Metagraphics TypeServer

Programming Reference Manual

User Name

Serial Number

Registration

www.metagraphics.com

WARRANTY

With respect to the physical diskette and documentation enclosed herein, Metagraphics warrants the same to be free from defects in materials and workmanship for a period of 30 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Metagraphics will replace at no charge the defective diskette or documentation. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, special, incidental, consequential, or other similar claims.

METAGRAPHICS SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO DEFECTS IN THE DISKETSE AND DOCUMENTATION. IN NO EVENT SHALL METAGRAPHICS BE LIABLE FOR ANY LOSS OF PROFIT OR ANY COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL OR OTHER DAMAGES.

TRADEMARK

Metagraphics TypeServer™, **MetaWINDOW**, **Media!Lab**, **FontWINDOW**, **FontBUILDER** and **TypeBUILDER** are trademarks of Metagraphics.

COPYRIGHT

METAGRAPHICS
PO Box 225
Woodinville, WA 98072-0225 U.S.A.

Phone: 425-844-1110
Fax: 425-844-1112
Web: .... http://www.metagraphics.com
Email: .... sales@metagraphics.com
Support: .... support@metagraphics.com

10 9 8 7 6 5 4 3 2 1 Printed in U.S.A.

Contents

Chapter 1 - Introduction 1

Chapter 2 - Installation 7

Chapter 3 - TypeServer Example Programs 9

Chapter 1 - Introduction

Welcome To TypeServer

Welcome to **Metagraphics TypeServer**™. TypeServer is an advanced TrueType rasterizing engine for real-time and embedded applications. Using industry standard TrueType fonts, TypeServer creates typeset-quality bitmap text for virtually any device, at any size, and at any resolution. TypeServer is designed for use in tight memory-constrained environments, and is portable for use on a wide variety of platforms and processors. Supporting both ASCII and UNICODE character sets, TypeServer works with international and multilingual TrueType fonts.

- Advanced TrueType rasterizing toolkit for use in real-time and embedded OEM applications.
- Uses industry-standard TrueType fonts for creating typeset-quality bitmap text for any device, at any size, and at any resolution.
- Supports both 8-bit ASCII and 16-bit Unicode character sets.
- Works with TrueType fonts stored either on disk, in memory, or in ROM.
- Written in pure ANSI C - platform and processor independent.
  Includes class libraries for use with C++
- Easy to use object-oriented API.
- Easily interfaces with other graphic programming tools.
- Tight fast code - Performance-tuned C components for popular processors,
  plus enhanced assembly language optimizations for Intel CPU's.
- Fully ROMable – Reentrant thread-safe design uses no static variables,
  no floating-point math, and is little-endian/big-endian neutral.
- Royalty-Free application distribution.

TypeServer's tight portable design makes it ideal for use in a wide variety of products needing high-quality scaleable fonts and type. Applications include: real-time and embedded products, operating systems, medical instrumentation, handheld computers, personal digital assistants (PDA's), web browsers, multimedia servers, television set-top boxes, printer engines, Internet appliances, avionic displays, industrial controls, web applications, and more.

CrystalType

TypeServer incorporates Metagraphics' **CrystalType**™ rendering technology that products typeset-quality text at virtually any size or resolution. CrystalType performs optimized font rasterization for monochrome, anti-alias grayscale, anti-alias color, or LCD display bitmaps.

- Performs rasterization of TrueType character outlines into typeset-quality bitmap text.
- Applies advanced anti-aliasing to produce clean well-formed type that is easy to read even at small sizes.
- Maintains precise sub-pixel character positioning accurate to 1/64th or 1/16th of a pixel (source code compile-time select).
- Performs grayscale and color-blend rasterization to virtually any type of bitmap device.

TypeServer Features

TypeServer prvoides an advanced TrueType rasterizing engine incorporating the following features:

Industry-Standard TrueType Fonts

TypeServer creates typeset-quality bitmap characters and text using industry-standard TrueType fonts. TrueType offers the largest source of freeware, shareware and OEM fonts available. TypeServer supports both regular TrueType fonts (.ttf files), and TrueType font collections (.ttc files) that combine multiple TrueType fonts within a single file.

ASCII/Unicode Support

TypeServer's drawing functions allow you to render text compiled either within an ASCII or Unicode programming environment. Additionally TypeServer provides ASCII and Unicode type-specific functions if your application needs to use character encodings of a specific type. For example, an application built in an ASCII programming environment can render Unicode text using TypeServer's Unicode-specific drawing functions. Similarly applications built in a Unicode programming environment can render ASCII encoded text using TypeServer's ASCII-specific drawing functions. This flexibility allows you to write portable applications that can be compiled on different platforms in either ASCII or Unicode environments. Appendix A provides additional information on writing portable code that can be compiled and used on either ASCII or Unicode based platforms.

Precision Type Quality

TypeServer incorporates Metagraphics' CrystalType™ rendering technology that produces high quality, clean character output. Using scaleable outline definitions within TrueType fonts, CrystalType performs optimized rendering for either monochrome, anti-aliased grayscale, anti-aliased color or LCD display output. CrystalType maintains sub-pixel accuracy for precise character rendering and positioning.

To maximize performance TypeServer uses fast fixed-point math rather than floating-point (additional information on fixed-point data types and operations is contained in Appendix A of the **Metagraphics C/C++ Programming Guidelines** manual).

TypeServer's fix-point math provides sub-pixel accuracy that is scalable to optimize operations for a specific processor. For 32- and 64-bit CPU's, computations and positions are maintained to an accuracy of 1/64th of a pixel. For 16-bit and 8-bit CPU's, sub-pixel accuracy is usually maintained in units of 1/16ths of a pixel.   (1/64th or 1/16th sub-pixel accuracy is a TypeServer source code compile-time define option.)

Real-Time and Embedded Systems

TypeServer has been designed from the ground up for use in embedded and real-time operating environments. TypeServer uses no static variables and all functions are fully reentrant.

Thread-Safe Design

TypeServer supports reentrant operations in multi-threaded environments using synchronizing mutex (mutually exclusive) functions. A mutex is invoked to acquire a lock on a critical system resource before executing a thread-critical code sequence. A requesting thread will be blocked if another thread has already acquired the lock on the same resource. TypeServer's use of mutex interlocks support insures robust and reentrant operation in multi-threaded environments.

Disk, Memory or ROM-Based Fonts

TypeServer uses Metagraphics' enhanced "flash-frame I/O" that supports fonts resident either on disk, in RAM memory, or in ROM. Flash-frame I/O uses pointers to directly access font data, eliminating the need and overhead to repeatedly transfer small data records through I/O buffers. In the case of memory or ROM-based fonts, the entire font is accessed directly as one contiguous block.

Reliability

TypeServer has been extensively tested on multiple platforms and operating systems. Internally, TypeServer implements robust error detection and reporting mechanisms that aid in quickly detecting and isolating run-time errors.

Performance Optimized Rendering

TypeServer's base design is written in pure ANSI C and is optimized for fast type rendering. TypeServer "release" libraries provided with the **TypeServer Source Code Site Kit** provide additional assembly-language optimized functions for Intel x86/Pentium based platforms.

Designed to CMM Software Engineering Standards

TypeServer has been designed and written using the Capability Maturity Model (CMM) software engineering standards developed by Carnegie Mellon University and the U.S. Department of Defense. For additional information on software engineering CMM standards please see the Carnegie Mellon University web site at http://www.sei.cmu.edu/cmm/cmm.html. Information on Metagraphics software programming standards is provided in the "**Metagraphics C/C++ Programming Guidelines**" manual, also available on-line at http://www.metagraphics.com/pubs/MetagraphicsCodingGuide.pdf.

Platform and Processor Independent

The TypeServer source code is written in pure ANSI C for ease in porting to other processors and operating system platforms. TypeServer is big-endian/little-endian neutral, and uses no floating point math operations. TypeServer has already been tested with many popular compilers and hardware platforms. An updated list of compilers and platforms directly supported by TypeServer is available on the Metagraphics TypeServer web site at http://www.metagraphics.com/typeserver/platforms.htm.

Real-Time and Embedded Operating Systems

TypeServer's re-entrant and thread-safe design makes it ideal for use in real-time and embedded operating system products. TypeServer uses no static variables. An updated list of real-time and embedded products supported by TypeServer is available on the Metagraphics TypeServer web site at http://www.metagraphics.com/typeserver/platforms.htm#RTOS.

x86/Pentium Protected-Mode Environments

TypeServer is shipped with pre-compiled libraries for use with popular Intel x86/Pentium 16- and 32-bit protected-mode environments. An updated list of protected-mode platforms directly supported by TypeServer is available on the Metagraphics TypeServer web site at http://www.metagraphics.com/typeserver/platforms.htm#DPMI.

Microsoft Windows

TypeServer also ships with pre-compiled libraries for use on Win32 platforms including Windows 2000/Me/98/95. An updated list of Windows supported platforms and compilers is available on the Metagraphics TypeServer web site at http://www.metagraphics.com/typeserver/platforms.htm#WIN32.

Portable to New Platforms and Processors

TypeServer's native ANSI C design makes it easily portable to new processors, C compilers and operating system platforms. Information on porting TypeServer to new platforms and compilers is provided in the **TypeServer Source Code Design Manual**. Additional new processors and OS platforms are added to TypeServer based on customer needs and requests. An updated list of supported processors, compilers and OS platforms is maintained on the Metagraphics TypeServer web site at http://www.metagraphics.com/typeserver/platforms.htm.

TypeServer Utilities

TypeServer includes the following Windows application utilities:

TypeEmbedder.exe

Windows program that converts binary TrueType fonts (.ttf) into C header files (.h) that can be compiled and linked directly into your application.

TypeViewer.exe

Windows program to display TrueType fonts, validate font integrity, and display internal font information.

TypeServer Products

Metagraphics offers two TypeServer products: **TypeServer Developer Kit**, and **TypeServer Source Code Site Kit**.

TypeServer Developer Kit

The **TypeServer Developer Kit** provides the necessary libraries, documentation and license for a single programmer to develop applications using TypeServer. The **TypeServer Developer Kit** includes the ***TypeServer Programming Reference Manual***; TypeServer Developer CD containing: TypeServer developer libraries, TypeServer header files, example programs, and help files; and one year *EXPRESS Support Service*.

TypeServer Source Code Site Kit

(**TypeServer Developer Kit** prerequisite)
TypeServer Source Code Site Kit provides the additional components, documentation and licensing needed for optimizing TypeServer for use in a commercial application product. The **TypeServer Source Code Site Kit** includes the ***TypeServer Source Code Design Manual***, TypeServer Royalty-Free Application Distribution License, TypeServer "Release" optimized libraries, TypeServer C Source Code, TypeServer Assembly Language Source Code (Intel processors), Source Code Site License, and one year *EXPRESS Source Code Service*.

Developing with TypeServer

To speed development, TypeServer provides separate libraries for development, release and internal source code debug needs. Conditional `#define`'s within the TypeServer source code support three types of library builds: "Develop", "Release", and TypeServer Source Code "Debug".

"Develop" Libraries

Developer libraries are provided with the **TypeServer Developer Kit**. These libraries include expanded error checking and reporting to aid in quickly isolating application programming errors during development. (Expanded error checking adds overheads, however, both in increased size and slower performance.)

"Release" Libraries

"Release" libraries are provided with the **TypeServer Source Code Site Kit**. These libraries are designed for finished applications, and include reduced internal error checking and optimize TypeServer for maximum performance and minimum size. Using the configuration options in the TypeServer source code, you can create custom release libaries that are smaller and faster including only those features used by your application.

"Debug" Libraries

TypeServer source code "Debug" libraries are provided with the **TypeServer Source Code Site Kit**. These libraries enable source level debugging of the TypeServer source code itself.

Using This Manual

This manual provides an introduction to TypeServer including its basic features and a general programming overview. Several additional resources provide further detailed information on TypeServer use, programming and design.

Additional Resources

The following additional documentation is available covering topics of Metagraphics TypeServer use, programming, design and operation.

Metagraphics C/C++ Programming Guidelines

The **Metagraphics C/C++ Programming Guidelines** documents the programming guidelines used in developing Metagraphics products. The Coding Guide documents basic naming conventions, data types, code formatting, documentation, programming design, code optimization, global utility functions, macros, filename conventions, and other program design and coding conventions. A copy of this document may be downloaded from the Metagraphics web site at http://www.metagraphics.com/pubs/MetagraphicsCodingGuide.pdf.

Metagraphics TypeServer Programming Reference Manual

The **Metagraphics TypeServer Programming Reference Manual** (this manual) provides detailed reference information for programming applications using Metagraphics TypeServer. The Programming Reference Manual documents TypeServer structures, data types, enumerations, macros, math and utility functions. The bulk of the reference provides in-depth descriptions for TypeServer server, font and strike functions, plus information on function calling parameters and data structures.

Metagraphics TypeServer Source Code Design Manual

The **Metagraphics TypeServer Source Code Design Manual** provides detailed information on the internal design and operation of the Metagraphics TypeServer code. (This manual is only available to developers with the **Metagraphics TypeServer Source Code Site Kit**.)

Chapter 2 - Installation

Hardware and Software Requirements

To use the TypeServer Development Toolkit you will need a PC compatible system with a minimum of 32K memory (64K or more recommended) running Microsoft Windows 95/98/Me, Windows 2000 or Windows NT4. TypeServer will need between 2Mb to 4Mb of disk space to store executables, help files, example programs, supplementary documentation, and sample TrueType fonts.

Directory Structure

The default TypeServer installation directory is:

```
C:\Dev\TypeServ\
```

When running the TypeServer setup program you may change the default directory to another path of your choosing. Within the `\TypeServ` directory, subdirectories are organized in the following manner:

```
📁 \dev           root level development directory (name what you like)

   📁 TypeServ          TypeServer directory
      📁 bin               executable utilities
      📁 doc               documentation files
      📁 help              help files
      📁 include           public .h/.hpp/.mk include files

      📁 examples          example programs
         📁 hello-ts          hello-ts.c example for 32-bit protected mode
         📁 typeview          typeview.c example for 32-bit protected mode
         📁 typeviewer        typeviewer.cpp example for Microsoft WIN32
         📁 {-others-}        (other example programs)

      📁 lib               public TypeServer library files
         📁 _debug            TypeServer source code debug libraries
         📁 _develop          development/test libraries
         📁 _release          release optimized libraries

      📁 src               TypeServer source code (with source code product)
```

Installing TypeServer

Before installing TypeServer, please review the *Metagraphics TypeServer Software License Agreement* that is included with your product. If you have any questions regarding this agreement, please contact us directly either by phone or email to sales@metagraphics.com.

IMPORTANT - Please also review the README file included on your TypeServer distribution disk for last minute details, additions or clarifications that didn't make the printed documentation.

To install the Metagraphics TypeServer Developer Toolkit:

1. Insert the TypeServer distribution disk into your drive.
2. From the Windows "Start" menu, select "Run…".
3. Type "`d:\setup`". (If your drive is not `d:`, type the appropriate letter instead.)
4. Choose "Ok" to install.
5. Follow the instructions on screen.

Product Registration

Metagraphics offers comprehensive technical support to assist individuals, consultants and corporate developers to fully utilize your graphic programming tools. Before we can help you, however, *we need to know who you are!* Please take a few moments now to complete and return the product registration card enclosed with your TypeServer product, or even easier, complete the on-line product registration form at http://www.metagraphics.com/register/. Only as a registered user can you access the full benefits of your Metagraphics product:

- Technical support.
- Notification and download access to free service updates.
- Access to registered-developer on-line support pages
- Notification and special pricing on upgrades and new products.
- Subscription to Metagraphics' **MetaTRENDS** electronic newsletter for the latest information on service updates, programming techniques, new product releases, advance product announcements and other important news.

Next Steps

After installing TypeServer you may either continue and build run one of the **TypeServer Example Programs** (Chapter 3, next), or skip ahead to the **TypeServer Basics Overview** (Chapter 4).

Chapter 3 - TypeServer Example Programs

Sample Programs

TypeServer includes a set of example programs for each supported platform and compiler showing how to use TrueType fonts within an application program. Before compiling and running the TypeServer example programs or your own application programs, you will minimally need to set the paths to the TypeServer "`include\`" and "`library\`" directories so that your compiler and linker will find the necessary header and library files.

IMPORTANT - Please refer to the README file provided on your TypeServer distribution disk for information about new sample programs and platforms that were added after this manual was printed.

Building WIN32 Applications with Microsoft Visual C/C++

Before building applications for WIN32, you need to add the paths for the TypeServer "`\include`" and "`\lib`" directories to your Visual C++ "**Tools | Options... | Directories**" settings.

Start Microsoft Visual C and from the main menu select "**Tools | Options... | Directories**". Under the "**Show directories for:**" menu, select "**Include files**" and under the "**Directories**" list add the path to the TypeServer "`\include`" directory. The path to the TypeServer include directory should look something like this (your drive and path prefix may differ):

```
C:\DEV\TYPESERV\INCLUDE
```

Under the "**Show directories for:**" menu, now select "**Library files**" and under the "**Directories**" list add the path to the TypeServer "`\lib`" directory. The path to the TypeServer library directory should look something like this (your drive and path prefix may differ):

```
C:\DEV\TYPESERV\LIB
```

If you have the TypeServer Source Code product, you should also set the path to the TypeServer source code files. (If you do not have the TypeServer source code, you will not have a "`\typeserv\src`" directory and you can skip this setting.) Under the "**Show directories for:**" menu, select "**Source files**" and under the "**Directories**" list add the path to the TypeServer "`\src`" directory. The path to the TypeServer source directory should look something like this (your drive and path prefix may differ):

```
C:\DEV\TYPESERV\SRC
```

Building and Running TypeViewer.cpp for WIN32

The WIN32 example program `TypeViewer.cpp` is located in the `TypeServ\Examples\TypeViewer-Win32` subdirectory. This example shows how to use TypeServer in a 32-bit Windows application. Optionally this sample program can use MetaWINDOW as a graphics library and demonstrates how you can use TypeServer as an external graphics library under WIN32. The `\TypeViewer-Win32` subdirectory contains a `typeviewer.dsw` workspace file with pre-configured options set for either building a "debug" or "release" version of `TypeViewer.exe` for WIN32.

To rebuild `TypeViewer.exe`, simply open the `typeviewer.dsw` workspace file. From the main Visual C menu select "**Built | Clean**", and then select "**Build | Rebuild All**". "**Project | Settings | Compiler | Warning Level**" is set to level 4 (maximum error checking), and the program should compile and link without any errors or warnings. Once the program has finished compiling and linking, you can select "**Build | !Execute**" to run the program.

When TypeViewer begins it will open a blank display window. From the TypeViewer main menu select "**File | Open...**", then choose a TrueType font for TypeServer to render and display.

Building Applications for Other Platforms and Compilers

To build applications for non-Windows platforms and other compilers, TypeServer uses a combination of batch and makefiles to simplify the build process. Batch files (.bat) simply invoke Microsoft Nmake.exe with the appropriate makefile and command line switches to build the desired target executable. For example, to build the `typeview.c` example program for Phar Lap TNT v8 using Microsoft Visual C v6, there are two batch files provided: `typeview_tn8vc6-debug.bat` and `typeview_tn8vc6-release.bat`. These two batch files invoke Microsoft Nmake.exe with a makefile and command line options to build either the "debug" or "release" version of `typeview.exe` for Phar Lap TNT v8 using Microsoft Visual C v6. The batch files themselves are very simple:

```
REM typeview_tn8vc6-debug.bat
nmake -f typeview_tn8vc6.mak CLEAN ALL

REM typeview_tn8vc6-release.bat
nmake -f typeview_tn8vc6.mak CLEAN ALL CFG=RELEASE
```

Editing .mk Makefile `!INCLUDES`

The `typeview_tn8vc6.mak` makefile defines the actual Nmake procedure for compiling and linking the `typeview.exe` program. To define the necessary paths for the compiler and linker, the makefile includes a shared `.mk` file that is located in the `typeserv\include\` directory. Based on the target platform and compiler, the example program makefiles and TypeServer source code makefiles all reference one of the `.mk` files in `typeserv\include` to define the compiler and linker path settings. Before running the batch and makefiles you will need to check and edit the .mk files with the proper path settings for your system. The `.mk` makefile-includes are named in a six character "`ppxccy.mk`" format where "pp" is a two character platform designator, "x" is a platform version number, "cc" is a compiler designator, and "y" is a compiler version number. For example:

```
   typeserver/include/
      et9vc6.mk - makefile include for Phar Lap ETS v9 + MS Visual C v6
      tn7hc3.mk - makefile include for Phar Lap TNT v7 + Metaware High C v3
      tn8hc3.mk - makefile include for Phar Lap TNT v8 + Metaware High C v3
      tn8vc6.mk - makefile include for Phar Lap TNT v8 + MS Visual C v6
```

The .mk makefile-include defines the path settings for your compiler, platform, TypeServer and MetaWINDOW (if used) root directories. Edit these to the correct paths for your system:

```
# TN8VC6.MK - Phar Lap TNT v8, Microsoft Visual C/C++ v6 Makefile Include
# Edit the following paths for the appropriate _ROOT paths for your system
# (NOTE - certain tools such as Phar Lap 386LIB & 386LINK
#          only work with 8 character pathnames)

DIR_CC_ROOT  = \progra~1\micros~2\vc98            # <== C compiler root
DIR_TNT_ROOT = \dev\_tools\pharlap\tnt80          # <== Phar Lap TNT root
DIR_TS_ROOT  = \dev\typeserv                      # <== TypeServer root
DIR_MW_ROOT  = \dev\_tools\pharlap\tnt80\mw386vc  # <== MetaWINDOW root
```

Building and Running with Phar Lap TNT and Microsoft Visual C

Once the root paths in the .mk makefile-includes have been set properly, you can then execute the batch files to compile and rebuild the example programs. For example, from Windows Explorer you can simply double click on the filename "`typeview_tn8vc6-release.bat` to execute the batch file that will invoke Nmake to compile `typeview.c` and rebuild `typeview.exe` for Phar Lap TNT v8 using Microsoft Visual C v6 .

With Phar Lap TNT, graphic applications can only execute directly under DOS (TNT graphic applications cannot run from a Windows DOS-box). To set up for running TNT from DOS we suggest you set up your `config.sys` and `autoexec.bat` files in the following manner:

```
   CONFIG.SYS:
      DOS=HIGH,UMB
      SHELL=C:COMMAND.COM /E:4096 /P
      DEVICE=C:\WINDOWS\HIMEM.SYS
      FILES=40
      STACKS=9,256
      BUFFERS=30
```

```
AUTOEXEC.BAT:
   REM SET BASIC PATH SETTINGS
   REM Basic system path
     SET PATHBASE=C:\WINDOWS;C:\WINDOWS\COMMAND
   REM Base path to Microsoft Visual C++ v6
     SET PATHVC6=C:\PROGRA~1\MICROS~2\VC98
   REM Base path to Phar Lap TNT v8.0
     SET PATHTNT8=C:\dev\_tools\PharLap\TNT80
   REM Base path to Phar Lap TNT v7.0
     SET PATHTNT7=C:\dev\_tools\PharLap\TNT70
   REM Base path to Metaware High C
     SET PATHHC=C:\dev\_tools\HIGHC36
   REM Base path to Metagraphics TypeServer
     SET TYPESERV=C:\dev\typese~1

     SET PATH=%PATHBASE%
```

Your `config.sys` and `autoexec.bat` may also contain different or additional settings. Defining the path prefixes in your `autoexec.bat` as illustrated above makes it easy to set up a batch file to configure your paths when you switch to DOS (a copy of this batch file is provided in `typeserver\doc\tn8vc6.bat`):

```
REM TN8VC6.BAT
REM Set paths for Phar Lap TNT v8 and Microsoft Visual C++ v6

REM Based on PATHBASE, PATHVC and PATHTNT7 settings in autoexec.bat:
  REM Basic system path
    REM SET PATHBASE=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\SYSTEM
  REM Base path to Microsoft Visual C++ v6.0
    REM SET PATHVC6=C:\PROGRA~1\MICROS~2\VC98
  REM Base path to Phar Lap TNT v8.0
    REM SET PATHTNT8=C:\dev\_tools\PharLap\TNT80
  REM Base path to Metagraphics TypeServer
    REM SET TYPESERV=C:\dev\typeserv
SET PATH=%PATHBASE%;%PATHTNT8%\Bin;%PATHVC6%\Bin
SET INCLUDE=%PATHTNT8%\INCLUDE;%PATHVC6%\INCLUDE
SET LIB=%PATHTNT8%\LIB;%PATHVC6%\LIB
SET METAPATH=\dev\_tools\PharLap\Tnt80\mw386vc
cd %TYPESERV%\examples\
```

Once in DOS you can execute `TN8VC6.BAT` to set paths and switch to the `typeserv\examples\` directory. From here you can switch to the example program subdirectory and run the example executable that you just built.

Building and Running with Phar Lap TNT and Metaware C

Similar to building and running with Phar Lap TNT and Microsoft Visual C, you will first need to check and set the root path settings contained in the `tn7hc3.mk` makefile-include. The makefile-include defines the path settings for your compiler, platform, TypeServer and MetaWINDOW (if used) root directories. Edit these to the correct paths for your system:

```
# TN7HC3.MK - Phar Lap TNT v7, Metaware High C v3 Makefile Include
# Edit the following paths for the appropriate _ROOT paths for your system
# (NOTE - certain tools such as Phar Lap 386LIB & 386LINK
#         only work with 8 character pathnames)

DIR_CC_ROOT  = \dev\_tools\HighC36                # <== C compiler root
DIR_TNT_ROOT = \dev\_tools\pharlap\tnt70          # <== Phar Lap TNT root
DIR_TS_ROOT  = \dev\typeserv                      # <== TypeServer root
DIR_MW_ROOT  = \dev\_tools\pharlap\tnt70\mw386hc  # <== MetaWINDOW root
```

Once the root paths in the .mk makefile-includes have been set properly, you can then execute the batch files to compile and rebuild the example programs. For example, from Windows Explorer you can simply double click on the filename "`typeview_tn7hc3-release.bat`" to execute the batch file that will invoke Nmake to compile `typeview.c` and rebuild `typeview.exe` for Phar Lap TNT v7 using Metaware High C v3.

With Phar Lap TNT, graphic applications can only execute directly under DOS (TNT graphic applications cannot run from a Windows DOS-box). To set up for running TNT from DOS we suggest you set up your `config.sys` and `autoexec.bat` files in the following manner:

```
   CONFIG.SYS:
      DOS=HIGH,UMB
      SHELL=C:COMMAND.COM /E:4096 /P
      DEVICE=C:\WINDOWS\HIMEM.SYS
      FILES=40
      STACKS=9,256
      BUFFERS=30

   AUTOEXEC.BAT:
      REM SET BASIC PATH SETTINGS
      REM Basic system path
        SET PATHBASE=C:\WINDOWS;C:\WINDOWS\COMMAND
      REM Base path to Microsoft Visual C++ v6
        SET PATHVC6=C:\PROGRA~1\MICROS~2\VC98
      REM Base path to Phar Lap TNT v8.0
        SET PATHTNT8=C:\dev\_tools\PharLap\TNT80
      REM Base path to Phar Lap TNT v7.0
        SET PATHTNT7=C:\dev\_tools\PharLap\TNT70
      REM Base path to Metaware High C
        SET PATHHC=C:\dev\_tools\HIGHC36
      REM Base path to Metagraphics TypeServer
        SET TYPESERV=C:\dev\typese~1

        SET PATH=%PATHBASE%
```

Your `config.sys` and `autoexec.bat` may also contain different or additional settings. Defining the path prefixes in your `autoexec.bat` as illustrated above makes it easy to set up a batch file to configure your paths when you switch to DOS (a copy of this batch file is provided in `typeserv\doc\tn7hc3.bat`):

```
REM TN7HC3.BAT - Set paths for Phar Lap TNT v7 and Metaware High C v3

REM Based on PATHBASE, PATHHC and PATHTNT7 settings in autoexec.bat:
  REM Basic system path
    REM SET PATHBASE=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\SYSTEM
  REM Base path to Metaware High C
    REM SET PATHHC=C:\APPS\HIGHC
  REM Base path to Phar Lap TNT v7.0
    REM SET PATHTNT=C:\dev\_tools\PharLap\TNT70
  REM Base path to Metagraphics TypeServer
    REM SET TYPESERV=C:\dev\typeserv
SET PATH=%PATHBASE%;%PATHTNT7%\Bin;%PATHHC%\Bin
SET INCLUDE=%PATHTNT7%\INCLUDE;%PATHHC%\INCLUDE
SET LIB=%PATHTNT7%\LIB;%PATHHC%\LIB
SET METAPATH=\dev\_tools\PharLap\Tnt80\mw386hc
cd %TYPESERV%\examples\
```

Once in DOS you can execute `TN7HC3.BAT` to set paths and switch to the `typeserv\examples\` directory. From here you can switch to the example program subdirectory and run the example executable that you just built.

Building and Running with Phar Lap ETS and Microsoft Visual C

If you will be using TypeServer in an application that uses a graphics mode video display (eg. VGA 640x480 256-color), Phar Lap ETS requires that the graphics hardware initialization be performed either within the ETS kernel startup while in real-mode, or as a custom hardware initialization as part of your application when it executes in 32-bit protected-mode. If you are using a standard graphics card or video chipset that incorporates VGA or VESA ROM BIOS, it's usually easiest to modify the ETS `pcati2.c` kernel module to perform graphics mode initialization as part of the ETS kernel startup. The procedure customizing ETS graphics initialization this is outlined in the document `\typeserv\examples\typeview-ets9e\mwfaq0029.htm` provided with your TypeServer software (a copy of this document is also available on-line at http://www.metagraphics.com/metawindow/faq/mwfaq0029.htm).

Before compiling the TypeServer ETS example programs, first check that the root paths in the `\typeserv\include\et9vc6.mk` makefile-include are set properly for your system. TypeServer provides two example programs, `typeview.c` and `mwts-ets.c`, that show how to use TypeServer both by itself, and in combination with Metagraphics MetaWINDOW graphics programming toolkit (to use and run `mwts-ets.c` you will also need a copy of MetaWINDOW). Near the beginning of each C source file there are two `#defines` that need to be set to match the video mode in use:

```
/* IMPORTANT!!! **********************************************************
 * The following setting must match the video mode initialized within the
 * ETS EkCustomBiosInit() function contained in ETS kernal module pcati2.c
 * (also see http://www.metagraphics.com/metawindow/faq/mwfaq029.htm).
 */
#define  VIDEOMODE    VESA800x600X     /* VESA VGA 800x600   256-color */
```

```
/* IMPORTANT!!! *********************************************************
 * The following setting must match the VESA base physical linear frame
 * buffer address for the graphics card in use.  The VESA linear frame
 * buffer address can be determined using the Metagraphics "vesainfo.exe"
 * utility (http://www.metagraphics.com/metawindow/files.htm#VESAINFO).
 */
#define  FRAMEBUFFER 0xE8000000UL   /* example value */
```

Once the root paths in the `et9vc6.mk` makefile-include has been checked and the video settings in the example program C source code files are set, you can use the make and batch files provided to compile and rebuild the example programs. For example, from Windows Explorer you can simply double click on the filename "`typeview_et9vc6-release.bat` to execute the batch file that will invoke Nmake to compile `typeview.c` and rebuild `typeview.exe` for Phar Lap ETS v9 using Microsoft Visual C v6. Similiarly, double clicking on the filename "`mwts-ets_et9vc6-release.bat` executes the batch file that will invoke Nmake to compile `mwts-ets.c` and rebuild `mwts-ets.exe` for Phar Lap ETS v9 using Microsoft Visual C v6.

Chapter 4 - TypeServer Basics

TypeServers, Bitmaps, Fonts and Strikes

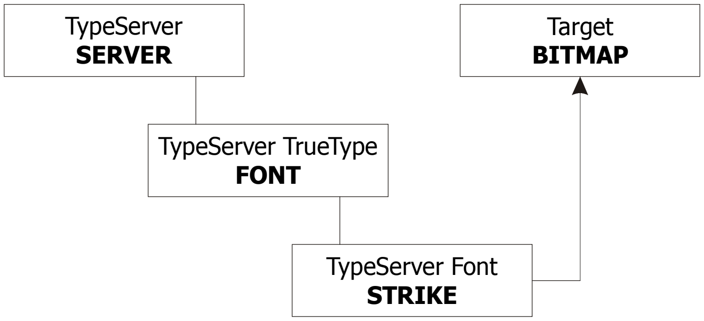
Metagraphics TypeServer is an advanced TrueType rasterizing engine that can be integrated directly with your C or C++ application. TypeServer provides a library of callable functions that process scaleable TrueType fonts and rasterizes characters into pixel images based on attributes set by your program. The TypeServer API uses an object-oriented design that makes it easy to use for conventional, real-time and embedded applications. The TypeServer architecture supports both reentrant and multi-threaded operation, and can be used either by itself or in combination with other tools from Metagraphics or other vendors. At the highest level, there are two basic components:

1. A TypeServer **Server** object - the Metagraphics TypeServer itself
2. A **Bitmap** object - the target bitmap to where characters are rendered

As part of its design, a TypeServer **Server** uses a hierarchy of two additional elements:

3. A TypeServer **Font** object - a scaleable TrueType font, and
4. A TypeServer **Strike** object - a font strikeface that references a given TypeServer **Font** and an output **Bitmap**, and defines rendering attributes for character size, spacing, orientation, path angle, color and other details.

The following chart illustrates the basic hierarchy of the TypeServer components:



In C, TypeServer objects are referenced using a "handles" that are identifiers returned by TypeServer to the application when an instance of that object is created. For C++, TypeServer objects correspond to a class instance of the associated object. The following table lists the C handle data types and C++ classes used with Metagraphics TypeServer.

Object Type	**C Object Handle**[1]	**C++ Class Instance**
TypeServer Server	`TSSERVER`	`tsCServer`
Metagraphics Bitmap	`MGBITMAP`	`mgCBitmap`
TypeServer Font	`TSFONT`	`tsCFont`
TypeServer Strike	`TSSTRIKE`	`tsCStrike`

TypeServer Basics

There are six basic steps in using TypeServer:

1. Create a TypeServer "Server" to perform rendering.
2. Define a "Bitmap" to render to.
3. Select and open a TrueType "Font".
4. Create a font "Strike" and set attributes for size, color, orientation, etc.
5. Draw text (repeat steps 3-5, as necessary).
6. Terminate - close all strikes, close all fonts, close all bitmaps, close TypeServer.

The following sections provide an outline on how each of these steps are performed.

The TypeServer "Server"

Client applications create and initialize a TypeServer **Server** for TrueType rendering by calling either the C `tsServer_Create()` function, or the C++ `tsCServer()` constructor. These functions create and initialize a TypeServer renderer, and return either a handle for a C `TSSERVER` object or C++ `tsCServer` class instance.

```
tsServer_Create(
         MGSYSTEM  *mgSystem,       /* in/out, system-handle */
         TSSERVER  *tsServer );     /* output, server-handle */

// tsCServer constructor
tsCServer::tsCServer(
         mgCSystem *mgSystem );     // in/out, system-instance
```

[1] To minimize potential application and/or system name conflicts, data type, structure and function names that are used across multiple Metagraphics products begin with a generic “MG” prefix, or “mgC” prefix for C++ classes. Names of TypeServer-specific objects are prefixed with “TS”, or “tsC” for C++ classes.

Typically a single application needs only to create and use a single TypeServer. A single TypeServer can render multiple different fonts, in numerous sizes, to several different bitmaps. For example, a single TypeServer could scale and render different fonts for both a display bitmap at one resolution, and a separate printer bitmap at a different resolution.

In special operating system uses, a separate server may be created for each independent task. TypeServer contains no static data variables, so each server instance maintains its own local state information. Using this design, a single TypeServer DLL can support multiple server instances in a reentrant and thread safe manner.

The following steps outline the procedure in C for creating a TypeServer instance (C++ follows the same steps using C++ classes and class methods):

1. Declare a TypeServer server-handle and system-handle, and initialize each of the handles with `NULL` to start.

```
TSSERVER      tsServer=NULL;   /* TypeServer server-handle   */
MGSYSTEM      mgSystem=NULL;   /* Metagraphics system-handle */
```

2. Initialize TypeServer and return both a server-handle and system-handle.

```
/* create and initialize a TypeServer */
result = tsServer_Create( &mgSystem, &tsServer );
if ( FAILED(result) )
   /* perform error handling */
```

Since both the system-handle and server-handle are `NULL` on entry, `tsServer_Create()` creates both a new Metagraphics "system-instance" and TypeServer "server-instance". Reference handles for use when calling other TypeServer functions are returned in the `MGSYSTEM` and `TSSERVER` variables declared by your application.

Destroying Your TypeServer

When finished with type rendering, the client application must call `tsServer_Destroy()` (C) to close the TypeServer and release system resources that it has allocated.

```
result = tsServer_Destroy( &mgSystem, &tsServer );
```

In C++, the `~tsCServer()` destructor will automatically release all associated resources.

Using TypeServer with other Metagraphics Products - `MGSYSTEM`

As you may have noted in the function description, `tsServer_Create()` takes pointers to both a TypeServer "server-handle" and a Metagraphics "system-handle". The Metagraphics "system" instance is a set of common services shared by all Metagraphics products. The system instance, "`MGSYSTEM`" (C) and system class "`mgCSystem`" (C++) provide a centralized implementation for memory management, file I/O, mutexes and other basic system services that are used by all Metagraphics products. If you use multiple Metagraphics products in a single application, the first creation method will create a Metagraphics "system-instance" that is then shared and used by the other Metagraphics product functions.

For example, if you are using TypeServer together with Metagraphics MetaWINDOW v6[2], the opening call to `mwGraphics_Create()` will create both a Metagraphics "system" instance and MetaWINDOW "graphics" instance (remember to first initialize all handles to `NULL`). Later when the system-handle reference is a passed to `tsServer_Create()`, `tsServer_Create()` will create only a new TypeServer "server" instance since the system-handle has been previously defined by `mwGraphics_Create()` and is no longer `NULL`:

```
MGSYSTEM      mgSystem=NULL;   /* Metagraphics system-handle   */
MWGRAPHICS    mwGraphics=NULL; /* MetaWINDOW graphics-handle   */
TSSERVER      tsServer=NULL;   /* TypeServer server-handle     */

 /* create both MGSYSTEM and MWGRAPHICS */
 result = mwGraphics_Create( &mgSystem, &mwGraphics );
 if ( FAILED(result) )
    /* perform error handling */

 /* create TSSERVER only (MGSYSTEM already exists) */
 result = tsServer_Create( &mgSystem, &tsServer );
 if ( FAILED(result) )
    /* perform error handling */
```

When using multiple products as illustrated above, upon completion destructors should be called in their reverse nested order:

```
 /* destroy the TSSERVER instance */
 result = tsServer_Destroy( &mgSystem, &tsServer );

 /* destroy the MWGRAPHICS instance and MGSYSTEM instance */
 result = mwGraphics_Destroy( &mgSystem, &mwGraphics );
```

TypeServer "Bitmaps"

Metagraphics TypeServer processes TrueType font outlines and converts them into pixel images that are written to a raster bitmap. The destination bitmap for type rendering is defined by referencing a C `MGBITMAP` object handle or C++ `mgCBitmap` class instance. TypeServer has broad flexibility and can work with many different bitmap types and formats. Bitmaps for TypeServer use generally fall into three categories:

1. An existing bitmap in memory that has been allocated and created independently of TypeServer, such as an application-created bitmap or a Metagraphics MetaWINDOW bitmap.
2. A video hardware bitmap that is located at a predefined frame-buffer address.

[2] MetaWINDOW v5 does use `mgSystem`, but can still also be used with TypeServer.

3. A TypeServer created bitmap that your application can also directly access.

The C `mgBitmap_Create()` function or C++ `mgCBitmap()` constructor can be used to create a Metagraphics **Bitmap** object for any of these cases.

```
MRESULT mgBitmap_Create(                /* return, result code (0=normal) */
         MGSYSTEM      mgSystem,        /* input,  system-handle          */
         MGBITMAPINFO *bitmapInfo,      /* in/out, pointer to bitmap info */
         MGBITMAP     *bitmapHandle );  /* output, bitmap-handle          */

// mgCBitmap constructor
mgCBitmap::mgCBitmap(
         mgCSystem    *mgcSystem,       // input,  system instance
         MGBITMAPINFO *bitmapInfo );    // in/out, bitmap info
```

In defining a TypeServer bitmap, the client application first fills out a `MGBITMAPINFO` structure that is passed to the C `mgBitmap_Create()` function, or to the C++ `mgCBitmap()` constructor. The bitmap information structure specifies how the bitmap is defined, or if a new bitmap should be created.

TypeServer can work with bitmaps defined in several different ways:

1. Application defined bitmaps.
2. TypeServer created bitmaps.
3. MetaWINDOW defined bitmaps.
4. Windows defined bitmaps.

Application Defined Bitmaps

The C `mgBitmap_Create()` function or C++ `mgCBitmap()` constructor can be used to define either an existing bitmap that has been created elsewhere by your application, or a video frame-buffer bitmap that is located in addressable memory. To define an existing bitmap, the application provides a definition for the size, format and location of the bitmap in memory. This is accomplished by first zeroing a `MGBITMAPINFO` structure, setting parameters in selected fields of the structure, and then calling the C `mgBitmap_Create()` function or C++ `mgCBitmap()` constructor. For C, `mgBitmap_Create()` returns an handle for the bitmap that can be used when calling other TypeServer functions. For C++, the resulting `mgCBitmap` instance is used to reference the bitmap to other C++ TypeServer functions. The following section outlines the steps to define an existing bitmap for TypeServer use.

Defining an Existing Bitmap

To define an existing bitmap for TypeServer use, minimally the following eight `MGBITMAPINFO` fields need to be set before calling the C `mgBitmap_Create()` function, or the C++ `mgCBitmap()` constructor:

```
      /* minimally the following eight (8) MGBITMAPINFO fields
       * must be set to define an existing external bitmap */
INT32         structSize;  /* number of bytes in this structure     */
INT32         pixWidth;    /* pixel width (pixels x, horizontally)  */
INT32         pixHeight;   /* pixel height (pixels y, vertically)   */
int           pixBits;     /* bits per pixel (1,8,16,24 or 32)      */
int           pixPlanes;   /* number of planes per pixel (1 or 4)   */
int           pixResX;     /* pixels per inch, horizontally         */
int           pixResY;     /* pixels per inch, vertically           */
void         *surface;     /* pointer to bitmap surface memory area */
```

For indexed-color bitmaps (256-colors or less), a pointer to an RGB colorTable must also be specified:

```
MGCOLORRGB   *colorTable;  /* pointer to RGB color table array      */
```

Metagraphics TypeServer can also work with special-case bitmaps including: 1) OS/2 "bottom-up" style bitmaps, and 2) special video extended raster line bitmaps. For special-case bitmaps you may define the following additional attributes:

```
int           rowBytes;    /* number of bytes per raster line      */
int           pitch;       /* delta bytes between raster lines     */
```

The following steps outline the procedure in C to define an existing bitmap for TypeServer use (C++ follows the same steps using C++ classes and class methods):

1. Declare a server-handle, system-handle, bitmap-handle, and an MGBITMAPINFO structure.

```
TSSERVER      tsServer=NULL; /* TypeServer server-handle     */
MGSYSTEM      mgSystem=NULL; /* Metagraphics system-handle   */
MGBITMAP      mgBitmap=NULL; /* Metagraphics bitmap-handle   */
MGBITMAPINFO  mgBitmapInfo;  /* bitmapInfo structure         */
```

For indexed-color bitmaps (256-colors and less) also need an RGB color table defined:

```
/* RGB colorTable for 16- and 256-color bitmaps */
static MGCOLORRGB  myColorTable[NUMCOLORS+1];
```

2. Initialize TypeServer and return both a server-handle and system-handle.

```
/* create and initialize a TypeServer */
result = tsServer_Create( &mgSystem, &tsServer );
if ( FAILED(result) )
   /* perform error handling */
```

3. Zero all MGBITMAPINFO fields and initialize the structSize member with the size of the structure. The _InitStruct() macro can be used to zero out a structure and set the first structSize member accordingly. Define the pixel dimensions of the bitmap by setting the .pixWdith and .pixHeight members:

```
_InitStruct( mgBitmapInfo );    /* zero structure and set .structSize */
mgBitmapInfo.pixWidth  = 1024;  /* bitmap pixel width                */
mgBitmapInfo.pixHeight = 768;   /* bitmap pixel height                */
```

4. Define the pixel format by setting the pixBits and pixPlanes values. pixBits is the number of bits per pixel that can either be 1, 8, 16, 24 or 32 for monochrome, 256-color, 65K-color, 16M-color or 16M plus alpha, respectively. pixPlanes is the number of planes per pixel which should be set to 4 for 16-color modes and 1 for all other modes (i.e. for EGA/VGA 16-color bitmaps set pixBits=1 and pixPlanes=4).

```
    mgBitmapInfo.pixBits   = 24;     /* bits per pixel   */
    mgBitmapInfo.pixPlanes = 1;      /* planes per pixel */
```

For 16- or 256-color bitmaps, you also need to define and set the pointer to the RGB color table (for further information on color tables, see the color table information in the expanded description of MGBITMAPINFO).

```
    /* for 16- or 256-color bitmaps, also define and set a colorTable */
    myBitmapInfo.colorTable= &myColorTable;  /* pointer to colorTable */
```

5. TypeServer must also know the number of pixels per inch (or dots per inch, DPI) at which to render. A typical 1024x768 display screen, for example, is normally defined with a resolution of 96 DPI. DPI resolutions for hardcopy devices, such as laser printers can be much higher - typically 300 DPI, 600 DPI or even more. While most devices have the same resolution both horizontally and vertically, Metagraphics TypeServer allows you to define horizontal and vertical resolutions independently, if needed.

```
    mgBitmapInfo.pixResX = 96; /* 96 pixels/inch horizontally */
    mgBitmapInfo.pixResY = 96; /* 96 pixels/inch vertically   */
```

6. Now set the surface pointer to the low memory address of the bitmap pixel area:

```
    /* set the pointer to where the bitmap is located in memory */
    mgBitmapInfo.surface  = myBitmapSurfacePointer;
```

7. For standard bitmaps we can now call the C mgBitmap_Create() function or C++ mgCBitmap() constructor to initialize and return a bitmap handle to use for rendering.

```
    result = mgBitmap_Create( mgSystem, &mgBitmapInfo, &mgBitmap );
    if ( FAILED(result) )
        /* handle error condition */
```

When mgBitmap_Create() returns, myBitmap contains a handle that we can use for type rendering. On return, the MGBITMAPINFO structure passed in by pointer will also be updated with additional information that TypeServer and your application can use to access the bitmap.

8. IMPORTANT - When finished with type rendering, the client application must call mgBitmap_Destroy() (C) to release the bitmap handle and any TypeServer resources that have been allocated for it.

```
    result = mgBitmap_Destroy( &myBitmap );
```

In C++, the `~ttCbitmap()` destructor will automatically release all associated resources. Note that `mgBitmap_Destroy()` will not destroy any resources pre-allocated to the bitmap by your application or other software. Previously allocated resources, if any, will need to be released by the client application or by the software that originally allocated them.

Special-Case Bitmaps

Metagraphics TypeServer can also work with a variety of special-case bitmaps such as: 1) IBM OS/2 inverted "bottom-up" style bitmaps, and 2) extended raster line bitmaps required for some video cards. For special case bitmaps define the following additional attributes:

```
    int            rowBytes;    /* number of bytes in a single raster line */
    int            pitch;       /* displacement bytes between raster lines */
```

rowBytes

`rowBytes` is the number of bytes in each raster line. Normally this is an even multiple of the native CPU word size to insure that individual raster lines are aligned at memory boundaries. (For example, `rowBytes` is normally a multiple of 4 for 32-bit processors, or a multiple of 2 for 16-bit processors). `rowBytes` should always be a positive value.

pitch - Extended Raster Line Cases

`pitch` is the byte displacement between the start of one raster line and the next successive raster line. For most standard bitmaps this is normally the same value as `rowBytes`. For certain hardware video modes, however, the separation between successive raster lines may be *larger* than `rowBytes`. For certain display adapters, for example, 800x600 256-color modes have 800 bytes per raster line, but raster lines are separated at addresses of 1024 (a nice power-of-2 number for hardware simplicity). For this type of case, `rowBytes` would be set to 800, and `pitch` would be set to 1024.

pitch - "Bottom-Up" Style Bitmaps

In a standard bitmap the first y=0 raster line is located at the start of the bitmap surface area with successive raster lines at increasing higher memory address locations. IBM OS/2, however, introduced a special reversed "bottom-up" style bitmap where the first y=0 raster line *is at the end of the bitmap area*, and successive raster lines occur at decreasing memory addresses. (For OS/2 style bitmaps, the low memory address to the bitmap surface points to the *last* raster line instead of the first.) Although used infrequently, Microsoft Windows also supports "bottom-up" style bitmaps.

To define a "bottom-up" style bitmap to TypeServer, simply set the `pitch` value to the negative byte displacement for moving between successive raster lines in decreasing memory address order (most "bottom-up" style bitmaps simply use `"pitch = -rowBytes"`).

TypeServer Created Bitmaps

If you wish, a new local memory bitmap can be automatically allocated for rendering. To create a new bitmap, your application needs to define the size and type of bitmap you wish created. This is accomplished by first clearing a `MGBITMAPINFO` structure to zero, setting parameters in selected fields of the structure, and then calling `mgBitmap_Create()`. Upon return, `mgBitmap_Create()` returns an object handle for the bitmap that can be used when calling other TypeServer functions. On return the MGBITMAPINFO structure is also updated with the access specifics if your application needs to access the bitmap surface directly. The following steps illustrate how to define and create a new bitmap for TypeServer use.

Creating a New Bitmap

If you don't have a specific bitmap that you wish to render to, TypeServer can allocate one in memory for you that your application can also freely access. To have a bitmap dynamically created, minimally the following seven MGBITMAPINFO fields need to be set before calling the C mgBitmap_Create() function, or the C++ mgCBitmap() constructor:

```
           /* minimally the following seven (7) MGBITMAPINFO */
           /* fields must be set to create a new bitmap      */
    INT32         structSize;  /* number of bytes in this structure     */
    INT32         pixWidth;    /* pixel width (pixels x, horizontally)  */
    INT32         pixHeight;   /* pixel height (pixels y, vertically)   */
    int           pixBits;     /* bits per pixel (1,8,16,24 or 32)      */
    int           pixPlanes;   /* number of planes per pixel (1 or 4)   */
    int           pixResX;     /* pixels per inch, horizontally         */
    int           pixResY;     /* pixels per inch, vertically           */
```

The following steps outline the procedure in C for creating a new bitmap for TypeServer use (C++ follows the same steps using C++ classes and class methods):

1. Declare a server-handle, system-handle, bitmap-handle, and an MGBITMAPINFO structure.

```
    TSSERVER      tsServer=NULL; /* TypeServer server-handle       */
    MGSYSTEM      mgSystem=NULL; /* Metagraphics system-handle     */
    MGBITMAP      mgBitmap=NULL; /* Metagraphics bitmap-handle     */
    MGBITMAPINFO  mgBitmapInfo;  /* Metagraphics bitmapInfo struct */
```

For indexed-color bitmaps (256-colors and less) an RGB color table must also be defined:

```
    /* RGB colorTable for 16- and 256-color bitmaps */
    static MGCOLORRGB   myColorTable[NUMCOLORS+1];
```

2. Initialize TypeServer and return both a server-handle and system-handle.

```
    /* create and initialize a TypeServer */
    result = tsServer_Create( &MGSystem, &TSServer );
    if ( FAILED(result) )
       /* perform error handling */
```

3. Zero all MGBITMAPINFO fields and initialize the structSize member with the size of the structure The _InitStruct() macro can be used to zero out a structure and set the first structSize member accordingly. Set the .pixWdith and .pixHeight members for the size of the bitmap desired:

```
    _InitStruct( mgBitmapInfo );    /* zero structure and set .structSize */
    mgBitmapInfo.pixWidth  = 1024;  /* bitmap pixel width                 */
    mgBitmapInfo.pixHeight = 768;   /* bitmap pixel height                */
```

4. Define the pixel format by setting the pixBits and pixPlanes members. pixBits is the number of bits per pixel, and can either be 1, 8, 16, 24 or 32 for monochrome, 256-color, 65K-color, 16M-color or 16M-color+alpha, respectively. pixPlanes is the number of planes per pixel and should be set to 4 for 16-color modes and 1 for all other modes (i.e. for 16-color EGA/VGA modes set pixBits=1 and pixPlanes=4).

```
    mgBitmapInfo.pixBits   = 24;     /* bits per pixel   */
    mgBitmapInfo.pixPlanes = 1;      /* planes per pixel */
```

5. TypeServer must also know the number of pixels per inch (or dots per inch, DPI) at which to render. A typical 1024x768 display screen, for example, is normally defined with a resolution of 96 DPI. DPI resolutions for hardcopy devices, such as laser printers can be much higher - typically 300 DPI, 600 DPI or even more. While most devices have the same resolution both horizontally and vertically, you can define horizontal and vertical resolutions independently if needed.

```
    mgBitmapInfo.pixResX = 96;   /* 96 pixels/inch horizontally */
    mgBitmapInfo.pixResY = 96;   /* 96 pixels/inch vertically   */
```

For indexed-color bitmaps (256-colors and less), you also need to define and set the pointer to the RGB color table (for further information on color tables, see the color table information in the expanded description of MGBITMAPINFO).

```
    /* for 16- or 256-color bitmaps, also define and set a colorTable */
    mgBitmapInfo.colorTable= &myColorTable;  /* pointer to colorTable */
```

6. All ready, call mgBitmap_Create() to create the bitmap and return a reference handle to use.

```
    result = mgBitmap_Create( mgSystem, &mgBitmapInfo, &mgBitmap );
    if ( FAILED(result) )
        /* handle error condition */
```

Since the surface pointer in our MGBITMAPINFO struct is NULL, mgBitmap_Create() will automatically allocate space for the bitmap in system memory, and then set the surface and other access variables accordingly. When mgBitmap_Create() returns, mgBitmap contains a handle by which we can reference the bitmap to other TypeServer functions. The MGBITMAPINFO structure we passed in will also be updated with accessing information about the bitmap:

```
    int          rowBytes;    /* number of bytes in a single raster line   */
    int          pitch;       /* displacement bytes between raster lines   */
    void        *surface;     /* pointer to bitmap surface memory area     */
    void        *rasterY0;    /* pointer to y=0 raster line                */
```

7. IMPORTANT - When finished with type rendering, the client application must call mgBitmap_Destroy() (C) to release the bitmap instance and any TypeServer resources that have been allocated for it.

```
    result = mgBitmap_Destroy( &mgBitmap );
```

In C++, the ~tsCBitmap() destructor will automatically release all associated resources.

MetaWINDOW Defined Bitmaps

TypeServer provides integrated support for working with Metagraphics' MetaWINDOW graphics programming toolkit and bitmaps. To use TypeServer with your MetaWINDOW program you only need to provide a reference pointer to either the MetaWINDOW `grafMap` or `grafPort` data structure that references the bitmap to draw to. The TypeServer example program, `typeview.c`, provides an example showing how to use TypeServer with an application also using MetaWINDOW.

If a MetaWINDOW `grafMap` pointer is provided, TypeServer will render text directly to the bitmap in terms of native bitmap coordinates. If a MetaWINDOW `grafPort` pointer is provided, TypeServer will automatically perform coordinate and positioning transformations defined for the port, and will render text to the port's attached bitmap.

To use TypeServer with MetaWINDOW, the `MGBITMAPINFO structsize` variable, and either the `grafPort` or `grafMap` variables (one or the other) need to be set before calling the C `mgBitmap_Create()` function or the C++ `mgCBitmap()` constructor:

```
    INT32         structSize;  /* number of bytes in this structure     */

    void         *grafPort;    /* pointer to MetaWINDOW grafPort struct */
-or-
    void         *grafMap;     /* pointer to MetaWINDOW grafMap struct  */
```

For 16- and 256-color modes, an RGB color palette will also need to be specified:

```
    MWPALDATA     mwPalData[NUMCOLORS]; /* (NUMCOLORS = 16 or 256) */
```

The TypeServer "`typeview.c`" example program outlines the basic steps in C to define an existing MetaWINDOW bitmap for TypeServer use (C++ follows these same steps using C++ classes and class methods):

1. Declare a MetaWINDOW `grafPort` pointer, `grafMap` pointer, a server-handle, a system-handle, a bitmap-handle and an `MGBITMAPINFO` structure.

```
   MWGRAFPORT  *mwGrafPort;    /* pointer to MW grafPort structure  */
   MWGRAFMAP   *mwGrafMap;     /* pointer to MW grafMap structure   */
   TSSERVER     tsServer=NULL; /* TypeServer server-handle          */
   MGSYSTEM     mgSystem=NULL; /* Metagraphics system-handle        */
   MGBITMAP     mgBitmap=NULL; /* Metagraphics bitmap-handle        */
   MGBITMAPINFO mgBitmapInfo;  /* Metagraphics bitmapInfo structure */
```

16- and 256-color bitmaps will also need RGB color palette data defined:

```
    /* hardware RGB color palette for 16- and 256-color bitmaps */
    MWPALDATA  mwPalData[NUMCOLORS]; /* (NUMCOLORS = 16 or 256) */
```

2. Create a TypeServer and return both its server-handle and system-handle.

```
/* create and initialize a TypeServer */
result = tsServer_Create( &mgSystem, &tsServer );
if ( FAILED(result) )
   /* perform error handling */
```

3. Get pointers to the current MetaWINDOW grafPort and grafMap structures.

```
GetPort( &mwGrafPort );          /* get address of current port   */
mwGrafMap = mwGrafPort->portMap; /* get address of current bitmap */
```

For 256- and 16-color bitmaps, we also need to define and set the pointer to the RGB color palette (for further information on color tables, see the color table information in the expanded description of `MGBITMAPINFO`).

```
/* allocate and initialize a default colorTable for the bitmap */
result = MW_SetColorTable( mgSystem, mwGrafMap );
if ( FAILED(result) )
    /* perform error handling */

/* if this is a VGA hardware bitmap, read the hardware   */
/* color-palette and set the bitmap colorTable to match. */
if ( mwGrafMap->devMode != 0 )
{
    /* read the VGA hardware palette */
    ReadPalette( 0, 0, 255, mwPalData );

    /* set the bitmap colorTable to the hardware color-palette */
    result = MW_SetColors( mwGrafMap, mwPalData, 0, 255 );
    if ( FAILED(result) )
        /* perform error handling */
}
```

4. Zero all `MGBITMAPINFO` fields and initialize the `structSize` member with the size of the structure (the TypeServer `_InitStruct()` function provides a simplified method to zero out a structure and set the first `structSize` member automatically). Set the pointer to the MetaWINDOW bitmap `grafMap` structure. The C `mgBitmap_Create()` function or C++ `mgCBitmap()` constructor can then be called to initialize and return a bitmap instance to use for rendering.

```
/* initialize the MGBITMAPINFO structure */
_InitStruct( mgBitmapInfo );   /* zero structure and set .structSize  */

/* set the .grafMap pointer to reference the MetaWINDOW bitmap */
mgBitmapInfo.mwGrafMap = mwGrafMap;

result = mgBitmap_Create( mgSystem, &mgBitmapInfo, &mgBitmap );
if ( FAILED(result) )
    /* perform error handling */
```

When `mgBitmap_Create()` returns, `mgBitmap` contains an instance handle for use in type rendering. On return, the `MGBITMAPINFO` structure passed in by pointer will also be updated with additional information that TypeServer uses to access the bitmap.

5. IMPORTANT - When finished with type rendering, the client application must call the C `mgBitmap_Destroy()` function to release the bitmap instance and any TypeServer resources that have been allocated for it.

```
    result = mgBitmap_Destroy( &mgBitmap );
```

In C++, the `~mgCBitmap()` destructor will automatically release all associated resources. Note that `mgBitmap_Destroy()` will not destroy any resources pre-allocated to the bitmap by your application or other software. Previously allocated resources, if any, will need to be released by the client application or by the software that originally allocated them.

Windows Defined Bitmaps

TypeServer provides integrated support for working with Microsoft Windows graphic bitmaps. When used under Windows, TypeServer can automatically create a Windows device independent bitmap (DIB) that can be used by TypeServer, and also accessed directly by your application and Windows. The TypeServer Windows example program, `typeviewer.cpp`, provides an example showing how to use TypeServer with Windows compatible bitmaps.

To define a Windows compatible bitmap you only need to initialize five fields in the `MGBITMAPINFO` structure before calling the C `mgBitmap_Create()` function or the C++ `mgCBitmap()` constructor:

The following steps outline the procedure in C to define a Windows compatible bitmap for TypeServer use (C++ follows these same steps using C++ classes and class methods):

1. Declare a server-handle, system-handle, bitmap-handle and an `MGBITMAPINFO` structure.

```
   TSSERVER      tsServer=NULL; /* TypeServer server-handle     */
   MGSYSTEM      mgSystem=NULL; /* Metagraphics system-handle   */
   MGBITMAP      mgBitmap=NULL; /* Metagraphics bitmap-handle   */
   MGBITMAPINFO  mgBitmapInfo;  /* bitmapInfo structure         */
```

2. Create a TypeServer and return both its server-handle and system-handle.

```
    /* create and initialize a TypeServer */
    result = tsServer_Create( &MGSystem, &TSServer );
    if ( FAILED(result) )
       /* perform error handling */
```

3. Zero all `MGBITMAPINFO` fields and initialize the `structSize` member with the size of the structure (the TypeServer `_InitStruct()` function provides a simplified method to zero out a structure and set the first `structSize` member automatically). Initialize the `.pixWidth`, `.pixHeight`, `.pixBits`, `.pixResX` and `.pixResY` fields to define size and pixel format of the bitmap. The C `mgBitmap_Create()` function or C++ `mgCBitmap()` constructor can then be called to initialize and return a bitmap instance to use for rendering.

```
    /* initialize the MGBITMAPINFO structure and define */
    /* the basic size and format of the bitmap we want. */
    _InitStruct( mgBitmapInfo );   /* zero structure and set .structSize  */
    mgBitmapInfo.pixWidth  = DisplayPixWidth;  /* bitmap pixel width */
    mgBitmapInfo.pixHeight = DisplayPixHeight; /* bitmap pixel height*/
    mgBitmapInfo.pixBits   = 24;               /* bits per pixel     */
    mgBitmapInfo.pixResX   = logPixelsX;       /* pixels per inch, X */
    mgBitmapInfo.pixResY   = logPixelsY;       /* pixels per inch, Y */

    /* create the bitmap and return its handle */
    result = mgBitmap_Create( mgSystem, &mgBitmapInfo, &mgBitmap );
    if ( FAILED(result) )
        /* handle error condition */
```

When `mgBitmap_Create()` returns, `mgBitmap` contains an instance handle that can be used for type rendering.   On return the `MGBITMAPINFO` structure passed in by pointer will also be updated with additional information that TypeServer uses to access the bitmap, plus the Windows device context handle (`HDC hdc`), Windows bitmap handle (`HBITMAP hbitmap`) and pointer to the Windows DIB bitmap info structure (`BITMAPINFO *dibInfo`).

4. IMPORTANT - When finished with type rendering, the client application must call the C `mgBitmap_Destroy()` function to release the bitmap instance and any TypeServer resources that have been allocated for it.

```
    result = mgBitmap_Destroy( &myBitmap );
```

In C++, the `~mgCBitmap()` destructor will automatically release all associated resources. Note that `mgBitmap_Destroy()` will not destroy any resources pre-allocated to the bitmap by your application or other software. Previously allocated resources, if any, will need to be released by the client application or by the software that originally allocated them.

TypeServer "Fonts"

TypeServer processes data from a TrueType "**Font**" that is stored either in memory or in an external disk file. In C, a font is selected by calling either `tsFont_OpenMemory()` or `tsFont_OpenFile()` depending if the font has been preloaded into memory or ROM, or is in a file stored on disk. These functions open a specified TrueType font and return a handle of the resulting `TSFONT` object for reference by the client application. In C++, the `tsCFont()` constructor creates an appropriate TypeServer `tsCFont` instance based on the parameter type specified.

```
tsFont_OpenFile(                 /* C, Open a TrueType font file          */
         TSSERVER      tsServer,       /* input,  server-handle           */
   const TCHAR        *filePathName,   /* input,  font path and filename  */
         LONG          fileOffset,     /* input,  font offset within file */
         int           fontNumber,     /* input,  font number within file */
         TSFONT       *tsFont );       /* output, font instance handle    */
```

```
tsFont_OpenMemory(                   /* Open a TrueType font in memory     */
         TSSERVER      tsServer,       /* input,  server-handle              */
         void         *fontMemory,     /* input,  font memory address        */
         LONG          fontSize,       /* input,  font memory size           */
         int           fontNumber,     /* input,  font number within file */
         TSFONT       *tsFont );       /* output, font instance handle       */

// tsCFont font file constructor
tsCFont::TSCFont(                    // Open a TrueType font file
   const tsCServer    *tscServer,      // input,  server instance
   const TCHAR        *filePathName,   // input,  font path and filename
         LONG          fileOffset,     // input,  font offset within file
         int           fontNumber );   // input,  font number within file

// tsCFont memory font constructor
tsCFont::tsCFont(                    // Open a TrueType font in memory
   const tsCServer    *tscServer,      // input,  server instance
         void         *fontMemory,     // input,  font memory address
         LONG          fontSize,       // input,  font memory size
         int           fontNumber );   // input,  font number within file
```

Once a TrueType font has been opened additional information about it can be accessed from its `TSFONTINFO` structure. A pointer to the font info structure can be retrieved using the C `tsFont_GetInfoPtr()` function, or C++ `tsCFont::GetInfoPtr()` method.

```
TSFONTINFO*  tsFont_GetInfoPtr(     /* get font information pointer       */
         TSFONT        tsFont,         /* input,  font-handle                */
         int           fontInfoSize ); /* input,  TSFONTINFO struct size  */

TSFONTINFO*  tsCFont::GetInfoPtr(   // get font information pointer
         int           fontInfoSize ); // input,  TSFONTINFO struct size
```

After finished with a font, your program should call `tsFont_Destroy()` (C) or the `~tsCFont()` destructor (C++) to close the font and release system resources that have been allocated for it:

```
tsFont_Destroy(              /* C, close a TrueType font   */
       TSFONT     *tsFont );       /* in/out, font-handle  */

// tsCFont destructor
tsCFont::~tsCFont(  );       // C++, close a TrueType font
```

TypeServer can have multiple fonts open at one time for rendering text that includes mixed fonts. For example, a given server could have Arial normal, Arial-Bold, Times-Roman and Symbol TrueType fonts open for processing at the same time. Any number of fonts could theoretically be opened (given enough memory) if an application requires mixed rendering of multiple fonts.

TrueType fonts are scaleable outlines that can be rendered at a variety of sizes and resolutions. To convert TrueType outlines into pixel images, the application program must set parameters defining the rendering attributes for character size, color, spacing, transfer mode, target destination bitmap and more. These rendering attributes are defined by creating a font **Strike.** A font Strike is the actual object used to perform rendering.

TypeServer Font "Strikes"

As noted previously, TrueType fonts contain scaleable outline definitions for the characters represented within the font. TypeServer scales and rasterizes these definitions for rendering at various sizes and orientations.

A font **Strike** defines the rendering attributes used by TypeServer to produce bitmap text and character images from TrueType fonts. These attributes may be set by the client application to define the size and appearance of the typeface to be drawn. Strike functions can be divided into four groups:

- Strike creation and deletion.
- Strike state information
- Strike attribute control
- Strike rendering

Just as a single TypeServer can have multiple fonts open at one time, a single font can have multiple strikes defined at one time. For example, once a TypeServer is created and an Arial font is opened, several different strikes could be created for rendering text for different size characters and/or devices. A single Arial font could have multiple strikes defined for rendering characters at 10-point @ 96DPI, 14-point @ 96DPI, 10-point @ 300DPI and 16-point @ 300DPI. The font strike is the actual object that your application uses for drawing text.

Creating a Strike

The C `tsStrike_Create()` function, or C++ `tsCStrike::tsCStrike()` constructor is used to create and initialize a font strike. These functions return either a handle for a C `TSSTRIKE` object, or a C++ `tsCStrike` class instance.

```
tsStrike_Create(                    /* create a font strike          */
         TSFONT        font,           /* input,  font-handle        */
         MGBITMAP      bitmap,         /* input,  bitmap-handle      */
         TSSTRIKE     *strike );       /* output, strike-handle      */

tsCStrike::tsCStrike(               // font strike contructor
         tsCFont      *font,           // input,  font instance
         mgCBitmap    *bitmap );       // input,  bitmap instance
```

When a font strike is created it is initialized with a default set of rendering attributes. Rendering attributes can be changed by the application at any time by calling the various strike attribute functions (see "Setting Strike Attributes", below).

Strike Information

Once a font strike has been constructed, information about it can be accessed from its `TSSTRIKEINFO` data structure. A pointer to this structure can be retrieved using the `GetInfoPtr()` function.

```
TSSTRIKEINFO*  tsStrike_GetInfoPtr(  /* Get pointer to strike information */
     TSSTRIKE        strike,          /* input,  strike instance handle  */
     int             strikeInfoSize ); /* input,  TSSTRIKEINFO struct size*/

TSSTRIKEINFO*  tsCStrike::GetInfoPtr(// Get pointer to strike information
     int             strikeInfoSize ); // input,  TSSTRIKEINFO struct size
```

Setting Strike Attributes

When a font strike is created it is initialized with a default set of rendering attributes. The application program can change these attributes at any time by calling the associated function to set a new value. A font strike defines the following rendering attributes:

Strike Attribute	Function(s)	Default
Type Size	`tsStrike_SetTypeSize()`	10.0 points
Type Alignment	`tsStrike_SetAlign()`	`tsLEFT_BASELINE`
Background Color	`tsStrike_SetColors()`	White
Character Color	`tsStrike_SetColors()`	Black
Character-Extra	`tsStrike_SetJustify()`	0.0
Word-Extra	`tsStrike_SetJustify()`	0.0
Line Spacing	`tsStrike_SetLineSpacing()`	12.0 points
Transfer Mode	`tsStrike_SetRasterOp()`	`tsCOPY`
Edge Smoothing	`tsStrike_SetSmoothing()`	`tsSMOOTH16`

Type Size

Type size defines the size at which characters in the font are drawn. The size is based on an "EM-square" character, which roughly encloses the capital letter "M". Based on the selection of the `TSUNITS` enumeration (either `tsPIXELS` or `tsPOINTS`), the type size can be specified either in terms of pixels or typographic-points (1/72nds of an inch). In both cases the type size is defined using the fixed-point `FIXDOT`[3] data type that allows fractional values. The default type size when a strike is first created is 10.0 points (EM-square size: 10 points high by 10 points wide).

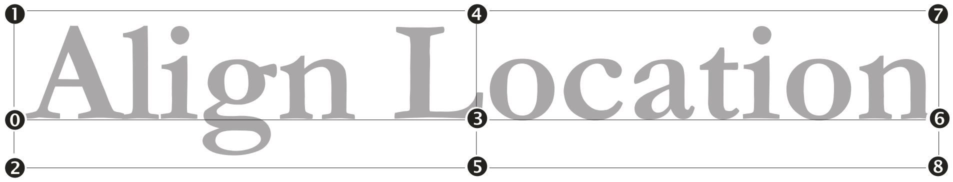
[3] "**`FIXDOT`**" **Fixed-Point Data Type** - Many strike settings are defined using the `FIXDOT` fixed-point data type. `FIXDOT` is a conditional data type dependent upon if the keyword "`FIXMATH26`" is defined or not. If `FIXMATH26` is *not* defined (typically for 16-bit processors where integers are 2 bytes in size), `FIXDOT` is defined equal to the `F12DOT4` data type that incorporates a 12-bit signed integer and a 4-bit unsigned fraction (16-bits total). `F12DOT4` provides fractional accuracy to 1/16th of a unit. For platforms where `FIXMATH26` is defined (typically for 32- and 64-bit processors), `FIXDOT` is defined equal to the `F26DOT6` data type which is a 26-bit signed integer and a 6-bit unsigned fraction (32-bits total). `F26DOT6` provides fractional accuracy to 1/64th of a unit. `IntToFix()` and `FixToInt()` are conditional macros that can be used to convert integers to and from `FIXDOT` format. For additional information on fixed-point data types please see Appendix A of the **Metagraphics C/C++ Programming Guidelines** manual (this document may be downloaded on-line at http://www.metagraphics.com/pubs/MetagraphicsCodingGuide.pdf).

```
    tsStrike_SetTypeSize( TSSTRIKE strike, FIXDOT  charHeight,     /* C */
                                           TSUNITS units );

    tsCStrike::SetTypeSize( FIXDOT charHeight, TSUNITS units );    // C++
```

Type Alignment

Type alignment specifies the location for positioning when a character or string is drawn. The TSALIGN enumeration allows you can specify one of nine possible alignment positions. The default alignment position when a strike is first created is tsLEFT_BASELINE.



```
     0-tsLEFT_BASELINE      3-tsCENTER_BASELINE      6-tsRIGHT_BASELINE
     1-tsLEFT_TOP           4-tsCENTER_TOP           7-tsRIGHT_TOP
     2-tsLEFT_BOTTOM        5-tsCENTER_BOTTOM        8-tsRIGHT_BOTTOM

    tsStrike_SetAlign( TSSTRIKE strike, TSALIGN alignLocation );  /* C */

    tsCStrike::SetAlign( TSALIGN alignLocation );                 // C++
```

BackColor / CharColor

The backColor and charColor attributes define the background color and character color of the characters when they are rendered. When a strike is first created, the background color is set to white and the character color is set to black. TypeServer provides two functions that allow you to specify the character and background colors either in terms of specific pixel values, or by generic RGB colors.

```
tsStrike_SetColors(    TSSTRIKE strike, MGCOLORPIX pixChar,         /* C */
                                        MGCOLORPIX pixBack );

tsStrike_SetColorsRGB( TSSTRIKE strike, MGCOLORRGB rgbChar,         /* C */
                                        MGCOLORRGB rgbBack );

tsCStrike::SetColors(    MGCOLORPIX pixChar, MGCOLORPIX pixBack );  // C++

tsCStrike::SetColorsRGB( MGCOLORRGB rgbChar, MGCOLORRGB rgbBack );  // C++
```

Character-Extra / Word-Extra

SetJustify() allows you to widen or tighten the spacing between characters or words. charExtra and wordExtra are typically defined when creating lines of text with left and right justified margins.

charExtra defines an amount of additional spacing to widen or tighten the distance between each character. The charExtra value is added to the normal character width, with positive values expanding the character spacing and negative values tightening the character spacing. When a strike is first created the default charExtra setting is 0.0.

wordExtra is added *only* to the widths of blank "space" characters (which is different from the charExtra attribute that is added to the widths of *all* characters, including spaces). Positive values expand the word spacing and negative values tighten the word spacing. When a strike is first created the default wordExtra setting is 0.0.

charExtra=0.0 pt; wordExtra=0.0 pt
charExtra=+1.0 pt; wordExtra=0.0 pt
charExtra= -1.0 pt; wordExtra= 0.0 pt
charExtra=0.0 pt; wordExtra=+2.0 pt
charExtra=0.0 pt; wordExtra=-1.0 pt
charExtra=+1.0 pt; wordExtra=+1.0 pt
charExtra=-1.0 pt; wordExtra=-1.0 pt

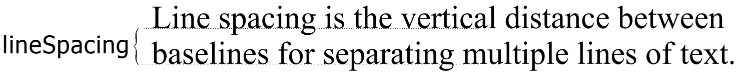
Based on the selection of the TSUNITS enumeration (tsPIXELS or tsPOINTS), charExtra and wordExtra values can be specified either in terms of pixels or typographic-points (1/72^{nds} of an inch). When a strike is first created the character-extra and word-extra attributes are initialized to a default value of zero (0.0).

```
tsStrike_SetJustify( TSSTRIKE strike,                      /* C */
                     FIXDOT   charExtra, FIXDOT wordExtra,
                     TSUNITS  units );

tsCStrike::SetJustify( FIXDOT  charExtra, FIXDOT wordExtra,
                       TSUNITS units );                    // C++
```

Line Spacing

Line spacing defines the distance for positioning consecutive lines of text vertically (specifically the distance between each baseline). Based on the selection of the TSUNITS enumeration, either tsPIXELS or tsPOINTS, line spacing can be specified either in terms of pixels or typographic-points. When a strike is first created line spacing is initialized to a default value of twelve points (12.0 pt).


lineSpacing
Line spacing is the vertical distance between
baselines for separating multiple lines of text.

```
tsStrike_SetLineSpacing( TSSTRIKE strike,                      /* C */
                         FIXDOT lineSpacing, TSUNITS  units );

tsCStrike::SetLineSpacing( FIXDOT lineSpacing, TSUNITS units ); // C++
```

Transfer Mode

The transfer mode defines how the rendered character image is combined into the destination bitmap. TypeServer supports are four "standard" rasterOps, and four "transparent" rasterOps that are defined in the TSRASTEROP enumeration. For "transparent" rasterOps, only non-background colors of the character are drawn.

```
tsStrike_SetRasterOp( TSSTRIKE strike, TSRASTEROP rasterOp ); /* C */

tsCStrike::SetRasterOp( TSRASTEROP rasterOp );                // C++
    typedef enum TSRASTEROP_
    {   tsCOPY,              /* replace             */
        tsMERGE,             /* OR                  */
        tsERASE,             /* AND                 */
        tsINVERT,            /* XOR                 */
        tsTRANSPARENT_COPY,  /* transparent replace */
        tsTRANSPARENT_MERGE, /* transparent OR      */
        tsTRANSPARENT_ERASE, /* transparent AND     */
        tsTRANSPARENT_INVERT /* transparent XOR     */
    } TSRASTEROP;
```

Edge Smoothing

Metagraphics TypeServer incorporates an advanced type-rendering engine to produce precise, high quality characters with smooth anti-aliased edges. The amount of smoothing applied is controlled by the strike `smooth` attribute. The TypeServer `TSSMOOTH` enumeration specifies one of four smoothing options: `tsSMOOTH0` (no smoothing), `tsSMOOTH4` (4-level anti-aliasing), `tsSMOOTH16` (16-level anti-aliasing, default), or `tsSMOOTH256` (256-level anti-aliasing).

Smoothing levels define the number of incremental colors TypeServer can use when rendering characters to the output bitmap. For rendering text in black and white, for example, this is the number of intermediate grayscale levels that can be used for anti-aliasing the character edges. The default smoothing when a strike is first created is `tsSMOOTH16` (16-level anti-aliasing).

TypeServer will automatically reduce the smoothing level if it exceeds the number of colors in the destination bitmap. When rendering to a monochrome bitmap, for example, no anti-aliasing can be performed and rendering will automatically be processed as `tsSMOOTH0`. For 16- and 256-color modes that require color tables, TypeServer may also reduce the smoothing if the color table for the destination bitmap does not contain colors that are close enough to the aliasing colors needed.

Note that edge smoothing isn't free - the higher the smoothing level, the higher the processing overhead. Usually `tsSMOOTH16` is more than adequate (and it certainly is much better than what Windows has offered for many years). You can try `tsSMOOTH256` to see if you notice a difference (as with many things in graphics design, "beauty is in the eye of the beholder").

```
tsStrike_SetSmoothing( TSSTRIKE strike, TSSMOOTH smoothLevel ); /* C */

tsCStrike::SetSmoothing( TSSMOOTH smoothLevel );                // C++
```

Determining Type Dimensions

Based on the current strike attributes, the pixel dimensions of a given character or string can be determined by calling a strike "get extent" function. The "get extent" functions compute the pixel height and width of a specified character or string. TypeServer provides separate character and string functions for each of three basic character types:

C Functions

```
/* ASCII/Unicode functions (TCHAR, 8 or 16-bit dependent on "_UNICODE")*/
    tsStrike_GetCharExtent()      /* get the dimensions of a character */
    tsStrike_GetStringExtent()    /* get the dimensions of a string    */

/* ASCII-specific functions (CHAR, 8-bit characters) */
    tsStrike_GetCharExtentA()     /* get the dimensions of a character */
    tsStrike_GetStringExtentA()   /* get the dimensions of a string    */

/* Unicode-specific functions (WCHAR, 16-bit characters) */
    tsStrike_GetCharExtentW()     /* get the dimensions of a character */
    tsStrike_GetStringExtentW()   /* get the dimensions of a string    */
```

C++ Methods

```
// ASCII/Unicode functions (TCHAR, 8 or 16-bit dependent on "_UNICODE")
    tsCStrike::GetCharExtent()    // get the dimensions of a character
    tsCStrike::GetStringExtent()  // get the dimensions of a string

// ASCII-specific functions (CHAR, 8-bit characters)
    tsCStrike::GetCharExtentA()   // get the dimensions of a character
    tsCStrike::GetStringExtentA() // get the dimensions of a string
// Unicode-specific functions (WCHAR, 16-bit characters)

    tsCStrike::GetCharExtentW()   // get the dimensions of a character
    tsCStrike::GetStringExtentW() // get the dimensions of a string
```

The C function prototypes for the TCHAR version of the extent functions is shown below for illustration (see Chapter 6, "**Library Reference**", for additional details on all of the functions):

```
/* get the dimensions of a ASCII/Unicode (TCHAR) character */
MRESULT tsStrike_GetCharExtent(
      TSSTRIKE     strike,        /* input,  strike-handle            */
  const TCHAR        character,     /* input,  character                */
      FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of a ASCII/Unicode (TCHAR) string */
MRESULT tsStrike_GetStringExtent(
      TSSTRIKE     strike,        /* input,  strike-handle            */
  const TCHAR       *string,        /* input,  character string         */
      FIXSIZE     *extent );      /* output, character width & height  */
```

Type Rendering

Once the TypeServer has been initialized and a font opened with strike attributes defined, you can begin rendering characters and text strings to a bitmap. TypeServer provides separate character and string functions for each of three basic character types:

C Functions

```
/* ASCII/Unicode functions (TCHAR, 8 or 16-bit dependent on "_UNICODE" */
    tsStrike_DrawChar()          /* render a single character         */
    tsStrike_DrawString()        /* render a string of characters     */
```

```
/* ASCII-specific functions (CHAR, 8-bit characters) */*/
    tsStrike_DrawCharA()          /* render a single character         */
    tsStrike_DrawStringA()        /* render a string of characters     */

/* Unicode-specific functions (WCHAR, 16-bit characters) */*/
    tsStrike_DrawCharW()          /* render a single character         */
    tsStrike_DrawStringW()        /* render a string of characters     */
```

C++ Methods

```
// ASCII/Unicode functions (TCHAR, 8 or 16-bit dependent on "_UNICODE")
    tsCStrike::DrawChar()         // render a single character
    tsCStrike::DrawString()       // render a string of characters

// ASCII-specific functions (CHAR, 8-bit characters)
    tsCStrike::DrawCharA()        // render a single character
    tsCStrike::DrawStringA()      // render a string of characters

// Unicode-specific functions (WCHAR, 16-bit characters)
    tsCStrike::DrawCharW()        // render a single character
    tsCStrike::DrawStringW()      // render a string of characters
```

The C function prototypes for the `TCHAR` version of the extent functions is shown below for illustration (see Chapter 6, "Library Reference", for additional details on all of the functions):

```
/* draw a ASCII/Unicode (TCHAR) character */
MRESULT tsStrike_DrawChar(
        TSSTRIKE    strike,         /* input,  strike-handle           */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate */
  const TCHAR       character );    /* input,  character to draw       */

/* draw a ASCII/Unicode (TCHAR) string */
MRESULT tsStrike_DrawString(
        TSSTRIKE     strike,        /* input,  strike-handle           */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate */
  const TCHAR       *string );      /* input,  string to draw          */
```

The *"How To Write ASCII & UNICODE Portable Code"* tutorial in **Appendix B** provides a discussion on how you can write code that will work compatibility in either 8-bit ASCII or 16-bit Unicode environments.

Closing a Strike

After finished using a font strike, your program should call `tsStrike_Destroy()` (C) or the `~tsCStrike()` destructor (C++) to close the strike and release system resources that have been allocated for it.

```
tsStrike_Destroy(                  /* C, close a font strike    */
         TSSTRIKE    *tsStrike );       /* in/out, font-handle  */

// tsCStrike destructor
tsCStrike::~tsCStrike(  );         // C++, close a font strike
```

Chapter 5 - Basic Data Types, Structures & Macros

Basic Data Types

The basic data types noted below are documented in the **Metagraphics C/C++ Programming Guidelines** manual. **Metagraphics C/C++ Programming Guidelines,** Chapter 2 *"Basic Data Types"* and Appendix A *"Computational Data Types"* provide additional detail for these data types.

Data Type	Win32 Type	Description
BYTE	unsigned char	8-bit unsigned integer
CHAR	signed char	8-bit signed integer, and/or ASCII character
WCHAR	unsigned short	16-bit Unicode character
⬄**TCHAR**	CHAR or WCHAR	8- or 16-bit character, depending if "_UNICODE" is defined
INT16	short	16-bit signed integer
INT32	long	32-bit signed integer
⬄**INT**	int	16- or 32-bit (or 64-bit) native signed integer
UINT16	unsigned short	16-bit unsigned integer
UINT32	unsigned long	32-bit unsigned integer
⬄**UINT**	unsigned int	16- or 32-bit (or 64-bit) native unsigned integer
⬄**LONG**	long	32-bit (or 64-bit) signed long integer
⬄**ULONG**	unsigned long	32-bit (or 64-bit) unsigned long integer
F12DOT4	signed short	Signed 12.4 fixed-point fractional integer (16-bits)
F26DOT6	signed long	Signed 26.6 fixed-point fractional integer (32-bits)
⬄**FIXDOT**	int	F26DOT6 or F12DOT4, depending if "FIXMATH26" is defined

⬄ Indicates machine-dependent variable size data type.

TypeServer uses only integer arithmetic and requires no floating-point math. The FIXDOT data type is used to represent sizes, distances, lengths and angular measurements with fractional precision. FIXDOT is conditionally defined equal to either F12DOT4 or F26DOT6 depending if the keyword "FIXMATH26" is defined or not. Normally FIXMATH26 is undefined for 16-bit integer platforms, and defined for 32-bit (or larger) platforms. (With the TypeServer source code it is possible to use F26DOT6 on 16-bit platforms, but performance considerations should be taken into account.)

The F12DOT4 and F26DOT6 data types provide fractional accuracy to 1/16th or 1/64th of a unit, respectively. For example, a pixel position in F26DOT6 format provides accuracy to 1/64th of a pixel. Again remember that FIXDOT is a conditional data type that is defined equal to F12DOT4 or F26DOT6, normally depending upon the size of 'int' as either 16- or 32-(or greater) bits, respectively.

The "Math and Utilities" section of this chapter documents utility functions and macros that TypeServer provides to convert common data types to and from FIXDOT (and F26DOT6, F16DOT16) formats.

TypeServer Structures and Data Types

The following sections provide an alphabetical summary of the data types, data structures, enumerations, macros and utility functions used by Metagraphics TypeServer.

FIXDOT Data Types

To provide fractional accuracy when specifying measurements and sizes, TypeServer uses a special FIXDOT data type for fixed-point values. The FIXDOT data type can actually be one of two fixed-point data formats, depending if the keyword "FIXMATH26" is defined or not.

FIXDOT Equal to F26DOT6

FIXMATH26 is normally defined when operating on 32-bit and larger CPU's. When FIXMATH26 is defined, the FIXDOT data type is equated to the 32-bit F26DOT6 data type that provides a 26-bit integer mantissa and a 6-bit fixed-point fraction. The F26DOT6 fixed-point data type provides fractional accuracy to 1/64th of a unit.

FIXDOT Equal to F12DOT4

When FIXMATH26 is not defined (typically for CPU's with an int size of 16-bits), the FIXDOT data type is equated to the 16-bit F12DOT4 fixed-point data type that uses a 12-bit integer mantissa and a 4-bit fixed-point fraction. The F12DOT4 fixed-point data type provides fractional accuracy to 1/16th of a unit. Note that when using F12DOT4 fixed-point values, care must be taken to insure that values do not exceed the +/-2048 mantissa limit (for angular measurements and type sizes in points or pixels this is usually not a problem).

By recompiling the TypeServer source code, it is possible to define FIXMATH26 for use even on 16-bit systems. Since F26DOT6 is 32-bits, this adds some performance overhead on 16-bit platforms, but provides higher accuracy and allows the use of larger fixed-point values. For additional information on fixed-point data types and functions, please see the **Metagraphics C/C++ Programming Guidelines** manual, Appendix A *"Computational Data Types"*.

Type Definitions

mgtypes.h

```
typedef signed short  F12DOT4; /* signed fixed-point, 12.4 (16-bits) */

typedef signed long   F26DOT6; /* signed fixed-point, 26.6 (32-bits) */

#ifdef   FIXMATH26
typedef signed long   FIXDOT;  /* FIXDOT is F26.6 (32-bits) */
#else /*!FIXMATH26*/
typedef signed short  FIXDOT;  /* FIXDOT is F12.4 (16-bits) */
#endif
```

FIXPOINT struct

Structure Definition

mgtypes.h

```
/* fixed-point coordinate point structure (F26.6 or F12.4) */
typedef struct FIXPOINT_
{
    FIXDOT   x;    /* horizontal x */
    FIXDOT   y;    /* vertical y   */
} FIXPOINT;
```

FIXRECT struct

Structure Definition

mgtypes.h

```
/* fixed-point rectangle data structure (F26.6 or F12.4) */
typedef struct FIXRECT_
{
    FIXDOT    xMin;   /* left   */
    FIXDOT    yMin;   /* top    */
    FIXDOT    xMax;   /* right  */
    FIXDOT    yMax;   /* bottom */
} FIXRECT;
```

FIXSIZE struct

Structure Definition

mgtypes.h

```
/* fixed-point width/height size structure (F26.6 or F12.4 ) */
typedef struct FIXSIZE_
{
    FIXDOT   width;    /* horizontal x */
    FIXDOT   height;   /* vertical y   */
} FIXSIZE;
```

MGBITMAPINFO struct

Structure Definition

mgbitmap.h

```
typedef struct _MGBITMAPINFO
{
  INT32         structSize;  /* number of bytes in this struct            */
  UINT32        objectType;  /* fourCC object type identifier (='MGBI')   */
  INT32         pixWidth;    /* pixel width (pixels x, horizontally)      */
  INT32         pixHeight;   /* pixel height (pixels y, vertically)       */
  int           pixBits;     /* bits per pixel (1,2,4,8,15,16,24 or 32)   */
  int           pixBytes;    /* bytes per pixel (pixBits/8=0,1,2,3 or 4)  */
  int           pixPlanes;   /* number of planes per pixel (1 or 4)       */
  int           pixResX;     /* pixels per inch, horizontally             */
  int           pixResY;     /* pixels per inch, vertically               */
  int           rowBytes;    /* number of bytes in a single raster line   */
  int           pitch;       /* byte displacement between raster lines    */
  RECT          limitsRect;  /* limits rectangle (0,0,pixWidth,pixHeight) */
  int           rasterOp;    /* current rasterOp blit transfer mode       */
  void         *surface;     /* pointer to bitmap surface memory area     */
  void         *topY;        /* pointer to y=0 raster line                */
  MGCOLORRGB   *colorTable;  /* pointer to indexed RGB colorTable         */
  MGCOLORPIX    xparColor,   /* transparent color pixel value             */
  BOOL          transActive; /* TRUE if color translation is active       */
  BYTE         *transTable;  /* pointer to color translate table          */
  HDC           hdc;         /* handle to optional Windows DC             */
  HBITMAP       hbitmap;     /* handle to optional Windows BITMAP         */
  BITAMPINFO   *dibInfo;     /* ptr to optional Windows DIB BITMAPINFO    */
  MWGRAFPORT   *grafPort;    /* ptr to optional MetaWINDOW grafPort struc */
  MWGRAFMAP    *grafMap;     /* ptr to optional MetaWINDOW grafMap struct */
  BYTE        **rowTable[4]; /* ptr to raster rowTable address array(s)   */
  FIXDOT        fixResX;     /* pixels per inch, horizontal (fixed-point) */
  FIXDOT        fixResY;     /* pixels per inch, vertical (fixed-point)   */
} MGBITMAPINFO;
```

The MGBITMAPINFO is used to define a graphics bitmap that TypeServer renders to. TypeServer can work with many different bitmap types and formats, including:

1. External bitmaps that have been created independently of TypeServer, such as a custom application-created bitmap or a bitmap created by another graphics library.
2. Video hardware bitmaps which are located at a predefined address in video memory.
3. TypeServer created bitmaps that your application may also directly access.

The procedure for defining various bitmap types is outlined in the "**TypeServer Bitmaps**" description within Chapter 4 of this manual.

Structure Variables

`INT32       structSize`

Size of this `MGBITMAPINFO` structure, in bytes.

`UINT32      objectType`

A FourCC '`MGBI`' character code identifying an `MGBITMAPINFO` structure.

`INT        pixWidth`

Bitmap pixel width (horizontal, X).

`INT        pixHeight`

Bitmap pixel height (vertical, Y).

`INT        pixBits`

Number of bits per pixel: 1 (2-color), 2 (4-color), 4 (16-color), 8 (256-color), 15 (32K-color), 16 (64K-color), 24 (16M-color) or 32 (16M-color+alpha).

`INT        pixBytes`

Number of bytes per pixel (=`pixBits/8` = 0, 1, 2, 3 or 4).

`INT        pixPlanes`

Number of planes per pixel: 1, or 4 for VGA 16-color bitmaps (for 16-color VGA bitmaps `pixBits=1` and `pixPlanes=4`, for anything else `pixPlanes=1`).

`INT        pixResX, PixResY`

Number of pixels per inch, horizontal (X) and vertical (Y).

`INT        rowBytes`

Number of bytes in a single raster line. Normally this is an even multiple of the native CPU word size to insure that raster lines begin on aligned memory boundaries. (For example, `rowBytes` is normally evenly divisible by 4 on 32-bit processors, or evenly divisible by 2 on 16-bit processors.) `rowBytes` is always a positive value.

`INT        pitch`

`pitch` is the byte displacement between the start of one raster line to the beginning of the next. In some video modes, the separation between raster lines may be larger than `rowBytes`. For example, with certain display adapters 800x600 256-color modes have 800 bytes per raster line, but raster lines are separated at addresses of 1024 (a nice power of 2 for hardware simplicity). For this case, `rowBytes` would be set to 800, and `pitch` would be set to 1024.

`pitch` can either be positive or negative, depending if the bitmap is in a standard "top-down" format (positive), or special OS/2 style "bottom-up" format (negative). In a standard "top-down" bitmap, the first Y=0 raster line is located at the start of the bitmap surface area at a low memory address, with successive raster lines at increasing higher memory address locations. IBM OS/2, however, introduced a special reversed "bottom-up" style bitmap where the first y=0 raster line *is at the end of the bitmap area*, and successive raster lines occur at decreasing memory addresses. (For OS/2 "bottom-up" style bitmaps, the low memory address to the bitmap surface points to the *last* raster line instead of the first.) Although used infrequently, Microsoft Windows also supports "bottom-up" style bitmaps.

For OS/2 "bottom-up" style bitmaps, `pitch` is a negative byte displacement for moving between successive raster lines in decreasing memory address order (most "bottom-up" style bitmaps use "`pitch = -rowbytes`"). For standard "top-down" style bitmaps, `pitch` is a positive byte displacement for moving between successive raster lines in increasing memory address order (most "top-down" style bitmaps use "`pitch = rowBytes`").

`RECT        limitsRect`

A `RECT` structure initialized with the pixel limits of the bitmap: `(0,0,pixWidth,pixHeight)`.

`INT         rasterOp`

Current rasterOp transfer mode (see `MGRASTEROP` enumeration).

`void       *surface`

Low memory pointer to the start of the raster bitmap.

`void       *topY`

Memory pointer to the starting Y=0 raster line. For standard "top-down" bitmaps, the `topY` and `surface` pointers are the same (`topY = surface`). For OS/2 style "bottom-up" bitmaps, the `topY` Y=0 raster line is the last line in the bitmap memory (normally, `topY = surface + (pixHeight-1)*rowBytes`).

`MGCOLORRGB *colorTable`

Used with 16- and 256-color indexed bitmaps only. `colorTable` points to an array of RGB values defining the colors for 16- or 256-indexed color bitmaps. Each value defines the actual RGB color associated with the corresponding pixel value. (For non color-indexed bitmaps with greater than 256-colors, the `colorTable` pointer is `NULL`.)

The size of the `colorTable` array is actually one greater than the number of colors in the bitmap. For a 16-color bitmap, `colorTable` contains 17 entries; for a 256-color bitmap, `colorTable` contains 257 entries. The one additional entry located at the end of the `colorTable` array is termed the "colorTable signature" or "`CTSig`". `CTSig` is a counter that is incremented each time any of the RGB values in the main `colorTable` array is changed. A change in `CTSig` is commonly used to identify when a dependent precomputed setting, such as a color translation table, must be recomputed after the `colorTable` array has been updated with one or more new RGB values.

`MGCOLORPIX  xparColor`

Defines the transparent color when performing blit transfers with color transparency. The transparent color is set by calling the C `mgBitmap_SetTransparentColor()` function, or C++ `mgCBitmap::SetTransparaentColor()` method.

`BOOL        transActive`

Used with indexed-color bitmaps (16- and 256-color) only. `TRUE` indicates that blit transfers are to be color translated to the destination bitmap's colorTable.

`BYTE       *transTable`

Used with indexed-color bitmaps (16- and 256-color) only. `transTable` points to a pre-computed array for translating source bitmap pixel color indexes to destination bitmap pixel color indexes. This array is computed automatically by the blit functions when performing pixel transfers where the source and destination bitmaps have different RGB `colorTable` settings.

`HDC          hdc`

Handle to optional Microsoft Windows "Device Context" associated with this bitmap.

`BITMAPINFO *dibInfo`

Pointer to optional Microsoft Windows `BITMAPINFO` structure associated with this bitmap.

`MWGRAFPORT *grafPort`

Pointer to optional MetaWINDOW `grafPort` structure associated with this bitmap.

`MWGRAFMAP  *grafMap`

Pointer to optional MetaWINDOW `grafMap` structure associated with this bitmap.

`BYTE      **rowTable[4]`

Pointer(s) to optional raster line rowTable address array(s).

`FIXDOT      fixResX`

Pixels per inch, horizontal (fixed-point equivalent of `pixResX`).

`FIXDOT      fixResY`

Pixels per inch, vertical (fixed-point equivalent of `pixResY`).

MGCOLORPIX type

Typedef Definition

mgtypes.h

```
/* formatted bitmap pixel value (color index or encoded RGB)
 * If operating on a 16-bit CPU *AND* the maximum supported pixel size
 * is 16-bits or less, define MGCOLORPIX as a 16-bit short integer.
 * If running on a larger CPU *OR* if the maximum supported pixel size
 * is larger than 16-bits, define MGCOLORPIX as a long integer.
 */
#if      (UINT_MAX == 0xFFFFU) && (MGCONFIG_BITMAPSUPPORT_MAXBITS <= 16)
typedef unsigned short  MGCOLORPIX;
#else  /*(UINT_MAX  > 0xFFFFU) || (MGCONFIG_BITMAPSUPPORT_MAXBITS  > 16)
typedef unsigned long   MGCOLORPIX;
#endif
```

`MGCOLORPIX` defines pixel values in terms of bitmap-specfic pixel formats. The following table illustrates the various bitmap pixel formats that `MGCOLORPIX` may represent based on the associated bitmap in use:

Bitmap Bits-Per-Pixel	MGCOLORPIX
1	color index, 0 or 1
2	color index, 0 to 3
4	color index, 0 to 15
8	color index, 0 to 255

15	5:5:5 RGB color
16	5:6:5 RGB color
24	8:8:8 RGB color
32	8:8:8:8 RGBA color

`MGCOLORPIX` defines a format-specific pixel value used when reading or writing pixels to a bitmap. TypeServer bitmap `RGBToPix()` and `PixToRGB()` functions can be used to convert bitmap-specific pixel values to display-independent RGB colors, and vice-versa (see Chapter 6, "Library Reference", for expanded details on these functions):

C Functions

```
MGCOLORPIX  mgBitmap_RGBToPix(     /* convert RGB color to pixel value  */
            MGBITMAP    bitmap,          /* input,  bitmap-handle         */
            MGCOLORRGB  rgbColor );      /* input,  RGB color             */

MGCOLORRGB  mgBitmap_PixToRGB(     /* convert pixel value to RGB color */
            MGBITMAP    bitmap,          /* input,  bitmap-handle         */
            MGCOLORPIX  pixelValue );    /* input,  pixel value           */
```

C++ Methods

```
MGCOLORPIX  mgCBitmap::RGBToPix(   // convert RGB color to pixel value
            MGCOLORRGB  rgbColor );      // input,  RGB color

MGCOLORRGB  mgCBitmap::PixToRGB(   // convert pixel value to RGB color
            MGCOLORPIX  pixelValue );    // input,  pixel value
```

MGCOLORRGB type

Typedef Definition

mgtypes.h

```
/* RGB color value */
typedef unsigned long  MGCOLORRGB;
```

`MGCOLORRGB` defines an RGB color in a standard 32-bit format containing 8-bits intensity for each red, green and blue component, plus an optional 8-bits of alpha transparency. The RGB intensity components are encoded in the following form:

```
         3322 2222 2222 1111 1111 11
    Bit: 1098 7654 3210 9876 5432 1098 7654 3210  MGCOLORRGB
         ---- ---- ---- ---- ---- ---- bbbb bbbb  blue intensity (0-255)
         ---- ---- ---- ---- gggg gggg ---- ----  green intensity (0-255)
         ---- ---- rrrr rrrr ---- ---- ---- ----  red intensity (0-255)
         aaaa aaaa ---- ---- ---- ---- ---- ----  alpha transparency (0-255)
```

The macros `RGB_Make()`, `RGBA_Make()`, `RGB_GetRed()`, `RGB_GetGreen()`, `RGB_GetBlue()` and `RGB_GetAlpha()` provide easy to use methods for accessing `MGCOLORRGB` values. `MGCOLORPIX` defines a format-specific pixel value to use when reading or writing pixels to a bitmap. TypeServer bitmap RGBToPix() and PixToRGB() functions (see MGCOLORPIX, above) can be used to convert bitmap-specific pixel values to generic RGB colors, and vice-versa (see Chapter 6, "Library Reference", for full details on mgBitmap_RGBToPix() and mgBitmap_PixToRGB()).

POINT struct

Structure Definition

windows.h / mgtypes.h

```
/* point structure definition */
typedef struct tagPOINT
{   int     x;   /* horizontal x-coordinate */
    int     y;   /* vertical y-coordinate   */
} POINT;
```

RECT struct

Structure Definition

windows.h / mgtypes.h

```
/* rectangle structure definition */
typedef struct _RECT
{   int left;   /* x-min */
    int top;    /* y-min */
    int right;  /* x-max */
    int bottom; /* y-max */
} RECT;
```

RGB_GetRed(), RGB_GetGreen(), RGB_GetBlue(), RGB_GetAlpha()

C Syntax

```
/* return the red, green, blue or alpha components of an RGB color */
int  RGB_GetRed(   MGCOLORRGB  rgbColor );  /* input,  RGB color  */
int  RGB_GetGreen( MGCOLORRGB  rgbColor );  /* input,  RGB color  */
int  RGB_GetBlue(  MGCOLORRGB  rgbColor );  /* input,  RGB color  */
int  RGB_GetAlpha( MGCOLORRGB  rgbColor );  /* input,  RGB color  */
```

The four RGB_Get macros return the associated component, 0 to 255, from an RGB color.

Parameters

MGCOLORRGB rgbColor (input)

RGB color value.

Macro Definitions

mgtypes.h

```
#define  RGB_GetBlue(rgb)    ((rgb)        & 0xFF)
#define  RGB_GetGreen(rgb)  (((rgb) >>  8) & 0xFF)
#define  RGB_GetRed(rgb)    (((rgb) >> 16) & 0xFF)
#define  RGB_GetAlpha(rgb)  (((rgb) >> 24) & 0xFF)
```

See Also

`RGB_Make()`, `RGBA_Make()`, `MGCOLORPIX`, `MGCOLORRGB`, `mgBitmap_PixToRGB()`, `mgBitmap_RGBToPix()`

RGB_Make(), RGBA_Make()

C Syntax

```
/* encode RGB components into an RGB color */
MGCOLORRGB  RGB_Make(
       int    red,           /* input,  red intensity       (0-255) */
       int    green,         /* input,  green intensity     (0-255) */
       int    blue );        /* input,  blue intensity      (0-255) */

/* encode RGBA components into an RGB color */
MGCOLORRGB  RGBA_Make(
       int    red,           /* input,  red intensity       (0-255) */
       int    green,         /* input,  green intensity     (0-255) */
       int    blue,          /* input,  blue intensity      (0-255) */
       int    alpha );       /* input,  alpha transparency  (0-255) */
```

The `RGB_Make()` macro creates an RGB color from the supplied values.

Parameters

`int   red, green, blue, alpha  (input)`

Red, green, blue and alpha components for the color to be created. These should be integer values in the range 0 to 255.

Macro Definitions

mgtypes.h

```
#define RGB_Make(r,g,b)  ((MGCOLORRGB) (((r) << 16) | ((g) << 8)) | (b)))

#define RGBA_Make(r,g,b,a) ((MGCOLORRGB) \
                         (((a) << 24) | ((r) << 16) | ((g) << 8)) | (b)))
```

See Also

`RGB_GetRed()`, `RGB_GetGreen()`, `RGB_GetBlue()`, `RGB_GetAlpha()`, `MGCOLORPIX,`, `MGCOLORRGB`, `mgBitmap_PixToRGB()`, `mgBitmap_RGBToPix()`

TSALIGN enum

Enumeration Definition

typeserv.h

```
/* text alignment attributes */
typedef enum TSALIGN_
{
    tsLEFT_BASELINE   =0x00,            /* default */
    tsLEFT_BOTTOM     =0x01,
    tsLEFT_TOP        =0x02,
    tsCENTER_BASELINE =0x10,
    tsCENTER_BOTTOM   =0x11,
    tsCENTER_TOP      =0x12,
    tsRIGHT_BASELINE  =0x20,
    tsRIGHT_BOTTOM    =0x21,
    tsRIGHT_TOP       =0x22
} TSALIGN;
```

TSFONTINFO struct

Structure Definition

typeserv.h

```
typedef struct TSFONTINFO_
{
    INT32         structSize,     /* size of this structure (bytes)       */
    UINT32        objectType;     /* _FourCC object type (='TSFI')        */
    UINT          numFonts;       /* number of fonts within this file     */
    LONG          pathNameChars;  /* number of chars in filename string   */
    LONG          pathNameBytes;  /* number of bytes in filename string   */
    TCHAR        *filePathName;   /* pointer to path & filename string    */
    LONG          fileOffset;     /* starting offset of TTF font file     */
    LONG          fileSize;       /* # of bytes in the file memory buffer */
    void         *fontBuffer;     /* pointer to font file memory buffer   */
} TSFONTINFO;
```

`INT32   structSize`

Size of this `TSFONTINFO` structure, in bytes.

`UINT32  objectType`

A _FourCC '`TSFI`' character code identifying a `TSFONTINFO` structure.

`UINT    numFonts`

Number of fonts within this font file.

`LONG    pathNameChars`

Number of characters in the filename string.

`LONG    pathNameBytes`

Number of bytes in the filename string. `pathNameBytes=pathNameChars` for ASCII character strings, `pathNameBytes=(2 * pathNameChars)` for Unicode strings.

TCHAR *filePathName

Number of bytes in the filename string. pathNameBytes=pathNameChars for ASCII character strings, pathNameBytes=(2 * pathNameChars) for Unicode strings. NULL indicates font is resident in memory.

LONG   fileOffset

Offset within the file where the TrueType font actually begins. Normally fileOffset is 0, but this may be set to a positive value if the font is embedded within larger resource file.

void *fontBuffer

For memory based fonts, fontBuffer points to the start of the TrueType font in memory. For file based fonts, fontBuffer is NULL, and filePathName points to the font file path name.

TSSTRIKEINFO struct

Structure Definition

typeserv.h

```
typedef struct TSSTRIKEINFO_   /* strike rendering attribute information */
{
    INT32      structSize,     /* number of bytes in this struct         */
    UINT32     objectType,     /* FourCC object type (='TSSI')           */
    UINT16     iFont;          /* font number within file                */
    UINT16     numChars;       /* number of characters in this font      */
    UINT16     minChar;        /* minimum character code in this font    */
    UINT16     maxChar;        /* maximum character code in this font    */
    FIXDOT     pointSize;      /* char size in points (1pt = 1/72 inch)  */
    FIXDOT     emHeight;       /* em-square character height    (pixels) */
    FIXDOT     emWidth;        /* em-square character width     (pixels) */
    FIXDOT     ascender;       /* character ascender            (pixels) */
    FIXDOT     descender;      /* character descender           (pixels) */
    FIXDOT     height;         /* character height              (pixels) */
    FIXDOT     maxWidth;       /* maximum character width       (pixels) */
    FIXDOT     maxHeight;      /* maximum character height      (pixels) */
    FIXDOT     leading;        /* baseline spacing              (pixels) */
    FIXDOT     charExtra;      /* extra inter-character spacing (pixels) */
    FIXDOT     spaceExtra;     /* extra inter-word spacing      (pixels) */
    FIXDOT     orientation;    /* character orientation angle  (degrees) */
    FIXDOT     slant;          /* character slant angle        (degrees) */
    FIXDOT     path;           /* character path angle         (degrees) */
    FIXPOINT   location;       /* current x,y, drawing location (pixels) */
    MGCOLORPIX pixBack;        /* background color pixel value           */
    MGCOLORPIX pixChar;        /* character color pixel value            */
    MGCOLORRGB rgbBack;        /* background RGB color                   */
    MGCOLORRGB rgbChar;        /* character RGB color                    */
    TSALIGN    align;          /* character alignment position           */
    TSRASTEROP rasterOp;       /* rasterOp transfer mode                 */
    TSSMOOTH   smoothLevel;    /* edge smoothing level                   */
} TSSTRIKEINFO;
```

Structure Variables

INT32   structSize

Size of the TSSTRIKEINFO structure, in bytes.

`UINT32      objectType`

A _FourCC '`TSSI`' character code identifying a TSSTRIKEINFO structure.

`UINT16      iFont`

Sequence number (starting from 0) of this font in a TrueType file that contains multiple fonts .

`UINT16      numChars`

Number of displayable characters in this font.

`UINT16      minChar`

Minimum Unicode character code in this font.

`UINT16      maxChar`

Maximum Unicode character code in this font.

`FIXDOT      pointSize`

Character size expressed in typographic "points" (1-point equals 1/72 of an inch).

`FIXDOT      emHeight, emWidth`

Em-square height and width expressed in pixels. This is roughly the size of an upper-case "M", and is the size of the basic grid within which characters in the font are defined.

`FIXDOT      ascender, descender`

Maximum distances above and below the baseline that characters extend, in pixels.

`FIXDOT      height`

Character height, in pixels (`height = ascender + descender`).

`FIXDOT      maxWidth, maxHeight`

Maximum pixel spacing from one character to the next, horizontally and vertically.

`FIXDOT      baseline`

Vertical spacing between baselines, in pixels.

`FIXDOT      charExtra`

Pixel distance to add to the spacing between characters. `charExtra` is commonly used for creating justified lines of text. Positive values widen the character spacing, negative values tighten the character spacing. The default value is 0.0 (no extra character spacing).

`FIXDOT      wordExtra`

Pixel distance to add to the spacing between words (actually this distance is added to the spacing of all blank "space" characters). `spaceExtra` is commonly used when drawing justified lines of text. Positive values widen word spacing, negative values tighten word spacing. The default value is 0.0 (no extra word spacing).

`FIXDOT      path`

Direction angle, in degrees, for drawing a string of characters. The default path, 0.0 degrees, draws characters in a standard horizontal line. Positive degree values rotate text strings counter-clockwise, negative values rotate text strings clockwise. The default value is 0.0 degrees.

`FIXDOT      orientation`

Rotation angle, in degrees, for rotating individual characters within a line of text. The default orientation, 0.0 degrees, draws characters in their normal perpendicular orientation to the text path. Positive degree values rotate characters counter-clockwise, negative values rotate characters clockwise. The default value is 0.0 degrees.

`FIXDOT      slant`

Slant angle, in degrees, for tilting characters left or right. The default slant, 0.0 degrees, draws characters in their normal perpendicular orientation to the text path. Positive degree values slant characters to the left, negative values slant characters to the right. The default value is 0.0 degrees.

`FIXPOINT    location`

Current x,y pixel drawing location. After each character is drawn, the `location` coordinate is updated to the start of where the next character would normally begin. If a `NULL` pointer is specified for the text position in a drawing function call (such as to `tsStrike_DrawString`), `TSSTRIKEINFO.location` is used as the position for continued drawing.

`MGCOLORPIX  pixBack`

Character background color pixel-value. `pixBack` is the bitmap pixel-value closest matching the `rgbBack` background RGB color. If `rasterOp` is set to one of the "transparent" transfer modes, the character background is not drawn (only the foreground character is drawn).

`MGCOLORPIX  pixChar`

Character foreground color pixel-value. `pixChar` is the bitmap pixel-value closest matching the `rgbChar` character RGB color.

`MGCOLORRGB  rgbBack`

Character background RGB color (this color is the RGB version of the `pixBack` pixel-value).

`MGCOLORRGB  rgbChar`

Character foreground RGB color (this color is the RGB version of the `pixChar` pixel-value).

`TSALIGN     align`

`align` defines the alignment position for drawing characters and text. `align` is based on the `TSALIGN` enumeration, and can be any of nine settings:

```
tsLEFT_TOP        tsCENTER_TOP        tsRIGHT_TOP
tsLEFT_BASELINE   tsCENTER_BASELINE   tsRIGHT_BASELINE
tsLEFT_BOTTOM     tsCENTER_BOTTOM     tsRIGHT_BOTTOM
```

`TSRASTEROP  rasterOp`

RasterOp mode to use when drawing characters to the target bitmap. `rasterOp` controls how the character images are combined onto the target bitmap. Based on the TSRASTEROP enumeration, rasterOp may be any of eight settings:

```
tsCOPY,                /* replace (default)   */
tsMERGE,               /* OR                  */
tsERASE,               /* AND                 */
tsINVERT,              /* XOR                 */
tsTRANSPARENT_COPY,    /* transparent replace */
tsTRANSPARENT_MERGE,   /* transparent OR      */
tsTRANSPARENT_ERASE,   /* transparent AND     */
tsTRANSPARENT_INVERT   /* transparent XOR     */
```

TSSMOOTH   smoothLevel

Anti-aliased smoothing level to use when rendering character edges. smoothLevel is based on the TSSMOOTH enumeration and can be any of four settings:

```
tsSMOOTH0              /* no edge smoothing                 */
tsSMOOTH4              /* 4-level anti-aliasing             */
tsSMOOTH16             /* 16-level anti-aliasing (default) */
tsSMOOTH256            /* 256-level anti-aliasing           */
```

TSRASTEROP enum

Enumeration Definition

typeserv.h

```
/* RasterOp transfer attributes */
typedef enum TSRASTEROP_
{   tsCOPY,               /* replace             */
    tsMERGE,              /* OR                  */
    tsERASE,              /* AND                 */
    tsINVERT,             /* XOR                 */
    tsTRANSPARENT_COPY,   /* transparent replace */
    tsTRANSPARENT_MERGE,  /* transparent OR      */
    tsTRANSPARENT_ERASE,  /* transparent AND     */
    tsTRANSPARENT_INVERT  /* transparent XOR     */
} TSRASTEROP;
```

TSUNITS enum

Many TypeServer functions allow you to choose a convenient unit-of-measure for specifying type attributes. The TSUNITS enumeration allows you to specify character sizes and spacings either in terms of an absolute pixel height, or in terms of a device-independent typographic-point size (where 1 typographic-point equals 1/72nd of an inch).

Enum Definition

typeserv.h

```
/* Units of Measure */
typedef enum TSUNITS_
{   tsPIXELS,               /* units are pixels             */
    tsPOINTS,               /* units are typographic-points */
} TSUNITS;
```

Windows Structures and Data Types

The following Microsoft Windows structures and data types are provided here for reference.

Windows RGBQUAD struct

Structure Definition

wingdi.h

```
/* Windows GDI RGB color definition (UINT32) */
typedef struct tagRGBQUAD
{
    BYTE  rgbBlue;      /* blue intensity  */
    BYTE  rgbGreen;     /* green intensity */
    BYTE  rgbRed;       /* red intensity   */
    BYTE  rgbReserved;  /* reserved flags  */
} RGBQUAD;
```

Windows BITMAPINFO struct

Structure Definition

wingdi.h

```
/* Windows GDI device-independent bitmap (DIB) */
typedef struct tagBITMAPINFO
{
    BITMAPINFOHEADER bmiHeader;     /* BITMAPINFOHEADER structure */
    RGBQUAD          bmiColors[1];  /* bitmap color table array   */
} BITMAPINFO;
```

Windows BITMAPINFOHEADER struct

Structure Definition

wingdi.h

```
/* Windows GDI device-independent bitmap (DIB) header */
typedef struct tagBITMAPINFOHEADER
{
    DWORD  biSize;          /* size of this header, in bytes              */
    LONG   biWidth;         /* bitmap width, in pixels                    */
    LONG   biHeight;        /* bitmap height, in pixels                   */
    WORD   biPlanes;        /* planes per pixel - always 1                */
    WORD   biBitCount;      /* bits per pixel - 1,4,8,16,24 or 32         */
    DWORD  biCompression;   /* compression format (0=uncompressed)        */
    DWORD  biSizeImage;     /* size of the bitmap image, in bytes         */
    LONG   biXPelsPerMeter; /* horizontal pixels per meter                */
    LONG   biYPelsPerMeter; /* vertical pixels per meter                  */
    DWORD  biClrUsed;       /* number of color table entries used (0=all) */
    DWORD  biClrImportant;  /* number of important colors (0=all)         */
}  BITMAPINFOHEADER;
```

Math and Utility Functions

The following sections describe common math and utility functions that can be used by client application programs when using Metagraphics TypeServer.

PointsToPixels()

C Syntax **mgtypes.h**

```
/* convert typographic points to pixels (fixed-point) */
FIXDOT PointsToPixels(
       FIXDOT   pointSize,   /* input,  fixed-point point size      */
       FIXDOT   dpi );       /* input,  fixed-point pixels per inch */
```

The `PointsToPixels()` function converts a typographic point size value to a pixel size value at a given resolution. Using the standard of "72 points per inch", this function uses fixed-point arithmetic to perform the following calculation:

```
    pixelSize = ( pointSize * dpi ) / 72.0;
```

PixelsToPoints()

C Syntax **mgtypes.h**

```
/* Convert pixels to points ( fixed-point ) */
FIXDOT PixelsToPoints(
       FIXDOT   pixelSize,   /* input,  fixed-point pixel size      */
       FIXDOT   dpi );       /* input,  fixed-point pixels per inch */
```

The `PixelsToPoints()` function converts a `pixelSize` value at a given resolution (dpi, dots per inch) to a typographic `pointSize` value. Using the standard of "72 points per inch", the function uses fixed-point arithmetic to perform the following calculation:

```
    pointSize = ( pixelSize * 72.0 ) / dpi;
```

IntToFix()

C Syntax **mgtypes.h**

```
/* convert integer to fixed-point */
FIXDOT  IntToFix(
        int   intValue );  /* input,  integer value  */
```

`IntToFix()` is a C macro that shifts integer values to their appropriate fixed-point position:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define IntToFix(intValue)  ( (intValue) << 6 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define IntToFix(intValue)  ( (intValue) << 4 )
    #endif
```

FixToInt()

C Syntax

mgtypes.h

```
/* round and convert fixed-point to integer*/
int      FixToInt(
         FIXDOT    fixValue );  /* input, fixed-point value */
```

`FixToInt()` is a macro that is defined as follows:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define FixToInt(fixValue)  ( ( (fixValue) + 0x20 ) >> 6 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define FixToInt(fixValue)  ( ( (fixValue) + 0x08 ) >> 4 )
    #endif
```

FixTruncToInt()

C Syntax

mgtypes.h

```
/* truncate fixed-point to integer*/
int      FixTruncToInt(
         FIXDOT    fixValue );  /* input, fixed-point value */
```

`FixTruncToInt()` is a macro that is defined as follows:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define FixToInt(fixValue)  ( (fixValue) >> 6 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define FixToInt(fixValue)  ( (fixValue) >> 4 )
    #endif
```

FixFloor()

C Syntax

mgtypes.h

```
/* floor FIXDOT value to next lowest integer */
FIXDOT   FixFloor(
         FIXDOT    fixValue );  /* input, fixed-point value */
```

`FixFloor()` is a macro that is defined as follows:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define FixToInt(fixValue)  (FIXDOT)( (fixValue) & -64 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define FixFloor(fixValue)  (FIXDOT)( (fixValue) & -16 )
    #endif
```

FixCeil()

C Syntax

mgtypes.h

```
/* ceil FIXDOT value to next highest integer */
FIXDOT   FixCeil(
         FIXDOT    fixValue );  /* input, fixed-point value */
```

`FixCeil()` is a macro that is defined as follows:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define FixToInt(fixValue)  (FIXDOT)( ((fixValue)+63) & -64 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define FixFloor(fixValue)  (FIXDOT)( ((fixValue)+15) & -16 )
    #endif
```

FloatToFix()

C Syntax

mgtypes.h

```
/* convert floating-point value to fixed-point */
FIXDOT FloatToFix(
       float   floatValue ); /* input,  floating point value  */
```

`FloatToFix()` is a macro that is defined as follows:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define FloatToFix(floatValue)  (FIXDOT)( ((floatValue)*64.0) + 0.5 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define FloatToFix(floatValue)  (FIXDOT)( ((floatValue)*16.0) + 0.5 )
    #endif
```

FixToFloat()

C Syntax

mgtypes.h

```
/* convert fixed-point to floating point*/
float  FixToFloat(
       FIXDOT  fixValue ); /* input, fixed-point value */
```

`FixToFloat()` is a macro that is defined as follows:

```
    #ifdef   FIXMATH26         /* FIXDOT is F26.6 */
    #define FixToFloat(fixValue)  ( (fixValue) / 64.0 )
    #else /*!FIXMATH26*/       /* FIXDOT is F12.4 */
    #define FixToFloat(fixValue)  ( (fixValue) / 16.0 )
    #endif
```

Fixed-Point Constants

For convenience, the following fixed-point constants are defined for the `FIXDOT` data type:

Definitions

mgtypes.h

```
   #ifdef   FIXMATH26      /* FIXDOT is F26DOT6 */
   #define  FIXDOT_ONE        0x0040   /* = 1    = 1.0    */
   #define  FIXDOT_HALF       0x0020   /* = 1/2  = 0.5    */
   #define  FIXDOT_FOURTH     0x0010   /* = 1/4  = 0.25   */
   #define  FIXDOT_EIGHTH     0x0008   /* = 1/8  = 0.125  */
   #define  FIXDOT_SIXTEENTH  0x0004   /* = 1/16 = 0.0625 */
   #else /*!FIXMATH26*/    /* FIXDOT is F12DOT4 */
   #define  FIXDOT_ONE        0x0010   /* = 1    = 1.0    */
   #define  FIXDOT_HALF       0x0008   /* = 1/2  = 0.5    */
   #define  FIXDOT_FOURTH     0x0004   /* = 1/4  = 0.25   */
   #define  FIXDOT_EIGHTH     0x0002   /* = 1/8  = 0.125  */
   #define  FIXDOT_SIXTEENTH  0x0001   /* = 1/16 = 0.0625 */
   #endif
```

Here are some examples for specifying fixed-point values:

```
   10.5   = IntToFix( 10) + FIXDOT_HALF
  -22.25  = IntToFix(-22) – FIXDOT_FOURTH
   12.625 = IntToFix( 12) + FIXDOT_HALF + FIXDOT_EIGHTH
```

You can also use the `FloatToFix()` function, but keep in mind that this requires a floating-point multiply and a floating-point add (this is usually much slower than using the integer shift and adds, shown above):

```
    10.5 =  FloatToFix( 10.5 );
```

_ZeroMemory(), _ZeroStruct(), _InitStruct()

C Syntax

mgtypes.h

```
/* clear a block of memory to zero */
void _ZeroMemory( void *buffer, size_t byteCount );

/* clear a structure to zero */
void _ZeroStruct( void *structure );

/* zero a structure and set the first member (INT32)     */
/* equal to the size (in bytes) of the structure itself. */
void _InitStruct( void *structure );
```

_ZeroMemory(), _ZeroStruct() and _InitStruct() are macros that are defined as follows:

```
/* clear a block of memory to zero */
#ifdef    __cplusplus
inline  void _ZeroMemory( void *buf, size_t byteCount )
   { memset(buf, 0, byteCount); };
#else /* not __cplusplus */
#define _ZeroMemory(buf, byteCount)  memset(buf, 0, byteCount)
#endif /* __cplusplus */

/* clear a structure to zero */
#define _ZeroStruct(structure)  _ZeroMemory(structure, sizeof(*(structure)))

/* zero out a structure and set the first member (INT32) */
/* equal to the size (in bytes) of the structure itself. */
#define _InitStruct(structure) {                                    \
        _ZeroMemory(&(structure), sizeof(structure));               \
        *(INT32*) &(structure) = sizeof(structure);                 \
        }
```

Chapter 6 - TypeServer Library Reference

Class and Function Descriptions

This chapter provides a detailed description of each Metagraphics TypeServer. Listings are organized by class group, and alphabetically by function name within each group. Each listing provides specific information about what each function does, the parameters it takes, and pertinent information related to using the function. Function descriptions provide the following information:

Title

Title and summary of what the method does.

Syntax

Method calling prototype.

Description

In depth description of what the method does, the parameters it takes and any details you need to know about using the function.

Parameters

Description of each parameter.

Returns

Description of values (if any) returned.

Comments

Special notes and considerations.

See Also…

Related routines that you might wish to read more about.

Examples

Sample code or reference to example program.

Bitmap Functions

A TypeServer `MGBITMAP` object (C) and `mgCBitmap` class instance (C++) defines a bitmap for drawing and rendering operations. TypeServer provides flexibility to work with a wide range of bitmap types and formats. Bitmaps for TypeServer use generally fall into three categories:

1. External bitmaps that have been pre-allocated and created independently of TypeServer.
2. Video hardware bitmaps which are located at a predefined frame-buffer address.
3. TypeServer created bitmaps that your application can also access.

TypeServer bitmap objects provide the following capabilities:

- Creation of a `MGBITMAP` object (C) or `mgCBitmap` class instance (C++), and optional bitmap surface and colorTable.
- Support for standard pixel formats up to 24-bits per pixel.
- Performance optimized standard and transparent blits for transferring images between multiple bitmaps.
- Optimized solid-fill.
- Definition of custom bitmaps for special use operations.
- 16- and 256-color colorTable support, for:
    - Computing a color translation table for blitting between two bitmaps with different colorTables.
    - Remapping bitmap pixels to the nearest color of a new colorTable.
    - Locating the closest colorTable match for a given RGB color.

Function Groups

Bitmap functions are organized into the following groups:

Bitmap Creation and Deletion Functions:

C

```
mgBitmap_Create()       /* create a bitmap instance  */
mgBitmap_Destroy()      /* destroy a bitmap instance  */
```

C++

```
mgCBitmap::mgCBitmap()  // bitmap constructor
mgCBitmap::~mgCBitmap() // bitmap destructor
```

Bitmap Operating Functions

C

```
mgBitmap_Blit()                  /* transfer image bitmap to bitmap  */
mgBitmap_FillSolid()             /* solid pixel fill                 */
mgBitmap_FillSolidRGB()          /* solid RGB fill                   */
mgBitmap_GetInfoPtr()            /* get pointer to bitmapInfo        */
mgBitmap_PixToRGB()              /* convert pixel value to RGB color */
mgBitmap_RGBToPix()              /* convert RGB color to pixel value */
mgBitmap_SetRasterOp()           /* set transfer mode                */
mgBitmap_SetTopdown()            /* convert bitmap row format        */
mgBitmap_SetTransparentColor()   /* set transparent color            */
mgBitmap_SetTransparent()        /* enable/disable transparent blits */
```

C++

```
mgCBitmap::Blit()                // transfer image bitmap to bitmap
mgCBitmap::FillSolid()           // solid pixel fill
mgCBitmap::FillSolidRGB()        // solid RGB fill
mgCBitmap::GetInfoPtr()          // get pointer to bitmapInfo
mgCBitmap::PixToRGB()            // convert pixel value to RGB color
mgCBitmap::RGBToPix()            // convert RGB color to pixel value
mgCBitmap::SetRasterOp()         // set transfer mode
mgCBitmap::SetTopdown()          // convert bitmap row format
mgCBitmap::SetTransparentColor() // set transparent color
mgCBitmap::SetTransparent()      // enable/disable transparent blits
```

Bitmap colorTable and palette functions for 16- and 256-color bitmaps

C

```
mgBitmap_ComputeTranslate()      /* compute color translation table   */
mgBitmap_FindClosestRGB()        /* find closest RGB                  */
mgBitmap_MapColors()             /* set colorTable & remap pixels     */
mgBitmap_SetColors()             /* set new colorTable                */
mgBitmap_SetTranslate()          /* enable color translation          */
mgBitmap_CacheTranslate()        /* enable/disable colorTable caching */
```

C++

```
mgCBitmap::ComputeTranslate()    // compute color translation table
mgCBitmap::FindClosestRGB()      // find closest RGB
mgCBitmap::MapColors()           // set colorTable & remap pixels
mgCBitmap::SetColors()           // set new colorTable
mgCBitmap::SetTranslate()        // enable color translation
mgCBitmap::CacheTranslate()      // enable/disable colorTable caching
```

Windows-specific functions

C

```
mgBitmap_BlitDC()                  /* solid blit to Windows DC    */
mgBitmap_CreateIdentityPalette()   /* create Win identity palette */
mgBitmap_CreatePalette()           /* create Win logical palette  */
```

C++

```
  mgCBitmap::BlitDC()                // solid blit to Windows DC
  mgCBitmap::CreateIdentityPalette() // create Win identity palette
  mgCBitmap::CreatePalette()         // create Win logical palette
```

Prototypes

Following is a summary of the bitmap function prototypes defined in the `mgbitmap.h` header file. The information in the `mgbitmap.h` header file may include updated information that includes additions or changes that supercede the printed document. Please review `mgbitmap.h` for current specifications.

C Prototypes

```
/* bitmap creation */
MRESULT  mgBitmap_Create(
         MGSYSTEM       mgSystem,      /* input,  system-handle           */
         MGBITMAPINFO  *bitmapInfo,    /* in/out, bitmap information      */
         MGBITMAP      *bitmapHandle); /* output, bitmap-handle           */

/* bitmap destruction */
MRESULT  mgBitmap_Destroy(
         MGBITMAP      *bitmapHandle); /* in/out, bitmap-handle           */

/* get bitmap information pointer */
MGBITMAPINFO*  mgBitmap_GetInfoPtr(    /* return, ptr to bitmapInfo data  */
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         int            structSize );  /* input,  MGBITMAPINFO struct size*/

/* transfer image, bitmap to a bitmap */
MRESULT  mgBitmap_Blit(
         MGBITMAP       srcBitmap,    /* input, source bitmap-handle      */
         MGBITMAP       dstBitmap,    /* input, destination bitmap-handle */
   const RECT          *dstRect,      /* input, destination rectangle(s)  */
   const RECT          *srcRect,      /* input, source rectangle(s)       */
         int            rectCount );  /* input, number of rects to blit   */

/* solid fill by pixel value */
MRESULT  mgBitmap_FillSolid(
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         MGCOLORPIX     pixelValue,    /* input,  fill pixel value        */
   const RECT          *rectList,      /* input,  rectangle(s) to fill    */
         int            rectCount );   /* input,  number of rects to fill */

/* solid fill by RGB color */
MRESULT  mgBitmap_FillSolidRGB(
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         MGCOLORRGB     rgbColor,      /* input,  fill RGB value          */
   const RECT          *rectList,      /* input,  rectangle(s) to fill    */
         int            rectCount );   /* input,  number of rects to fill */
```

```
/* get bitmap information pointer */
MGBITMAPINFO* GetInfoPtr(              /* return, ptr to bitmapInfo data  */
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         int            structSize );  /* input,  bitmapInfo struct size  */

/* convert pixel value to RGB color */
MGCOLORRGB  mgBitmap_PixToRGB(         /* return, RGB color               */
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         MGCOLORPIX     pixelValue );  /* input,  pixel value             */

/* convert RGB color to pixel value */
MGCOLORPIX  mgBitmap_RGBToPix(         /* return, pixel value             */
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         MGCOLORRGB     rgbColor );    /* input,  RGB color               */

/* Set transfer mode */
MRESULT  mgBitmap_SetRasterOp(
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         MGRASTEROP     rasterOp );    /* input,  transfer mode           */

/* convert bitmap to top-down or bottom-up format */
MRESULT  mgBitmap_SetTopdown(
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         BOOL           topDown );     /* input,  topDn(TRUE)/botUp(FALSE)*/

/* set transparent color */
MRESULT  mgBitmap_SetTransparentColor(
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         MGCOLORPIX     pixelValue );  /* input,  transparent pixel value */

/* enable/display transparent blits */
MRESULT  mgBitmap_SetTransparent(
         MGBITMAP    bitmapHandle,  /* input,  bitmap-handle             */
         BOOL        transparent ); /* input, enable(TRUE)/disable(FALSE)*/

/* colorTable and palette functions for 16- and 256-color bitmaps - - - - */

/* set new color table values */
MRESULT  mgBitmap_SetColors(
         MGBITMAP       bitmapHandle,  /* input,  bitmap instance handle  */
   const MGCOLORRGB    *rgbColors,     /* input,  new colorTable values   */
         UINT           startIndex,    /* input,  starting table index    */
         UINT           numEntries );  /* input,  number for values to set*/

/* set new colorTable values and remap pixel colors */
MRESULT  mgBitmap_MapColors(
         MGBITMAP       bitmapHandle,  /* input,  bitmap instance handle  */
   const MGCOLORRGB    *rgbColors,     /* input,  new colorTable values   */
         UINT           startIndex,    /* input,  starting table index    */
         UINT           numEntries );  /* input,  number for values to set*/

/* return index in current colorTable most closely matching given RGB */
MGCOLORPIX  FindClosestRGB(            /* return, closest matching index  */
   const MGCOLORRGB     rgbColor );    /* input,  RGB color to locate     */
```

```
/* enable color translation to destination bitmap */
MRESULT  mgBitmap_SetTranslate(
         MGBITMAP       bitmapHandle,  /* input,  bitmap instance handle  */
         BOOL           translate );   /* input,  translate=TRUE          */

/* compute color translation table for a destination bitmap */
MRESULT  mgBitmap_ComputeTranslate(
         MGBITMAP       bitmapHandle,  /* input, bitmap-handle           */
         MGBITMAP       dstBitmapHdl); /* input, destination bmap handle */

/* Windows-Specific Functions - - - - - - - - - - - - - - - - - - - - - - - - */

/* create a Windows logical palette from the bitmap color table */
MRESULT  mgBitmap_CreatePalette(
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle           */
         HPALETTE      *hPalette );    /* output, Windows palette-handle  */

/* create a Windows logical identity palette from the bitmap color table */
MRESULT  mgBitmap_CreateIdentityPalette(
         MGBITMAP       bitmapHandle,  /* input,  bitmap instance handle  */
         BOOL           remapPixels,   /* input,  remap pixels? (yes=TRUE)*/
         HPALETTE      *hPalette );    /* output, Windows palette-handle  */

/* non-transparent blit to a Windows Device Context */
MRESULT  mgBitmap_BlitDC(
         MGBITMAP       srcBitmap,    /* input, source bitmap-handle      */
         HDC            dstDc,        /* input, destination DC            */
         const RECT    *dstRect,      /* input, destination blit rectangle*/
         int            srcX, srcY ); /* input, source blit origin        */
```

C++ Prototypes

```
class mgCBitmap : virtual public tsCObject
{
  public:  // Public Methods

    // construct a mgCBitmap object
    mgCBitmap(
        MGBITMAPINFO  *bitmapInfo );  // in/out, bitmap information

    // copy constructor
    mgCBitmap( const mgCBitmap& );

    // assignment operator
    mgCBitmap& operator=( const mgCBitmap& );

    // destructor
    ~mgCBitmap();

    // blit to a destination bitmap
    MRESULT  Blit(
         mgCBitmap     *dstBitmap,    // input, destination bitmap
         const RECT    *srcRect,      // input, source blit rectangle
         const RECT    *dstRect,      // input, destination blit rectangle
         int            numBlits=1 ); // input, number of rects to blit
```

```
// solid color fill
MRESULT  FillSolid(
     MGCOLORPIX     fillColor,     // input,  fill color pixel value
     const RECT    *rectList,      // input,  rectangle(s) to fill
     int            rectCount=1 ); // input,  number of rects to fill

// get bitmap information pointer
MGBITMAPINFO*  GetInfoPtr(         // return, ptr to bitmapInfo data
     int            structSize );  // input,  MGBITMAPINFO struct size

// convert pixel value to RGB color
MGCOLORRGB  PixToRGB(              // return, RGB color
     MGCOLORPIX     pixelValue );  // input,  pixel value

// convert an RGB color to a pixel value
MGCOLORPIX  RGBToPix(              // return, pixel value
     MGCOLORRGB     rgbColor );    // input,  RGB color

// Set rasterOp transfer mode
MRESULT  SetRasterOp(
     MGRASTEROP     rasterOp );    // input,  transfer mode

// convert bitmap to top-down or bottom-up format
MRESULT  SetTopdown(
     BOOL           topDown );     // input,  topDn(TRUE)/botUp(FALSE)

// set transparent color
MRESULT  SetTransparentColor(
     MGCOLORPIX     pixelvalue );  // input,  transparent pixel value

// enable/disable transparent blits
MRESULT  SetTransparent(
     BOOL           transparency); // input, enable(TRUE)/disable(FALSE)

// colorTable and palette functions for 16- and 256-color bitmaps - - -

// compute color translation table for a destination bitmap */
MRESULT  ComputeTranslate(
     mgCBitmap     *dstBitmap );   // input,  destination bitmap

// return index in current colorTable most closely matching given RGB
MGCOLORPIX  FindClosestRGB(        // return, closest matching index
     MGCOLORRGB     rgbColor );    // input,  RGB color to locate

// set new colorTable values and remap pixel colors
MRESULT  MapColors(
const MGCOLORRGB   *rgbColors,     // input,  new colorTable values
     UINT           startIndex,    // input,  starting table index
     UINT           numEntries );  // input,  number for values to set
```

```
    // set new color table values
    MRESULT  SetColors(
    const MGCOLORRGB   *rgbColors,     // input,  new colorTable values
         UINT           startIndex,    // input,  starting table index
         UINT           numEntries );  // input,  number for values to set

    // enable color translation to destination bitmap
    MRESULT  SetTranslate(
         BOOL           translate );   // input,  translate=TRUE

    // System-Specific Functions - - - - - - - - - - - - - - - - - - - - -

    // create a Windows logical palette from the bitmap color table
    MRESULT  CreatePalette(
         HPALETTE      *hPalette );   // output, Windows palette handle
    // create a Windows logical identity palette from the color table

    MRESULT  CreateIdentityPalette(
         BOOL           remapPixels,  // input,  remap pixels? (yes=TRUE)
         HPALETTE      *hPalette );   // output, Windows palette-handle

    // non-transparent blit to a Windows Device Context
    MRESULT  BlitDC(
         HDC            dstDc,        // input, destination DC
         const RECT    *srcRect,      // input, source blit rectangle
         const RECT    *dstRect,      // input, destination blit rectangle
         int            numBlits=1 ); // input, number of rects to blit

};  // class mgCBitmap
```

Function Descriptions

The following sections provide detailed descriptions of each TypeServer bitmap function, organized alphabetically by function name.

mgBitmap_Blit() - blit to a destination bitmap

C Syntax

mgbitmap.h

```
/* blit to a destination bitmap */
MRESULT  mgBitmap_Blit(
         MGBITMAP     srcBitmap,   /* input, source bitmap-handle         */
         MGBITMAP     dstBitmap,   /* input, destination bitmap-handle    */
   const RECT        *srcRect,     /* input, source blit rectangle(s)     */
   const RECT        *dstRect,     /* input, destination blit rectangle(s)*/
         int          numBlits );  /* input, number of rects to blit      */
```

C++ Syntax mgbitmap.hpp

```
// blit to a destination bitmap
MRESULT  mgCBitmap::Blit(
         mgCBitmap   *dstBitmap,    // input, destination bitmap
   const RECT        *srcRect,      // input, source blit rectangle(s)
   const RECT        *dstRect,      // input, destination blit rectangle(s)
         int          numBlits=1 ); // input, number of rects to blit
```

The `Blit()` function transfers pixels of the source bitmap to a destination bitmap object. The `MGBITMAPINFO.blitType` variable selects the type of blit performed. `Blit()` transfers pixels from `srcRect` rectangle(s) on the source bitmap to the `dstRect` rectangle(s) on the destination bitmap.

If `MGBITMAPINFO.blitType` (set by `SetRasterOp()`) is set for a "transparent" transfer mode, only pixels with colors that do not match the `MGBITMAPINFO.xparColor` transparent color (set by `SetTransparentColor()`) are copied. For non-transparent blits, all pixels within the `srcRect` rectangle(s) are copied to the destination bitmap.

Clip testing is performed to insure that both the destination rectangle and source rectangle are wholly contained within their respective bitmaps. If the rectangle limits extend beyond the edge of the bitmap, the rectangle is automatically clipped to the appropriate bitmap limit.

Parameters

`srcBitmap  (C, input)`

Source bitmap-handle.

`dstBitmap  (C, input)`

Destination bitmap-handle.

`*dstBitmap (C++, input)`

Pointer to destination bitmap class instance.

`*srcRect   (input)`

Pointer to an array of one or more `RECTs` defining the location(s) on the source bitmap to blit from.

`*dstRect   (input)`

Pointer to an array of one or more `RECTs` defining the location(s) on the destination bitmap to blit to.

`numBlits   (input)`

Number of rectangles to blit (1 or greater).

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

Blit() supports only 1:1 blit transfers - "stretched" blits between source and destination rectangles with different sizes are not supported. In operation, the upper-left corner of the srcRect is used to define the source origin of the blit, and dstRect defines the destination location and dimensions of the blit (the lower-right corner coordinates of srcRect are ignored).

See Also

mgBitmap_BlitDC()- device context, mgBitmap_SetRasterOp()

mgBitmap_BlitDC() - blit to a Windows DC bitmap

C Syntax

mgbitmap.h

```
/* non-transparent blit to a Windows Device Context */
MRESULT  mgBitmap_BlitDC(
         MGBITMAP srcBitmap,    /* input, source bitmap handle          */
         HDC      dstDc,        /* input, destination DC                */
   const RECT    *srcRect,      /* input, source blit rectangle(s)      */
   const RECT    *dstRect,      /* input, destination blit rectangle(s) */
         int      numRects );   /* input, number of rects to blit       */
```

C++ Syntax

mgbitmap.hpp

```
// non-transparent blit to a Windows Device Context
MRESULT  mgCBitmap::BlitDC(
         HDC      dstDc,        // input, destination DC
   const RECT    *srcRect,      // input, source blit rectangle(s)
   const RECT    *dstRect,      // input, destination blit rectangle(s)
         int      numRects=1 ); // input, number of rects to blit
```

The device context BlitDC() function transfers pixels of the bitmap to the specified destination device context. BlitDC() performs a non-transparent transfer that replaces all the pixels within the dstRect rectangle on the destination device context with pixels from the source bitmap.

Parameters

srcBitmap (C, input)

Source bitmap-handle.

dstDc   (input)

Windows handle to the destination device context.

*srcRect   (input)

Pointer to an array of one or more RECTs defining the location(s) on the source bitmap to blit from.

*dstRect   (input)

Pointer to an array of one or more RECTs defining the location(s) on the destination bitmap to blit to.

numRects (input)

Number of rectangles to blit (1 or greater).

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

There are no means available to perform a transparent blit to a device context (transparent blits can only be performed between two TypeServer bitmaps).

The BlitDC() transfer to a device context is performed in either of two manners depending if the source bitmap has a device context or not. If the source bitmap has a device context, the transfer is performed using the faster Windows WinGBitBlt() function. If the source bitmap does not have a device context, the transfer is performed using the (slower) Windows StretchDIBits() function.

BlitDC() supports only 1:1 blit transfers - "stretched" blits between source and destination rectangles with different sizes are not supported. In operation, the upper-left corner of the srcRect is used to define the source origin of the blit, and dstRect defines the destination location and dimensions of the blit (the lower-right corner coordinates of srcRect are ignored).

See Also

mgBitmap_Blit()

mgBitmap_ComputeTranslate() - compute color-translation table

C Syntax mgbitmap.h

```
/* compute color translation table for a destination bitmap */
MRESULT  mgBitmap_ComputeTranslate
         MGBITMAP       srcbitmap,   /* input, source bitmap-handle      */
         MGBITMAP       dstBitmap);  /* input, destination bitmap-handle */
```

C++ Syntax mgbitmap.hpp

```
// compute color translation table for a destination bitmap
MRESULT  mgCBitmap::ComputeTranslate
         mgCBitmap     *dstBitmap);  // input, destination bitmap
```

Description

For indexed-color bitmaps using a color lookup table (256-colors or less), ComputeTranslate() compares the colorTable of the source bitmap with the colorTable of a target destination bitmap, dstBitmap. If the colorTables do not match, ComputeTranslate() calculates the MGBITMAPINFO.transTable color translation table and sets MGBITMAPINFO.transActive to TRUE enabling pixel color translation when blitting to a destination bitmap with a differing colorTable.

If the colorTables match, `ComputeTranslate()` sets `MGBITMAPINFO.transActive` to `FALSE` indicating that no color translation is needed.

Parameters

`srcBitmap   (C, input)`

Source bitmap-handle.

`dstBitmap   (C, input)`

Destination bitmap-handle.

`*dstBitmap  (C++, input)`

Pointer to the destination bitmap.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

When comparing colorTables, only colorTable entries with flag bytes *not set* to `mColorNotUsed`, `mColorPrivate` or `mColorNoTranslate` are tested. Blitting between bitmaps with matching colorTables is the fastest of all blits (needs no color translation), and is specially optimized.

See Also

`mgBitmap_SetTranslate(), MGBITMAPINFO`

mgBitmap_Create() - create a bitmap instance

C Syntax **mgbitmap.h**

```
/* create a TypeServer bitmap object */
MRESULT  mgBitmap_Create(
         MGSYSTEM       mgSystem,      /* input,  system-handle           */
         MGBITMAPINFO  *bitmapInfo,    /* in/out, bitmap information      */
         MGBITMAP      *bitmap );      /* output, bitmap instance handle  */
```

C++ Syntax **mgbitmap.hpp**

```
// construct a mgCBitmap instance
MRESULT  mgCBitmap::mgCBitmap(
         mgCSystem     *mgcSystem,     // input,  system instance
         MGBITMAPINFO  *bitmapInfo );  // in/out, bitmap information
```

Description

Metagraphics TypeServer provides broad flexibility in working with a wide variety of bitmap types and formats. The `mgBitmap_Create()` function (C) and `mgCBitmap()` constructor (C++) is used to create a bitmap object for any of the following common situations:

1. An external bitmap that has been pre-allocated and created independently of TypeServer,
2. A video hardware bitmap located at a predefined frame-buffer address, or
3. A Metagraphics bitmap with a dynamically-created pixel surface that your application can also access.

In defining a bitmap, your application first fills out a MGBITMAPINFO structure that is passed to the create function. The bitmap information provides basic details about the size, format and location of the bitmap in memory, or if a new bitmap should be created. Upon return, mgBitmap_Create() provides a handle for the bitmap that can be used when calling other TypeServer functions. Also on return, other associate variables in the MGBITMAPINFO structure are also initialized.

Parameters

`*mgSystem    (C, input)`

System-handle returned by the tsServer_Create() function.

`*mgCSystem   (C++, input)`

Pointer to the Metagraphics system-instance returned by tsServer::tsServer() constructor.

`*bitmapInfo  (input/output)`

Pointer to a MGBITMAPINFO structure defining the size, format and location of the bitmap in memory.

`*bitmap      (C, output)`

Pointer to a MGBITMAP variable where the instance-handle for the bitmap is to be returned. When declaring this variable, your application should initialize it to a value of NULL (see Comments, below).

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

For C, the bitmap parameter should be initialized to a value of NULL. Calling mgBitmap_Create() with a non-NULL bitmap parameter will result in an MRESULT warning. This convention is used to insure that a second mgBitmap_Create() call does not overwrite a previously opened bitmap-handle before mgBitmap_Destroy() has been called:

```
    /* declare a bitmap instance handle and initialize it to NULL */
    MGBITMAP  myBitmap=NULL;
```

To release dynamically allocated resources, all bitmaps created by mgBitmap_Create() must be released by your application when no longer needed (and before program termination) by calling mgBitmap_Destroy().

The Chapter 4 "**TypeServer Bitmaps**" section provides step by step instructions showing how to define either external or dynamically-created bitmaps.

See Also

mgBitmap_Destroy(), tsServer_Create()

mgBitmap_CreatePalette() - create Windows logical palette

C Syntax

mgbitmap.h

```
/* create a Windows logical palette from the bitmap color table */
MRESULT  mgBitmap_CreatePalette(
         MGBITMAP       bitmap,       /* input,  bitmap instance handle   */
         HPALETTE      *hPalette );   /* output, Windows palette handle   */
```

C++ Syntax

mgbitmap.hpp

```
// create a Windows logical palette from the bitmap color table
MRESULT  mgCBitmap::CreatePalette(
         HPALETTE      *hPalette );   // output, Windows palette handle
```

Description

CreatePalette() creates a Windows logical palette corresponding to the current color table, and returns a handle suitable for realizing the palette on the physical display.

Parameters

bitmap   (C, input)

TypeServer bitmap instance handle.

*hPalette (output)

Windows system handle to the new logical identity palette.

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

The Windows system palette handle returned by CreatePalette() must be manually deleted by the caller via Windows DeleteObject() when the logical palette is no longer needed.

See Also

mgBitmap_CreateIdentityPalette(), mgBitmap_SetColors(), mgBitmap_MapColors()

mgBitmap_CreateIdentityPalette() - create identity palette

C Syntax **mgbitmap.h**

```
/* create a Windows logical identity palette from the bitmap color table */
MRESULT  mgBitmap_CreateIdentityPalette(
         MGBITMAP       bitmap,       /* input,  bitmap instance handle   */
         BOOL           remapPixels,  /* input,  remap pixels? (yes=TRUE) */
         HPALETTE      *hPalette );   /* output, Windows palette handle   */
```

C++ Syntax **mgbitmap.hpp**

```
// create a Windows logical identity palette from the bitmap color table
MRESULT  mgCBitmap::CreateIdentityPalette(
         BOOL           remapPixels,  // input,  remap pixels? (yes=TRUE)
         HPALETTE      *hPalette );   // output, Windows palette handle
```

Description

`CreateIdentityPalette()` modifies the bitmap color table replacing the first ten and last ten color entries (colors 0-9 and 246-255) with the Windows system reserved colors. From the updated color table, `CreateIdentityPalette()` creates a new logical palette and returns a Windows system handle for use by the calling function. When the new palette is realized for the display, a 1 to 1 identity mapping of bitmap colors to display colors will exist enabling `BlitDC()` to execute at optimized speed without the overhead of color translation.

The Windows system palette handle returned by `CreateIdentityPalette()` must be manually deleted by the caller via Windows `DeleteObject()` when the logical palette is no longer needed.

Parameters

`bitmap    (C, input)`

TypeServer bitmap instance handle.

`remap     (input)`

If `TRUE`, bitmap pixels for the modified color table entries are remapped to the next closest match of the updated color table.

`*hPalette (output)`

Windows system handle to the new logical identity palette.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

Windows reserves 20 color table entries (0-9 and 246-255) in the display palette for system use. Although these colors are given generic meanings (red, blue, yellow, etc.), the specific RGB values for these entries vary based on the display hardware. `CreateIdentityPalette()` replaces entries 0-9 and 246-255 in the bitmap color table with the exact values from the current system palette.

(If you have previously called Windows SetSystemPaletteUse(SYSPAL_NOSTATIC) to reduce the number of reserved colors to just entries 0 and 255 (black and white), then only the first and last color table values will be replaced.)

See Also

mgBitmap_CreatePalette(), mgBitmap_SetColors(), mgBitmap_MapColors()

mgBitmap_Destroy() - destroy bitmap instance

C Syntax

mgbitmap.h

```
/* delete bitmap object */
MRESULT  mgBitmap_Destroy(
         MGBITMAP      *bitmap);  /* in/out, bitmap instance handle      */
```

C++ Syntax

mgbitmap.hpp

```
// delete bitmap object
mgCBitmap::~mgCBitmap();  // destructor
```

Description

The Destroy() function releases all memory and system resources that have been allocated for the specified bitmap. This function must be called to close and release all resources allocated to a bitmap instance by mgBitmap_Create().

Parameters

*bitmap   (C, input/output)

Pointer to the bitmap instance handle to close. As part of its processing, Destroy() will reset the bitmap-handle to NULL.

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

See Also

mgBitmap_Create()

mgBitmap_FillSolid(), mgBitmap_FillSolidRGB - solid fill

C Syntax **mgbitmap.h**

```
/* solid color fill by pixel value*/
MRESULT  mgBitmap_FillSolid(
         MGBITMAP       bitmap,        /* input,  bitmap-handle           */
         MGCOLORPIX     pixelValue,    /* input,  fill color pixel value  */
         const RECT    *rectList,      /* input,  rectangle(s) to fill    */
         int            numRects );    /* input,  number of rects to fill */

/* solid color fill by RGB color */
MRESULT  mgBitmap_FillSolidRGB(
         MGBITMAP       bitmap,        /* input,  bitmap-handle           */
         MGCOLORRGB     rgbColor,      /* input,  fill RGB color          */
         const RECT    *rectList,      /* input,  rectangle(s) to fill    */
         int            numRects );    /* input,  number of rects to fill */
```

C++ Syntax **mgbitmap.hpp**

```
// solid color fill by pixel value */
MRESULT  mgCBitmap::FillSolid(
         MGCOLORPIX     pixelValue,    // input,  fill color pixel value
         const RECT    *rectList,      // input,  rectangle(s) to fill
         int            numRects=1 );  // input,  number of rects to fill

// solid color fill by pixel value */
MRESULT  mgCBitmap::FillSolidRGB(
         MGCOLORRGB     rgbColor,      // input,  fill RGB color
         const RECT    *rectList,      // input,  rectangle(s) to fill
         int            numRects=1 );  // input,  number of rects to fill
```

Description

`FillSolid()` and `FillSolidRGB()` fill one or more rectangular areas on a bitmap with a specified solid fill color. The two functions allow you to define the fill color either in terms of a specific pixel value for the bitmap, or in terms of a generic RGB color.

Parameters

```
TSSTRIKE    strike      (C, input)
```

Strike-handle to set the foreground character and background colors for.

```
MGCOLORPIX  pixChar     (SetColors(), input)
```

Formatted pixel value for the character.

```
MGCOLORPIX  pixBack     (SetColors(), input)
```

Formatted pixel value for the background.

Pixel values are format-specific for the attached output bitmap. If the output bitmap is 256 colors or less, the values are indexes into the bitmap's colorTable. If the bitmap is greater than 256-colors, pixel values contain encoded RGB formatted for the specific bitmap type (for example,. a 16 bit-per-pixel bitmap stores RGB values in a 5:6:5 format within a single 2 byte value). The mgBitmap_RGBToPix() function can be used to convert generic RGB colors to bitmap pixel values.

```
MGCOLORRGB  rgbChar     (SetColorsRGB(), input)
```

Generic RGB color for the character image.

```
MGCOLORRGB  rgbBack     (SetColorsRGB(), input)
```

Generic RGB color for the character background.

The SetColorsRGB() function may be used to specify character colors using generic RGB values. The RGB_Make(r,g,b) macro provides a simple method for encoding R,G,B intensity components into an generic RGB value.

FillSolid() and FillSolidRGB() fill one or more rectangular areas on the bitmap with a specified fill color.

Parameters

```
bitmap      (C, input)
```

Bitmap-handle.

```
pixelValue  (FillSolid(), input)
```

Direct pixel value for the fill color. Pixel values are format-specific for the bitmap. If the output bitmap is 256 colors or less, the values are indexes into the bitmap's colorTable. If the bitmap is greater than 256-colors, pixels are encoded RGB values formatted for the specific bitmap type (for example,. a 16 bit-per-pixel bitmap stores RGB values in a 5:6:5 format within a single 16-bit value). The mgBitmap_RGBToPix() function can be used to convert generic RGB colors to bitmap pixel values.

```
rgbColor    (FillSolidRGB(), input)
```

Generic RGB fill color. The RGB_Make(r,g,b) macro provides a simple method for encoding R,G,B intensities into an generic RGB value.

```
*rectList  (input)
```

Pointer to the first member of an array of rectangles defining area(s) in the bitmap to fill.

```
numRects   (input)
```

Number of rectangles to fill.

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

mgBitmap_FillSolidRGB() is actually a macro that invokes mgBitmap_RGBToPix() prior to calling mgBitmap_FillSolid():

```
#define mgBitmap_FillSolidRGB(bmap,rgb,rectlist,rectcnt)  \
    mgBitmap_FillSolid(bmap, mgBitmap_RGBToPix(bmap,rgb), rectlist, rectcnt)
```

mgBitmap_FindClosestRGB() - find closest RGB index

C Syntax

mgbitmap.h

```
/* return index in current colorTable most closely matching given RGB */
MGCOLORPIX  mgBitmap_FindClosestRGB(   /* return, closest matching index */
         MGBITMAP       bitmap,        /* input,  bitmap-handle          */
         MGCOLORRGB     rgbColor );    /* input,  RGB color to locate    */
```

C++ Syntax

mgbitmap.hpp

```
// return index in current colorTable most closely matching given RGB
MGCOLORPIX  mgCBitmap::FindClosestRGB( // return, closest matching index
         MGCOLORRGB     rgbColor );    // input,  RGB color to locate
```

Description

For 16- and 256-color bitmaps, `FindClosestRGB()` locates the color in the bitmap's colorTable that most closely matches the specified RGB color.

Parameters

`bitmap   (C, input)`

Bitmap-handle.

`rgbColor (input)`

RGB color to find the closest color table match for.

`*index   (output)`

Index in the current colorTable that most closely matches the specified `rgbColor` value.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

`FindClosestRGB()` is designed for use with 16- and 256-color bitmaps where pixel values are indexes into an external RGB colorTable. For non-colorTable bitmaps (greater than 256-colors), `FindClosestRGB()` is not applicable and will simply return the same `rgbColor` value as input.

mgBitmap_GetInfoPtr() - get pointer to BitmapInfo

C Syntax

mgbitmap.h

```
/* Get bitmap information pointer */
MGBITMAPINFO*  mgBitmap_GetInfoPtr(   /* return, ptr to bitmapInfo data   */
         MGBITMAP       bitmap,       /* input,  bitmap-handle            */
         int            structSize ); /* input,  MGBITMAPINFO struct size */
```

C++ Syntax

mgbitmap.hpp

```
// Get bitmap information pointer
MGBITMAPINFO*  mgCBitmap::GetInfoPtr( // return, pointer to bitmapInfo data
         int            structSize,   // input,  MGBITMAPINFO struct size
```

GetInfoPtr() returns a pointer to the bitmap information record for the bitmap instance. The returned bitmapInfo pointer can be used for "read-only" access to the bitmap information.

Parameters

MGBITMAP bitmap   (C, input)

Bitmap-handle.

int   structSize (input)

Size of the MGBITMAPINFO structure (see Comments, below).

Returns

GetInfoPtr() returns with a pointer to the bitmap's MGBITMAPINFO information record.

Comments

To provide compatibility with updated TypeServer versions running from DLL's, a structure size parameter is passed to verify and identify the structure version used in the client application. Updated DLL's may use updated structures that include new additional variables. The structSize parameter allows updated versions of this function to optionally support calls from existing applications that were compiled using older structure definitions. For more information, see the **Metagraphics Programming Guidelines** manual discussion on *"Enhancing 'struct' Compatibility".*

Example

```
    MGBITMAPINFO  *pBitmapInfo;      /* pointer to bitmapInfo */

    pBitmapInfo = mgBitmap_GetInfoPtr( sizeof(MGBITMAPINFO) );
    _ASSERT( pBitmapInfo != NULL );  /* debug check */
```

mgBitmap_MapColors() - remap pixels to match new colorTable

C Syntax

mgbitmap.h

```
/* set new colorTable values and remap pixel colors */
MRESULT  mgBitmap_MapColors(
         MGBITMAP     bitmap,        /* input,  bitmap instance handle  */
   const MGCOLORRGB  *rgbColors,     /* input,  new colorTable values   */
         int          startIndex,    /* input,  starting table index    */
         int          numEntries );  /* input,  number for values to set*/
```

C++ Syntax

mgbitmap.hpp

```
// set new colorTable values and remap pixel colors
MRESULT  mgCBitmap::MapColors(
   const MGCOLORRGB  *rgbColors,     // input,  new colorTable values
         int          startIndex,    // input,  starting table index
         int          numEntries );  // input,  number for values to set
```

Description

For 16- and 256-color bitmaps that use palette based colorTables, `MapColors()` copies one or more new RGB color definitions to the bitmap colorTable. Based on the values of the old color table, `MapColors()` additionally translates bitmap pixels to their closest matching colors in the new color table.

Parameters

`MGBITMAP    bitmap      (C, input)`

Bitmap-handle.

`MGCOLORRGB *rgbColors   (input)`

Pointer to the first member of an array of `MGCOLORRGB`'s defining the RGB color values to set in the bitmap colorTable.

`int         startIndex  (input)`

Specifies the first color table entry to set.

`int         numEntries  (input)`

Specifies the number of color table entries to set.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

ColorTables contain one additional "colorTable-signature" as their very last entry (16-color colorTables actually contain 17 elements, 256-color colorTables contain 257 elements). The "colorTable-signature" is a counter that is incremented each time a colorTable change is made by `MapColors()`. If needed, external functions that are optimized using precomputed variables (such as a color translation table) based on a given colorTable, can use the colorTable-signature to determine when a colorTable has changed and when associated precomputed variables need to be recalculated.

See Also

mgBitmap_SetColors(), mgBitmap_CreatePalette(), mgBitmap_CreateIdentityPalette()

mgBitmap_PixToRGB() - convert pixel value to RGB color

C Syntax

mgbitmap.h

```
/* convert pixel value to RGB color */
MGCOLORRGB  mgBitmap_PixToRGB(         /* return, RGB color            */
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle        */
         MGCOLORPIX     pixelValue );  /* input,  pixel value          */
```

C++ Syntax

mgbitmap.hpp

```
// convert pixel value to RGB color
MGCOLORRGB  mgCBitmap::PixToRGB(       // return, RGB color
         MGCOLORPIX     pixelValue );  // input,  pixel value
```

Description

PixToRGB() converts a pixel value in the current bitmap pixel format to a generic RGB color.

If the bitmap uses colorTable based pixel indexes (bitmaps with 256-colors or less), PixToRGB() returns the generic RGB color for the specified colorTable index.

If the bitmap uses RGB-encoded pixel values (bitmaps with more than 256-colors), PixToRGB() converts the format-specific RGB bitmap pixel value to a generic RGB color. For example if the bitmap in use is 16 bits-per-pixel, PixToRGB() will convert a 16-bit RGB pixel value to a generic RGB color.

Parameters

```
MGBITMAP     bitmapHandle   (C, input)
```

Bitmap-handle.

```
MGCOLORPIX   pixelValue     (input)
```

Bitmap pixel value to return the generic RGB color for.

Returns

PixToRGB() returns an MGCOLORRGB generic RGB color corresponding to the pixel value for the bitmap in use.

See Also

mgBitmap_RGBToPix(), RGB_Make(), MGCOLORPIX, MGCOLORRGB

mgBitmap_RGBToPix() - convert RGB color to pixel value

C Syntax
mgbitmap.h

```
/* convert RGB color to pixel value */
MGCOLORPIX  mgBitmap_RGBToPix(         /* return, pixel value       */
         MGBITMAP       bitmapHandle,  /* input,  bitmap-handle     */
         MGCOLORRGB     rgbColor );    /* input,  RGB color         */
```

C++ Syntax
mgbitmap.hpp

```
// convert RGB color to pixel value
MGCOLORPIX  mgCBitmap::RGBToPix(       // return, pixel value
         MGCOLORRGB     rgbColor );    // input,  RGB color
```

Description
`RGBToPix()` converts a generic RGB color to a format-specific pixel value for the bitmap in use.

If the bitmap uses indexed colors with an associated colorTable (bitmaps with 256-colors or less), `RGBToPix()` returns the pixel value for the colorTable index that most closely matches the specified RGB color. For 256-color bitmaps for example, the `RGBToPix()` will return a pixel value between 0 to 255 corresponding to the colorTable index that most closely matches the specified RGB color.

If the bitmap uses RGB-encoded pixel values (bitmaps with more than 256-colors), `RGBToPix()` converts the generic RGB color to the format-specific RGB pixel value for the bitmap in use. For example if the bitmap in use is 16 bits-per-pixel, `RGBToPix()` will encode the 24-bit generic RGB color into the appropriate 16-bit RGB pixel format for the bitmap in use.

Parameters
```
MGBITMAP     bitmapHandle   (C, input)
```
Bitmap-handle.

```
MGCOLORRGB  *rgbColor       (output)
```
Generic RGB color. (The `RGB_MAKE()` macro provides a simple means to define generic RGB colors.)

Returns
`RGBToPix()` returns an `MGCOLORPIX` bitmap-specific pixel value that most closely matches the RGB color specified.

See Also
`mgBitmap_RGBToPix(), RGB_Make(), MGCOLORPIX, MGCOLORRGB`

mgBitmap_SetColors() - set new colorTable

C Syntax
mgbitmap.h

```
MRESULT  mgBitmap_SetColors(
         MGBITMAP       bitmap,        /* input,  bitmap instance handle  */
   const MGCOLORRGB    *rgbColors,     /* input,  new colorTable values   */
         int            startIndex,    /* input,  starting table index    */
         int            numEntries );  /* input,  number for values to set*/
```

C++ Syntax
mgbitmap.hpp

```
// set new color table values
MRESULT  mgCBitmap::SetColors(
   const MGCOLORRGB    *rgbColors,     // input,  new colorTable values
         int            startIndex,    // input,  starting table index
         int            numEntries );  // input,  number for values to set
```

Description
`SetColors()` copies one or more new RGB color definitions to the bitmap colorTable. Unlike `MapColors()`, `SetColors()` loads the new colorTable values only, and does not change any of the pixels in the bitmap itself.

Parameters
`MGBITMAP    bitmap      (C, input)`

Bitmap-handle.

`MGCOLORRGB *rgbColors   (input)`

Pointer to the first member of an array of `MGCOLORRGB`'s defining the RGB color values to set in the bitmap colorTable.

`int         startIndex  (input)`

First colorTable entry to set.

`int         numEntries  (input)`

Number of colorTable entries to set.

Returns
An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments
`SetColors()` only affects the bitmap colorTable (the pixel data in the bitmap itself is not changed). `MapColors()` can be used to both set new colorTable values, and remap the pixel data to match the new colorTable settings.

ColorTables contain one additional "colorTable-signature" as their very last entry (16-color colorTables actually contain 17 elements, 256-color colorTables contain 257 elements). The "colorTable-signature" is a counter that is incremented any time a colorTable change is made by `SetColors()`. For optimization, external functions can use precomputed variables (such a color translation table) based on a given colorTable setting. The colorTable-signature is used to determine when the colorTable has changed, and when dependent precomputed variables need to be recalculated.

See Also

`mgBitmap_MapColors(), mgBitmap_CreatePalette(),`
`mgBitmap_CreateIdentityPalette()`

mgBitmap_SetRasterOp() - set rasterOp transfer mode

C Syntax

mgbitmap.h

```
/* Set rasterOp transfer mode */
MRESULT  mgBitmap_SetRasterOp(
         MGBITMAP       bitmap,        /* input,  bitmap instance handle  */
         MGRASTEROP     rasterOp );    /* input,  transfer mode to set    */
```

C++ Syntax

mgbitmap.hpp

```
// Set rasterOp transfer mode
MRESULT  mgCBitmap::SetRasterOp(
         MGRASTEROP     rasterOp );    // input,  transfer mode to set
```

Description

`SetRasterOp()` defines how pixels are combined when blitting to a destination bitmap. Four "standard" rasterOps, and four "transparent" rasterOps are defined in the `MGRASTEROP` enumeration:

```
    typedef enum
    {   ropCopy,              /* replace             */
        ropMerge,             /* OR                  */
        ropErase,             /* AND                 */
        ropInvert,            /* XOR                 */
        ropTransparentCopy,   /* transparent replace */
        ropTransparentMerge,  /* transparent OR      */
        ropTransparentErase,  /* transparent AND     */
        ropTransparentInvert  /* transparent XOR     */
    } MGRASTEROP;
```

For transparent rasterOps, calling `SetTransparentColor()` defines the transparency color. When copying images with a transparent rasterOp, only the non-transparent colors of the source image are transferred.

Parameters

`bitmap   (C, input)`

Bitmap-handle.

`rasterOp (input)`

Selected `MGRASTEROP` transfer mode (see above).

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

See Also

mgBitmap_SetTransparentColor(), mgBitmap_Blit()

mgBitmap_SetTopdown() - set bitmap format

C Syntax

mgbitmap.h

```
/* convert bitmap to top-down or bottom-up format */
MRESULT  mgBitmap_SetTopdown(
         MGBITMAP       bitmap,        /* input,  bitmap instance handle  */
         BOOL           topDown );     /* input,  topDn(TRUE)/botUp(FALSE)*/
```

C++ Syntax

mgbitmap.hpp

```
// convert bitmap to top-down or bottom-up format
MRESULT  mgCBitmap::SetTopdown(
         BOOL           topDown );     // input,  topDn(TRUE)/botUp(FALSE)
```

Description

SetTopdown() sets the format of the bitmap surface raster lines in memory.

Calling SetTopdown(TRUE) sets the bitmap to a top-down organization where the top Y=0 raster line is first in memory.   (This is the normal organization used by virtually all graphics hardware and 99.9% of the software in the world.)

Calling SetTopdown(FALSE) sets the bitmap to a bottom-up organization where the bottom raster is first in memory and the top Y=0 raster line is last in memory.   (This format is used by IBM OS/2, and occurs in rare instances under Microsoft Windows.)

Parameters

bitmap   (C, input)

TypeServer bitmap instance handle.

topDown (input)

TRUE, sets the bitmap to top-down organization (top Y=0 rasterline is first in memory).
FALSE, sets the bitmap to bottom-up organization (bottom rasterline is first in memory).

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

The bitmap memory orientation, top-down or bottom-up, does not affect the appearance of any graphics or how the bitmap appears when copied to the screen. The bitmap orientation only defines how the bitmap is stored in memory, and how it is accessed internally.

Metagraphics TypeServer supports and works equally well with either top-down or bottom-up bitmaps. Some Windows display drivers, however, work better with one orientation or the other. Certain third party libraries also may only be able to work with bitmaps in one format. `SetTopdown()` allows you to manually change the bitmap raster line memory organization if you desire.

mgBitmap_SetTranslate() - enable/disable color translation

C Syntax

mgbitmap.h

```
MRESULT  mgBitmap_SetTranslate(
         MGBITMAP       bitmap,        /* input,  bitmap instance handle  */
         BOOL           translate );   /* input,  translate=TRUE          */
```

C++ Syntax

mgbitmap.hpp

```
// enable color translation to destination bitmap
MRESULT  mgCBitmap::SetTranslate(
         BOOL           translate );   // input,  translate=TRUE
```

Description

For 16- and 256-color bitmaps using RGB colorTables, `SetTranslate()` enables or disables color translated blits. When active, blits to a destination bitmap first check the colorTables of both bitmaps. If the RGB colorTables of both bitmaps match, blit transfers to the destination bitmap proceed directly. If the RGB colorTables do not match, a pixel translation table is computed and blits to the destination are performed using pixel translated blits.

Parameters

`bitmap    (C, input)`

bitmap-handle.

`translate (input)`

`TRUE`, enables color-translated blits if colorTables do not match.
`FALSE`, disables color-translated blits.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

`SetTranslate(TRUE)` simply enables translated blits and colorTable match checking. If the RGB entries of both colorTables match, faster non-translated blits are used. Only when the two colorTables differ are translated blits performed. The `transActive` field of the `MGBITMAPINFO` structure indicates when translation is actually active.

`SetTranslate(FALSE)` disables translated blits and colorTable checking. If the colorTables of the source and destination bitmaps are not the same, blitted images will appear with the wrong colors in the destination bitmap. Translation is enabled by default, and should only be disabled for special cases.

See Also

`mgBitmap_ComputeTranslate()`

mgBitmap_SetTransparentColor() - set transparent pixel value

C Syntax

mgbitmap.h

```
/* set transparent color */
MRESULT  mgBitmap_SetTransparentColor(
         MGBITMAP       bitmap,        /* input,  bitmap instance handle  */
         MGCOLORPIX     xparColor );   /* input,  transparent color       */
```

C++ Syntax

mgbitmap.hpp

```
// set transparent color
MRESULT  mgCBitmap::SetTransparentColor(
         MGCOLORPIX     xparColor );   // input,  transparent color
```

Description

`SetTransparentColor()` defines the "transparent" pixel color for the bitmap. For transparent drawing operations, bitmap pixels matching the specified `xparColor` value are not copied to the destination bitmap.

Parameters

`MGBITMAP     bitmap    (C, input)`

Bitmap-handle.

`MGCOLORPIX   xparColor (input)`

Transparent color value (index for 16- and 256-color bitmaps, RGB value for 16- and 24-bit per pixel bitmaps).

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

`SetTransparentColor()` sets the `MGBITMAPINFO.xparColor` field equal to the specified value. (The default `xparColor` is black (0).)

See Also

`mgBitmap_SetTransparent(), mgBitmap_Blit()`

TypeServer Functions

A TSSERVER instance (C) or tsCServer class object (C++) defines a Metagraphics TypeServer for processing scalable TrueType fonts into bitmap pixel images. When opening a TypeServer, the create call returns a handle to a server instance (C) or class object (C++) that can then be used in subsequent TypeServer function calls. Normally a single application needs to create only one TypeServer. In special operating system uses, a separate server may be created for each independent task. TypeServer contains no static data variables, and each server instance maintains its own local state information. Using this design, a single TypeServer DLL can support multiple server instances in a reentrant and thread safe fashion.

Functions

At the highest level, there are two functions associated with creating and terminating a TypeServer instance:

C Functions

```
tsServer_Create()   /* create & initialize a new TypeServer instance */
tsServer_Destroy()  /* release & delete a TypeServer instance        */
```

C++ Methods

```
tsServer::tsCServer()    // TypeServer constructor
tsServer::~tsCServer()   // TypeServer destructor
```

Prototypes

Following is a summary of the TypeServer function prototypes defined in the typeserv.h (for C) and typeserv.hpp (for C++) header files. The information in the header files may include updated information that includes additions or changes that supercede the printed document. Please review the associated header files for current specifications.

C Prototypes **typeserv.h**

```
/* create and initialize a TypeServer instance */
MRESULT tsServer_Create(
        MGSYSTEM *mgSystem,          /* in/out, system-handle */
        TSSERVER *tsServer );        /* output, server-handle */

/* release and delete a TypeServer instance    */
MRESULT tsServer_Destroy(
        MGSYSTEM *mgSystem,          /* in/out, system-handle */
        TSSERVER *tsServer );        /* in/out, server-handle */
```

C++ Prototypes **typeserv.hpp**

```
class tsCServer : virtual public tsCObject
{
  public:  // Public Methods
```

```
    // constructor – create and initialize a TypeServer instance
    tsCServer(
        mgCSystem  *mgSystem );    // in/out, system instance

    // copy constructor
    tsCServer( const tsCServer& );

    // assignment operator
    tsCServer& operator=( const tsCServer& );

    // destructor - release and delete a TypeServer instance
    ~tsCServer(
        mgCSystem  *mgSystem );    // in/out, system instance

};  // class tsCServer
```

Function Descriptions

The following section provides descriptions for the main TypeServer create and delete functions.

tsServer_Create() - create a TypeServer instance

C Syntax

typeserv.h

```
/* create and initialize a new TypeServer instance */
MRESULT tsServer_Create(
        MGSYSTEM   *mgSystem,        /* in/out, system-handle */
        TSSERVER   *tsServer );      /* output, server-handle */
```

C++ Syntax

typeserv.hpp

```
// constructor – create and initialize a TypeServer instance
tsCServer::tsCServer(
        mgCSystem  *mgSystem );     // in/out, system instance
```

Description

`tsServer_Create()` C function and `tsCServer()` C++ constructor creates and initializes a new Metagraphics TypeServer instance. Once a TypeServer has been created, the client application can then select and open TrueType fonts by calling `tsFont_OpenFile()` or `tsFont_OpenMemory()` (C), or the `tsCFont()` constructor (C++).

While it is possible to open multiple server instances, normally a single application needs only one server for all rendering. In a multi-threaded environment, individual threads may each open their own servers if the threads execute independently.

C Parameters

```
MGSYSTEM   *mgSystem    (C, in/out)
```

Pointer to an MGSYSTEM handle where either an existing system-instance handle is stored, or where a new system-instance handle is to be returned. When declaring this variable, your application should initialize it to a value of NULL (see **Comments**, below).

```
mgCSystem  *mgSystem    (C++, in/out)
```

Pointer to an mgCSystem class where either the instance pointer of an existing system-instance is stored, or where a new system-instance pointer is to be returned. When declaring this variable, your application should initialize it to a value of NULL (see **Comments**, below).

```
TSSERVER   *tsServer    (C, output)
```

Pointer to a TSSERVER variable where the instance handle for the new TypeServer is to be returned. When declaring this variable, your application should initialize it to a value of NULL (see **Comments**, below).

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Example

When using TypeServer by itself, tsServer_Create() creates and initializes both the system and server instances (see the MGSYSTEM note below for information when using TypeServer with multiple Metagraphics products):

```
MGSYSTEM      mgSystem=NULL;   /* Metagraphics system-handle   */
TSSERVER      tsServer=NULL;   /* TypeServer server-handle     */

/* create both MGSYSTEM and TSSERVER */
result = tsServer_Create( &mgSystem, &tsServer );
if ( FAILED(result) )
   /* perform error handling */

/*   :
 *   :
 */

/* destroy TSSERVER and MGSYSTEM instances */
result = tsServer_Destroy( &mgSystem, &tsServer );
```

Comments

As illustrated above, the MGSYSTEM and TSSERVER handles should be initialized to a value of NULL before calling tsServer_Create(). This convention is used to insure that a second tsServer_Create() call does not overwrite a previously opened server-handle before tsServer_Destroy() has been called:

```
/* declare MGSYSTEM & TSSERVER handles and initialize to NULL */
MGSYSTEM  mgSystem=NULL;
TSSERVER  tsServer=NULL;
```

All TypeServer instances created by `tsServer_Create()` should be released by your application when no longer needed (and before program termination) by calling `tsServer_Destroy()`.

Upon successful return, the returned server-handle can be used in calling the following functions:

```
tsServer_Destroy()
tsFont_OpenFile()
tsFont_OpenMemory()
```

MGSYSTEM - Using TypeServer with other Metagraphics Products

As noted in the function prototype, `tsServer_Create()` takes pointers to both a TypeServer "server-instance" handle and a Metagraphics "system-instance" handle. The Metagraphics "system-instance" provides a set of common services shared by all Metagraphics products. The system-instance, "`MGSYSTEM`" (C) and system class "`mgCSystem`" (C++) provide a centralized implementation for memory management, file I/O, mutex access and other basic system services that are used by Metagraphics products. If you use multiple Metagraphics products in a single application, the first creation method will create a Metagraphics system-instance that is then shared and used by the other Metagraphics product functions.

For example, if you are using TypeServer together with Metagraphics MetaWINDOW v6, the opening call to `mwGraphics_Create()` will create both a Metagraphics system-instance and MetaWINDOW graphics-instance (remember that all handles must first be initialized to `NULL`). Later when the system-instance handle is a passed to `tsServer_Create()`, `tsServer_Create()` will create only a new TypeServer "server-instance" since the system-instance handle has been previously defined by `mwGraphics_Create()` and is no longer `NULL`:

```
MGSYSTEM      mgSystem=NULL;   /* Metagraphics system-handle   */
MWGRAPHICS    mwGraphics=NULL; /* MetaWINDOW graphics-handle   */
TSSERVER      tsServer=NULL;   /* TypeServer server-handle     */

/* create both MGSYSTEM and MWGRAPHICS */
result = mwGraphics_Create( &mgSystem, &mwGraphics );
if ( FAILED(result) )
   /* perform error handling */

/* create TSSERVER only (MGSYSTEM now already exists) */
result = tsServer_Create( &mgSystem, &tsServer );
if ( FAILED(result) )
   /* perform error handling */

/*
 * When using multiple Metagraphics products, destructors
 * need to be called in their reverse nested order:
 */

/* destroy the TSSERVER instance */
result = tsServer_Destroy( &mgSystem, &tsServer );

/* destroy the MWGRAPHICS instance and MGSYSTEM instance */
result = mwGraphics_Destroy( &mgSystem, &mwGraphics );
```

See Also

`tsServer_Destroy(), tsFont_OpenFile(), tsFont_OpenMemory()`

tsServer_Destroy() - destroy a TypeServer instance

C Syntax

typeserv.h

```
/* release and delete a TypeServer instance */
MRESULT tsServer_Destroy(
        MGSYSTEM   *mgSystem,    /* in/out, system-handle */
        TSSERVER   *tsServer );  /* in/out, server-handle */
```

C++ Syntax

typeserv.hpp

```
// release and delete a TypeServer instance
tsCServer::~tsCServer(
        mgCSystem  *mgSystem );  // in/out, system-instance
```

Description

`tsServer_Destroy()` closes the TypeServer previously initialized by `tsServer_Create()`. Upon closing, all associated resources previously created and used by the server are released.

Parameters

`MGSYSTEM   *mgSystem    (C,   in/out)`

Pointer to the `MGSYSTEM` handle that was originally passed to `tsServer_Create()`. If `tsServer_Create()` initialized this handle, the close process will destroy the system-instance and reset this handle to `NULL`.

`mgCSystem  *mgSystem    (C++, in/out)`

Pointer to the `mgCSystem` instance that was originally passed to `tsCServer()`. If `tsCServer()` initialized this instance, the `~tsCServer()` destructor will destroy the system instance and reset this pointer to `NULL`.

`TSSERVER   *tsServer    (C,   in/out)`

Pointer to the `TSSERVER` handle that was initially returned by `tsServer_Create()`. As part of the close process this server-handle is reset to `NULL`.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

See Also

`tsServer_Create()`

Font Functions

Once a TypeServer has been created and opened, the application can choose and open fonts for rendering. For applications that require displaying a mix of typefaces, virtually any number of fonts can be opened for rendering at one time (given adequate memory). Performance overhead may become a factor if fonts must be repeatedly opened and closed in memory limited environments. Metagraphics TypeServer allows you to work with TrueType fonts stored either as files on disk, or as preloaded files stored in memory or ROM.

Functions

There are four basic font functions: two for opening a TrueType font either from disk or memory, a font info, and a single function to close a font:

C Functions

```
    tsFont_OpenFile()          /* open a TrueType font file       */
    tsFont_OpenMemory()        /* open a TrueType font in memory  */
    tsFont_GetFontInfoPtr()    /* get pointer to font information */
    tsFont_Destroy()           /* close a TrueType font           */
```

C++ Functions

```
    tsCFont::tsCFont()        // TrueType font file constructor
    tsCFont::tsCFont()        // TrueType font memory constructor
    tsCFont::GetFontInfoPtr() // get pointer to font information
    tsCFont::~tsCFont()       // TrueType font destructor
```

Prototypes

Following is a summary of the TypeServer function prototypes defined in the `typeserv.h` header file. The information in the `typeserv.h` header file may include updated information with additions or changes that supercede the printed document. Please review `typeserv.h` for current specifications.

C Prototypes **typeserv.h**

```
/* open a TrueType font file   */
MRESULT tsFont_OpenFile(
        TSSERVER     server,        /* input,  server-handle            */
  const TCHAR       *filePathName,  /* input,  font path and file name  */
        LONG         fileOffset,    /* input,  offset to start of font  */
        int          fontNumber,    /* input,  font number within file  */
        TSFONT      *font );        /* output, font-handle              */

/* Open a TrueType font in memory */
MRESULT tsFont_OpenMemory(
        TSSERVER     server,        /* input,  server-handle            */
        void        *fontMemory,    /* input,  font memory address      */
        LONG         fontSize,      /* input,  font memory size         */
        int          fontNumber,    /* input,  font number within file  */
        TSFONT      *font );        /* output, font-handle              */
```

```
/* Get pointer to font information */
TSFONTINFO*  tsFont_GetInfoPtr(    /* return, pointer to fontInfo struct */
         TSFONT       font,        /* input,  font-handle                */
         int          structSize );/* input,  TSFONTINFO struct size     */

/* release and close a TrueType font */
MRESULT tsFont_Destroy(
         TSFONT      *font );      /* input, font instance to close    */
```

C++ Prototypes typeserv.h

Following is a summary of the TypeServer font function prototypes defined in the typeserv.hpp header file. The information in the typeserv.hpp header file may include updated information with additions or changes that supercede the printed document. Please review typeserv.hpp for current specifications.

```
class tsCFont : virtual public tsObject
{
  public:  // Public Methods

    // constructor, open a TrueType font file
    tsCFont(
        tsCServer   *server,        // input,  server instance
  const TCHAR       *filePathName,  // input,  font path and file name
        LONG         fileOffset=0,  // input,  offset to start of font
        int          fontNumber=0 );// input,  font number within file

    // constructor, font in memory
    tsCFont(
        tsCServer   *server,        // input,  server instance
        void        *fontMemory,    // input,  font memory address
        LONG        *fontSize,      // input,  font memory size
        LONG         fontNumber=0 );// input,  font number within file

    // copy constructor
    tsCFont( const tsCFont& );

    // assignment operator
    tsCFont& operator=( const tsCFont& );

    // destructor
    ~tsCFont();

    // Get font information pointer
    TSFONTINFO*  GetInfoPtr(         // return, pointer to fontInfo struct
         int          structSize );  // input,  TSFONTINFO struct size

};  // class tsCFont
```

Function Descriptions

The following section provides descriptions for the TrueType font open, close and information functions.

tsFont_OpenFile() - open a TrueType font disk file

C Syntax typeserv.h

```
/* open a TrueType font file   */
MRESULT tsFont_OpenFile(
        TSSERVER   server,          /* input,  server-handle            */
  const TCHAR     *filePathName,    /* input,  font path and file name  */
        LONG       fileOffset,      /* input,  offset to start of font  */
        int        fontNumber,      /* input,  font number within file  */
        TSFONT    *font );          /* output, font-handle              */
```

C++ Syntax typeserv.hpp

```
// constructor, open a TrueType font file
tsCFont::tsCFont(
        tsCServer *server,          // input,  server instance
  const TCHAR     *filePathName,    // input,  font path and file name
        LONG       fileOffset=0,    // input,  offset to start of font
        int        fontNumber=0 );  // input,  font number within file
```

Description

`tsFont_OpenFile()` opens a specified TrueType font file (default, `.ttf`) located on disk. (Use `tsFont_OpenMemory()` to open fonts that have been preloaded into memory or ROM.) The open function returns a font-handle that can be used to reference the font with other TypeServer functions. Once a font file has been opened you can then create a font strike with attributes for rendering (see `tsStrike_Create()`).

Parameters

`TSSERVER   server         (C, input)`

Server-handle for the TypeServer instance to use for rendering this font (this TypeServer handle must have been previously created by `tsServer_Create()`).

`tsCServer *server         (C++, input)`

Pointer to the TypeServer class instance to use for rendering this font (this TypeServer class object must have been previously created by the `tsServer()` constructor.

`TCHAR     *filePathName  (input)`

Path and file name for the TrueType font to be opened.

`LONG        fileOffset    (input)`

If the font is contained within a larger resource file that contains multiple files packed into one, `fileOffset` specifies the byte offset from the start of the resource file to where the font file begins. For standard .ttf fonts that are not packed into a multi-file resource, `fileOffset` is 0.

`int         fontNumber    (input)`

TrueType permits multiple fonts to be combined into a single file. `fontNumber` specifies the specific font within the TrueType file to be opened. The first font within the file is number 0.

Multiple fonts within a single TrueType file is *not* the typical case. You can always open a

TrueType font file with index 0. Once open, you can call `GetFontInfoPtr()` to return information about the number of fonts in the file. If a different font is needed, close the original font, and then reopen the font file with a selected index. Specifying an index greater than the number of fonts in the file will return an `MRESULT` "failure" error.

```
TSFONT    *font          (C, output)
```

Pointer to a `TSFONT` variable where the instance-handle for the opened font is to be returned. When declaring this variable, your application should initialize all handles to `NULL` (see **Comments**, below).

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

For C, the font instance handle should be initialized to a value of `NULL` before calling `OpenFile()`. Calling `OpenFile()` with a non-`NULL` handle will result in an `MRESULT` warning. This convention is used to insure that a second `tsFont_Open...()` call does not overwrite a previously opened font before `tsFont_Destroy()` has been called:

```
    /* declare a font-handle & initialize it to NULL */
    TSFONT  myFont = NULL;
```

All font instances created by `tsFont_OpenFile()` should be released by your application when no longer needed (and before program termination) by calling `tsFont_Destroy()`.

Upon successful open, the returned font-handle can be used in calling the following functions:

```
     tsFont_Destroy()
     tsFont_GetFontInfo()
     tsStrike_Create()
```

See Also

`tsFont_Destroy(), tsFont_OpenMemory(), tsFont_GetFontInfo(), tsStrike_Create()`

tsFont_OpenMemory() - open a memory resident TrueType font

C Syntax

typeserv.h

```
/* open a TrueType font in memory */
MRESULT tsFont_OpenMemory(
        TSSERVER     server,        /* input,  server-handle            */
        void        *fontMemory,    /* input,  font memory address      */
        LONG        *fontSize,      /* input,  font memory size         */
        int          fontNumber,    /* input,  font number within file  */
        TSFONT      *font );        /* output, font-handle              */
```

C++ Syntax

typeserv.hpp

```
// constructor, font in memory
tsCFont::tsCFont(
        tsCServer   *server,        // input,  server instance
        void        *fontMemory,    // input,  font memory address
        LONG        *fontSize,      // input,  font memory size
        int          fontNumber=0 );// input,  font number within file
```

Description

`tsFont_OpenMemory()` opens a TrueType font that has been preloaded into memory or is stored in ROM. `tsFont_OpenMemory()` returns a font-handle that can then be used to reference the font with other TypeServer functions. Once a font file has been opened you can then create a font strike with attributes for rendering (see `tsStrike_Create()`).

Parameters

`TSSERVER     server      (C, input)`

Server-handle to use for rendering this font (this TypeServer handle must have been previously created by `tsServer_Create()`).

`tsCServer   *server     (C++, input)`

Pointer to the TypeServer class instance to use for rendering this font (this TypeServer class object must have been previously created by the `tsCServer()` constructor.

`void        *fontMemory (input)`

Specifies the starting address where the TrueType font is located in memory.

`LONG         fontSize    (input)`

Specifies the size of the buffer where the font is stored in memory.

`int          fontNumber  (input)`

TrueType permits multiple fonts to be combined into a single file. `fontNumber` specifies the specific font within file to be used. The first font within the file is number 0.

Multiple fonts within a single file is *not* the typical case. You can always open a TrueType font file with index 0. Once open, you can call `GetInfoPtr()` to return information about the number of fonts in the file. If a different font is desired, close the original font, and then reopen the font file with a selected index. Specifying a `fontNumber` greater than the number of fonts in the file will return a "failed" `MRESULT` value.

`*font        (C, output)`

Pointer to a `TSFONT` variable where the instance handle for the opened font is to be returned. When declaring this handle, your application should initialize it to a value of `NULL` (see **Comments**, below).

Returns

For C, an `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

The font instance handle should be initialized to a value of `NULL` before calling `OpenMemory()`. Calling `OpenMemory()` with a non-`NULL` font-handle will result in an `MRESULT` warning. This convention is used to insure that a second `tsFont_Open()` call does not overwrite a previously opened font-handle before `tsFont_Destroy()` has been called:

```
    /* declare a font-handle & initialize it to NULL */
    TSFONT  myFont = NULL;
```

All font instances created by `OpenFile()` should be released by your application when no longer needed (and before program termination) by calling `tsFont_Destroy()` (C) or the `~tsCFont()` destructor (C++).

Upon successful open, the returned font-handle can be used in calling the following functions:

```
     tsFont_Destroy()
     tsFont_GetFontInfo()
     tsStrike_Create()
```

See `tsFont_OpenFile()` for opening fonts that are stored on disk.

See Also

`tsFont_Destroy(), tsFont_OpenFile(), tsFont_GetFontInfo(), tsStrike_Create()`

tsFont_Destroy() - destroy a font instance

C Syntax

typeserv.h

```
/* release and close a TrueType font */
MRESULT tsFont_Destroy(
        TSFONT   *font );   /* in/out, handle of font close */
```

C++ Syntax

typeserv.hpp

```
// release and close a TrueType font
tsCFont::~tsCFont();       // destructor
```

Description

`tsFont_Destroy()` closes the font previously initialized by `tsFont_OpenFile()`or `tsFont_OpenMemory()`. Closing releases all resources previously created and used by the font.

Parameters

`*font  (C, input/output)`

Pointer to the font-handle that was initially set by `tsFont_OpenFile()`or `tsFont_OpenMemory()`. As part of the close process, the font-handle is reset to a value of `NULL`.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

See Also

`tsFont_OpenFile(), tsFont_OpenMemory()`

tsFont_GetInfoPtr() - get pointer to fontInfo

C Syntax

typeserv.h

```
/* Get font information pointer */
TSFONTINFO* tsFont_GetInfoPtr(   /* return, pointer to fontInfo struct */
    TSFONT       font,           /* input,  font-handle                */
    int          fontInfoSize ); /* input,  TSFONTINFO struct size     */
```

C++ Syntax

typeserv.hpp

```
// get font information pointer
TSFONTINFO* tsCFont::GetInfoPtr( // return, pointer to fontInfo struct
    int          fontInfoSize ); // input,  TSFONTINFO struct size
```

Description

`GetInfoPtr()` returns a pointer to the font information record for the specified font instance. The returned `fontInfo` pointer can be used for "read-only" access to the font information.

Parameters

`TSFONT       font         (C, input)`

Instance handle for the font that was previously opened with `tsFont_OpenFile()` or `tsFont_OpenMemory()`.

```
int           fontInfoSize (input)
```

Size of the `TSFONTINFO` structure (see **Comments**, below).

Returns

`GetInfoPtr()` returns with a pointer to the bitmap's `TSFONTINFO` information record.

Comments

To provide compatibility with updated TypeServer versions running from DLL's, a structure size parameter is passed to verify and identify the structure version used in the client application. Updated DLL's may use updated structures that include new additional variables. The `structSize` parameter allows updated versions of this function to optionally support calls from existing applications that were compiled using older structure definitions. For more information, see the **Metagraphics Programming Guidelines** manual discussion on *"Enhancing 'struct' Compatibility".*

Example

```
    TSFONTINFO  *pFontInfo;      /* pointer to fontInfo */

    pFontInfo = tsFont_GetInfoPtr( sizeof(TSFONTINFO) );
    _ASSERT( pFontInfo != NULL );  /* debug check */
```

See Also

`tsFont_OpenFile(), tsFont_OpenMemory(), TSFONTINFO`

Strike Functions

A font "strike" defines the rendering attributes used by TypeServer to produce bitmap text and character images from TrueType fonts. These attributes may be set by the client application to define the size and appearance of the typeface to be drawn. Strike functions can be divided into four groups:

- Strike creation and deletion.
- Strike state information
- Strike attribute control
- Strike rendering

Creation and Deletion

The client application creates and deletes a strike by calling the associated TypeServer functions:

C Functions

```
    tsStrike_Create()          /* create a font strike    */
    tsStrike_Destroy()         /* delete a font strike    */
```

C++ Functions

```
    tsCStrike::tsCStrike()     // strike constructor
    tsCStrike::~tsCStrike()    // strike destructor
```

Strike Information

A font strike defines attribute control information that can be set for rendering a TrueType font. A default set of attributes are initialized when a strike is first created. The client application may read the current strike attribute settings at any time using the following TypeServer functions:

C Functions

```
    tsStrike_GetInfoPtr()      /* get pointer to strikeInfo            */
```

C++ Functions

```
    tsCStrike::GetInfoPtr()    // get pointer to strikeInfo
```

Strike Attributes

Font strike attributes define settings that control how characters are drawn to the destination bitmap. The following table summarizes the strike attributes, callable attribute functions, and the default values the attributes are initialized to when a strike is first created.

Strike Attribute	Function(s)	Default
Character Size	`tsStrike_SetTypeSize()`	10.0 points
Character Alignment	`tsStrike_SetAlign()`	`leftBaseline`

Justify Char-Spacing	`tsStrike_SetJustify()`	0.0
Justify Word-Spacing	`tsStrike_SetJustify()`	0.0
Line Spacing	`tsStrike_SetLineSpacing()`	12.0 points
Background Color	`tsStrike_SetColors()`	White
Character Color	`tsStrike_SetColors()`	Black
Transfer Mode	`tsStrike_SetRasterOp()`	`tsCOPY`
Edge Smoothing	`tsStrike_SetSmoothing()`	`tsSMOOTH16`

Type Dimensions & Rendering

The following TypeServer functions are called to compute the dimensions of a text string, and to perform the rendering of TrueType characters to a target bitmap:

C Functions

```
/* ASCII/Unicode functions (TCHAR, 8 or 16-bit dependent on "_UNICODE")*/
    tsStrike_DrawChar()           /* render a single character         */
    tsStrike_DrawString()         /* render a string of characters     */
    tsStrike_GetCharExtent()      /* get the dimensions of a character */
    tsStrike_GetStringExtent()    /* get the dimensions of a string    */

/* ASCII-specific functions (CHAR, 8-bit characters) */
    tsStrike_DrawCharA()          /* render a single character         */
    tsStrike_DrawStringA()        /* render a string of characters     */
    tsStrike_GetCharExtentA()     /* get the dimensions of a character */
    tsStrike_GetStringExtentA()   /* get the dimensions of a string    */

/* Unicode-specific functions (WCHAR, 16-bit characters) */
    tsStrike_DrawCharW()          /* render a single character         */
    tsStrike_DrawStringW()        /* render a string of characters     */
    tsStrike_GetCharExtentW()     /* get the dimensions of a character */
    tsStrike_GetStringExtentW()   /* get the dimensions of a string    */
```

C++ Functions

```
// ASCII/Unicode functions (TCHAR, 8 or 16-bit dependent on "_UNICODE")
    tsCStrike::DrawChar()         // render a single character
    tsCStrike::DrawString()       // render a string of characters
    tsCStrike::GetCharExtent()    // get the dimensions of a character
    tsCStrike::GetStringExtent()  // get the dimensions of a string

// ASCII-specific functions (CHAR, 8-bit characters)
    tsCStrike::DrawCharA()        // render a single character
    tsCStrike::DrawStringA()      // render a string of characters
    tsCStrike::GetCharExtentA()   // get the dimensions of a character
    tsCStrike::GetStringExtentA() // get the dimensions of a string
```

```
  // Unicode-specific functions (WCHAR, 16-bit characters)
      tsCStrike::DrawCharW()        // render a single character
      tsCStrike::DrawStringW()      // render a string of characters
      tsCStrike::GetCharExtentW()   // get the dimensions of a character
      tsCStrike::GetStringExtentW() // get the dimensions of a string
```

Prototypes

C Prototypes

typeserv.h

```
/* - - - - - Begin tsStrike functions - - - - - */

/* create a font strike instance*/
MRESULT tsStrike_Create(
        TSFONT      font,           /* input,  font-handle               */
        MGBITMAP    bitmap          /* input,  bitmap-handle             */
        TSSTRIKE   *strike );       /* output, new strike-handle         */

/* release & close a font strike */
MRESULT tsStrike_Destroy(
        TSSTRIKE   *strike );       /* in/out, strike-handle to close    */

/* draw a ASCII/Unicode (TCHAR) character */
MRESULT tsStrike_DrawChar(
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate   */
  const TCHAR       character );    /* input,  character to draw         */

/* draw an ASCII character (CHAR) */
MRESULT tsStrike_DrawCharA(
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate   */
  const CHAR        character );    /* input,  ASCII character to draw   */

/* draw a Unicode character (WCHAR) */
MRESULT tsStrike_DrawCharW(
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate   */
  const WCHAR       character );    /* input,  Unicode character to draw */

/* draw a ASCII/Unicode (TCHAR) string */
MRESULT tsStrike_DrawString(
        TSSTRIKE     strike,        /* input,  strike-handle             */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate   */
  const TCHAR       *string );      /* input,  string to draw            */

/* draw an ASCII string (CHAR) */
MRESULT tsStrike_DrawStringA(
        TSSTRIKE     strike,        /* input,  strike-handle             */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate   */
  const CHAR        *string );      /* input,  ASCII string to draw      */
```

```
/* draw a Unicode string (WCHAR) */
MRESULT tsStrike_DrawStringW(
        TSSTRIKE     strike,        /* input,  strike-handle             */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate   */
  const WCHAR       *string );      /* input,  Unicode string to draw    */

/* get the dimensions of a ASCII/Unicode (TCHAR) character */
MRESULT tsStrike_GetCharExtent(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const TCHAR        character,     /* input,  character                 */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of an ASCII (CHAR) character */
MRESULT tsStrike_GetCharExtentA(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const CHAR         character,     /* input,  ASCII character           */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of a Unicode (WCHAR) character */
MRESULT tsStrike_GetCharExtentW(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const WCHAR        character,     /* input,  Unicode character         */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of a ASCII/Unicode string (TCHAR, "_UNICODE") */
MRESULT tsStrike_GetStringExtent(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const TCHAR       *string,        /* input,  text string               */
        FIXSIZE     *extent );      /* output, string width & height     */

/* get the dimensions of an ASCII string (CHAR) */
MRESULT tsStrike_GetStringExtentA(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const CHAR        *string,        /* input,  ASCII text string         */
        FIXSIZE     *extent );      /* output, string width & height     */

/* get the dimensions of a Unicode string (WCHAR) */
MRESULT tsStrike_GetStringExtentA(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const WCHAR       *string,        /* input,  Unicode text string       */
        FIXSIZE     *extent );      /* output, string width & height     */

/* get pointer to strikeInfo */
TSSTRIKEINFO* tsStrike_GetInfoPtr(  /* return, pointer to strikeInfo     */
        TSSTRIKE       strike,      /* input,  strike-handle             */
        int            infoSize );  /* input,  TSSTRIKEINFO struct size  */

/* set the text alignment point */
MRESULT tsStrike_SetAlign(
        TSSTRIKE     strike,        /* input,  strike-handle             */
        TSALIGN      align );       /* input,  alignment position        */
```

```
/* set character and background colors by pixel value */
MRESULT tsStrike_SetColors(         /* return, completion code(0=success)*/
        TSSTRIKE     strike,        /* input,  strike-handle             */
        MGCOLORPIX   pixChar,       /* input,  character pixel value     */
        MGCOLORPIX   pixBack );     /* input,  background pixel value    */

/* set character and background colors by RGB value */
MRESULT tsStrike_SetColorsRGB(      /* return, completion code(0=success)*/
        TSSTRIKE     strike,        /* input,  strike-handle             */
        MGCOLORRGB   rgbChar,       /* input,  character RGB color       */
        MGCOLORRGB   rgbBack );     /* input,  background RGB color      */

/* set justification spacing */
MRESULT tsStrike_SetJustify(        /* return, completion code(0=success)*/
        TSSTRIKE     strike,        /* input,  strike-handle             */
        FIXDOT       charExtra,     /* input,  charExtra spacing         */
        FIXDOT       wordExtra,     /* input,  wordExtra spacing         */
        TSUNITS      units );       /* input,  units of measure          */

/* set the transfer mode rasterOp */
MRESULT tsStrike_SetRasterOp(       /* return, completion code(0=success)*/
        TSSTRIKE     strike,        /* input,  strike-handle             */
        TSRASTEROP   rasterOp );    /* input,  bitmap transfer mode      */

/* set anti-alias edge smoothing */
MRESULT tsStrike_SetSmoothing(      /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle             */
        TSSMOOTH    smoothLevel );  /* input,  smoothing level           */

/* set the type size */
MRESULT tsStrike_SetTypeSize(       /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXDOT      charHeight,     /* input,  character height          */
        TSUNITS     units );        /* input,  units of measure          */

/* - - - - - End tsStrike functions - - - - - */
```

C++ Prototypes typeserv.h

```
class tsCStrike : virtual public tsCObject
{
  public:  // Public Methods

    // constructor
    tsCStrike(
            tsCFont    *font,         // input,  TrueType font
            mgCBitmap  *bitmap );     // input,  bitmap to render to

    // copy constructor
    tsCStrike( const tsCStrike& );

    // assignment operator
    tsCStrike& operator=( const tsCStrike& );
```

```
// destructor
~tsCStrike();

// draw a ASCII/Unicode (TCHAR) character
MRESULT DrawChar(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const TCHAR       character );  // input,  character to draw

// draw an ASCII character (CHAR)
MRESULT DrawCharA(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const CHAR        character );  // input,  ASCII character to draw

// draw an Unicode character (WCHAR)
MRESULT DrawCharW(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const WCHAR       character );  // input,  Unicode character to draw

// draw a ASCII/Unicode (TCHAR) string
MRESULT DrawString(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const TCHAR      *string );     // input,  string to draw

// draw an ASCII string (CHAR)
MRESULT DrawStringA(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const CHAR       *string );     // input,  ASCII string to draw

// draw a Unicode string (WCHAR)
MRESULT DrawStringA(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const WCHAR      *string );     // input,  Unicode string to draw

// get the dimensions of a ASCII/Unicode (TCHAR) character
MRESULT GetCharExtent(
  const TCHAR       character,    // input,  character
        FIXSIZE    *extent );     // output, character width & height

// get the dimensions of an ASCII (CHAR) character
MRESULT GetCharExtentA(
  const CHAR        character,    // input,  ASCII character
        FIXSIZE    *extent );     // output, character width & height

// get the dimensions of a Unicode (WCHAR) character
MRESULT GetCharExtentA(
  const WCHAR       character,    // input,  Unicode character
        FIXSIZE    *extent );     // output, character width & height

// get the dimensions of a ASCII/Unicode (TCHAR) string
MRESULT GetStringExtent(
  const TCHAR      *string,       // input,  string
        FIXSIZE    *extent );     // output, string width & height
```

```
    // get the dimensions of an ASCII (CHAR) string
    MRESULT GetStringExtentA(
      const CHAR       *string,       // input,  ASCII string
            FIXSIZE    *extent );     // output, string width & height

    // get the dimensions of a Unicode (WCHAR) string
    MRESULT GetStringExtentW(
      const WCHAR      *string,       // input,  Unicode string
            FIXSIZE    *extent );     // output, string width & height

    // get pointer to strikeInfo
    TSSTRIKEINFO* GetInfoPtr(           // return, pointer to strikeInfo
            int            infoSize );  // input,  TSSTRIKEINFO struct size

    // set the text alignment position
    MRESULT SetAlign(
            TSALIGN     align );      // input,  alignment position

    // set character colors
    MRESULT SetColors(
            MGCOLORPIX  charColor,    // input,  character pixel value
            MGCOLORPIX  backColor );  // input,  background pixel value

    // set justification spacing
    MRESULT SetJustify(
            FIXDOT      charExtra,    // input,  charExtra spacing
            FIXDOT      wordExtra,    // input,  wordExtra spacing
            TSUNITS     units );      // input,  units of measure

    // set the transfer mode rasterOp
    MRESULT SetRasterOp(
            TSRASTEROP  rasterOp );   // input,  bitmap transfer mode

    // set anti-alias edge smoothing
    MRESULT SetSmoothing(
            TSSMOOTH    smoothLevel );// input,  smoothing level

    // set the type size
    MRESULT SetTypeSize(
            FIXDOT      charHeight,   // input,  character height
            TSUNITS     units );      // input,  unit of measure

};  // end class tsCStrike
```

Function Descriptions

The following sections provide detailed descriptions for each individual strike function.

tsStrike_Create() - create a strike instance

C Syntax typeserv.h

```
/* create a font strike instance*/
MRESULT tsStrike_Create(
            TSFONT      font,           /* input,  font instance handle   */
            MGBITMAP    bitmap          /* input,  bitmap instance handle */
            TSSTRIKE   *strike );       /* output, strike instance handle */
```

C++ Syntax typeserv.hpp

```
// constructor
tsCStrike::tsCStrike(
            tsCFont    *font,           // input,  source font
            mgCBitmap  *bitmap );       // input,  bitmap to render to
```

Description

A strike is an instance of a font with a defined set of attributes for point size, device resolution, rotation angle, path angle and other settings needed for rendering to a raster bitmap. The application program may create and open multiple strikes of the same font with different attributes. For example, a 10-point strike and a 14-point strike for a given font can be defined at the same time.

`tsStrike_Create()` initializes and creates a strike with a default set of attributes. Other strike functions can be called to change the attributes as needed. The default attributes initialized by `tsStrike_Create()` are as follows:

Strike Attribute	Function(s)	Default
Character Size	`tsStrike_SetTypeSize()`	10.0 points
Character Alignment	`tsStrike_SetAlign()`	`leftBaseline`
Justify Char-Spacing	`tsStrike_SetJustify()`	0.0
Justify Word-Spacing	`tsStrike_SetJustify()`	0.0
Line Spacing	`tsStrike_SetLineSpacing()`	12.0 points
Background Color	`tsStrike_SetColors()`	White
Character Color	`tsStrike_SetColors()`	Black
Transfer Mode	`tsStrike_SetRasterOp()`	`tsCOPY`
Edge Smoothing	`tsStrike_SetSmoothing()`	`tsSMOOTH16`

C Parameters

`TSFONT    font     (C, input)`

Font-handle of the TrueType font to use in this strike (the font-handle is returned by `tsFont_OpenFile()`or `tsFont_OpenMemory()` when the font is first opened).

`MGBITMAP  bitmap   (C, input)`

Bitmap-handle for the destination bitmap where the rasterized character images are to be rendered (the bitmap-handle is returned by `mgBitmap_Create()` when the bitmap-instance is created).

```
TSSTRIKE  *strike  (C, output)
```

Pointer to a `TSSTRIKE` handle where the new strike-handle is returned. When declaring this variable, your application should initialize it to a value of `NULL` (see **Comments**, below).

C++ Parameters

```
tsCFont   *font    (C++, input)
```

Pointer to the TrueType font-instance for this strike. A C++ font-instance is returned by the `tsCFont()` constructor when a TrueType font is opened.

```
mgCBitmap *bitmap  (C++, input)
```

Pointer to a bitmap-instance for the destination bitmap where the rasterized character images are to be rendered. A C++ bitmap-instance is returned by the `mgCBitmap()` constructor when the bitmap is defined.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

In C, the strike-handle should be initialized to a value of `NULL` before calling `tsStrike_Create()`. Calling `tsStrike_Create()` with a non-`NULL` strike-handle will result in an `MRESULT` warning. This convention is used to insure that a second `tsStrike_Create()` call does not overwrite a previously opened strike-handle before `tsStrike_Destroy()` has been called:

```
    /* declare a strike instance handle and initialize it to NULL */
    TSSTRIKE  myStrike = NULL;
```

All bitmap instances created by `tsStrike_Create()` should be released by your application when no longer needed (and before program termination) by calling `tsStrike_Destroy()`.

See Also

`tsFont_OpenFile(), tsFont_OpenMemory(), tsStrike_Destroy()`

tsStrike_Destroy() - destroy a strike instance

C Syntax

typeserv.h

```
/* release & close a font strike  */
MRESULT tsStrike_Destroy(
        TSSTRIKE  *strike );    /* in/out, handle of strike to close */
```

C++ Syntax

typeserv.hpp

```
// release & close a font strike
tsCStrike::~tsCStrike();     // destructor
```

Description

`tsStrike_Destroy()` closes the strike previously created by `tsStrike_Create()`. This call releases all resources previously created and used by the strike.

Parameters

`TSSTRIKE  *strike  (input/output)`

Pointer to the strike-handle that was initially returned by `tsStrike_Create()`. As part of the close process, the strike-handle is reset to `NULL`.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

See Also

`tsStrike_Create()`

tsStrike_DrawChar() - draw a character

C Syntax

typeserv.h

```
/* draw a ASCII/Unicode (TCHAR) character */
MRESULT tsStrike_DrawChar(
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate   */
  const TCHAR       character );    /* input,  character to draw         */

/* draw an ASCII character (CHAR) */
MRESULT tsStrike_DrawCharA(
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate   */
  const CHAR        character );    /* input,  ASCII character to draw   */

/* draw a Unicode character (WCHAR) */
MRESULT tsStrike_DrawCharW(
        TSSTRIKE    strike,         /* input,  strike-handle             */
        FIXPOINT   *location,       /* in/out, starting X,Y coordinate   */
  const WCHAR       character );    /* input,  Unicode character to draw */
```

C++ Syntax

typeserv.hpp

```
// draw a ASCII/Unicode (TCHAR) character
MRESULT tsCStrike::DrawChar(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const TCHAR       character );  // input,  character to draw
```

```
// draw an ASCII (CHAR) character
MRESULT tsCStrike::DrawCharA(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const CHAR        character );  // input,  ASCII character to draw

// draw a Unicode (WCHAR) character
MRESULT tsCStrike::DrawCharW(
        FIXPOINT   *location,     // in/out, starting X,Y coordinate
  const WCHAR       character );  // input,  Unicode character to draw
```

Description

Using the current attributes of the font strike, `DrawChar()` draws a single character positioned at a specified pixel coordinate `location`. If the character is not defined in the current font, the font's "missing" symbol is drawn.

Parameters

`TSSTRIKE   strike      (C, input)`

Strike-handle of the font strike to use for drawing.

`FIXPOINT  *location    (input)`

Pointer to the starting pixel coordinate where the character is to be drawn. The X,Y coordinates are defined using the `FIXDOT` data type that allows fractional pixel positioning. After the string is drawn, the coordinate location is advanced in the direction of the strike "path angle" (normally to the right) where the start of the next character or string would typically begin.

`TCHAR/CHAR/WCHAR  character  (input)`

Character to be drawn. `TCHAR` type characters are native to the compiled environment and are either ASCII or Unicode dependent if the identifier `_UNICODE` is defined. (If `_UNICODE` is not defined, `TCHAR` is defined as type "`CHAR`" for 8-bit ASCII characters. If `_UNICODE` is defined, `TCHAR` is defined as type "`WCHAR`" for 16-bit Unicode characters.) Type `CHAR` characters are standard 8-bit ASCII characters. Type `WCHAR` characters are 16-bit Unicode characters.

Returns

An `MRESULT` value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

The `location` parameter is also updated when the function returns, and is positioned to the location where the start of the next character or string should begin.

Comments

The `DrawChar()` function is actually a macro that changes depending if the identifier `_UNICODE` is defined or not. If `_UNICODE` is not defined, `DrawChar()` is defined as `DrawCharA()` (ASCII), which is the TypeServer function that draws 8-bit ASCII character strings. If `_UNICODE` is defined, `DrawChar()` is defined as `DrawCharW()` which is the TypeServer function for drawing 16-bit Unicode characters. The Appendix B tutorial, "**Writing Code for ASCII & Unicode Portability**", provides a discussion on how you can write C and C++ programs that will operate compatibly with either ASCII or Unicode environments.

See Also

Strike Attributes, `tsStrike_DrawString()`

Example

```
    TSSTRIKE   myStrike;
    FIXPOINT   pixCoord;

    /* draw the character 'A' at pixel location 80.0,100.5 */
    pixCoord.x = IntToFix(80);
    pixCoord.y = IntToFix(100) + FIXDOT_HALF;
    result = tsStrike_DrawChar( myStrike, &pixCoord, 'A' );
    _ASSERT( SUCCEEDED(result) );
```

tsStrike_DrawString() - draw a character string

C Syntax

typeserv.h

```
/* draw a ASCII/Unicode (TCHAR) string */
MRESULT tsStrike_DrawString(
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate   */
  const TCHAR       *string );      /* input,  string to draw            */

/* draw an ASCII string (CHAR) */
MRESULT tsStrike_DrawStringA(
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate   */
  const CHAR        *string );      /* input,  ASCII string to draw      */

/* draw a Unicode string (WCHAR) */
MRESULT tsStrike_DrawStringW(
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate   */
  const WCHAR       *string );      /* input,  Unicode string to draw    */
```

C++ Syntax

typeserv.hpp

```
    // draw a ASCII/Unicode (TCHAR) string
    MRESULT tsCStrike::DrawString(
            FIXPOINT   *location,     // in/out, starting X,Y coordinate
      const TCHAR      *string );     // input,  string to draw

    // draw an ASCII string (CHAR)
    MRESULT tsCStrike::DrawStringA(
            FIXPOINT   *location,     // in/out, starting X,Y coordinate
      const CHAR       *string );     // input,  ASCII string to draw

    // draw a Unicode string (WCHAR)
    MRESULT tsCStrike::DrawStringA(
            FIXPOINT   *location,     // in/out, starting X,Y coordinate
      const WCHAR      *string );     // input,  Unicode string to draw
```

Description

Using the current attributes of the font strike, `DrawString()` draws a character string positioned at a specified starting pixel coordinate `location`. Characters in the string that are not defined within the font are drawn using the font's "missing" symbol.

Parameters

```
TSSTRIKE   strike      (C, input)
```

Strike-handle of the font strike to use for drawing.

```
FIXPOINT  *location    (input)
```

Pointer to the starting pixel coordinate where the string is to be drawn. The X,Y coordinates are defined using the `FIXDOT` data type that allows fractional pixel positioning. After the string is drawn, the coordinate location is advanced in the direction of the strike "path angle" (normally to the right) where the start of the next character or string would typically begin.

```
TCHAR/CHAR/WCHAR  *string  (input)
```

Pointer to the string of characters to be drawn. `TCHAR` type characters are native to the compiled environment and are either ASCII or Unicode dependent if the identifier `_UNICODE` is defined (if `_UNICODE` is not defined, `TCHAR` is type "`CHAR`" for 8-bit ASCII characters; if `UNICODE` is defined, `TCHAR` is type "`WCHAR`" for 16-bit Unicode characters). Type `CHAR` characters are standard 8-bit ASCII characters. Type `WCHAR` characters are 16-bit Unicode characters.

Returns

An `MRESULT` value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

The `location` parameter is updated when the function returns. This updated to the position where the next character in a continuation of string would begin.

Comments

The `DrawString()` function is actually a macro that changes depending if the identifier `_UNICODE` is defined or not. If `_UNICODE` is not defined, `DrawString()` is defined as `DrawStringA()` (ASCII), which is the TypeServer function that draws 8-bit ASCII character strings. If `_UNICODE` is defined, `DrawString()` is defined as `DrawStringW()` which is the TypeServer function for drawing 16-bit Unicode characters. The Appendix B tutorial, "**Writing Code for ASCII & Unicode Portability**", provides a discussion on how you can write C and C++ programs that will operate compatibly with either ASCII or Unicode environments.

Example

```
    TSSTRIKE   myStrike;
    FIXPOINT   pixCoord;

    /* create TypeServer, define bitmap, open font and create strike... */

    /* draw the string "Hello World" at pixel location (80.0, 100.5)    */
    pixCoord.x = IntToFix(80);
    pixCoord.y = IntToFix(100) + FIXDOT_HALF;
    result = tsStrike_DrawString( myStrike,
                             &pixCoord, _TEXT("Hello World") );
    _ASSERT( SUCCEEDED(result) );
```

See Also

Strike Attributes, `tsStrike_DrawChar()`

tsStrike_GetCharExtent() - get the dimensions of a character

C Syntax

typeserv.h

```
/* get the dimensions of a ASCII/Unicode (TCHAR) character */
MRESULT tsStrike_GetCharExtent(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const TCHAR        character,     /* input,  character                 */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of an ASCII (CHAR) character */
MRESULT tsStrike_GetCharExtentA(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const CHAR         character,     /* input,  ASCII character           */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of a Unicode (WCHAR) character */
MRESULT tsStrike_GetCharExtentW(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const WCHAR        character,     /* input,  Unicode character         */
        FIXSIZE     *extent );      /* output, character width & height  */
```

C++ Syntax

typeserv.hpp

```
// get the dimensions of a ASCII/Unicode (TCHAR) character
MRESULT tsCStrike::GetCharExtent(
  const TCHAR       character,    // input,  character
        FIXSIZE    *extent );     // output, character width & height

// get the dimensions of an ASCII (CHAR) character
MRESULT tsCStrike::GetCharExtentA(
  const CHAR        character,    // input,  ASCII character
        FIXSIZE    *extent );     // output, character width & height
```

```
// get the dimensions of a Unicode (WCHAR) character
MRESULT tsCStrike::GetCharExtentW(
  const WCHAR       character,    // input,  Unicode character
        FIXSIZE    *extent );     // output, character width & height
```

The `GetCharExtent()` function computes the pixel height and width of a specified character. The width and height are computed without regards to any clipping.

Parameters

```
TSSTRIKE  strike     (C, input)
```

Strike-handle.

```
TCHAR/CHAR/WCHAR  character (input)
```

Character to return the dimension of. `TCHAR` type characters are native to the compiled environment and are either ASCII or Unicode dependent if the identifier `_UNICODE` is defined. (If `_UNICODE` is not defined, `TCHAR` is defined as type "`CHAR`" for 8-bit ASCII characters. If `_UNICODE` is defined, `TCHAR` is defined as type "`WCHAR`" for 16-bit Unicode characters.) Type `CHAR` characters are standard 8-bit ASCII characters. Type `WCHAR` characters are 16-bit Unicode characters.

```
FIXSIZE  *extent     (output)
```

Pointer to a `FIXSIZE` structure where the pixel dimensions of the character are to be returned. Height and width dimensions are returned using the `FIXDOT` data type that provides fractional pixel accuracy.

Returns

An `MRESULT` value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

`GetCharExtent()` returns the basic height and width of a character. Because some fonts kern character pairs, the sum of the extents of individual characters may be greater than the extent of the characters when computed together as a string by `GetStringExtent()`. `GetCharExtent()` also does not take into account inter-character and inter-word spacing adjustments set by `SetJustify()`.

See Also

`tsStrike_GetStringExtent(), tsStrike_SetJustify()`

tsStrike_GetStringExtent() - get the dimensions of a string

C Syntax typeserv.h

```
/* get the dimensions of a ASCII/Unicode (TCHAR) string */
MRESULT tsStrike_GetStringExtent(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const TCHAR       *string,        /* input,  character string          */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of an ASCII (CHAR) character */
MRESULT tsStrike_GetStringExtentA(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const CHAR        *string,        /* input,  ASCII string              */
        FIXSIZE     *extent );      /* output, character width & height  */

/* get the dimensions of a Unicode (WCHAR) character */
MRESULT tsStrike_GetStringExtentW(
        TSSTRIKE     strike,        /* input,  strike-handle             */
  const WCHAR       *string,        /* input,  Unicode string            */
        FIXSIZE     *extent );      /* output, character width & height  */
```

C++ Syntax typeserv.hpp

```
// get the dimensions of a ASCII/Unicode (TCHAR) character
MRESULT tsCStrike::GetStringExtent(
  const TCHAR      *string,       // input,  character string
        FIXSIZE    *extent );     // output, character width & height

// get the dimensions of an ASCII (CHAR) character
MRESULT tsCStrike::GetStringExtentA(
  const CHAR       *string,       // input,  ASCII string
        FIXSIZE    *extent );     // output, character width & height

// get the dimensions of a Unicode (WCHAR) character
MRESULT tsCStrike::GetStringExtentW(
  const WCHAR      *string,       // input,  Unicode string
        FIXSIZE    *extent );     // output, character width & height
```

The `GetStringExtent()` function computes the pixel height and width of a specified character string. The width and height are computed without regards to any clipping.

Parameters

`TSSTRIKE  strike     (C, input)`

Strike-handle.

`TCHAR/CHAR/WCHAR  *string  (input)`

Pointer to the string of characters to have the dimensions computed for. `TCHAR` type characters are native to the compiled environment and are either ASCII or Unicode dependent if the identifier `_UNICODE` is defined (if `_UNICODE` is not defined, `TCHAR` is type "`CHAR`" for 8-bit ASCII characters; if `UNICODE` is defined, `TCHAR` is type "`WCHAR`" for 16-bit Unicode characters). Type `CHAR` characters are standard 8-bit ASCII characters. Type `WCHAR` characters are 16-bit Unicode characters.

```
FIXSIZE  *extent   (output)
```

Pointer to a FIXSIZE structure where the pixel dimensions of the character are to be returned. Height and width dimensions are returned using the FIXDOT data type that provides fractional pixel accuracy.

Returns

An MRESULT value is returned indicating success or failure of the function call. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

The GetStringExtent() function is actually a macro that is conditioned on whether the identifier _UNICODE is defined or not. (If _UNICODE is not defined, GetStringExtent() is defined as GetStringExtentA() which is the TypeServer function for computing sizes using 8-bit ASCII character strings; if _UNICODE is defined, GetStringExtent() is defined as GetStringExtentW() which is the TypeServer function for computing sizes using 16-bit Unicode characters.) The Appendix B tutorial, "**Writing Code for ASCII & Unicode Portability**", provides a discussion on how you can write C and C++ programs that will operate compatibly with either ASCII or Unicode environments.

See Also

tsStrike_GetCharExtent(), tsStrike_GetStringExtent(), tsStrike_SetJustify()

tsStrike_GetInfoPtr() - get pointer to strikeInfo

C Syntax

typeserv.h

```
/* get pointer to the strike information structure */
TSSTRIKEINFO* tsStrike_GetInfoPtr(  /* return, pointer to strikeInfo     */
        TSSTRIKE       strike,      /* input,  strike-handle             */
        int            infoSize );  /* input,  TSSTRIKEINFO struct size  */
```

C++ Syntax

typeserv.hpp

```
// get pointer to the strike information structure
TSSTRIKEINFO* tsCStrike::GetInfoPtr(// return, pointer to strikeInfo
        int            infoSize );  // input,  TSSTRIKEINFO struct size
```

The GetInfoPtr() returns a pointer to the strike's TSSTRIKEINFO data structure that contains the current rendering attributes for the strike.

Parameters

```
TSSTRIKE       strike          (C, input)
```

Strike-handle.

```
int            infoSize    (input)
```

Size of the TSSTRIKEINFO structure (see **Comments**, below).

Returns

GetInfoPtr() returns with a pointer to the bitmap's TSFONTINFO information record.

Comments

To provide compatibility with updated TypeServer versions running with DLL's, a structure size parameter is passed to verify and identify the structure version used in the client application. For more information please see the **Metagraphics C/C++ Programming Guidelines** manual discussion on *"Enhancing 'struct' Compatibility".*

Example

```
    TSSTRIKEINFO  *pStrikeInfo;      /* pointer to strikeInfo */

    pStrikeInfo = tsStrike_GetInfoPtr( sizeof(TSSTRIKEINFO) );
    _ASSERT( pStrikeInfo != NULL );  /* debug check */
```

See Also

Strike Attributes, TSSTRIKEINFO

tsStrike_SetAlign() - set text alignment position

C Syntax

typeserv.h

```
/* set the text alignment position */
MRESULT tsStrike_SetAlign(
        TSSTRIKE     strike,        /* input,  strike-handle            */
        TSALIGN      align );       /* input,  alignment position       */
```

C++ Syntax

typeserv.hpp

```
// set the text alignment position
MRESULT tsCStrike::SetAlign(
        TSALIGN      align );       // input,  alignment position
```

Description

SetAlign() defines the alignment position for the starting location when characters and strings are drawn.

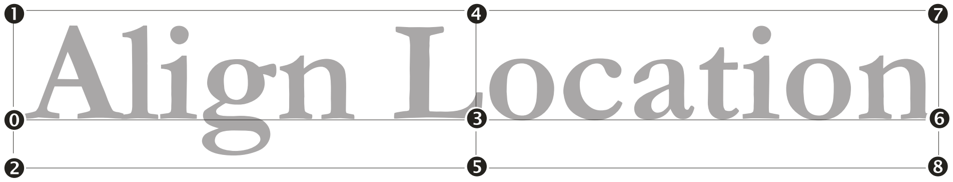
Parameters

TSSTRIKE   strike   (C, input)

Strike-handle to set the alignment for.

TSALIGN   align   (input)

One of nine possible alignment positions:



```
0-tsLEFT_BASELINE     3-tsCENTER_BASELINE     6-tsRIGHT_BASELINE
1-tsLEFT_TOP          4-tsCENTER_TOP          7-tsRIGHT_TOP
2-tsLEFT_BOTTOM       5-tsCENTER_BOTTOM       8-tsRIGHT_BOTTOM
```

Returns

An MRESULT value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

The default character alignment location when a strike is first created is tsLEFT_BASELINE.

See Also

tsStrike_Create()

Example

```
   /* set character align right_top     */
   tsStrike_SetAlign( mystrike, tsRIGHT_TOP );

   /* set character align left_baseline */
   tsStrike_SetAlign( mystrike, tsLEFT_BASELINE );
```

tsStrike_SetColors(), tsStrike_SetColorsRGB - set character colors

C Syntax

typeserv.h

```
/* set character and background colors by pixel value */
MRESULT tsStrike_SetColors(       /* return, completion code(0=success)*/
        TSSTRIKE    strike,       /* input,  strike-handle             */
        MGCOLORPIX  pixChar,      /* input,  character pixel value     */
        MGCOLORPIX  pixBack );    /* input,  background pixel value    */

/* set character and background colors by RGB value */
MRESULT tsStrike_SetColorsRGB(    /* return, completion code(0=success)*/
        TSSTRIKE    strike,       /* input,  strike-handle             */
        MGCOLORRGB  rgbChar,      /* input,  character RGB color       */
        MGCOLORRGB  rgbBack );    /* input,  background RGB color      */
```

C++ Syntax

typeserv.hpp

```
// set character and background colors by pixel value
MRESULT tsCStrike::SetColors(     // return, completion code (0=success)
        MGCOLORPIX  pixChar,      // input,  character pixel value
        MGCOLORPIX  pixBack );    // input,  background pixel value
```

```
// set character and background colors by RGB value
MRESULT tsCStrike::SetColorsRGB(  // return, completion code (0=success)
        MGCOLORRGB  rgbChar,      // input,  character RGB color
        MGCOLORRGB  rgbBack );    // input,  background RGB color
```

Description

`SetColors()` and `SetColorsRGB()` allow you to define the color for characters drawn either in terms of direct pixel values (specific to the attached output bitmap), or in terms of generic RGB colors. These values specific the foreground character color and background color of the characters when they are rendered. When a strike is first created, the background color is set to white and the character color is is set to black.

Parameters

`TSSTRIKE    strike      (C, input)`

Strike-handle to set the foreground character and background colors for.

`MGCOLORPIX  pixChar     (SetColors(), input)`

Formatted pixel value for the character.

`MGCOLORPIX  pixBack     (SetColors(), input)`

Formatted pixel value for the background.

Pixel values are format-specific for the attached output bitmap. If the output bitmap is 256 colors or less, the pixel values are indexes into the bitmap's colorTable. If the bitmap is greater than 256-colors, pixel values contain encoded RGB formatted for the specific bitmap type (for example, a 16 bit-per-pixel bitmap stores RGB values in a 5:6:5 format within a single 2 byte value). The `mgBitmap_RGBToPix()` function can be used to convert generic RGB colors to bitmap pixel values.

`MGCOLORRGB  rgbChar     (SetColorsRGB(), input)`

Generic RGB color for the character image.

`MGCOLORRGB  rgbBack     (SetColorsRGB(), input)`

Generic RGB color for the character background.

The `SetColorsRGB()` function may be used to specify character colors using generic RGB values. The `RGB_Make(r,g,b)` macro provides a convenient method for encoding R,G,B intensity components into an generic RGB value.

When rendering anti-aliased characters, aliased halftone pixels are drawn using percentages between the character color and background (eg. a 50% halftone pixel is mid-point between the character and background colors). When a "transparent" transfer mode is set (`SetRasterOp()`), background color pixels are not drawn, and only the foreground image of the character is transferred.

Returns

An MRESULT value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

See Also

tsStrike_Create(), RGB_Make(), mgBitmap_RGBToPix(), mgBitmap_PixToRGB()
tsStrike_SetRasterOp()

tsStrike_SetJustify() - set extra character and/or word spacing

C Syntax **typeserv.h**

```
/* set extra character and/or word spacing */
MRESULT tsStrike_SetJustify(
        TSSTRIKE  strike,        /* input,  strike-handle                 */
        FIXDOT    charExtra,     /* input,  extra inter-character spacing */
        FIXDOT    wordExtra,     /* input,  extra inter-word spacing      */
        TSUNITS   units );       /* input,  units (pixels or points)      */
```

C++ Syntax **typeserv.hpp**

```
// set extra character and/or word spacing
MRESULT tsCStrike::SetJustify(
        FIXDOT    charExtra,     // input,  extra inter-character spacing
        FIXDOT    wordExtra,     // input,  extra inter-word spacing
        TSUNITS   units );       // input,  units (pixels or points)
```

Description

SetJustify() allows you to widen or tighten the spacing between characters or words. charExtra and wordExtra are typically defined when creating lines of text with justified left and right margins.

charExtra defines an amount of additional spacing to widen or tighten the distance between each character. The charExtra value is added to the normal character width, with positive values expanding the character spacing and negative values tightening the character spacing. When a strike is first created the default charExtra setting is 0.0.

wordExtra is added *only* to the widths of blank "space" characters (which is different from the charExtra attribute that is added to the widths of *all* characters, including spaces). The wordExtra value is added to the normal "space" character width, with positive values expanding the word spacing and negative values tightening the word spacing. When a strike is first created the default wordExtra setting is 0.0.

charExtra=0.0 pt; wordExtra=0.0 pt

charExtra=+1.0 pt; wordExtra=0.0 pt

charExtra= -1.0 pt; wordExtra= 0.0 pt

charExtra=0.0 pt; wordExtra=+2.0 pt

charExtra=0.0 pt; wordExtra=-1.0 pt

charExtra=+1.0 pt; wordExtra=+1.0 pt

charExtra=-1.0 pt; wordExtra=-1.0 pt

Parameters

```
TSSTRIKE  strike     (C, input)
```

Strike-handle to set the `charExtra` and `wordExtra` attributes for.

```
FIXDOT    charExtra  (input)
```

Amount of spacing by which to widen or tighten the distance between characters when drawing a string of text. `charExtra` is added to the width of *all* characters (including "space" characters). Positive values increase the spacing between characters, while negative values condense the character spacing. `charExtra` is specified using the `FIXDOT` data type that provides fractional accuracy. The units of measure may be either in absolute pixels or device-independent typographic-points ($1/72^{nds}$ of an inch), as specified by the `units` parameter.

```
FIXDOT    wordExtra  (input)
```

Amount of spacing by which to widen or tighten the distance between words when drawing a string of text. `wordExtra` is added to the width of blank "space" characters only. Positive values increase the spacing between words, while negative values condense the word spacing. `wordExtra` is specified using the `FIXDOT` data type to provide fractional accuracy. The units of measure may be either in absolute pixels or device-independent typographic-points ($1/72^{nds}$ of an inch), as specified by the `units` parameter.

```
TSUNITS   units      (input)
```

Defined from the `TSUNITS` enumeration, `units` may be set to either `tsPIXELS` or `tsPOINTS`. `units` defines the unit of measure for the `charExtra` and `wordExtra` parameters.

Returns

An `MRESULT` value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

Note again that `wordExtra` expands or compresses the spacing only for blank-space characters separating words. The `charExtra` expands or condenses the spacing between *all* characters (including blank-space characters).

Example

```
/* expand the character spacing by 0.21 pixels */
tsStrike_SetJustify( myStrike, FloatToFix(0.21), 0, tsPIXELS );
```

```
/* condense the character spacing by .25 pixels */
tsStrike_SetJustify( myStrike, -FIXDOT_FOURTH, 0, tsPIXELS );

/* expand the character spacing by 1 point */
tsStrike_SetJustify( myStrike, FIXDOT_ONE, 0, tsPIXELS );

/* expand the word spacing by 0.21 pixels */
tsStrike_SetJustify( myStrike, 0, FloatToFix(0.21), tsPIXELS );

/* condense the word spacing by 2 pixels */
tsStrike_SetJustify( myStrike, 0, -IntToFix(2), tsPIXELS );

/* expand the both character and word spacing by 1 point */
tsStrike_SetJustify( myStrike, FIXDOT_ONE, FIXDOT_ONE, tsPOINTS );
```

See Also

`TSUNITS, tsStrike_SetLineSpacing()`

tsStrike_SetLineSpacing() - set line spacing

C Syntax

typeserv.h

```
/* set line spacing */
MRESULT tsStrike_SetLineSpacing(
        TSSTRIKE   strike,         /* input,  strike-handle            */
        FIXDOT     lineSpacing,    /* input,  line spacing             */
        TSUNITS    units );        /* input,  units (pixels or points) */
```

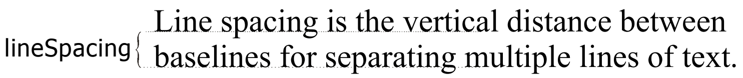
C++ Syntax

typeserv.hpp

```
// set line spacing
MRESULT tsCStrike::SetLineSpacing(
        FIXDOT     lineSpacing,    // input,  line spacing
        TSUNITS    units );        // input,  units (pixels or points)
```

Description

`SetLineSpacing()` sets the `TSSTRIKEINFO.leading` attribute that defines the distance for positioning consecutive lines of text vertically. The `TSSTRIKEINFO.leading` value defines the distance between consecutive baselines of text. The line spacing can be specified either in terms of typographic-points or in terms of pixels, as selected by the `units` parameter. When a strike is first created line leading is initialized to a default value of twelve points (12.0 pt).



Line spacing is the vertical distance between
baselines for separating multiple lines of text.

Parameters

`TSSTRIKE  strike       (C, input)`

Strike-handle to set the line spacing for.

`FIXDOT    lineSpacing  (input)`

Distance for spacing successive lines of text. `lineSpacing` is defined using the `FIXDOT` data type that provides fractional accuracy, and may be supplied either in terms of typographic-points or pixels as defined by the `units` parameter.

```
TSUNITS   units        (input)
```

Defined using the `TSUNITS` enumeration, `units` may be set to either `tsPIXELS` or `tsPOINTS`. `units` selects the unit of measure for the `lineSpacing` parameter.

Returns

An `MRESULT` value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

The default leading distance when a strike is first created is twelve points (12.0 pts).

Example

```
/* set the line spacing to 16.25 pixels */
tsStrike_SetLineSpacing( myStrike, IntToFix(16) + FIXDOT_FOURTH, tsPIXELS );

/* set the line spacing to 14.0 pixels */
tsStrike_SetLineSpacing( myStrike, IntToFix(14), tsPIXELS );

/* set the line spacing to 10.21 points */
tsStrike_SetLineSpacing( myStrike, FloatToFix(10.21), tsPOINTS );
```

See Also

`tsStrike_SetTypeSize()`

tsStrike_SetRasterOp() - set the rasterOp transfer mode

C Syntax **typeserv.h**

```
/* set the transfer mode rasterOp */
MRESULT tsStrike_SetRasterOp(
        TSSTRIKE     strike,        /* input,  strike-handle           */
        TSRASTEROP   rasterOp );    /* input,  bitmap transfer mode      */
```

C++ Syntax **typeserv.hpp**

```
// set the transfer mode rasterOp
MRESULT tsCStrike::SetRasterOp(
        TSRASTEROP   rasterOp );    // input,  bitmap transfer mode
```

Description

The strike rasterOp defines how rendered characters are combined into the destination bitmap. TypeServer uses the `TSRASTEROP` enumeration to define four "standard" rasterOps, and four "transparent" rasterOps (see below). For "transparent" rasterOps, only the foreground color pixels of the character affects the target bitmap (background-color pixels are not transferred).

Parameters

```
TSSTRIKE   strike     (C, input)
```

Strike-handle to set the rasterOp for.

```
TSRASTEROP rasterOp   (input)
```

One of eight possible rasterOp transfer modes:

```
          tsCOPY,               /* replace (default)         */
          tsMERGE,              /* OR character into bitmap  */
          tsERASE,              /* AND character into bitmap */
          tsINVERT,             /* XOR character into bitmap */
          tsTRANSPARENT_COPY,   /* transparent replace       */
          tsTRANSPARENT_MERGE,  /* transparent OR            */
          tsTRANSPARENT_ERASE,  /* transparent AND           */
          tsTRANSPARENT_INVERT  /* transparent XOR           */
```

Returns

An MRESULT value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

The default rasterOp transfer mode when a strike is first created is tsCOPY.

Example

```
   /* set the strike rasterOp for transparent-copy */
   tsStrike_SetRasterOp( myStrike, tsTRANSPARENT_COPY );
```

See Also

TSRASTEROP, tsStrike_SetColors()

tsStrike_SetSmoothing() - set edge smoothing

C Syntax

typeserv.h

```
/* set anti-alias edge smoothing */
MRESULT tsStrike_SetSmoothing(
        TSSTRIKE    strike,         /* input,  strike-handle            */
        TSSMOOTH    smoothLevel );  /* input,  smoothing level          */
```

C++ Syntax

typeserv.hpp

```
// set anti-alias edge smoothing
MRESULT tsCStrike::SetSmoothing(
        TSSMOOTH    smoothLevel );  // input,  smoothing level
```

Description

The strike's smooth attribute defines the number of incremental colors TypeServer uses when rendering characters. When rendering text in black and white, for example, this is the number of intermediate grayscale levels that is used for anti-aliasing the character edges. In the general case, the incremental smoothing colors will range evenly between the strike's current `backColor` and `charColor` settings as defined by `tsStrike_SetColors()`.

Parameters

```
TSSTRIKE   strike       (C, input)
```

Strike instance handle to set the smoothing level for.

```
TSSMOOTH   smoothLevel  (input)
```

`TTSMOOTH` enumeration specifying one of four edge smoothing options:

```
          tsSMOOTH0        /* no edge smoothing                */
          tsSMOOTH4        /* 4-level anti-aliasing            */
          tsSMOOTH16       /* 16-level anti-aliasing (default) */
          tsSMOOTH256      /* 256-level anti-aliasing          */
```

Returns

An `MRESULT` value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see `MRESULT`, Appendix A).

Comments

TypeServer will automatically reduce the smoothing level if it exceeds the number of colors in the destination bitmap. When rendering to a monochrome bitmap, for example, no anti-aliasing can be performed and rendering will automatically be processed as `tsSMOOTH0`. For 16- and 256-color modes that require color tables, TypeServer may also reduce the smoothing if the color table for the destination bitmap does not contain colors that are close enough to the aliasing colors needed. Note that edge smoothing isn't free – the higher the smoothing level, the higher the processing overhead.

Example

```
    /* set for 4-level anti-alias edge smoothing */
    tsStrike_SetSmoothing( myStrike, tsSMOOTH4 );
```

See Also

`TSSMOOTH`

tsStrike_SetTypeSize() - set the type size

C Syntax

typeserv.h

```
/* set the type size */
MRESULT tsStrike_SetTypeSize(
        TSSTRIKE    strike,       /* input,  strike-handle            */
        FIXDOT      typeSize,     /* input,  character height         */
        TSUNITS     units );      /* input,  units of measure         */
```

C++ Syntax

typeserv.hpp

```
// set the type size
MRESULT tsCStrike::SetTypeSize(
        FIXDOT      typeSize,     // input,  character height
        TSUNITS     units );      // input,  unit of measure
```

Description

SetTypeSize() specifies the size characters to render. The character size is expressed in terms of fixed-point FIXDOT data type that allows fractional sizes. The size specifications are based on a theoretic "EM-square" character, which roughly encloses the capital letter "M". The type size can be specified either in terms of typographic-points (1/72^{nds} of an inch) or in terms of pixels, as selected by the units parameter. The default size when a strike is first created is 10.0 points.

Parameters

TSSTRIKE   strike   (C, input)

Strike instance handle to set the character size for.

FIXDOT   typeSize   (input)

Character type size. The type size is specified using the FIXDOT data type that provides fractional accuracy. The type size can be specified either in terms of typographic-points (1/72^{nds} of an inch) or in terms of a specific pixel size. The units parameter (see below) selects whether the type size is specified either in terms of points or pixels.

TSUNITS   units   (input)

Defined using the TSUNITS enumeration, units may be set to either tsPIXELS or tsPOINTS. units selects the unit of measure for the typeSize parameter.

Returns

An MRESULT value is returned indicating success, failure, or warning information. A zero or positive value indicates success, while a negative value indicates failure. Non-zero values contain additional detail information about special result conditions (see MRESULT, Appendix A).

Comments

When changing type sizes, you should also update the strike line-spacing attribute that controls the distance between baselines when drawing multiple lines of text. Normal line spacing is usually 120% of the type size. After setting the type size, use tsStrike_SetLineSpacing() to set the line-spacing value as needed.

Example

```
/* set the type size to 12.0 pixels */
tsStrike_SetTypeSize( myStrike, IntToFix(12), tsPIXELS );

/* set the type size to 10.5 pixels */
tsStrike_SetTypeSize( myStrike, IntToFix(10)+FIXDOT_HALF, tsPIXELS );

/* set the type size to 8.3 pixels  */
tsStrike_SetTypeSize( myStrike, FloatToFix(8.3), tsPIXELS );
```

```
   /* set the type size to 12.0 points */
   tsStrike_SetPointSize( myStrike, IntToFix(12), tsPOINTS );

   /* set the type size to 10.5 points */
   tsStrike_SetTypeSize( myStrike, IntToFix(10)+FIXDOT_HALF, tsPOINTS );

   /* set the type size to 8.3 points  */
   tsStrike_SetPointSize( myStrike, FloatToFix(8.3), tsPOINTS );
```

See Also

`tsStrike_CreateStrike(), tsStrike_SetLineSpacing()`

Appendix A - TypeServer MRESULT Return Codes

MRESULT Coding

Most TypeServer functions provide back an “MRESULT” completion code when a function returns. The MRESULT return code is either 16- or 32-bits, dependent on the native integer size for the compiler and target platform. MRESULT is conditionally defined as either MRESULT16 or MRESULT32 depending on the integer size:

```
    #define  MRESULT16  signed short
    #define  MRESULT32  signed long

    #if    sizeof(int) == 2
    #define  MRESULT  MRESULT16
    #elif  sizeof(int) >= 4
    #define  MRESULT  MRESULT32
    #else
    #error "Unsupported 'int' size!"
    #endif
```

MRESULT Coding

mgerror.h / tserror.h

Non-zero MRESULT32 return values return an error code, internal tag location, facility or library, and function ID:

```
     3322 2222 2222 1111 1111 11
Bit: 1098 7654 3210 9876 5432 1098 7654 3210  MRESULT32 signed integer
     s--- ---- ---- ---- ---- ---- ---- ----  success(0) or fail(1)
     -i-- ---- ---- ---- ---- ---- ---- ----  informational(0) or warning(1)
     --ee eeee eeee ---- ---- ---- ---- ----  error code (0-1023)
     ---- ---- ---- tttt ---- ---- ---- ----  tag location (0-15)
     ---- ---- ---- ---- gggg gg-- ---- ----  group* (0-63)
     ---- ---- ---- ---- ---- --ff ffff ffff  function (0-1023)
```

* The following groups are currently defined:

0 = Application
1 = Metagraphics MetaWINDOW library
2 = Metagraphics Media!Lab library
3 = Metagraphics Media!Key library
4 = Metagraphics TypeServer library

Non-zero `MRESULT16` return values return an error code and internal tag location:

```
     1111 11
Bit: 5432 1098 7654 3210  MRESULT16 signed integer
     s--- ---- ---- ----  success(0) or fail(1)
     -i-- ---- ---- ----  informational(0) or warning(1)
     --ee eeee eeee ----  error code (0-1023)
     ---- ---- ---- tttt  tag location (0-15)
```

Return Codes

Follows is a list of the standard completion and error codes returned by TypeServer functions (for additional details please see `mgerror.h` and `tserror.h`):

```
     1111 11
Bit: 5432 1098 7654 3210  MRESULT
     s--- ---- ---- ----  success(0) or fail(1)
     --ee eeee eeee ----  error code (0-1023)

/* success, normal return */
#define M_OK                 ((MRESULT)0x0000)

/* undefined error        */
#define MERR_FAILED          ((MRESULT)0x8010)

/* version expiration (contact Metagraphics for new update) */
#define MERR_EXPIRED         ((MRESULT)0x8020)

/* error, bad pointer     */
#define MERR_BADPOINTER      ((MRESULT)0x8030)

/* error, bad handle      */
#define MERR_BADHANDLE       ((MRESULT)0x8040)

/* error, bad structure size */
#define MERR_BADSTRUCTSIZE   ((MRESULT)0x8050)

/* error, bad argument    */
#define MERR_INVALIDARG      ((MRESULT)0x8060)

/* error, bad structure variable  */
#define MERR_BADSTRUCTVAR    ((MRESULT)0x8070)

/* error, redundant call  */
#define MERR_REDUNDANTCALL   ((MRESULT)0x8080)

/* error, out of memory   */
#define MERR_OUTOFMEMORY     ((MRESULT)0x80E0)

/* error, memory leak     */
#define MERR_MEMORYLEAK      ((MRESULT)0x80F0)

/* error, not implemented */
#define MERR_NOTIMPLEMENTED  ((MRESULT)0x8100)
```

```
/* error, file I/O error  */
#define MERR_FILEIOERROR     ((MRESULT)0x8110)

/* error, end of file     */
#define MERR_ENDOFFILE       ((MRESULT)0x8120)

/* error, bad data        */
#define MERR_BADDATA         ((MRESULT)0x8130)

/* error, runaway loop    */
#define MERR_RUNAWAYLOOP     ((MRESULT)0x8140)

/* error, CreateDIBSection*/
#define MERR_CREATEDIB       ((MRESULT)0x8810)
```

Appendix B - Writing Code for ASCII & Unicode Portability

Writing for ASCII & Unicode Portability

As the need to broaden applications onto new platforms and into new markets expands, designing code for language portability becomes a growing importance. Just as familiarity in using `int`, `short` and `long` integer types is important for designing platform portable code, familiarity in handling different character types is important for designing language portable code.

While many compiler and operating systems today remain 8-bit ASCII orientated, a growing number of platforms are now also using 16-bit Unicode as a language standard. To be platform independent an application needs to be capable of running in either an ASCII or Unicode based environment. In addition, there will be cases when working on an ASCII platform where you may need to handle Unicode-specific text, and vice-versa on a Unicode-based platform where you may need to handle ASCII-specific text. The desired goal is to maintain a single source code base that is portable to any platform, and that supports both ASCII and Unicode needs.

Similar to size-specific `INT16(short)`, `INT32(long)` and generic `INT(int)` types for integer uses, the basis for language portability for text starts with the definition of three basic character types: size-specific `CHAR` (8-bit), `WCHAR` ("wide" char, 16-bit), and generic `TCHAR` (conditional 8- or 16-bit).

Data Type	Win32 Type	Description
CHAR	`signed char`	8-bit signed integer, and/or ASCII character
WCHAR	`unsigned short`	16-bit Unicode character
⇔**TCHAR**	`CHAR or WCHAR`	8- or 16-bit character, depending if "`_UNICODE`" is defined

⇔ Indicates variable-size platform dependent conditional data type.

CHAR, WCHAR and TCHAR types

`CHAR` is the 8-bit ASCII-specific character type, and `WCHAR` ("wide char") is a 16-bit Unicode-specific character type. `TCHAR` (generic "text char") is a platform dependent character type that is conditionally equal to either `CHAR` on ASCII platforms, and equal to `WCHAR` on Unicode platforms. The defined identifier "`_UNICODE`" is used to identify if the native environment is a Unicode based platform. If

_UNICODE is undefined, then TCHAR is defined equated to ASCII CHAR; if _UNICODE is defined, TCHAR is equated to Unicode WCHAR.

```
#define  CHAR   signed char
#define  WCHAR  unsigned short

#ifndef _UNICODE
#define  TCHAR  CHAR    /* platform is ASCII   */
#else /*ifdef _UNICODE*/
#define  TCHAR  WCHAR   /* platform is Unicode */
#endif /*_UNICODE*/
```

Literal Characters

The standard C/C++ single-quote (') method for specifying a single literal character works for all three character data types:

```
CHAR   charASCII   = 'A';   /* this is an 8-bit ASCII character       */
WCHAR  charUnicode = 'B';   /* this is a 16-bit Unicode character     */
TCHAR  charSystem  = 'C';   /* ASCII or Unicode depending on platform */
```

The variable charUnicode will be a 16-bit value 0x0041, which is the Unicode representation for the letter B. (Keep in mind that Intel processors store multibyte values with the least significant bytes first, so the bytes are actually stored in memory in the sequence 0x42, 0x00 - remember this when examining a hex dump of Unicode text in memory.)

Literal Strings

Literal ASCII CHAR Strings

Using the standard C/C++ double-quote (") method for specifying literal strings works for ASCII only, but will not work for Unicode character strings.

```
CHAR  strASCII[] = "this is an ASCII string of 8-bit characters";
```

Literal Unicode WCHAR Strings

The ANSI C extension for defining literal Unicode strings is to precede the first double-quote with the capital letter L (as in "Long"). The L preceding the first double-quote is required, and there can not be any spaces between the L and the first double-quote. The L tells the compiler that you want the string to be stored as 16-bit WCHAR characters.

```
WCHAR strUnicode[] = L"this is a Unicode string of 16-bit characters";
```

Literal System TCHAR Strings

For the conditional TCHAR character type, we need a method to conditionally define strings either as an 8-bit ASCII CHAR string, or as a 16-bit Unicode WCHAR string. A method to handle this is to define a special TEXT() macro that performs this function.

```
#ifndef  _UNICODE
#define  __T(s)  s       /* platform is ASCII   */
#else  /* ifdef _UNICODE */
#define  __T(s)  L##s    /* platform is Unicode */
#endif /*_UNICODE*/

#define  TEXT(s)  __T(s)
```

L##s is a somewhat obscure C syntax, but it's an ANSI C specification that uses the ## "token paste" operator to have the C preprocessor concatenate the letter L with the token quoted string s. With the above #define TEXT() macro we can now specify TCHAR strings that are conditionally either ASCII or Unicode based on the target platform:

```
TCHAR strSystem[] = TEXT("ASCII or Unicode string depending on platform");
```

ANSI C ASCII/Unicode Library Functions

In addition to the basic character types and literal specifications, we also need support for common library string manipulation functions. The latest ANSI C STRING.H header file fortunately includes library definitions supporting functions for both 8-bit ASCII and 16-bit Unicode. Similar to the strlen() function that returns the number of characters in an ASCII string, ANSI C now also provides a wcslen() function that returns the number of characters in a Unicode string (very important – the ASCII strlen() function will not return the proper length of a Unicode string!). There is a similar matching Unicode WCHAR function for most of the standard ASCII CHAR string functions. For use with the platform-dependent TCHAR type, a third set of conditional functions are also defined. The following table summarizes the data types and function names associated with each of the CHAR, WCHAR and TCHAR character types.

	ASCII	Unicode	Generic
character size	8-bit	16-bit	8- or 16-bit
type	CHAR	WCHAR	TCHAR
literal character	'.'	'.'	'.'
literal string	"..."	L"..."	TEXT("...")
get character string length	strlen()	wcslen()	tcslen()
find character in string	strchr()	wcschr()	tcschr()
find character, ignore case	strichr()	wcsichr()	tcsichr()
reverse-find character	strrchr()	wcsrchr()	tcsrchr()
reverse-find char, ignore case	strrichr()	wcsrichr()	tcsrichr()

find substring	`strstr()`	`wcsstr()`	`tcsstr()`
find substring, ignore case	`stristr()`	`wcsistr()`	`tcsistr()`
copy string	`strcpy()`	`wcscpy()`	`tcscpy()`
copy string, w/max	`strncpy()`	`wcsncpy()`	`tcsncpy()`
concatenate string	`strcat()`	`wcscat()`	`tcscat()`
concatenate string (w/max)	`strncat()`	`wcsncat()`	`tcsncat()`
compare string	`strcmp()`	`wcscmp()`	`tcscmp()`
compare string, max	`strncmp()`	`wcsncmp()`	`tcsncmp()`
compare string, ignore case	`stricmp()`	`wcsicmp()`	`tcsicmp()`
compare string, nocase, max	`strnicmp()`	`wcsnicmp()`	`tcsnicmp()`
get non-matching char index	`strspn()`	`wcsspn()`	`tcsspn()`
get matching char index	`strcspn()`	`wcscspn()`	`tcscspn()`
find next token	`strtok()`	`wcstok()`	`tcstok()`
locate matching character	`strpbrk()`	`wcspbrk()`	`tcspbrk()`
format data to stdout	`printf()`	`wprintf()`	`tprintf()`
format data to file	`fprintf()`	`fwprintf()`	`ftprintf()`
format data to string	`sprintf()`	`swprintf()`	`stprintf()`
format data to string, w/max	`snprintf()`	`snwprintf()`	`sntprintf()`
format arglist to stdout	`vprintf()`	`vwprintf()`	`vtprintf()`
format arglist to file	`vfprintf()`	`vfwprintf()`	`vftprintf()`
format arglist to string	`vsprintf()`	`vswprintf()`	`vstprintf()`
format args to string, w/max	`vsnprintf()`	`vsnwprintf()`	`vsntprintf()`
open file	`fopen()`	`wfopen()`	`tfopen()`

The `CHAR`, `WCHAR` and `TCHAR` types, character and string literal specifiers, and the string library functions outlined above provide a cohesive and portable method for handling characters and text. These tools facilitate writing single-source applications that can be targeted for new platforms and languages.

TypeServer ASCII/Unicode Functions

Metagraphics TypeServer provides both type-specific and generic functions for `CHAR`, `WCHAR` and `TCHAR` data types. (TypeServer C function names are prefixed with "`tsStrike_`"; C++ method names are unique within the `tsCStrike` class.)

	ASCII	Unicode	Generic
type	`CHAR`	`WCHAR`	`TCHAR`
get char dimensions	`GetCharExtentA()`	`GetCharExtentW()`	`GetCharExtentT()`
get string dimensions	`GetStringExtentA()`	`GetStringExtentW()`	`GetStringExtentT()`
draw character	`DrawCharA()`	`DrawCharW()`	`DrawCharT()`
draw string	`DrawStringA()`	`DrawStringW()`	`DrawStringT()`

MetaWINDOW ASCII/Unicode Functions

Metagraphics MetaWINDOW provides both type-specific and generic functions for `CHAR`, `WCHAR` and `TCHAR` data types. (For 16-bit `WCHAR` and `TCHAR` Unicode types, the associated `...W` and `...T` functions automatically perform Unicode to glyph position translation.)

	ASCII	Unicode	Generic
type	`CHAR`	`WCHAR`	`TCHAR`
get character width	`CharWidth()`	`CharWidthW()`	`CharWidthT()`
get string width	`StringWidth()`	`StringWidthW()`	`StringWidthT()`
draw character	`DrawChar()`	`DrawCharW()`	`DrawCharT()`
draw string	`DrawString()`	`DrawStringW()`	`DrawStringT()`

Support for earlier glyph-position strings is also provided:

	ASCII (8-bit)	Glyph (16-bit)	Generic (8/16)
type	`CHAR`	`USHORT`	`TCHAR`
get character width	`CharWidth()`	`CharWidth16()`	
get string width	`StringWidth()`	`StringWidth16()`	
draw character	`DrawChar()`	`DrawChar16()`	
draw string	`DrawString()`	`DrawString16()`	

Appendix C - typeserv.h Include File

The typeserv.h Include File

Note - The `typeserv.h` header file included with your software may include updates added after this manual was printed. Please see the `\typeserv\include\typeserv.h` file provided with your TypeServer software for authoritative reference.

```
/**************************************************************************
 *  typeserv.h  -  Metagraphics TypeServer(tm) Header File                 *
 *                                                                        *
 *  Copyright (c) Metagraphics - All Rights Reserved                      *
 *                                                                        *
 *  The source code contained herein includes proprietary information of  *
 *  Metagraphics. Use of this source code is strictly limited under the   *
 *  terms of the Metagraphics Software License Agreement.  This source    *
 *  code may not be reproduced, copied or distributed, in whole or in     *
 *  part, without the prior written consent of Metagraphics.              *
 *     http://www.metagraphics.com  -  email: support@metagraphics.com    *
 *   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  *
 *                                                                        *
 *  DESCRIPTION:                                                          *
 *  This header file provides application definitions for constants,      *
 *  variables, data structures and function prototypes for Metagraphics   *
 *  TypeServer(tm).                                                       *
 *                                                                        *
 *  typeserv.h is the only TypeServer header file that should be included *
 *  by a client application.  (Other "ts*.h" headers are internal header  *
 *  files that should not be included by application programs.)           *
 *                                                                        *
 *  See Also:    Metagraphics TypeServer Programming Reference Manual     *
 *               Metagraphics TypeServer Programming On-Line Help         *
 *               Metagraphics TypeServer Source Code Reference Manual     *
 *               Metagraphics C/C++ Programming Guidelines Manual         *
 *                                                                        *
 **************************************************************************
 */

#ifndef  TYPESERV_H
/*=====================================================*/
#define  TYPESERV_H    /* (don't include twice) */
```

```
#ifdef   __VISUALC__
#pragma once           /* (for MS Visual C, parse only once) */
#endif /*__VISUALC__*/

/* includes for other needed TypeServer header files */
#include "metagraphics.h"   /* Metagraphics data types and defines      */
#include "tserror.h"        /* TypeServer function ID's and error codes */

#ifdef   __cplusplus
extern "C" {            /* if C++, disable name managling - - - - - - - - */
#endif /*__cplusplus*/

/* ------------------- TypeServer Enumerated Types -------------------- */

/* text alignment attributes */
typedef enum TSALIGN_
{
    tsLEFT_BASELINE   =0x00,            /* default */
    tsLEFT_BOTTOM     =0x01,
    tsLEFT_TOP        =0x02,
    tsCENTER_BASELINE =0x10,
    tsCENTER_BOTTOM   =0x11,
    tsCENTER_TOP      =0x12,
    tsRIGHT_BASELINE  =0x20,
    tsRIGHT_BOTTOM    =0x21,
    tsRIGHT_TOP       =0x22
} TSALIGN;

/* RasterOp transfer attributes */
typedef enum TSRASTEROP_
{
    tsCOPY,                     /* replace, default         */
    tsMERGE,                    /* OR                       */
    tsERASE,                    /* AND                      */
    tsINVERT,                   /* XOR                      */
    tsTRANSPARENT_COPY,         /* transparent replace      */
    tsTRANSPARENT_MERGE,        /* transparent OR           */
    tsTRANSPARENT_ERASE,        /* transparent AND          */
    tsTRANSPARENT_INVERT        /* transparent XOR          */
} TSRASTEROP;

/* edge smoothing attributes */
typedef enum TSSMOOTH_
{
    tsSMOOTH0,                  /* no smoothing                     */
    tsSMOOTH4,                  /* 4-level anti-aliasing            */
    tsSMOOTH16,                 /* 16-level anti-aliasing (default) */
    tsSMOOTH256                 /* 256-level anti-aliasing          */
} TSSMOOTH;
```

```
/* units of measure selection*/
typedef enum TSUNITS_
{
    tsPIXELS,                   /* units are pixels             */
    tsPOINTS                    /* units are typographic-points */
} TSUNITS;


/* ------------------- TypeServer Info Structures -------------------- */

/* ServerInfo structure */
typedef struct  TSSERVERINFO_
{
    INT32          structSize;  /* size of this structure (bytes)        */
    UINT32         objectType;  /* _FourCC object name (='TSVI')         */
} TSSERVERINFO;

/* FontInfo structure */
typedef struct  TSFONTINFO_
{
    INT32         structSize;   /* size of this structure (bytes)        */
    UINT32        objectType;   /* _FourCC object name (='TSFI')         */
    UINT          numFonts;     /* number of fonts within this file      */
    LONG          pathNameBytes;/* number of bytes in filename string    */
    LONG          pathNameChars;/* number of chars in filename string    */
    TCHAR        *filePathName; /* pointer to path & filename string     */
    LONG          fileOffset;   /* starting file offset                  */
    LONG          fontSize;     /* # of bytes in the file memory buffer  */
    void         *fontBuffer;   /* pointer to font file memory buffer    */
} TSFONTINFO;
```

```
/* StrikeInfo structure */
typedef struct  TSSTRIKEINFO_
{
    INT32         structSize;   /* size of this structure (in bytes)     */
    UINT32        objectType;   /* _FourCC object name (='TSSI')         */
    UINT16        iFont;        /* font number of this font              */
    UINT16        numChars;     /* number of characters in this font     */
    UINT16        minChar;      /* minimum character code in this font   */
    UINT16        maxChar;      /* maximum character code in this font   */
    FIXDOT        pointSize;    /* char size in points (1pt = 1/72 inch) */
    FIXDOT        emHeight;     /* em-square character height   (pixels) */
    FIXDOT        emWidth;      /* em-square character width    (pixels) */
    FIXDOT        ascender;     /* character ascent             (pixels) */
    FIXDOT        descender;    /* character descent            (pixels) */
    FIXDOT        height;       /* character height             (pixels) */
    FIXDOT        maxWidth;     /* maximum character width      (pixels) */
    FIXDOT        maxHeight;    /* maximum character height     (pixels) */
    FIXDOT        leading;      /* baseline spacing             (pixels) */
    FIXDOT        charExtra;    /* extra inter-character spacing(pixels) */
    FIXDOT        wordExtra;    /* extra inter-word spacing     (pixels) */
    FIXDOT        path;         /* character path angle        (degrees) */
    FIXDOT        orientation;  /* character orientation angle (degrees) */
    FIXDOT        slant;        /* character slant angle       (degrees) */
    FIXPOINT      location;     /* current x,y drawing location (pixels) */
    MGCOLORRGB    backColor;    /* background color pixel value          */
    MGCOLORRGB    charColor;    /* character color pixel value           */
    TSALIGN       align;        /* character alignment position          */
    TSRASTEROP    rasterOp;     /* blit rasterOp transfer mode           */
    TSSMOOTH      smoothLevel;  /* anti-alias smoothing level            */
} TSSTRIKEINFO;


/* ================== Global Utility Macros ============= */

/* clear a block of memory to zero  */
#ifdef    __cplusplus
inline  void tsZeroMemory( void *buf, size_t byteCount )
   { memset(buf, 0, byteCount); };
#else /* not __cplusplus */
#define tsZeroMemory(buf, byteCount)  memset(buf, 0, byteCount)
#endif /* __cplusplus */

/* clear a structure to zero        */
#define tsZeroStruct(structure)  tsZeroMemory(structure, \
                                    sizeof(*(structure)))

/* zero a struct & set the first member to the size of the struct itself */
#define tsInitStruct(structure) {                                   \
        tsZeroMemory(&(structure), sizeof(structure));              \
        *(INT32*) &(structure) = sizeof(structure);                 \
        }
```

```
/* _DECLARE_HANDLE macro with strict type checking        */
/*    _DECLARE_HANDLE( name );                            */
/* equivalence:                                           */
/*    typedef const struct name__ { int unused; } *name;  */
#ifndef    _DECLARE_HANDLE
#define    _DECLARE_HANDLE(name)     \
    struct name##__ { int unused; }; \
    typedef const struct name##__ *name
#endif /* !_DECLARE_HANDLE */

/* TypeServer handles */
_DECLARE_HANDLE( TSSERVER );
_DECLARE_HANDLE( TSFONT   );
_DECLARE_HANDLE( TSSTRIKE );


/***************************************************************************
 *          Metagraphics TypeServer API Function Prototypes
 ***************************************************************************
 */

#ifndef  TSEXTERN               /* Define TSEXTERN if it hasn't   */
#define  TSEXTERN  extern       /*   been defined already.        */
#endif /*TSEXTERN*/

#ifndef  MRESULT                /* Define MRESULT if it isn't     */
#define  MRESULT   long         /* function result return code    */
#endif /*MRESULT*/


/* ----------------- TypeServer Management Functions ------------------ */

TSEXTERN
MRESULT tsServer_Create(     /* Open and initialize a TypeServer         */
        MGSYSTEM      *system,   /* in/out, pointer to system-handle     */
        TSSERVER      *server ); /* output, pointer to server-handle     */

TSEXTERN
MRESULT tsServer_Destroy(    /* Close and release a TypeServer           */
        MGSYSTEM      *system,   /* in/out, pointer to system-handle     */
        TSSERVER      *server ); /* in/out, pointer to server-handle     */

TSEXTERN
TSSERVERINFO* tsServer_GetInfoPtr(/* return, pointer to serverInfo       */
        TSSERVER       server,    /* input,  server-handle               */
        int            infoSize); /* input,  TSSERVERINFO struct size    */
```

```
/* --------------------- Font Management Functions --------------------- */

TSEXTERN                        /* Open a font file                      */
MRESULT tsFont_OpenFile(            /* return, completion code(0=success)*/
        TSSERVER       server,      /* input,  server-handle             */
  const TCHAR         *filePathName,/* input,  font path and file name   */
        LONG           fileOffset,  /* input,  offset to start of font   */
        int            fontNumber,  /* input,  font number within file   */
        TSFONT        *font );      /* output, font-handle               */

TSEXTERN                        /* Open a font in memory or ROM          */
MRESULT tsFont_OpenMemory(          /* return, completion code(0=success)*/
        TSSERVER       server,      /* input,  server-handle             */
        void          *fontMemory,  /* input,  font memory address       */
        LONG           fontSize,    /* input,  font memory size          */
        int            fontNumber,  /* input,  font number within file   */
        TSFONT        *font );      /* output, font-handle               */

TSEXTERN                        /* Close and release a font              */
MRESULT tsFont_Destroy(             /* return, completion code(0=success)*/
        TSFONT        *font );      /* in/out, font-handle to close      */

TSEXTERN                        /* Get pointer to font information       */
TSFONTINFO* tsFont_GetInfoPtr(      /* return, pointer to fontInfo       */
        TSFONT         font,        /* input,  font-handle               */
        int            infoSize );  /* input,  TSFONTINFO struct size    */


/* -------------------- Strike Management Functions -------------------- */

TSEXTERN                        /* Create a font strike                  */
MRESULT tsStrike_Create(            /* return, completion code(0=success)*/
        TSFONT         font,        /* input,  font-handle               */
        LONG           fontIndex,   /* input,  font index                */
        MGBITMAP       bitmap,      /* input,  bitmap to render to       */
        TSSTRIKE      *strike );    /* output, pointer to strike-handle  */

TSEXTERN                        /* Destroy a font strike                 */
MRESULT tsStrike_Destroy(           /* return, completion code(0=success)*/
        TSSTRIKE      *strike );    /* in/out, pointer to strike-handle  */

TSEXTERN                        /* Get pointer to strike information     */
TSSTRIKEINFO* tsStrike_GetInfoPtr(  /* return, pointer to strikeInfo     */
        TSSTRIKE       strike,      /* input,  strike-handle             */
        int            infoSize );  /* input,  TSSTRIKEINFO struct size  */


/*    -  -  -  -  -    Strike Attribute Functions    -  -  -  -  -  -    */
TSEXTERN                        /* Set text alignment position           */
MRESULT tsStrike_SetAlign(          /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle             */
        TSALIGN     align );        /* input,  alignment location        */
```

```
TSEXTERN                        /* Set character & background by pixel   */
MRESULT tsStrike_SetColors(         /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        MGCOLORPIX  pixChar,        /* input,  character pixel value       */
        MGCOLORPIX  pixBack );      /* input,  background pixel value      */

TSEXTERN                        /* Set character & background by RGB     */
MRESULT tsStrike_SetColorsRGB(      /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        MGCOLORRGB  rgbChar,        /* input,  character RGB color         */
        MGCOLORRGB  rgbBack );      /* input,  background RGB color        */

TSEXTERN                        /* Set character spacing adjustments     */
MRESULT tsStrike_SetJustify(        /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        FIXDOT      charExtra,      /* input,  charExtra spacing           */
        FIXDOT      wordExtra,      /* input,  wordExtra spacing           */
        TSUNITS     units );        /* input,  units of measure            */

TSEXTERN                        /* Set line spacing                      */
MRESULT tsStrike_SetLineSpacing(    /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        FIXDOT      lineSpacing,    /* input,  line spacing                */
        TSUNITS     units );        /* input,  units of measure            */

TSEXTERN                        /* Set the transfer mode rasterOp        */
MRESULT tsStrike_SetRasterOp(       /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        TSRASTEROP  rasterOp );     /* input,  bitmap transfer mode        */

TSEXTERN                        /* Set edge smoothing                    */
MRESULT tsStrike_SetSmoothing(      /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        TSSMOOTH    smoothLevel );  /* input,  smoothing level             */

TSEXTERN                        /* Set character size                    */
MRESULT tsStrike_SetTypeSize(       /* return, completion code(0=success)*/
        TSSTRIKE    strike,         /* input,  strike-handle               */
        FIXDOT      charHeight,     /* input,  character height            */
        TSUNITS     units );        /* input,  units of measure            */
```

```
/*  -  -  -  -  -  -  -  -   Type Rendering   -  -  -  -  -  -  -  -  -  -
 *
 * TypeServer functions tsStrike_DrawChar() and tsStrike_DrawString()
 * perform character rendering using type TCHAR characters or strings.
 * Dependent if the symbol "_UNICODE" is defined or not, type TCHAR is
 * defined conditionally as either an 8-bit ASCII character or a 16-bit
 * Unicode character. The tsStrike_DrawChar() and tsStrike_DrawString()
 * functions are similarly conditionally defined to reference a specific
 * 8-bit or 16-bit TypeServer drawing function dependent if the symbol
 * "_UNICODE" is defined or not.  The effective prototypes for these
 * functions are as follows:
 *
 * TSEXTERN                     // Render a character                    //
 * MRESULT tsStrike_DrawChar(       // return, completion code(0=success)//
 *         TSSTRIKE     strike,     // input,  strike-handle             //
 *         FIXPOINT    *location,   // in/out, starting X,Y coordinate   //
 *   const TCHAR        character );// input,  character to draw         //
 *
 * TSEXTERN                     // Render a character string             //
 * MRESULT tsStrike_DrawString(     // return, completion code(0=success)//
 *         TSSTRIKE     strike,     // input,  strike-handle             //
 *         FIXPOINT    *location,   // in/out, starting X,Y coordinate   //
 *   const TCHAR       *string,     // input,  characterString to draw   //
 *         int          numChars ); // input,  # of characters to draw   //
 */

#ifndef  CHAR
typedef char CHAR;
#endif /*CHAR*/
#ifndef  WCHAR
typedef unsigned short WCHAR;
#endif /*WCHAR*/

#ifndef  _UNICODE       /*  8-bit ASCII   */
/* typedef char           TCHAR; */
#define tsStrike_DrawChar         tsStrike_DrawCharA
#define tsStrike_DrawString       tsStrike_DrawStringA
#define tsStrike_GetCharExtent    tsStrike_GetCharExtentA
#define tsStrike_GetStringExtent  tsStrike_GetStringExtentA
#else  /*_UNICODE*/     /* 16-bit Unicode */
/* typedef unsigned short TCHAR; */
#define tsStrike_DrawChar         tsStrike_DrawCharW
#define tsStrike_DrawString       tsStrike_DrawStringW
#define tsStrike_GetCharExtent    tsStrike_GetCharExtentW
#define tsStrike_GetStringExtent  tsStrike_GetStringExtentW
#endif /*_UNICODE*/
```

```
#ifndef TSCONFIG_USE_DRAW_MACROS /* use tsStrike_DrawXX functions */

TSEXTERN                        /* Render an 8-bit ASCII character      */
MRESULT tsStrike_DrawCharA(         /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate  */
  const CHAR         character );   /* input,  8-bit ASCII character    */

TSEXTERN                        /* Render a 16-bit Unicode character    */
MRESULT tsStrike_DrawCharW(         /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate  */
  const WCHAR        character );   /* input,  16-bit Unicode character */

TSEXTERN                        /* Render an 8-bit ASCII string         */
MRESULT tsStrike_DrawStringA(       /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate  */
  const CHAR        *string );      /* input,  ASCII string to draw     */

TSEXTERN                        /* Render a 16-bit Unicode string       */
MRESULT tsStrike_DrawStringW(       /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate  */
  const WCHAR       *string );      /* input,  16-bit Unicode string    */

TSEXTERN                        /* Get extent of an ASCII character     */
MRESULT tsStrike_GetCharExtentA(    /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
  const CHAR         character,     /* input,  8-bit ASCII character    */
        FIXSIZE     *extent );      /* output, character width & height */

TSEXTERN                        /* Get extent of a Unicode character    */
MRESULT tsStrike_GetCharExtentW(    /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
  const WCHAR        character,     /* input,  16-bit Unicode character */
        FIXSIZE     *extent );      /* output, character width & height */

TSEXTERN                        /* Get extent of an 8-bit ASCII string  */
MRESULT tsStrike_GetStringExtentA(  /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
  const CHAR        *string,        /* input,  8-bit ASCII string       */
        FIXSIZE     *extent );      /* output, string width & height    */

TSEXTERN                        /* Get extent of a 16-bit Unicode string*/
MRESULT tsStrike_GetStringExtentW(  /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
  const WCHAR       *string,        /* input,  16-bit Unicode string    */
        FIXSIZE     *extent );      /* output, string width & height    */

#else /*TSCONFIG_USE_DRAW_MACROS - use tsStrike_DrawText macros (faster)*/

#define tsStrike_DrawCharA( strike, location, character )    \
        tsStrike_DrawText( strike, location, FALSE, 1, &character )
```

```
#define tsStrike_DrawCharW( strike, location, character )    \
        tsStrike_DrawText( strike, location, TRUE, 1, &character )

#define tsStrike_DrawStringA( strike, location, string )     \
        tsStrike_DrawText( strike, location, FALSE, -1, string )

#define tsStrike_DrawStringW( strike, location, string )     \
        tsStrike_DrawText( strike, location, TRUE, -1, string )

#define tsStrike_GetCharExtentA( strike, character, extent )    \
        tsStrike_GetTextExtent( strike, FALSE, 1, &character, extent )

#define tsStrike_GetCharExtentW( strike, character, extent )    \
        tsStrike_GetTextExtent( strike, TRUE, 1, &character, extent )

#define tsStrike_GetStringExtentA( strike, string, extent )    \
        tsStrike_GetTextExtent( strike, FALSE, -1, string, extent )

#define tsStrike_GetStringExtentW( strike, string, extent )    \
        tsStrike_GetTextExtent( strike, TRUE, -1, string, extent )

#endif /*TSCONFIG_USE_DRAW_MACROS*/
TSEXTERN                        /* Render ASCII or Unicode Characters   */
MRESULT tsStrike_DrawText(          /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
        FIXPOINT    *location,      /* in/out, starting X,Y coordinate  */
        BOOL         isUnicode,     /* input,  TRUE=text is Unicode     */
        int          numChars,      /* input,  character count (-1=all) */
  const void        *text );        /* input,  Unicode/ASCII to draw    */

TSEXTERN                        /* Get the dimensions of a text string  */
MRESULT tsStrike_GetTextExtent(     /* return, result code (0=success)  */
        TSSTRIKE     strike,        /* input,  strike-handle            */
        BOOL         isUnicode,     /* input,  TRUE=text is Unicode     */
        int          numChars,      /* input,  character count (-1=all) */
  const void        *text,          /* input,  Unicode/ASCII text       */
        FIXSIZE     *textSize );    /* output, string width & height    */


#ifdef   __cplusplus
}                       /* - - - - - - - - - - - - - - - - - - - - - - - */
#endif /*__cplusplus*/

#endif /* TYPESERV_H    /*===============================================*/

/* End of File - typeserv.h */
```

Appendix D - Contacting Technical Support

For Additional Support

At Metagraphics our goal is simple - provide the finest in software products and support. When you selected Metagraphics TypeServer, you chose more than just a great world-class product - you also chose a company that believes in standing behind their products with great support. If you have questions about our products or require assistance, we are here to help. However, most of the time you'll find the answer right here in the product manual, on-line help (`\typeserv\help\-typeserv.hlp`), `README.DOC` file (`\typeserv\readme.doc`), or on-line FAQ notes (http://www.metagraphics.com/typeserver/faq/) – for fastest information please check these areas first. If you still have questions, you can reach Metagraphics Technical Support as follows:

- Main Metagraphics TypeServer Web Page:
  http://www.metagraphics.com/typeserver/
- Online Technical FAQ Notices (Frequently Asked Questions):
  http://www.metagraphics.com/typeserver/faq.htm
- Updated and New Online Sample Programs:
  http://www.metagraphics.com/typeserver/files.htm
- Email Support:
  typeserver-support@metagraphics.com
- Fax Support:
  425-844-1112

Direct Metagraphics support is only available to registered product developers. Please be sure to complete and return the registration card provided with your Metagraphics product, or register online at: http://www.metagraphics.com/typeserver/

To speed support, please include the following information with all technical support inquiries:

- Your registered user name, company name, and registered user ID number.
- Your Metagraphics product name and version number.
- Your Metagraphics product serial number.
- Compiler and compiler version you are using (e.g. Microsoft C++ v6.0).

- Platform and platform version (e.g. Phar Lap ETS, v11.0).
- Using the batch, make or IDE files provided, do the sample programs run?

If your question is programming related, a short(!) program sample attached to your email can greatly speed response.

To realize the full benefits of Metagraphics support, we need to know who you are! Please be sure to complete and return you product registration card, or complete the on-line registration at http://www.metagraphics.com/typeserver/ - please take a few moments now if you haven't done so already. Only as a registered user can you realize the full benefits of your Metagraphics product:

- Technical support.
- Notification and download access to free service updates.
- Access to Metagraphics Developer Support Email List-Servers.
- Notification and special pricing on upgrades and new products.
- Subscription to Metagraphics' MetaTRENDS e-letter for current information on service updates, new release updates, programming techniques, and upcoming product release plans.

Index