sample number for timing.
        return (sampleNum);
}
```

After a call to the example `gather_mtc_data` function, the `WORD` array pointed to by `pPress` will be filled with data. Three things are important to know about that data:

- The data is in the range of `0` to `MTC_FullScale`. `MTC_FullScale` is defined to be `1023` in the mtc.h.
- The order of the data is the order in which data is streamed from the MTC Express. You cannot make any assumptions about how the geometric configuration of the taxels relates to the order of the data. The API provides several means to associate the data with the physical locations of the taxels; these are described in the section entitled "Pad Configuration (Mapping)".
- Only the latest data is reported. If several samples have been received in-between calls to `MTC_Get…Data`, all but the latest data are thrown away.

The functions `MTC_GetNormalizedData` and `MTC_GetRawData` both return immediately (i.e. they do not wait for new data to arrive). They will fill `pPress` with the latest data that has been received, provided it has not been accessed by a previous call to `MTC_GetNormalizedData` or `MTC_GetRawData`. In other words, the API will report any given data only once. `The MTC_Get…Data` functions return the sample number of that data. If no data has arrived since the last call to an `MTC_Get…Data` function, then they will return `0`.

Processing the Pressure Data

As mentioned previously, the pressure data is reported as a `WORD` array. Both raw and normalized data are in the range of `0` to `MTC_FullScale`. And, as also previously mentioned, it is your responsibility to allocate memory for the data. After the API has delivered the data, you are free to use it as you will. In most cases, it will be necessary to relate the data to a physical location on the pad. Correlating data to the physical location is such an important topic that it has its own section in this guide – please refer to "Pad Configuration (Mapping)".

Closing a connection to the MTC Express

Before your application exits, it must disconnect from the MTC Express using the `MTC_Delete` function. This allows the API to close the serial port and free system resources. The following code segment demonstrates a proper MTC Express shut-down and freeing the local data memory.

```
 void disconnect_from_mtc(WORD *pPress)
{
// Stop data stream & close the connection
MTC_Delete(myMTC);

// Render MTC handle invalid
myMTC = (MTCHandle)0;

// Free pointer to data storage
free(pPress);
pPress = NULL;
}
```

A Complete Example

The foregoing has provided enough information for you to create a simple application. An example application is given on the CD in the following location:

For Macintosh – **MTC Express:Projects:Mac:Example1-basic:**

For Windows – **E:\Projects\Win32\MSVC Example1 Basic**

Pad Configuration (Mapping)

The Mapping File

Your MTC Express contains an embedded array of 72 pressure sensing elements called taxels. Information regarding the physical configuration of the pad is stored in a disk file called the mapping file. The mapping file defines the number of taxels, the xy-coordinates of individual taxels in millimetres, and the row/column locations of the taxels, among other things.

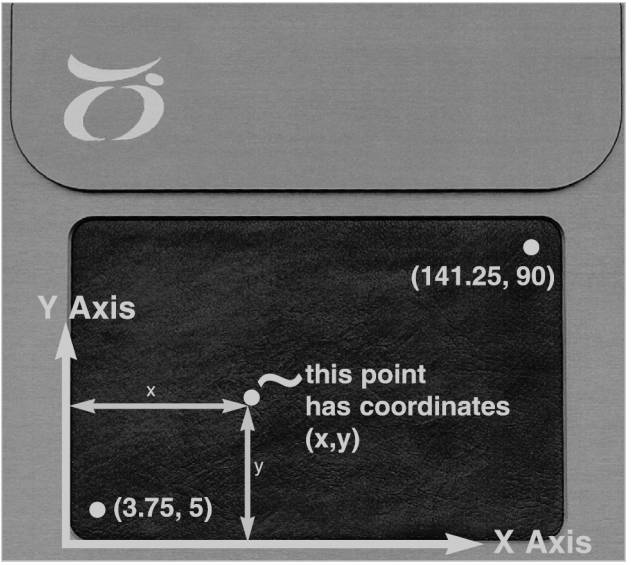
For the purposes of the demonstration application provided with your MTC Express, the mapping file is named *mapping.txt*, and it is located in the same folder as the application. However, the name of the mapping file and its location are specified by the developer (that is, you). You specify the name and location when you call the `MTC_Create` function, as shown previously.

The API requires a specific format of the mapping file. In order to use the API, you will probably never need to know what the format of the mapping file is (or even to look at the mapping file, for that matter). However, for curious developers, the mapping file format is described in Appendix B.

Where are the taxels?

If your application works with the taxel data stream (as opposed to the pointer information), you will almost certainly need to know the physical location of the taxel to which any given data applies. The API provides two means to do this.

The first way to determine the location of a taxel is to access the xy-coordinate information. The figure below shows the Cartesian coordinate frame used to specify the xy-coordinate information. As you can see, the left side of the pad corresponds to x=0, the bottom of the pad corresponds to y=0, and the origin of the coordinate frame (i.e. the location x=0, y=0) is at a point slightly off the lower left corner of the pad.



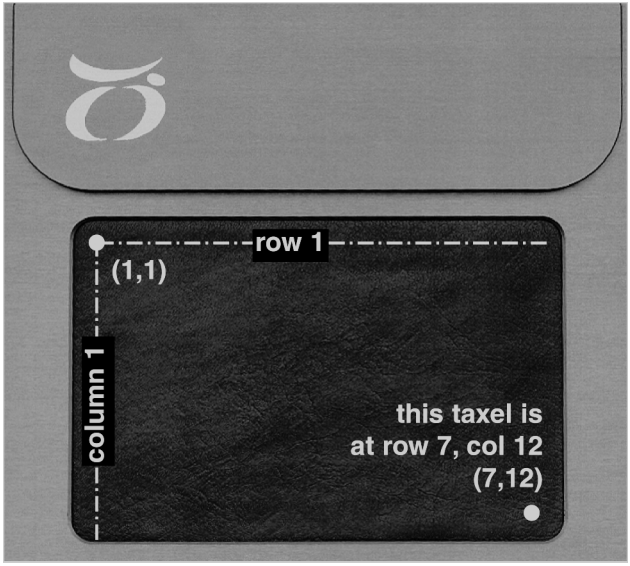
The API function that you can use to determine the size of the pad is:

```
MTC_GetConfig()
```

The API functions you can use to access the xy-coordinates of the taxels are:

```
MTC_GetTaxelX()
MTC_GetTaxelY()
```

The second way to determine the location of a taxel is to access the row-column coordinates. Although the row-column coordinates are similar to the xy-coordinates in some ways, using them may be simpler in some instances. The figure below illustrates the row-column coordinate frame. Notice that unlike the Cartesian coordinate system, the row-column coordinates start at (1,1) in the top left corner (this corresponds to the typical way arrays and matrices are indexed).



The API function that you can use to determine the size of the pad is:

```
MTC_GetConfig()
```

The API functions that you can use to access the row-column coordinates (i.e. indices) of the taxels are:

```
MTC_GetTaxelCol()
MTC_GetTaxelRow()
```

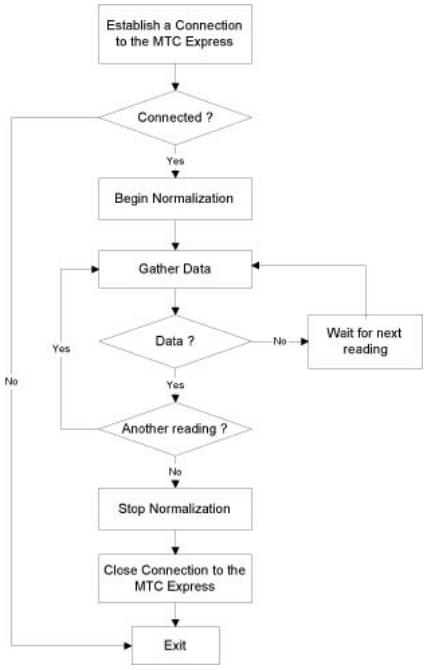
Calibration (Normalization)

The nature of the MTC Express Smart Fabric surface is such that the response of an individual taxel to a given applied pressure may not be identical to the response of another taxel under the same pressure. Therefore, the API provides a means to calibrate, or *normalize*, the pad.

The normalization procedure should be performed when you first receive the MTC Express and when you want to adjust the sensitivity level. Refer to the Part 1: Owner's Guide for additional information about the procedure. Normalization only needs to be performed occasionally because the results can be saved in a normalization file. For the purposes of the demonstration programs provided with your MTC Express, the normalization file is named normal.txt, and it is located in the same folder as the application program. You specify the name and location when you call the `MTC_Create` function, as shown previously.

Normalizing the MTC Express

Your application can support the process of normalization, if desired. The normalization procedure requires you to gather data from the MTC Express while the user applies uniform pressure over the pad, as described in Part 1: Owner's Guide. This is illustrated in the following functional flow diagram.



The following code example demonstrates the normalization procedure when using the API.

```
#include "mtc_mac.h"
#include "mtc.h"

BOOL normalize_my_mtc()
{
        MTCHandle       myMTC;
        MTCCreate       myCreate;
        MTCConfig       myMTCConfig;
        long            sampleNum;
        BOOL            allDone;
        WORD            *pPress;

        // Establish a connection
        myCreate.inputPort = (char *)"\p.AIn";
        myCreate.outputPort = (char *)"\p.AOut";
        myCreate.mappingFile = "mapping.txt";
        myCreate.normalFile = "normal.txt";
        myMTC = MTC_New(myCreate);
        if (!myMTC)
                return FALSE;

        // Find the number of taxels in the
        // MTC Express and allocate the memory
        // needed for data storage.
        MTC_GetConfig(myMTC, &myMTCConfig);
        pPress = malloc(myMTCConfig.nTaxels * sizeof(WORD));
        if(!pPress)
        {
                // Report memory allocation error
                MTC_Delete(myMTC)
                return(FALSE);
        }

        // Start the data stream
        MTC_StartSendingData(myMTC, MTCCompNone, 500);

        // Configure the API to begin the
        // normalization procedure
        MTC_BeginNormalization(myMTC);

        // Loop here until the user indicates he's
        // completed the normalization procedure
        do
        {
```

```
                // gather data
                sampleNum = MTC_GetNormalizedData(
                myMTC, pPress);

                // placeholder for your function
                // to wait for user's signal
                allDone = isNormalizationDone();

                // use the data here, if desired.

        }
        while (!allDone);

        // Stop the normalization procedure
        MTC_EndNormalization(myMTC, TRUE);

        // Disconnect from the MTC Express
        MTC_Delete(myMTC);

        // Render MTC handle invalid
        myMTC = (MTCHandle)0;

        // Free the taxel data storage memory
        free(pPress);
        pPress = NULL;

        return(TRUE);
}
```

This example is very similar to previous examples. `MTC_New` is called to establish a connection, `MTC_Config` is called to determine how many taxels there are, and `MTC_StartSendingData` is called to initiate the data stream. Then, `MTC_BeginNormalization` is called to start the normalization process. The only parameter passed to `MTC_BeginNormalization` is the handle to the MTC.

Following the foregoing set-up, the program enters a loop which gathers data. It is essential that this loop is carried out – data gathering is necessary for the normalization to work, and API does not automatically call for data. That's just the way it is. At any rate, if your application allows the user to normalize the MTC Express, it will most likely be necessary to display the data to the user. On every call to `MTC_GetNormalizedData` made in this loop, the API updates the calibration parameters.

At some point, the user will decide that the normalization process is complete, and will strike a key or make a menu choice. In the example above, the call to `isNormlalizationDone` is a placeholder for your application to insert the appropriate way to check if the user has indicated that normalization is complete. After exiting the loop,

`MTC_EndNormalization` is called. That function signals to the API to freeze the calibration parameters. The second parameter to `MTC_EndNormalization` is a `BOOL` which directs the API to whether or not to overwrite the existing normalization file.