MetaStock®

Developer's Kit

For MetaStock Solution Providers

Version 9.1

Printed in the USA

Equis International
90 South 400 West
Suite 620
Salt Lake City, UT USA
84101

http://www.equis.com

9/23/05

9 8 7 6 5 4 3 2 1 **IP**

Table of Contents

Introduction

Overview

The MetaStock® Developer's Kit includes applications and documentation for tools used to customize MetaStock and access MetaStock data.

The **Equis® Custom User Interface Utility** allows developers to add their own commands to the MetaStock Custom Toolbar, Tools menu and Help menu.

The **MetaStock Formula Organizer Enhancements** are installed with this toolkit. Documentation for these enhancements is included in this manual.

The **DDE Server** is provided with MetaStock Professional (versions 7.0 and later), and MetaStock FX. Full documentation for this feature is included in this manual.

The **MetaStock External Function (MSX) Application Programming Interface (API)** allows software developers to dynamically add externally defined functions to the MetaStock Formula Language. These functions can be called from Custom Indicators, Explorations, System Tests, and Experts. This feature is available in MetaStock, MetaStock Professional (versions 7.0 and later), as well as MetaStock FX.

The MetaStock **File Library (MSFL) Application Programming Interface (API)** provides developers with the tools necessary to integrate applications with the MetaStock data format.

This manual contains the instructions necessary to implement each of these applications. It assumes that the developer is an experienced programmer familiar with security price data and with creating and calling dynamic link libraries.

Typography Conventions

The following typographic conventions are used throughout this manual:

Typeface	Significance
`Monospaced type`	Program code.
`Italic Monospace`	Filenames
Sans Serif	Text in the program interface (**Bold** indicates buttons)
ALL CAPS	Mnemonics such as error codes, messages and defined values.
Italics	Function names, variable names, structure names or identifiers. (This type is also used to emphasize certain words.)

System Requirements

- Windows NT 4.0 (Service Pack 6a or higher)/
  Windows 2000 (Service Pack 1 or higher)/
  Windows XP
- 166 MHz Pentium CPU
- 32 MB of RAM
- 50 MB of free hard disk space

Setup

1. Insert the Program CD into your drive. The setup program should start automatically. If the auto-run feature of Windows isn't enabled on your system,
   a. Click **Start** and choose **Run**.
   b. Type “D:\SETUP.EXE” in the **Open** box and click **OK**.
   (“D” represents the letter assigned to your CD-ROM drive. If your drive is assigned a different letter, use it instead of “D”.)
2. Follow the on-screen instructions. You will be prompted to enter a Setup Key. Your Setup Key is found on the back of the CD case.

Supported Compilers

The MetaStock Developer’s Kit supports the following compilers:

- Borland® C++ Builder (Versions 5.0 and above)
- Borland® Delphi™ (Versions 3, 4 and 5)
- GCC 2.9.5.2
- Microsoft Visual C++™ (Versions 4.0 and above)
- Microsoft Visual Basic™ 6.0
- PowerBASIC®/DLL Version 6

Installed Files

After installing, the following directories and files should exist in the installation directory. Below is a short description of each file and directory.

Directory/File	Description
`UI`	Contains *`eqcustui.exe`* (the Equis Custom User Interface application) and *`ReadMe.Doc`* which contains information provided since the printing of this manual.
`MSFL\DATA`	This directory contains sample MetaStock price data.
`MSFL\DLL`	
`msfl91.dll`	The release version of the MSFL DLL.
`msfl91d.dll`	The debug version of the MSFL DLL.
`MSFL\DEF`	
`msfl.def`	The module definition file for the MSFL DLLs
`MSFL\INCLUDE`	
`msfl.h`	The C header file containing the MSFL defines, structures and function prototypes.
`msflutil.h`	The prototypes for a small collection of helpful C++ functions.
`msflutil.cpp`	A small collection of helpful C++ functions.
`msfl.bas`	The Microsoft Visual Basic module containing the MSFL function and type declares.
`msflutil.bas`	A Microsoft Visual Basic module containing helpful routines.
`msfl.pas`	The Delphi unit containing the MSFL constants, records and function declarations.
`msfl.inc`	The PowerBASIC module containing the MSFL function and type declares.
`MSFL\LIB\BC`	
`msfl91.lib`	The release link library for Borland C++ Builder 4 compiler.
`msfl91d.lib`	The debug link library for Borland C++ Builder 4 compiler.
`MSFL\LIB\GCC`	
`msfl91.lib`	The release link library for the gcc 2.9.5.2 compiler.
`msfl91d.lib`	The debug link library for the gcc 2.9.5.2 compiler.
`MSFL\LIB\VC`	
`msfl91.lib`	The release link library for Microsoft Visual C++ 6.0 compiler.
`msfl91d.lib`	The debug link library for Microsoft Visual C++ 6.0 compiler.
`MSFL\SAMPLES\BC`	The sample application for Borland C++ Builder 4 compiler.
`MSFL\SAMPLES\C-CONSOLE`	A C/C++ sample console application for the Borland C++ Builder 4, gcc 2.9.5.2, and Microsoft Visual C++ 6.0 compilers.
`MSFL\SAMPLES\DELPHI`	The sample application for Borland Delphi 4.
`MSFL\SAMPLES\PB`	The sample application for PowerBASIC/DLL 6.0.
`MSFL\SAMPLES\VB`	The sample application for Microsoft Visual Basic 6.0.
`MSFL\SAMPLES\VC`	The sample application for the Microsoft Visual C++ 6.0 compiler.
`MSX`	Contains *`MSXTest.exe`*, sample DLLs, sample data, and *`ReadMe.Doc`* containing additional information provided since the printing of this manual.
`MSX\C`	Samples and templates for Microsoft Visual C++ (Versions 4.0 and above), and Borland C++ (Versions 5.0 and above).
`MSX\DELPHI`	Samples and templates for Borland Delphi Versions 3, 4 and 5.
`MSX\PBasic`	Samples and templates for PowerBASIC/DLL Version 6.

Getting Help

Due to the complexity of the programming languages and development environments, Equis International is only able to provide minimal technical support for the MetaStock Developer's Kit. We will help you in understanding how to use the MetaStock Developer's Kit, but we cannot aid in writing or debugging your application or DLL.

This manual explains the use of the MetaStock Developer's Kit, but not the programming techniques required to effectively use it. The sample applications can be a good source of information as well as an excellent starting point.

CAUTION: Failure to follow the programming guidelines may result in the corruption of other MetaStock variables and/or loss of the user's data. Equis International shall not be responsible for any damages of any type caused by the MetaStock Developer's Kit.

Equis International is committed to enhancing the MetaStock Developer's Kit. If you are having a problem directly related to the Developer's Kit, you may contact Equis by mail or by the Internet.

By Mail

Equis International
MS Dev Kit Support
90 South 400 West, Suite 620
Salt Lake City, UT 84101

By Internet

msdevkit@equis.com

When contacting Equis by Internet, please include the module you are working with in the subject line of your e-mail message (For example, MSX, MSFL, etc.).

Modifying the MetaStock User Interface

Introduction

EqCustUI is a utility that can be used to customize the user interface in MetaStock. Specifically, it can be used to add and delete buttons on the MetaStock Custom toolbar. It can also be used to add and delete menu items in the reserved section of either the MetaStock Tools menu or the MetaStock Help menu.

Restrictions

Please note that the utility cannot be used to modify any other MetaStock toolbar or menu other than those listed above. It should also be noted that MetaStock must be installed on the user's machine before using the EqCustUI utility. If this is not done then the utility will not be able to locate the files to modify.

Using the EqCustUI

EqCustUI uses the command line for all input. Other than error messages, there is no user interface associated with this utility. This makes the utility useful for those third-party developers that use DOS batch files to install their MetaStock add-on products. For those third-party developers that use an installation program, please use the Windows API function `CreateProcess` to launch the EqCustUI utility.

IMPORTANT: EqCustUI locks the MetaStock files that it modifies while it is modifying them. For this reason, you must wait for the utility to finish its processing and exit before creating another instance. Failure to do so will produce undesirable results, as the second instance of EqCustUI will not be able to access the files it needs.

The following is a C/C++ example of how to use the `CreateProcess` function to modify the MetaStock user interface.

```
   BOOL                 bProcessCreated;
   STARTUPINFO          si;
   PROCESS_INFORMATION  pi;

   si.cb                = sizeof (si);
   si.lpReserved        = NULL;
   si.lpTitle           = NULL;
   si.dwFlags           = STARTF_USESHOWWINDOW;
   si.wShowWindow       = SW_SHOWNORMAL;
   si.cbReserved2       = 0;
   si.lpReserved2       = NULL;

// Spawn EqCustUI to modify the MetaStock user interface
   bProcessCreated      = CreateProcess (
                        _T("c:\\source\\equis apps\\eqcustui\\debug\\eqcustui.exe"),
                        _T("EqCustUI.exe \"Toolbar.Add(www.equis.com, Equis on the
                        web)\""),
                        NULL,
                        NULL,
                        FALSE,
                        NORMAL_PRIORITY_CLASS,
                        NULL,
                        NULL,
                        &si,
                        &pi);

   // Wait for EqCustUI to finish
   if (bProcessCreated)
   WaitForSingleObject (pi.hProcess, 5000);
```

Note: The command line must be enclosed in quotes (`" "`) if spaces are used. While this is optional if there are no spaces in the command line, it is recommended that the command line always be enclosed in quotes to avoid problems in the future.

The MetaStock user interface is modified by specifying an *object.command* pair on the utility's command line. The keywords *Toolbar* and *Menu* are the only objects accepted by the EqCustUI utility. The *Toolbar* object accepts the commands *Add* and *Delete*. The *Menu* object accepts the commands *AddItem*, *DeleteItem*, *AddPopupItem*, and *DeletePopupItem*. Objects and commands must be separated by a period. For example, *Toolbar.AddItem* is valid whereas *ToolbarAddItem* is not. The EqCustUI utility does not allow the third-party developer to specify or change the order of buttons and menu items.

IMPORTANT: The EqCustUI utility is *not* copied to the end user's computer when they install MetaStock. For this reason the EqCustUI utility must be included with each third-party solution that modifies the MetaStock user interface. Failure to do so will cause the third-party setup program to fail as it will not be able to modify the MetaStock user interface. It should also be noted that if the EqCustUI utility is temporarily copied to the end user's hard drive, it must be deleted when the third-party setup program finishes.

Commands

Toolbar.Add

`Toolbar.Add(<Command>,<Tip>[,<Parameters>])`

Parameters

Command	Specifies the action to be associated with the new button. Any syntax that can be specified in the Windows "Run" dialog (Start> Run) can be specified here. This includes executable files, internet URLs, and documents that are associated with a valid program on the system.
Tip	Specifies the status bar prompt and tooltip that will be associated with the new button. The pipe character (`|`) can be used to specify separate strings for the status bar prompt and the tooltip. In this case, the format of this parameter is `"status bar prompt string"|"tooltip string"`. If this parameter does not contain the pipe character then the same string will be used for both the status bar prompt and the tooltip.
Parameters	Specifies the parameter list that will be passed to the executable file when the user selects the new toolbar button. The [and] characters specify that this parameter is optional. These characters should not be used literally on the EqCustUI utility command line. For example, `Toolbar.Add(<Command>,<Tip>[,<Parameters>])` can be interpreted as `Toolbar.Add(<Command>,<Tip>)` or `Toolbar.Add(<Command>,<Tip>,<Parameters>)`. Either method is valid.

Remarks

- This adds a button to the MetaStock *Custom* toolbar.
- The EqCustUI utility always adds the button to the MetaStock toolbar using the icon associated with *Command*.
- There is no method for the developer to specify another icon to use, or to specify the order in which the button is placed on the custom toolbar.

Note: This command does not modify existing buttons as duplicates are ignored. To modify a button the developer must first delete it and then add it back in with the appropriate changes.

Example

```
EqCustUI "Toolbar.Add(www.mycompany.com,Browse our web
     site|mycompany.com)"
```

Or

```
EqCustUI "Toolbar.Add(dlwin.exe,Express download,/express)"
```

Toolbar.Delete

```
Toolbar.Delete(<Command>)
```

Parameters

Command Specifies the action (executable file, internet URL, document, etc.) that is associated with the button to delete.

Remarks

- This removes a button from the MetaStock *Custom* toolbar.
- The toolbar is searched until a button is found that is associated with this file.
- If a button is found it will be deleted.

IMPORTANT: Third-party developers should only delete the buttons they have created on the *Custom* toolbar. A third-party developer should *never* delete another developer's button on the toolbar. Doing so is a violation of the license agreement.

Example

```
    EqCustUI "Toolbar.Delete(www.mycompany.com)"
```

Menu.AddItem

```
Menu.AddItem(<Location>,<Menu>,<Command>)
```

Parameters

Location Specifies the placement of the new menu item. Please note that MetaStock uses two different and distinct menus. The first menu (Main) is used when a chart is *not* opened on the screen, whereas the second menu (Chart) is used when a chart *is* opened on the screen. For this reason menu items must be added to both the Main menu and the Chart menu to be visible at all times.
The following are the different locations available.

Placement	**Location** *(The menu is located:)*
`Main-Tools`	On the Tools menu when no chart is opened.
`Main-Help`	On the Help menu when no chart is opened.
`Chart-Tools`	On the Tools menu when a chart is opened.
`Chart-Help`	On the Help menu when a chart is opened.

Menu Specifies the string to be placed on the menu.

Command Specifies the action that is associated with this menu item. Any syntax that can be specified in the Windows run dialog (Start> Run) can be specified here. This includes executable files, internet URLs, and documents that are associated with a valid program on the system.

MetaStock supports several predefined literals that have special meaning. These literals will be replaced with the appropriate value when the user selects the menu item. The greater-than and less-than (< >) characters must be included.

The following are the literals supported by MetaStock.

Literal	Description
<symbol>	This literal is replaced with the symbol name of the security on the active chart. This literal is only valid when a chart is opened in MetaStock. Therefore, the *Location* parameter must be *Chart-Tools* or *Chart-Help*.
<name>	This literal is replaced with the actual name of the security on the active chart. This literal is only valid when a chart is opened in MetaStock. Therefore, the *Location* parameter must be *Chart-Tools* or *Chart-Help.*
<periodicity>	This literal is replaced with the periodicity of the security on the active chart. Please note that this is the periodicity of the underlying security as it was created with the MSFL, not the current periodicity that the user has compressed to. Valid values for this literal are *Intraday*, *Daily*, *Weekly*, *Monthly*, *Quarterly*, and *Yearly*. This literal is only valid when a chart is opened in MetaStock. Therefore, the *Location* parameter must be *Chart-Tools* or *Chart-Help*.

Remarks

- Adds a menu item to the reserved section of either the MetaStock **Tools** menu or the MetaStock **Help** menu.
- As of this writing, the reserved section on the **Tools** menu is directly above Default colors and styles, whereas on the **Help** menu it is directly above About MetaStock. The reserved sections are subject to change at any time.
- There is no method for the developer to specify the order in which the menu item is placed in the specified reserved section.
- MetaStock must be restarted for menu modifications to take effect.

Note: This command does not modify existing menu items. To modify a menu item the developer must first delete it and then add it back in with the appropriate changes.

Example

```
EqCustUI "Menu.AddItem(Main-Help,My company on the
     web,www.mycompany.com)"
```

Menu.DeleteItem

```
Menu.DeleteItem(<Location>,<Menu>,<Command>)
```

Parameters

Location Specifies the placement of the menu item.
Please note that MetaStock uses two different and distinct menus.
The first menu (Main) is used when a chart is *not* opened on the screen, whereas the second menu (Chart) is used when a chart *is* opened on the screen.

The following are the different locations available.

Placement	Location *(The menu is located:)*
`Main-Tools`	On the Tools menu when no chart is opened.
`Main-Help`	On the Help menu when no chart is opened.
`Chart-Tools`	On the Tools menu when a chart is opened.
`Chart-Help`	On the Help menu when a chart is opened.

Menu	Specifies the string of the item that is to be deleted.
Command	Specifies the action that is associated with the menu item to be deleted.

Remarks

- This removes a menu item from the reserved section of either the MetaStock **Tools** menu or the MetaStock **Help** menu.
- All three command parameters must be specified to be able to delete a menu item.
- MetaStock must be restarted for menu modifications to take effect.

IMPORTANT: Third-party developers should only delete the menu items that they have added.
A third-party developer should *never* delete another developer's menu item.
Doing so is a violation of the license agreement.

Example

```
EqCustUI "Menu.DeleteItem(Main-Help,My company on the
     web,www.mycompany.com)"
```

Menu.AddPopupItem

```
Menu.AddPopupItem(<Location>,<ParentMenu>,<Menu>,<Command>)
```

Parameters

Location Specifies the placement of the new menu item. Please note that MetaStock uses two different and distinct menus. The first menu (Main) is used when a chart is *not* opened on the screen, whereas the second menu (Chart) is used when a chart *is* opened on the screen. For this reason menu items must be added to both the Main menu and the Chart menu to be visible at all times.

The following are the different locations available.

Placement	**Location** *(The menu is located:)*
`Main-Tools`	On the Tools menu when no chart is opened.
`Main-Help`	On the Help menu when no chart is opened.
`Chart-Tools`	On the Tools menu when a chart is opened.
`Chart-Help`	On the Help menu when a chart is opened.

ParentMenu Specifies the name of the parent menu item. Multiple levels can be nested by using the forward slash (/) character. However, parent menus may *not* begin or end with the / separator character. For example, “My company/Support” would be a valid parent menu.

Menu Specifies the string to be placed on the menu.

Command Specifies the action that is associated with this menu item. Any syntax that can be specified in the Windows run dialog (Start> Run) can be specified here. This includes executable files, internet URLs, and documents that are associated with a valid program on the system. MetaStock supports several predefined literals that have special meaning. These literals will be replaced with the appropriate value when the user selects the menu item. The greater-than and less-than (< >) characters must be included.

The following are the literals supported by MetaStock.

Literal	Description
<symbol>	This literal is replaced with the symbol name of the security on the active chart. This literal is only valid when a chart is opened in MetaStock. Therefore, the *Location* parameter must be *Chart-Tools* or *Chart-Help*.
<name>	This literal is replaced with the actual name of the security on the active chart. This literal is only valid when a chart is opened in MetaStock. Therefore, the *Location* parameter must be *Chart-Tools* or *Chart-Help*.
<periodicity>	This literal is replaced with the periodicity of the security on the active chart. Please note that this is the periodicity of the underlying security as it was created with the MSFL, not the current periodicity that the user has compressed to. Valid values for this literal are *Intraday*, *Daily*, *Weekly*, *Monthly*, *Quarterly*, and *Yearly*. This literal is only valid when a chart is opened in MetaStock. Therefore, the *Location* parameter must be *Chart-Tools* or *Chart-Help*.

Remarks

- This creates a popup (nested) menu and its associated menu item(s).
- This command will create the parent menu item if it does not already exist.
- As of this writing, the reserved section on the **Tools** menu is directly above Default colors and styles, whereas on the **Help** menu it is directly above About MetaStock. The reserved sections are subject to change at any time.
- There is no method for the developer to specify the order in which the menu item is placed in the specified reserved section.
- MetaStock must be restarted for menu modifications to take effect.

Note: This command does not modify existing menu items. To modify a menu item the developer must first delete it and then add it back in with the appropriate changes.

Example

```
EqCustUI "Menu.AddPopupItem(Main-Help,My company,My company
     on the web,www.mycompany.com)"
```

Menu.DeletePopupItem

```
Menu.DeletePopupItem(<Location>,<ParentMenu>,<Menu>
   ,<Command>)
```

Parameters

Location Specifies the placement of the menu item.
Please note that MetaStock uses two different and distinct menus.
The first menu (Main) is used when a chart is *not* opened on the screen, whereas the second menu (Chart) is used when a chart *is* opened on the screen.

The following are the different locations available.

Placement	**Location** *(The menu is located:)*
`Main-Tools`	On the Tools menu when no chart is opened.
`Main-Help`	On the Help menu when no chart is opened.
`Chart-Tools`	On the Tools menu when a chart is opened.
`Chart-Help`	On the Help menu when a chart is opened.

ParentMenu	Specifies the name of the parent menu item. Multiple levels can be nested by using the forward slash (/) character. However, parent menus may *not* begin or end with the / separator character. For example, "My company/Support" would be a valid parent menu; while "My company/Support/" is not valid.
Menu	Specifies the string of the item that is to be deleted.
Command	Specifies the action that is associated with the menu item to be deleted

Remarks

- Removes a menu from a popup (nested) menu.
- If the popup menu is empty after the deletion it will be removed.
- All four command parameters must be specified to be able to delete a menu item.
- MetaStock must be restarted for menu modifications to take effect.

IMPORTANT: Third-party developers should only delete the menu items that they have added.
A third-party developer should *never* delete another developer's menu item.
Doing so is a violation of the license agreement.

Example

```
    EqCustUI "Menu.DeletePopupItem(Main-Help,My company,My
         company on the web,www.mycompany.com)"
```

Command Line Switches

The following is a list of command line switches that the EqCustUI utility supports.

`/h` **Help.** Displays a help screen. No other switches or commands are processed.

`/q` **Quiet.** Prevents the EqCustUI utility from displaying error messages.

Errors

The Windows API function *GetExitCodeProcess* can be used to retrieve the EqCustUI exit code. For batch files, the ERRORLEVEL command can be used to change program flow.

The following is a list of possible exit codes that can be returned by EqCustUI.

Returned Value	Meaning
0	The operation was successful. No error was encountered.
1	Cannot open the custom toolbar storage file. It is either locked or MetaStock is not installed.
2	Cannot open the custom menu storage file. It is either locked or MetaStock is not installed.
3	Out of memory.
4	Cannot add the button to the toolbar.
5	Cannot delete the button from the toolbar.
6	Cannot find the button on the toolbar.
7	Cannot add menu item.
8	Cannot delete menu item.
9	Cannot create popup (nested) menu.
10	Cannot open popup (nested) menu.
11	Cannot delete popup (nested) menu.
12	Cannot open the specified program, document, or internet resource.
13	Cannot read a file.
14	Cannot write to a file.

Formula Organizer Enhancements

Introduction

The Formula Organizer is a wizard (included with MetaStock versions 6.5 and above) that allows you to import and export any MetaStock formula-based files including custom indicators, system tests, explorations, and experts. For example, you can use the Formula Organizer to import a set of add-on custom indicators, experts, etc. purchased from a third-party. You could also create a set of add-on indicators, explorations, and so on to distribute to your colleagues, and even protect your formulas with a password.

Enhancements to the Formula Organizer to import and export DLLs created with the MetaStock External Functions API (MSX) were installed with this toolkit. Enhancements were also added to facilitate the distribution of formulas by providing copyright information, importing and exporting of templates, and creation of self-extracting installation files.

IMPORTANT: These enhancements are only available with the Formula Organizer included with MetaStock version 7.0 (any version) and above. You must have this toolkit ***and*** MetaStock 7.0 or above installed to export MSX DLLs and templates.

If a MetaStock user that has not installed this toolkit attempts to export formulas that call MSX DLLs, the user will be warned that anyone using these formulas must already have the DLLs installed. Importing MSX DLLs and templates is available to any user of MetaStock 7.0 or above. ***Importing MSX DLLs and templates does not require this toolkit, but does require MetaStock (any version) 7.0 or above.***

The following table indicates the capabilities of the currently released versions of Formula Organizer and the Developer's Kit.

Formula Organizer Function	without MDK	with MDK
Import custom indicators	✓	✓
Import system tests	✓	✓
Import explorations	✓	✓
Import experts	✓	✓
Import templates	✓	✓
Import MSX DLLs	✓	✓
Export custom indicators	✓	✓
Export system tests	✓	✓
Export explorations	✓	✓
Export experts	✓	✓
Option to include linked multimedia files with exported experts	✓	✓
Export Templates		✓
Export MSX DLLs		✓
Create self-extracting installs		✓

Formula Organizer Function	without MDK	with MDK
Password protect exported components	✓	✓
Password protect self-extracting install		✓
Copyright notice on self-extracting install		✓

CAUTION: Formula Organizer is not backward compatible, e.g. tools exported by the Formula Organizer cannot be imported by an earlier version of the Formula Organizer.
For example, if you export using 7.03, a user with 7.01 cannot import your file. If you export using 7.2, no user with 7.0*x* can import your file. The self-extracting installer will not launch any version of Formula Organizer previous to 7.0.

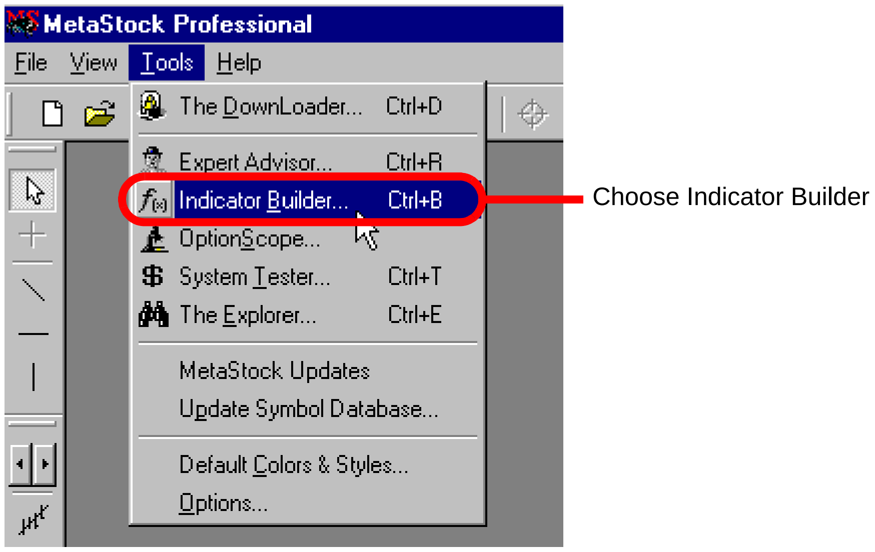
Using the Formula Organizer to Export

When the Developer's Kit is installed with MetaStock 7.0 or above, additional dialogs are included in the Formula Organizer's export process. You will be prompted to:

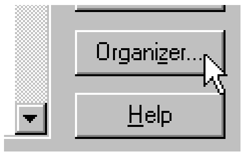
- Choose formulas to export,
- Choose templates to export,
- Choose DLLs to export,
- Include a copyright information text file,
- Create a self-extracting installation file,
- Password-protect the formulas, and
- Password-protect the self-extracting installation file.

To Export using the Formula Organizer

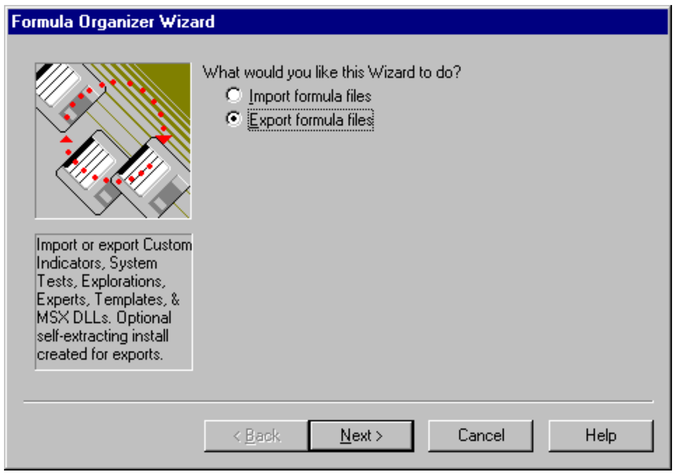
1. Start any one of MetaStock's formula tools (Indicator Builder, System Tester, The Explorer, Expert Advisor).
For example, select **Tools>** Indicator Builder to start the Indicator Builder.



2. Click the **Organizer** button.



3. Choose **Export formula files**, then click **Next**.

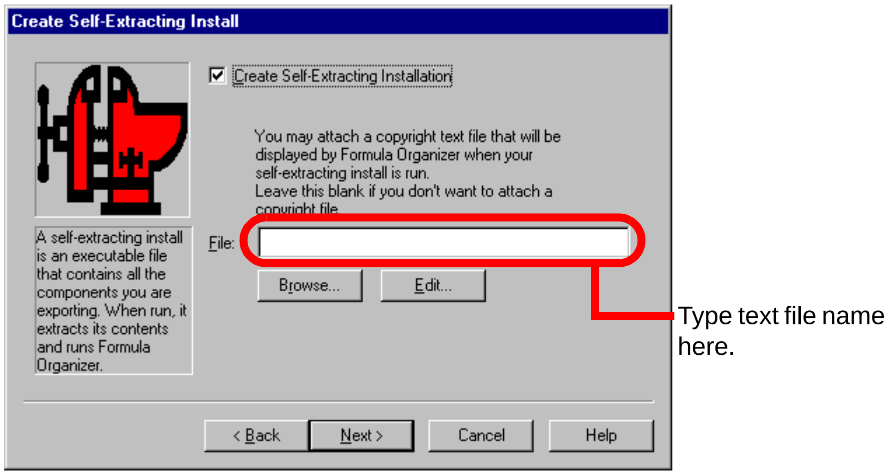

Formula Organizer Wizard

What would you like this Wizard to do?

Import formula files [✓ Export formula files]

Import or export Custom Indicators, System Tests, Explorations, Experts, Templates, & MSX DLLs. Optional self-extracting install created for exports.

4. Choose the Indicators, System Tests, Explorations, Experts, Templates, and DLLs to Export. After choosing each type of tool to export, click **Next** to go to the next selection dialog.
5. To create a self-extracting installation file, check Create Self-Extracting Installation. In this example, the name of this file will be *`FOSetup.exe`*, but you may change it after the export process is complete.


Create Self-Extracting Install

☑ Create Self-Extracting Installation

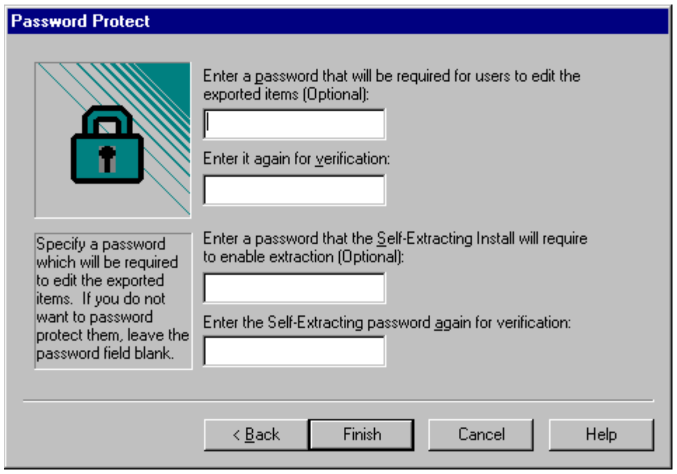
You may attach a copyright text file that will be displayed by Formula Organizer when your self-extracting install is run.
Leave this blank if you don't want to attach a copyright file.

File:

A self-extracting install is an executable file that contains all the components you are exporting. When run, it extracts its contents and runs Formula Organizer.

6. To include copyright information, or anything else that you would like to be displayed when the installation file is run, type the file name of the text file containing this information in the box shown above. This information is displayed after the extraction and before the Formula Organizer begins importing the formulas. The user is forced to click **OK** to proceed from the dialog displaying this text file.
7. Click the **Edit** button to launch your default text editor. You may edit an existing file, or create a new text file to include.
8. Click **Next** to continue the export process.
9. Type the folder where you want the installation file (or formula files if you did not choose to create an installation file) to be created. Click **Next** to continue.
10. Enter a password for the exported items, if desired.
    Users will be prompted for the formula password any time they attempt to view the formula for any of the tools you included in this export (for example, if a user selects **Tools>** Indicator Builder, and then selects one of the indicators you exported, the password prompt will appear when he or she clicks **Edit**).

Note: If you created a self-extracting installation file, you may enter a password for the file in this dialog. If you did not choose to create a self-extracting installation file, this option will not be displayed in the dialog. The user will be prompted for the installation password immediately after the installation file is run.


Password Protect

Enter a password that will be required for users to edit the exported items (Optional):

Enter it again for verification:

Specify a password which will be required to edit the exported items. If you do not want to password protect them, leave the password field blank.

Enter a password that the Self-Extracting Install will require to enable extraction (Optional):

Enter the Self-Extracting password again for verification:

11. Click **Finish** to complete the export process.

Using the Self-extracting Installation File

The self-extracting installation file that you created (`FOSetup.exe`) contains all the files that were exported. You may rename `FOSetup.exe` to any other name you wish (e.g., *`SuperTools.exe`*) using Windows Explorer. Be sure to leave the .exe extension.

When the user runs `FOSetup.exe`, a temporary folder is created, the files are extracted to it, and the user's copy of the Formula Organizer (`FormOrg.exe`) will be executed to import the files. If the self-extractor detects multiple versions of `FormOrg.exe` it will require the user to select the desired version to run. After the import, the temporary folder will be removed.

If any MSX DLLs are to be imported, they are copied to a temporary folder called "*`~MSXIMPORTDLLS~`*" located under the user's "External Function DLLs" folder. If MetaStock is running when FormOrg is finishing, FormOrg sends a signal to MetaStock to load the new DLLs. Otherwise, when MetaStock starts up it checks for MSX DLLs in this folder and, if any exist, they are moved to the "External Function DLLs" folder.

The password that can be applied to a self-extracting install allows you to distribute your tools via a web page or email. The compressed tools are encrypted and require the correct password to be extracted.

Installing the self-extracting installation file

Installation of the self-extracting installation file follows theis process. Once the process is started, only steps 2 and 5 require user interaction.

1. The system is searched for `formorg.exe`. If multiple versions are detected, the user is prompted to choose one.
2. If the installation file has been password protected, the user is prompted to enter the password.
3. A temporary folder is created to hold the installation files.
4. The installation files are unzipped into the temporary folders.
5. If a copyright file was included, the contents of that file are displayed. The user must click the **OK** button to proceed.
6. The formula files are imported by `formorg.exe`.
7. The temporary folders and their contents are removed.

DDE Data Interface

Overview

The Equis Dynamic Data Exchange Server (`EqDdeSrv.exe`) is installed as part of MetaStock Professional version 7.0 and above. EqDdeSrv forms a general interface bridge between the Equis Data Server, `EqDatSrv.exe`, and user applications. It allows any DDE client to receive security price information as a single snapshot (cold-link) or to receive price data changes as they occur (hot-link). One of the most common and easy-to-use DDE clients available to most users is a spreadsheet, such as Microsoft Excel. Programmers may choose to write their own DDE clients to interface with EqDdeSrv in order to perform specialized functions. Applications that use EqDdeSrv may be distributed to any MetaStock Professional user.

IMPORTANT: Never distribute `EqDdeSrv.exe` with your application. Your user will already have the correct version for his data vendor.

Background

Dynamic Data Exchange, or DDE, is a communication protocol, based on the messaging system built into Windows, which allows the transmission of messages between programs. A conversation is established between two programs, which then post messages to each other. In all cases, one of the programs is the *server* and the other is the *client*. The DDE server program has access to data that it can make available to the DDE client program. All DDE conversations are defined by a set of three character strings:

"Service" (also called "Application"), "Topic", and "Item" strings

The "Service" string is generally the DDE server file name, without the extension. The vendor who wrote the DDE server defines the legal values for the "Topic" and "Item" strings. The DDE client obtains data from the DDE server by specifying "Topic" and "Item" strings. For example, EqDdeSrv uses the "Topic" string to identify a security symbol, and the "Item" string to identify the specific price field for the specified security.

Implementation

EqDdeSrv depends upon EqDatSrv, the Equis Data Server which also accompanies MetaStock Professional. EqDatSrv is not a DDE server. It is intended to supply real-time data to MetaStock Professional, and is configured to work with the real-time data feed that the MetaStock customer has purchased. There are significant differences between data vendors and the way they supply data. EqDatSrv's job is to interface with a particular vendor and normalize the data feed into a reasonably standard format, hiding the significant differences from MetaStock Professional. The programmatic interface to EqDatSrv can be complex, and is limited to specific programming language constraints. EqDdeSrv, on the other hand, will work with any DDE client that sends through the correct character strings. When EqDdeSrv is started, it will in turn start up EqDatSrv (if necessary). EqDdeSrv is both a client and a server, in that it obtains data from EqDatSrv as a client, then provides that data as a DDE server.

Interface

The **Service**, or **Application**, string is always "**EQDDESRV**". **Topic** is generally the security or index symbol, with the exception of the reserved topic "**SYSTEM**". A discussion of the System topic appears later in this document (see page 20).

Note: Keep in mind that security and index symbols may be vendor-specific. If your client application contains hard-coded security or index symbols, it may be limited to working only with the data vendor you are using.)

Item is one of the following strings:

"OPEN"	`Opening price`
"HIGH"	`High price so far today`
"LOW"	`Low price so far today`
"LAST"	`Latest price today`
"PREVCLOSE"	`Previous trading day's closing (last) price`
"CHANGE"	`LAST - PREVCLOSE`
"TOTALVOL"	`Total volume today`
"YDTOTALVOL"	`Total volume yesterday (futures only)`
"TRADEVOL"	`Volume of last trade (this value will be 0 with some data vendors until the first trade occurs after requesting values for the specific security)`
"DATE"	`Last trade date (mm/dd/yyyy)`
"TIME"	`Last trade time (hh:mm)`
"OPENINT"	`Open interest (if applicable, otherwise 0)`
"BID"	`Bid`
"ASK"	`Ask`
"BIDSIZE"	`Bid size`
"ASKSIZE"	`Ask size`

Note: *EqDdeSrv* always returns price data as a formatted string (CF_TEXT type). With the exception of the Date and Time strings (mm/dd/yyyy and hh:mm), all strings are formatted as a floating point number with a maximum of four decimal places.

Running EqDdeSrv.exe

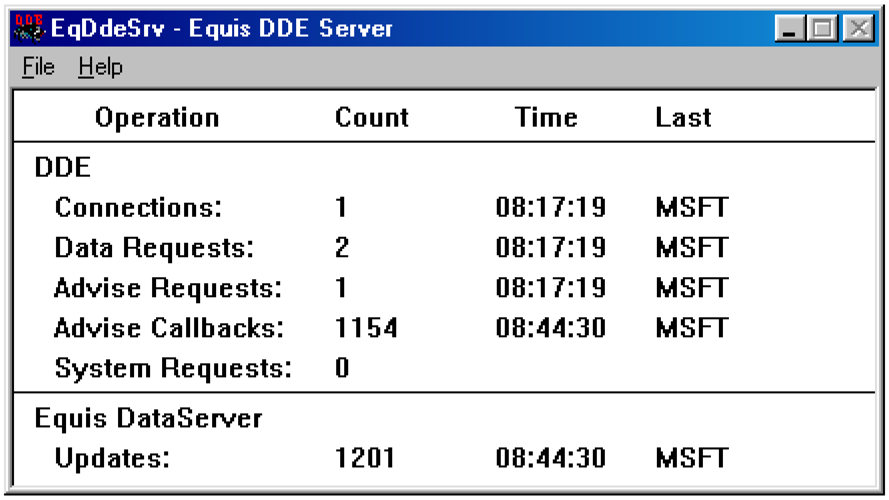
EqDdeSrv can be started by clicking on the "Equis DDE Server" shortcut in the Equis program folder. If you want your application to start *`EqDdeSrv.exe`*, you can locate it by examining the registry at: "`HKEY_CURRENT_USER\Software\Equis\Common`". Beginning with version 7.0 of MetaStock, there will be a registry key below "Common" for each version that is currently installed (i.e. "`7.0`"). Below the version number is a registry key "`File Paths`" which contains a string "`ProgramPath`".
The ProgramPath string indicates the folder where the MetaStock executables are located. Append the folder "`\`*`Servers`*" to the program path and verify the existence of *`EqDdeSrv.exe`*.

CAUTION: If there are multiple versions of MetaStock installed, your application should display a dialog with each of the versions and allow the user to pick the correct one.

Note: Before starting EqDdeSrv from your program, attempt to establish a connection with it to determine if it is already running. Arbitrarily running *`EqDdeSrv.exe`* when it is already running will cause EqDdeSrv to display its summary window.

After starting, EqDdeSrv will appear as an icon in the System Tray. A right-click with the mouse on the icon will present the following four menu choices: Open, Help, About, and Close.

Open causes EqDdeSrv to present a summary screen similar to this:



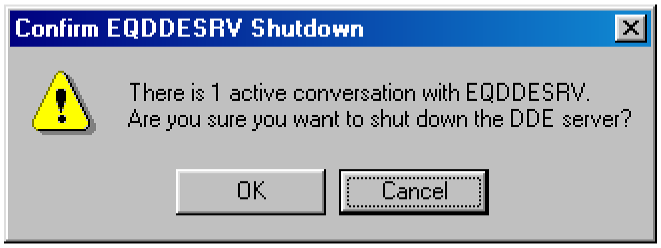
Operation	Count	Time	Last
DDE			
Connections:	1	08:17:19	MSFT
Data Requests:	2	08:17:19	MSFT
Advise Requests:	1	08:17:19	MSFT
Advise Callbacks:	1154	08:44:30	MSFT
System Requests:	0		
Equis DataServer			
Updates:	1201	08:44:30	MSFT

The summary screen shows a current count of several DDE functions, including the time of the last operation for each function and the symbol involved in the last operation. In addition, the summary screen shows the activity of the Equis Data Server, which is supplying data to the DDE server. All counts (with the exception of "Connections") and times can be reset using the Reset Counts menu option under File.

The entries in the "Operation" column are as follows:

Entry	Meaning
Connections	Number of active connections, or conversations with DDE clients. There is 1 active connection for each security requested by each client. For example, if a client is requesting updates for several price fields for only two securities, there will be two connections for that client represented in the "Count" column.
Data Requests	Number of cold-link requests that the DDE Server has received.
Advise Requests	Number of hot-link requests that the DDE Server has received.
Advise Callbacks	Number of hot-link updates that the DDE Server has processed.
System Requests	Number of "System" Topic requests already serviced by the DDE server. The last System function (Item) is displayed in the "Last" column.
EqDatSrv Updates	Number of updates that the DDE server has received from the Equis Data Server.

CAUTION: If you attempt to close EqDdeSrv while there are active conversations (DDE clients receiving data from EqDdeSrv), a warning similar to the following is displayed:



There is 1 active conversation with EQDDESRV.
Are you sure you want to shut down the DDE server?

System Topic

The “System” topic allows a DDE Client to obtain certain information about the DDE Server. `#define` entries for the standard system topics are included in the *`ddeml.h`* file that accompanies most Windows compilers. Although use of the `#define` is recommended, the actual string constants are presented here. Only the Topic and Item fields are shown in the table. The following System topics are supported by EqDdeSrv:

Topic	Item	Returned Data
System	Topics	Tab delimited list of all securities with active connections.
System	SysItems	Tab delimited list of all supported System Topics: “Topics”, “SysItems”, “Status”, “Formats”, and “TopicItemList”.
System	Status	EqDdeSrv always returns “Ready”.
System	Formats	EqDdeSrv always returns “Text”. CF_TEXT is the only format supported by EqDdeSrv.
Security Symbol	TopicItemList	**Note:** A security symbol is specified in the **Topic** field. Returned data is a tab delimited list of all the price fields that have an active advise request by any DDE Client. For example, if two DDE clients have hot-links to Microsoft stock, and the first is watching “LAST” and “TRADEVOL”, and the second is watching “LAST” and “OPEN”, the returned data would by a tab delimited string containing “OPEN”, “LAST”, and “TRADEVOL”.

Examples

Microsoft Excel Example.

Microsoft Excel has the ability to act as a general DDE client. You can specify a DDE hot-link in any cell by entering a formula of the form:

```
=Server|'Topic'!Item
```

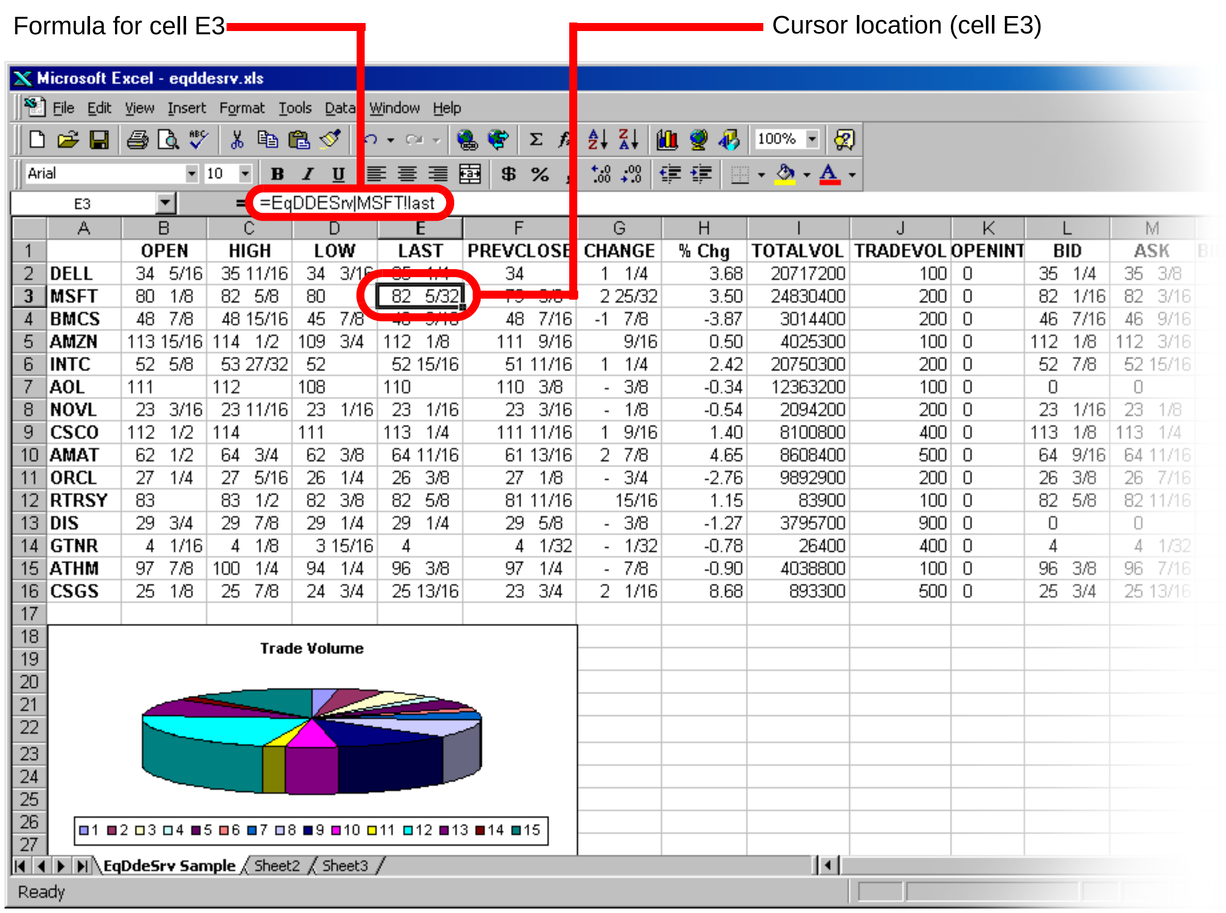
The server is separated from the topic by the vertical solid bar character, and the topic is separated from the item by an exclamation point. For example, to observe the constantly updated last price for Microsoft, you would enter the following formula in any cell:

```
=EQDDESRV|'MSFT'!LAST
```

Notes:

- The case of the topic is important only if the data vendor is case sensitive.
- The date and time strings are converted by Excel to Julian dates. You must apply Excel date and time formatting to view these fields in MM/DD/YYYY format.

The Excel screen shown on the next page has a DDE formula in each cell that is displaying a price value. The `%Chg` column is calculated from the LAST and PREVCLOSE columns. All values update in real time, and the pie chart at the bottom constantly reflects the changes in the TRADEVOL column. The cursor is on cell **E3**, and you can observe the formula in the Excel edit line:


Formula for cell E3
Cursor location (cell E3)
Microsoft Excel - eqddesrv.xls
File Edit View Insert Format Tools Data Window Help
Arial 10
E3
Trade Volume
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
EqDdeSrv Sample Sheet2 Sheet3
Ready

E3 =EqDDESrv|MSFT!last

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		OPEN	HIGH	LOW	LAST	PREVCLOSE	CHANGE	% Chg	TOTALVOL	TRADEVOL	OPENINT	BID	ASK
2	DELL	34 5/16	35 11/16	34 3/16	35 1/4	34	1 1/4	3.68	20717200	100	0	35 1/4	35 3/8
3	MSFT	80 1/8	82 5/8	80	82 5/32	79 3/8	2 25/32	3.50	24830400	200	0	82 1/16	82 3/16
4	BMCS	48 7/8	48 15/16	45 7/8	46 9/16	48 7/16	-1 7/8	-3.87	3014400	200	0	46 7/16	46 9/16
5	AMZN	113 15/16	114 1/2	109 3/4	112 1/8	111 9/16	9/16	0.50	4025300	100	0	112 1/8	112 3/16
6	INTC	52 5/8	53 27/32	52	52 15/16	51 11/16	1 1/4	2.42	20750300	200	0	52 7/8	52 15/16
7	AOL	111	112	108	110	110 3/8	- 3/8	-0.34	12363200	100	0	0	0
8	NOVL	23 3/16	23 11/16	23 1/16	23 1/16	23 3/16	- 1/8	-0.54	2094200	200	0	23 1/16	23 1/8
9	CSCO	112 1/2	114	111	113 1/4	111 11/16	1 9/16	1.40	8100800	400	0	113 1/8	113 1/4
10	AMAT	62 1/2	64 3/4	62 3/8	64 11/16	61 13/16	2 7/8	4.65	8608400	500	0	64 9/16	64 11/16
11	ORCL	27 1/4	27 5/16	26 1/4	26 3/8	27 1/8	- 3/4	-2.76	9892900	200	0	26 3/8	26 7/16
12	RTRSY	83	83 1/2	82 3/8	82 5/8	81 11/16	15/16	1.15	83900	100	0	82 5/8	82 11/16
13	DIS	29 3/4	29 7/8	29 1/4	29 1/4	29 5/8	- 3/8	-1.27	3795700	900	0	0	0
14	GTNR	4 1/16	4 1/8	3 15/16	4	4 1/32	- 1/32	-0.78	26400	400	0	4	4 1/32
15	ATHM	97 7/8	100 1/4	94 1/4	96 3/8	97 1/4	- 7/8	-0.90	4038800	100	0	96 3/8	96 7/16
16	CSGS	25 1/8	25 7/8	24 3/4	25 13/16	23 3/4	2 1/16	8.68	893300	500	0	25 3/4	25 13/16

Simple C Example

```
/* EqDDeDemo - This DDE client is a short example of how to obtain price
   data from the Equis DDE Server (EQDDESRV.EXE).
*/

#include <windows.h>
#include <ddeml.h>
#define WM_USER_INIT_DDE (WM_USER + 1) // Event to initialize DDE

// Window callback
LRESULT  CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

// DDE callback
HDDEDATA CALLBACK DdeCallback (UINT, UINT, HCONV, HSZ, HSZ, HDDEDATA, DWORD, DWORD);
DWORD idInst; // global program instance
HCONV hConv;  // global handle to DDE conversation
HWND  hWnd;   // global handle to window

char  szValue[20]; // receives values obtained from DDE server
char  szAppName[] = "EqDdeDemo";

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpCmdLine,
                   int       nCmdShow)
{
    MSG         Msg;
    WNDCLASSEX  WndClass;

    strcpy(szValue, "<wait>"); // Initialize value string

    // Fill the Wind Class structure
    WndClass.cbSize        = sizeof(WndClass);
    WndClass.style         = CS_HREDRAW | CS_VREDRAW;
    WndClass.lpfnWndProc   = WndProc;
    WndClass.cbClsExtra    = 0;
    WndClass.cbWndExtra    = 0;
    WndClass.hInstance     = hInstance;
    WndClass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    WndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    WndClass.lpszMenuName  = NULL;
    WndClass.lpszClassName = szAppName;
    WndClass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    RegisterClassEx (&WndClass);

    // hard-code a small window size & location for demo purposes
    hWnd = CreateWindow (szAppName, "Equis DDE Client Demo",
                         WS_OVERLAPPEDWINDOW,
                         100, 100, 250, 70,
                         NULL, NULL, hInstance, NULL);

    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);

    // Initialize the DDEML library
    if (DdeInitialize (&idInst, (PFNCALLBACK) &DdeCallback,
                       APPCLASS_STANDARD | APPCMD_CLIENTONLY, 0L))
    {
        MessageBox (hWnd, "Unable to initialize DDE client.",
                    szAppName, MB_ICONEXCLAMATION | MB_OK);
        DestroyWindow(hWnd);
        return FALSE;
    }

    // Start the DDE conversation
    SendMessage(hWnd, WM_USER_INIT_DDE, 0, 0L);

    while (GetMessage(&Msg, NULL, 0, 0))
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
DdeUninitialize(idInst);
   return Msg.wParam;
```

```
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    static char  szService[] = "EQDDESRV";
    static char  szTopic[]   = "MSFT"; // Microsoft stock;
    static char  szItem[]    = "LAST";

    switch (nMsg)
    {
    case WM_USER_INIT_DDE:
        {
            HSZ         hszService;
            HSZ         hszTopic;
            HSZ         hszItem;
            HDDEDATA    hData;

            // Try to connect to DDE server
            hszService = DdeCreateStringHandle (idInst, szService, 0);
            hszTopic   = DdeCreateStringHandle (idInst, szTopic, 0);
            hConv      = DdeConnect (idInst, hszService, hszTopic, NULL);

            if (hConv == NULL)
            {
                // Server isn't loaded - try to load it...
                WinExec (szService, SW_SHOWMINNOACTIVE);
                hConv = DdeConnect(idInst, hszService, hszTopic, NULL);
            }

            // Finished with the service and topic string handles
            DdeFreeStringHandle (idInst, hszService);
            DdeFreeStringHandle (idInst, hszTopic);

            if (hConv == NULL) // Couldn't start the server!
            {
                MessageBox(hWnd,
                           "Unable to connect with EqDdeSrv.exe",
                           szAppName, MB_ICONEXCLAMATION | MB_OK);
                return 0;
            }

            // Request current value of MSFT "Last" price (cold-link).
            // This is a synchronous snap-shot of data.
            hszItem = DdeCreateStringHandle (idInst, szItem, 0);
            hData = DdeClientTransaction(NULL, 0, hConv, hszItem,
                                         CF_TEXT, XTYP_REQUEST, 3000, NULL);
            if (hData != NULL)
            {
                DdeGetData(hData, (unsigned char *) szValue, sizeof(szValue), 0);
                InvalidateRect(hWnd, NULL, FALSE);
                DdeFreeDataHandle(hData);
            }

    // Request notification of changes in MSFT "Last" price (hot-link).
    // This will cause XTYP_ADVDATA events to be sent to the dde callback function
    // each time the "Last" value changes.
            DdeClientTransaction(NULL, 0, hConv, hszItem, CF_TEXT,
                                 XTYP_ADVSTART | XTYPF_ACKREQ, 3000, NULL);
            DdeFreeStringHandle (idInst, hszItem);
        }
        return 0;
    case WM_PAINT:
        {
            HDC          hDc;
            PAINTSTRUCT  ps;
            char         szBuf[100];

            hDc = BeginPaint (hWnd, &ps);
            TextOut(hDc, 10, 10, szBuf,
                    wsprintf (szBuf, "Microsoft Last Value:  %s",  szValue));
            EndPaint(hWnd, &ps);
        }
        return 0;
    case WM_CLOSE:
        if (hConv != NULL)
        {
            HSZ hszItem;
```

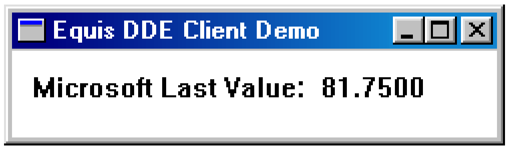
```
                hszItem = DdeCreateStringHandle (idInst, szItem, 0);
                DdeClientTransaction (NULL, 0, hConv, hszItem, CF_TEXT, XTYP_ADVSTOP,
                                         3000, NULL);
                DdeFreeStringHandle (idInst, hszItem);
                DdeDisconnect (hConv);
            }
            break;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
        default:
            break;
        }
        return DefWindowProc (hWnd, nMsg, wParam, lParam);
    }

    HDDEDATA CALLBACK DdeCallback (UINT nType, UINT nFmt, HCONV hConv,HSZ hSz1, HSZ hSz2,
                                      HDDEDATA hData, DWORD dwData1, DWORD dwData2)
    {
        HDDEDATA Rtrn;

        switch (nType)
        {
        case XTYP_ADVDATA:
            if (nFmt == CF_TEXT)
            {
                /*
                // You could get the name of the item being updated
                //   if you wish by calling DdeQueryString:
                char szItem[20];
                DdeQueryString (idInst, hSz2, szItem, sizeof(szItem), 0);
                */
                // Get the actual data
                DdeGetData(hData, (unsigned char *) szValue, sizeof(szValue), 0);
                InvalidateRect(hWnd, NULL, FALSE);
                Rtrn = (HDDEDATA) DDE_FACK;
            }
            else
                Rtrn = (HDDEDATA) DDE_FNOTPROCESSED;
            break;
        case XTYP_DISCONNECT:
            hConv = NULL;
            Rtrn = NULL;
            break;
        default:
            Rtrn = NULL;
            break;
        }

        return Rtrn;
    }
```

The output of the preceding program is a small window similar to the following, with a constantly updated price value:



Microsoft Last Value: 81.7500

Suggested Resources

- *Programming Windows 95* by Charles Petzold, Microsoft Press
- *C/C++ User's Journal*, August 1998, “Encapsulating DDE” by Giovanni Bavestrelli (Presents an excellent C++ framework for developing DDE applications)
- Microsoft Developers Network.

MetaStock External Functions (MSX)

Introduction

This chapter explains the use of the MSX API and provides several examples of how to use it correctly.   It contains the instructions necessary to implement MSX DLLs, and assumes that the developer is an experienced programmer familiar with security price data and with creating dynamic link libraries.

The MetaStock External Function (MSX) Application Programming Interface (API) allows software developers to dynamically add externally defined functions to the MetaStock Formula Language. This feature is available in all releases of MetaStock above (and including) version 7.0.

When MetaStock initializes, it scans a pre-defined folder, looking for any DLLs that correctly implement the MSX API. When an MSX DLL is found, the functions that it implements are automatically added to the MetaStock Formula Language.

These new functions can be used to create Custom Indicators, Explorations, System Tests and Expert Advisors using MetaStock's formula tools.

The MSX API supports any programming language that meets the following criteria:

- Exports DLL functions by name
- Supports the Windows `stdcall` stack frame convention
- Creates 32-bit DLLs for Windows 2000, Windows XP, or Windows NT version 4.0 or greater (commonly called a Win32 DLL)

Note: Microsoft Visual Basic does not have the capability to produce a Win32 DLL. Therefore, MSX DLLs cannot be written in Microsoft Visual Basic.   A good alternative for VB programmers is PowerBASIC, an inexpensive compiled Basic that is syntax-compatible with VB and can produce Win32 DLLs.

MSX DLL Capabilities

The functions that can be implemented in MSX DLLs are similar in behavior to the standard built-in MetaStock functions. In other words, MSX functions can be written to perform calculations based on any available price data or results of other functions. All MSX DLL functions return a data array. This exactly parallels the behavior of the MetaStock built-in functions. The returned data array can be plotted by Custom Indicators or used in any way that a standard built-in function can be used.

MSX DLLs can perform calculations of virtually unlimited complexity. You have the full power of conventional programming languages like C or Pascal with all of their logic, data manipulation and rich flow-control capabilities.

Things that you *can* do with an MSX DLL

Things that you ***can*** do with an MSX DLL include:

- Implement functions not provided with MetaStock.
- Perform complex calculations on price data.
- Provide multiple functions in a single MSX DLL.
- Access stored MetaStock price data using MSFL (included in the MetaStock Developer’s Kit — see the MSFL documentation later in this manual for details).

- Create functions that can be used by Custom Indicators, System Tests, Explorations, and Experts.
- Distribute your compiled MSX DLL to other users.

Things that you *cannot* do with an MSX DLL

Things that you ***cannot*** do with an MSX DLL include:

- Manipulate GUI functions, including plotting and user dialogs.
- Access the standard MetaStock built-in functions from within your DLL.

Getting Assistance

Due to the complexity of programming languages and development environments, Equis is able to provide only minimal technical support for the MSX API.
We will help with understanding how to use the MSX API, but we cannot aid in writing or debugging your DLL.

Important Notes

- This manual explains the use of the MSX API and provides several examples of how to use it correctly.
- It is imperative that you read this entire chapter, in the order presented, to successfully create an MSX DLL.
- It is essential that you follow all specified programming guidelines and API requirements. External DLLs receive pointers to data structures allocated by MetaStock. Failure to follow the MSX programming guidelines may result in your DLL modifying memory outside the boundaries of the defined data structures, potentially corrupting other MetaStock variables and causing loss of the user’s data.

Note: Equis shall not be responsible for any damages of any type caused by MSX DLLs.

For more information on Technical Support for the MetaStock Developer’s Kit, see page 4.

Overview

MetaStock will automatically recognize and load any MSX DLL that exists in the “**External Function DLLs**” folder, which is a subfolder of the MetaStock system folder.

An MSX DLL implements one or more external functions that can be called from within any formula in MetaStock. In order to implement an external function, an MSX DLL must perform two basic tasks:

- Define the function syntax including the function name, number of arguments, and argument types.
- Calculate the results of the function when it is called and return those results to MetaStock.

Each external function has a unique name that identifies the function within a MetaStock formula. The syntax for each function can define up to nine arguments that supply numeric data for calculations or control the behavior of the calculation.

MetaStock users call external functions from within formulas by using the External Formula function:

```
ExtFml("DLL Name.Function Name",arg1,…,argn)
```

As an example, if an MSX DLL named *MyDLL* implements a function called *MyFunction,* which accepts a single price data argument, the function can be called from any MetaStock formula by the following:

```
ExtFml("MyDLL.MyFunction", close)
```

MSX DLLs export two to four initialization functions by name. These functions are used by MetaStock to query the DLL about the functions that are implemented in the DLL. While the DLL initialization functions themselves are rigidly defined by the MSX API, the external functions that they define are extremely flexible.

As mentioned earlier, external functions have names that are defined by the MSX DLL and can have up to nine arguments. Each argument defined for a function can be one of four types:

- **Data Arrays** (e.g., Open, High, Low, Close, etc., or the results of another function)
- **Numeric Constants** (e.g., 10, 20, -50, etc.)
- **String Constants** (e.g., “Hello World”, etc.)
- **Customized sets** (e.g., SIMPLE, EXPONENTIAL, etc.)

Details and examples of how a DLL defines these arguments are presented later in this document.

Extreme care must be taken by the programmer to ensure that all MSX DLL functions are well-behaved. Because they are called directly by MetaStock, any fatal exception caused by an MSX DLL function will affect MetaStock — possibly causing a forced shutdown and loss of user data. MetaStock attempts to trap all common exceptions, but authors of MSX DLLs should not rely on having their exceptions handled by MetaStock. Any serious exception that MetaStock traps will cause the DLL containing the offending function to be detached and the external functions it contains will not be available to the MetaStock user.

Function Prototype Section

The functions defined in an MSX DLL fall into two categories: *Initialization* Functions and *Calculation* (or External) Functions. Initialization functions are called by MetaStock during startup to determine what external functions are available and what arguments they require. Calculation functions are the functions that are available to MetaStock users who use your MSX DLL.

Initialization Functions

MSXInfo

This is the first function called in an MSX DLL. It returns basic information about the DLL and verifies that it is a valid MSX DLL. This function is required and will always be called during initialization.

C

```
BOOL __stdcall MSXInfo (MSXDLLDef *a_psDLLDef)
```

Delphi Pascal

```
function MSXInfo (var a_psDLLDef : MSXDLLDef)
    : LongBool; stdcall;
```

PowerBASIC/DLL

```
FUNCTION MSXInfo SDECL ALIAS “MSXInfo” ( _
    a_psDLLDef AS MSXDLLDef PTR) EXPORT AS LONG
```

Parameters

a_psDLLDef	Pointer to the *MSXDLLDef* structure to be filled with copyright, number of external functions, and MSX version. See page 32.

Return Values

- MSX_SUCCESS if successful
- MSX_ERROR for internal error

MSXNthFunction

This function is called once during initialization for each external function specified by the *MSXInfo* call. See *"MSXInfo"* (page 27) for more details on using this function.

C

```
BOOL __stdcall       MSXNthFunction (int a_iNthFunc,
                     MSXFuncDef *a_psFuncDef)
```

Delphi Pascal

```
function MSXNthFunction (a_iNthFunc: Integer;
                     var  a_psFuncDef: MSXFuncDef)
                     :LongBool; stdcall;
```

PowerBASIC/DLL

```
FUNCTION MSXNthFunction SDECL ALIAS "MSXNthFunction" ( _
       BYVAL  a_iNthFunc as LONG, _
              a_psFuncDef AS MSXFuncDef PTR) EXPORT AS LONG
```

Parameters

a_iNthFunc	The zero-based index indicating which function's information is requested.
a_psFuncDef	Pointer to the *MSXFuncDef* data structure (page 32) to be filled in with external function information..

Return Values

- MSX_SUCCESS if successful
- MSX_ERROR for internal error

MSXNthArg

This function is called once during initialization for each argument specified for each external function. If none of the external functions have arguments this function will not be called and is not required.

C

```
BOOL __stdcall MSXNthArg ( int a_iNthFunc,
                           int a_iNthArg,
            MSXFuncArgDef *a_psFuncArgDef)
```

Delphi Pascal

```
Function MSXNthArg (a_iNthFunc: Integer;
                    a_iNthArg: Integer;
            var     a_psFuncArgDef: MSXFuncArgDef)
: LongBool; stdcall;
```

PowerBASIC/DLL

```
FUNCTION MSXNthArg SDECL ALIAS "MSXNthArg" _
          BYVAL      a_iNthFunc AS LONG, _
          BYVAL      a_iNthArg AS LONG, _
                     a_psFuncArgDef AS MSXFuncArgDef PTR)
          EXPORT AS LONG
```

Parameters

a_iNthFunc	The zero-based index indicating which function's information is requested.
a_iNthArg	The zero-based index indicating which argument of the specified function's information is requested.

a_psFuncArgDef	Pointer to the *MSXFuncArgDef* data structure (page 33) to be filled in with external function argument information.

Return Values

- MSX_SUCCESS if successful
- MSX_ERROR for internal error

MSXNthCustomString

This function is called once during initialization for each Custom Argument variation specified for each external function. If none of the external functions have custom arguments this function will not be called and is not required.

C

```
BOOL
__stdcall
MSXNthCustomString (  int a_iNthFunc,
                      int a_iNthArg,
                      int a_iNthString,
          MSXFuncCustomString *a_psCustomString)
```

Delphi Pascal

```
function
MSXNthCustomString (  a_iNthFunc:  Integer;
                      a_iNthArg:   Integer;
                      a_iNthString:Integer;
          var         a_psCustomString: MSXFuncCustomString)
                      : LongBool; stdcall;
```

PowerBASIC/DLL

```
FUNCTION MSXNthCustomString SDECL ALIAS “MSXNthCustomString” ( _
    BYVAL  a_iNthFunc AS LONG, _
    BYVAL  a_iNthArg AS LONG, _
    BYVAL  a_iNthString AS LONG, _
           a_psCustomString as MSXFuncCustomString PTR) _
    EXPORT AS LONG
```

Parameters

a_iNthFunc	The zero-based index indicating which function’s information is requested.
a_iNthArg	The zero-based index indicating which argument of the specified function’s information is requested.
a_iNthString	The zero-based index indicating which string of the custom argument of the specified function is requested.
a_psCustomString	Pointer to the *MSXFuncCustomString* data structure (page 34) to be filled in with external function custom argument information.

Return Values

- MSX_SUCCESS if successful
- MSX_ERROR for internal error

Calculation Functions

All external calculation functions have the following prototype:

C

```
BOOL
__stdcall
<FuncName> (const MSXDataRec      *a_psDataRec,
    const  MSXDataInfoRecArgsArray *a_psDataInfoArgs,
    const  MSXNumericArgsArray     *a_psNumericArgs,
    const  MSXStringArgsArray      *a_psStringArgs,
    const  MSXCustomArgsArray      *a_psCustomArgs,
           MSXResultRec            *a_psResultRec)
```

Delphi Pascal

```
function
<FuncName> (const a_psDataRec:    PMSXDataRec;
    const  a_psDataInfoArgs:       PMSXDataInfoRecArgsArray;
    const  a_psNumericArgs:        PMSXNumericArgsArray;
    const  a_psStringArgs:         PMSXStringArgsArray;
    const  a_psCustomArgs:         PMSXCustomArgsArray;
    var    a_psResultRec:          MSXResultRec)
: LongBool; stdcall;
```

PowerBASIC/DLL

```
FUNCTION
<FuncName>             SDECL ALIAS “<FuncName>” _
    (a_psDataRec      AS MSXDataRec PTR, _
     a_psDateInfoArgs AS MSXDataInfoRecArgsArray PTR, _
     a_psNumericArgs  AS MSXNumericArgsArray PTR, _
     a_psStringArgs   AS MSXStringArgsArray PTR, _
     a_psCustomArgs   AS MSXCustomArgsArray PTR, _
     a_psResultRec    AS MSXResultRec PTR) EXPORT AS LONG
```

Note: **<FuncName>** is the name of your function. The name listed in the EXPORTS section or ALIAS string of your code and the name returned by the *MSXNthFunction* (page 28) must exactly match the spelling and case of this function name.

Parameters

a_psDataRec	The read-only data structure that contains all available price data and security details. This structure is always passed to all calculation functions, regardless of their defined argument lists. (page 36).
a_psDataInfoArgs	The read-only data array arguments expected by the function. (page 38).
a_psNumericArgs	The read-only Numeric (*float*) arguments expected by the function. (page 39).
a_psStringArgs	The read-only String arguments expected by the function. (page 39).
a_psCustomArgs	The read-only custom argument ID’s expected by the function. (page 39).
a_psResultRec	A data structure containing the data array that your function will fill with data to be returned to MetaStock. Be sure to set both *iFirstValid* and *iLastValid* in *a_psResultRec->psResultArray* before returning from your function.

Cautions:

- Do *not* write values to the *a_psResultRec->psResultArray->pfValue* array beyond the index value of *a_psDataRec->sClose.iLastValid*. Writing beyond that point *will* corrupt MetaStock system memory, and may cause a loss of user data.
- MetaStock does not support *iLastValue* indexes greater than the iLastValue index of the ‘Close’ data array. See page 58 for more details.

Return Values

- MSX_SUCCESS if successful
- MSX_ERROR for internal error

Data Types

The MSX API defines several data types and structures, which are used to transfer information between MetaStock and MSX DLLs.

Formats

This section is an overview of the different data types.

Dates

Dates are of type *long* and are stored in a year, month, day format (YYYYMMDD) with the year being a four digit year (e.g. January 15, 1997 is stored in a *long* as 19970115). Valid dates range from January 1, 1800 to December 31, 2200.

Times

Times are of type *long* and are stored in hour, minutes, tick order (HHMMTTT) (e.g. 10:03:002 is stored in a *long* as 1003002). Times are always in twenty-four hour format. Notice that the last segment of the time is not seconds, but ticks. MetaStock uses a tick count instead of seconds to handle the case of multiple ticks per second without duplication of records. The first tick of a minute is 000, the second is 001, and so on, up to 999. If more than 1000 ticks occur in any given minute, the 999 tick count will repeat until the minute is reset. This condition is not an error. Valid times range from 00:00:000 to 23:59:999.

Strings

Strings are of type *char* and are stored as null (zero) terminated char sequences. Care must be taken when writing to any strings passed into your DLL that characters are not written beyond the defined string boundaries. All writeable strings in MSX structures are of MSX_MAXSTRING size. MSX_MAXSTRING is defined in the *`MSXStruc.h`* (*`MSXStruc.inc`* for Delphi, *`MSXStruc.bas`* for PowerBASIC) file. When writing data to a string be sure to include the null byte at the end.

CAUTION: ***Writing beyond the defined string boundaries may cause an unrecoverable exception that could result in loss of user data.***

Variable Notation

The MSX data structures, function prototypes, and source code templates use a form of Hungarian notation to designate the type of each variable or structure member. The type prefixes each variable name. A list of the notations used by the MSX API follows:

Notation	Type	Description
`c`	char	Character, an 8-bit signed value.
`f`	float	Single precision, 32-bit floating point number.
`lf`	double	Double precision (long float) 64-bit floating point number.
`i`	int	Integer, a 32-bit signed value. (PowerBASIC integers are 16-bit.)
`l`	long	Long integer, a 32-bit signed value.
`p`	All	Pointer, a 32-bit address.
`s`	All	Structure, a user defined type.
`sz`	char	Null terminated string of signed characters.
`b`	BOOL	Boolean int (32-bit) values.
`l_`	All	Local variables (defined inside function).
`a_`	All	Arguments (passed into functions).

Initialization Structures

The following structures are used to communicate between MetaStock and the MSX API initialization functions. These structures allow the MSX DLL to give MetaStock information regarding function names and function syntax information.

Note: All data structure examples are shown using "C" syntax, and are found in the *`MSXStruc.h`* file included with the MetaStock Developer's Kit. Corresponding data structure definitions for Delphi Pascal and PowerBASIC/DLL can be found in the *`MSXStruc.pas`* and *`MSXStruc.bas`* files, respectively.

MSXDLLDef structure

This structure contains fields that define the DLL copyright, the number of external functions exported by the DLL, and the MSX version number. It is used exclusively by the *MSXInfo* function. See *"MSXInfo"* (page 27) for more details on using this function.

```
typedef struct
{
     char  szCopyright[MSX_MAXSTRING];
     int   iNFuncs;
     int   iVersion;
} MSXDLLDef;
```

Parameters

szCopyright	A copyright or other information about this DLL should be copied into this string. Care must be taken not to write more than MSX_MAXSTRING characters.
iNFuncs	The number of external functions exported by this DLL.
iVersion	The MSX version number. This should be set to the constant MSX_VERSION.

MSXFuncDef structure

This structure describes the attributes of an external function that will be exported by the DLL for use by the MetaStock Formula Language. It is used exclusively by *MSXNthFunction* function. See "MSXNthFunction" on page 28 for more details on using this function.

```
typedef struct
{
     char  szFunctionName[MSX_MAXSTRING];
     char  szFunctionDescription [MSX_MAXSTRING];
     int   iNArguments;
} MSXFuncDef;
```

Parameters

szFunctionName	The exported name of the external function. This is the function name that will be used in the *ExtFml()* call by the MetaStock user.
Note:	This name must *exactly match* the spelling and case used in the EXPORTS or ALIAS section of your code (see example programs). This is the name used by the *GetProcAddress* system call to obtain the address of this function at runtime.
szFunctionDescription	The longer description of the external function. This is displayed in the MetaStock Paste Functions Dialog.

iNArguments	The number of arguments that the external function expects. You may specify up to a maximum of MSX_MAXARGS (9) arguments. Functions with zero arguments are valid. All functions, regardless of the number of defined arguments, will have access to security price data and security detail information for performing calculations. Price data and security details are automatically supplied to all functions without the need for an explicit argument.

MSXFuncArgDef structure

This structure defines a specific argument for an external function. It is used exclusively by the *MSXNthArg* function. See *"MSXNthArg"* (page 28) for more details on using this function.

```
typedef struct
{
     int   iArgType;
     char  szArgName[MSX_MAXSTRING];
     int   iNCustomStrings;
} MSXFuncArgDef;
```

Parameters

iArgType The argument type. There are four valid argument types:

MSXDataArray	Specifies that the argument can be a security price array (e.g., Open, High, Low, Close, etc.), a numeric constant (e.g., 10, 20, -5, etc.), or the result of another function (e.g., Mov(c,30,s)).
MSXNumeric	Specifies that the argument must be a numeric constant (e.g., -5, 10, 20, -15, etc.). Checks for valid constant ranges must be made during calculation (i.e., MetaStock does not know, and therefore cannot check, if the number is within a valid range).
MSXString	Specifies that the argument must be a quote-enclosed string (e.g., "MSFT" or "COMPLEX"). When a calculation is called, the string supplied by the user is passed to the MSX DLL calculation function exactly as it appears within the quotes.
MSXCustom	Specifies that the argument must be from a defined set of possible entries. This is also known as a "custom string" argument. As an example, a custom string could be defined that allows a user to enter SIMPLE, EXPONENTIAL or WEIGHTED for an argument. When a user calls the external function, this argument must contain the text SIMPLE, EXPONENTIAL or WEIGHTED (not enclosed in quotes). If the argument contains some other sequence of characters, it is flagged as a syntax error. **Note:** Custom arguments are not case-sensitive. A user could enter SiMPle for the example mentioned above and it would be accepted for SIMPLE.

szArgName	This is the name of the argument displayed by the MetaStock Paste Functions dialog. This name is also used to identify the argument to the user when they have a syntax error in their *ExtFml()* call. For example, the names of the second and third parameters of the MetaStock built-in moving average function MOV are 'PERIODS' and 'METHOD' respectively. If a user left out the second parameter when entering the *ExtFml()* function in a Custom Indicator, MetaStock would place the cursor at the location of the second argument and display a message similar to: **`PERIODS expected`**.
iNCustomStrings	The number of custom strings associated with this argument, if it is of type *MSXCustom*. For example, if you were defining the third parameter of the MetaStock built-in moving average function MOV, this entry would be 14 for: EXPONENTIAL, SIMPLE, TIMESERIES, TRIANGULAR, WEIGHTED, VARIABLE, VOLUMEADJUSTED, E, S, T, TRI, W, VAR, and VOL.

Notes:

- MetaStock does *not* do a partial match on custom strings
- All legal variations of a string must be specified
- Case is ignored

MSXFuncCustomString structure

This structure defines an allowable string for a custom argument in an external function. It is used exclusively by the *MSXNthCustomString* function. See "*MSXNthCustomString*" (page 29) for more details on using this function.

```
typedef struct
{
     char  szString[MSX_MAXSTRING];
     int   iID;
} MSXFuncCustomString;
```

Parameters

szString	This is the definition of the string. The case will be ignored when MetaStock attempts to match a user argument with this string. Names may consist of alphanumeric characters only (**A** through **Z, a** through **z,** and **0** through **9**). Spaces or other special characters are not allowed.
iID	This is a numeric ID associated with this string. When a call is made to calculate a function in an MSX DLL, this ID is passed to the calculation function rather than the string (szString) itself. Each string must have an ID value. If the MSX DLL defines multiple strings that are synonyms for the same argument (e.g., SIMPLE, SIM, S) then the same ID value should be associated with each string.

IMPORTANT: Strings used to define custom arguments must consist only of alphanumeric characters, e.g., **A**…**Z**, **a**…**z**, **0**…**9**. No spaces or special characters are allowed. If illegal characters are detected in your custom arguments, MetaStock will fail the loading of the DLL and the external functions that it implements will not be available for use.

Calculation Structures

These structures are used by MetaStock and MSX DLLs during the calculation of indicators. All calculation functions in an MSX DLL share a common prototype. The structures that are passed to these functions contain all the security price data (e.g., Open, High, Low, etc.), security details, and function argument data required for calculation of the indicator.

MSXDateTime structure

The date time structure defines a date and time. An array of *MSXDateTime* structures is included in the *MSXDataRec* structure (page 36). For non-intraday data the *lTime* member of the structure is set to zero. The section titled "Formats" on page 31 has more information on the date and time formats.

```
typedef struct
{
  long lDate;
  long lTime;
} MSXDateTime;
```

Parameters

There are no parameters for this structure.

MSXDataInfoRec structure

All numeric data used within indicator calculations is stored in an *MSXDataInfoRec* structure. This includes price data (Open, High, Low etc.), numeric constants and the results of all calculations. The section titled "Data Storage and Calculations" on page 56 has more details regarding the use of this structure.

```
typedef struct
{
  float    *pfValue;
  int      iFirstValid;
  int      iLastValid;
} MSXDataInfoRec;
```

Parameters

pfValue	Pointer to an array of *float*. This array contains all values for a specific set of data.
iFirstValid	Index of the first valid array entry.
iLastValid	Index of the last valid array entry.

Notes:

- There are cases where a data array may be empty. For example, if a chart contained no Open Interest data, the *MSXDataInfoRec* structure for Open Interest data would be empty. An empty array of data is identified by an *iFirstValid* value greater than the *iLastValid* value. Typically, *iFirstValid* would be 0 and *iLastValid* would be -1.
- This occurs on a regular basis during indicator calculations. It is *not* an error condition. Your code should be prepared to routinely identify and handle an empty data array. As a rule, any indicator calculated on an empty data array results in an empty data array.

MSXDataRec structure

The *MSXDataRec* structure is used by MetaStock to supply security price data to indicator calculations. All relevant price data for a specific security is contained in this structure. This structure is automatically supplied to all MSX DLL calculation functions without the necessity of a specific argument. Please see “Programming Guidelines” starting on page 56 for more details about the use of this structure.

The *MSXDataRec* structure contains a pointer to an array of *MSXDateTime* structures (page 35), as well as all the following price *MSXDataInfoRec* structures: Open, High, Low, Close, Volume, Open Interest, and Indicator (page 35). It is available to all external functions in an MSX DLL. The “Function Prototype Section” on page 27 has more details.

```
typedef struct
{
  MSXDateTime     *psDate;
  MSXDataInfoRec  sOpen;
  MSXDataInfoRec  sHigh;
  MSXDataInfoRec  sLow;
  MSXDataInfoRec  sClose;
  MSXDataInfoRec  sVol;
  MSXDataInfoRec  sOI;
  MSXDataInfoRec  sInd;
      char        *pszSecurityName;
      char        *pszSymbol;
      char        *pszSecurityPath;
      char        *pszOnlineSource;
      int         iPeriod;
      int         iInterval;
      int         iStartTime;
      int         iEndTime;
      int         iSymbolType;
  } MSXDataRec;
```

Parameters

psDate	Pointer to an array of *MSXDateTime* structures. See page 35.
sOpen	The open price *MSXDataInfoRec* structure. See page 35.
sHigh	The high price *MSXDataInfoRec* structure. See page 35.
sLow	The low price *MSXDataInfoRec* structure. See page 35.
sClose	The close price *MSXDataInfoRec* structure. See page 35.
sVol	The volume *MSXDataInfoRec* structure. See page 35.
sOI	The open interest *MSXDataInfoRec* structure. See page 35.
sInd	When an external function is used in a custom indicator, the *sInd* structure contains the data the custom indicator was *dropped on*. If dropped on a high/low/close bar, equivolume, candlevolume, or candlestick price plot, the structure contains the closing price. If dropped on another indicator, the values of that indicator are contained in the structure. When an external function is used in a System Test or Exploration, this *sInd* structure contains data for the *selected plot*. The selected plot is the price or indicator that has been selected with the mouse. When an external function is used in an Expert, this structure contains data for the plot selected when the expert was *initially* attached to the chart. It does not contain the currently selected plot, as is the case with system tests and explorations. If your MSX DLL requires the *sInd* structure to contain valid data in order to return correct values, it is important that you clearly instruct users of these requirements.

Note: The *sInd* structure is identical (from the end user’s perspective) to the MetaStock Formula Language’s “P” variable discussed in the MetaStock *User's Manual*.

pszSecurityName	Descriptive name of the security.
pszSymbol	Symbol name of the security.
pszSecurityPath	Path to the location where the security is stored. This string may be in UNC format. Under some circumstances this string may be empty. An empty string should not be treated as an error.
pszOnlineSource	Unused – reserved for future use.
iPeriod	‘**D**’aily, ‘**W**’eekly, ‘**M**’onthly, ‘**Y**’early, ‘**Q**’uarterly, ‘**I**’ntraday. Additional values may be introduced in the future – do not consider values outside the currently defined set to be an error.
iInterval	**0** = tick, **other value** = minutes compression. Valid for *iPeriod* = ‘I’ntraday only, otherwise undefined.
iStartTime	Interval start time in 24-hour HHMM format. Valid for *iPeriod* = ‘**I**’ntraday only, otherwise undefined.
iEndTime	Interval end time in 24-hour HHMM format. Valid for *iPeriod* = ‘**I**’ntraday only, otherwise undefined.
iSymbolType	Unused – reserved for future use.

Function Argument structures

Programmer-defined arguments are passed to the external functions in four argument arrays: *MSXDataInfoRecArgsArray* (page 38), *MSXNumericArgsArray* (page 39), *MSXStringArgsArray* (page 39), and *MSXCustomArgsArray* (page 39). Each of the arrays is zero-based, and arguments are added to the arrays as they are defined in the external function argument list from left to right. In other words, if the argument list contains three *MSXNumeric* arguments, regardless of how many other arguments of other types there are in the argument list, the three *MSXNumeric* arguments will be found in the first three entries of the *MSXNumericArgsArray* argument array.

An example will help illustrate the use of the argument arrays. If an external function is defined with the following parameter list:

```
MyFunc(DataArray1, Numeric, DataArray2, String, DataArray3,
     Custom)
```

then the function arguments would be found in the following argument array locations:

DataArray1	a_psDataInfoArgs->psDataInfoRecs[0]
Numeric	a_psNumericArgs->fNumerics[0]
DataArray2	a_psDataInfoArgs->psDataInfoRecs[1]
String	a_psStringArgs->pszStrings[0]
DataArray3	a_psDataInfoArgs->psDataInfoRecs[2]
Custom	a_psCustomArgs->iCustomIDs[0]

For more examples, see the Sample DLL Programs chapter starting on page 63, the installed sample program source code, and the installed program templates (*MSXTmplt.cpp* and *MSXTmplt.pas*).

Notes:

- Each argument array can contain up to MSX_MAXARGS (9) entries, exactly corresponding to the number of arguments of that type defined for the particular external function.
- The number of valid entries in each argument array is specified in its *iNRecs* structure member. The sum of the *iNRecs* members of the four argument arrays will equal the number of arguments defined for the external function.

MSXDataInfoRecArgsArray structure

MSXDataArray (page 33) arguments required by an external function are included in this array.

```
typedef struct
{
  MSXDataInfoRec
 *psDataInfoRecs[MSX_MAXARGS];
     int   iNRecs;
} MSXDataInfoRecArgsArray;
```

Parameters

psDataInfoRecs	An array of pointers to *MSXDataInfoRecs* (page 35).
iNRecs	The number of valid entries in the *psDataInfoRecs* array.

MSXNumericArgsArray structure

Numeric (*float*) arguments required by an external function are included in this array.

```
typedef struct
{
     float fNumerics[MSX_MAXARGS];
     int   iNRecs;
} MSXNumericArgsArray;
```

Parameters

fNumerics	An array of floats.
iNRecs	The number of valid entries in the *fNumerics* array.

MSXStringArgsArray structure

String arguments required by an external function are included in this array.

```
typedef struct
{
     char  *pszStrings[MSX_MAXARGS];
     int   iNRecs;
} MSXStringArgsArray;
```

Parameters

pszStrings	An array of pointers to null (zero) terminated character strings.
iNRecs	The number of valid entries in the *pszStrings* array.

MSXCustomArgsArray structure

Custom argument ID's required by an external function are included in this array.

```
typedef struct
{
     int   iCustomIDs[MSX_MAXARGS];
     int   iNRecs;
} MSXCustomArgsArray;
```

Parameters

iCustomIDs	An array of integers containing Custom ID's.
iNRecs	The number of valid entries in the *iCustomIDs* array.

MSXResultRec structure

Data returned from an external calculation function.

```
typedef struct
{
  MSXDataInfoRec *psResultArray;
     char  szExtendedError[MSX_MAXSTRING];
} MSXResultRec;
```

Parameters

psResultArray	Data array returned from calculation function.
szExtendedError	If the external function returns a value of MSX_ERROR, an extended description of the error can be copied to this string. MetaStock will display the contents of this string to the user indicating the cause of the error.

Examples

For examples of sample DLL creation programs in in C, Delphi Pascal, and PowerBASIC/DLL, see "Sample DLL Programs" on page 63.

Creating an MSX DLL

This section describes how to create an MSX DLL using Microsoft Visual C++, Borland C++ 4.0, Borland C++ 5.0 Borland Delphi, and PowerBASIC/DLL. Most of these development environments have considerable flexibility in creating DLLs. The approach presented here is merely one way to help you get started with your first DLL.

Note: Microsoft Visual Basic does not have the capability to produce a Win32 DLL. Therefore, MSX DLLS cannot be written in Microsoft Visual Basic. A good alternative for VB programmers is PowerBASIC, an inexpensive compiled Basic that is syntax-compatible with VB and can produce Win32 DLLs.

Microsoft Visual C++ 4.x, 5.0, and 6.0

The MetaStock Developer's Kit setup installs a new AppWizard called "MetaStock MSX DLL AppWizard" to the Microsoft Developer's Studio environment.

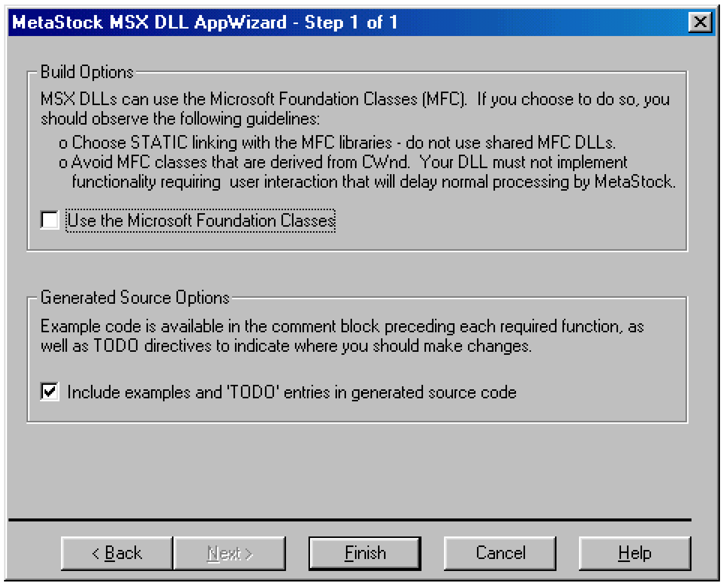
To Access the MSX AppWizard:

Stage I **For Version 4.x users:**

1. Select File> New> Project Workspace.
2. Click **Create**.

For Version 5.0 and 6.0 users:

1. Select **File>** New and then click the Projects tab.
2. Highlight MetaStock MSX DLL AppWizard, and select a name and location for the MSX DLL project.
3. Make sure "Create new workspace" is selected and click **OK**.
   All versions will be presented with the following wizard dialog:



MetaStock MSX DLL AppWizard - Step 1 of 1

Build Options

MSX DLLs can use the Microsoft Foundation Classes (MFC). If you choose to do so, you should observe the following guidelines:

o Choose STATIC linking with the MFC libraries - do not use shared MFC DLLs.
o Avoid MFC classes that are derived from CWnd. Your DLL must not implement functionality requiring user interaction that will delay normal processing by MetaStock.

☐ Use the Microsoft Foundation Classes

Generated Source Options

Example code is available in the comment block preceding each required function, as well as TODO directives to indicate where you should make changes.

☑ Include examples and 'TODO' entries in generated source code

You may wish to use some of the helpful MFC classes, such as `CString` and the many container classes. If you choose to create a DLL that uses MFC classes, please follow these guidelines:

- Do not use any classes within your DLL that are derived from `CWnd`.
- Your DLL should not have any user interface functionality.

- Equis ***strongly*** suggests that you statically link the MFC libraries to your DLL for the following reasons:
  - The MFC DLLs are installed as part of MetaStock, but you cannot be sure that they will be compatible with your DLL. Statically linking the MFC libraries will ensure compatibility and eliminate run-time conflicts. The additional overhead of the static MFC library will be minimal when using the lightweight MFC classes that are appropriate for use in an MSX DLL.
  - Using the MFC DLLs within your MSX DLL (dynamic linking) requires the correct and consistent use of the AFX_MANAGE_STATE macro in all exported functions.
    Failure to do so may corrupt MetaStock's use of MFC, resulting in system lock-ups and data loss.
  - A discussion of the AFX_MANAGE_STATE macro is beyond the scope of this manual. You may research its use in the Visual C++ on-line help.
  - Inclusion of the examples and TODO entries in the generated code is recommended.

Stage II **For all versions:**

1. Click **Finish** to create the project.
   The newly created project will contain all necessary project files.
   Notes:
   - The generated files define a user function called "*EmptyFunc*".
     You should replace "EmptyFunc" with your own function(s).
   - When you change the name of "*EmptyFunc*" in your source file, be sure to also change the EXPORT name in the DEF file.

Before compiling your project:

Before compiling, make sure the IDE is set to find the header file *`MSXStruc.h`*.
You can either copy it from the "*…\MDK\MSX\C*" install directory to your project directory, or you can set the IDE via "Tools> Options> Directories" and add the "*…\MDK\MSX\C*" install directory to the list of include directories.
If you plan to write more than one MSX DLL it would be best to use the latter approach.

Borland C++ Builder 4.0

Using the Borland C++ Builder Integrated Development Environment (IDE), create a new DLL project.

To create a new DLL project:

1. Select File> New from the IDE main menu.
2. Click the New tab.
3. Click the Console Wizard icon.
4. On the Console Application Wizard dialog screen:
   a. Select DLL under Execution Type.
   b. Click **Finish**.
      The IDE will create a new DLL project called "Project1", and will fill in a beginning DLL source file called *`Project1.cpp`* with a few comments and some startup code.
      Note: If you have been working in the IDE before creating this project, the project number may be greater than 1.
5. Delete all of the generated code in the Project1.cpp edit window except the line:
   "**`#include <condefs.h>`**"
6. Select **File>** Open from the IDE main menu.
7. Select Any file (*.*) under Files of type control.
8. Switch to the "*…\MDK\MSX\C*" install folder.

9. Select both the `MSXTmplt.cpp` and `MSXTmplt.def` files.
   Both files can be selected by holding down the Ctrl key while left-clicking on the files.
10. Click Open on the Open dialog.
    The two files will appear in the IDE under two new tabs.
11. Select the `MSXTmplt.cpp` tab.
12. Highlight the entire contents of the `MSXTmplt.cpp` file in the editor and copy it to the clipboard by selecting Edit> Copy from the IDE main menu (or pressing CTRL+C).
13. Switch back to the Project1.cpp tab.
14. Position the cursor below the "`#include <condefs.h>`" line.
15. Select Edit> Paste from the IDE main menu (or press CTRL+V).
    The MSXTemplate contents will be pasted into the new project.
16. Switch back to the `MSXTmplt.cpp` tab and close the edit window by pressing CTRL+F4.
17. Save the project under the name you will be calling your DLL by selecting File> Save Project As…
    Notes:
    - The **Save As…** dialog will be pointing to the MSX Template folder, so be sure to create or select a new folder for your project before saving it.
    - Be sure to replace the name `Project1.bpr` with the name you will be calling your DLL.

After saving your project:

1. Select the `MSXTmplt.def` tab.
2. Select File> Save As…, and save `MSXTmplt.def` in your project folder with the same name you gave your project.
3. Select Project> Add to Project… and select the `.def` file you just saved in your project directory.
   Notes:
   - The IDE will add a line to your `.cpp` file to include the definition file.
     For example, if you named your project Foo, the `foo.cpp` file will now contain the line "`USEDEF("Foo.def");`". At this point you may edit the `.cpp` and `.def` files to implement your own functions.
   - The template files define a user function called "*EmptyFunc*".
     You should replace "*EmptyFunc*" with your own function.
   - When you change the name of "*EmptyFunc*" in your source file, be sure to also change the EXPORT name in the DEF file.

Before compiling your project:

Before compiling your project you must make sure the `MSXStruc.h` file is available to your project. You can either copy the file from the …`\MDK\MSX\C` install directory to your project directory, or you can instruct the IDE where to search for it.

To do this:

1. Select Project> Options… from the IDE main menu.
2. Select the Directories> Conditionals tab.
3. Add the …`MDK/MSX/C` install directory to the Include path edit window.

Note: Borland C++ Builder complains about unused arguments in a function. You can suppress the compiler warning by including the following line above each function declaration: **`#pragma argsused`**.

Look at the provided sample DLLs and printed source code in the next chapter for specific programming requirements.

Borland C++ 5.0

Using the Borland C++ Integrated Development Environment (IDE), create a new **Dynamic Library** project.

To create a new Dynamic Library project:

1. This is done by selecting File> New> Project… from the IDE main menu.
   A project definition dialog will be displayed.
2. Specify the project path and name, using the name for your DLL as the project name.
3. Select Dynamic Library (.dll) under Target Type.
4. Select Win32 under **Platform**, and GUI under **Target Model**.
5. De-select all check-box options under **Frameworks**, **Controls**, and **Libraries**.
6. Select Static under **Libraries**.

   A window will open showing three files: `<projectname>.cpp`, `<projectname>.def,` and `<projectname>.rc`. These files do not yet exist.
7. Copy `MSXTmplt.cpp` and `MSXTmplt.def` from the `…\MDK\MSX\C` install directory to your project directory, and rename them to your project name. (You can do this with Windows Explorer.)

   For example, if your project is called "MyFuncs" and is located at `C:\MyFuncs`, copy `MSXTmplt.cpp` and `MSXTmplt.def` to `C:\MyFuncs`, and then rename `MSXTmplt.cpp` to `MyFuncs.cpp`, and `MSXTmplt.def` to `MyFuncs.def`.
8. At this point you can double click on the filenames in the IDE and edit them.

   Notes:
   - The template files define a user function called "*EmptyFunc*".
     You should replace "*EmptyFunc*" with your own function.
   - When you change the name of "*EmptyFunc*" in your source file, be sure to also change the EXPORT name in the DEF file.

You are now ready to modify the CPP file to implement your functions.

Before compiling your project:

Before compiling your project you must make sure the `MSXStruc.h` file is available to your project. You can either copy the file from the "`…\MDK\MSX\C`" install directory to your project directory, or you can instruct the IDE where to search for it.

To do this:

1. Select Options> Project from the IDE main menu.
2. Highlight Directories under **Topics**.
3. Add the MSX install directory to the Source Directories> Include edit window.

Look at the provided sample DLLs and printed source code in the next chapter for specific programming requirements.

Borland Delphi 3.0, 4.0, and 5.0

Using the Borland Delphi Integrated Development Environment (IDE), create a new DLL project.

To create a new DLL project:

1. Select File> New from the IDE main menu.
2. Click the DLL icon.

   The IDE will create a new DLL project and will fill in a beginning DLL source file with a few comments and some startup code.
3. Save the project under the name you will be calling your DLL by selecting File> Save Project As….

After saving your project:

1. Select File> Open from the IDE main menu.
2. Open the *`MSXTmplt.pas`* file, located in the …*`\MDK\MSX\Delphi`* install directory.
3. Highlight the entire contents of the *`MSXTmplt.pas`* file in the editor and copy it to the clipboard by selecting Edit> Copy from the IDE main menu (or pressing CTRL+C).
4. Switch back to the tab with your project name, and highlight the default comments and code that the IDE created.
5. Select Edit> Paste from the IDE main menu (or press CTRL+V) and the MSXTemplate contents will be pasted into the new project, replacing the code generated by the IDE.
6. Change "*`DelDll`*" in the "`library DelDll`" line to the name of your project.
7. Click the MSXTmplt tab.
8. Right-click anywhere in the text, and select "Close Page" or "Close File" (depending on the version of Delphi you are using).

Before compiling your project:

Note: In order to compile your DLL you will have to ensure that the *`MSXStruc.inc`* file can be found. You can either copy it from the …*`\MDK\MSX\Delphi`* install directory to your project directory, or you can add the …*`\MDK\MSX\Delphi`* directory to the list of files the Delphi IDE searches when compiling.

To do this:

1. Select Tools> Environment Options from the IDE main menu.
2. Click on the Library tab.
3. Add the *`MSX\Delphi`* install directory to the list of directories in the Library Path field.

PowerBASIC/DLL 6.0

1. Copy *`MSXTmplt.bas`* from the install folder …*`\MDK\MSX\PBasic`* to the folder where you are going to develop your DLL. (You can do this with Windows Explorer.)
2. Rename your new copy of *`MSXTmplt.bas`* to the name you want for your DLL. For example, if you were creating an MSX DLL called *`MyFuncs`*, you would rename *`MSXTmplt.bas`* to *`MyFuncs.bas`*.
3. Open the new file in the PowerBASIC IDE and proceed to make your changes.

Before compiling your project:

In order to compile your DLL you will have to ensure that the *`MSXStruc.bas`* file can be found. You can either copy it from the "…*`\MDK\MSX\PBasic`*" install directory to your project directory, or you can add the "…*`\MDK\MSX\PBasic`*" directory to the list of files the PowerBASIC IDE searches when compiling.

1. Select Window> Options….
2. Click the Compiler tab.
3. Type a semicolon at the end of the Paths> Include line.
4. Following the semicolon, enter the complete path to where the *`MSXStruc.bas`* file was installed (…*`\MDK\MSX\PBasic`*) .

Note: This operation will only need to be done once, as the path modification will remain until you change it.

Naming your DLL and Calculation Functions

Because MetaStock loads all MSX DLLs from the same folder, each MSX DLL must have a unique name. Be sure to give your DLL a descriptive name that is unlikely to conflict with a DLL name chosen by another developer. Long file names are supported, but keep in mind that MetaStock users will have to type the entire DLL name along with the function name to reference your functions.

Note: Function names must be unique only within a given DLL. Choose descriptive names for each of your functions as a courtesy to the MetaStock users who will be calling them.

Debugging Your MSX DLL

This section describes strategies for debugging MSX DLLs within the Integrated Development Environment (IDE) for Microsoft Visual C++, Borland C++ Builder 4.0, Borland C++ 5.0, and Borland Delphi. IDEs not listed here may have similar capabilities. (Even though the PowerBASIC/DLL IDE does not allow tracing into a running DLL, suggestions for debugging a PowerBASIC DLL are provided.)

General Approach

The general approach is to attach the debugger to *`MSXTest.exe`*, which in turn loads your MSX DLL. While you cannot trace into MSXTest itself, you can set breakpoints within your own DLL code to inspect the data structures being passed into and out of your functions at runtime.

If MSXTest has your MSX DLL loaded, you cannot recompile the DLL until you either close MSXTest, or point it to a different DLL folder. This is because MSXTest holds an open handle to all the DLLs it has loaded and the operating system will not permit you to delete a DLL that is attached to a running process. If you wish to set breakpoints in the initialization functions of your MSX DLL which has already been loaded by MSXTest, you can set the breakpoints and then instruct MSXTest to reload the DLLs. For a complete description of MSXTest read “Testing Your DLL With MSXTest” on page 48.

Microsoft Visual C++ 4.x, 5.0, and 6.0

Stage I Set the Active (Default) Configuration to “Debug” and compile your DLL.

To do this:

For Version 5.0 and 6.0:

1. Select Project> Settings… from the main menu.
2. Select Win32 Debug from the drop-down list on the left.
3. Click the **Debug** tab on the right.

For Version 4.x:

1. Select “Build> Settings…” from the main menu.
2. Select the “Win32 Debug” version of your project from the list box on the left.
3. Click the Debug tab on the right.
4. Make sure the **Category** drop-down list is set to General.

   You will see a dialog with four entry fields.

   a. In the Executable for debug session: field, enter the full path and file name for *`MSXTest.exe`* .
   i.e. “*`C:\Program Files\Equis\MDK\MSX\MSXTest.exe`*”.
   b. Leave the Working Directory field blank.
   c. [OPTIONAL] In the Program Arguments: field**,** enter the full path to the compiled debug version of your MSX DLL i.e. “*`C:\MyMSXDLL\Debug`*”.
   Notes:
      - Do *not* supply the name of the MSX DLL, only the path to it.
      - Alternatively, you may choose to leave this field blank and tell MSXTest where to find the debug version of your DLL after MSXTest starts.
   d. Leave the Remote executable path and file name: fields blank.
   e. Click **OK** to close the Settings dialog.

Stage II **For all versions:**

1. Set breakpoints in the source code for your MSX DLL the same as you would to debug a normal application.
2. Click **Go** (F5) to launch `MSXTest.exe`.

Note: You may be presented with a dialog stating that MSXTest does not contain debug information, and asking if you wish to continue. Click **OK** to continue.

As MSXTest makes calls to your DLL, the debugger will gain control when any breakpoints are encountered. You will be able to inspect the contents of any data structures that are within the scope of your breakpoint.

Borland C++ Builder 4.0

1. Compile your DLL.
2. Select Run> Parameters… from the main menu.
3. In the Host Application field, enter the complete path and filename for `MSXTest.exe`. (Example: `C:\Program Files\Equis\MDK\MSX\MSXTest.exe`).
4. [OPTIONAL] In the Parameters field, enter the full path to the compiled debug version of your MSX DLL. (Example: `C:\MyMSXDLL\`).
   Notes:
   - Do *not* supply the name of the MSX DLL, only the path to it.
   - Alternatively, you may choose to leave this field blank and tell MSXTest where to find the debug version of your DLL after MSXTest starts.
5. Click **OK** to close the **Run Parameters** dialog.
6. Set breakpoints in the source code for your MSX DLL the same as you would to debug a normal application.
7. When you click **Run**, `MSXTest.exe` will load.
   As MSXTest makes calls to your DLL, the debugger will gain control when any breakpoints are encountered. You will be able to inspect the contents of any data structures that are within the scope of your breakpoint.

Borland C++ 5.0

1. Compile your DLL.
2. Select Debug> Load… from the main menu.
3. In the Program name field, enter the complete path and filename for `MSXTest.exe`. (Example: `C:\Program Files\Equis\MDK\MSX\MSXTest.exe`).
4. [OPTIONAL] In the Arguments field, enter the full path to the compiled debug version of your MSX DLL. (Example: `C:\MyMSXDLL\`).
   Notes:
   - Do *not* supply the name of the MSX DLL, only the path to it.
   - Alternatively, you may choose to leave this field blank and tell MSXTest where to find the debug version of your DLL after MSXTest starts.
5. Click **OK** to close the **Load Program** dialog (and start `MSXTest.exe` running).
6. Set breakpoints in the source code for your MSX DLL the same as you would to debug a normal application.
   As MSXTest makes calls to your DLL, the debugger will gain control when any breakpoints are encountered. You will be able to inspect the contents of any data structures that are within the scope of your breakpoint.

Borland Delphi 3.0, 4.0, and 5.0

1. Compile your DLL.
2. Select Run> Parameters… from the main menu.
3. Click the **Local** tab.
4. In the Host Application field, enter the complete path and filename for `MSXTest.exe`. (Example: `C:\Program Files\Equis\MDK\MSX\MSXTest.exe`).

5. [OPTIONAL] In the Parameters field, enter the full path to the compiled debug version of your MSX DLL. (Example: `C:\MyMSXDLL\`).

   Notes:
   - Do *not* supply the name of the MSX DLL, only the path to it.
   - Alternatively, you may choose to leave this field blank and tell MSXTest where to find the debug version of your DLL after MSXTest starts.
6. Click **OK** to close the **Run Parameters** dialog.
7. Set breakpoints in the source code for your MSX DLL the same as you would to debug a normal application.
8. When you click **Run** (F9), `MSXTest.exe` will be launched.

   As MSXTest makes calls to your DLL, the debugger will gain control when any breakpoints are encountered. You will be able to inspect the contents of any data structures that are within the scope of your breakpoint.

PowerBASIC/DLL 6.0

You cannot trace directly into your PB/DLL code, but you can use the direct approach of displaying the state of the DLL by popup messages and outputting debug strings.

- Popup messages, via the *MSGBOX* command, will cause execution of the DLL to pause until they are acknowledged.
- You can sprinkle calls to the Win32 system call "*OutputDebugString*" throughout your DLL without adversely affecting its performance.
- A good debug message view utility such as "DebugView" found at `http://www.sysinternals.com` can be used to observe all the debug messages being displayed by your DLL.
- Be sure to append a line feed character to your output string when using *OutputDebugString*, i.e.: `OutputDebugString ("Just before initializing ReturnArray" + CHR$(10))`

  In either case you can display the values of variables and execution location information that can help you to track down run-time errors in your code.

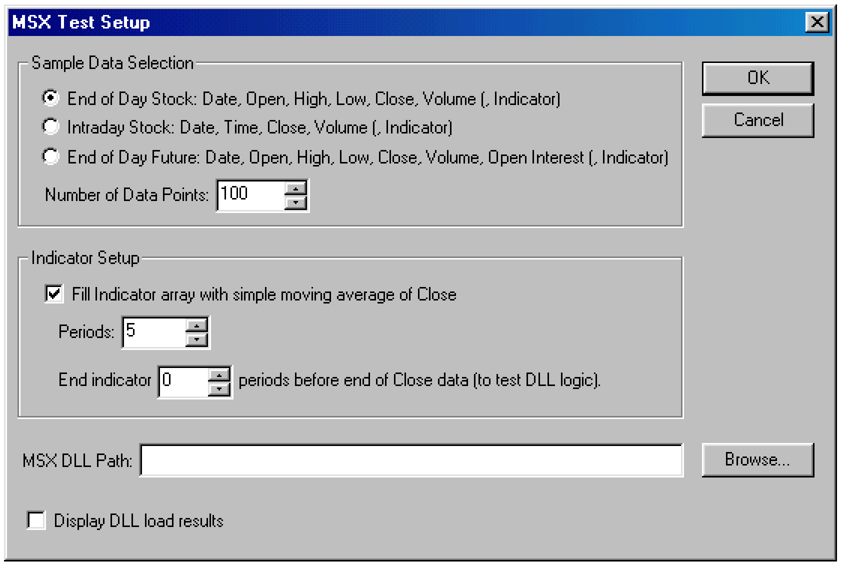
When you are finished debugging be sure to delete all the *OutputDebugString* and *MSGBOX* trace commands from your finished source.

Testing Your DLL With MSXTest

MSXTest will load your MSX DLLs and allow you to test the functions using actual data. You may select from three sets of provided data: end-of-day stock, real-time stock, or end-of-day futures. Each dataset has from 0 to 1000 datapoints available. You can control how many datapoints are available for testing your function. An interactive graphical tree-style display of your functions allows for input of argument values. Results following the execution of your external function are displayed in a spreadsheet format. You can print the results, export all results to a CSV (comma delimited) file, or even launch the application associated with CSV files (normally Microsoft Excel).

IMPORTANT: It is important that you thoroughly test your DLLs using MSXTest before distributing them to MetaStock users. MSXTest can isolate many potential problems that an MSX DLL might have.

The first time MSXTest is run, you will be presented with the setup screen shown below. Click **OK** to save your selections or **Cancel** to ignore changes.


MSX Test Setup

Sample Data Selection

(●) End of Day Stock: Date, Open, High, Low, Close, Volume (, Indicator)
( ) Intraday Stock: Date, Time, Close, Volume (, Indicator)
( ) End of Day Future: Date, Open, High, Low, Close, Volume, Open Interest (, Indicator)

Number of Data Points: 100

Indicator Setup

[✓] Fill Indicator array with simple moving average of Close

Periods: 5

End indicator 0 periods before end of Close data (to test DLL logic).

MSX DLL Path:

[ ] Display DLL load results

Note: Clicking **OK** will cause the sample data to be re-loaded, but DLLs will not be re-loaded until Load DLLs is selected from the main menu or toolbar.

The fields on the setup screen are defined as follows:

Sample Data Selection

Select one of three pre-defined data sets. The indicated data arrays will contain data. In all cases the Indicator data will be filled only if Fill Indicator array with simple moving average of Close is selected in the Indicator Setup section. You can select from 0 to 1000 data points (time periods). Smaller sets may make hand-checking of your functions easier. Setting the number of datapoints to 0 will test your DLL for handling empty data arrays.

Indicator Setup

This will fill the Indicator member of the internal *MSXDataRec* with a simple moving average of the Close data. You can specify the number of periods in the moving average and you can terminate the indicator early. This is useful for testing your DLL on data arrays that may be smaller on both ends than the price data.

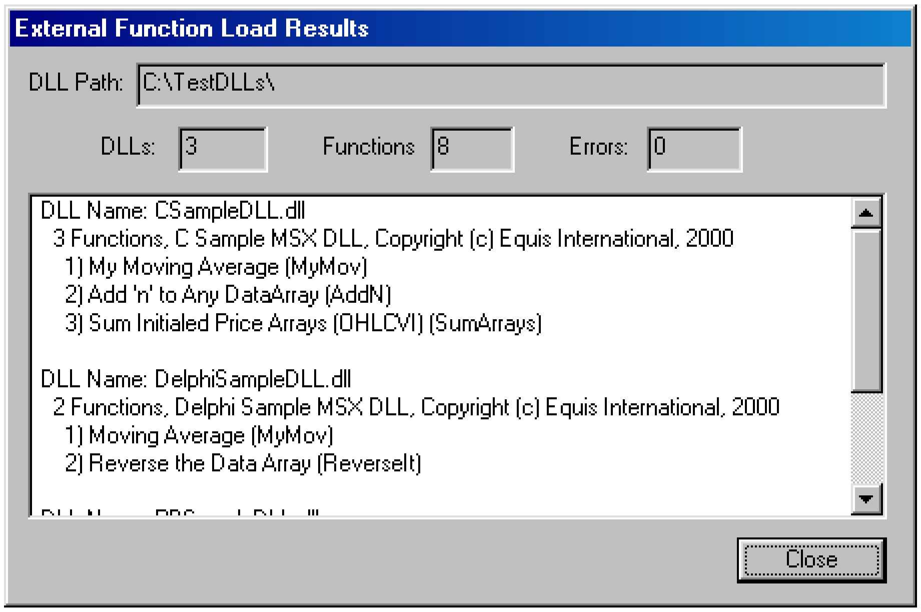
MSX DLL Path

This is where MSXTest looks for DLLs to load. If this is the first time MSXTest has been run this field will be blank and you should type in the path that contains the desired DLLs. Alternatively, you can use the **Browse** button to locate the DLL path.

Display DLL load results

MSXTest can display a summary of DLL load results, including any errors that were encountered. If this option is not checked the load results will be displayed only if there are errors. Selecting this option will always display the load results, regardless of errors.

Loading DLLs when “Display DLL load results” is selected in the Setup Screen will produce a window similar to the following:



External Function Load Results

DLL Path: C:\TestDLLs\

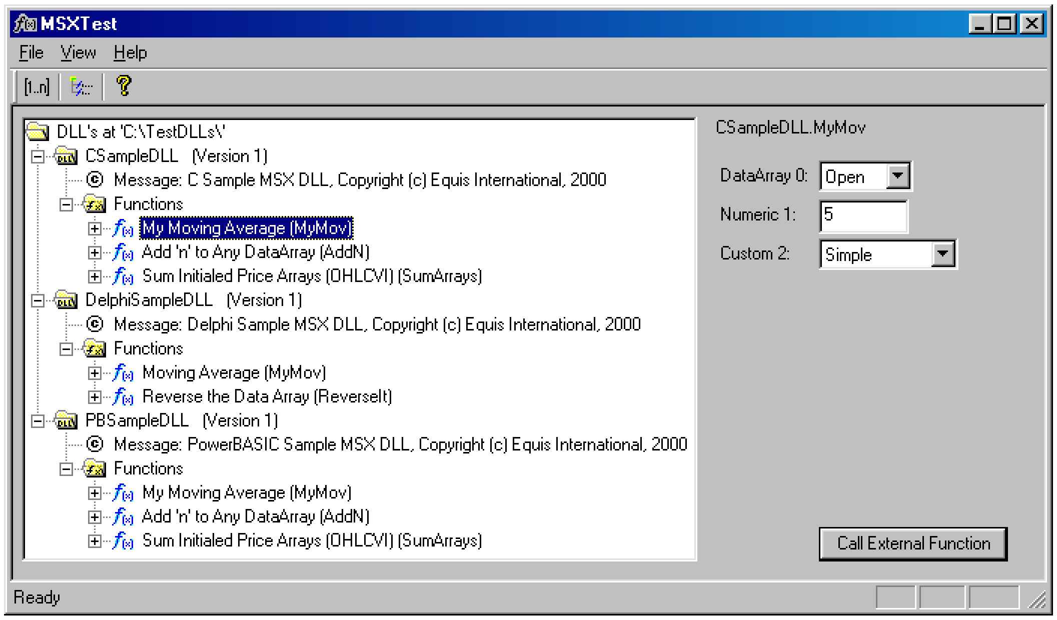
DLLs: 3 Functions 8 Errors: 0

```
DLL Name: CSampleDLL.dll
  3 Functions, C Sample MSX DLL, Copyright (c) Equis International, 2000
    1) My Moving Average (MyMov)
    2) Add 'n' to Any DataArray (AddN)
    3) Sum Initialed Price Arrays (OHLCVI) (SumArrays)

DLL Name: DelphiSampleDLL.dll
  2 Functions, Delphi Sample MSX DLL, Copyright (c) Equis International, 2000
    1) Moving Average (MyMov)
    2) Reverse the Data Array (ReverseIt)
```

The DLL path is displayed along with the number of DLLs found, the total number of functions, and the number of errors encountered while loading the DLLs. If the error count is non-zero, the specific error will be displayed in the tree view where it occurred, and the DLL containing the offending function will be marked invalid – its functions will not be available for testing until the problem is fixed and the DLLs are re-loaded.

The main screen appears as follows:



```
DLL's at 'C:\TestDLLs\'
  CSampleDLL    (Version 1)
    Message: C Sample MSX DLL, Copyright (c) Equis International, 2000
    Functions
      My Moving Average (MyMov)
      Add 'n' to Any DataArray (AddN)
      Sum Initialed Price Arrays (OHLCVI) (SumArrays)
  DelphiSampleDLL    (Version 1)
    Message: Delphi Sample MSX DLL, Copyright (c) Equis International, 2000
    Functions
      Moving Average (MyMov)
      Reverse the Data Array (ReverseIt)
  PBSampleDLL    (Version 1)
    Message: PowerBASIC Sample MSX DLL, Copyright (c) Equis International, 2000
    Functions
      My Moving Average (MyMov)
      Add 'n' to Any DataArray (AddN)
      Sum Initialed Price Arrays (OHLCVI) (SumArrays)
```

CSampleDLL.MyMov

DataArray 0: Open

Numeric 1: 5

Custom 2: Simple

The tree-view display shows that there were three DLLs loaded from `C:\TestDLLs\` — *`CSampleDLL`*, *`DelphiSampleDLL`*, and *`PBSampleDLL`*. (These DLLs are provided with complete source code as samples with this toolkit.) The functions contained in each DLL are displayed including description and name. Beneath each function is a list of the arguments for that function, including the argument type. Custom types can be further expanded to show each possible option.

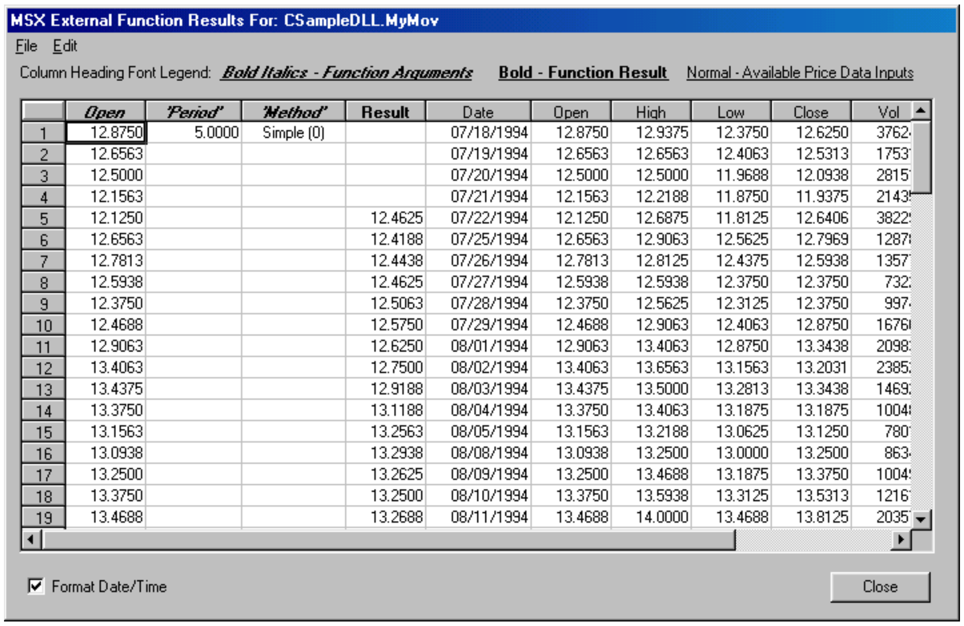
The following menu options are available:

Menu Option	Result
File> Setup	Display the setup dialog.
File> Load DLLs	Load (or re-load) the DLLs from the path defined in the setup dialog.
File> Stress Test…	Perform comprehensive stress tests on the selected function. This option is enabled only when a DLL function is selected. See "Stress Testing Your DLL Functions" on page 52 for more details.
File> Exit	Exit the MSXTest application.
View> Toolbar	Toggle display of toolbar.
View> Status Bar	Toggle display of status bar.
Help> About MSXTest…	Version information for MSXTest and summary of loaded DLL copyright strings.

The Toolbar contains shortcut buttons to the following menu options:

[1..n]	Setup
	LoadDLLs
?	About MSXTest

When a function or any of its arguments is highlighted in the display tree, the right side of the main screen displays all the arguments. You can fill them in as you wish and then click **Call External Function**. The function will be called with the specified arguments and the results will be displayed in a window similar to the following:



MSX External Function Results For: CSampleDLL.MyMov

File Edit

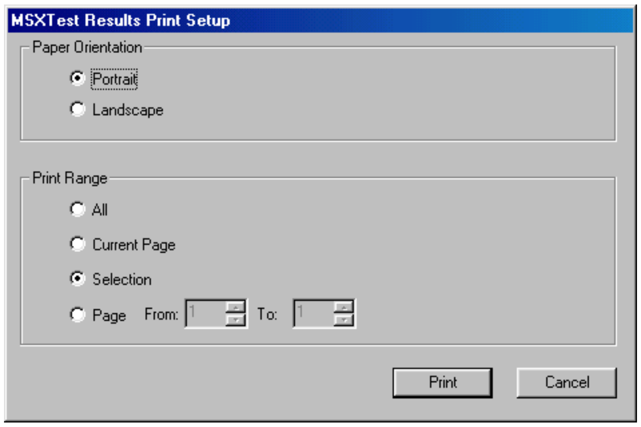
Column Heading Font Legend: ***Bold Italics - Function Arguments*** **Bold - Function Result** Normal - Available Price Data Inputs

	Open	***'Period'***	***'Method'***	**Result**	Date	Open	High	Low	Close	Vol
1	12.8750	5.0000	Simple (0)		07/18/1994	12.8750	12.9375	12.3750	12.6250	3762
2	12.6563				07/19/1994	12.6563	12.6563	12.4063	12.5313	1753
3	12.5000				07/20/1994	12.5000	12.5000	11.9688	12.0938	2815
4	12.1563				07/21/1994	12.1563	12.2188	11.8750	11.9375	2143
5	12.1250			12.4625	07/22/1994	12.1250	12.6875	11.8125	12.6406	3822
6	12.6563			12.4188	07/25/1994	12.6563	12.9063	12.5625	12.7969	1287
7	12.7813			12.4438	07/26/1994	12.7813	12.8125	12.4375	12.5938	1357
8	12.5938			12.4625	07/27/1994	12.5938	12.5938	12.3750	12.3750	732
9	12.3750			12.5063	07/28/1994	12.3750	12.5625	12.3125	12.3750	997
10	12.4688			12.5750	07/29/1994	12.4688	12.9063	12.4063	12.8750	1676
11	12.9063			12.6250	08/01/1994	12.9063	13.4063	12.8750	13.3438	2098
12	13.4063			12.7500	08/02/1994	13.4063	13.6563	13.1563	13.2031	2385
13	13.4375			12.9188	08/03/1994	13.4375	13.5000	13.2813	13.3438	1469
14	13.3750			13.1188	08/04/1994	13.3750	13.4063	13.1875	13.1875	1004
15	13.1563			13.2563	08/05/1994	13.1563	13.2188	13.0625	13.1250	780
16	13.0938			13.2938	08/08/1994	13.0938	13.2500	13.0000	13.2500	863
17	13.2500			13.2625	08/09/1994	13.2500	13.4688	13.1875	13.3750	1004
18	13.3750			13.2500	08/10/1994	13.3750	13.5938	13.3125	13.5313	1216
19	13.4688			13.2688	08/11/1994	13.4688	14.0000	13.4688	13.8125	2035

[✓] Format Date/Time

The incoming arguments are displayed first under ***Bold Italics***. The result array is always under **Bold** regular text, and is always labeled "**Result**". Following the result array all non-empty input data from the *MSXDataRec* is displayed. This consists of Date and Time, and the following *MSXDataInfoRec* arrays: Open, High, Low, Close, Vol, Open Interest, and Indicator.

The following menu options allow further manipulation of the result data:

Menu Option	Result
File> Open (CTRL+O)	Saves the spreadsheet to a temporary CSV (comma delimited) file, and launches the application associated by Windows with CSV files (normally Microsoft Excel).
File> Save As (CTRL+S)	Saves the data to a CSV (comma delimited) file. The CSV file format is easily imported into most Windows spreadsheet programs. The CSV extension is normally associated by Windows with Microsoft Excel.
File> Print (CTRL+P)	This option will bring up the following window to control which portions of the spreadsheet are to be sent to the default printer:  MSXTest Results Print Setup — Paper Orientation: [✓ Portrait] Landscape — Print Range: All, Current Page, [✓ Selection], Page From: 1 To: 1 — Print, Cancel
Edit> Copy (CTRL+C)	This option will copy the selected cells to the Windows clipboard. Select cells by clicking with the left mouse and dragging or by clicking any row or column header.
Edit> Copy All (CTRL+A)	This option will copy the entire spreadsheet to the Windows clipboard.

Notes:

- All the spreadsheet menu functions are available by right-clicking the mouse anywhere on the spreadsheet grid.
- The Format Date/Time checkbox will cause the Date and Time columns to be formatted as MM/DD/YYYY and MM:SS:TTT rather than YYYYMMDD and MMSSTTT respectively.

Stress Testing Your DLL Functions

The DLL Stress Test involves calling your DLL function thousands of times with many variations in the data arguments. Three types of tests are performed: Max/Min Data Array Tests, Special Case Data Array Tests, and Argument Range Tests.

Note: Most of these conditions should *not* occur in practice. The stress tests ensure that your DLL can handle extreme conditions without crashing or causing a run-time exception.

Max/Min data array tests

Max/Min data array tests consist of calling your DLL function with all possible combinations of the following data array setups:

Setup	Result
Empty	All data arrays are empty.
Max	Data arrays are either empty or filled with the maximum *float* value (FLT_MAX).
Min	Data arrays are either empty or filled with the minimum *float* value (FLT_MIN).
NegMax	Data arrays are either empty or filled with the negative maximum *float* value.
NegMin	Data arrays are either empty or filled with the negative minimum *float* value.
Zeros	Data arrays are either empty or filled with zeroes (0.0).
Alternating	Data arrays are either empty or filled with the following repeating sequence: FLT_MAX, -FLT_MAX, FLT_MIN, -FLT_MIN, 0.0.

Special Case data array tests

Special case data array tests consist of calling your DLL function with the following data array setups:

Setup	Result
One Element	Each data array is set with *iFirstValid* equal to *iLastValid*.
Illegal First/Last	*iFirstValid* is less than zero and *iLastValid* is greater than 0.
Unusual Empty	*iLastValid* is less than *iFirstValid* indicating empty, but they are set to unusual values rather than *iFirstValid* = 0, *iLastValid = -1*.
Close GT High	The values in the Close price array are greater than the values in the High data array.
Close LT Low	The values in the Close price array are less than the values in the Low data array.
Y2K	The date field in the *DateTime* structure contains all dates greater than 20000101.
Y2K Transition	The date field in the *DateTime* structure crosses over from dates in 1999 to 2000.
Invalid Dates	Invalid month and day components in the date field.
No Date	The date array is zero-filled.
Date Gaps	Occasional sequences of zero in the date array.
Date Sequence	Dates in date array suddenly jump back in time.
Invalid Times	Hour and minute components of time array are illegal.
Time w/o Date	The time array contains valid data, but the date array is zero-filled.
Random Ticks	The ticks component of the time field is set to random non-contiguous values.
Repeating High Ticks	The ticks component of the time field is incremented to 999, then several entries repeat at 999 before the rest of the time changes and the tick field is reset.

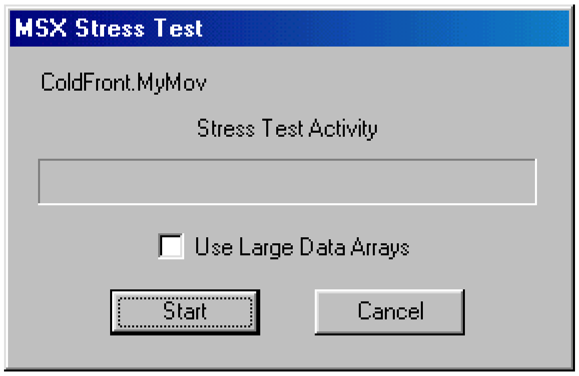
Argument range tests

Argument range tests call your function with each of the built-in data sets, and the function arguments set as follows:

Setup	Result
Numeric fields	These are set from 999999999999.00 to -999999999999.00. The value is modified toward 0.0 by 20% on each call.
String fields	These are called with empty strings, large strings containing all typeable characters, and a string of blanks.
Custom fields	These are set to all legal values defined by the function, INT_MIN, and INT_MAX.

Running a Stress Test

When you highlight a function, then select “File> Stress Test…” from the main menu you will be presented with a dialog box similar to the following:



Click **Start** to begin the Stress Test.

- A progress bar will indicate test activity, but is not representative of the total number of tests to be performed. The bar may be fully displayed from one to five times depending on the number of arguments defined for the function. There will be a great deal of disk activity during the test.
- At the conclusion of the Stress Test a text file will be displayed with the test results. Any function calls that produce a run-time exception or return a value of MSX_ERROR will be logged in the file.
- If the error is a fatal-type error, such as illegal memory writes or stack overflow, the test will be prematurely terminated.
- Min/Max data array test results will include a string indicating which data arrays contain data and which are empty. For example, the string “`O, , ,C,V, ,`” contains data in the Open, Close and Volume data arrays.
- Normally, the Stress Tests are performed with 1000 data points. This allows rapid testing for initial results or re-testing DLL modifications. Be sure to run the Stress Tests with the “Use Large Data Arrays” check box selected at least once before releasing your DLL. This option uses data arrays with 65500 data points. Large data arrays significantly slow the stress test, but will help to reveal floating point overflows and underflows that may be missed by the normal test.
- The results from all Stress Tests are stored in a folder named “Stress” located immediately below the folder containing *`MSXTest.exe`*. The result file name consists of the DLL name, Function Name, and “*`Stress.Txt`*”. For example, the results for *ColdFront.MyMov* would be found in the file “*`ColdFront.MyMov.Stress.Txt`*”.

CAUTION: ***Each time a stress test is run for a given function, the previous results for that function will be overwritten.***

Considerations for making sure your functions pass the stress test:

- Your function should never produce a math exception, such as overflow, underflow, or division by zero. In practice, your function should not receive values that would cause overflow or underflow conditions to occur, but because your function may receive as input the output of another external function you must be prepared to handle extreme values. The supplied template files, *`MSXTmplt.cpp`*, *`MSXTmplt.pas`*, and *`MSXTmplt.bas`*, contain a function that forces the passed value to lie within the minimum and maximum values for a single precision floating point number. If you perform your floating point calculations using doubles (double precision floating point), and then force the results into the required range, you can avoid most overflow and underflow conditions. See the sample DLLs for examples of using this approach.
- Test the value of the divisor before any mathematical division to avoid division by zero exceptions.
- Test all arguments for valid ranges. Return MSX_ERROR in the cases where a clearly defined argument type is out of bounds (such as an out-of-range Custom ID).
- Make sure you never access a data array with a negative index.
- Be careful about returning the MSX_ERROR result from your external functions. When MetaStock encounters that return type it will display an extended error message in a dialog box that will require user response. Report only errors that are significant problems the user needs to know about – not just exceptional situations your DLL wasn't equipped to handle.

Automating MSXTest From Your IDE

If the compiler IDE you are using supports user-defined tools you may find it useful to define MSXTest in the tool list. Using the specific IDE tool macros, specify the target directory of the project as a command line argument for MSXTest. When MSXTest starts up, it checks its command line arguments for a directory. If one is found, it sets that directory as the location to search for MSX DLLs.

For example, using Microsoft Visual C++ 6.0, you could define MSXTest as a tool by selecting "Tools> Customize" from the main menu. Select the "Tools" tab, and click the "New" icon. Enter *`MSXTest`* and press Enter. Fill in the Command field with the full path to where you installed MSXTest
(e.g. *`C:\Program Files\Equis\MDK\MSX\MSXTest.exe`*). Fill in the "Arguments" field with "`$(TargetDir)`", and leave the "Initial Directory" field blank.
Click "Close". Now when you select "Tools" from the main menu, you will see "MSXTest" as an entry. Most other compiler IDEs have similar capabilities.
Refer to your IDE documentation for specifics.

Testing Your DLL With MetaStock

CAUTION: Be sure you have fully tested your DLL with the MSXTest program before loading it with MetaStock.

To test your MSX DLL with MetaStock, copy it to the “External Function DLLs” folder located below the folder defined for Custom Indicators, System Tests, and Explorations. If the “External Function DLLs” folder does not already exist, you must first create it. (The default location is “*`C:\Program Files\Equis`*\MetaStock\*`External Function DLLs`*”.)

Note: If you are replacing an MSX DLL that already exists in the “External Function DLLs” folder you must first shut down MetaStock. Because MetaStock loads all available MSX DLLs at startup, they cannot be deleted or replaced until MetaStock shuts down and releases the DLLs.

Perform at least the following minimum tests:

- Verify that the correct text appears in the Paste Functions dialog and that the parser is correctly compiling the syntax for all external functions. This includes the display of meaningful error messages when a syntax error is detected in the use of an external function.
- Use Indicator Builder to write a sample indicator that calls a function in your DLL.
- Plot the indicator and check the values. Make sure that calculation results are correctly displayed in charts.
- Edit the plot via right-click.
- Repeat for each function in the DLL.

Programming Guidelines

This section discusses guidelines, limits and other considerations when creating MSX external functions. All examples in this section use “C” syntax. The syntax for Delphi Pascal and PowerBASIC is similar. See the source listings in the “Sample DLL Programs” chapter and included example programs for specific syntax requirements.

Data Storage and Calculations

Data Arrays

All numeric data used within indicator calculations is stored in structures known as **data arrays.** Data arrays are used to store price data (e.g., Open, High, Low, etc.), numeric constants, and the results of an indicator calculation. When MetaStock supplies security price data and numeric argument data to an MSX DLL function, data arrays are used. When an MSX DLL calculation function returns the results of an indicator to MetaStock, the result is returned in a data array.

Data array structures are implemented in the *MSXDataInfoRec* structure defined on page 35, and have three basic components:

- Data elements.
- First valid index.
- Last valid index.

The data elements are the actual numeric values associated with the data array. These values are stored in an array of floating point values. The first valid index and last valid index are used to define which data elements contain valid data. All data elements between the first valid index and the last valid index (inclusive) contain valid data. All elements outside of this range have undefined values and should be ignored for all calculations.

A data array is considered “empty” if the first valid index is greater than the last valid index. Empty data arrays are not uncommon and must be handled properly. Typically, an empty data array will have a first valid of 0 and a last valid of -1, although any combination of a first valid greater than a last valid should be considered empty. Empty data arrays occur when data is not available. For example, an Open Interest data array used for a security that does not have Open Interest. Likewise, the result of a 100-period moving average applied to a security price data array that contains only 90 data elements would be an empty data array.

First and last valid indexes are very important during indicator calculations. Calculations should always be restricted to data elements contained within the first valid/last valid range. Care must be taken to make sure that a data array produced from the result of a calculation has the correct first valid/last valid settings.

Two important concepts must be understood to correctly set the first valid and last valid indexes for the returned data array:

- Always restrict calculations to the union of the valid data ranges of all input data arrays used.
- The first valid and last valid values of a calculation result must maintain their position relative to the values of all input data arrays.

The following example will help to illustrate these concepts.

Assume that an MSX DLL implements a function that adds three data arrays together and then calculates a three period moving average of the sum. The following statistics apply to the three data arrays supplied as input to the function:

```
Data Array 1:First Valid = 1, Last Valid = 10
Data Array 2:First Valid = 3, Last Valid = 10
Data Array 3:First Valid = 1, Last Valid = 7
```

The data arrays could be visualized as follows:

Record	Data Array 1	Data Array 2	Data Array 3
1	3		1
2	2		1
3	2	2	2
4	3	2	2
5	2	3	3
6	1	2	2
7	1	2	1
8	2	2	
9	3	1	
10	2	1	

If the MSX DLL implements the calculations in two steps, the first step would involve the adding of the three data arrays to produce a temporary result array. In this case, the calculation result data array would look like this:
Result Array: First Valid = 3 Last Valid = 7

Record	Result Array
1	
2	
3	6
4	7
5	8
6	5
7	4
8	
9	
10	

Notice how the resulting array has first and last valid set to the union of all three of the input data arrays. Also note how each element of the result array maintains its position relative to the data elements used to calculate the result (the sum of all #3 data elements is stored in the #3 element of the resulting array). This is a very important concept and must be used in the calculation of all indicators in an MSX DLL. By correctly setting the first and last valid indexes, you will allow MetaStock to correctly determine where the indicator plot should start and end in relation to the data used for input into the indicator.

If the MSX DLL applies the three-period moving average to the result array above, the final result would look like this:
Final Result: First Valid = 5 Last Valid = 7

Record	Final Result Array
1	
2	
3	
4	
5	7
6	6.67
7	5.67
8	
9	
10	

Notice that the final array has the first valid set to 5. This is because the 3-period moving average does not come up to speed until the third element of the data on which it is applied. Since the input array had a first valid of 3, the 3-period moving average did not come up to speed until the fifth data element. Again, the last valid value is set to 7 because the input data array had a last valid of 7.

Security Price Data

When MetaStock calls a calculation function in an MSX DLL, it automatically gives the DLL access to security price data. The DLL does not have to explicitly declare an argument in the external function for access to the security data. This means that even if an external function has no arguments, the calculation functions in the DLL will still be given security price and detail data to work with.

MSX DLL calculation function

- An MSX DLL calculation function has no way of knowing which type of formula (e.g., Custom Indicator, System Test, etc.) is calling the external function. The external function calculation process only knows that it is given a set of data arrays that define the price data for the target security. The external calculation function simply performs the appropriate calculations and returns the resulting data array to MetaStock.

MSXDataRec structure

- MetaStock uses the *MSXDataRec* structure (page 36) to supply security price and detail data to an MSX DLL.
- In the case of an external function used in an Indicator, the *MSXDataRec* structure will contain the price and detail data for the base security of the chart where the custom indicator is being calculated.
    - For **System Tests**, the structure is loaded with base security data for the active chart when the system test was launched.
    - For **Explorations**, the structure is loaded with security data for the security currently being explored.
    - For **Experts**, the structure is loaded with the base security data for the chart where the expert is attached.
- The *MSXDataRec* structure contains seven data arrays stored in *MSXDataInfoRec* structures (page 35). These data arrays store all relevant price data for the security. Some of these arrays may be empty (see the discussion of empty data arrays in the *MSXDataInfoRec* section (page 35) if the security does not have data for that price field.
- Also contained in the *MSXDataRec* structure is a pointer to an array of *MSXDateTime* structures. This array contains date and time information for each data point. If a calculation function needs to access the date and time for the Nth bar of the security, it would reference the Nth element of the *psDate* array. Note that this is not a data array like the *sHigh*, *sLow*, etc.

sInd data array

- The data array *sInd* contains data for a user-selected indicator. In the case of a Custom Indicator this data array will contain the value for the chart object on which the indicator was dropped. For System Tests and Explorations, this array contains the selected plot (if there is one) of the active chart when the system Test or Exploration was started. For Experts, this array contains the selected plot (if there is one) of the chart when the Expert was attached. In all cases, if no plot is selected the *sInd* data array will be empty.
- Notice that the location of the data in these arrays is synchronized. The Nth element of each array corresponds to the same time frame.

sClose data array

- The *sClose* data array always contains the maximum data points. All other data arrays will contain equal to or less than the value of *sClose.iLastValid*.

iFirstValid, iLastValid settings

- The *iFirstValid* and *iLastValid* settings in the *sClose* data array are significant. Typically the number of data elements in this data array defines the maximum number of data elements stored in the other price arrays. This is important for determining the number of valid elements contained in the *psDate* array. For example, if the *sClose.iFirstValid* field contains 100 and the *sClose.iLastValid* field contains 200, you can be certain that the *psDate* array contains valid data at *psDate[100]* through *psDate[200]*.

Note: After a calculation is performed, the *a_psResultRec->psResultArray*'s *iLastValid* should never be greater than the *iLastValid* value of the *sClose* data array**.**

- The *iFirstValid* and *iLastValid* of *sClose* should be used to determine how much storage is available for all data arrays. All arrays have enough memory allocated to store up to *sClose.iLastValid* data points. Data returned in *a_psResultRec->psResultArray* from an MSX DLL must keep within these same storage constraints.

Note: The *a_psResultRec->psResultArray* data array returned from an MSX DLL must never have an *iFirstValid* that is less than *sClose.iFirstValid*. The *a_psResultRec->psResultArray* data array returned from an MSX DLL must never have an *iLastValid* that is greater than *sClose.iLastValid*.

Things to Remember

- Calculation functions must never modify any incoming arguments, with the exception of the result record (*a_psResultRec*). Incoming arguments are defined as 'const', where possible, in the provided templates to help ensure that no illegal modifications take place.
- Be sure to set *a_psResultRec->psResultArray->iFirstValid* and *a_psResultRec->psResultArray->iLastValid* in the returned *MSXResultRec* before returning from your function.
- If your function is returning MSX_ERROR indicating an internal error, be sure to copy an extended string message describing the cause of the error to *a_psResultRec->pszExtendedError*.
  Make sure your message string does not exceed MSX_MAXSTRING bytes.
- Never set *a_psResultRec->psResultArray->iFirstValid* less than *sClose.iFirstValid*.
- Never set *a_psResultRec->psResultArray->iLastValid* greater than *sClose.iLastValid*. Writing to *a_psResultRec->psResultArray->pfValue* beyond the value of *sClose.iLastValid* will corrupt MetaStock's heap.
- Be sure to check the *iFirstValid* and *iLastValid* of any *MSXDataInfoRec* arguments or *a_psDataRec* members you intend to use. Never assume that data will be available in any data array. If data is not available for your function to process, set *a_psResultRec->psResultArray->iFirstValid* to 0 and *a_psResultRec->psResultArray->iLastValid* to –1 to indicate that there is no valid data in the returning array. This method allows processing to continue in the most robust way possible.

User Interface Restrictions

While an MSX DLL calculation function is active, all other formula processing in MetaStock is suspended. ***For this reason, an MSX DLL must NOT, under any circumstances, request user input through message boxes or dialogs***. This includes the reporting of error conditions.

Users routinely leave MetaStock running for extended periods unattended.
MSX DLLs cannot assume that a user is available to respond to a message of any kind.
MSX DLLs must refrain from implementing *any* user interface.

Tech Note 1 – Using MSFL in an MSX DLL

If you use MSFL calls from your MSX DLL, there are a few issues you must consider.

- MetaStock loads and initializes the release version of the MSFL DLL at startup. If your MSX DLL is also using the release version of MSFL, it must not call *MSFL1_Initialize* (page 115). MSFL's initialization routine should be called only once during the lifetime of an application. If your MSX DLL calls *MSFL1_Shutdown* (page 124), then *MSFL1_Initialize*, the MSFL operations that MetaStock is performing will be corrupted. *Again, this is only if your MSX DLL is using the release version of the MSFL DLL.*
- If your MSX DLL uses the debug version of the MSFL DLL that was shipped with the MDK, you will need to call *MSFL1_Initialize* once from within your MSX DLL, and you will also need to call *MSFL1_Shutdown* when your MSX DLL is unloaded.
- The recommended approach is to make your MSX DLL respond correctly under both conditions by calling *MSFL1_GetMSFLState* (page 107) to determine if the MSFL DLL your MSX DLL uses is already loaded and initialized.
  If *MSFL1_GetMSFLState* returns MSFL_STATE_INITIALIZED, your MSX DLL is using the same MSFL DLL as MetaStock. If *MSFL1_GetMSFLState* returns MSFL_STATE_UNINITALIZED, your MSX DLL is using a different version of the MSFL DLL, and it must be initialized via a call to *MSFL1_Initialize*. If this is the case, set a flag in your MSX DLL code that can be checked during the DLLMain's DLL_PROCESS_DETACH logic that will indicate that *MSFL1_Shutdown* is required.

Note: The *MSFL1_GetMSFLState* logic cannot be performed during the DLLMain DLL_PROCESS_ATTACH, as MetaStock loads all MSX DLLs before it initializes the MSFL DLL. If your MSX DLL initializes the release version of MSFL before MetaStock gets a chance to, MetaStock will fail to load. Make the check from somewhere inside one of your exported user functions, setting a flag to ensure that the Initialize call is not made more than once.

- If you plan to distribute your MSX DLL (containing MSFL logic) to other MetaStock users:
  - Other MetaStock users must have the same version of the MSFL DLL that your MSX DLL is expecting to use. It is essential that your MSX DLL references the release build of MSFL if your MSX DLL will be executed by another MetaStock user. The debug version of MSFL DLL will (almost certainly) not be present on their system.
  - Realize that this approach may require a new version of your MSX DLL for each new version of MSFL that may be released in the future. For example, as of this writing, the current version of MSFL is `MSFL72.DLL`. If a future version of MetaStock is shipped with `MSFL73.DLL`, your MSX DLL will not find the `MSFL72.DLL` it expects, and will fail to load.
  - The recommended approach is to update your MSX DLL with each release of a new version of MSFL. This approach will ensure that your MSX DLL is tested with each new release, and that it will not attempt to load an untested version of the MSFL DLL.
  - An alternative approach is to dynamically load the MSFL library via the "LoadLibrary" system call, and then resolve each MSFL function you use via the "GetProcAddress" system call. This will allow you to make a minimal change to your MSX DLL to maintain compatibility with future MSFL releases. Attempt to load the most recent MSFL DLL first. If that fails, attempt to load the next most recent, etc. Your DLL must have a method of obtaining the current MSFL DLL name. You can hard-code the names of all the potential MSFL DLLs that may be present, retrieve the names from an external file that you can provide or document for your user to maintain, or search the windows system folder for "MSFL*.DLL" files. Searching for the file name in the system folder carries the potential risk of eventually encountering a future version of the MSFL DLL that is incompatible with older MetaStock files. In that case, your MSX DLL would fail to load or to operate correctly.

MSX Index

Sample DLL Programs

The following three programs demonstrate complete source code for implementing a moving average in C, Delphi Pascal, and PowerBASIC/DLL.

“C” Example

```
//==================================================================
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "MSXStruc.h"

// we don't want C++ name-mangling
#ifdef __cplusplus
extern "C" {
#endif


//---------------------------------------------------------------------------

BOOL __stdcall MSXInfo (MSXDLLDef *a_psDLLDef)
{
  // copy in your copyright information...
  strncpy (a_psDLLDef->szCopyright, "Copyright (c) ColdFront Logic, Inc., 2000",
                      sizeof (a_psDLLDef->szCopyright)-1);
  a_psDLLDef->iNFuncs = 1; // One calculation function;
  a_psDLLDef->iVersion = MSX_VERSION;
  return MSX_SUCCESS;
}


//---------------------------------------------------------------------------

BOOL __stdcall MSXNthFunction (int a_iNthFunc, MSXFuncDef *a_psFuncDef)
{
  BOOL l_bRtrn = MSX_SUCCESS;

  switch (a_iNthFunc)
  {
  case 0: // a_iNthFunc is zero-based
    strcpy (a_psFuncDef->szFunctionName, "MyMov");
    strcpy (a_psFuncDef->szFunctionDescription, "My Moving Average");
    a_psFuncDef->iNArguments = 3; // 3 arguments: data array, periods, method
    break;
  default:
    l_bRtrn = MSX_ERROR;
    break;
  }
  return l_bRtrn;
}


//---------------------------------------------------------------------------

BOOL __stdcall MSXNthArg (int a_iNthFunc, int a_iNthArg,
                         MSXFuncArgDef *a_psFuncArgDef)
{
  BOOL l_bRtrn = MSX_SUCCESS;
  a_psFuncArgDef->iNCustomStrings = 0;

  switch (a_iNthFunc)
  {
  case 0:
    switch (a_iNthArg)
    {
```

```
    case 0:
      a_psFuncArgDef->iArgType = MSXDataArray; // data array
      strcpy (a_psFuncArgDef->szArgName, "DataArray");
      break;
    case 1:
      a_psFuncArgDef->iArgType = MSXNumeric; // Numeric
      strcpy (a_psFuncArgDef->szArgName, "Period");
      break;
    case 2:
      a_psFuncArgDef->iArgType = MSXCustom; // CustomType
      a_psFuncArgDef->iNCustomStrings = 4;
      strcpy (a_psFuncArgDef->szArgName, "Method");
      break;
    default:
      l_bRtrn = MSX_ERROR;
      break;
    }
    break;
  default:
    l_bRtrn = MSX_ERROR;
    break;
  }
  return l_bRtrn;
}
//-----------------------------------------------------------------------------
BOOL __stdcall MSXNthCustomString (int a_iNthFunc, int a_iNthArg,
                                   int a_iNthString,
                                   MSXFuncCustomString *a_psCustomString)
{
  BOOL l_bRtrn = MSX_SUCCESS;

  typedef struct
  {
    char *szString;
    int  iID;
  } LocalStringElement;

  LocalStringElement l_sTheStrings[] =
  {
    {"Simple",  0}, {"S", 0},
    {"Weighted",1}, {"W", 1}
  };

  switch (a_iNthFunc)
  {
  case 0:
    switch (a_iNthArg)
    {
    case 2:
      if(a_iNthString >= 0 && a_iNthString < NMyMovCustStrings)
      {
        strncpy (a_psCustomString->szString,
                        l_sTheStrings[a_iNthString].szString,
                 MSX_MAXSTRING-1);
        a_psCustomString->iID = l_sTheStrings[a_iNthString].iID;
      }
      break;
    default:
      l_bRtrn = MSX_ERROR;
      break;
    }
    break;
  default:
    l_bRtrn = MSX_ERROR;
    break;
  }
  return l_bRtrn;
}

// **************************************************************************
// This local utility function is used to help ensure that no overflows
//   or underflows will occur during calculations.  The MSXTest program
//   Stress Test function will call your DLL with a wide range of values,
//   including positive and negative values of FLT_MAX and FLT_MIN.
// Perform all intermediate calculations using doubles and then force the
//   results into the range of a float.
// *********************************************************************
#define MSXMax(a,b) (((a) > (b)) ? (a) : (b))
#define MSXMin(a,b) (((a) < (b)) ? (a) : (b))
double ForceFloatRange (double a_lfDbl)
{
  if (a_lfDbl > 0.0)
  {
    a_lfDbl = MSXMin (a_lfDbl, double(FLT_MAX)); // force pos num <= FLT_MAX
    a_lfDbl = MSXMax (a_lfDbl, double(FLT_MIN)); // force pos num >= FLT_MIN
  }
  else
  {
    if (a_lfDbl < 0.0)
    {
      a_lfDbl = MSXMax (a_lfDbl, double(-FLT_MAX)); // force neg num >= -FLT_MAX
      a_lfDbl = MSXMin (a_lfDbl, double(-FLT_MIN)); // force neg num <= -FLT_MIN
    }
  }
  return a_lfDbl;
}

//---------------------------------------------------------------------
// This is an example of a local function used for calculations.  This
//   one calculates a moving average on the source data array, and puts
//   the results in the result array.  It differentiates its processing
//   based on whether the moving average is to be weighted or not.
//---------------------------------------------------------------------
void MovingAverage (const MSXDataInfoRec *a_psSrc, MSXDataInfoRec *a_psRslt,
                    int a_iPeriod, BOOL a_bIsWeighted)
{
  int    l_iIndex = a_psSrc->iFirstValid;
  int    l_iMaxIndex = a_psSrc->iLastValid;
  double l_lfSum = 0.0;
  double l_lfDivisor;
  int    i;

  if (a_bIsWeighted)
    // sum of the digits formula
    l_lfDivisor = double(a_iPeriod) * (double(a_iPeriod)+1.0) / 2.0;
  else
    l_lfDivisor = double(a_iPeriod);
  while ((l_iIndex + a_iPeriod - 1) <= l_iMaxIndex)
  {
    l_lfSum = 0.0;
    for (i=0; i<a_iPeriod; i++) {
      if (a_bIsWeighted)
        l_lfSum += a_psSrc->pfValue[l_iIndex+i] * (i + 1.0); // weighted
      else
        l_lfSum += a_psSrc->pfValue[l_iIndex+i];              // simple
      l_lfSum = ForceFloatRange (l_lfSum);
    }
    a_psRslt->pfValue[l_iIndex + a_iPeriod - 1] =
                        float (ForceFloatRange(l_lfSum / l_lfDivisor));
    l_iIndex++;
  }
  a_psRslt->iFirstValid = a_psSrc->iFirstValid + a_iPeriod - 1;
  a_psRslt->iLastValid = l_iMaxIndex;
}

//------------------------------------------------------------------------
// The following function demonstrates use of three argument types:
//   MSXDataArray, MSXNumeric and MSXCustom.
//   A MovingAverage is calculated on the input DataArray for input Periods.
//   Two moving average methods are available, specified by the Custom ID.
//------------------------------------------------------------------------


BOOL __stdcall MyMov (const MSXDataRec *a_psDataRec,
                      const MSXDataInfoRecArgsArray *a_psDataInfoArgs,
                      const MSXNumericArgsArray *a_psNumericArgs,
                      const MSXStringArgsArray *a_psStringArgs,
                      const MSXCustomArgsArray *a_psCustomArgs,
                      MSXResultRec *a_psResultRec)
{
  BOOL l_bRtrn = MSX_SUCCESS;
  // We expect 3 arguments, 1 DataArray, 1 Numeric and 1 Custom, in that order
  // The arguments will be found at:
  //   DataArray: a_psDataInfoArgs[0]
  //   Numeric  : a_psNumericArgs->fNumerics[0];
  //   Custom   : a_psCustomArgs->iCustomIDs[0];
  const MSXDataInfoRec *l_psData;
  int                  l_iPeriod;
  int                  l_iMethod;
  int                  l_iIndex;
  int                  l_iMaxIndex;

  if (a_psDataInfoArgs->iNRecs == 1 &&
      a_psNumericArgs->iNRecs == 1 &&
      a_psCustomArgs->iNRecs == 1)
  {
    l_psData = a_psDataInfoArgs->psDataInfoRecs[0];
    l_iPeriod = int(a_psNumericArgs->fNumerics[0]); // truncate
    l_iMethod = a_psCustomArgs->iCustomIDs[0];
    l_iIndex = l_psData->iFirstValid;
    l_iMaxIndex = l_psData->iLastValid;

    if (l_iPeriod > 0 && (l_iIndex + l_iPeriod - 1) <= l_iMaxIndex)
    {
      switch (l_iMethod)
      {
      case 0: // Simple
      case 1: // Weighted
        MovingAverage (l_psData, a_psResultRec->psResultArray, l_iPeriod, l_iMethod);
        break;
      default:
        strncpy (a_psResultRec->szExtendedError, "Invalid method",
                 sizeof(a_psResultRec->szExtendedError)-1);
        a_psResultRec->psResultArray->iFirstValid = 0;
        a_psResultRec->psResultArray->iLastValid = -1;
        l_bRtrn = MSX_ERROR;
        break;
      }
    }
    else
    {
      a_psResultRec->psResultArray->iFirstValid = 0;
      a_psResultRec->psResultArray->iLastValid = -1;
    }
  }
  return l_bRtrn;
}

#ifdef __cplusplus
}
#endif


//=======================================================
This is the DEF file (ColdFront.DEF) for the above example:

LIBRARY
DESCRIPTION 'MSX External Functions.'
EXPORTS
  MSXInfo
  MSXNthFunction
  MSXNthArg
  MSXNthCustomString
  MyMov
```

Delphi Pascal Example

```
library DelphiSampleDLL;   // Causes a .DLL file to be built
Uses SysUtils,          // Brings in szString funcs and exception trapping
     Math;              // Brings in various math functions
{$I MSXStruc.inc}             // Include the MSX datastructure definitions

function MSXInfo (var a_psDLLDef: MSXDLLDef): LongBool; stdcall;
begin
  StrLCopy (a_psDLLDef.szCopyright, 'Copyright (c) FantasticFuncs, 2000',
            MSX_MAXSTRING-1);
  a_psDLLDef.iNFuncs := 1;
  a_psDLLDef.iVersion := MSX_VERSION;
  MSXInfo := MSX_SUCCESS;
end;

function MSXNthFunction (a_iNthFunc: Integer;
                         var a_psFuncDef: MSXFuncDef): LongBool; stdcall;
  var l_bRtrn : LongBool;
begin
  l_bRtrn := MSX_SUCCESS;
  case a_iNthFunc of
    0: begin
         StrCopy(a_psFuncDef.szFunctionName, 'MyMov');
         StrCopy(a_psFuncDef.szFunctionDescription, 'My Moving Average');
         a_psFuncDef.iNArguments := 3; // 3 arguments: data array, periods, method
       end;
    else
      l_bRtrn := MSX_ERROR;
  end;
  MSXNthFunction := l_bRtrn;
end;

function MSXNthArg(a_iNthFunc: Integer; a_iNthArg: Integer;
                   var a_psFuncArgDef: MSXFuncArgDef): LongBool; stdcall;
 var l_bRtrn : LongBool;
begin
 l_bRtrn := MSX_SUCCESS;
 case a_iNthFunc of
   0:
     case a_iNthArg of
       0: begin
            a_psFuncArgDef.iArgType := MSXDataArray;
            StrCopy(a_psFuncArgDef.szArgName, 'DataArray');
          end;
       1: begin
            a_psFuncArgDef.iArgType := MSXNumeric;
            StrCopy(a_psFuncArgDef.szArgName, 'Period');
          end;
       2: begin
            a_psFuncArgDef.iArgType := MSXCustom;
            StrCopy(a_psFuncArgDef.szArgName, 'Method');
            a_psFuncArgDef.iNCustomStrings := 4;
          end;
       else
         l_bRtrn := MSX_ERROR;
     end;
   else
     l_bRtrn := MSX_ERROR;
   end;
   MSXNthArg := l_bRtrn;
end;

function MSXNthCustomString(a_iNthFunc: Integer;
                            a_iNthArg: Integer;
                            a_iNthString: Integer;
                            var a_psCustomString: MSXFuncCustomString):LongBool;
                        stdcall;
  var l_bRtrn : LongBool;
begin
  l_bRtrn := MSX_SUCCESS;

  case a_iNthFunc of
    0:
      case a_iNthArg of
        2:
          // see MSXTmplt.pas to see an alternative to the nested
          //   switch used below
          case a_iNthString of
            0: begin
                 StrCopy(a_psCustomString.szString, 'Simple');
                 a_psCustomString.iID := 0;
               end;
            1: begin
                 StrCopy(a_psCustomString.szString, 'S');
                 a_psCustomString.iID := 0;
               end;
            2: begin
                 StrCopy(a_psCustomString.szString, 'Weighted');
                 a_psCustomString.iID := 1;
               end;
            3: begin
                 StrCopy(a_psCustomString.szString, 'W');
                 a_psCustomString.iID := 1;
               end;
            else
              l_bRtrn := MSX_ERROR;
          end;
        else
          l_bRtrn := MSX_ERROR;
      end;
    else
      l_bRtrn := MSX_ERROR;
  end;
  MSXNthCustomString := l_bRtrn;
end;


// ***********************************************************************
// This local utility function is used to help ensure that no overflows
//   or underflows will occur during calculations.  The MSXTest program
//   Stress Test function will call your DLL with a wide range of values,
//   including positive and negative values of MaxSingle and MinSingle.
// Perform all intermediate calculations using Doubles and then force the
//   results into the range of a Single.
// ***********************************************************************

function ForceSingleRange (a_lfDbl:Double) : Double;
begin
  if (a_lfDbl > 0.0) then
    begin
      if (a_lfDbl > MaxSingle) then
        a_lfDbl := MaxSingle
      else
        if (a_lfDbl < MinSingle) then
          a_lfDbl := MinSingle;
    end
  else
    if (a_lfDbl < 0.0) then
      begin
        if (a_lfDbl < -MaxSingle) then
          a_lfDbl := -MaxSingle
        else
          if (a_lfDbl > -MinSingle) then
            a_lfDbl := -MinSingle;
      end;
  ForceSingleRange := a_lfDbl;
end;


//***************************************************************************
// This is an example of a local procedure used for calculations.  This one
//   calculates a simple moving average on the source data array, and puts the
//   results in the result array.
// ****************************************************************************


procedure SimpleMovingAverage (const a_psSrc: PMSXDataInfoRec;
                               var   a_psResult: MSXDataInfoRec;
```

```
                                       a_iPeriod : Integer);
  var l_iIndex   : Integer;
      l_iMaxIndex: Integer;
      l_lfSum     : Double;
      i          : Integer;
begin
  l_iIndex := a_psSrc.iFirstValid;
  l_iMaxIndex := a_psSrc.iLastValid;
  l_lfSum := 0.0;

  for i:= 0 to a_iPeriod-1 do
    l_lfSum := ForceSingleRange(l_lfSum + a_psSrc.pfValue[l_iIndex+i]);
  l_lfSum := ForceSingleRange (l_lfSum);
  l_iIndex := l_iIndex + a_iPeriod - 1;
  while l_iIndex <= l_iMaxIndex do
    begin
      a_psResult.pfValue[l_iIndex] := ForceSingleRange(l_lfSum / a_iPeriod);
      l_iIndex := l_iIndex + 1;
      l_lfSum := l_lfSum - a_psSrc.pfValue[l_iIndex-a_iPeriod];
      if l_iIndex <= l_iMaxIndex then
        l_lfSum := ForceSingleRange(l_lfSum + a_psSrc.pfValue[l_iIndex]);
    end;
  a_psResult.iFirstValid := a_psSrc.iFirstValid + (a_iPeriod - 1);
  a_psResult.iLastValid := l_iMaxIndex;
end;


//*****************************************************************************
// This is an example of a local procedure used for calculations.  This one
//   calculates a weighted moving average on the source data array, and puts the
//   results in the result array.
//****************************************************************************
procedure WeightedMovingAverage (const a_psSrc: PMSXDataInfoRec;
                                 var   a_psResult: MSXDataInfoRec;
                                       a_iPeriod : Integer);
  var l_iIndex   : Integer;
      l_iMaxIndex: Integer;
      l_lfSum    : Double;
      l_lfDivisor: Double;
      i          : Integer;
begin
  l_iIndex := a_psSrc.iFirstValid;
  l_iMaxIndex := a_psSrc.iLastValid;
  // Sum of Digits formula
  l_lfDivisor := ForceSingleRange(a_iPeriod * (a_iPeriod+1.0) / 2.0);

  while ((l_iIndex + a_iPeriod - 1) <= l_iMaxIndex) do
    begin
      l_lfSum := 0.0;
      for i := 0 to a_iPeriod-1 do
        l_lfSum := ForceSingleRange(l_lfSum + a_psSrc.pfValue[l_iIndex+i] *
                  (i + 1.0));
      a_psResult.pfValue[l_iIndex + a_iPeriod - 1] :=
                           ForceSingleRange(l_lfSum / l_lfDivisor);
      l_iIndex := l_iIndex + 1;
    end;
  a_psResult.iFirstValid := a_psSrc.iFirstValid + a_iPeriod - 1;
  a_psResult.iLastValid := l_iMaxIndex;
end;

// ----------------------------------------------------------------------------
// The following function demonstrates the use of three argument types:
//   MSXDataArray, MSXNumeric and MSXCustom.
//   A Moving Average is calculated on the input DataArray for input Periods.
//   Two moving average methods are available, specified the the Custom ID.
// ----------------------------------------------------------------------------

function MyMov (const a_psDataRec: PMSXDataRec;
                const a_psDataInfoArgs: PMSXDataInfoRecArgsArray;
                const a_psNumericArgs: PMSXNumericArgsArray;
                const a_psStringArgs: PMSXStringArgsArray;
                const a_psCustomArgs: PMSXCustomArgsArray;
                var a_psResultRec: MSXResultRec): LongBool; stdcall;

  var l_bRtrn    : LongBool;
```

```
      l_psData   : PMSXDataInfoRec;
      l_iPeriod  : Integer;
      l_iMethod  : Integer;
      l_iIndex   : Integer;
      l_iMaxIndex: Integer;
      l_sTmpRec  : MSXDataInfoRec;
begin
  // We expect 3 arguments, 1 DataArray, 1 Numeric and 1 Custom.
  // The arguments will be found at:
  //   DataArray: a_psDataInfoArgs->psDataInfoRecs[0]
  //   Numeric  : a_psNumericArgs->fNumerics[0]
  //   Custom   : a_psCustomArgs->iCustomIDs[0]
  if ((a_psDataInfoArgs.iNRecs = 1) and
      (a_psNumericArgs.iNRecs = 1) and
      (a_psCustomArgs.iNRecs = 1)) then
    begin
      l_bRtrn := MSX_SUCCESS;
      l_psData := a_psDataInfoArgs.psDataInfoRecs[0];
      l_iPeriod := Trunc (a_psNumericArgs.fNumerics[0]);
      l_iMethod := a_psCustomArgs.iCustomIDs[0];
      l_iIndex := l_psData.iFirstValid;
      l_iMaxIndex := l_psData.iLastValid;

      if (l_iPeriod > 0) and ((l_iIndex + l_iPeriod - 1) <= l_iMaxIndex) then
        case l_iMethod of
        0: SimpleMovingAverage (l_psData, a_psResultRec.psResultArray^,
                                l_iPeriod);
        2: WeightedMovingAverage (l_psData, a_psResultRec.psResultArray^,
                                  l_iPeriod);
        else
          begin
            StrLCopy(a_psResultRec->szExtendedError, ‘Invalid method’,
                     MSX_MAXSTRING-1);
            l_bRtrn := MSX_ERROR;
          end
        end
      else
        begin
          a_psResultRec.psResultArray.iFirstValid := 1;
          a_psResultRec.psResultArray.iLastValid := 0;
        end
    end
  else // wrong number of arguments passed!
    begin
      StrLCopy (a_psResultRec.szExtendedError,
                'Incorrect number of arguments',
                MSX_MAXSTRING-1);
      l_bRtrn := MSX_ERROR;
    end;

  if (l_bRtrn <> MSX_SUCCESS) then
    begin
      a_psResultRec.psResultArray.iFirstValid := 0;
      a_psResultRec.psResultArray.iLastValid := -1;
    end;

  MyMov := l_bRtrn;
end;

exports
 MSXInfo,
 MSXNthFunction,
 MSXNthArg,
 MSXNthCustomString,
 MyMov;

begin

end.
```

PowerBASIC/DLL Example

```
#COMPILE DLL                ' create a dll
OPTION EXPLICIT             ' require all variables to be declared
#INCLUDE "WIN32API.INC"     ' required equates and windows prototypes
#INCLUDE "MSXStruc.BAS"     ' MSX Data Structures

FUNCTION MSXInfo SDECL ALIAS "MSXInfo" (a_psDLLDef AS MSXDLLDef PTR) _
                                        EXPORT AS LONG
  ' copy in your copyright information...
  @a_psDLLDef.szCopyright =  "Copyright (c) PBDemo Inc., 2000"
  ' Set the number of functions we are exporting
  @a_psDLLDef.iNFuncs = 1 ' One calculation function
  @a_psDLLDef.iVersion = %MSX_VERSION
  MSXInfo = %MSX_SUCCESS
END FUNCTION

FUNCTION MSXNthFunction SDECL ALIAS "MSXNthFunction" ( _
                                     BYVAL a_iNthFunc AS LONG, _
                        a_psFuncDef AS MSXFuncDef PTR) EXPORT AS LONG

  MSXNthFunction = %MSX_SUCCESS

  SELECT CASE a_iNthFunc
  CASE 0  ' a_iNthFunc is zero-based
    @a_psFuncDef.szFunctionName = "MyMov"
    @a_psFuncDef.szFunctionDescription = "My Moving Average"
    ' 3 arguments: data array, periods, method
    @a_psFuncDef.iNArguments = 3
  CASE ELSE
    MSXNthFunction = %MSX_ERROR
  END SELECT
END FUNCTION

FUNCTION MSXNthArg SDECL ALIAS "MSXNthArg" ( _
                                BYVAL a_iNthFunc AS LONG, _
                                BYVAL a_iNthArg AS LONG, _
                        a_psFuncArgDef AS MSXFuncArgDef PTR) EXPORT AS LONG

  MSXNthArg = %MSX_SUCCESS

  @a_psFuncArgDef.iNCustomStrings = 0

  SELECT CASE a_iNthFunc
  CASE 0
    SELECT CASE a_iNthArg
    CASE 0
      @a_psFuncArgDef.iArgType = %MSXDataArray ' DataArray;
      @a_psFuncArgDef.szArgName = "DataArray"
    CASE 1
      @a_psFuncArgDef.iArgType = %MSXNumeric ' Numeric
      @a_psFuncArgDef.szArgName = "Period"
    CASE 2
      @a_psFuncArgDef.iArgType = %MSXCustom ' CustomType
      @a_psFuncArgDef.iNCustomStrings = 4
      @a_psFuncArgDef.szArgName = "Method"
    CASE ELSE
      MSXNthArg = %MSX_ERROR
    END SELECT
  CASE ELSE
    MSXNthArg = %MSX_ERROR
  END SELECT
END FUNCTION

FUNCTION MSXNthCustomString SDECL ALIAS "MSXNthCustomString" ( _
                                    BYVAL a_iNthFunc AS LONG, _
                                    BYVAL a_iNthArg AS LONG, _
                                    BYVAL a_iNthString AS LONG, _
                        a_psCustomString AS MSXFuncCustomString PTR) EXPORT AS LONG

  MSXNthCustomString = %MSX_SUCCESS
  @a_psCustomString.szString = ""
  @a_psCustomString.iID = -1
  SELECT CASE a_iNthFunc
  CASE 0
    SELECT CASE a_iNthArg
    CASE 2
      SELECT CASE a_iNthString
      CASE 0
        @a_psCustomString.szString = "Simple"
        @a_psCustomString.iID = 0
      CASE 1
        @a_psCustomString.szString = "S"
        @a_psCustomString.iID = 0
      CASE 2
        @a_psCustomString.szString = "Weighted"
        @a_psCustomString.iID = 1
      CASE 3
        @a_psCustomString.szString = "W"
        @a_psCustomString.iID = 1
      CASE ELSE
        MSXNthCustomString = %MSX_ERROR
      END SELECT
    CASE ELSE
      MSXNthCustomString = %MSX_ERROR
    END SELECT
  CASE ELSE
    MSXNthCustomString = %MSX_ERROR
  END SELECT
END FUNCTION


' ************************************************************************
' This local utility function is used to help ensure that no overflows
'   or underflows will occur during calculations.  The MSXTest program
'   Stress Test function will call your DLL with a wide range of values,
'   including positive and negative values of FLT_MAX AND FLT_MIN.
' Perform all intermediate calculations using doubles and then force the
'   results into the range of a single.
' ************************************************************************


FUNCTION ForceFloatRange (BYVAL a_lfDbl AS DOUBLE) AS DOUBLE
    LOCAL s_MaxSingle AS DOUBLE
    LOCAL s_MinSingle AS DOUBLE
    s_MaxSingle = 3.371E+38
    s_MinSingle = 8.431E-37

    IF a_lfDbl > 0.0 THEN
' force pos num <= s_MaxSingle
a_lfDbl = MIN (a_lfDbl, s_MaxSingle)
' force pos num >= s_MinSingle
       a_lfDbl = MAX (a_lfDbl, s_MinSingle)
    ELSE
       IF a_lfDbl < 0.0 THEN
      ' force neg num >= -s_MaxSingle
          a_lfDbl = MAX (a_lfDbl, -s_MaxSingle)
      ' force neg num <= -s_MinSingle
          a_lfDbl = MIN (a_lfDbl, -s_MinSingle)
       END IF
    END IF
    ForceFloatRange = a_lfDbl
END FUNCTION

'------------------------------------------------------------------------
' This is an example of a local function used for calculations.  This
'   one calculates a moving average on the source data array, and puts
'   the results in the result array.  It differentiates its processing
'   based on whether the moving average is to be weighted or not.
'-----------------------------------------------------------------------
SUB MovingAverage (a_psSrc AS MSXDataInfoRec PTR, _
                   a_psRslt AS MSXDataInfoRec PTR, _
                   BYVAL a_iPeriod AS LONG, _
                   BYVAL a_bIsWeighted AS LONG)
  LOCAL l_iIndex AS LONG
  LOCAL l_iMaxIndex AS LONG
  LOCAL l_lfSum AS DOUBLE
  LOCAL l_lfDbl AS DOUBLE
  LOCAL l_fDivisor AS DOUBLE
  LOCAL i AS INTEGER

  l_iIndex = @a_psSrc.iFirstValid
  l_iMaxIndex = @a_psSrc.iLastValid
  l_lfSum = 0.0

  IF a_bIsWeighted = %TRUE THEN
     ' sum of the digits formula
    l_fDivisor = CDBL(a_iPeriod) * (CDBL(a_iPeriod)+1.0) / 2.0
  ELSE
    l_fDivisor = CDBL(a_iPeriod)
  END IF
  l_fDivisor = CSNG(ForceFloatRange (l_fDivisor))
  IF l_fDivisor = 0.0 THEN
    l_fDivisor = 1.0
  END IF
  WHILE ((l_iIndex + a_iPeriod - 1) <= l_iMaxIndex)
    l_lfSum = 0.0
    FOR i = 0 TO a_iPeriod-1
      IF a_bIsWeighted = %TRUE THEN
    ' weighted
        l_lfSum = l_lfSum + @a_psSrc.@pfValue[l_iIndex+i] * (i + 1.0)
      ELSE
    ' simple
        l_lfSum = l_lfSum + @a_psSrc.@pfValue[l_iIndex+i]
      END IF
    NEXT i

    l_lfSum = ForceFloatRange(l_lfSum)
    l_lfDbl = ForceFloatRange(l_lfSum / l_fDivisor)
    @a_psRslt.@pfValue[l_iIndex + a_iPeriod - 1] = CSNG(l_lfDbl)
    l_iIndex = l_iIndex + 1
  WEND
  @a_psRslt.iFirstValid = @a_psSrc.iFirstValid + a_iPeriod - 1
  @a_psRslt.iLastValid = l_iMaxIndex
END SUB

'-----------------------------------------------------------------------------
' The following function demonstrates use of three argument types:
'   MSXDataArray, MSXNumeric and MSXCustom.
'   A MovingAverage is calculated on the input DataArray for input Periods.
'   Three moving average methods are available, specified by the Custom ID.
'-----------------------------------------------------------------------------
FUNCTION MyMov SDECL ALIAS "MyMov" ( _
                            a_psDataRec AS MSXDataRec PTR, _
                            a_psDataInfoArgs AS MSXDataInfoRecArgsArray PTR, _
                            a_psNumericArgs AS MSXNumericArgsArray PTR, _
                            a_psStringArgs AS MSXStringArgsArray PTR, _
                            a_psCustomArgs AS MSXCustomArgsArray PTR, _
                            a_psResultRec AS MSXResultRec PTR) EXPORT AS LONG
LOCAL l_bRtrn AS LONG

  ' We expect 3 arguments, 1 DataArray, 1 Numeric and 1 Custom, in that order
  ' The arguments will be found at:
  '   DataArray: @a_psDataInfoArgs.psDataInfoRecs(0)
  '   Numeric  : @a_psNumericArgs.fNumerics(0);
  '   Custom   : @a_psCustomArgs.iCustomIDs(0);

  LOCAL l_psData AS MSXDataInfoRec PTR
  LOCAL l_iPeriod AS LONG
  LOCAL l_iMethod AS LONG
  LOCAL l_iIndex AS LONG
  LOCAL l_iMaxIndex AS LONG

  l_bRtrn = %MSX_SUCCESS

  IF (@a_psDataInfoArgs.iNRecs = 1 AND _
      @a_psNumericArgs.iNRecs = 1 AND _
      @a_psCustomArgs.iNRecs = 1) THEN
    l_psData = @a_psDataInfoArgs.psDataInfoRecs(0)
    ' truncate any fractional period
    l_iPeriod = FIX(@a_psNumericArgs.fNumerics(0))
    l_iMethod = @a_psCustomArgs.iCustomIDs(0)
```

```
      l_iIndex = @l_psData.iFirstValid
      l_iMaxIndex = @l_psData.iLastValid

      IF (l_iPeriod > 0 AND (l_iIndex + l_iPeriod - 1) <= l_iMaxIndex) THEN
          SELECT CASE l_iMethod
          CASE 0 ' Simple
               CALL MovingAverage (@l_psData, @a_psResultRec.@psResultArray, _
                                   l_iPeriod, %FALSE)
        CASE 1 ' Weighted
          CALL MovingAverage (@l_psData, @a_psResultRec.@psResultArray, _
                              l_iPeriod, %TRUE)
        CASE ELSE
          ' Somehow we got called with an invalid argument
          @a_psResultRec.szExtendedError =  "Undefined method argument"
          l_bRtrn = %MSX_ERROR ' report this AS an ERROR
        END SELECT
      ELSE
        @a_psResultRec.@psResultArray.iFirstValid = 0
        @a_psResultRec.@psResultArray.iLastValid = -1
      END IF
    ELSE ' wrong number of arguments!
      @a_psResultRec.szExtendedError = "Wrong number of arguments"
      l_bRtrn = %MSX_ERROR
    END IF

    IF (l_bRtrn <> %MSX_SUCCESS) THEN ' only for serious errors...
      @a_psResultRec.@psResultArray.iFirstValid = 0
      @a_psResultRec.@psResultArray.iLastValid = -1
    END IF

    MyMov = l_bRtrn
  END FUNCTION
```

References

See the following included source files for data structure definitions and additional examples: *MSXStruc.h*, *MSXTmplt.cpp*, *CSampleDLL.c*, *MSXStruc.inc*, *MSXTmplt.pas*, *DelphiSampleDLL.pas*, *MSXStruc.bas*, *MSXTmplt.bas*, and *PBSampleDLL.bas*.

MetaStock File Library (MSFL)

Introduction

The MetaStock File Library (MSFL) Application Programming Interface (API) provides developers with the tools necessary to integrate applications with the MetaStock data format. The MSFL API provides support for reading security names and price data.

This manual contains the function descriptions and instructions necessary to access MetaStock data files using the MSFL. It assumes that the user is an experienced programmer familiar with security price data and calling dynamic link library (DLL) functions. The primary function of the MSFL is to remove many of the complexities of accessing MetaStock data files from the application program through a set of easy-to-use functions.

What's New

The 9.0 version of the MSFL has only minor changes. The PowerBasic sample app for the MSFL and MSX uses the latest version (i.e. version 7.03).

See the section titled "Change Record" on page 137 for details of all other changes.

Application Integration

Using the MSFL DLL in your application is different for each development environment. Common to all development platforms is the need for Windows to locate and load the MSFL DLL. The MSFL DLL, in order of preference, should be copied to the Windows system directory, application directory, or other directory in the search path.

Note: The version of the MSFL DLL should be checked before overwriting an existing copy. Refer to the *GetFileVersionInfo* function in the Win32™ SDK.

Below are basic instructions for several of the most common development environments. Refer to the development documentation for specifics on using it with third-party DLL's.

C/C++

The following steps must be taken to create a C/C++ application that uses the MSFL DLL.

1. Set the include path for the *msfl.h* header file. Refer to the compiler documentation for the specifics of setting an include path.
2. Add the link library to the project.
   Link libraries for Microsoft Visual C++ 6.0 and Borland C++ Builder 4 are provided. Refer to the compiler documentation for the specifics of adding a library to the project.
3. If you are using a different compiler or the link libraries provided are incompatible with the compiler version you are using, you should be able to build the link libraries using the definition file (i.e. *msfl.def*) and the tools provided with the compiler. For example, Visual C++ uses `LIB` with the `/DEF` switch, while Borland C++ Builder uses the `IMPLIB` utility program.

Visual Basic

The following points should be observed to create a Visual Basic application that uses the MSFL DLL.

1. Add the `msfl.bas` module to the project, it can be found in the `msfl/include` folder. The `msfl.bas` module contains the MSFL function and type declares.
2. Take care when using DLL procedures.
   Microsoft Visual Basic cannot verify that you are passing correct values to the MSFL DLL procedures. If you pass incorrect values, the procedure may fail, which may cause your application to shut down. This doesn't cause permanent harm to your application, but may cause data corruption and require the user to reload and restart the application.
3. Be sure to check the return values of MSFL functions to test for errors or messages.
   A convenient error testing function is provided in the `msflutil.bas` module, which also resides in the `msfl/include` folder.
4. Special attention must be taken when dealing with string data.
   Like the Windows API, the MSFL uses null-terminated strings. These strings differ from those used in Visual Basic. In addition, variable- and fixed-length strings differ in Visual Basic; for example, comparing a fixed-length string containing “ABC” may not equal a variable-length string containing the exact same text.

   The `msflutil.bas` module, which resides in the `msfl/include` folder, provides two convenient string conversion functions.
   - **NullTerminate(strNullMe As String) As String**
     NullTerminate null-terminates a variable or fixed length string.
     All fixed-length strings must be null-terminated before calling any of the MSFL functions.
   - **Extract(strFixed As String, strJunk As String) As String**
     Extract returns a variable-length string containing the legitimate portion of a string. Pass a string to the *strJunk* parameter which contains the illegitimate character, typically either a space or null (i.e. `Chr(0)`). *Extract* should be used on all strings returned from the MSFL.

Delphi

The following steps must be taken to create a Delphi form that uses the MSFL DLL.

1. Add the `msfl.pas` unit to the project; it can be found in the `msfl/include` folder. The `msfl.pas` unit contains the MSFL constant, record and function declarations.
2. Add the MSFL unit to the form’s **uses** clause.

PowerBASIC

The following points should be observed to create a PowerBASIC application that uses the MSFL DLL.

1. Ensure that the `msfl.inc` module is included in any file of your project that makes MSFL calls. The `msfl.inc` module can be found in the `msfl/include` folder. The `msfl.inc` module contains the MSFL function and type declares.
2. Be sure to check the return values of MSFL functions to test for errors or messages. A convenient error testing function is provided in the *msfl.inc* module.

Getting Help

Due to the complexity of the programming languages and development environments, Equis International is only able to provide minimal technical support for the MetaStock File Library (MSFL). We will help in understanding how to use the MSFL, but we cannot aid in writing or debugging your application.

This manual explains the use of the MSFL, but not the programming techniques required to effectively use it. Equis International provides this library as a tool to access MetaStock data, but how it is used is up to you.

CAUTION: Some functions in the MSFL can cause loss of the user's data if used improperly or inappropriately. Equis International shall not be responsible for any damages of any type caused by application programs that use the MSFL.

See "Getting Help" on page 4 for more information on obtaining Technical Support for the MetaStock Developer's Kit.

The sample applications included on the CD-ROM can also be a good source of information as well as an excellent starting point.

Overview

The MSFL was developed to provide multi-user and networking support as well as limit the amount of information that an application program is required to know about the actual storage of MetaStock data. An application program calls the MSFL functions to read and write both security and price data. All file handles and other low-level considerations are handled by the MSFL. However, the application program is responsible for ensuring that directories opened on removable media are closed before the media is removed from the drive.

MSFL Function Levels

MSFL functions are organized into two levels.

- **Level 1** functions (prefixed by "**MSFL1_**") are the basic I/O functions needed to access MetaStock data files.
- **Level 2** functions (prefixed by "**MSFL2_**") provide support for common tasks that require multiple Level 1 function calls. Using Level 2 functions is recommended over multiple calls to Level 1 functions because of their ease of use and increased efficiency.

Securities

The term "security" is used throughout this manual to refer to stocks, bonds, mutual funds, commodities, currencies, futures, indices, options, etc. The MetaStock format does not differentiate between any of the above types of securities. Each MetaStock directory can contain from 0 to 6,000 securities.

Price Data

Each security has from 0 to 65,500 price records associated with it. In terms of a relational database, there is a one-to-many relationship between the securities and their associated price records. Each price record contains the security price for the period (e.g. tick, day, week, etc.) denoted by the time/date. The fields in the price record largely depend on the security type.

Composites

Composites are simulated securities composed of two securities. The price data is calculated from the two securities that make up the composite. The first security of the composite is known as the “primary security,” and the second is known as the “secondary security.” Price data can only be calculated for the common records between the primary and secondary securities.

Except for a few restrictions, the application program can use composite securities like standard securities. The MSFL manages the matching and calculation of the simulated price data.

Because composites price records are calculated rather than being stored on disk, there are no record numbers associated with composite price records. Any functions that require a record number cannot be used with composite securities. In addition, any functions that return record numbers will always return a zero for a composite’s record number.

If the application attempts to call a function that cannot be used with composite securities, the MSFL function will return an MSFL_ERR_SECURITY_IS_A_COMPOSITE error.

Multi-user Support

To provide multi-user support, the MSFL implements two types of locking, directory and security. Directory locking is internal to the MSFL. Security locking is initiated by the application program via MSFL function calls.

Directory

When the MSFL requires exclusive access to one or more of the files in a directory, the directory is often locked, restricting access to other users. In most cases, the MSFL internal retry period will shield the application from noticing this situation. However, if the directory remains locked, an MSFL_ERR_DIR_IS_BUSY error is returned.

Security

Security locking allows the application to gain access to securities and their price data. Multiple users are permitted to “prevent write” (read) lock a security, but only one user is permitted to write or full lock a security at any given time.

Because of the multi-processing nature of Microsoft Windows, even single-user applications should guard against multiple applications and/or users accessing the same data. Thus, the MSFL requires even single-user applications to lock and unlock securities. For detailed information on locking securities, see the “Security Locking” section in *Using The Library* (page 86).

Reserved File Names

The MetaStock file format reserves several file names for storing price data and security information. In addition, the MSFL also uses several temporary files. To avoid conflict and possible data loss, the application program should not use the file names listed.

File Name	File Type
~MSFL.INX	MSFL security index
~MSFL.LCK	MSFL security lock information
~MSFL.SEC	MSFL security information
~MSFL.USR	MSFL user information
~NONMSFL.USR	Non-MSFL user information
MASTER	Security information
EMASTER	Security information
XMASTER	Security information
F.DAT*	Price data
F.MWD*	Price data
F.DOP*	Price data format
F.TMP*	MSFL temporary price data
SRT.TMP*	MSFL temporary sort file
C.MWS*	MetaStock Smart Chart
**.MWC*	MetaStock chart
**.MWT*	MetaStock template
**.MWL*	MetaStock layout

CD-ROM Support

The MSFL is able to read MetaStock data directly from a CD-ROM drive and other read-only media. The multi-user MSFL files (~MSFL*.*) are created in the directory specified by the *GetTempPath* Windows API call. A maximum of 4,095 sets of MSFL files can exist in the temporary directory at any given time.

Data Types

The MSFL uses several data types to retrieve and return information to the application program. These data types are consistent throughout the library. The specifics of each field are documented in the Structures section (page 81) of this manual.

Formats

This section is an overview of the different field types and not the specific fields.

Dates

Dates are of type *long* and are stored in a year, month, day format with the year being a four digit year (e.g. 15 January 1997 is stored in a long as 19970115). Valid dates range from 1 January 1800 to 31 December 2200.

Times

Times are of type *long* and are stored in hour, minutes, tick order (e.g. 10:03:002 is stored in a long as 1003002). Times are always in twenty-four hour format. Notice that the last segment of the time is not seconds, but ticks. The tick count is used instead of seconds to handle the case of multiple ticks per minute without duplication of records. The first tick of a minute is 000, the second is 001, and so on, up to 999. Valid times range from 00:00:000 to 23:59:999.

Price Data Items

Price data items (e.g. open, close, volume, etc.) are of type *float* and stored in the IEEE floating point format.

Symbols

Symbols are of type *char* and are stored as null-terminated strings. The symbol contains the ticker symbol for the security. Symbols are case sensitive, so the application is responsible for case conversions. Only characters above ASCII 31 are allowed, and all symbols should be stripped of trailing spaces. The symbol must also be left-justified in the string (that is, no leading characters or spaces). The maximum length of a symbol is defined by MSFL_MAX_SYMBOL_LENGTH.

Note: The maximum symbol length does not include the terminating null; it is the maximum length of the symbol itself.

Data Field Combinations

MetaStock price data can consist of eight different fields (date, time, high, low, open, close, volume, and open interest). These fields are grouped logically together in several combinations (e.g. date, time, close, and volume). Like a database of personal information, it wouldn't make much sense if it only contained the street address, state, and zip code. To be meaningful, the database would also need the name, city, and perhaps a phone number. Likewise, there are only certain combinations of the eight fields in MetaStock price data that make sense.

The MSFL will only work with specific combinations of data fields. Even though each bit in the wDataAvailable field can be set separately, there are a limited number of valid combinations. The basic rules are:

- There must be four or more fields used.
- There can be no more than eight fields used.
- The fields, except for the time, must be used in the same order as they appear in the Field Combinations table (below). The *Time* field is used only for intraday securities.

Several combinations are possible based on the rules and will work with the MSFL; however, the entries in the following table are the only data field combinations supported by MetaStock. The table is organized with the data fields and their mnemonics running horizontally and the valid field combinations running vertically. For example, the first column in the Valid Field Combinations represents an intraday four-field combination. The checkmarks indicate that the fields for this combination are date, time, close and volume.

Fields	Mnemonic for Bits	Field Combinations						
Date	MSFL_DATA_DATE	✓	✓	✓	✓	✓	✓	✓
Time	MSFL_DATA_TIME	✓		✓			✓	✓
Close	MSFL_DATA_CLOSE	✓	✓	✓	✓	✓	✓	✓
Volume	MSFL_DATA_VOLUME	✓	✓	✓	✓	✓	✓	✓
High	MSFL_DATA_HIGH		✓	✓	✓	✓	✓	✓
Low	MSFL_DATA_LOW		✓	✓	✓	✓	✓	✓
Open	MSFL_DATA_OPEN				✓	✓	✓	✓
Open Interest	MSFL_DATA_OPENINT					✓		✓

The *wDataAvailable* or field combinations are created by bitwise OR-ing the mnemonics
(e.g. `MSFL_DATA_DATE | MSFL_DATA_CLOSE | MSFL_DATA_VOLUME | MSFL_DATA_HIGH`).

Types

The MSFL uses several defined data types. Except for the MSFL specific data type below, the data types are exactly the same as those in the Microsoft Windows Software Development Kit.

HSECURITY is a 32-bit value used as a handle to a security. The security handle uniquely identifies any security. Security handles are not persistent; that is, they are only valid while the directory is open. Once the directory is closed, the handle is invalid.

Variable Notation

The MSFL uses a form of Hungarian notation to designate the type of each variable or structure member. The type prefixes each variable name. Following is a list of the notations used by the MSFL.

Notation	Type	Description
`b`	BOOL	Boolean, a 32-bit value where zero equals false and any non-zero value equals true.
`c`	char	Character, an 8-bit signed value.
`dw`	DWORD	Double word, a 32-bit unsigned integer value.
`f`	float	A single precision, 32-bit, floating point number.
`h`	N/A	A handle, usually a 32-bit value.
`i`	int	Integer, a 32-bit signed value.
`l`	long	Long integer, a 32-bit signed value.
`p`	N/A	A pointer, a 32-bit address.
`s`	N/A	Structure, a user defined type.
`sz`	char	A null-terminated string of signed characters.
`uc`	BYTE	Byte or unsigned character, an 8-bit unsigned value.
`ui`	UINT	Unsigned integer, a 32-bit unsigned value.
`w`	WORD	Word, a 16-bit unsigned integer value.

Structures

The following structures are used to request, read and write data to and from MetaStock files. Since the MSFL provides only limited data validation, it is the responsibility of the application program to validate the data before writing it to the MetaStock files.

Date Time Structure

The date time structure is used to specify the date and time. It is both passed to and returned from many of the MSFL functions. If there is no time, the *lTime* member of the structure is set to zero. The section titled "Formats" on page 79 has more information on the date and time formats.

Structure

```
typedef struct tagDateTime
{
                      longlDate;
                      longlTime;
} DateTime_struct;
```

Security Information Structure

The security information structure is used to retrieve information about securities and to add new securities.

Structure

```
typedef struct
{
                      DWORDdwTotalSize;
                      HSECURITYhSecurity;
                      charszName[MSFL_MAX_NAME_LENGTH+1];
                      char
 szSymbol[MSFL_MAX_SYMBOL_LENGTH+1];
                      charcPeriodicity;
                      WORDwInterval;
                      BOOLbComposite;
                      BOOLbFlagged;
                      BYTEucDisplayUnits;
                      char
 szCompSymbol[MSFL_MAX_SYMBOL_LENGTH+1];
                      charcCompOperator;
                      floatfCompFactor1;
                      floatfCompFactor2;
                      longlFirstDate;
                      longlLastDate;
                      longlFirstTime;
                      longlLastTime;
                      longlStartTime;
                      longlEndTime;
                      longlCollectionDate;
                      longlMostRecentAdjDate;
                      floatfMostRecentAdjRatio;
                      WORDwDataAvailable;
} MSFLSecurityInfo_struct;
```

Fields

ID	Description
dwTotalSize	The size of the structure, in bytes. This member must be set to the structure size before calling any function that takes the structure as a parameter.
hSecurity	The security handle.
szName	The name of the security. This is not the symbol, but the name by which the security should be referred to by the user. Any trailing spaces should be stripped from the name. The maximum length of the name, not including the terminating null, is defined by MSFL_MAX_NAME_LENGTH.
szSymbol	The security’s ticker symbol. If this security is a composite, this is the symbol of the primary security. The maximum length of the symbol, not including the terminating null, is defined by MSFL_MAX_SYMBOL_LENGTH.
cPeriodicity	The periodicity of the security (i.e. “**D**”aily, “**W**”eekly, “**M**”onthly, or “**I**”ntraday). The valid periodicities are defined, (in a string) by MSFL_VALID_PERIODICITIES.
wInterval	The intraday interval of the security. This field indicates the interval, in minutes, between price data. For tick data and non-intraday securities, this field is set to zero. The minimum interval is defined by MSFL_MIN_INTERVAL and the maximum interval is defined by MSFL_MAX_INTERVAL.
bComposite	Boolean value indicating if the security is a composite.
bFlagged	Boolean value indicating if the security is flagged. The flagged status is used by application programs to perform tasks on several securities (i.e. all securities that are flagged).
ucDisplayUnits	The units in which the price data should be displayed: decimal or fractional. If equal to MSFL_DISPLAY_UNITS_DECIMAL, the price data should be displayed in decimal format. Otherwise, it indicates the denominator (e.g. a display units of 4 would indicate the price data should be displayed in ¼’s). Valid display units range from MSFL_MIN_DISPLAY_UNITS to MSFL_MAX_DISPLAY_UNITS.
szCompSymbol	The symbol of the secondary security in the composite. If this security is not a composite, *szCompSymbol* is a null string. The maximum length of the composite symbol, not including the terminating null, is defined by MSFL_MAX_SYMBOL_LENGTH.
cCompOperator	The composite operator (i.e. the mathematical operation to perform between the two securities in the composite: +, -, *, /). The valid operators are defined, in a string, by MSFL_VALID_OPERATORS. If the security is not a composite, it is zero.
fCompFactor1	The factor that the primary security’s price data is multiplied by, before performing the composite operation. If the security is not a composite, it is zero.
fCompFactor2	The factor that the secondary security’s price data is multiplied by, before performing the composite operation. If the security is not a composite, it is zero.
lFirstDate	The date of the first price record. For more information on date and time formats, see “Formats” (page 79).
lLastDate	The date of the last price record. For more information on date and time formats, see “Formats” (page 79).

ID	Description
lFirstTime	The time of the first price record. For end-of-day securities, this field is zero. For more information on date and time formats, see “Formats” (page 79).
lLastTime	The time of the last price record. For end-of-day securities, this field is zero. For more information on date and time formats, see “Formats” (page 79).
lStartTime	The trading start time. This time specifies when the real-time program should begin collecting price data for this security. For end-of-day securities, the time is zero. For more information on date and time formats, see “Formats” (page 79).
lEndTime	The trading end time. This time specifies when the real-time program should stop collecting price data for this security. For end-of-day securities, the time is zero. For more information on date and time formats, see “Formats” (page 79).
lCollectionDate	The date prior to the first date (i.e. *lFirstDate*) in which to collect price data. Since the first date indicates the date of the first record in the price data, it cannot be adjusted. The collection date indicates to the collection application, such as The DownLoader, to collect the price data between the collection date and the first date; thus, allowing collection of price data prior to the current price data. For more information on date and time formats, see “Formats” (page 79).
lMostRecentAdjDate	Reserved for future use.
fMostRecentAdjRatio	Reserved for future use.
wDataAvailable	The fields that are available in the associated price data file (e.g. date, time, open, close, volume, etc.). Use the data field mnemonics to set and determine the price data fields available. For more details, see “Data Field Combinations” (page 80).

Price Record Structure

The price record structure is used to read and write price data. All possible price fields are in the structure. However, not all fields will always be used. The *wDataAvailable* field indicates the fields used in the price record. The section titled “Data Field Combinations” on page 80 has more detailed information.

Structure

```
typedef struct
{
                      longlDate;
                      longlTime;
                      floatfOpen;
                      floatfHigh;
                      floatfLow;
                      floatfClose;
                      floatfVolume;
                      floatfOpenInt;
                      WORDwDataAvailable;
} MSFLPriceRecord_struct;
```

Fields

ID	Description
lDate	The closing date of the period (i.e. a “period” being a tick, minute, day, week, month, etc.). For more information on date and time formats, see “Formats” (page 79).
lTime	The closing time of the intraday period. For more information on date and time formats, see “Formats” (page 79).

ID	Description
fOpen	The price that the security first traded during the period.
fHigh	The highest price that the security traded during the period.
fLow	The lowest price that the security traded during the period.
fClose	The price of the last trade for the security during the period.
fVolume	The volume for the period. The volume may be entered in any units (e.g. ones, tens, hundreds, etc.), as long as the units are consistent. Hundreds is most commonly used.
fOpenInt	The number of open contracts at the previous day’s close. Open interest may be in any units. Open interest is typically available only for futures and options.
wDataAvailable	The fields that are available in the data file (e.g. date, time, open, close, volume, etc.). Use the data field mnemonics to set and determine the price data fields available (page 80).

Using the Library

This section describes important concepts and details on using the MSFL with your application.

Outline

The basic outline for writing an application that uses the MSFL is as follows.

1. Initialize the MSFL.
   See:
   - *MSFL1_Initialize* (page 115)
   - The section titled “Initialization” on page 87 has more on initializing the MSFL.
2. Open the directory.
   See:
   - *MSFL1_OpenDirectory* (page 119)
3. Obtain the security handle(s).
   See:
   - *MSFL1_GetFirstSecurityInfo* (page 103)
   - *MSFL1_GetLastSecurityInfo* (page 106)
   - *MSFL1_GetSecurityHandle* (page 111)
   - *MSFL2_GetSecurityHandles* (page 125), *etc.*
4. Lock the security.
   See:
   - *MSFL1_LockSecurity* (page 116)
5. Process the price data.
   See:
   - *MSFL1_ReadDataRec* (page 122)
   - *MSFL2_ReadMultipleRecs* (page 128), *etc.*
6. Unlock the security.
   See:
   - *MSFL1_UnlockSecurity* (page 124)
7. Move to the next security.
   See:
   - *MSFL1_GetNextSecurityInfo* (page 108)
   - *MSFL1_GetPrevSecurityInfo* (page 109), *etc.*
8. Close the directory.
   See:
   - *MSFL1_CloseDirectory* (page 92)
9. Shut down the MSFL.
   See:
   - *MSFL1_Shutdown* (page 124)

Initialization

Before calling any of the price data or security functions, the library must be initialized by calling *MSFL1_Initialize* (page 115). After using the library and before terminating the application program, the library must be shut down with a call to *MSFL1_Shutdown* (page 124).

Directory Opening

Before any of the MSFL functions can be used to read securities or price data, the directory containing the MetaStock files must first be opened via *MSFL1_OpenDirectory* (page 119).

The MSFL manages directories similar to the way operating systems manage files. When opening a directory, a directory number is returned. This directory number can be thought of as a handle. Once the directory is open, it remains open until it is closed or until the MSFL is shutdown. As with files, multiple directories can be open concurrently. The MSFL limits the number of open directories to MSFL_MAX_OPEN_DIRECTORIES.

Security Locking

Before accessing a security or its price data, the security must first be locked via *MSFL1_LockSecurity* (page 116). Locking prevents other users or applications from modifying or deleting the security while it is in use. Since locking may prevent other users and applications from accessing data, the application should unlock the security as soon as it is finished using the security.

When locking composite securities, the primary and secondary securities are also locked internally by the MSFL. Thus, if the composite or any of its parts cannot be locked, the lock fails.

An application cannot concurrently lock the same security multiple times.
If the application attempts to lock a security that is already locked by the application, the lock will fail.

If the security was locked by another user, the *MSFL1_GetLastFailedLockInfo* (page 105) function can be used to retrieve the user name or number of users with the security locked.

Lock Types

There are three different lock types that can be applied to a security by the application program. To allow other users and applications to share security and price data, the application should always use the lowest lock type available for the operation. Following is a list of the different lock types in order of precedence.

Full	A “full” lock, defined by MSFL_LOCK_FULL_LOCK, provides the application with total control of the security. The application can edit the security information as well as read and write price data. Other users cannot lock the security, nor can this security be used by a composite security.
Write	A “write” lock, defined by MSFL_LOCK_WRITE_LOCK, allows the application to read and write security price data, but does not allow the application to edit the security information (e.g. name, symbol, composite factor, etc.). When a security is “write” locked, the security cannot be locked by other users. Since composite price data cannot be written, composite securities cannot be “write” locked. Attempting to “write” lock a composite will result in an error.
Prevent Write	A “prevent write” lock, defined by MSFL_LOCK_PREV_WRITE_LOCK, allows the application to read but not write price data. When a security is “prevent write” locked, other users cannot “write” or “full” lock the security. However, the security can be “prevent write” locked by multiple users.

Data Assumptions and Requirements

The MSFL assumes that the price data is in ascending date/time order. If there are duplicate price records grouped together, the MSFL will find the first record for a given date/time.

The MSFL requires that all the securities in a directory are unique (i.e. the ticker symbol, periodicity, and interval of one security does not match that of another). Composites may have a duplicate ticker symbol, periodicity, and interval, but only one composite of each operator (i.e. add, subtract, multiply, and divide) is allowed.

IMPORTANT: A directory with duplicate securities cannot be opened unless the securities are merged; see *MSFL1_OpenDirectory* (page 119) for more information.

Error Handling

The MSFL allows the application program to detect and recover from a variety of errors resulting from any of its functions. All functions in the MSFL return error conditions in the form of negative return codes. Other informational messages are returned as positive return codes. Always use the mnemonic error codes defined in appropriate header file (i.e. *msfl.h*, *msfl.bas*, *msfl.inc* or *msfl.pas*). The values of the error codes may change in future versions, so by using the mnemonic, the application program will remain compatible.

CAUTION: If the MSFL encounters a serious problem with the operating system or its internal tables, an MSFL_ERR_MSFL_CORRUPT error code is returned. If the application encounters this error, the MSFL should be shut down and the application program should exit. Any additional calls to the MSFL will not be performed and the MSFL_ERR_MSFL_CORRUPT error code will be returned. In some instances, the operating system itself may be corrupt, so it is recommended that the user reboot the computer after exiting the application.

For a complete list of error codes, see the “Error Codes”section of this manual beginning on page 131.

Functions

This section contains an alphabetical list of the available MSFL functions.

The function prototypes, structures, error codes, message codes, and miscellaneous defines are contained in the header file (*msfl.h*, *msfl.bas*, *msfl.inc* or *msfl.pas*). This header file should be included in each module that uses any of the MSFL functions.

Return Values

Unless otherwise stated in the function, all MSFL functions return an MSFL error code. On successful completion, MSFL_NO_ERR is returned; in the event of an error, the specific MSFL error code is returned. For a complete list of error codes, see the “Error Codes”section of this manual beginning on page 131.

Some functions return an MSFL message to indicate the action taken by the function. These messages are documented in the functions themselves. For a complete list of message codes, see the “Message Codes” section of this manual beginning on page 136.

Listed By Name

Here is a name-ordered list of the MSFL functions (with a link to their description page).

Function Name	Page
MSFL1_CloseDirectory	92
MSFL1_FindDataDate	92
MSFL1_FindDataRec	93
MSFL1_FormatDate	94
MSFL1_FormatTime	95
MSFL1_GetCurrentDataPos	96
MSFL1_GetDataPath	97
MSFL1_GetDataRecordCount	97
MSFL1_GetDayMonthYear	98
MSFL1_GetDirectoryNumber	99
MSFL1_GetDirectoryStatus	100
MSFL1_GetDirNumberFromHandle	100
MSFL1_GetErrorMessage	102
MSFL1_GetFirstSecurityInfo	103
MSFL1_GetHourMinTicks	104
MSFL1_GetLastFailedLockInfo	105
MSFL1_GetLastFailedOpenDirInfo	106
MSFL1_GetLastSecurityInfo	106
MSFL1_GetMSFLState	107
MSFL1_GetNextSecurityInfo	108
MSFL1_GetPrevSecurityInfo	109
MSFL1_GetRecordCountForDateRange	110
MSFL1_GetSecurityCount	111
MSFL1_GetSecurityHandle	111
MSFL1_GetSecurityID	112
MSFL1_GetSecurityInfo	113
MSFL1_GetSecurityLockedStatus	114
MSFL1_Initialize	115
MSFL1_LockSecurity	116
MSFL1_MakeMSFLDate	117
MSFL1_MakeMSFLTime	118
MSFL1_OpenDirectory	119
MSFL1_ParseDateString	120
MSFL1_ParseTimeString	121
MSFL1_ReadDataRec	122
MSFL1_SeekBeginData	123
MSFL1_SeekEndData	123
MSFL1_Shutdown	124
MSFL1_UnlockSecurity	124
MSFL2_GetSecurityHandles	125
MSFL2_ReadBackMultipleRecs	126
MSFL2_ReadDataRec	127
MSFL2_ReadMultipleRecs	128
MSFL2_ReadMultipleRecsByDates	129

Listed By Type

Here are type-ordered lists of the MSFL functions (each linked to their description page).

Data.

Function Name	Page
MSFL1_GetDataRecordCount	97
MSFL1_GetRecordCountForDateRange	110
MSFL1_ReadDataRec	122
MSFL2_ReadBackMultipleRecs	126
MSFL2_ReadDataRec	127
MSFL2_ReadMultipleRecs	128
MSFL2_ReadMultipleRecsByDates	129

Date / Time.

Function Name	Page
MSFL1_FormatDate	94
MSFL1_FormatTime	95
MSFL1_GetDayMonthYear	98
MSFL1_GetHourMinTicks	104
MSFL1_MakeMSFLDate	117
MSFL1_MakeMSFLTime	118
MSFL1_ParseDateString	120
MSFL1_ParseTimeString	121

Directory.

Function Name	Page
MSFL1_CloseDirectory	92
MSFL1_GetDataPath	97
MSFL1_GetDirectoryNumber	99
MSFL1_GetDirNumberFromHandle	100
MSFL1_GetDirectoryStatus	100
MSFL1_OpenDirectory	119

Error Reporting.

Function Name	Page
MSFL1_GetErrorMessage	102
MSFL1_GetLastFailedLockInfo	105
MSFL1_GetLastFailedOpenDirInfo	106

Locking.

Function Name	Page
MSFL1_GetSecurityLockedStatus	114
MSFL1_LockSecurity	116
MSFL1_UnlockSecurity	124

Search / Positioning.

Function Name	Page
MSFL1_FindDataDate	92
MSFL1_FindDataRec	93
MSFL1_GetCurrentDataPos	96
MSFL1_SeekBeginData	123
MSFL1_SeekEndData	123

Security.

Function Name	Page
MSFL1_GetFirstSecurityInfo	103
MSFL1_GetLastSecurityInfo	106
MSFL1_GetNextSecurityInfo	108
MSFL1_GetPrevSecurityInfo	109
MSFL1_GetSecurityCount	111
MSFL1_GetSecurityHandle	111
MSFL1_GetSecurityID	112
MSFL1_GetSecurityInfo	113
MSFL2_GetSecurityHandles	125

System.

Function Name	Page
MSFL1_GetMSFLState	107
MSFL1_Initialize	115
MSFL1_Shutdown	124

Reference

The following pages describe, in alphabetical order, the functions in the MetaStock File Library. The discussion of each function includes a section that illustrates the function syntax — in C, Visual Basic, Delphi and PowerBASIC – followed by these sections.

ID	Description
Locking	Indicates the minimal lock the application must have on the security before calling the function.
Return Value	Provides the most common return values. In most cases other MSFL error/message codes may be returned. The section titled “Messages and Errors” on page 131 has a complete listing of these codes.
Parameters	Describes each argument passed to the function.
Remarks	Provides a brief description of the function and any additional notes on its use.
See Also	Provides the names of related functions.

MSFL1_CloseDirectory

C

```
int MSFL1_CloseDirectory(char cDirNumber)
```

Visual Basic

```
MSFL1_CloseDirectory    (ByVal cDirNumber As Byte) As Long
```

Delphi

```
MSFL1_CloseDirectory    (cDirNumber : char) : integer;
```

PowerBASIC

```
MSFL1_CloseDirectory    (BYVAL cDirNumber AS BYTE) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_NOT_OPEN** if the directory is not open

Parameters

ID	Description
cDirNumber	Identifies the directory in which to close.

Remarks

- Closes an open directory. All open files in the directory are closed.
- When the last user closes a directory, the MASTER, EMASTER, and XMASTER files are updated with any changes made while the directory was open and the temporary MSFL files are removed from the directory.

See Also

- *MSFL1_OpenDirectory* (page 119)
- *MSFL1_Shutdown* (page 124)

MSFL1_FindDataDate

C

```
int MSFL1_FindDataDate( HSECURITY hSecurity,
  DateTime_struct *psRecordDate,
  WORD *pwRecordNum,
  int iFindMode)
```

Visual Basic

```
MSFL1_FindDataDate(     ByVal hSecurity As Long,
  psRecordDate As DateTime_struct,
  pwRecordNum As Integer,
  ByVal iFindMode As Long) As Long
```

Delphi

```
MSFL1_FindDataDate(     hSecurity : HSECURITY;
  Var psRecordDate : DateTime_struct;
  Var pwRecordNum : word;
  iFindMode : integer) : integer;
```

PowerBASIC

```
MSFL1_FindDataDate(     BYVAL hSecurity AS DWORD,
  psRecordDate AS DateTime_struct,
  pwRecordNum As Word,
  BYVAL iFindMode As Long) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_MSG_NOT_AN_EXACT_MATCH** if successful, but an exact match was not found
- **MSFL_ERR_DATE_BEFORE_FIRST_REC** if the specified date/time is before the first price record
- **MSFL_ERR_DATE_AFTER_LAST_REC** if the specified date/time is after the last price record
- **MSFL_ERR_DATA_RECORD_NOT_FOUND** if a matching price record could not be found for the specified date/time
- **MSFL_ERR_SECURITY_HAS_NO_DATA** if the security has no price records

Parameters

ID	Description
hSecurity	Identifies the security.
psRecordDate	Points to a *DateTime_struct* structure containing the date/time to find. If the security is not an intraday security, the time is ignored. If an exact match is not found, *psRecordDate* receives the date/time of the record found.
pwRecordNum	Points to a WORD that receives the record number for the price record found. If the record is not found or if the security is a composite the record number is returned as zero.
iFindMode	Indicates what type of search to perform to locate the price record. Following are the different modes available. • MSFL_FIND_CLOSEST_PREV If an exact match is not found, find the previous closest record. • MSFL_FIND_CLOSEST_NEXT If an exact match is not found, find the next closest record. • MSFL_FIND_EXACT_MATCH Find an exact date/time match.

Remarks

- Finds the price record for the specified date/time and sets the current data position to that record.

See Also

- *MSFL1_FindDataRec* (page 93)
- *MSFL1_GetCurrentDataPos* (page 96)
- *MSFL1_SeekBeginData* (page 123)
- *MSFL1_SeekEndData* (page 123)

MSFL1_FindDataRec

C

```
int MSFL1_FindDataRec(HSECURITY hSecurity,
  WORD wRecordNum,
  DateTime_struct *psRecordDate)
```

Visual Basic

```
MSFL1_FindDataRec(      ByVal hSecurity As Long,
  ByVal wRecordNum As Integer,
  psRecordDate As DateTime_struct) As Long
```

Delphi

```
MSFL1_FindDataRec(      hSecurity : HSECURITY;
  wRecordNum : word;
  Var psRecordDate : DateTime_struct) : integer;
```

PowerBASIC

```
MSFL1_FindDataRec(      BYVAL hSecurity AS  DWORD,
                        BYVALwRecordNumAs Word,
  psRecordDate AS DateTime_struct) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_RECORD_OUT_OF_RANGE** if a price record does not exist for the specified record number
- **MSFL_ERR_SECURITY_HAS_NO_DATA** if the security has no price records

Parameters

ID	Description
hSecurity	Identifies the security.
wRecordNum	Specifies the record number to find. Record numbers are one based.
psRecordDate	Points to a *DateTime_struct* structure that receives the date/time of price record found.

Remarks

- Finds the price record for the specified record number and sets the current data position to that record.

Note: This function cannot be used with composite securities.

See Also

- *MSFL1_FindDataDate* (page 92)
- *MSFL1_GetCurrentDataPos* (page 96)
- *MSFL1_SeekBeginData* (page 123)
- *MSFL1_SeekEndData* (page 123)

MSFL1_FormatDate

C

```
int MSFL1_FormatDate(   LPSTR pszDateString,
  WORD wStringSize,
  long lDate);
```

Visual Basic

```
MSFL1_FormatDate(       ByVal pszDateString As String,
  ByVal wStringSize As Integer,
  ByVal lDate As Long) As Long
```

Delphi

```
MSFL1_FormatDate(       pszDateString : LPSTR;
  wStringSize : word;
  lDate : integer) : integer;
```

PowerBASIC

```
MSFL1_FormatDate(       pszDateString AS ASCIIZ,
  BYVAL wStringSize As Word,
  BYVAL lDate As Long) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_DATE** if the date to be formatted is invalid
- **ERROR_INSUFFICIENT_BUFFER** if the date string is not large enough

Parameters

ID	Description
pszDateString	Points to a null-terminated string that receives the formatted date string.
wStringSize	Indicates the maximum string length that *pszDateString* can receive, including the terminating null.
lDate	The MSFL date to be formatted.

Remarks

- Formats an MSFL date as a date string. The string is formatted based on the Windows short date format, using the default system locale.

See Also

- *MSFL1_FormatTime* (page 95)
- *MSFL1_GetDayMonthYear* (page 98)
- *MSFL1_ParseDateString* (page 120)
- *MSFL1_ParseTimeString* (page 121)

MSFL1_FormatTime

C

```
int MSFL1_FormatTime(    LPSTR pszTimeString,
  WORD wStringSize,
  long lTime,
  BOOL bIncludeTicks);
```

Visual Basic

```
MSFL1_FormatTime(        ByVal pszTimeString As String,
  ByVal wStringSize As Integer,
  ByVal lTime As Long,
  ByVal bIncludeTicks As Long) As Long
```

Delphi

```
MSFL1_FormatTime(        pszTimeString : LPSTR;
  wStringSize : word;
  lTime : integer;
  bIncludeTicks : bool) : integer;
```

PowerBASIC

```
MSFL1_FormatTime(        pszTimeString AS ASCIIZ,
  BYVAL wStringSize As Word,
  BYVAL lTime As Long,
  BYVAL bIncludeTicks AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_TIME** if the time to be formatted is invalid
- **ERROR_INSUFFICIENT_BUFFER** if the time string is not large enough

Parameters

ID	Description
pszTimeString	Points to a null-terminated string that receives the formatted time string.
wStringSize	Indicates the maximum string length that *pszTimeString* can receive, including the terminating null.
lTime	The MSFL time to be formatted.
bIncludeTicks	Indicates if the time string will include ticks (e.g., 10:51:002 AM), or hours and minutes only (e.g., 10:51 AM).

Remarks

- Formats an MSFL time as a time string. The string is formatted based on the Windows time format, using the default system locale.

See Also

- *MSFL1_FormatDate* (page 94)
- *MSFL1_GetHourMinTicks* (page 104)
- *MSFL1_ParseDateString* (page 120)
- *MSFL1_ParseTimeString* (page 121)

MSFL1_GetCurrentDataPos

C

```
int MSFL1_GetCurrentDataPos(HSECURITY hSecurity,
  WORD *pwRecordNum,
  DateTime_struct *psRecordDate)
```

Visual Basic

```
MSFL1_GetCurrentDataPos(ByVal hSecurity As Long,
  pwRecordNum As Integer,
  psRecordDate As DateTime_struct) As Long
```

Delphi

```
MSFL1_GetCurrentDataPos(hSecurity : HSECURITY;
  Var pwRecordNum : word;
  Var psRecordDate : DateTime_struct) : integer;
```

PowerBASIC

```
MSFL1_GetCurrentDataPos(BYVAL hSecurity AS DWORD,
  pwRecordNum As Word,
  psRecordDate AS DateTime_struct) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_LOCKED** if the security is not locked

Parameters

ID	Description
hSecurity	Identifies the security.
pwRecordNum	Points to a WORD that receives the record number of the current data position. If the record is a composite the record number is returned as zero.
psRecordDate	Points to a *DateTime_struct* structure (page 81) that receives the date/time of price record at the current data position.

Remarks

- Gets the record number and the date/time of the price record at the current data position. If the security is a composite the record number is returned as zero.

See Also

- *MSFL1_FindDataDate* (page 92)
- *MSFL1_FindDataRec* (page 93)
- *MSFL1_SeekBeginData* (page 123)
- *MSFL1_SeekEndData* (page 123)

MSFL1_GetDataPath

C

```
int MSFL1_GetDataPath(  char cDirNumber,
  LPSTR pszPath,
  BOOL bRemoveTrailingSlash)
```

Visual Basic

```
MSFL1_GetDataPath(
  ByVal cDirNumber As Byte,
  ByVal pszPath As String,
  ByVal bRemoveTrailingSlash As Long) As Long
```

Delphi

```
MSFL1_GetDataPath(      cDirNumber : char;
  pszPath : LPSTR;
  bRemoveTrailingSlash : BOOL) : integer;
```

PowerBASIC

```
MSFL1_GetDataPath(      BYVAL cDirNumber AS BYTE,
  pszPath AS ASCIIZ,
  BYVAL bRemoveTrailingSlash AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_NOT_OPEN** if the specified directory is not open

Parameters

ID	Description
cDirNumber	Identifies the directory.
pszPath	Points to a null-terminated string that receives the path. The path can be up to MSFL_MAX_PATH bytes in length.
bRemoveTrailingSlash	Specifies whether or not the trailing backslash should be removed from the path. (e.g. *`C:\MetaStock Data\Stocks\`* would be returned as *`C:\MetaStock Data\Stocks`*). This flag has no effect on root paths (i.e. *`C:\`* will always be returned as C:\).

Remarks

- Gets the path for the specified directory.

See Also

- *MSFL1_GetDirectoryNumber* (page 99)
- *MSFL1_GetDirectoryStatus* (page 100)
- *MSFL1_OpenDirectory* (page 119)

MSFL1_GetDataRecordCount

C

```
int MSFL1_GetDataRecordCount(HSECURITY hSecurity,
  WORD *pwNumOfDataRecs)
```

Visual Basic

```
MSFL1_GetDataRecordCount(ByVal hSecurity As Long,
  pwNumOfDataRecs As Integer) As Long
```

Delphi

```
MSFL1_GetDataRecordCount(hSecurity : HSECURITY;
  Var pwNumOfDataRecs : word ) : integer;
```

PowerBASIC

```
MSFL1_GetDataRecordCount(BYVAL hSecurity AS DWORD,
  pwNumOfDataRecs As Word) As Long
```

Locking

- Prevent Write Lock

Return Values

- MSFL_NO_ERR if successful
- MSFL_ERR_SECURITY_NOT_LOCKED if the security is not locked

Parameters

ID	Description
hSecurity	Identifies the security.
pwNumOfDataRecs	Points to a WORD that receives the number of price records.

Remarks

- Gets the number of price records for the specified security.
- For composite securities, the number of price records returned is an estimate. The actual number of records will be equal to or less than what is reported by this function.

See Also

- *MSFL1_GetRecordCountForDateRange* (page 110)

MSFL1_GetDayMonthYear

C

```
int MSFL1_GetDayMonthYear(WORD *pwDay,
  WORD *pwMonth,
  WORD *pwYear,
  long lDate);
```

Visual Basic

```
MSFL1_GetDayMonthYear(pwDay As Integer,
  pwMonth As Integer,
  pwYear As Integer,
  ByVal lDate As Long) As Long
```

Delphi

```
MSFL1_GetDayMonthYear(Var pwDay : word;
  Var pwMonth : word;
  Var pwYear : word
  lDate:longint) : integer;
```

PowerBASIC

```
MSFL1_GetDayMonthYear(pwDay As Word,
  pwMonth As Word,
  pwYear As Word,
  BYVAL lDate As Long) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_DATE** if the date is invalid

Parameters

ID	Description
pwDay	Points to a WORD that receives the day of the month.
pwMonth	Points to a WORD that receives the month; January = 1, February = 2, and so on.

ID	Description
pwYear	Points to a WORD that receives the year. The year is always a four-digit year (e.g., 2001).
lDate	The MSFL date to be extracted.

Remarks

- Extracts an MSFL date into its components: day, month and year.

See Also

- *MSFL1_FormatDate* (page 94)
- *MSFL1_GetHourMinTicks* (page 104)
- *MSFL1_MakeMSFLDate* (page 117)

MSFL1_GetDirectoryNumber

C

```
int MSFL1_GetDirectoryNumber(LPCSTR pszPath,
  char *pcDirNumber)
```

Visual Basic

```
MSFL1_GetDirectoryNumber(ByVal pszPath As String,
  pcDirNumber As Byte) As Long
```

Delphi

```
MSFL1_GetDirectoryNumber(pszPath : LPCSTR;
  Var pcDirNumber : char) : integer;
```

PowerBASIC

```
MSFL1_GetDirectoryNumber(pszPath AS ASCIIZ,
  pcDirNumber AS BYTE) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_NOT_OPEN** if the directory is not open
- **MSFL_ERR_DIR_DOES_NOT_EXIST** if the directory does not exist

Parameters

ID	Description
pszPath	Points to a null-terminated string that contains the path. The path can be up to MSFL_MAX_PATH bytes in length.
pcDirNumber	Points to a character that receives the directory number for the path.

Remarks

- Gets the directory number associated with an open directory.

See Also

- *MSFL1_GetDataPath* (page 97)
- *MSFL1_GetDirectoryStatus* (page 100)
- *MSFL1_GetDirNumberFromHandle* (page 100)
- *MSFL1_OpenDirectory* (page 119)

MSFL1_GetDirNumberFromHandle

C

```
int MSFL1_GetDirNumberFromHandle(HSECURITY hSecurity,
  char *pcDirNumber)
```

Visual Basic

```
MSFL1_GetDirNumberFromHandle(ByVal hSecurity As Long,
  pcDirNumber As Byte) As Long
```

Delphi

```
MSFL1_GetDirNumberFromHandle(hSecurity : HSECURITY;
  Var pcDirNumber : char ) : integer;
```

PowerBASIC

```
MSFL1_GetDirNumberFromHandle(BYVAL hSecurity AS DWORD,
     pcDirNumber AS BYTE) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_SECURITY_HANDLE** if the handle is invalid

Parameters

ID	Description
hSecurity	Identifies the security.
pcDirNumber	Points to a character that receives the directory number for the security.

Remarks

- Gets the directory number for the specified security.

See Also

- *MSFL1_GetSecurityHandle* (page 111)
- *MSFL1_GetDirectoryNumber* (page 99)

MSFL1_GetDirectoryStatus

C

```
int MSFL1_GetDirectoryStatus( char cDirNumber,
  LPCSTR pszDirectory,
  MSFLDirectoryStatus_struct *psDirStatus)
```

Visual Basic

```
MSFL1_GetDirectoryStatus(ByVal cDirNumber As Byte,
  ByVal pszDirectory As String,
  psDirStatus As MSFLDirectoryStatus_struct) As Long
```

Delphi

```
MSFL1_GetDirectoryStatus(cDirNumber : char;
  pszDirectory : LPCSTR;
  Var psDirStatus : MSFLDirectoryStatus_struct) : integer;
```

PowerBASIC

```
MSFL1_GetDirectoryStatus(BYVAL cDirNumber AS BYTE,
  pszDirectory AS ASCIIZ,
  psDirStatus AS MSFLDirectoryStatus_struct) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_NOT_OPEN** if the directory is not open
- **MSFL_ERR_DIR_DOES_NOT_EXIST** if the directory does not exist

Parameters

ID	Description
cDirNumber	Identifies the directory. If zero, the path pointed to by *pszDirectory* is used instead. The *cDirNumber* is typically used to obtain the status of an open directory, while *pszDirectory* is typically used to obtain the status of a closed directory.
pszDirectory	Points to a null-terminated string that contains the directory. This pointer can be left null if the directory is specified by *cDirNumber.*
psDirStatus	Points to an *MSFLDirectoryStatus_struct* structure (page 101) that receives the directory status. The *dwTotalSize* member must be set to the structure size before calling *MSFL1_GetDirectoryStatus* (page 100).

Remarks

- Gets the directory status information.
- The directory can be specified either by cDirNumber or pszDirectory. For example, if the directory is open, the directory status can be retrieved by making the following call: `MSFL1_GetDirectoryStatus(cDirNumber, NULL, &sDirStatus)`. If the directory is not open, the directory status can be retrieved by making the following call: `MSFL1_GetDirectoryStatus(0, szDirectory, &sDirStatus)`.
- The *MSFLDirectoryStatus_struct* structure is defined as follows:

```
typedef struct
{
  DWORD      dwTotalSize;
  BOOL       bExists;
  BOOL       bInUse;
  BOOL       bMetaStockDir;
  WORD       wDriveType;
  BOOL       bOpen;
  BOOL       bReadOnly;
  BOOL       bUserInvalid;
  char       cDirNumber;
  DWORD      dwNumOfSecurities;
} MSFLDirectoryStatus_struct;
```

Fields

ID	Description
dwTotalSize	The size of the structure, in bytes.
bExists	Boolean value indicating if the directory exists.
bInUse	Boolean value indicating if the directory is in use by one or more MSFL users.
bMetaStockDir	Boolean value indicating if the directory contains MetaStock files.

ID	Description
wDriveType	The drive type, which can be any one of the following: • MSFL_DRIVE_TYPE_UNKNOWN The drive type is unknown. • MSFL_DRIVE_TYPE_REMOVABLE The drive is removable media. • MSFL_DRIVE_TYPE_FIXED The drive is fixed (i.e. a hard drive). • MSFL_DRIVE_TYPE_REMOTE The drive is remote (i.e. a network drive). • MSFL_DRIVE_TYPE_CD_ROM The drive is a local CD-ROM drive – network CD-ROM drives are reported as remote drives. • MSFL_DRIVE_TYPE_RAM_DISK The drive is a RAM disk.
bOpen	Boolean value indicating if the directory is open.
bReadOnly	Boolean value indicating if the directory is on read-only media. This field is only defined if the directory is open.
bUserInvalid	Boolean value indicating if the user is invalid. A user is invalid when another user with the same user ID is forced into the directory already in use by the current user. This field is only defined if the directory is open.
cDirNumber	The directory number, if the directory is open.
dwNumOfSecurities	The number of securities in the directory, if the directory is open. Remember if the directory is not open, *bReadOnly*, *bUserInvalid*, *cDirNumber*, and *dwNumOfSecurities* are undefined. In other words, the directory must be open to determine if the directory is read-only.

See Also

- *MSFL1_GetDataPath* (page 97),
- *MSFL1_GetDirectoryNumber* (page 99),
- *MSFL1_GetSecurityCount* (page 111)
- *MSFL1_OpenDirectory* (page 119)

MSFL1_GetErrorMessage

C

```
LPSTR MSFL1_GetErrorMessage( int iErr,
  LPSTR pszErrorMessage,
  WORD wMaxMsgLength)
```

Visual Basic

```
MSFL1_GetErrorMessage(  ByVal iErr As Long,
  ByVal pszErrorMessage As String,
  ByVal wMaxMsgLength As Integer) As String
```

Delphi

```
MSFL1_GetErrorMessage(   iErr : integer;
  pszErrorMessage : LPSTR;
  wMaxMsgLength : WORD ) : LPSTR;
```

PowerBASIC

```
MSFL1_GetErrorMessage(  BYVAL iErr As Long,
  pszErrorMessage AS ASCIIZ,
  BYVAL wMaxMsgLength As Word) AS STRING
```

Locking

- None

Return Values

- A pointer to the error message string.
  The pointer returned is the same as the pointer passed as the input argument pszErrorMessage

Parameters

ID	Description
iErr	Indicates the MSFL error.
pszErrorMessage	Points to a null-terminated string that receives the error message.
wMaxMsgLength	Indicates the maximum message length that *pszErrorMessage* can receive, not including the terminating null. The maximum length error message that the MSFL will return is defined by MSFL_MAX_ERR_MSG_LENGTH.

Remarks

- Returns a string error message for the specified MSFL error code.

See Also

- *MSFL1_GetLastFailedLockInfo* (page 105)
- *MSFL1_GetLastFailedOpenDirInfo* (page 106)

MSFL1_GetFirstSecurityInfo

C

```
int MSFL1_GetFirstSecurityInfo(char cDirNumber,
  MSFLSecurityInfo_struct *psSecurityInfo)
```

Visual Basic

```
MSFL1_GetFirstSecurityInfo(ByVal cDirNumber As Byte,
  psSecurityInfo As MSFLSecurityInfo_struct) As Long
```

Delphi

```
MSFL1_GetFirstSecurityInfo(cDirNumber : char;
  Var psSecurityInfo : MSFLSecurityInfo_struct) : integer;
```

PowerBASIC

```
MSFL1_GetFirstSecurityInfo(BYVAL cDirNumber AS BYTE,
  psSecurityInfo AS MSFLSecurityInfo_struct) As Long
```

Locking

- None

Return Values

- **MSFL_MSG_LAST_SECURITY_IN_DIR** if successful and this security is the last security in the directory
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if the directory is empty

Parameters

ID	Description
cDirNumber	Identifies the directory.
psSecurityInfo	Points to an *MSFLSecurityInfo_struct* structure (page 82) that receives the security information. The *dwTotalSize* member must be set to the structure size before calling MSFL1_GetFirstSecurityInfo.

Remarks

- Gets the security information for the first security in the directory.

Note: If the security is not locked when calling this function, the security information returned may not reflect changes made by another user. In addition, the starting and ending dates and times for composite securities may not reflect changes made to the primary or secondary securities.

See Also

- *MSFL1_GetLastSecurityInfo* (page 106)
- *MSFL1_GetNextSecurityInfo* (page 108)
- *MSFL1_GetPrevSecurityInfo* (page 109)
- *MSFL1_GetSecurityInfo* (page 113)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetHourMinTicks

C

```
int MSFL1_GetHourMinTicks(WORD *pwHour,
  WORD *pwMin,
  WORD *pwTicks,
  long lTime);
```

Visual Basic

```
MSFL1_GetHourMinTicks(  pwHour As Integer,
  pwMin As Integer,
  pwTicks As Integer,
  ByVal lTime As Long) As Long
```

Delphi

```
MSFL1_GetHourMinTicks(  Var pwHour : word;
  Var pwMin : word;
  Var pwTicks : word;
  lTime : longint) : integer;
```

PowerBASIC

```
MSFL1_GetHourMinTicks(  pwHour As Word,
  pwMin As Word,
  pwTicks As Word,
  BYVAL lTime As Long) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_TIME** if the time is invalid

Parameters

ID	Description
pwHour	Points to a WORD that receives the hour. The hour is always in 24-hour format.
pwMin	Points to a WORD that receives the minutes.
pwTicks	Points to a WORD that receives the ticks.
lTime	The MSFL time to be extracted.

Remarks

- Extracts an MSFL time into its components: hour, minutes and ticks.

See Also

- *MSFL1_FormatTime* (page 95)
- *MSFL1_GetDayMonthYear* (page 98)
- *MSFL1_MakeMSFLTime* (page 118)

MSFL1_GetLastFailedLockInfo

C

```
int MSFL1_GetLastFailedLockInfo(LPSTR pszAppName,
  LPSTR pszUserName,
  UINT *puiUsersWithLock,
  UINT *puiLockType)
```

Visual Basic

```
MSFL1_GetLastFailedLockInfo(ByVal pszAppName As String,
  ByVal pszUserName As String,
  puiUsersWithLock As Long,
  puiLockType As Long) As Long
```

Delphi

```
MSFL1_GetLastFailedLockInfo(pszAppName : LPSTR;
  pszUserName : LPSTR;
  Var puiUsersWithLock : UINT;
  Var puiLockType : UINT) : integer;
```

PowerBASIC

```
MSFL1_GetLastFailedLockInfo(pszAppName AS ASCIIZ,
                            pszUserName AS ASCIIZ,
                            puiUsersWithLock AS DWORD,
                            puiLockType AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful

Parameters

ID	Description
pszAppName	Points to a null-terminated string that receives the application name. The application name can be up to MSFL_MAX_APP_NAME_LENGTH bytes, not including the terminating null.
pszUserName	Points to a null-terminated string that receives the user name. The user name can be up to MSFL_MAX_USER_NAME_LENGTH bytes, not including the terminating null. If the security was not locked, the user name is returned as “no users.” If the user is unknown, the user name is returned as “unknown user.”
puiUsersWithLock	Points to an unsigned integer that receives the number of users that had the security locked.
puiLockType	Points to an unsigned integer that receives the lock type. Following are the possible lock types. • MSFL_LOCK_PREV_WRITE_LOCK The security is prevent write locked. • MSFL_LOCK_WRITE_LOCK The security is write locked. • MSFL_LOCK_FULL_LOCK The security is full locked.

Remarks

- Gets the user information for the user(s) who had the security locked when the last security lock failed. If multiple users had the security locked, the application name and user name are returned blank.

See Also

- *MSFL1_LockSecurity* (page 116)
- *MSFL1_GetErrorMessage* (page 102)

MSFL1_GetLastFailedOpenDirInfo

C

```
int MSFL1_GetLastFailedOpenDirInfo(LPSTR pszAppName,
  LPSTR pszUserName)
```

Visual Basic

```
MSFL1_GetLastFailedOpenDirInfo(ByVal pszAppName As String,
  ByVal pszUserName As String) As Long
```

Delphi

```
MSFL1_GetLastFailedOpenDirInfo(pszAppName : LPSTR;
  pszUserName : LPSTR) : integer;
```

PowerBASIC

```
MSFL1_GetLastFailedOpenDirInfo(pszAppName AS ASCIIZ,
  pszUserName AS ASCIIZ) As Long
```

Locking

- None

Return Values

- MSFL_NO_ERR if successful

Parameters

ID	Description
pszAppName	Points to a null-terminated string that receives the application name. The application name can be up to MSFL_MAX_APP_NAME_LENGTH bytes, not including the terminating null.
pszUserName	Points to a null-terminated string that receives the user name. The user name can be up to MSFL_MAX_USER_NAME_LENGTH bytes, not including the terminating null. If a non-MSFL application is using the directory, the user name is returned blank.

Remarks

- Gets the user information for the last failed directory open. If the directory is in use by a non-MSFL application or if the user is already using the directory, this function can be used to retrieve the user information and display a message for the user.

Note: This function will only return the user information if *MSFL1_OpenDirectory* (page 119) fails with an MSFL_ERR_USER_ID_ALREADY_IN_DIR or an MSFL_ERR_NON_MSFL_USER_IN_DIR error.

See Also

- *MSFL1_OpenDirectory* (page 119)
- *MSFL1_GetErrorMessage* (page 102)

MSFL1_GetLastSecurityInfo

C

```
int MSFL1_GetLastSecurityInfo(char cDirNumber,
  MSFLSecurityInfo_struct *psSecurityInfo)
```

Visual Basic

```
MSFL1_GetLastSecurityInfo(ByVal cDirNumber As Byte,
  psSecurityInfo As MSFLSecurityInfo_struct) As Long
```

Delphi

```
MSFL1_GetLastSecurityInfo(cDirNumber : char;
  Var psSecurityInfo : MSFLSecurityInfo_struct) : integer;
```

PowerBASIC

```
MSFL1_GetLastSecurityInfo(BYVAL cDirNumber AS BYTE,
  psSecurityInfo AS MSFLSecurityInfo_struct) As Long
```

Locking

- None

Return Values

- **MSFL_MSG_FIRST_SECURITY_IN_DIR** if successful and this security is the first security in the directory
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if the directory is empty

Parameters

ID	Description
cDirNumber	Identifies the directory.
psSecurityInfo	Points to an *MSFLSecurityInfo_struct* structure (page 82) that receives the security information. The *dwTotalSize* member must be set to the structure size before calling MSFL1_GetLastSecurityInfo.

Remarks

- Gets the security information for the last security in the directory.

Note: If the security is not locked when calling this function, the security information returned may not reflect changes made by another user. In addition, the starting and ending dates and times for composite securities may not reflect changes made to the primary or secondary securities.

See Also

- *MSFL1_GetFirstSecurityInfo* (page 103)
- *MSFL1_GetNextSecurityInfo* (page 108)
- *MSFL1_GetPrevSecurityInfo* (page 109)
- *MSFL1_GetSecurityInfo* (page 113)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetMSFLState

C

```
int MSFL1_GetMSFLState(void)
```

Visual Basic

```
MSFL1_GetMSFLState() As Long
```

Delphi

```
MSFL1_GetMSFLState : integer;
```

PowerBASIC

```
MSFL1_GetMSFLState() As Long
```

Locking

- None

Return Values

- **MSFL_STATE_INITIALIZED** The MSFL is initialized and available for use
- **MSFL_STATE_UNINITIALIZED** The MSFL has not been initialized and must be initialized before any of the MSFL functions can be used to access MetaStock files
- **MSFL_STATE_CORRUPT** The MSFL is in a corrupt state. The MSFL internal tables have been damaged or the operating system has been corrupted. At this point the MSFL should be shutdown and the application program should be terminated

Parameters

- None

Remarks

- Returns the state of the MetaStock file library.

See Also

- *MSFL1_Initialize* (page 115)
- *MSFL1_Shutdown* (page 124)

MSFL1_GetNextSecurityInfo

C

```
int MSFL1_GetNextSecurityInfo(HSECURITY hSecurity,
  MSFLSecurityInfo_struct *psSecurityInfo)
```

Visual Basic

```
MSFL1_GetNextSecurityInfo(ByVal hSecurity As Long,
  psSecurityInfo As MSFLSecurityInfo_struct) As Long
```

Delphi

```
MSFL1_GetNextSecurityInfo(hSecurity : HSECURITY;
  Var psSecurityInfo : MSFLSecurityInfo_struct) : integer;
```

PowerBASIC

```
MSFL1_GetNextSecurityInfo(BYVAL hSecurity AS DWORD,
  psSecurityInfo AS MSFLSecurityInfo_struct) As Long
```

Locking

- None

Return Values

- **MSFL_MSG_LAST_SECURITY_IN_DIR** if successful and this security is the last security in the directory
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if there are no more securities in the directory

Parameters

ID	Description
hSecurity	Identifies the security. If the handle is zero, the current directory and position from the last call to any one of the *MSFL1_GetxxxxSecurityInfo* functions is used. This allows the application to step through the list of securities by repeated calls to this function.
psSecurityInfo	Points to an *MSFLSecurityInfo_struct* structure (page 82) that receives the security information. The *dwTotalSize* member must be set to the structure size before calling MSFL1_GetNextSecurityInfo.

Remarks

- Gets the security information for the next security in the directory.

Note: If the security is not locked when calling this function, the security information returned may not reflect changes made by another user. In addition, the starting and ending dates and times for composite securities may not reflect changes made to the primary or secondary securities.

See Also

- *MSFL1_GetFirstSecurityInfo* (page 103)
- *MSFL1_GetLastSecurityInfo* (page 106)
- *MSFL1_GetPrevSecurityInfo* (page 109)
- *MSFL1_GetSecurityCount* (page 111)
- *MSFL1_GetSecurityInfo* (page 113)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetPrevSecurityInfo

C

```
int MSFL1_GetPrevSecurityInfo(HSECURITY hSecurity,
  MSFLSecurityInfo_struct *psSecurityInfo)
```

Visual Basic

```
MSFL1_GetPrevSecurityInfo(ByVal hSecurity As Long,
  psSecurityInfo As MSFLSecurityInfo_struct) As Long
```

Delphi

```
MSFL1_GetPrevSecurityInfo(hSecurity : HSECURITY;
  Var psSecurityInfo : MSFLSecurityInfo_struct) : integer;
```

PowerBASIC

```
MSFL1_GetPrevSecurityInfo(BYVAL hSecurity AS DWORD,
  psSecurityInfo AS MSFLSecurityInfo_struct) As Long
```

Locking

- None

Return Values

- **MSFL_MSG_FIRST_SECURITY_IN_DIR** if successful and this security is the first security in the directory
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if there are no more securities in the directory

Parameters

ID	Description
hSecurity	Identifies the security. If the handle is zero, the current directory and position from the last call to any one of the *MSFL1_GetxxxxSecurityInfo* functions is used. This allows the application to step through the list of securities by repeated calls to this function.
psSecurityInfo	Points to an *MSFLSecurityInfo_struct* structure (page 82) that receives the security information. The *dwTotalSize* member must be set to the structure size before calling MSFL1_GetPrevSecurityInfo.

Remarks

- Gets the security information for the previous security in the directory.

Note: If the security is not locked when calling this function, the security information returned may not reflect changes made by another user. In addition, the starting and ending dates and times for composite securities may not reflect changes made to the primary or secondary securities.

See Also

- *MSFL1_GetFirstSecurityInfo* (page 103)
- *MSFL1_GetLastSecurityInfo* (page 106)
- *MSFL1_GetNextSecurityInfo* (page 108)
- *MSFL1_GetSecurityCount* (page 111)
- *MSFL1_GetSecurityInfo* (page 113)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetRecordCountForDateRange

C

```
int MSFL1_GetRecordCountForDateRange(HSECURITY hSecurity,
  const DateTime_struct *psFirstDate,
  const DateTime_struct *psLastDate,
  WORD *pwNumOfDataRecs)
```

Visual Basic

```
MSFL1_GetRecordCountForDateRange(ByVal hSecurity As Long,
  psFirstDate As DateTime_struct,
  psLastDate As DateTime_struct,
  pwNumOfDataRecs As Integer) As Long
```

Delphi

```
MSFL1_GetRecordCountForDateRange(hSecurity : HSECURITY;
  const psFirstDate : DateTime_struct;
  const psLastDate : DateTime_struct;
  Var pwNumOfDataRecs : word) : integer;
```

PowerBASIC

```
MSFL1_GetRecordCountForDateRange(BYVAL hSecurity AS DWORD,
  psFirstDate AS DateTime_struct,
  psLastDate AS DateTime_struct,
  pwNumOfDataRecs As Word) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_HAS_NO_DATA** if the security has no price records

Parameters

ID	Description
hSecurity	Identifies the security.
psFirstDate	Points to a *DateTime_struct* structure (page 81) that specifies the date/time of the first record in the date range.
psLastDate	Points to a *DateTime_struct* structure (page 81) that specifies the date/time of the last record in the date range.
pwNumOfDataRecs	Points to a WORD that receives the number of price records within the specified date range.

Remarks

- Gets the number of price records within a date range.
- For composite securities, the number of price records returned is an estimate. The actual number of records will be equal to or less than what is reported by this function.

See Also

- *MSFL1_GetDataRecordCount* (page 97)

MSFL1_GetSecurityCount

C

```
int MSFL1_GetSecurityCount(char cDirNumber,
  DWORD *pdwNumOfSecurities)
```

Visual Basic

```
MSFL1_GetSecurityCount(ByVal cDirNumber As Byte,
  pdwNumOfSecurities As Long) As Long
```

Delphi

```
MSFL1_GetSecurityCount(cDirNumber : char;
  Var pdwNumOfSecurities : DWORD) : integer;
```

PowerBASIC

```
MSFL1_GetSecurityCount(BYVAL cDirNumber AS BYTE,
  pdwNumOfSecurities AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_NOT_OPEN** if the directory is not open
- **MSFL_ERR_NOT_A_MS_DIR** if the directory does not contain MetaStock files

Parameters

ID	Description
cDirNumber	Identifies the directory.
pdwNumOfSecurities	Points to the DWORD that receives the number of securities in the specified directory.

Remarks

- Gets the number of securities in the directory.

See Also

- *MSFL1_GetDirectoryStatus* (page 100)

MSFL1_GetSecurityHandle

C

```
int MSFL1_GetSecurityHandle(MSFLSecurityIdentifier_struct
  *psSecurityID,
  HSECURITY *phSecurity)
```

Visual Basic

```
MSFL1_GetSecurityHandle(psSecurityID
  As MSFLSecurityIdentifier_struct,
  phSecurity As Long) As Long
```

Delphi

```
MSFL1_GetSecurityHandle(const psSecurityID :
  MSFLSecurityIdentifier_struct;
  Var phSecurity : HSECURITY) : integer;
```

PowerBASIC

```
MSFL1_GetSecurityHandle(psSecurityID AS
  MSFLSecurityIdentifier_struct,
  phSecurity AS DWORD) As Long
```

Locking

- None

Return Values

- MSFL_NO_ERR if successful
- MSFL_ERR_SECURITY_NOT_FOUND if the security is not found

Parameters

ID	Description
psSecurityID	Points to an *MSFLSecurityIdentifier_struct* structure that specifies the security. The *dwTotalSize* member must be set to the structure size before calling *MSFL1_GetSecurityHandle*.
phSecurity	Points to an HSECURITY that receives the security handle for the specified security.

Remarks

- Gets the security handle for the specified security.

See Also

- *MSFL1_GetSecurityID* (page 112)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetSecurityID

C

```
int MSFL1_GetSecurityID(HSECURITY hSecurity,
  MSFLSecurityIdentifier_struct *psSecurityID)
```

Visual Basic

```
MSFL1_GetSecurityID(  ByVal hSecurity As Long,
  psSecurityID As MSFLSecurityIdentifier_struct) As Long
```

Delphi

```
MSFL1_GetSecurityID(  hSecurity : HSECURITY;
  Var psSecurityID : MSFLSecurityIdentifier_struct) :
  integer;
```

PowerBASIC

```
MSFL1_GetSecurityID(  BYVAL hSecurity AS DWORD,
  psSecurityID AS MSFLSecurityIdentifier_struct) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if the security has been deleted or if the handle is invalid

Parameters

ID	Description
hSecurity	Identifies the security.
psSecurityID	Points to an *MSFLSecurityIdentifier_struct* structure (page 112) that receives the security identifier. The *dwTotalSize* member must be set to the structure size before calling MSFL1_GetSecurityID.

Remarks

- Gets the security identifier for the specified security.
- The *MSFLSecurityIdentifier_struct* structure is defined as follows.

```
typedef struct
{
   DWORD dwTotalSize;
         char  cDirNumber;
         char  szSymbol[MSFL_MAX_SYMBOL_LENGTH+1];
         char  cPeriodicity;
         WORD  wInterval;
         BOOL  bComposite;
         char  szCompSymbol[MSFL_MAX_SYMBOL_LENGTH+1];
         char  cCompOperator;
} MSFLSecurityIdentifier_struct;
```

Fields

ID	Description
dwTotalSize	The size of the structure, in bytes.
cDirNumber	The directory number returned by the *MSFL1_OpenDirectory* (page 119) function.
szSymbol	The security's ticker symbol. If this security is a composite, this is the symbol of the primary security. The maximum length of the symbol, not including the terminating null, is defined by MSFL_MAX_SYMBOL_LENGTH.
cPeriodicity	The periodicity of the security (i.e. "D"aily, "W"eekly, "M"onthly, or "I"ntraday). The valid periodicity's are defined, in a string, by MSFL_VALID_PERIODICITIES.
wInterval	The intraday interval of the security. This field indicates the interval, in minutes, between price data. For tick data and non-intraday securities, this field is set to zero. The minimum interval is defined by MSFL_MIN_INTERVAL and the maximum interval is defined by MSFL_MAX_INTERVAL.
bComposite	Boolean value indicating if the security is a composite.
szCompSymbol	The symbol of the secondary security in the composite. If this security is not a composite, *szCompSymbol* is a null string. The maximum length of the symbol, not including the terminating null, is defined by MSFL_MAX_SYMBOL_LENGTH.
cCompOperator	The composite operator (i.e. the mathematical operation to perform between the two securities in the composite: +, -, *, /). The valid operators are defined, in a string, by MSFL_VALID_OPERATORS. If the security is not a composite, it is zero.

See Also

- *MSFL1_GetSecurityHandle* (page 111)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetSecurityInfo

C

```
int MSFL1_GetSecurityInfo(HSECURITY hSecurity,
  MSFLSecurityInfo_struct *psSecurityInfo)
```

Visual Basic

```
MSFL1_GetSecurityInfo(ByVal hSecurity As Long,
  psSecurityInfo As MSFLSecurityInfo_struct) As Long
```

Delphi

```
MSFL1_GetSecurityInfo(hSecurity : HSECURITY;
  Var psSecurityInfo : MSFLSecurityInfo_struct) : integer;
```

PowerBASIC

```
MSFL1_GetSecurityInfo(BYVAL hSecurity AS DWORD,
  psSecurityInfo AS MSFLSecurityInfo_struct) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if the security has been deleted or if the handle is invalid

Parameters

ID	Description
hSecurity	Identifies the security.
psSecurityInfo	Points to the *MSFLSecurityInfo_struct* structure (page 82) that receives the security information. The *dwTotalSize* member must be set to the structure size before calling MSFL1_GetSecurityInfo.

Remarks

- Gets the security information for the specified security.

Note: If the security is not locked when calling this function, the security information returned may not reflect changes made by another user. In addition, the starting and ending dates for composite securities may not reflect changes made to the primary or secondary securities.

See Also

- *MSFL1_GetFirstSecurityInfo* (page 103)
- *MSFL1_GetLastSecurityInfo* (page 106)
- *MSFL1_GetNextSecurityInfo* (page 108)
- *MSFL1_GetPrevSecurityInfo* (page 109)
- *MSFL2_GetSecurityHandles* (page 125)

MSFL1_GetSecurityLockedStatus

C

```
int MSFL1_GetSecurityLockedStatus(HSECURITY hSecurity,
  int *piLockStatus,
  UINT *puiLockType)
```

Visual Basic

```
MSFL1_GetSecurityLockedStatus(ByVal hSecurity As Long,
  piLockStatus As Long,
  puiLockType As Long) As Long
```

Delphi

```
MSFL1_GetSecurityLockedStatus(hSecurity : HSECURITY;
  Var piLockStatus : integer;
  Var puiLockType : UINT) : integer;
```

PowerBASIC

```
MSFL1_GetSecurityLockedStatus(BYVAL hSecurity AS DWORD,
  piLockStatus As Long,
  puiLockType AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_FOUND** if the security could not be found

Parameters

ID	Description
hSecurity	Identifies the security.
piLockStatus	Points to an integer that receives the lock status. Following are the possible lock status codes. • MSFL_LOCK_STATUS_UNLOCKED The security is not locked by this user or any other user. • MSFL_LOCK_STATUS_LOCKED_CURRENT The security is locked by this user. • MSFL_LOCK_STATUS_LOCKED_OTHER The security is locked by another user. • MSFL_LOCK_STATUS_LOCKED_COMP_CUR The security is locked as part of a composite security by this user. • MSFL_LOCK_STATUS_LOCKED_COMP_OTH The security is locked as part of a composite security by another user.
puiLockType	Points to an UINT that receives the lock type. Following are the possible lock types. • MSFL_LOCK_PREV_WRITE_LOCK The security is prevent write locked. • MSFL_LOCK_WRITE_LOCK The security is write locked. • MSFL_LOCK_FULL_LOCK The security is full locked.

Remarks

- Gets the lock status of the specified security.

See Also

- *MSFL1_LockSecurity* (page 116)
- *MSFL1_UnlockSecurity* (page 124)

MSFL1_Initialize

C

```
int MSFL1_Initialize(LPCSTR pszAppName,
  LPCSTR pszUserName,
  int iInterfaceVersion)
```

Visual Basic

```
MSFL1_Initialize(       ByVal pszAppName As String,
  ByVal pszUserName As String,
  ByVal iInterfaceVersion As Long) As Long
```

Delphi

```
MSFL1_Initialize(       pszAppName : LPCSTR;
  pszUserName : LPCSTR;
  iInterfaceVersion: integer ) : integer;
```

PowerBASIC

```
MSFL1_Initialize(       pszAppName AS ASCIIZ,
  pszUserName AS ASCIIZ,
  BYVAL iInterfaceVersion As Long) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_ALREADY_INITIALIZED** if the MSFL is currently initialized
- **MSFL_ERR_INSUFFICIENT_MEM** if there is insufficient memory to initialize the MSFL
- **MSFL_ERR_INVALID_USER_ID** if the application name and/or user name are invalid

Parameters

ID	Description
pszAppName	Points to a null-terminated string that contains the application name. The maximum length of the application name is defined by MSFL_MAX_APP_NAME_LENGTH.
pszUserName	Points to a null-terminated string that contains the user name. The maximum length of the user name is defined by MSFL_MAX_USER_NAME_LENGTH.
iInterfaceVersion	Indicates the MSFL DLL interface version. The current DLL interface version is defined by MSFL_DLL_INTERFACE_VERSION and can simply be passed into the MSFL1_Initialize function.

Remarks

- Initializes the MetaStock File Library by creating the internal tables and buffers. Once the MSFL is successfully initialized, it cannot be initialized again without first shutting down. Also, if the application successfully initializes the MSFL, it must shut down the MSFL (via *MSFL1_Shutdown* (page 124) before exiting. Failure to do so may cause corruption of files and memory leaks. It may also keep directories open and securities locked.
- The application and user names constitute the MSFL user ID. The MSFL user ID is used to distinguish between users in a data directory. By including the application name, the same user can access the same directory with two different applications (e.g. MetaStock and The DownLoader).

Note: Before calling this function, you must setup the key structure as documented in the Initialization section (page 87).

See Also

- *MSFL1_GetMSFLState* (page 107)
- *MSFL1_Shutdown* (page 124)

MSFL1_LockSecurity

C

```
int MSFL1_LockSecurity( HSECURITY hSecurity,
  UINT uiLockType)
```

Visual Basic

```
MSFL1_LockSecurity(     ByVal hSecurity As Long,
  ByVal uiLockType As Long) As Long
```

Delphi

```
MSFL1_LockSecurity(     hSecurity : HSECURITY;
  uiLockType : UINT) : integer;
```

PowerBASIC

```
MSFL1_LockSecurity(     BYVAL hSecurity AS DWORD,
  BYVAL uiLockType AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_TOO_MANY_SEC_LOCKED** if the application attempted to lock more than the maximum number of securities (i.e. MSFL_MAX_LOCKED_SECURITIES)
- **MSFL_ERR_SECURITY_LOCKED** if the security is locked by this or another application

Parameters

ID	Description
hSecurity	Identifies the security to lock.
uiLockType	Specifies the lock type. Applications reading MetaStock price data can simply pass MSFL_LOCK_PREV_WRITE_LOCK.

Remarks

- Locks the specified security. For more information on security locking and the lock types, see Security Locking (page 87).
- An application cannot concurrently lock the same security multiple times. Nor can an application lock more than the maximum number of locked securities per application (i.e. MSFL_MAX_LOCKED_SECURITIES).

Note: When locking composite securities, the primary and secondary securities are also locked. In addition, composite securities cannot be write locked.

See Also

- *MSFL1_GetLastFailedLockInfo* (page 105)
- *MSFL1_GetSecurityLockedStatus* (page 114)
- *MSFL1_UnlockSecurity* (page 124)

MSFL1_MakeMSFLDate

C

```
int MSFL1_MakeMSFLDate(long *plDate,
  WORD wMonth,
  WORD wDay,
  WORD wYear);
```

Visual Basic

```
MSFL1_MakeMSFLDate(     plDate As Long,
  ByVal wMonth As Integer,
  ByVal wDay As Integer,
  ByVal wYear As Integer) As Long
```

Delphi

```
MSFL1_MakeMSFLDate(     Var plDate : longint;
  wMonth : word;
  wDay : word;
  wYear : word) : integer;
```

PowerBASIC

```
MSFL1_MakeMSFLDate(     plDate As Long,
  BYVAL wMonth As Word,
  BYVAL wDay As Word,
  BYVAL wYear As Word) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_DATE** if the constructed date is invalid

Parameters

ID	Description
plDate	Points to a long that receives the MSFL date.
wMonth	The month; January = 1, February = 2, and so on.
wDay	The day of the month.

ID	Description
wYear	The year. The year can be two or four digits. If the year is two digits, the Windows cutoff year is used to determine the century. If the Windows cutoff year is not found in the registry, a default cutoff year of twenty-nine is used. In other words, if the two-digit year is less than or equal to twenty-nine, *MSFL1_MakeMSFLDate* (page 117) will assume the century to be 2000. If the two-digit year is greater than twenty-nine, a century of 1900 will be assumed.

Remarks

- Constructs an MSFL date from its components: day, month and year.

See Also

- *MSFL1_FormatDate* (page 94)
- *MSFL1_GetDayMonthYear* (page 98)
- *MSFL1_MakeMSFLTime* (page 118),
- *MSFL1_ParseDateString* (page 120)

MSFL1_MakeMSFLTime

C

```
int MSFL1_MakeMSFLTime(long *plTime,
  WORD wHour,
  WORD wMin,
  WORD wTicks);
```

Visual Basic

```
MSFL1_MakeMSFLTime(   plTime As Long,
  ByVal wHour As Integer,
  ByVal wMin As Integer,
  ByVal wTicks As Integer) As Long
```

Delphi

```
MSFL1_MakeMSFLTime(   Var plTime : longint;
  wHour : word;
  wMin : word;
  wTicks : word) : integer;
```

PowerBASIC

```
MSFL1_MakeMSFLTime(   plTime As Long,
  BYVAL wHour As Word,
  BYVAL wMin As Word,
  BYVAL wTicks As Word) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_TIME** if the constructed time is invalid

Parameters

ID	Description
plTime	Points to a long that receives the MSFL time.
wHour	The hour; must be in 24-hour format.
wMin	The minutes.
wTicks	The ticks. In cases where the ticks are unknown or not relevant, pass in zero for the ticks.

Remarks

- Constructs an MSFL time from its components: hour, minutes and ticks.

See Also

- *MSFL1_FormatTime* (page 95)
- *MSFL1_GetHourMinTicks* (page 104)
- *MSFL1_MakeMSFLDate* (page 117)
- *MSFL1_ParseTimeString* (page 121)

MSFL1_OpenDirectory

C

```
int MSFL1_OpenDirectory(LPCSTR pszDirectory,
  char *pcDirNumber,
  int iDirOpenFlags)
```

Visual Basic

```
MSFL1_OpenDirectory(    ByVal pszDirectory As String,
  pcDirNumber As Byte,
  ByVal iDirOpenFlags As Long) As Long
```

Delphi

```
MSFL1_OpenDirectory(    pszDirectory : LPCSTR;
  Var pcDirNumber : char;
  iDirOpenFlags : integer ) : integer;
```

PowerBASIC

```
MSFL1_OpenDirectory(    pszDirectory AS ASCIIZ,
  pcDirNumber AS BYTE,
  BYVAL iDirOpenFlags As Long) As Long
```

Locking

- None

Return Values

- **MSFL_MSG_NOT_A_METASTOCK_DIR** if successful, but the directory does not contain MetaStock files
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_ALREADY_OPEN** if the application has the directory open and the MSFL_DIR_ALLOW_MULTI_OPEN flag was not passed to the open file
- **MSFL_ERR_DIR_DOES_NOT_EXIST** if the directory does not exist
- **MSFL_ERR_DUPLICATE_SECURITIES** if there are duplicate securities in the directory and the MSFL_DIR_MERGE_DUP_SECS flag was not passed to the open
- **MSFL_ERR_INVALID_DIR** if the directory is invalid
- **MSFL_ERR_TOO_MANY_DIRS_OPEN** if the application has opened the maximum number of directories (i.e. MSFL_MAX_OPEN_DIRECTORIES)
- **MSFL_ERR_USER_ID_ALREADY_IN_DIR** if a user with the same application and user name has the directory open and the MSFL_DIR_FORCE_USER_IN flag was not passed to the open

Parameters

ID	Description
pszDirectory	Points to a null-terminated string that contains the directory to open.
pcDirNumber	Points to a character that receives the directory number. The directory number can be thought of a handle to the open directory. If the open fails, the directory number is set to zero.

ID	Description
iDirOpenFlags	Specifies the open flags. The flags provide additional tasks to perform while opening the directory. Many of the tasks are provided to recover from common errors. Multiple flags can be passed in by simply bitwise OR-ing the flags (e.g. `MSFL_DIR_ALLOW_MULTI_OPEN | MSFL_DIR_MERGE_DUP_SECS`). The MSFL_DIR_NO_FLAGS is ignored when any other flags are used; the remaining flags can be used in any combination. Following is a list of the available directory open flags. • MSFL_DIR_NO_FLAGS Standard directory open. Return an error if the directory doesn't exist, if the user is already in the directory, or if there are duplicate securities in the directory. • MSFL_DIR_FORCE_USER_IN Open a directory that is already open by a user with the same application and user name. This situation can occur if the application terminated without closing the directory or if another user on the network is using the application with the same user name. This flag should only be used in response to the MSFL_ERR_USER_ID_ALREADY_IN_DIR error. • MSFL_DIR_MERGE_DUP_SECS If the directory contains duplicate securities, merge the price data for all the duplicate securities. This flag is generally used in response to the MSFL_ERR_DUPLICATE_SECURITIES error. • MSFL_DIR_ALLOW_MULTI_OPEN Allows the application to open the same directory multiple times. The MSFL keeps reference count – each time the directory is opened the reference count is incremented, each time the directory is closed the reference count is decremented. When the reference count is equal to zero, the directory is closed.

Remarks

- Opens the specified directory.
- Directories that do not contain MetaStock files can be opened; however, the MSFL_MSG_NOT_A_METASTOCK_DIR message is returned and any MSFL functions that operate with security or price data cannot be used.
- The number of concurrent open directories is limited to MSFL_MAX_OPEN_DIRECTORIES.

See Also

- *MSFL1_CloseDirectory* (page 92)
- *MSFL1_GetDirectoryNumber* (page 99)
- *MSFL1_GetDirectoryStatus* (page 100)
- *MSFL1_GetLastFailedOpenDirInfo* (page 106)

MSFL1_ParseDateString

C

```
int MSFL1_ParseDateString(long *plDate,
  LPCSTR pszDateString);
```

Visual Basic

```
MSFL1_ParseDateString(plDate As Long,
  ByVal pszDateString As String) As Long
```

Delphi

```
MSFL1_ParseDateString(Var plDate : longint;
  pszDateString : LPCSTR):integer;
```

PowerBASIC

```
MSFL1_ParseDateString(plDate As Long,
  pszDateString AS ASCIIZ) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_DATE** if the constructed date is invalid
- **MSFL_ERR_INVALID_FUNC_PARMS** if either parameter is null

Parameters

ID	Description
plDate	Points to a long that receives the MSFL date.
pszDateString	Points to a null-terminated string that contains the date. The date must be formatted according to the Windows short date format and use the same date separator (e.g., '/').

Remarks

- Constructs an MSFL date from the date string. The year can be two or four digits. If the year is two digits, the Windows cutoff year is used to determine the century. If the Windows cutoff year is not found in the registry, a default cutoff year of twenty-nine is used. In other words, if the two-digit year is less than or equal to twenty-nine, MSFL1_ParseDateString will assume the century to be 2000. If the two-digit year is greater than twenty-nine, a century of 1900 will be assumed.

See Also

- *MSFL1_FormatDate* (page 94)
- *MSFL1_GetDayMonthYear* (page 98)
- *MSFL1_MakeMSFLDate* (page 117)
- *MSFL1_ParseTimeString* (page 121)

MSFL1_ParseTimeString

C

```
int MSFL1_ParseTimeString(long *plTime,
  LPCSTR pszTimeString);
```

Visual Basic

```
MSFL1_ParseTimeString(  plTime As Long,
  ByVal pszTimeString As String) As Long
```

Delphi

```
MSFL1_ParseTimeString(  Var lTime : longint;
  pszTimeString : LPCSTR) : integer;
```

PowerBASIC

```
MSFL1_ParseTimeString(  plTime As Long,
  pszTimeString AS ASCIIZ) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_TIME** if the constructed time is invalid
- **MSFL_ERR_INVALID_FUNC_PARMS** if either parameter is null

Parameters

ID	Description
plTime	Points to a long that receives the MSFL time.
pszTimeString	Points to a null-terminated string that contains the time. The time must be formatted according to the Windows time format and use the same time separator (e.g., ':').

Remarks

- Constructs an MSFL time from the time string. If the time is read right-to-left, all fields must be present (i.e., the ticks/seconds, minutes, hours). If the time is read left-to-right, the minutes and ticks/seconds are optional. The time may be in either 24-hour format or 12-hour format. If the time is in 12-hour format, the PM symbol must be included in the time string.

See Also

- *MSFL1_FormatTime* (page 95)
- *MSFL1_GetHourMinTicks* (page 104)
- *MSFL1_MakeMSFLTime* (page 118),
- *MSFL1_ParseDateString* (page 120)

MSFL1_ReadDataRec

C

```
int MSFL1_ReadDataRec(  HSECURITY hSecurity,
  MSFLPriceRecord_struct *psPriceRec)
```

Visual Basic

```
MSFL1_ReadDataRec(      ByVal hSecurity As Long,
  psPriceRec As MSFLPriceRecord_struct) As Long
```

Delphi

```
MSFL1_ReadDataRec(      hSecurity : HSECURITY;
  Var psPriceRec : MSFLPriceRecord_struct) : integer;
```

PowerBASIC

```
MSFL1_ReadDataRec(      BYVAL hSecurity AS DWORD,
  psPriceRec AS MSFLPriceRecord_struct) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_END_OF_FILE** if the current data position is past the end of the file

Parameters

ID	Description
hSecurity	Identifies the security.
psPriceRec	Points to an *MSFLPriceRecord_struct* structure (page 84) that receives the price record.

Remarks

- Reads the price record at the current data position.
  If successful, the current data position is advanced to the next record.

See Also

- *MSFL2_ReadBackMultipleRecs* (page 126)
- *MSFL2_ReadDataRec* (page 127)
- *MSFL2_ReadMultipleRecs* (page 128)
- *MSFL2_ReadMultipleRecsByDates* (page 129)

MSFL1_SeekBeginData

C

```
int MSFL1_SeekBeginData(HSECURITY hSecurity)
```

Visual Basic

```
MSFL1_SeekBeginData(ByVal hSecurity As Long) As Long
```

Delphi

```
MSFL1_SeekBeginData(hSecurity : HSECURITY) : integer;
```

PowerBASIC

```
MSFL1_SeekBeginData(BYVAL hSecurity AS DWORD) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_LOCKED** if the security is not locked

Parameters

ID	Description
hSecurity	Identifies the security.

Remarks

- Moves the current data position to the first price record.
  Unlike MSFL1_SeekEndData, this function can be used with composite securities.

See Also

- *MSFL1_FindDataDate* (page 92)
- *MSFL1_FindDataRec* (page 93)
- *MSFL1_GetCurrentDataPos* (page 96)
- *MSFL1_SeekEndData* (page 123)

MSFL1_SeekEndData

C

```
int MSFL1_SeekEndData(HSECURITY hSecurity)
```

Visual Basic

```
MSFL1_SeekEndData(ByVal hSecurity As Long) As Long
```

Delphi

```
MSFL1_SeekEndData(hSecurity : HSECURITY) : integer;
```

PowerBASIC

```
MSFL1_SeekEndData(BYVAL hSecurity AS DWORD) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_LOCKED** if the security is not locked

Parameters

ID	Description
hSecurity	Identifies the security.

Remarks

- Moves the current data position to the end of the price data. This function is generally used to move the current data position in preparation for appending price records.

Note: This function cannot be used with composite securities.

See Also

- *MSFL1_FindDataDate* (page 92)
- *MSFL1_FindDataRec* (page 93)
- *MSFL1_GetCurrentDataPos* (page 96)
- *MSFL1_SeekBeginData* (page 123)

MSFL1_Shutdown

C

```
int MSFL1_Shutdown(void)
```

Visual Basic

```
MSFL1_Shutdown() As Long
```

Delphi

```
MSFL1_Shutdown : integer;
```

PowerBASIC

```
MSFL1_Shutdown() As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_NOT_INITIALIZED** if the MSFL is not initialized

Parameters

- None

Remarks

- Shuts down the previously initialized MetaStock file library. During shutdown, all open files are closed and the internal tables and buffers are freed.

See Also

- *MSFL1_GetMSFLState* (page 107)
- *MSFL1_Initialize* (page 115)

MSFL1_UnlockSecurity

C

```
int MSFL1_UnlockSecurity(HSECURITY hSecurity)
```

Visual Basic

```
MSFL1_UnlockSecurity(ByVal hSecurity As Long) As Long
```

Delphi

```
MSFL1_UnlockSecurity(hSecurity : HSECURITY) : integer;
```

PowerBASIC

```
MSFL1_UnlockSecurity(BYVAL hSecurity AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_SECURITY_NOT_LOCKED** if the security is not locked

Parameters

ID	Description
hSecurity	Identifies the security to unlock.

Remarks

- Unlocks the specified security.
- When the security is unlocked, any changes to the security information (e.g. first date, last date, security name, etc.) are saved to the security file. All associated files are closed.

See Also

- *MSFL1_GetSecurityLockedStatus* (page 114)
- *MSFL1_LockSecurity* (page 116)

MSFL2_GetSecurityHandles

C

```
int MSFL2_GetSecurityHandles(char cDirNumber,
  HSECURITY hStartingSecurity,
  DWORD dwMaxHandles,
  HSECURITY *pahSecurity,
  DWORD  *pdwHandleCount)
```

Visual Basic

```
MSFL2_GetSecurityHandles(ByVal cDirNumber As Byte,
  ByVal hStartingSecurity As Long,
  ByVal dwMaxHandles As Long,
  pahSecurity As Long,
  pdwHandleCount As Long) As Long
```

Delphi

```
MSFL2_GetSecurityHandles(cDirNumber : char;
  hStartingSecurity : HSECURITY;
  dwMaxHandles : DWORD;
  Var pahSecurity : HSECURITY;
  Var pdwHandleCount : DWORD) : integer;
```

PowerBASIC

```
MSFL2_GetSecurityHandles(BYVAL cDirNumber AS BYTE,
  BYVAL hStartingSecurity As Long,
  BYVAL dwMaxHandles AS DWORD,
  pahSecurity AS DWORD,
  pdwHandleCount AS DWORD) As Long
```

Locking

- None

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_DIR_NOT_OPEN** if the directory is not open
- **MSFL_ERR_INVALID_SECURITY_HANDLE** if the starting security handle is invalid

Parameters

ID	Description
cDirNumber	Identifies the directory.
hStartingSecurity	Identifies the first security of the block. To start at the first security in a directory, the handle can be set to zero or to the security handle of the first security. Otherwise, it must be set to a valid security handle within the directory.
dwMaxHandles	Specifies the maximum number of handles to be stored in the array pointed to by *pahSecurity.*
pahSecurity	Points to an array of HSECURITY that receives the block of security handles.
pdwHandleCount	Points to a DWORD that receives the actual number of handles returned in the array pointed to by *pahSecurity.*

Remarks

- Gets the security handles for a block of securities.

See Also

- *MSFL1_GetFirstSecurityInfo* (page 103)
- *MSFL1_GetLastSecurityInfo* (page 106)
- *MSFL1_GetNextSecurityInfo* (page 108)
- *MSFL1_GetPrevSecurityInfo* (page 109)
- *MSFL1_GetSecurityCount* (page 111)
- *MSFL1_GetSecurityInfo* (page 113)

MSFL2_ReadBackMultipleRecs

C

```
int MSFL2_ReadBackMultipleRecs(HSECURITY hSecurity,
  MSFLPriceRecord_struct *pasPriceRec,
  const DateTime_struct *psLastRecDate,
  WORD *pwReadCount,
  int iFindMode)
```

Visual Basic

```
MSFL2_ReadBackMultipleRecs(ByVal hSecurity As Long,
  pasPriceRec As MSFLPriceRecord_struct,
  psLastRecDate As DateTime_struct,
  pwReadCount As Integer,
  ByVal iFindMode As Long) As Long
```

Delphi

```
MSFL2_ReadBackMultipleRecs(hSecurity : HSECURITY;
  Var pasPriceRec: MSFLPriceRecord_struct;
  const psLastRecDate : DateTime_struct;
  Var pwReadCount : word;
  iFindMode : integer) : integer;
```

PowerBASIC

```
MSFL2_ReadBackMultipleRecs(BYVAL hSecurity AS DWORD,
  pasPriceRec AS MSFLPriceRecord_struct,
  psLastRecDate AS DateTime_struct,
  pwReadCount As Word,
  BYVAL iFindMode As Long) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_MSG_LESS_RECORDS_READ** if successful, but fewer records were read than requested
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_RECORDS** if the read count is invalid
- **MSFL_ERR_END_OF_FILE** if the current data position is past the end of the file
- **MSFL_ERR_SECURITY_HAS_NO_DATA** if the security has no price records

Parameters

ID	Description
hSecurity	Identifies the security.
pasPriceRec	Points to an *MSFLPriceRecord_struct* structure (page 84) array that receives the price records.
psLastRecDate	Points to a *DateTime_struct* structure (page 81) that contains the date/time of the record at which to start reading.

ID	Description
pwReadCount	Points to a WORD that contains the number of records to read. It also receives the actual number of records read. The maximum records that can be read per call is limited to MSFL_MAX_READ_WRITE_RECORDS.
iFindMode	Indicates what type of search to perform to locate the price record at which to start reading. Following are the different modes available. • MSFL_FIND_CLOSEST_PREV If an exact match is not found, find the previous closest record. • MSFL_FIND_CLOSEST_NEXT I f an exact match is not found, find the next closest record. • MSFL_FIND_EXACT_MATCH Find an exact date/time match. • MSFL_FIND_USE_CURRENT_POS Skip the find and use the current position. When this mode is used, the contents of *psLastRecDate* are ignored.

Remarks

- Reads backward from the specified date/time for the specified number of price records. If successful, the current data position is set to the previous record. Since this function reads backward the data position is set to the previous record rather than the next record.

See Also

- *MSFL1_ReadDataRec* (page 122)
- *MSFL2_ReadDataRec* (page 127)
- *MSFL2_ReadMultipleRecs* (page 128)
- *MSFL2_ReadMultipleRecsByDates* (page 129)

MSFL2_ReadDataRec

C

```
int MSFL2_ReadDataRec(  HSECURITY hSecurity,
  const DateTime_struct *psRecordDate,
  MSFLPriceRecord_struct *psPriceRec,
  int iFindMode)
```

Visual Basic

```
MSFL2_ReadDataRec(      ByVal hSecurity As Long,
  psRecordDate As DateTime_struct,
  psPriceRec As MSFLPriceRecord_struct,
  ByVal iFindMode As Long) As Long
```

Delphi

```
MSFL2_ReadDataRec(      hSecurity : HSECURITY;
  const psRecordDate : DateTime_struct;
  Var psPriceRec   : MSFLPriceRecord_struct;
  iFindMode : integer) : integer;
```

PowerBASIC

```
MSFL2_ReadDataRec(      BYVAL hSecurity AS DWORD,
  psRecordDate AS DateTime_struct,
  psPriceRec AS MSFLPriceRecord_struct,
  BYVAL iFindMode As Long) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_NO_ERR** if successful
- **MSFL_ERR_END_OF_FILE** if the current data position is past the end of the file
- **MSFL_ERR_SECURITY_HAS_NO_DATA** if the security has no price records

Parameters

ID	Description
hSecurity	Identifies the security.
psRecordDate	Points to a *DateTime_struct* structure (page 81) that contains the date/time of the record to read.
psPriceRec	Points to an *MSFLPriceRecord_struct* structure (page 84) that receives the price record.
iFindMode	Indicates what type of search to perform to locate the price record to read. Following are the different modes available. • MSFL_FIND_CLOSEST_PREV If an exact match is not found, find the previous closest record. • MSFL_FIND_CLOSEST_NEXT If an exact match is not found, find the next closest record. • MSFL_FIND_EXACT_MATCH Find an exact date/time match. • MSFL_FIND_USE_CURRENT_POS Skip the find and use the current position. When this mode is used, the contents of *psRecordDate* are ignored.

Remarks

- Reads a price record for the specified date/time.
  This function is equivalent to calling *MSFL1_FindDataDate* (page 92) to find the date/time and then calling *MSFL1_ReadDataRec* (page 122) to read the price record.

See Also

- *MSFL1_ReadDataRec* (page 122)
- *MSFL2_ReadBackMultipleRecs* (page 126)
- *MSFL2_ReadMultipleRecs* (page 128)
- *MSFL2_ReadMultipleRecsByDates* (page 129)

MSFL2_ReadMultipleRecs

C

```
int MSFL2_ReadMultipleRecs(HSECURITY hSecurity,
  MSFLPriceRecord_struct *pasPriceRec,
  const DateTime_struct *psFirstRecDate,
  WORD *pwReadCount,
  int iFindMode)
```

Visual Basic

```
MSFL2_ReadMultipleRecs(ByVal hSecurity As Long,
  pasPriceRec As MSFLPriceRecord_struct,
  psFirstRecDate As DateTime_struct,
  pwReadCount As Integer,
  ByVal iFirstFindMode As Long) As Long
```

Delphi

```
MSFL2_ReadMultipleRecs(hSecurity : HSECURITY;
  Var pasPriceRec : MSFLPriceRecord_struct;
  const psFirstRecDate : DateTime_struct;
  Var pwReadCount : word;
  iFindMode : integer) : integer;
```

PowerBASIC

```
MSFL2_ReadMultipleRecs(BYVAL hSecurity AS DWORD,
  pasPriceRec AS MSFLPriceRecord_struct,
  psFirstRecDate AS DateTime_struct,
  pwReadCount As Word,
  BYVAL iFindMode As Long) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_MSG_LESS_RECORDS_READ** if successful, but fewer records were read than requested
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_RECORDS** if the read count is invalid
- **MSFL_ERR_END_OF_FILE** if the current data position is past the end of the file
- **MSFL_ERR_SECURITY_HAS_NO_DATA** if the security has no price records

Parameters

ID	Description
hSecurity	Identifies the security.
pasPriceRec	Points to an *MSFLPriceRecord_struct* structure (page 84) array that receives the price records.
psFirstRecDate	Points to a *DateTime_struct* structure (page 81) that contains the date/time of the record at which to start reading.
pwReadCount	Points to a WORD that contains the number of records to read. It also receives the actual number of records read. The maximum records that can be read per call is limited to MSFL_MAX_READ_WRITE_RECORDS.
iFindMode	Indicates what type of search to perform to locate the first price record to read. Following are the different modes available. • MSFL_FIND_CLOSEST_PREV If an exact match is not found, find the previous closest record. • MSFL_FIND_CLOSEST_NEXT If an exact match is not found, find the next closest record. • MSFL_FIND_EXACT_MATCH Find an exact date/time match. • MSFL_FIND_USE_CURRENT_POS Skip the find and use the current position. When this mode is used, the contents of *psFirstRecDate* are ignored.

Remarks

- Reads the number of price records indicated. If successful, the current data position is set to the next price record.

See Also

- *MSFL1_ReadDataRec* (page 122)
- *MSFL2_ReadBackMultipleRecs* (page 126)
- *MSFL2_ReadDataRec* (page 127)
- *MSFL2_ReadMultipleRecsByDates* (page 129)

MSFL2_ReadMultipleRecsByDates

C

```
int MSFL2_ReadMultipleRecsByDates(HSECURITY hSecurity,
  MSFLPriceRecord_struct *pasPriceRec,
  const DateTime_struct *psFirstRecDate,
  const DateTime_struct *psLastRecDate,
  WORD *pwMaxReadCount,
  int iFirstFindMode)
```

Visual Basic

```
MSFL2_ReadMultipleRecsByDates(ByVal hSecurity As Long,
  pasPriceRec As MSFLPriceRecord_struct,
  psFirstRecDate As DateTime_struct,
  psLastRecDate As DateTime_struct,
  pwMaxReadCount As Integer,
  ByVal iFirstFindMode As Long) As Long
```

Delphi

```
MSFL2_ReadMultipleRecsByDates(hSecurity : HSECURITY;
  Var pasPriceRec : MSFLPriceRecord_struct;
  const psFirstRecDate : DateTime_struct;
  const psLastRecDate : DateTime_struct;
  Var pwMaxReadCount : word;
  iFirstFindMode : integer) : integer;
```

PowerBASIC

```
MSFL2_ReadMultipleRecsByDates(BYVAL hSecurity AS DWORD,
  pasPriceRec AS MSFLPriceRecord_struct,
  psFirstRecDate AS DateTime_struct,
  psLastRecDate AS DateTime_struct,
  pwMaxReadCount As Word,
  BYVAL iFirstFindMode As Long) As Long
```

Locking

- Prevent Write Lock

Return Values

- **MSFL_MSG_LESS_RECORDS_READ** if successful, but fewer records were read than requested
- **MSFL_MSG_MORE_RECORDS_IN_RANGE** if successful, but there were more records in the date range than the array could hold
- **MSFL_NO_ERR** if successful
- **MSFL_ERR_INVALID_RECORDS** if the read count is invalid

Parameters

ID	Description
hSecurity	Identifies the security.
pasPriceRec	Points to an *MSFLPriceRecord_struct* structure (page 84) array that receives the price records.
psFirstRecDate	Points to a *DateTime_struct* structure (page 81) that contains the date/time of the record at which to start reading.
psLastRecDate	Points to a *DateTime_struct* structure (page 81) that contains the date/time of the record at which to stop reading.
pwMaxReadCount	Points to a WORD that contains the maximum number of records to read (i.e. the maximum number of records the array can hold). It also receives the actual number of records read.
iFirstFindMode	Indicates what type of search to perform to locate the first price record to read. Following are the different modes available. • MSFL_FIND_CLOSEST_PREV If an exact match is not found, find the previous closest record. • MSFL_FIND_CLOSEST_NEXT If an exact match is not found, find the next closest record. • MSFL_FIND_EXACT_MATCH Find an exact date/time match.

Remarks

- Reads the price record(s) for the date range indicated. It stops reading when it reaches a price record with a date/time greater than *psLastDate* or when the maximum number of records have been read. If successful, the current data position is set to the next price record.

See Also

- *MSFL1_ReadDataRec* (page 122)
- *MSFL2_ReadBackMultipleRecs* (page 126)
- *MSFL2_ReadDataRec* (page 127)
- *MSFL2_ReadMultipleRecs* (page 128)

Messages and Errors

Error Codes

The following is a list of the possible error codes that can be returned from the MSFL functions. On successful completion, MSFL_NO_ERR or an MSFL message code is returned; in the event of an error, the specific MSFL error code is returned.

Note: The *MSFL1_GetErrorMessage* (page 102) function can be used to generate an error message string. These are listed below.

-400: MSFL_ERR_NOT_INITIALIZED
Attempted to use an MSFL function without first initializing the MSFL.

-399: MSFL_ERR_ALREADY_INITIALIZED
Attempted to initialize the MSFL after it had already been initialized.

-398: MSFL_ERR_MSFL_CORRUPT
The MSFL or operating system is corrupt. The internal MSFL tables have been damaged or operating system is now unstable. The MSFL must be shutdown.

-397: MSFL_ERR_OS_VER_NOT_SUPPORTED
The Windows version is below the minimum required (Windows 95 or Windows NT 3.1).

-396: MSFL_ERR_SHARE_NOT_LOADED
File sharing is not loaded.

-395: MSFL_ERR_INSUFFICIENT_FILES
Attempted to initialize the MSFL with insufficient file handles.

-394: MSFL_ERR_INSUFFICIENT_MEM
Insufficient memory to perform requested function (i.e. the function called requires more memory on the heap).

-393: MSFL_ERR_INVALID_USER_ID
The user name and/or application name are invalid.

-392: MSFL_ERR_INVALID_TEMP_DIR
The Windows temp directory is invalid.

-391: MSFL_ERR_DLL_INCOMPATIBLE
The MSFL DLL is incompatible with the application.

-375: MSFL_ERR_INVALID_DRIVE
The drive is invalid.

-374: MSFL_ERR_INVALID_DIR
The directory is invalid.

-373: MSFL_ERR_DIR_DOES_NOT_EXIST
Directory does not exist.

-372: MSFL_ERR_UNABLE_TO_CREATE_DIR
Unable to create the directory.

-371: MSFL_ERR_DIR_ALREADY_OPEN
Directory is already open. Attempted to open an open directory.

-370: MSFL_ERR_DIR_NOT_OPEN
Attempted to call an MSFL function with a directory that is not open.

-369: MSFL_ERR_TOO_MANY_DIRS_OPEN
The maximum number of concurrent open directories has already been reached; opening another directory is not possible.

-368: MSFL_ERR_ALREADY_A_MS_DIR
Attempted to build MetaStock files in an existing MetaStock directory.

-367: MSFL_ERR_NOT_A_MS_DIR
The directory is not a MetaStock directory.

-366: MSFL_ERR_DIR_IS_BUSY
The files are in a state where only one user can access them.

-365: MSFL_ERR_USER_ID_ALREADY_IN_DIR
This user ID (i.e. program and user name) already has this directory open.

-364: MSFL_ERR_TOO_MANY_USERS_IN_DIR
The maximum number of users have already opened the directory.

-363: MSFL_ERR_INVALID_USER
The user is invalid because another user with the same application name and user name opened the directory.

-362: MSFL_ERR_NON_MSFL_USER_IN_DIR
The directory is in use by a non-MSFL application, the data cannot be accessed until the single user application is finished.

-361: MSFL_ERR_DIR_IS_READ_ONLY
The directory is read-only; therefore, the operation cannot be performed.

-360: MSFL_ERR_MAX_FILES_IN_TEMP_DIR
Too many MSFL files exist in the Windows temp directory.

-355: MSFL_ERR_INVALID_XMASTER_FILE
The XMASTER file is corrupt.

-354: MSFL_ERR_INVALID_INDEX_FILE
The index file is corrupt.

-353: MSFL_ERR_INVALID_LOCK_FILE
The lock file is corrupt.

-352: MSFL_ERR_INVALID_SECURITY_FILE
The security file is corrupt.

-351: MSFL_ERR_INVALID_USERS_FILE
The user file is corrupt.

-350: MSFL_ERR_CRC_ERROR
A CRC error occurred while accessing a file.

-349: MSFL_ERR_DRIVE_NOT_READY
The drive is not ready.

-348: MSFL_ERR_GENERAL_FAILURE
A general failure occurred while accessing the disk.

-347: MSFL_ERR_MISC_DISK_ERROR
A general disk error occurred while accessing the disk.

-346: MSFL_ERR_SECTOR_NOT_FOUND
Sector not found.

-345: MSFL_ERR_SEEK_ERROR
An error occurred while seeking in the file.

-344: MSFL_ERR_UNKNOWN_MEDIA
Unknown disk media type.

-343: MSFL_ERR_WRITE_PROTECTED
The disk is write protected.

-342: MSFL_ERR_DISK_IS_FULL
The disk is full, unable to write data.

-341: MSFL_ERR_NOT_SAME_DEVICE
The device (e.g. disk drive) has changed.

-340: MSFL_ERR_NETWORK_ERROR
A network error occurred while accessing files on the network.

-325: MSFL_ERR_LOCK_VIOLATION
Unable to unlock a locked region of a file.

-324: MSFL_ERR_INVALID_LOCK_TYPE
The lock type is an unknown type.

-323: MSFL_ERR_FILE_LOCKED
File is locked by another user.

-322: MSFL_ERR_TOO_MANY_SEC_LOCKED
The maximum number of securities are already locked – the application cannot lock additional securities.

-321: MSFL_ERR_SECURITY_LOCKED
Security is locked by another user.

-320: MSFL_ERR_SECURITY_NOT_LOCKED
The security is not locked, but must be to perform the operation.

-319: MSFL_ERR_IMPROPER_LOCK_TYPE
The lock type is incorrect for the operation requested.

-300: MSFL_ERR_END_OF_FILE
End of the file.

-299: MSFL_ERR_ERROR_OPENING_FILE
Unable to open the file.

-298: MSFL_ERR_ERROR_READING_FILE
Error reading the file.

-297: MSFL_ERR_ERROR_WRITING_FILE
Error writing to the file.

-296: MSFL_ERR_FILE_DOESNT_EXIST
File does not exist.

-295: MSFL_ERR_INVALID_FILE_HANDLE
The file handle is invalid.

-294: MSFL_ERR_PERMISSION_DENIED
Permission to access a file was denied.

-293: MSFL_ERR_SEEK_PAST_EOF
The seek went past the end of the file.

-292: MSFL_ERR_MISC_FILE_ERROR
A miscellaneous file error occurred while accessing a file.

-275: MSFL_ERR_UNABLE_TO_READ_ALL
Unable to read all the records requested.

-274: MSFL_ERR_UNABLE_TO_WRITE_ALL
Unable to write all the records requested.

-250: MSFL_ERR_ALL_SYMB_NOT_LOADED
One or more of the symbols in the directory were invalid; thus, all of the securities in the directory were not loaded.

-249: MSFL_ERR_UNABLE_TO_RESYNCH
Unable to resynchronize the security files.

-248: MSFL_ERR_FILES_IN_DIR_CHANGED
The files in the directory have changed (i.e. they are not the same files the directory was opened with).

-247: MSFL_ERR_UNRECOGNIZED_VERSION
The MetaStock files are not a recognized version.

-225: MSFL_ERR_INVALID_COMP_SYMBOL
The composite symbol is invalid.

-224: MSFL_ERR_INVALID_SYMBOL
The ticker symbol is invalid.

-200: MSFL_ERR_DIFFERENT_DATA_FORMATS
The securities are of different data formats (i.e. the periodicity, interval, or price fields of the securities do not match).

-199: MSFL_ERR_DUPLICATE_SECURITIES
Attempted to open a directory that contains duplicate securities.

-198: MSFL_ERR_DUPLICATE_SECURITY
Adding the security would duplicate an existing security.

-197: MSFL_ERR_PRIMARY_SEC_NOT_FOUND
The primary security of the composite cannot be found.

-196: MSFL_ERR_SECONDARY_SEC_NOT_FOUND
The secondary security of the composite cannot be found.

-195: MSFL_ERR_SECURITY_HAS_COMPOSITES
Security cannot be deleted because there are one or more composites that depend on the security.

-194: MSFL_ERR_SECURITY_HAS_NO_DATA
There is no price data for the security.

-193: MSFL_ERR_SECURITY_IS_A_COMPOSITE
Security is a composite.

-192: MSFL_ERR_SECURITY_NOT_COMPOSITE
Security is not a composite.

-191: MSFL_ERR_SECURITY_NOT_FOUND
The security was not found in the directory.

-190: MSFL_ERR_TOO_MANY_SECURITIES
The maximum number of securities per directory has already been reached; adding additional securities is not possible.

-189: MSFL_ERR_TOO_MANY_COMPOSITES
The maximum number of composites per directory has already been reached.

-188: MSFL_ERR_SECURITIES_ARE_THE_SAME
The securities are the same security. Attempted to merge the security with itself.

-175: MSFL_ERR_INVALID_DATE
The date is invalid.

-174: MSFL_ERR_INVALID_TIME
The time is invalid.

-173: MSFL_ERR_INVALID_INTERVAL
The interval is invalid.

-172: MSFL_ERR_INVALID_PERIODICITY
The periodicity is invalid.

-171: MSFL_ERR_INVALID_OPERATOR
The composite operator is invalid.

-170: MSFL_ERR_INVALID_FIELD_ORDER
The data fields used are not in the same order as documented on page 80.

-169: MSFL_ERR_INVALID_RECORDS
The record numbers are not within a valid range for the operation.

-168: MSFL_ERR_INVALID_DISPLAY_UNITS
The display units are outside the valid range.

-167: MSFL_ERR_INVALID_SECURITY_HANDLE
The security handle is not valid.

-150: MSFL_ERR_ADDING_WOULD_OVERFLOW
Inserting or adding records would exceed the maximum number of records that can be stored.

-149: MSFL_ERR_DATA_FILE_IS_FULL
The price data file is full.

-148: MSFL_ERR_DATA_RECORD_NOT_FOUND
A matching price record was not found for the date/time.

-147: MSFL_ERR_DATA_NOT_SORTED
The price data is not in date/time sort order.

-146: MSFL_ERR_DATE_AFTER_LAST_REC
The date/time requested is after the date/time of the last price record.

-145: MSFL_ERR_DATE_BEFORE_FIRST_REC
The date/time requested is before the date/time of the first price record.

-144: MSFL_ERR_RECORD_IS_A_DUPLICATE
The record duplicates an existing record.

-143: MSFL_ERR_RECORD_OUT_OF_RANGE
The record number is out of range.

-142: MSFL_ERR_RECORD_NOT_FOUND
The security record was not found – most likely an invalid security handle.

-125: MSFL_ERR_BUFFER_NOT_ATTACHED
A composite buffer was not attached to the locked composite.

-124: MSFL_ERR_INVALID_FUNC_PARMS
One or more of the function parameters are invalid.

-123: MSFL_ERR_UNKNOWN_FIELDS_REQ
The number of data fields used is below the minimum or above the maximum.

-100: MSFL_ERR_INVALID_FUNCTION_CALL
The MSFL DLL is not initialized correctly to perform the request function call. Refer to the “Initialization” section (page 87) for details on initializing the MSFL DLL.

0: MSFL_NO_ERR
Operation completed successfully.

Message Codes

The following is list of the possible message codes that can be returned from some MSFL functions. The message codes are positive return codes; whereas the error codes are negative return codes. Thus, if an MSFL function is successful the error will be equal to or greater than MSFL_NO_ERR.

0: MSFL_NO_MSG
No message.

1: MSFL_MSG_NOT_A_MetaStock_DIR
Not a MetaStock data directory.

2: MSFL_MSG_CREATED_DIR
Created directory.

3: MSFL_MSG_BUILT_MetaStock_DIR
Created empty MetaStock files in the directory.

4: MSFL_MSG_CREATED_N_BUILT_DIR
Created directory and empty MetaStock files.

5: MSFL_MSG_FIRST_SECURITY_IN_DIR
This is the first security in the directory.

6: MSFL_MSG_LAST_SECURITY_IN_DIR
This is the last security in the directory.

25: MSFL_MSG_NOT_AN_EXACT_MATCH
The record found was not an exact match.

50: MSFL_MSG_OVERWROTE_RECORDS
Overwrote existing records.

51: MSFL_MSG_LESS_RECORDS_DEL
Fewer records were deleted than requested.

52: MSFL_MSG_LESS_RECORDS_READ
Fewer records were read than requested.

53: MSFL_MSG_MORE_RECORDS_IN_RANGE
There are more records within the specified date range.

Change Record

The following section lists (in reverse version order) a short description of all changes made in earlier versions of the MSFL and the MSFL Developer's Kit.

Changes in Version 9.0

The 9.0 version of the MSFL has only minor changes.

- The PowerBasic sample app for the MSFL and MSX uses the latest version (i.e., version 7.03).

Changes in Version 8.0

- The maximum number of securities per directory (i.e. MSFL_MAX_NUM_OF_SECURITIES) was increased from 2,000 to 6,000.
- A few minor problems were fixed.

Changes in Version 7.2

- The performance was increased for many operations.
- The following functions were added:
    - *MSFL1_FormatDate* (page 94)
    - *MSFL1_FormatTime* (page 95)
    - *MSFL1_GetDayMonthYear* (page 98)
    - *MSFL1_GetHourMinTicks* (page 104)
    - *MSFL1_MakeMSFLDate* (page 117)
    - *MSFL1_MakeMSFLTime* (page 118)
    - *MSFL1_ParseDateString* (page 120)
    - *MSFL1_ParseTimeString* (page 121)
- A PowerBASIC sample was added.
- A C console sample was added.
- Project/make files are provided for the Visual C++ 6.0, Borland C++ Builder 4, and gcc 2.95.2 compilers.
- A few minor problems were fixed.

Changes in Version 7.0

- A Borland C++ Builder sample was added.
- A few minor problems were fixed.

Changes in Version 6.51

- The allowed date range was expanded from the 1900's to include the 1800's and up to 31 December 2200. The new valid date range is from 1 January 1800 to 31 December 2200.
- The tick count for all times was increased from two digits to three digits. See the **Formats** section (page 79) for details on the new time format.
- The maximum number of price records per security (i.e. MSFL_MAX_DATA_RECORDS) was increased from 32,766 to 65,500.
- The maximum number of price records per read/write (i.e. MSFL_MAX_READ_WRITE_RECORDS) was increased from 32,766 to 65,500.
- A new flag (i.e. MSFL_DIR_ALLOW_MULTI_OPEN) was added to the "MSFL1_OpenDirectory" function allowing a folder to be opened more than once. See *MSFL1_OpenDirectory* (page 119) for details.
- The Visual Basic types and Delphi records were changed to match the documentation as well as the C/C++ structures.

Changes in Version 6.5

- The maximum number of securities per directory (i.e. MSFL_MAX_NUM_OF_SECURITIES) was increased from 255 to 2,000.
- The maximum length of a security name (i.e. MSFL_MAX_NAME_LENGTH) was increased from 16 to 45 characters.
- The maximum number of price records per read/write (i.e. MSFL_MAX_READ_WRITE_RECORDS) was increased from 1,927 to 32,766.
- Universal naming convention (UNC) and long file names support was added.
- The library was converted from a C library to a 32-bit DLL to allow access via Visual Basic, Delphi, and other development environments.
- Security handles replaced data requests and extended symbols.
- String error messages were added.
- Structure types where changed for 32-bit.
- A total size member was added to many of the structures.
- The following functions were added:
    - *MSFL1_GetDirNumberFromHandle* (page 100)
    - *MSFL1_GetErrorMessage* (page 102)
    - *MSFL1_GetSecurityHandle* (page 111)
    - *MSFL1_GetSecurityID* (page 112)
    - *MSFL2_GetSecurityHandles* (page 125)

MSFL Index

Index