FUNCTIONTRAK

User Guide

Version 1.0
August 2026

Scott Daughtry
Software by Daughtry

Complexity Made Simple

Table of Contents

Contents

Table of Contents 2

Version History 6

What's New in This Release 6

Introduction 7

Installing FunctionTrak 9

Step 1 - Download the Trial Edition 9

Step 2 - Evaluate the Trial Edition 9

Step 3 - Purchase FunctionTrak 10

Step 4 - Activate the Registered Version 10

Step 5 - Export the Source Code Library 10

Step 6 - Import the Library into WinDev 10

Updating FunctionTrak 11

License Types 11

- Trial 11
- Registered 12

Using the Function Testbed 12

Selecting a Function 13

Reviewing the Example 14

Experimenting with Your Own Values 14

Learning Before Coding 14

From Testbed to Project 14

- Tips for Getting the Most from the Function Testbed 15

Using FunctionTrak Effectively 15

Think in Building Blocks 15

Favor General Solutions 16

Build Reliable Applications 16

Learn Before You Integrate 16

Develop Consistent Coding Habits 17

Function Reference Conventions 17

Procedure Name 17

Purpose 17
Syntax 18
Parameters 18
Return Value 18
Practical Example 18
Developer Tip 19
Related Procedures 19
Notes 19
Naming Conventions 19
Cross References 19
Function Reference 20
Date/Time Function Reference 22
FT_DateToJulian 23
FT_DaysBetween 25
FT_FormatDateInternational 27
FT_FormatDateTimeElapsed 30
FT_FormatDateUS 33
FT_MinutesBetweenDateTimes 36
FT_MonthsDaysBetween 40
FT_YearsMonthsDaysBetween 43
Environment Function Reference 47
Understanding Windows Application Folders 48
Why Are There Two "Program Files" Folders? 50
FT_GetEnvAppDataPath 51
FT_GetEnvComputerName 54
FT_GetEnvCommonProgramFilesPath 57
FT_GetEnvLocalAppDataPath 60
FT_GetEnvLogonServer 63
FT_GetEnvProgramFilesPath 66
FT_GetEnvProgramFilesX86Path 69
FT_GetEnvTempPath 72
FT_GetEnvUserDomain 75
File/Path Function Reference 78
FT_AppendLineToFile 79

FT_ChangeFileExtension 82
FT_ChangeFileName 86
FT_ChangeParentFolderName 90
FT_CountFiles 96
FT_CountFolders 100
FT_GenerateUniqueFileName 104
FT_GetFolderSize 109
FT_IsValidFileName 114
FT_IsValidFolderName 119
FT_ReplaceInFileName 123
FT_SafeFileName 129
FT_SafeFolderName 135
FT_SafePath 141
Financial Function Reference 148
FT_AmountToWords 149
FT_CalculatePercentageOf 155
FT_CalculatePercentage 159
Logic Function Reference 163
FT_IIF 164
FT_IsBetween 167
String Function Reference 171
FT_CleanString 172
FT_PadString 177
FT_LineCount 181
FT_ProperCase 185
FT_RemoveBlankLines 189
FT_SentenceCase 193
FT_Slugify 198
FT_SingleBlankLines 203
FT_StripHTML 208
FT_TitleCase 217
FT_WordWrap 223
Source Code Repository 230
License Protection 235

Developer Notes 237
Manual Revision History 239
Support and Registration 239

Version History

Version	Date	Description
1.0	July 2026	Initial release

What's New in This Release

Welcome to the initial release of **FunctionTrak**, a comprehensive collection of reusable utility procedures designed to simplify application development in WinDev. FunctionTrak provides production-ready functions that solve common programming challenges while handling many of the edge cases often overlooked in sample code.

Version 1.0 includes the following features:

- Approximately **47** reusable utility procedures organized into six functional categories.
- A fully interactive **Function Testbed** application allowing each procedure to be explored and tested without writing application code.
- Comprehensive **English and French** language support throughout the application.
- Production-ready source code protected by **LicenseTrak**, providing registered users with complete implementation details.
- A detailed **Developer Reference Manual** containing syntax, parameter descriptions, return values, practical examples, developer tips, and related functions.
- Hundreds of built-in test cases demonstrating expected behavior under normal and boundary conditions.
- Consistent function naming, return values, and coding conventions across the entire library.

FunctionTrak was designed with a single objective in mind: provide developers with reliable, reusable building blocks that can be combined to solve real-world programming problems quickly and consistently.

Whether you need to manipulate dates, sanitize file paths, calculate elapsed time, retrieve environment information, process strings, or perform common financial calculations, FunctionTrak provides well-tested procedures that can be incorporated directly into your own applications.

This is only the beginning. Future releases will continue to expand the library with additional function categories and enhancements based upon real-world development experience and customer feedback.

Feature	Included
Reusable Functions	✓
Interactive Testbed	✓
English/French Interface	✓
Source Code (Registered)	✓
LicenseTrak Protected	✓
Developer Reference Manual	✓
Practical Examples	✓
Real-World Programming Scenarios	✓

Introduction

Welcome to **FunctionTrak – The WinDev Function Toolbox**.

FunctionTrak is a collection of carefully designed, reusable utility procedures intended to simplify application development in **WinDev**. Rather than repeatedly solving the same programming challenges in every project, FunctionTrak provides a growing library of production-ready functions that can be incorporated directly into your own applications.

Each procedure has been written with three primary goals in mind:

- **Reliability** – Functions are designed to correctly handle both typical input and many real-world edge cases.
- **Consistency** – A common naming convention, coding style, and return value philosophy are used throughout the library.
- **Reusability** – Functions are designed to solve broad classes of programming problems rather than isolated tasks.

Unlike many code libraries that simply provide source code, FunctionTrak also includes an interactive **Function Testbed** that allows developers to execute each procedure, experiment with custom input, and immediately observe the results. This hands-on approach makes it easy to understand how each function behaves before incorporating it into a production application.

Registered users also receive the complete source code library, protected by LicenseTrak, allowing every implementation to be studied, modified, and integrated into commercial or personal projects.

Whether you are developing a small utility or a large enterprise application, FunctionTrak is intended to become a practical toolbox of dependable programming solutions that saves development time, promotes consistent coding practices, and reduces the need to repeatedly reinvent common functionality.

FunctionTrak is built upon a philosophy of small, focused procedures that perform one task exceptionally well and can be combined to solve larger, real-world programming problems.

We hope FunctionTrak becomes a valuable addition to your WinDev development environment.

Why FunctionTrak Exists

Every software developer eventually discovers that many programming problems are solved repeatedly throughout the lifetime of a project. Formatting dates, manipulating strings, validating filenames, calculating elapsed time, sanitizing file paths, retrieving environment information, and performing common business calculations are tasks that appear in application after application.

Although WinDev provides an extensive standard library, many real-world programming problems require additional logic to handle boundary conditions, invalid input, operating system differences, or business-specific requirements. As a result, developers often find themselves rewriting the same utility procedures for each new application they create.

FunctionTrak was created to eliminate that repetition.

Rather than beginning each project with a new collection of utility procedures, FunctionTrak provides a centralized library of reusable, production-ready building blocks that have been designed, tested, and refined for everyday software development.

Every procedure included in FunctionTrak has been written with several guiding principles:

- Solve a genuine programming problem.
- Perform one task exceptionally well.
- Handle common edge cases whenever practical.
- Produce predictable, consistent results.
- Be easy to understand, reuse, and maintain.

FunctionTrak is not intended to replace the WinDev standard library. Instead, it complements it by providing additional functionality that developers frequently find themselves creating independently.

For example, a simple percentage calculation can be used to calculate sales tax, gratuities, commissions, discounts, payroll bonuses, or insurance surcharges. Likewise, a safe filename routine can be used when exporting reports, generating log files, creating backup copies, or importing customer documents. The same reusable building block solves many different business problems.

This philosophy extends throughout the entire library.

Rather than providing hundreds of narrowly focused procedures, FunctionTrak emphasizes versatile, reusable components that can be combined to solve increasingly sophisticated programming tasks. By composing several small procedures together, developers can often implement complex business logic using simple, readable, and easily maintained code.

The objective has never been to create the largest function library available.

The objective has always been to create a library of functions that developers can trust.

During the development of FunctionTrak, every proposed procedure was evaluated against a simple question:

Does this function solve a unique problem, or is it merely another name for functionality that already exists?

If a procedure did not provide meaningful additional capability, it was intentionally omitted. This philosophy keeps the library concise, avoids unnecessary duplication, and encourages developers to learn a small set of versatile building blocks rather than dozens of specialized variations.

Installing FunctionTrak

Installing FunctionTrak consists of two distinct steps:

1. Install the FunctionTrak Trial Edition.
2. Register the product to unlock the protected source code library.

The trial edition contains the complete Function Testbed application, allowing every included function to be explored and evaluated before purchasing the registered version.

Step 1 - Download the Trial Edition

- Download the latest FunctionTrak Trial Edition installer from the **Software by Daughtry** web site.
- Run the installer and follow the on-screen instructions.
- The installer places the FunctionTrak application, documentation, and supporting files onto your computer.
- Once installation is complete, launch FunctionTrak from the Windows Start Menu or desktop shortcut.

Step 2 - Evaluate the Trial Edition

The trial edition allows you to:

- Explore every function included in the library.
- Execute built-in demonstrations.
- Experiment with your own input values.
- Review the included documentation.
- Become familiar with the Function Testbed.

The trial edition does **not** provide access to the protected FunctionTrak source code library.

Step 3 - Purchase FunctionTrak

- When you decide to purchase FunctionTrak, complete the ordering process described in the **Ordering Information** chapter of this manual.
- Following your purchase, a personalized **license.lic** file will be generated and delivered to you.
- This license file unlocks the registered features of FunctionTrak.

Step 4 - Activate the Registered Version

- Close FunctionTrak if it is currently running.
- Replace the existing **license.lic** file located in the FunctionTrak installation folder with the registered license file that was provided to you.
- The next time FunctionTrak is started, the application will automatically detect the registered license and enable the additional features available to registered users.
- No reinstallation is required.

Step 5 - Export the Source Code Library

After activating the registered version:

- Start FunctionTrak.
- Open the **Source Code Library**.
- Select the desired source code module.
- Export the **FunctionTrak.WL** file to a folder of your choice.

The exported file contains the complete WinDev procedure library ready for use within your own projects.

Step 6 - Import the Library into WinDev

To use FunctionTrak in an existing WinDev application:

1. Open your WinDev project.
2. Select **Project → Import Element**.
3. Browse to the exported **FunctionTrak.WL** file.
4. Complete the import wizard.

The FunctionTrak procedures are now available for immediate use within your project.

Updating FunctionTrak

When a newer version of FunctionTrak becomes available:

1. Install the updated FunctionTrak application.
2. Replace the trial **license.lic** file with your registered license (if necessary).
3. Export the updated **FunctionTrak.WL** file.
4. Import the updated library into your WinDev project.

Refer to the **Version History** chapter to review the procedures added or modified in each release.

Important

The FunctionTrak application and the FunctionTrak procedure library are separate components.

Installing the application allows you to explore and test the available procedures. To incorporate those procedures into your own WinDev projects, the FunctionTrak.WL file must first be exported from the Source Code Library and then imported into your project using WinDev's standard import mechanism.

License Types

FunctionTrak is available in two license types: **Trial** and **Registered**. Both editions include the same Function Testbed application, allowing developers to evaluate the available procedures and become familiar with the library. The primary difference between the two editions is access to the protected Source Code Library.

Trial

The Trial Edition is intended to allow developers to fully evaluate FunctionTrak before making a purchase decision. The Trial Edition includes:

- Complete Function Testbed application.
- Access to every function demonstration.
- Built-in examples and test cases.
- English and French language support.
- Complete Developer Reference Manual.

The Trial Edition does **not** include access to the protected Source Code Library. Although every function can be executed and evaluated within the Function Testbed, the FunctionTrak procedure library cannot be exported for use in your own WinDev projects until a registered license has been installed.

The Trial Edition is intended solely for product evaluation.

Registered

The Registered Edition provides unrestricted access to all FunctionTrak features. After purchasing FunctionTrak, you will receive a personalized **license.lic** file generated specifically for your license. Replacing the trial license with the registered license immediately enables the additional registered features.

The Registered Edition includes everything available in the Trial Edition, plus:

- Access to the protected Source Code Library.
- Export capability for the **FunctionTrak.WL** procedure library.
- Complete WinDev source code for all included procedures.
- Free updates for all Version 1.x releases.
- Technical support via e-mail.

Once the **FunctionTrak.WL** library has been exported and imported into your WinDev project, the procedures become part of your application and may be used in accordance with the FunctionTrak license agreement.

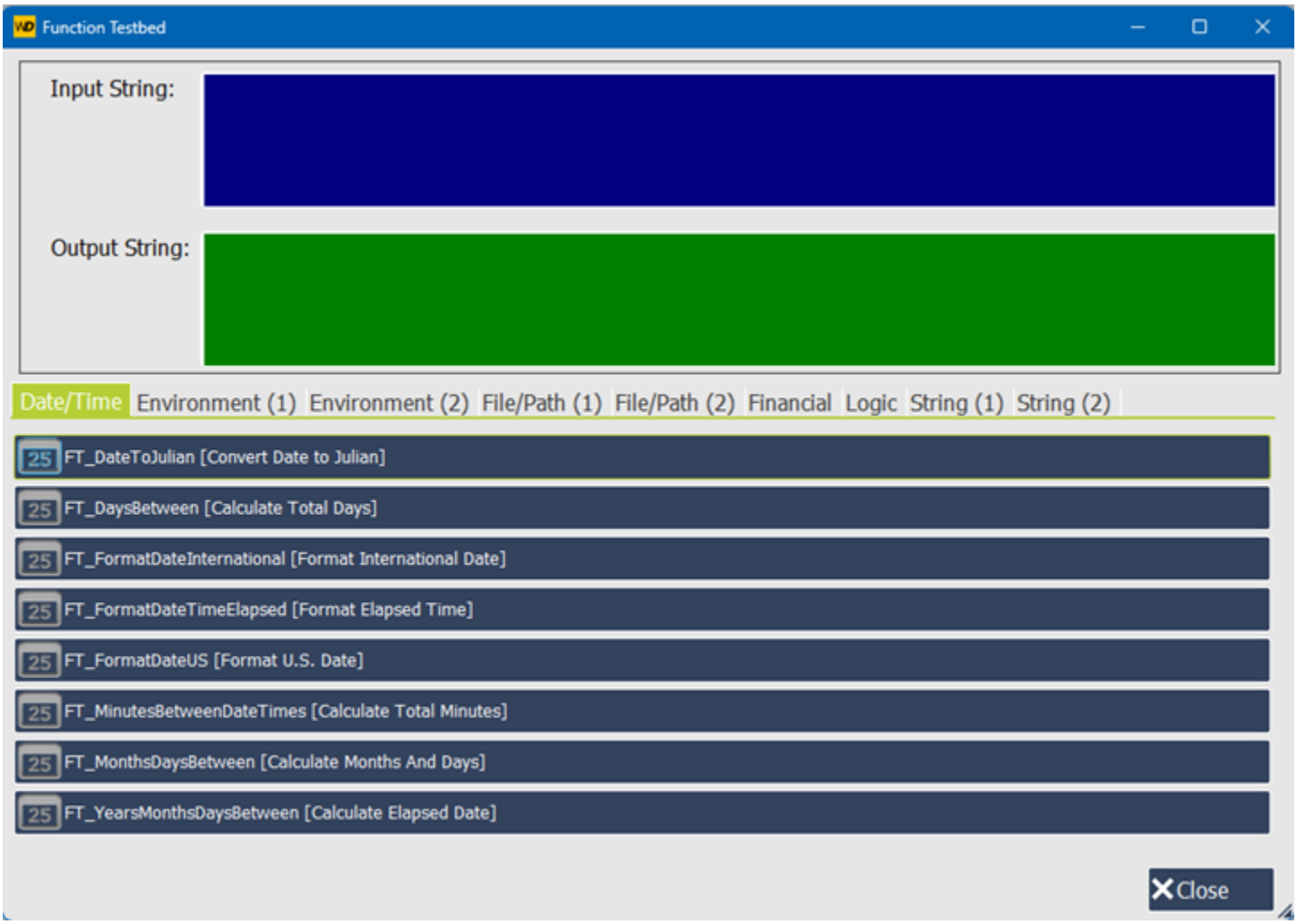
Note

The registered **license.lic** file authorizes access to the FunctionTrak Source Code Library. It does not modify your WinDev projects automatically. To use FunctionTrak procedures within your own applications, export the **FunctionTrak.WL** file from the Source Code Library and import it into your WinDev project as described in the **Installing FunctionTrak** chapter.

Using the Function Testbed

The Function Testbed is the heart of FunctionTrak. Rather than reading source code or documentation alone, the Function Testbed allows you to execute every included procedure, experiment with different input values, and immediately observe the results. This interactive approach makes it easy to understand how each procedure behaves before incorporating it into your own WinDev applications. Whether you are evaluating the Trial Edition or using the Registered Edition, the Function Testbed provides a safe environment for learning the library without writing a single line of code.

The Function Testbed in action:



Selecting a Function

Tabs organize the available functions into logical categories, which include:

- Date/Time
- Environment
- File/Path
- Financial
- Logic
- String

Selecting a category displays the procedures available within that group. Clicking a function button loads a demonstration designed specifically for that procedure. The input and function output are displayed at the top of the screen in vertically scrolling boxes.

Reviewing the Example

Each demonstration includes sample input values that illustrate the procedure's intended use. These examples have been chosen to demonstrate typical usage while also highlighting important behavior or special cases. Simply press the desired function's button on a tab to run the demonstration and review the results.

Experimenting with Your Own Values

One of the primary advantages of the Function Testbed is the ability to experiment. Modify the sample input values with your own data, then execute the procedure again to immediately observe the results. This allows you to answer questions such as:

- How does the procedure handle blank values?
- What happens if invalid data is supplied?
- How are edge cases processed?
- Does the output match the requirements of my application?

Because the demonstrations execute the actual FunctionTrak procedures, the behavior you observe is identical to the behavior you will receive after importing the library into your own projects.

Learning Before Coding

Many developers find it easier to understand a procedure by observing its behavior rather than reading implementation details.

The Function Testbed allows you to become familiar with a procedure before opening the source code.

This approach can significantly reduce development time by allowing you to identify the appropriate procedure, verify its behavior, and understand its expected output before integrating it into your application.

From Testbed to Project

Once you have identified the procedure you wish to use:

1. Open the procedure's documentation in this manual.
2. Review the syntax, parameters, and return values.
3. If you are using the Registered Edition, export the **FunctionTrak.WL** library from the Source Code Library.
4. Import the library into your WinDev project.
5. Begin calling the procedure from your application.

Because every demonstration uses the same production-ready procedures included in the exported library, there are no differences between the behavior observed in the Function Testbed and the behavior within your own application.

Tips for Getting the Most from the Function Testbed

- Execute the built-in demonstrations before trying your own data.
- Experiment with boundary conditions, such as blank strings, zero values, and unusually long input.
- Compare related procedures to understand when each is most appropriate.
- Review the corresponding section of this manual for implementation guidance and practical examples.
- Use the Function Testbed as a quick reference whenever you need to refresh your understanding of a procedure's behavior.

Using FunctionTrak Effectively

FunctionTrak was not designed to replace the WinDev standard library, nor was it intended to become a collection of hundreds of highly specialized procedures. Instead, FunctionTrak follows a simple philosophy:

Provide a concise set of dependable building blocks that can be combined to solve real-world programming problems.

Most applications require the same types of operations repeatedly—validating input, manipulating strings, formatting dates, calculating values, processing filenames, and interacting with the operating system. Rather than writing these routines for every new project, FunctionTrak encourages developers to reuse well-tested procedures that perform these tasks consistently.

The greatest benefit of FunctionTrak is realized when multiple procedures are used together.

Think in Building Blocks

Rather than searching for a procedure that solves an entire business problem, think of each procedure as a building block. For example, exporting a customer report might involve several FunctionTrak procedures working together:

- Retrieve today's date.
- Format the date for use in a filename.
- Remove invalid filename characters.
- Generate a unique filename if one already exists.
- Save the exported report.

No single procedure performs all of these tasks. Instead, several small, focused procedures are combined to create a reliable solution. This approach produces code that is easier to understand, easier to test, and easier to maintain.

Favor General Solutions

Many programming problems appear different on the surface while sharing the same underlying logic. For example, a percentage calculation may be used to determine:

- Sales tax
- Discounts
- Gratuities
- Commissions
- Payroll bonuses
- Credit card convenience fees

Rather than providing separate procedures for each calculation, FunctionTrak supplies a general-purpose percentage procedure that can be applied wherever percentage mathematics is required. This philosophy reduces unnecessary duplication while making the library easier to learn and remember.

Build Reliable Applications

Many programming errors occur not because the primary business logic is incorrect, but because unusual or unexpected input was not considered. When developing applications, consider how your users might unintentionally provide invalid or unexpected data. Examples include:

- Blank strings
- Missing filenames
- Invalid path characters
- Unexpected date formats
- Zero values
- Negative numbers
- Duplicate filenames

FunctionTrak procedures have been designed to handle many common situations consistently, allowing you to spend more time developing your application and less time rewriting utility code.

Learn Before You Integrate

The Function Testbed allows you to experiment with procedures before incorporating them into your own projects. Before writing code:

- Execute the built-in demonstration.
- Modify the sample data.
- Test several boundary conditions.
- Review the procedure's documentation.
- Confirm the output meets your application's requirements.

A few minutes of experimentation often saves considerably more time during debugging.

Develop Consistent Coding Habits

Using a common utility library encourages consistency throughout your applications. Rather than implementing different solutions in different projects, you can rely upon the same procedures and coding conventions every time. This consistency offers several advantages:

- Reduced development time
- Improved code readability
- Easier maintenance
- Fewer duplicate routines
- Greater confidence in tested code

As your collection of applications grows, the benefits of a consistent approach become increasingly valuable.

Function Reference Conventions

To make the FunctionTrak documentation easy to navigate, every procedure is documented using the same format. Understanding these conventions will help you quickly locate the information needed to evaluate and implement each procedure.

Procedure Name

The procedure name appears at the top of each reference page.

Example:

FT_SafeFileName()

Procedure names always begin with the **FT_** prefix to distinguish FunctionTrak procedures from WinDev's native functions and procedures.

Purpose

The **Purpose** section provides a concise description of what the procedure accomplishes and the problem it is intended to solve. This section answers the question:

"Why would I use this procedure?"

Syntax

The **Syntax** section displays the complete procedure declaration exactly as it should appear in your WinDev application. This allows you to quickly identify:

- Required parameters
- Optional parameters (if applicable)
- Return type

Parameters

Each parameter is documented individually. For every parameter you will find:

- Parameter name
- Data type
- Description
- Valid values or expected input (when appropriate)

Understanding the parameters before using a procedure helps prevent incorrect usage and unexpected results.

Return Value

The **Return Value** section describes what the procedure returns after execution.

Depending upon the procedure, this may be:

- A string
- A numeric value
- A Boolean value
- A date or time value
- A status code
- Another supported WinDev data type

Any special return conditions are also described in this section.

Practical Example

Every procedure includes at least one practical example demonstrating its intended use.

Examples are designed to reflect realistic programming scenarios rather than artificial demonstrations. Whenever possible, examples illustrate how the procedure might be incorporated into a production application.

Developer Tip

Many procedures include a **Developer Tip** highlighting useful techniques, common mistakes, performance considerations, or implementation suggestions. These notes are based upon practical development experience and are intended to help you get the most from the procedure.

Related Procedures

Many programming tasks can be accomplished by combining several procedures. The **Related Procedures** section identifies other FunctionTrak procedures that are commonly used together. Exploring these related procedures often leads to simpler and more maintainable solutions.

Notes

Some procedures include additional notes describing special behavior, limitations, operating system considerations, or implementation details that should be understood before using the procedure. When present, these notes should be reviewed carefully.

Naming Conventions

FunctionTrak follows consistent naming conventions throughout the library. These conventions include:

- Every procedure begins with the **FT_** prefix.
- Procedure names use PascalCase for readability.
- Parameter names are descriptive and consistent across related procedures.
- Return values are designed to be predictable whenever practical.

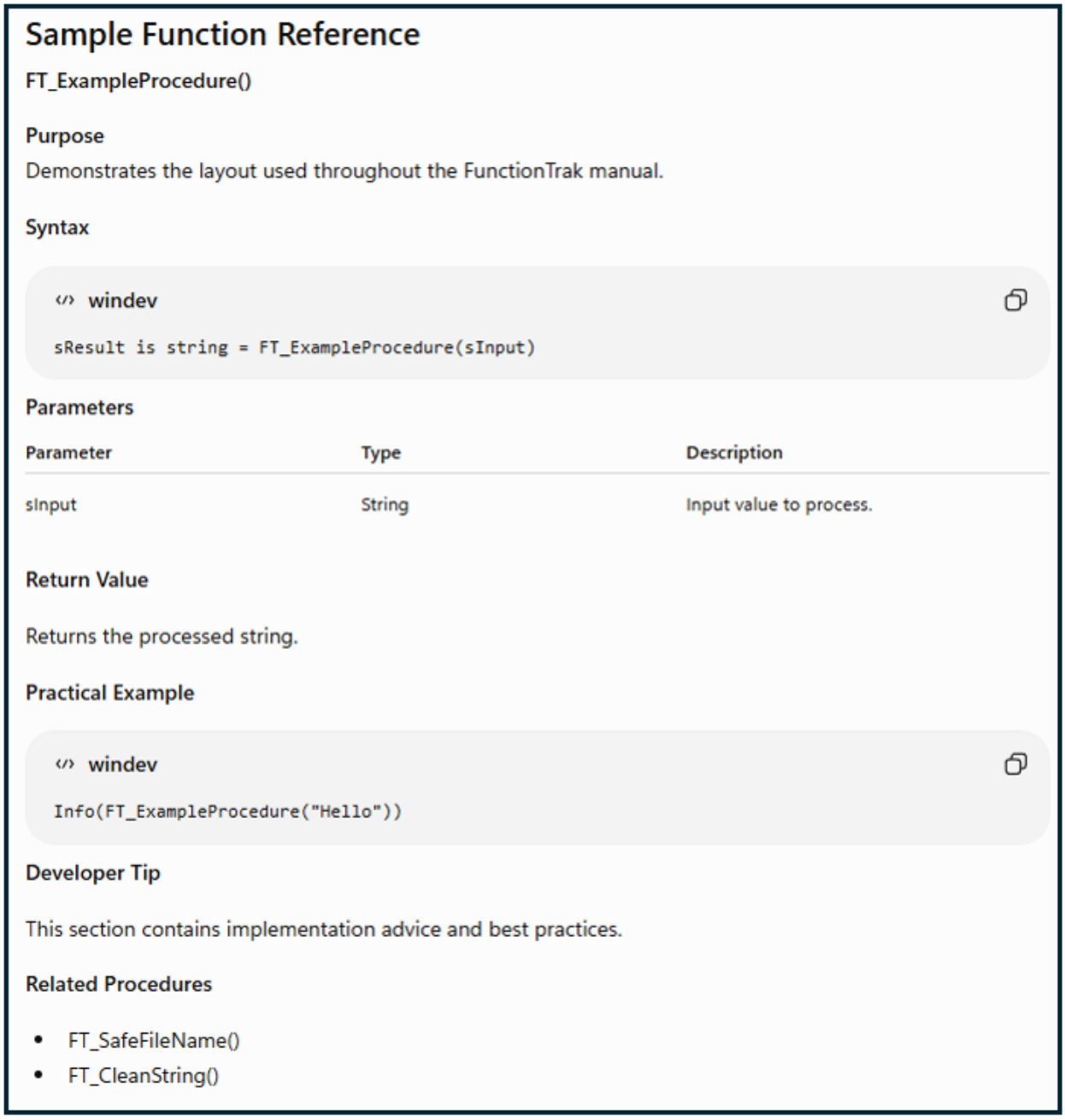
Maintaining consistent naming conventions improves readability and makes procedures easier to remember.

Cross References

Throughout this manual, references to other FunctionTrak procedures are provided whenever additional functionality may be useful.

Developers are encouraged to explore these cross references to discover procedures that work well together.

Many programming tasks are most effectively solved by combining several small procedures rather than relying upon a single specialized routine.



Sample Function Reference

FT_ExampleProcedure()

Purpose

Demonstrates the layout used throughout the FunctionTrak manual.

Syntax

windev

```
sResult is string = FT_ExampleProcedure(sInput)
```

Parameters

Parameter	Type	Description
sInput	String	Input value to process.

Return Value

Returns the processed string.

Practical Example

windev

```
Info(FT_ExampleProcedure("Hello"))
```

Developer Tip

This section contains implementation advice and best practices.

Related Procedures

- FT_SafeFileName()
- FT_CleanString()

Function Reference

The chapters that follow contain the complete reference documentation for every procedure included in FunctionTrak.

Procedures are organized into functional categories to make it easy to locate the functionality needed for a particular programming task. Within each category, every procedure is documented using the standard reference format described in the **Function Reference Conventions** chapter.

Each procedure reference includes:

- Purpose
- Syntax
- Parameters
- Return Value
- Practical Example
- Developer Tip (where applicable)
- Related Procedures
- Additional Notes (when appropriate)

The examples presented throughout this reference are intended to demonstrate practical, real-world usage rather than isolated programming techniques. Whenever possible, procedures are shown in the context of common development tasks to illustrate not only what each procedure does, but also how it can be incorporated into your own WinDev applications.

Although each procedure can be used independently, many of the most effective solutions are achieved by combining multiple FunctionTrak procedures. As you become familiar with the library, you will discover that small, focused procedures can be composed into powerful and maintainable solutions for a wide variety of programming challenges.

The following chapters contain the complete FunctionTrak procedure reference.

Date/Time Function Reference

The Date/Time procedures simplify many of the common calendar and time calculations encountered during application development. These procedures provide consistent handling of dates, elapsed time, formatting, and date arithmetic while reducing the need for repetitive utility code.

FT_DateToJulian

Purpose
Converts a WinDev **Date** value into its Julian day number equivalent, providing a simple integer representation that is ideal for date comparisons and elapsed-time calculations. Rather than performing calculations against formatted date strings, Julian day numbers allow dates to be compared using straightforward integer arithmetic, resulting in cleaner, more maintainable code.

Why This Procedure?
Although WinDev provides extensive date manipulation capabilities, many programming tasks become considerably simpler when dates are converted into a sequential numeric value.

FT_DateToJulian() was developed to provide a dependable conversion routine that can serve as the foundation for many other date-related calculations.

Key Features
✓ Correctly handles leap years.
✓ Supports the Gregorian calendar.
✓ Returns a simple integer suitable for mathematical calculations.
✓ Eliminates the need to perform arithmetic on formatted date strings.
✓ Serves as the foundation for several other FunctionTrak date procedures.

Syntax
nJulian is int = FT_DateToJulian(dDate)

Parameters

Parameter	Type	Description
dDate	Date	The WinDev Date value to convert.

Return Value

Type	Description
Integer	The Julian day number representing the specified date.

Remarks
FT_DateToJulian() converts a calendar date into a sequential day number that can be compared, sorted, or used in arithmetic operations.

Because the return value is an integer, determining the number of days between two dates becomes as simple as subtracting one Julian value from another.

Leap years are automatically taken into account during the conversion process, ensuring accurate calculations across month and year boundaries.

Real-World Uses

FT_DateToJulian() is useful whenever date arithmetic is required, including:

- Calculating employee length of service.
- Determining warranty or contract expiration dates.
- Computing the age of invoices or accounts receivable.
- Measuring project durations.
- Calculating document retention periods.
- Comparing dates without regard to formatting.
- Building historical reports or timelines.

Practical Example

An HR application needs to determine how many days an employee has been employed.

```
dHireDate is Date = "20180514"

nDaysWorked is int = FT_DateToJulian(Today()) - FT_DateToJulian(dHireDate)

Info("Employee has been employed for " + nDaysWorked + " days.")
```

Developer Tip

TIP: Julian dates are intended for calculations—not for display. Perform your calculations using the Julian values, then present the original calendar dates to the user in the desired format.

Common Pitfalls

1) Developers sometimes attempt to calculate elapsed time by manipulating formatted date strings or assuming every year contains exactly 365 days. Such approaches fail whenever leap years are encountered.

2) FT_DateToJulian() eliminates this problem by converting dates into a continuous day sequence before any calculations are performed.

Related Procedures

Procedure	Why You Might Use It
FT_DaysBetween()	Returns the elapsed number of days between two dates.
FT_MonthsDaysBetween()	Calculates elapsed months and remaining days.
FT_YearsMonthsDaysBetween()	Calculates elapsed years, months, and days.
FT_FormatDateUS()	Formats dates for display after calculations.
FT_FormatDateInternational()	Formats dates using an international standard.

FT_DaysBetween

Determine the exact number of elapsed days between two dates—without worrying about leap years, month lengths, or calendar arithmetic.

Purpose
Calculates the number of elapsed calendar days between two WinDev **Date** values and returns the result as an integer. FT_DaysBetween() eliminates the complexity of manually calculating date differences, automatically accounting for leap years, varying month lengths, and year boundaries.

Why This Procedure?
Calculating the number of days between two dates appears simple until your application spans February, leap years, different month lengths, or multiple calendar years. Rather than requiring developers to write and maintain their own date arithmetic, **FT_DaysBetween()** provides a dependable solution that produces accurate results regardless of the calendar dates supplied.

Key Features
- Automatically handles leap years.
- Correctly processes varying month lengths.
- Supports calculations across multiple calendar years.
- Returns positive or negative values depending on date order.
- Internally utilizes FT_DateToJulian() for consistent and reliable results.

Syntax
nDays is int = FT_DaysBetween(dStartDate, dEndDate)

Parameters

Parameter	Type	Description
dStartDate	Date	Beginning date.
dEndDate	Date	Ending date.

Return Value

Type	Description
Integer	Number of elapsed days between the two dates. Positive values indicate the ending date occurs after the starting date; negative values indicate the reverse.

Remarks
FT_DaysBetween() performs true calendar calculations rather than relying on approximations.
Because the calculation is based upon Julian day numbers, leap years and month boundaries are automatically handled without requiring any additional code.

The function may return either a positive or negative value depending on the order of the supplied dates, making it equally useful for determining both elapsed and remaining time.

Real-World Uses
FT_DaysBetween() is commonly used to:

- Calculate employee seniority.
- Determine invoice aging.
- Measure project durations.
- Track equipment warranty periods.
- Calculate subscription or license expiration.
- Determine the number of days until or since an event.
- Validate service-level agreements (SLAs).

Practical Example
Determine how many days remain before a software maintenance agreement expires.

```
dExpirationDate is Date = "20271231"
nDaysRemaining is int = FT_DaysBetween(Today(), dExpirationDate)
IF nDaysRemaining < 30 THEN
   Info("Maintenance expires in " + nDaysRemaining + " days.")
END
```

Developer Tip
If your application only needs the total number of elapsed days, use FT_DaysBetween() rather than calculating the difference yourself. Reserve FT_YearsMonthsDaysBetween() for situations where a human-readable result (such as "3 years, 2 months, 11 days") is required.

Common Pitfalls
Many developers assume that every month contains 30 days or every year contains 365 days. Those assumptions produce incorrect results whenever leap years or varying month lengths are involved. FT_DaysBetween() performs true calendar calculations and avoids these common errors.

Related Procedures

Procedure	Why You Might Use It
FT_DateToJulian()	Converts dates into sequential day numbers used internally by this procedure.
FT_MonthsDaysBetween()	Returns elapsed months and remaining days.
FT_YearsMonthsDaysBetween()	Returns elapsed years, months, and days.
FT_FormatDateUS()	Formats dates for U.S. presentation.
FT_FormatDateInternational()	Formats dates using international conventions.

FT_FormatDateInternational

Convert a WinDev Date value into the universally recognizable YYYY-MM-DD format—ideal for data exchange, filenames, logs, sorting, and technical records.

Purpose
Converts a valid WinDev **Date** value into an international date string formatted as: YYYY-MM-DD

The procedure defaults to the current system date when no date is supplied and returns an empty string when the supplied value is invalid.

Why This Procedure?
Date formats such as MM/DD/YYYY and DD/MM/YYYY can be ambiguous. A value such as 03/07/2026 may mean March 7 or July 3, depending on the reader's regional convention.

FT_FormatDateInternational() removes that ambiguity by producing a consistent year-month-day representation. Because the most significant component appears first, dates formatted this way also sort correctly as ordinary text.

Key Features

- Produces the consistent YYYY-MM-DD format.
- Eliminates ambiguity between U.S. and European date conventions.
- Preserves chronological order when formatted values are sorted alphabetically.
- Defaults to the current system date when no parameter is supplied.
- Validates the supplied date before formatting it.
- Returns an empty string instead of producing misleading output from an invalid date.
- Uses fixed-position extraction, ensuring predictable output without relying on workstation regional settings.

Syntax
sFormattedDate is string = FT_FormatDateInternational(dDate)

The date parameter may be omitted: sFormattedDate is string = FT_FormatDateInternational()

Parameters

Parameter	Type	Description
dDate	Date	Date to format. Defaults to the current system date when omitted.

Return Value

Type	Description
String	The formatted date in YYYY-MM-DD format. Returns an empty string when the supplied date is invalid.

Remarks
WinDev stores Date values internally in YYYYMMDD order. FT_FormatDateInternational() inserts hyphens between the year, month, and day components to create a clear, standardized display value. The procedure does not alter the original Date variable. It returns a new string intended for display, export, logging, filenames, or other text-based uses. Because the returned format begins with the year, a collection of these strings sorts naturally into chronological order:

- 2024-11-18
- 2025-03-02
- 2026-07-28

This makes the procedure especially useful wherever dates are stored or displayed as text.

Real-World Uses
FT_FormatDateInternational() is useful for:

- Exporting records to CSV, XML, JSON, or other exchange formats.
- Writing unambiguous dates into application logs.
- Creating date-based filenames and folder names.
- Displaying dates in technical reports.
- Passing dates between systems located in different countries.
- Generating sortable text values.
- Recording audit-trail and transaction dates.

Practical Example
Create a dated backup filename that sorts chronologically within Windows Explorer.

```
sBackupFile is string

sBackupFile = "CustomerBackup_" + FT_FormatDateInternational(Today()) + ".zip"
```

A backup created on July 28, 2026 would be named: CustomerBackup_2026-07-28.zip

Because the date is arranged from year to month to day, multiple backup files remain in chronological order when sorted by filename.

Developer Tip
Use FT_FormatDateInternational() when dates will be exchanged between applications, embedded in filenames, written to logs, or viewed by users in multiple countries. Use FT_FormatDateUS() when presenting a date specifically for a U.S. audience.

Common Pitfalls
1) A formatted date string is not the same as a WinDev Date value. Use the original Date variable for date arithmetic and comparisons; use the returned string for display, export, or storage in text-based formats.

2) Do not manually concatenate the year, month, and day throughout an application. Centralizing that behavior in FT_FormatDateInternational() ensures every screen, report, export, and log uses the same format and invalid-date policy.

Related Procedures

Procedure	Why You Might Use It
FT_FormatDateUS()	Formats a Date value using the familiar U.S. month-day-year convention.
FT_DateToJulian()	Converts a Date value into a numeric day representation for calculations.
FT_DaysBetween()	Calculates the signed number of days between two dates.
FT_FormatDateTimeElapsed()	Converts an elapsed DateTime interval into readable days, hours, and minutes.

FT_FormatDateTimeElapsed

Calculate and format the elapsed time between two WinDev DateTime values as a naturally readable string—ideal for logs, reports, performance statistics, and user-friendly status messages.

Purpose
Calculates the elapsed time between two WinDev **DateTime** values and returns the result as a naturally readable string.

Rather than requiring developers to calculate elapsed minutes and manually format the results, **FT_FormatDateTimeElapsed()** performs both operations automatically, producing output such as:
2 days, 5 hours, 17 minutes

The ending DateTime parameter is optional. When omitted, the procedure automatically uses the current system date and time, making it ideal for measuring the duration of ongoing operations.

Why This Procedure?
Applications frequently need to measure elapsed time for operations such as database rebuilds, software installations, backup processes, report generation, license usage, or application uptime. While calculating elapsed time is relatively straightforward, converting that value into a clean, grammatically correct string often results in repetitive formatting code being duplicated throughout an application.

FT_FormatDateTimeElapsed() eliminates that effort by calculating the elapsed interval, handling the formatting, and returning a concise, professional-looking result.

Key Features
- Automatically calculates the elapsed time between two DateTime values.
- Ending DateTime is optional and defaults to the current system date and time.
- Automatically separates the interval into days, hours, and minutes.
- Omits zero-value components for cleaner output.
- Produces grammatically correct singular and plural wording.
- Supports both positive and negative elapsed intervals.
- Validates all supplied DateTime values before processing.
- Returns an empty string when either DateTime is invalid.

Syntax
```
sElapsed is string = FT_FormatDateTimeElapsed(dtStartDateTime)

sElapsed is string = FT_FormatDateTimeElapsed(dtStartDateTime, dtEndDateTime)
```

Parameters

Parameter	Type	Description
dtStartDateTime	DateTime	Beginning DateTime of the elapsed interval.
dtEndDateTime	DateTime	Optional ending DateTime. If omitted, the current system date and time are used.

Return Value

Type	Description
String	A formatted elapsed time string. Returns an empty string if either supplied DateTime is invalid.

Remarks

FT_FormatDateTimeElapsed() internally calculates the number of elapsed minutes between the supplied DateTime values before converting the result into a naturally readable string.

Only non-zero components are included in the returned value, producing concise output. For example:
3 hours instead of **0 days, 3 hours, 0 minutes**

If both DateTime values are identical, the function returns: 0 minutes

When the ending DateTime occurs before the starting DateTime, the returned string is prefixed with a minus sign to preserve the direction of the elapsed interval. For example:
-2 days, 6 hours

This allows the calling application to determine whether the elapsed interval represents past or future time without losing information.

Real-World Uses

FT_FormatDateTimeElapsed() is useful for:

- Displaying software installation times.
- Measuring database maintenance operations.
- Reporting backup and restore durations.
- Displaying report generation statistics.
- Measuring import and export processing time.
- Recording application uptime.
- Showing license or subscription usage periods.
- Displaying the elapsed time since a user logged into an application.

Practical Example

Display the amount of time required to rebuild a database.

```
dtStart is DateTime = DateSys() + TimeSys()
// Rebuild the database...
Info("Database rebuild completed in " + FT_FormatDateTimeElapsed(dtStart))
```

Possible output: "Database rebuild completed in 3 minutes, 42 seconds"

If both the starting and ending DateTime values are known:
Info(FT_FormatDateTimeElapsed(dtStarted, dtFinished))

Developer Tip
Whenever possible, allow FT_FormatDateTimeElapsed() to determine the ending DateTime automatically. This reduces code and guarantees that the elapsed interval is measured using the current system date and time at the moment the function is called.

Common Pitfalls
1) Developers often calculate elapsed time correctly but repeatedly duplicate formatting logic throughout their applications. Centralizing this functionality in FT_FormatDateTimeElapsed() ensures that every screen, report, and log presents elapsed time using the same consistent format.

2) Remember that the returned value is intended for display purposes only. If additional calculations are required, perform those calculations using the original DateTime values rather than the formatted string.

Related Procedures

Procedure	Why You Might Use It
FT_MinutesBetweenDateTimes()	Returns the signed number of elapsed minutes used internally by this procedure.
FT_DaysBetween()	Calculates the elapsed number of calendar days between two dates.
FT_MonthsDaysBetween()	Returns elapsed months and remaining days.
FT_YearsMonthsDaysBetween()	Returns elapsed years, months, and days.
FT_FormatDateUS()	Formats Date values using the U.S. month/day/year convention.
FT_FormatDateInternational()	Formats Date values using the international year-month-day convention.

FT_FormatDateUS

Convert a WinDev Date value into the familiar U.S. MM/DD/YYYY format with built-in validation and a convenient current-date default.

Purpose
Formats a valid WinDev **Date** value using the standard U.S. month/day/year convention: MM/DD/YYYY

When no date is supplied, **FT_FormatDateUS()** automatically formats the current system date. If the supplied Date value is invalid, the procedure returns an empty string.

Why This Procedure?
WinDev Date values are stored internally in YYYYMMDD format, which is ideal for processing but not always suitable for display to U.S. users.

Developers often end up repeatedly extracting the month, day, and year components wherever a date must be shown in a message, report, export, or form.

FT_FormatDateUS() centralizes that work in one predictable procedure. It validates the date, applies a consistent display format, and avoids duplicating formatting logic throughout an application.

Key Features
- Converts WinDev Date values into MM/DD/YYYY format.
- Defaults to the current system date when the parameter is omitted.
- Validates the supplied Date value before formatting it.
- Returns an empty string when the date is invalid.
- Produces fixed-width month and day components with leading zeroes.
- Does not depend on the workstation's regional display settings.
- Provides consistent date presentation throughout an application.

Syntax
sFormattedDate is string = FT_FormatDateUS(dDate)

The date parameter may be omitted:
sFormattedDate is string = FT_FormatDateUS()

Parameters

Parameter	Type	Description
dDate	Date	Date to format. Defaults to the current system date when omitted.

Return Value

Type	Description
String	The supplied date formatted as MM/DD/YYYY. Returns an empty string when the Date value is invalid.

Remarks
WinDev stores Date values in YYYYMMDD order. **FT_FormatDateUS()** extracts the month, day, and year portions and rearranges them into the U.S. display convention. For example, the WinDev Date value: **20260728** is returned as: **07/28/2026**

The procedure does not alter the original Date variable. It returns a new string intended for display, reporting, exporting, logging, or other text-based uses. Because formatting is performed through fixed-position extraction, the result remains consistent regardless of the Windows regional settings configured on the workstation.

Real-World Uses
FT_FormatDateUS() is useful for:

- Displaying dates to U.S.-based users.
- Formatting dates in reports and printed forms.
- Creating user-facing confirmation messages.
- Writing readable dates into application logs.
- Exporting data intended for U.S. recipients.
- Displaying invoice, payment, order, and transaction dates.
- Standardizing date presentation across windows and reports.

Practical Example
Display the date on which a customer record was created.

```
sMessage is string
sMessage = "Customer record created on " + FT_FormatDateUS(Customer.DateCreated)
Info(sMessage)
```

Possible output: "Customer record created on 07/28/2026"

To display the current system date, no parameter is required: Info("Today's date is " + FT_FormatDateUS())

Developer Tip

Use FT_FormatDateUS() for dates intended primarily for U.S. users. Use FT_FormatDateInternational() for filenames, technical records, data exchange, or situations where chronological text sorting and international clarity are more important.

Common Pitfalls
1) A formatted date string is not a WinDev Date value. Continue using the original Date variable for comparisons, calculations, searches, and database operations. Use the returned string only when a text representation is needed.

2) The U.S. MM/DD/YYYY convention may be ambiguous to international users. A value such as 04/07/2026 can be interpreted as either April 7 or July 4. Use FT_FormatDateInternational() when the audience or destination system may not assume U.S. date conventions.

3) Do not repeatedly assemble formatted dates throughout the application with Middle(), Right(), and Left().

Using FT_FormatDateUS() ensures that validation and formatting behavior remain consistent everywhere.

Related Procedures

Procedure	Why You Might Use It
FT_FormatDateInternational()	Formats a Date value as YYYY-MM-DD for international use, filenames, logs, and chronological text sorting.
FT_DateToJulian()	Converts a Date value into a sequential numeric day value for date calculations.
FT_DaysBetween()	Returns the signed number of elapsed days between two dates.
FT_MonthsDaysBetween()	Expresses the interval between two dates as months and remaining days.
FT_YearsMonthsDaysBetween()	Expresses the interval between two dates as years, months, and remaining days.

FT_MinutesBetweenDateTimes

Calculate the signed number of elapsed minutes between two WinDev DateTime values—with built-in validation and an optional ending time that defaults to right now.

Purpose
Calculates the total number of elapsed minutes between two WinDev **DateTime** values. The result is signed, preserving the direction of the interval:

- A positive value indicates that the ending DateTime occurs after the starting DateTime.
- A negative value indicates that the ending DateTime occurs before the starting DateTime.
- Zero indicates that both DateTime values represent the same minute.

When the ending DateTime is omitted, **FT_MinutesBetweenDateTimes()** automatically uses the current system date and time.

Why This Procedure?
Calculating elapsed time between DateTime values usually involves creating a WinDev **Duration**, extracting its minute value, validating both inputs, and deciding how to handle an omitted ending time. **FT_MinutesBetweenDateTimes()** packages those steps into a single, reusable call.

The procedure is especially convenient for measuring operations that began at a known time and are still in progress. The developer supplies only the starting DateTime, and FunctionTrak calculates the elapsed minutes through the moment the procedure is called.

Key Features

- Calculates the total elapsed minutes between two DateTime values.
- Returns a signed result that preserves the direction of the interval.
- Accepts an optional ending DateTime.
- Defaults the ending DateTime to the current system date and time when omitted.
- Validates both DateTime values before performing the calculation.
- Returns -1 when either DateTime value is invalid.
- Uses WinDev's native Duration handling for the underlying calculation.
- Serves as the calculation engine for **FT_FormatDateTimeElapsed()**.

Syntax
nMinutes is int = FT_MinutesBetweenDateTimes(dtStartDateTime, dtEndDateTime)

The ending DateTime may be omitted:
nMinutes is int = FT_MinutesBetweenDateTimes(dtStartDateTime)

Parameters

Parameter	Type	Description
dtStartDateTime	DateTime	Beginning DateTime of the interval.
dtEndDateTime	DateTime	Optional ending DateTime. When omitted, the current system date and time are used.

Return Value

Type	Description
Integer	The signed total number of elapsed minutes between the two DateTime values. Returns -1 if either DateTime value is invalid.

Remarks

Subtracting one WinDev DateTime value from another produces a **Duration**. FT_MinutesBetweenDateTimes() performs that subtraction and returns the Duration's total minute value through its InMinutes property.
For example, an interval spanning one day and two hours returns: **1560**
because the complete interval contains 1,560 minutes.

The returned value is signed. If the ending DateTime occurs before the starting DateTime, the result is negative rather than being converted to an absolute value. For example:

```
dtStart  is DateTime = "20260728150000"
dtFinish is DateTime = "20260728130000"

nMinutes is int = FT_MinutesBetweenDateTimes(dtStart, dtFinish)
```

The procedure returns: **-120**

This behavior preserves the direction of the interval and can help expose accidentally reversed parameters.
When the ending DateTime is omitted, the procedure uses: DateSys() + TimeSys()

This makes it possible to measure an ongoing interval without separately creating a current DateTime value.

Real-World Uses

FT_MinutesBetweenDateTimes() is useful for:

- Measuring the duration of database operations.
- Tracking application or user-session uptime.
- Calculating employee or equipment usage time.
- Determining how long a record has remained pending.
- Enforcing timeout or expiration rules.
- Measuring import, export, backup, or reporting operations.
- Calculating minutes remaining until a scheduled event.
- Supplying elapsed-minute values to other FunctionTrak procedures.

Practical Example
Determine how many minutes a user has been logged into an application.

```
dtLoginTime is DateTime
nElapsedMinutes is int

// dtLoginTime was recorded when the user logged in
nElapsedMinutes = FT_MinutesBetweenDateTimes(dtLoginTime)
IF nElapsedMinutes >= 0 THEN
   Info("You have been logged in for " + nElapsedMinutes + " minutes.")
ELSE
   Error("The login DateTime is invalid.")
END
```

Because the ending DateTime is omitted, the procedure calculates the interval between dtLoginTime and the current system date and time. To calculate a completed interval, supply both values:

```
nProcessingMinutes is int
nProcessingMinutes = FT_MinutesBetweenDateTimes(dtProcessStarted, dtProcessFinished)
```

Developer Tip
1) Use FT_MinutesBetweenDateTimes() when the numeric minute value will be used in calculations, comparisons, timeout rules, or database storage.

2) Use **FT_FormatDateTimeElapsed()** when the result will be displayed to a user as readable text such as: 2 days, 3 hours, 15 minutes

Common Pitfalls
A negative result does not automatically indicate an error. It may simply mean that the ending DateTime occurs before the starting DateTime. For example, a valid reversed interval of one minute also returns: **-1**

However, -1 is also the procedure's invalid-input return value. Therefore, a caller that must distinguish an invalid DateTime from a legitimate negative one-minute interval should validate the DateTime values separately before calling the procedure.

```
IF DateTimeValid(dtStart) AND DateTimeValid(dtFinish) THEN
   nMinutes = FT_MinutesBetweenDateTimes(dtStart, dtFinish)
ELSE
   Error("An invalid DateTime value was supplied.")
END
```

COMMON PITFALLS
1) Do not apply Abs() to the returned value unless the direction of the interval is genuinely irrelevant. The sign may reveal reversed parameters or distinguish elapsed time from time remaining.
2) The procedure returns total elapsed minutes, not merely the minute component of the interval. An interval of two hours and fifteen minutes returns 135, not 15.

Related Procedures

Procedure	Why You Might Use It
FT_FormatDateTimeElapsed()	Calculates the same interval and presents it as readable days, hours, and minutes.
FT_DaysBetween()	Returns the signed number of elapsed days between two Date values.
FT_MonthsDaysBetween()	Expresses a date interval as complete months and remaining days.
FT_YearsMonthsDaysBetween()	Expresses a date interval as years, months, and remaining days.
FT_DateToJulian()	Converts a Date value into a sequential day number for date-based calculations.

FT_MonthsDaysBetween

Express the elapsed calendar time between two dates as complete months and remaining days—while correctly handling unequal month lengths and month-end dates.

Purpose
Calculates the elapsed calendar time between two WinDev **Date** values and returns a naturally readable string containing complete months and any remaining days. For example: **2 months, 5 days**

When the ending date is omitted, **FT_MonthsDaysBetween()** automatically uses the current system date. The procedure returns an empty string when either date is invalid or when the ending date occurs before the starting date.

Why This Procedure?
A month is not a fixed number of days. Depending on the calendar month, it may contain 28, 29, 30, or 31 days. Because of this, dividing an elapsed-day total by 30 does not produce a reliable calendar interval. It can yield misleading results around February, leap years, and month-end dates.

FT_MonthsDaysBetween() performs true calendar-based counting. It advances through complete months first and then calculates the remaining elapsed days, producing a result that reflects how people naturally describe calendar time.

Key Features
- Calculates complete calendar months between two dates.
- Calculates any remaining elapsed days after the complete months.
- Correctly accommodates months with different numbers of days.
- Handles month-end transitions such as January 31 to February 28.
- Automatically accounts for leap years through calendar-based date processing.
- Defaults the ending date to the current system date when omitted.
- Produces grammatically correct singular and plural wording.
- Omits zero-value components for concise output.
- Returns 0 days when both dates are identical.
- Validates both Date values before processing.
- Returns an empty string when the ending date precedes the starting date.

Syntax
sElapsed is string = FT_MonthsDaysBetween(dStartDate, dEndDate)

The ending date may be omitted:
sElapsed is string = FT_MonthsDaysBetween(dStartDate)

Parameters

Parameter	Type	Description
dStartDate	Date	Beginning date of the elapsed period.
dEndDate	Date	Optional ending date. Defaults to the current system date when omitted.

Return Value

Type	Description
String	A readable elapsed-calendar string containing complete months and remaining days. Returns an empty string if either date is invalid or the ending date precedes the starting date.

Remarks

FT_MonthsDaysBetween() does not estimate months by dividing the number of elapsed days by a fixed value. Instead, it begins with the starting date and advances one complete calendar month at a time. Once another complete month would extend beyond the ending date, the procedure uses **FT_DaysBetween()** to calculate the remaining days. For example: 01/15/2026 through 03/20/2026 returns: **2 months, 5 days.** Month-end dates receive calendar-aware treatment. For example: 01/31/2026 through 02/28/2026 returns: **1 month** Likewise: 08/31/2026 through 09/30/2026 returns: **1 month**

These results would be difficult to reproduce accurately by assuming that every month contains 30 days.
If the two supplied dates are identical, the procedure returns: 0 days

Only non-zero month and day components are included. Therefore, an exact two-month interval returns: 2 months rather than: 2 months, 0 days.

The procedure accepts only forward-moving or identical date ranges. If dEndDate occurs before dStartDate, an empty string is returned.

Real-World Uses

FT_MonthsDaysBetween() is useful for:

- Displaying the age of a pending request or support case.
- Showing the duration of a subscription or service period.
- Calculating how long an employee has occupied a position.
- Reporting the age of equipment, inventory, or customer records.
- Measuring the time between inspections or maintenance events.
- Displaying contract, lease, or warranty durations.
- Showing the time elapsed since an application event.
- Presenting calendar intervals in reports and status messages.

Practical Example
Display how long a customer's support request has remained open.

```
sElapsed is string
sElapsed = FT_MonthsDaysBetween( Support.DateOpened, DateSys())
IF sElapsed <> "" THEN
   Info("This support request has been open for " + ...
      sElapsed + ".")
ELSE
   Error("The support dates are invalid.")
END
```

Possible output: "This support request has been open for 3 months, 12 days."

Because the ending date defaults to the current system date, the call may also be shortened to:
sElapsed = FT_MonthsDaysBetween(Support.DateOpened)

Developer Tip
Use FT_MonthsDaysBetween() when the result should reflect recognizable calendar boundaries rather than a raw number of elapsed days. For example, users generally understand: 1 month more naturally than: 28 days when describing the interval from January 31 through February 28.

Use **FT_DaysBetween()** when an exact numeric day count is required for calculations, deadlines, or comparisons.

Common Pitfalls
1) Do not calculate calendar months by dividing an elapsed-day total by 30 or 30.44. That produces an estimate, not a true calendar interval, and may give confusing results around February and month-end dates.

2) The procedure does not return negative intervals. If the ending date occurs before the starting date, it returns an empty string. Verify the date order when processing user-entered values.

3) The returned value is formatted display text. Do not attempt to use it directly for mathematical comparisons or additional date calculations. Retain the original Date values for those operations.

4) A result of 1 month does not necessarily represent a fixed number of elapsed days. Depending on the supplied dates, one calendar month may span 28, 29, 30, or 31 days.

Related Procedures

Procedure	Why You Might Use It
FT_YearsMonthsDaysBetween()	Expresses longer calendar intervals as years, months, and remaining days.
FT_DaysBetween()	Returns the exact signed number of elapsed days between two dates.
FT_DateToJulian()	Converts a Date value into a sequential day number for date arithmetic.
FT_FormatDateUS()	Formats a Date value using the U.S. month/day/year convention.
FT_FormatDateInternational()	Formats a Date value as YYYY-MM-DD for international use and chronological text sorting.

FT_YearsMonthsDaysBetween

Express the elapsed calendar time between two dates as complete years, months, and remaining days—with proper handling for leap years, unequal month lengths, and month-end dates.

Purpose

Calculates the forward elapsed calendar time between two WinDev **Date** values and returns the result as a naturally readable string containing complete years, months, and remaining days. For example: 4 years, 2 months, 11 days

When the ending date is omitted, **FT_YearsMonthsDaysBetween()** automatically uses the current system date. The procedure returns an empty string when either date is invalid or when the ending date occurs before the starting date.

Why This Procedure?

Long calendar intervals cannot be calculated accurately by dividing an elapsed-day total by 365 or 30. Years may contain 365 or 366 days, while months may contain 28, 29, 30, or 31 days. Simple arithmetic estimates can therefore produce misleading results, especially when an interval crosses leap years, February, or month-end boundaries.

FT_YearsMonthsDaysBetween() performs true calendar-based counting. It counts complete years first, then complete remaining months, and finally calculates any leftover days. The result reflects how people naturally describe elapsed calendar time rather than presenting an approximation based on average month or year lengths.

Key Features

- Calculates complete calendar years between two dates.
- Calculates complete remaining calendar months.
- Calculates any remaining elapsed days.
- Correctly accommodates leap years.
- Correctly handles months with different numbers of days.
- Preserves sensible month-end behavior.
- Defaults the ending date to the current system date when omitted.
- Produces grammatically correct singular and plural wording.
- Omits zero-value components for concise output.
- Returns 0 days when both dates are identical.
- Validates both Date values before processing.
- Returns an empty string for reversed date ranges.

Syntax

```
sElapsed is string = FT_YearsMonthsDaysBetween(dStartDate, dEndDate)
```

The ending date may be omitted:

```
sElapsed is string = FT_YearsMonthsDaysBetween(dStartDate)
```

Parameters

Parameter	Type	Description
dStartDate	Date	Beginning date of the elapsed period.
dEndDate	Date	Optional ending date. Defaults to the current system date when omitted.

Return Value

Type	Description
String	A readable elapsed-calendar string containing complete years, months, and remaining days. Returns an empty string if either date is invalid or the ending date precedes the starting date.

Remarks

FT_YearsMonthsDaysBetween() performs the calculation in three stages:

1. First, it advances from the starting date one complete calendar year at a time until another full year would exceed the ending date.
2. Next, it advances one complete calendar month at a time until another full month would exceed the ending date.
3. Finally, it uses **FT_DaysBetween()** to calculate the remaining elapsed days.

This sequence is important because it preserves recognizable calendar boundaries instead of estimating years and months from a raw day count. For example: 03/15/2020 through 05/26/2026 returns: 6 years, 2 months, 11 days

The function does not treat every year as exactly 365 days or every month as exactly 30 days.

Only non-zero components are included in the returned string. An exact five-year interval therefore returns: 5 years rather than: 5 years, 0 months, 0 days

Similarly, a result containing years and days but no complete remaining months may appear as: 2 years, 8 days If both dates are identical, the procedure returns: 0 days

The procedure calculates forward elapsed time only. When dEndDate occurs before dStartDate, an empty string is returned rather than a negative interval.

Real-World Uses

FT_YearsMonthsDaysBetween() is useful for:

- Calculating a person's age.
- Displaying employee length of service.
- Measuring customer or vendor relationships.
- Reporting equipment age.
- Calculating contract, lease, or warranty durations.
- Displaying the age of a software installation or license.
- Measuring time since account creation.
- Reporting project or case duration.
- Showing time elapsed since inspections, repairs, or maintenance.
- Producing readable anniversary and milestone information.

Practical Example
Display how long an employee has worked for an organization.

```
sServiceTime is string
sServiceTime = FT_YearsMonthsDaysBetween(Employee.HireDate)
IF sServiceTime <> "" THEN
   Info(Employee.FullName + ...
      " has worked here for " + ...
      sServiceTime + ".")
ELSE
   Error("The employee hire date is invalid.")
END
```

Possible output: "Maria Lopez has worked here for 8 years, 4 months, 12 days."

Because the ending date defaults to the current system date, the developer only needs to supply the employee's hire date.

A completed historical interval may be calculated by supplying both dates:

```
sServiceTime = FT_YearsMonthsDaysBetween( Employee.HireDate, Employee.TerminationDate)
```

Developer Tip
Use FT_YearsMonthsDaysBetween() when the result should be understandable as a human calendar interval. For example: 3 years, 2 months, 6 days is generally more meaningful in a user interface or report than: 1163 days

Use **FT_DaysBetween()** when an exact numeric day count is required for calculations, comparisons, deadlines, or business rules.

Use **FT_MonthsDaysBetween()** when years are not relevant and the result should remain focused on months and days.

Common Pitfalls
1) Do not calculate years by dividing elapsed days by 365. That approach ignores leap years and may produce incorrect year, month, and day components.
2) Do not calculate months by dividing the remaining days by 30. Calendar months vary in length, and the resulting value would only be an estimate.
3) The procedure calculates forward intervals only. If the ending date precedes the starting date, it returns an empty string. Verify date order when processing user-entered values.
4) The returned value is formatted display text. Retain the original Date values when additional calculations or comparisons are required.
5) A calendar year does not always contain the same number of elapsed days. An interval reported as 1 year may span either 365 or 366 days depending on the dates involved.

Related Procedures

Procedure	Why You Might Use It
FT_MonthsDaysBetween()	Expresses a shorter calendar interval as complete months and remaining days.
FT_DaysBetween()	Returns the exact signed number of elapsed days between two dates.
FT_DateToJulian()	Converts a Date value into a sequential day number for date calculations.
FT_MinutesBetweenDateTimes()	Calculates the signed total minutes between two DateTime values.
FT_FormatDateTimeElapsed()	Presents a DateTime interval as readable days, hours, and minutes.
FT_FormatDateUS()	Formats a Date value using the U.S. month/day/year convention.
FT_FormatDateInternational()	Formats a Date value as YYYY-MM-DD for international use and chronological text sorting.

Environment Function Reference

The Environment procedures retrieve information about the operating system and current user environment. These procedures provide a consistent interface for accessing commonly used Windows folders, user information, and system-specific values.

Understanding Windows Application Folders

One of the most common mistakes made by Windows developers is storing application files in the wrong location. Windows provides several special folders, each intended for a different type of information. Choosing the correct folder improves compatibility with Windows, simplifies backups, supports multiple user accounts, and follows Microsoft's recommended application design guidelines. FunctionTrak includes several procedures that retrieve these locations automatically, allowing your applications to remain portable across different Windows installations.

AppData (Roaming)
Typical location: C:\Users\<UserName>\AppData\Roaming

The **Roaming AppData** folder is intended for **user-specific information that should follow the user** between computers in a Windows domain environment.
Although roaming profiles are less common today, this folder remains the preferred location for information such as:

- Application preferences
- User settings
- Configuration files
- Dictionaries
- Templates
- Small user databases
- License information tied to a specific user

For example: C:\Users\John\AppData\Roaming\Software by Daughtry\FunctionTrak

If another person logs into the same computer, they receive their own independent AppData folder.

Use FT_GetEnvAppDataPath() when storing information that belongs to the current user rather than the computer.

Local AppData: Typical location: C:\Users\<UserName>\AppData\Local

The **Local AppData** folder is also user-specific, but unlike Roaming AppData, its contents remain on the current computer.
This is the preferred location for information such as:

- Cache files
- Downloaded content
- Search indexes
- Thumbnail databases
- Temporary working files
- Large local databases
- Performance caches

For example: C:\Users\John\AppData\Local\Software by Daughtry\FunctionTrak\Cache

Because these files can often be recreated automatically, there is usually no reason to copy them between computers.

Use FT_GetEnvLocalAppDataPath() for information that belongs to the current user but should remain on the local workstation.

ProgramData
Typical location: C:\ProgramData
Unlike the previous folders, **ProgramData** is shared by **every user account on the computer**.
This folder is intended for application data that should be accessible regardless of which user is logged in.

Typical examples include:

- Shared application databases
- Common configuration files or Machine-wide settings
- License information for all users
- Shared templates
- Common resources

For example: C:\ProgramData\Software by Daughtry\LicenseTrak

Unlike AppData, information stored in ProgramData is not duplicated for each Windows user.

Use FT_GetEnvProgramDataPath() when multiple users of the same computer need access to the same application data.

Which Folder Should I Use?

Store This...	**Recommended Folder**
User preferences	AppData (Roaming)
Configuration files	AppData (Roaming)
User-specific license information	AppData (Roaming)
Cache files	Local AppData
Downloaded content	Local AppData
Temporary working files	Local AppData
Shared application database	ProgramData
Machine-wide license information	ProgramData
Shared resources	ProgramData
Shared configuration	ProgramData

Best Practices
✓ Store **user settings** in **AppData (Roaming)**.
✓ Store **temporary or machine-specific data** in **Local AppData**.
✓ Store **shared application data** in **ProgramData**.
✓ Never assume these folders reside on the C: drive. Windows installations can use different drives or customized profile locations.
✓ Avoid hardcoding paths such as: C:\Users\John\AppData

Instead, retrieve these locations through the operating system using the appropriate FunctionTrak procedure.

Why Are There Two "Program Files" Folders?

On a 64-bit edition of Windows, the operating system supports both **64-bit** and **32-bit** applications.

To keep them separate, Windows uses two installation folders:

Folder	Intended Contents
Program Files	Native 64-bit applications
Program Files (x86)	32-bit applications running under the WOW64 compatibility layer

This separation helps Windows load the correct system libraries, simplifies software management, and allows both 32-bit and 64-bit versions of the same application to coexist on the same computer.

Most applications never need to worry about these details—Windows handles them automatically—but installers, migration utilities, diagnostic tools, and system utilities often need to know which location to examine.

FT_GetEnvAppDataPath

Retrieve the current user's Windows AppData folder with a clean, ready-to-use path—without requiring manual parsing of environment variables.

Purpose

Returns the path to the current user's Windows **AppData** folder.

The returned value contains only the folder path and does **not** include the "APPDATA=" prefix that may be returned by WinDev's SysEnvironment() function.

If the AppData environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?

Many Windows applications store user-specific configuration files, settings, templates, logs, and other application data within the user's **AppData** folder.
Although WinDev provides the SysEnvironment() function, developers must still verify that the environment variable exists, locate the equals sign separating the variable name from its value, and extract the usable path. **FT_GetEnvAppDataPath()** performs those tasks automatically, returning a clean folder path that can be used immediately by the application.

Key Features

- Retrieves the current user's Windows AppData folder.
- Removes the "APPDATA=" prefix automatically.
- Validates that the environment variable was successfully retrieved.
- Verifies that the returned value follows the expected NAME=VALUE format.
- Returns only the usable folder path.
- Returns an empty string if the environment variable is unavailable or improperly formatted.
- Eliminates repetitive parsing code throughout an application.

Syntax

sAppDataPath is string = FT_GetEnvAppDataPath()

Parameters

This procedure has no parameters.

Return Value

Type	Description
String	The current user's AppData folder path. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks

The procedure retrieves the Windows **APPDATA** environment variable using WinDev's SysEnvironment() function.

Some versions of WinDev return environment variables in the form:
APPDATA=C:\Users\Scott\AppData\Roaming

Rather than returning this complete string, **FT_GetEnvAppDataPath()** extracts everything following the first equals sign and returns only: C:\Users\Scott\AppData\Roaming

The returned path is immediately suitable for file and folder operations.

If the environment variable cannot be obtained or does not contain the expected NAME=VALUE format, the procedure returns an empty string rather than passing invalid data back to the caller.

This defensive behavior allows applications to detect unusual system configurations gracefully.

Real-World Uses
FT_GetEnvAppDataPath() is useful for:

- Storing application configuration files.
- Saving user preferences.
- Creating application log files.
- Maintaining user-specific databases.
- Writing cache or temporary application data.
- Storing license files or activation information.
- Creating application-specific folders beneath AppData.
- Locating per-user resources that should roam with a Windows profile.

Practical Example
Create an application folder beneath the user's AppData directory.

```
sAppDataPath is string
sApplicationFolder is string
sAppDataPath = FT_GetEnvAppDataPath()
IF sAppDataPath <> "" THEN
   sApplicationFolder = sAppDataPath + "\Software by Daughtry\FunctionTrak"
   fMakeDir(sApplicationFolder)
ELSE
   Error("Unable to locate the AppData folder.")
END
```

Possible resulting folder:
C:\Users\Scott\AppData\Roaming\Software by Daughtry\FunctionTrak

Developer Tip
TIP: AppData is the preferred location for storing **user-specific** configuration information, preferences, and data that should remain private to each Windows user account.

If your application stores data that should be shared by every user on the computer, consider using **FT_GetEnvProgramDataPath()** instead.

Common Pitfalls

1) Do not assume every Windows installation returns a valid AppData environment variable. Although uncommon, unusual system configurations or restricted execution environments may prevent the value from being available. Always verify that the returned string is not empty.

2) Avoid hardcoding paths such as: C:\Users\<UserName>\AppData\Roaming
Windows installations may reside on different drives, user profile locations can vary, and localized versions of Windows may differ. Retrieving the location through the operating system is significantly more reliable.

3) The returned value is the user's **Roaming AppData** folder, not the Local AppData folder. Use **FT_GetEnvLocalAppDataPath()** when data should remain on the local computer and should not roam with a user's Windows profile.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvLocalAppDataPath()	Retrieves the user's Local AppData folder for machine-specific data.
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder for application data used by all users.
FT_GetEnvUserProfilePath()	Retrieves the current user's Windows profile folder.
FT_GetEnvTempPath()	Retrieves the Windows temporary folder for temporary files.
FT_GetEnvHomeDrive()	Retrieves the user's home drive for constructing user-specific paths.

FT_GetEnvComputerName

Retrieve the Windows computer (host) name with built-in validation, returning a clean system identifier ready for logging, licensing, networking, and diagnostic purposes.

Purpose
Returns the Windows **computer name** (also known as the **host name**) of the system on which the application is running.

The returned value contains only the computer name and does **not** include the "COMPUTERNAME=" prefix that may be returned by WinDev's SysEnvironment() function.

If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?
Many applications need to identify the workstation on which they are executing. Computer names are commonly used in audit logs, diagnostic reports, licensing systems, inventory applications, and network administration tools.

Although WinDev provides access to environment variables through SysEnvironment(), developers must still verify that the value exists, confirm that it is properly formatted, and extract the usable computer name. **FT_GetEnvComputerName()** performs those steps automatically, returning a clean workstation name that is ready for immediate use.

Key Features
- Retrieves the Windows computer (host) name.
- Automatically removes the "COMPUTERNAME=" prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the workstation name.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax
sComputerName is string = FT_GetEnvComputerName()

Parameters
This procedure has no parameters.

Return Value

Type	Description
String	The Windows computer (host) name. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks
The procedure retrieves the Windows **COMPUTERNAME** environment variable using WinDev's SysEnvironment() function.

Some versions of WinDev return environment variables in the form:
COMPUTERNAME=OFFICE-PC

Rather than returning the complete string, **FT_GetEnvComputerName()** extracts everything following the first equals sign and returns only: OFFICE-PC

This value is immediately suitable for display, logging, licensing, or storage within an application.
If the environment variable is unavailable or does not follow the expected NAME=VALUE format, the procedure returns an empty string rather than returning potentially invalid data.

Real-World Uses
FT_GetEnvComputerName() is useful for:

- Recording workstation names in application logs.
- Displaying diagnostic information.
- Identifying systems during troubleshooting.
- Generating workstation-specific reports.
- Supporting licensing systems that identify individual computers.
- Auditing user activity across multiple workstations.
- Recording the origin of imported or exported data.
- Displaying system information within an About or Diagnostics window.

Practical Example
Record the workstation name when a user logs into the application.
Trace("User logged in from workstation: " + FT_GetEnvComputerName())

Possible output:
User logged in from workstation: SALES-PC-07
Or display the computer name in an About dialog:
EDT_ComputerName = FT_GetEnvComputerName()

Developer Tip
1) Computer names are excellent identifiers for logging and diagnostics, but they should not be relied upon as unique security identifiers. Workstations can be renamed by administrators, replaced, or reimaged. If an application requires stronger identification, consider combining the computer name with additional information such as the current user name or other licensing identifiers.

2) Do not confuse the **computer name** with the current **Windows user name**. Multiple users may log into the same computer, and the same user may log into different computers. Use **FT_GetEnvUserName()** when the identity of the logged-in user is required.

3) Do not assume every execution environment provides a valid computer name. While uncommon, restricted or specialized environments may not expose the expected environment variable. Always verify that the returned string is not empty.

4) Avoid repeatedly calling SysEnvironment("COMPUTERNAME") and parsing its results throughout an application. Using FT_GetEnvComputerName() centralizes the validation and parsing logic, making future maintenance easier and ensuring consistent behavior.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvUserName()	Retrieves the Windows user account currently logged into the system.
FT_GetEnvUserProfilePath()	Retrieves the current user's Windows profile folder.
FT_GetEnvHomeDrive()	Retrieves the user's home drive.
FT_GetEnvAppDataPath()	Retrieves the current user's AppData folder.
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder used by all users.

FT_GetEnvCommonProgramFilesPath

Retrieve the Windows Common Program Files folder with built-in validation, returning a clean path for accessing shared program components used by multiple applications.

Purpose

Returns the path to the Windows **Common Program Files** folder.

The returned value contains only the folder path and does **not** include the **"COMMONPROGRAMFILES="** prefix that may be returned by WinDev's **SysEnvironment()** function.

If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?

Windows provides a dedicated **Common Program Files** folder for components that are shared among multiple applications, such as common libraries, plug-ins, COM objects, shared resources, and vendor-specific support files.

Although WinDev's **SysEnvironment()** function provides access to this environment variable, developers must still validate the returned value, locate the separator between the variable name and its value, and extract the usable folder path.

FT_GetEnvCommonProgramFilesPath() performs those tasks automatically, providing a clean folder path that is ready for immediate use.

Key Features

- Retrieves the Windows Common Program Files folder.
- Automatically removes the **"COMMONPROGRAMFILES="** prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected **NAME=VALUE** format.
- Returns only the usable folder path.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax

sCommonProgramFiles is string = FT_GetEnvCommonProgramFilesPath()

Parameters

This procedure has no parameters

Return Value

Type	Description
String	The Windows Common Program Files folder path. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks

The procedure retrieves the Windows **COMMONPROGRAMFILES** environment variable using WinDev's **SysEnvironment()** function.

Some versions of WinDev return environment variables in the form:

COMMONPROGRAMFILES=C:\Program Files\Common Files

Rather than returning the complete string, **FT_GetEnvCommonProgramFilesPath()** extracts everything following the first equals sign and returns only:

C:\Program Files\Common Files

This folder is intended for components shared by multiple applications installed on the computer.

If the environment variable is unavailable or is not returned in the expected **NAME=VALUE** format, the procedure returns an empty string, allowing the application to detect and handle the condition safely.

Real-World Uses

FT_GetEnvCommonProgramFilesPath() is useful for:

- Locating shared application libraries.
- Accessing vendor-supplied runtime components.
- Installing or locating common plug-ins.
- Reading shared configuration files.
- Accessing COM or ActiveX support files.
- Building installer paths.
- Displaying system configuration information.
- Diagnosing installation or deployment problems.

Practical Example

Construct the path to a shared application folder.

```
sCommonFiles is string
sVendorFolder is string
sCommonFiles = FT_GetEnvCommonProgramFilesPath()

IF sCommonFiles <> "" THEN
   sVendorFolder = sCommonFiles + "\Software by Daughtry"
   Info(sVendorFolder)
ELSE
   Error("Unable to locate the Common Program Files folder.")
END
```

Possible output:

```
C:\Program Files\Common Files\Software by Daughtry
```

Developer Tip

TIP: The **Common Program Files** folder is intended for components that may be shared by multiple applications. Application-specific data files, user settings, and configuration information generally belong in locations such as **AppData** or **ProgramData**, depending on whether the information is user-specific or shared across the computer.

Common Pitfalls

1) Do not confuse **Program Files** with **Common Program Files**. Program Files typically contains individual application installations, while Common Program Files is reserved for components intended to be shared among multiple applications.

2) Avoid hardcoding paths such as: **C:\Program Files\Common Files**

Windows may be installed on a different drive, and 32-bit and 64-bit environments can differ. Retrieving the location through the operating system is more reliable and portable.

3) Always verify that the returned value is not empty before using it. Although rare, the environment variable may be unavailable or malformed in restricted or non-standard execution environments.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder used for application data accessible to all users.
FT_GetEnvAppDataPath()	Retrieves the current user's AppData folder for user-specific configuration and data.
FT_GetEnvLocalAppDataPath()	Retrieves the Local AppData folder for machine-specific user data.
FT_GetEnvUserProfilePath()	Retrieves the current user's Windows profile folder.
FT_GetEnvTempPath()	Retrieves the Windows temporary folder for temporary files.

FT_GetEnvLocalAppDataPath

Retrieve the current user's Windows Local AppData folder with built-in validation, returning a clean path for machine-specific application data that does not roam with the user's profile.

Purpose
Returns the path to the current user's Windows **Local AppData** folder.

The returned value contains only the folder path and does **not** include the "LOCALAPPDATA=" prefix that may be returned by WinDev's SysEnvironment() function.

If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?
Windows provides two primary locations for storing user-specific application data:

- **AppData (Roaming)** for data that may follow the user between computers within a Windows domain.
- **Local AppData** for data that remains on the current computer.

Many applications store caches, temporary databases, downloaded content, indexes, thumbnails, and other machine-specific resources within the **Local AppData** folder.

Although WinDev's SysEnvironment() function retrieves the environment variable, developers must still validate the returned value, locate the separator between the variable name and its value, and extract the usable folder path.

FT_GetEnvLocalAppDataPath() performs those steps automatically, returning a clean folder path ready for immediate use.

Key Features

- Retrieves the current user's Windows Local AppData folder.
- Automatically removes the "LOCALAPPDATA=" prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the usable folder path.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax
sLocalAppDataPath is string = FT_GetEnvLocalAppDataPath()

Parameters
This procedure has no parameters.

Return Value

Type	Description
String	The current user's Local AppData folder path. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks

The procedure retrieves the Windows **LOCALAPPDATA** environment variable using WinDev's SysEnvironment() function. Some versions of WinDev return environment variables in the form:

LOCALAPPDATA=C:\Users\Scott\AppData\Local

Rather than returning the complete string, **FT_GetEnvLocalAppDataPath()** extracts everything following the first equals sign and returns only: C:\Users\Scott\AppData\Local

Unlike the **Roaming AppData** folder, the Local AppData folder is intended for information that remains on the current computer. Typical contents include caches, temporary working files, search indexes, downloaded resources, and other data that does not need to accompany a user's Windows profile.

If the environment variable cannot be retrieved or does not follow the expected NAME=VALUE format, the procedure returns an empty string, allowing the calling application to handle the condition safely.

Real-World Uses

FT_GetEnvLocalAppDataPath() is useful for:

- Storing application cache files.
- Maintaining local databases.
- Saving downloaded content.
- Creating thumbnail or preview caches.
- Storing search indexes.
- Saving temporary working files.
- Maintaining machine-specific configuration data.
- Creating application folders that should remain on the local computer.

Practical Example

Create a cache folder beneath the Local AppData directory.

```
sLocalAppData is string
sCacheFolder is string
sLocalAppData = FT_GetEnvLocalAppDataPath()
IF sLocalAppData <> "" THEN
   sCacheFolder = sLocalAppData + "\Software by Daughtry\FunctionTrak\Cache"
   fMakeDir(sCacheFolder)
ELSE
   Error("Unable to locate the Local AppData folder.")
END
```

Possible resulting folder:

C:\Users\Scott\AppData\Local\Software by Daughtry\FunctionTrak\Cache

Developer Tip
Use **Local AppData** for files that are specific to the current computer, such as caches, indexes, downloaded content, and temporary databases.

Use **FT_GetEnvAppDataPath()** instead when storing user preferences or configuration files that may benefit from roaming with the user's Windows profile in a managed network environment.

Common Pitfalls
1) Do not confuse **Local AppData** with **Roaming AppData**. Although both are user-specific folders, they serve different purposes. Local AppData is intended for data that remains on the current computer, while Roaming AppData is designed for information that can accompany the user between domain-connected systems.

2) Avoid hardcoding paths such as: C:\Users\<UserName>\AppData\Local

Windows installations may reside on different drives, user profile locations can vary, and localized versions of Windows may organize profiles differently. Retrieving the location through the operating system is significantly more reliable.

3) Always verify that the returned value is not empty before using it. Although uncommon, restricted or non-standard execution environments may not expose the expected environment variable.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvAppDataPath()	Retrieves the Roaming AppData folder for user settings and configuration that may follow the user between computers.
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder used by all users on the computer.
FT_GetEnvTempPath()	Retrieves the Windows temporary folder for temporary files.
FT_GetEnvUserProfilePath()	Retrieves the current user's Windows profile folder.
FT_GetEnvCommonProgramFilesPath()	Retrieves the Windows Common Program Files folder used by shared application components.

FT_GetEnvLogonServer

Retrieve the Windows logon server with built-in validation, returning the authentication server responsible for the current Windows login session.

Purpose
Returns the name of the Windows **logon server** that authenticated the current user's Windows session.

The returned value contains only the server name and does **not** include the "LOGONSERVER=" prefix that may be returned by WinDev's SysEnvironment() function.

If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?
In business and enterprise environments, users commonly authenticate against a Windows domain controller. Knowing which server authenticated the current session can be valuable when troubleshooting authentication issues, diagnosing network problems, generating audit logs, or collecting system information.

Although WinDev's SysEnvironment() function provides access to this environment variable, developers must still verify that the value exists, confirm that it follows the expected format, and extract the usable server name.

FT_GetEnvLogonServer() performs those steps automatically, returning a clean server name ready for immediate use.

Key Features
- Retrieves the Windows logon server.
- Automatically removes the "LOGONSERVER=" prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the usable server name.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax
sLogonServer is string = FT_GetEnvLogonServer()

Parameters
This procedure has no parameters.

Return Value

Type	Description
String	The Windows logon server. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks
The procedure retrieves the Windows **LOGONSERVER** environment variable using WinDev's SysEnvironment() function.

Some versions of WinDev return environment variables in the form: LOGONSERVER=\\DC01

Rather than returning the complete string, **FT_GetEnvLogonServer()** extracts everything following the first equals sign and returns only: \\DC01

In Active Directory environments, this value typically identifies the **domain controller** that authenticated the user's Windows logon session.

On standalone computers or home systems that are not members of a Windows domain, the returned value may instead reference the local computer or may not be available at all, depending on the Windows configuration.

If the environment variable is unavailable or does not follow the expected NAME=VALUE format, the procedure returns an empty string, allowing the application to detect and handle the condition safely.

Real-World Uses
FT_GetEnvLogonServer() is useful for:

- Recording authentication server information in audit logs.
- Troubleshooting Windows domain login issues.
- Displaying diagnostic system information.
- Identifying which domain controller authenticated a user.
- Supporting enterprise help desk investigations.
- Recording workstation authentication details.
- Collecting environment information for support reports.
- Assisting network administrators during troubleshooting.

Practical Example
Display diagnostic information when a user submits a technical support request.

```
sLogonServer is string
sLogonServer = FT_GetEnvLogonServer()
IF sLogonServer <> "" THEN
   Trace("Authenticated by: " + sLogonServer)
ELSE
   Trace("Logon server unavailable.")
END
```

Possible output: Authenticated by: \\DC01

Developer Tip
This function is particularly valuable in corporate environments that use **Microsoft Active Directory**. Including the logon server in diagnostic reports can help administrators determine which domain controller authenticated a user's session, making it easier to troubleshoot replication or authentication problems.
For consumer applications intended primarily for home users, this information is often less useful because many personal computers are not members of a Windows domain.

Common Pitfalls
1) Do not assume the returned value identifies the user's file server, mail server, or application server. The logon server specifically identifies the system that authenticated the Windows login session.

2) The returned value commonly begins with two backslashes (\\). This is normal—it represents the server's UNC network name.

3) Do not assume every Windows installation provides a meaningful logon server. Standalone computers, workgroup environments, kiosk systems, and some virtual environments may return a different value or none at all. Always verify that the returned string is not empty.

3) The logon server may change between Windows sessions. If a user logs on at a different time or against a different domain controller, the returned value may differ even though the computer and user account remain the same.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvUserName()	Retrieves the currently logged-in Windows user.
FT_GetEnvComputerName()	Retrieves the Windows computer (host) name.
FT_GetEnvUserProfilePath()	Retrieves the current user's profile folder.
FT_GetEnvHomeDrive()	Retrieves the user's home drive.
FT_GetEnvAppDataPath()	Retrieves the current user's AppData folder.

FT_GetEnvProgramFilesPath

Retrieve the Windows Program Files folder with built-in validation, returning a clean installation path for locating application executables and program resources.

Purpose

Returns the path to the Windows **Program Files** folder.
The returned value contains only the folder path and does **not** include the "PROGRAMFILES=" prefix that may be returned by WinDev's SysEnvironment() function.
If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?

The **Program Files** folder is the standard Windows location for installing desktop applications. Many installers, utilities, and maintenance tools need to locate this directory when installing software, locating executables, or accessing shared application resources.

Although WinDev's SysEnvironment() function provides access to the **PROGRAMFILES** environment variable, developers must still validate the returned value, verify that it follows the expected format, and extract the usable folder path.

FT_GetEnvProgramFilesPath() performs those steps automatically, returning a clean installation path that is immediately ready for use.

Key Features

- Retrieves the Windows Program Files folder.
- Automatically removes the "PROGRAMFILES=" prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the usable folder path.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax

sProgramFilesPath is string = FT_GetEnvProgramFilesPath()

Parameters

This procedure has no parameters.

Return Value

Type	Description
String	The Windows Program Files folder path. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks
The procedure retrieves the Windows **PROGRAMFILES** environment variable using WinDev's SysEnvironment() function.

Some versions of WinDev return environment variables in the form: PROGRAMFILES=C:\Program Files

Rather than returning the complete string, **FT_GetEnvProgramFilesPath()** extracts everything following the first equals sign and returns only: C:\Program Files

This folder is the default installation location for most 64-bit Windows applications.

If the environment variable cannot be retrieved or is not returned in the expected NAME=VALUE format, the procedure returns an empty string, allowing the application to detect and handle the condition safely.

It is important to remember that the actual location of the Program Files folder is determined by Windows. While it commonly resides on the C: drive, Windows can be installed on another drive, making it unsafe to assume a fixed path.

Real-World Uses
FT_GetEnvProgramFilesPath() is useful for:

- Locating installed applications.
- Building installer and uninstaller paths.
- Finding executable files.
- Accessing application resource folders.
- Searching for installed software.
- Displaying system configuration information.
- Diagnosing installation issues.
- Creating maintenance and update utilities.

Practical Example
Locate an application's installation folder.

```
sProgramFiles is string
sApplicationFolder is string
sProgramFiles = FT_GetEnvProgramFilesPath()
IF sProgramFiles <> "" THEN
   sApplicationFolder = sProgramFiles + "\Software by Daughtry\FunctionTrak"
   Info(sApplicationFolder)
ELSE
   Error("Unable to locate the Program Files folder.")
END
```

Possible output: C:\Program Files\Software by Daughtry\FunctionTrak

Developer Tip
Use **Program Files** for **installed application binaries and program resources**.

Do **not** store user settings, databases, log files, or other writable data within the Program Files folder. Modern versions of Windows protect this directory with User Account Control (UAC), and applications running under normal user privileges may not have permission to modify its contents.
Instead:

- Store **user-specific data** in **AppData**.
- Store **machine-wide shared data** in **ProgramData**.
- Store **temporary files** in the **Temp** folder.

Following these Windows conventions results in applications that are more secure, more portable, and compatible with modern versions of Windows.

Common Pitfalls
1) Do not confuse **Program Files** with **ProgramData**.

- **Program Files** contains installed applications.
- **ProgramData** contains shared application data.

These folders serve entirely different purposes.

2) Avoid hardcoding: C:\Program Files

Windows may be installed on another drive, and localized or customized installations may place the folder elsewhere. Always retrieve the location from Windows.

3) Do not write configuration files, databases, or log files into the Program Files folder. Since Windows Vista, this location is protected by UAC, and write operations may fail unless the application is running with elevated privileges.

4) On 64-bit Windows, both **Program Files** and **Program Files (x86)** may exist. The **PROGRAMFILES** environment variable typically references the native Program Files directory for the executing process. Applications that need to distinguish between 32-bit and 64-bit installation folders should account for Windows architecture when determining installation locations.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvCommonProgramFilesPath()	Retrieves the Common Program Files folder used for components shared by multiple applications.
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder used for writable application data accessible to all users.
FT_GetEnvAppDataPath()	Retrieves the current user's Roaming AppData folder for user-specific settings and configuration.
FT_GetEnvLocalAppDataPath()	Retrieves the Local AppData folder for caches and machine-specific user data.

FT_GetEnvProgramFilesX86Path

Retrieve the Windows Program Files (x86) folder with built-in validation, returning a clean installation path for 32-bit applications on 64-bit Windows systems.

Purpose
Returns the path to the Windows **Program Files (x86)** folder.
The returned value contains only the folder path and does **not** include the "PROGRAMFILES(X86)=" prefix that may be returned by WinDev's SysEnvironment() function.
If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?
On 64-bit editions of Windows, Microsoft separates installed applications into two primary locations:

- **Program Files** for 64-bit applications.
- **Program Files (x86)** for 32-bit applications.

Many installation utilities, migration tools, maintenance programs, and diagnostic applications need to locate the 32-bit installation directory.

Although WinDev's SysEnvironment() function provides access to the **PROGRAMFILES(X86)** environment variable, developers must still validate the returned value, verify that it follows the expected format, and extract the usable folder path.

FT_GetEnvProgramFilesX86Path() performs those tasks automatically, returning a clean installation path ready for immediate use.

Key Features

- Retrieves the Windows Program Files (x86) folder.
- Automatically removes the "PROGRAMFILES(X86)=" prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the usable folder path.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax
sProgramFilesX86 is string = FT_GetEnvProgramFilesX86Path()

Parameters
This procedure has no parameters.

Return Value

Type	Description
String	The Windows Program Files (x86) folder path. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks
The procedure retrieves the Windows **PROGRAMFILES(X86)** environment variable using WinDev's SysEnvironment() function. Some versions of WinDev return environment variables in the form: PROGRAMFILES(X86)=C:\Program Files (x86)

Rather than returning the complete string, **FT_GetEnvProgramFilesX86Path()** extracts everything following the first equals sign and returns only: C:\Program Files (x86)

This folder exists primarily on **64-bit editions of Windows** and serves as the default installation location for 32-bit applications.

On 32-bit versions of Windows, the **PROGRAMFILES(X86)** environment variable typically does not exist. In that case, the procedure returns an empty string, allowing the application to determine that a separate 32-bit installation directory is unavailable.

If the environment variable cannot be retrieved or does not follow the expected NAME=VALUE format, the procedure also returns an empty string.

Real-World Uses
FT_GetEnvProgramFilesX86Path() is useful for:

- Locating installed 32-bit applications.
- Building installer and uninstaller paths.
- Supporting migration or upgrade utilities.
- Searching for legacy software installations.
- Accessing 32-bit application resources.
- Diagnosing installation problems.
- Displaying system configuration information.
- Detecting Windows architecture during installation or maintenance.

Practical Example
Locate a legacy 32-bit application.

```
sProgramFilesX86 is string
sLegacyApplication is string
sProgramFilesX86 = FT_GetEnvProgramFilesX86Path()
IF sProgramFilesX86 <> "" THEN
   sLegacyApplication = sProgramFilesX86 + "\Legacy Software\Application"
   Info(sLegacyApplication)
ELSE
   Info("Program Files (x86) is not available.")
END
```

Possible output: C:\Program Files (x86)\Legacy Software\Application

Developer Tip
The presence of a valid **Program Files (x86)** folder generally indicates that the application is running on a **64-bit edition of Windows**.

If this procedure returns an empty string, the system is often a 32-bit Windows installation (or another environment where the variable is unavailable).

This can be useful when writing installers or utilities that need to adapt their behavior based on the operating system architecture.

Common Pitfalls

1) Do not assume the **Program Files (x86)** folder exists on every computer. It is normally present only on **64-bit Windows**.

2) Do not confuse **Program Files** and **Program Files (x86)**.

- **Program Files** typically contains native 64-bit applications.
- **Program Files (x86)** typically contains 32-bit applications.

Choosing the wrong location may prevent your application from locating the software it expects.

3) Avoid hardcoding: C:\Program Files (x86)

Windows may be installed on another drive, and customized installations may place the folder elsewhere. Always retrieve the location from Windows rather than assuming a fixed path.

4) As with the standard Program Files directory, applications should not store writable user data, configuration files, or databases in Program Files (x86). Those belong in AppData or ProgramData, depending on whether the data is user-specific or shared.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvProgramFilesPath()	Retrieves the standard Program Files folder used primarily for native 64-bit applications.
FT_GetEnvCommonProgramFilesPath()	Retrieves the Common Program Files folder used for components shared by multiple applications.
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder used for writable application data.
FT_GetEnvAppDataPath()	Retrieves the current user's Roaming AppData folder.
FT_GetEnvLocalAppDataPath()	Retrieves the Local AppData folder for machine-specific user data.

FT_GetEnvTempPath

Retrieve the Windows temporary folder with built-in validation and automatic fallback, returning a clean path for storing temporary working files.

Purpose
Returns the path to the Windows **temporary folder**.

The procedure first attempts to retrieve the **TEMP** environment variable. If **TEMP** is unavailable, it automatically falls back to the **TMP** environment variable.

The returned value contains only the folder path and does **not** include the "TEMP=" or "TMP=" prefix that may be returned by WinDev's SysEnvironment() function.

If neither environment variable can be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?
Virtually every Windows application creates temporary files at some point—whether for report generation, file conversion, data import/export, document previews, or intermediate processing.

Windows traditionally supports **two** environment variables that may identify the temporary folder:

- **TEMP**
- **TMP**

Although both commonly point to the same location, this is not guaranteed. Some systems define only one of the two variables.

Rather than requiring developers to check both variables themselves, **FT_GetEnvTempPath()** automatically attempts **TEMP** first, then **TMP**, validates the returned value, removes the variable name, and returns a clean folder path.

The result is a more reliable solution that works across a wider range of Windows configurations.

Key Features

- Retrieves the Windows temporary folder.
- Attempts **TEMP** first.
- Automatically falls back to **TMP** if necessary.
- Removes the "TEMP=" or "TMP=" prefix automatically.
- Validates that an environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the usable folder path.
- Returns an empty string if neither variable is available.
- Eliminates repetitive parsing and fallback logic throughout an application.

Syntax
sTempPath is string = FT_GetEnvTempPath()

Parameters
This procedure has no parameters.

Return Value

Type	Description
String	The Windows temporary folder. Returns an empty string if neither TEMP nor TMP is available or cannot be parsed successfully.

Remarks
The procedure first retrieves the Windows **TEMP** environment variable using WinDev's SysEnvironment() function.

If no value is returned, it automatically attempts to retrieve **TMP**.

Some versions of WinDev return environment variables in the form:
TEMP=C:\Users\Scott\AppData\Local\Temp
or
TMP=C:\Users\Scott\AppData\Local\Temp

Rather than returning the complete string, **FT_GetEnvTempPath()** extracts everything following the first equals sign and returns only: C:\Users\Scott\AppData\Local\Temp

Although **TEMP** and **TMP** usually reference the same folder, Windows does not require this. FunctionTrak's fallback logic ensures the procedure remains functional even if only one of the two variables has been defined. The temporary folder is intended for files that exist only while an application is running or until they are no longer needed. Applications should never rely on temporary files remaining available indefinitely, as Windows, disk cleanup utilities, or users may delete them at any time.

Real-World Uses
FT_GetEnvTempPath() is useful for:

- Creating temporary reports.
- Extracting archive files.
- Importing or exporting data.
- Generating PDF or document previews.
- Performing file conversions.
- Creating temporary databases.
- Storing intermediate processing files.
- Saving temporary downloads prior to installation.

Practical Example
Create a temporary working file.

```
sTempFolder is string
sWorkFile is string
sTempFolder = FT_GetEnvTempPath()
IF sTempFolder <> "" THEN
   sWorkFile = sTempFolder + "\ImportData.tmp"
   // Process the temporary file...
ELSE
   Error("Unable to locate the Windows temporary folder.")
END
```

Possible resulting path: C:\Users\Scott\AppData\Local\Temp\ImportData.tmp

Developer Tip
The Windows temporary folder is intended for **short-lived working files**. Applications should delete temporary files when they are no longer needed. While Windows periodically cleans temporary folders automatically, leaving unnecessary files behind wastes disk space and can eventually affect system performance. A good practice is to create uniquely named temporary files and remove them immediately after processing is complete.

Common Pitfalls
1) Do not assume **TEMP** always exists. Some Windows configurations define only **TMP**, which is why this procedure automatically checks both variables.

2) Never store permanent application data in the temporary folder. Windows, Disk Cleanup, Storage Sense, third-party cleanup utilities, or users may remove temporary files without warning.

3) Avoid creating predictable temporary filenames such as: Import.tmp

Multiple copies of your application—or even different applications—could attempt to use the same file simultaneously. Whenever practical, generate unique filenames to avoid collisions.

4) Always verify that the returned value is not empty before creating files. Although uncommon, highly restricted or embedded Windows environments may not expose either environment variable.

Related Procedures

Procedure	Why You Might Use It
FT_GetEnvLocalAppDataPath()	Retrieves the Local AppData folder for caches and longer-lived machine-specific user data.
FT_GetEnvAppDataPath()	Retrieves the Roaming AppData folder for user preferences and configuration.
FT_GetEnvProgramDataPath()	Retrieves the shared ProgramData folder for machine-wide application data.
FT_GetEnvUserProfilePath()	Retrieves the current user's Windows profile folder.
FT_GetEnvProgramFilesPath()	Retrieves the Program Files installation directory.

FT_GetEnvUserDomain

Retrieve the Windows user domain with built-in validation, returning the Windows domain or local computer name associated with the current user account.

Purpose
Returns the Windows **user domain** associated with the currently logged-in user.
The returned value contains only the domain name and does **not** include the "USERDOMAIN=" prefix that may be returned by WinDev's SysEnvironment() function.
If the environment variable cannot be retrieved or is returned in an unexpected format, the procedure returns an empty string.

Why This Procedure?
In Windows, every user account belongs to a **security domain**.

In a corporate environment, this is typically the organization's **Active Directory domain**. On a standalone or home computer, the user domain is usually the local computer name.

Knowing the user's domain can be useful for authentication, auditing, diagnostics, licensing, and technical support applications.

Although WinDev's SysEnvironment() function provides access to the **USERDOMAIN** environment variable, developers must still verify that the value exists, confirm it follows the expected format, and extract the usable domain name.

FT_GetEnvUserDomain() performs those steps automatically, returning a clean domain name ready for immediate use.

Key Features
- Retrieves the Windows user domain.
- Automatically removes the "USERDOMAIN=" prefix.
- Validates that the environment variable exists.
- Confirms the returned value follows the expected NAME=VALUE format.
- Returns only the usable domain name.
- Returns an empty string if the value cannot be retrieved or parsed.
- Eliminates repetitive parsing code throughout an application.

Syntax
sUserDomain is string = FT_GetEnvUserDomain()

Parameters
This procedure has no parameters.

Return Value

Type	Description
String	The Windows user domain. Returns an empty string if the environment variable is unavailable or cannot be parsed successfully.

Remarks
The procedure retrieves the Windows **USERDOMAIN** environment variable using WinDev's SysEnvironment() function.

Some versions of WinDev return environment variables in the form: USERDOMAIN=CONTOSO

Rather than returning the complete string, **FT_GetEnvUserDomain()** extracts everything following the first equals sign and returns only: CONTOSO

On computers that are members of a Windows domain, this value identifies the **Active Directory domain** that owns the user's account. On standalone computers or home systems that are not joined to a domain, Windows typically returns the **local computer name** instead.

For this reason, developers should not assume that the returned value always represents a network domain—it may simply identify the local workstation.

If the environment variable cannot be retrieved or does not follow the expected NAME=VALUE format, the procedure returns an empty string.

Real-World Uses
FT_GetEnvUserDomain() is useful for:

- Building domain\username login names.
- Recording audit information.
- Logging authentication details.
- Creating diagnostic reports.

Practical Example
Display the fully qualified Windows account name.

```
sDomain is string
sUser is string
sDomain = FT_GetEnvUserDomain()
sUser = FT_GetEnvUserName()
IF sDomain <> "" AND sUser <> "" THEN
   Info(sDomain + "\" + sUser)
END
```

Possible output on a domain-joined computer: CONTOSO\JDoe

Possible output on a standalone computer: OFFICE-PC\Scott

Developer Tip
This function works especially well when paired with **FT_GetEnvUserName()**. Together they allow you to construct the familiar Windows account format: DOMAIN\UserName

This format is widely used in enterprise software, security logs, audit records, and technical support utilities.

Common Pitfalls
1) Do not assume the returned value always represents an Active Directory domain. On standalone computers, Windows commonly returns the **computer name** instead.

2) Do not confuse **USERDOMAIN** with **LOGONSERVER**.
- **USERDOMAIN** identifies the domain (or local computer) that owns the user account.
- **LOGONSERVER** identifies the server that authenticated the current Windows logon session.

Although related, they represent different pieces of information.

3) The user domain should not be used as a unique identifier. Multiple users may belong to the same domain, and computer names can be changed over time.

4) Always verify that the returned value is not empty before using it in application logic or diagnostic output.

Developers—even experienced Windows developers—mix up these three concepts:

Environment Variable	What It Identifies
USERDOMAIN	The domain (or local computer) that owns the user account.
LOGONSERVER	The server that authenticated the Windows logon.
COMPUTERNAME	The computer the application is currently running on.

Those three values are often—but not always—the same. For example, on a home PC:
USERDOMAIN = OFFICE-PC
COMPUTERNAME = OFFICE-PC
LOGONSERVER = \\OFFICE-PC

On a corporate network:
USERDOMAIN = CONTOSO
COMPUTERNAME = SALES-LT-217
LOGONSERVER = \\DC03

Related Procedures

Procedure	**Why You Might Use It**
FT_GetEnvUserName()	Retrieves the currently logged-in Windows user name.
FT_GetEnvLogonServer()	Retrieves the server that authenticated the current Windows session.
FT_GetEnvComputerName()	Retrieves the local Windows computer (host) name.
FT_GetEnvUserProfilePath()	Retrieves the current user's Windows profile folder.
FT_GetEnvHomeDrive()	Retrieves the user's home drive.

File/Path Function Reference

The File/Path procedures simplify working with filenames, folders, and file system paths. They provide reliable methods for validating names, generating unique filenames, manipulating paths, and preparing file-related data for production applications.

FT_AppendLineToFile

Append a line of text to a file, automatically creating the file if it does not already exist.

Purpose

Appends a single line of text to the end of a text file.

If the specified file does not exist, the procedure automatically creates it before writing the supplied text. The procedure performs complete error checking throughout the operation and returns **True** only if the entire process succeeds.

Why This Procedure?

Appending information to a text file is a common programming task used for logging, audit trails, diagnostics, exporting data, and maintaining simple text databases.

Although WinDev provides the necessary low-level file functions, developers typically need to:

- Determine whether the file already exists.
- Create the file if necessary.
- Open an existing file in append mode.
- Write the new line.
- Close the file.
- Verify that every operation completed successfully.

FT_AppendLineToFile() encapsulates all of these steps into a single, easy-to-use procedure, reducing repetitive code while improving reliability.

Key Features

- Appends a line to an existing text file.
- Automatically creates the file if it does not exist.
- Performs complete error checking.
- Verifies successful file creation.
- Verifies successful file opening.
- Verifies successful write operations.
- Verifies that the file closes successfully.
- Returns a simple True/False success indicator.
- Eliminates repetitive file handling code.

Syntax

bSuccess is boolean = FT_AppendLineToFile(sFileName, sText)

Parameters

Parameter	Description
sFileName	The text file to append to. If the file does not exist, it is created automatically.
sText	The text to append as a new line.

Return Value

Type	Description
Boolean	**True** if the line was successfully written and the file was closed successfully; otherwise **False**.

Remarks

The procedure first validates that a filename has been supplied. If the file already exists, it is opened using append mode so that the new text is written after the existing contents. If the file does not exist, a new text file is created automatically. The supplied text is written using WinDev's fWriteLine() function, ensuring that each call appends a complete line terminated with the appropriate end-of-line characters. Finally, the procedure closes the file and verifies that the close operation completed successfully.

Only when **every** operation succeeds does the procedure return **True**.

Real-World Uses

FT_AppendLineToFile() is useful for:

- Application log files.
- Error logging.
- Diagnostic tracing.
- Audit trails.
- Exporting reports.
- Recording program activity.
- Tracking application events.
- Building simple text-based data files.

Practical Example

Append an entry to an application log.

```
IF NOT FT_AppendLineToFile("Application.log", DateSys() + " " + TimeSys() + " - Application started") THEN
   Error("Unable to update the log file.")
END
```

Possible resulting lines injected into the Application.Log file:

```
2026-07-29 08:15:01 - Application started
2026-07-29 08:15:43 - User logged in
2026-07-29 08:16:11 - Customer record saved
```

Developer Tip

This procedure is ideal for maintaining log files because it automatically creates the file during the first write. There is no need to perform a separate "create if missing" step before logging begins. Many developers call this procedure from a dedicated logging routine, allowing every significant event in an application to be recorded with a single function call.

Common Pitfalls

1) This procedure appends **one line** of text. If the supplied string already contains embedded carriage returns or line feeds, multiple physical lines may be written to the file.

2) The procedure does not create missing folders. If the specified directory does not exist, the file cannot be created and the procedure returns **False**. Create the folder first if necessary.

3) Always check the Boolean return value. A failed write may result from insufficient permissions, a locked file, a full disk, or an invalid path.

4) Do not assume that a successful write guarantees the file was successfully closed. This procedure explicitly verifies the close operation before reporting success.

Related Procedures

Procedure	Why You Might Use It
FT_FileToString()	Reads the contents of an entire text file into a string.
FT_StringToFile()	Creates or overwrites a text file from a string.
FT_IsValidFileName()	Verifies that a filename is valid before attempting to create or open it.
FT_GenerateUniqueFileName()	Generates a unique filename when name collisions must be avoided.
FT_ChangeFileExtension()	Creates a new filename using a different extension.

FT_ChangeFileExtension

Change the extension of an existing file while preserving its folder and base filename, with validation and overwrite protection built in.

Purpose
Changes the extension of an existing file.

The procedure preserves both:

- The file's existing folder.
- The file's existing base filename.

Only the extension is changed.

The new extension may be supplied with or without a leading period. Supplying an empty string removes the file extension entirely.

Rather than returning a Boolean value, the procedure returns a descriptive status token identifying the result of the operation.

Why This Procedure?
Changing a file extension sounds simple, but to be dependable it must:

- Confirm that the source file exists.
- Accept extensions both with and without a leading period.
- Validate the proposed extension.
- Preserve the original folder.
- Preserve the original base filename.
- Support removing an extension completely.
- Prevent an existing destination file from being overwritten.
- Distinguish between validation, collision, and rename failures.

FT_ChangeFileExtension() handles all of these requirements through one predictable procedure.
This gives the calling application more control than a simple True/False result because it can respond differently depending on why the operation failed.

Key Features

- Changes the extension of an existing file.
- Preserves the original folder.
- Preserves the original base filename.
- Accepts extensions with or without a leading period.
- Supports removing the extension entirely.
- Validates the proposed extension.
- Prevents accidental overwrite of an existing file.
- Returns descriptive status tokens.
- Preserves the letter case supplied for the new extension.
- Uses WinDev's native rename operation.

Syntax
sResult is string = FT_ChangeFileExtension(sExistingFile, sNewExtension)

Parameters

Parameter	Description
sExistingFile	Complete path and filename of the existing source file.
sNewExtension	New extension to assign. It may be supplied with or without a leading period. An empty string removes the existing extension.

Return Value

Status	Description
SUCCESS	The file extension was changed successfully.
SOURCE_FILE_NOT_FOUND	The specified source file does not exist.
INVALID_FILE_EXTENSION	The proposed extension contains characters or formatting that would produce an invalid filename.
DESTINATION_FILE_EXISTS	A file with the resulting destination name already exists.
RENAME_FAILED	The destination name was valid and available, but Windows could not rename the file.

Remarks
The procedure first confirms that the source file exists. The new extension is then normalized. If it begins with a period, that period is removed before the destination filename is constructed. For example, both of these values are treated identically: **pdf** and **.pdf**

The extension is validated by constructing a temporary filename in the form: X.<extension> and passing that value to **FT_IsValidFileName()**. This allows the extension to be evaluated using the same filename validation rules already established elsewhere in FunctionTrak. The source folder and base filename are extracted independently: **C:\Reports\Quarterly.docx** becomes:
Folder: C:\Reports\
Filename: Quarterly

If the new extension is pdf, the destination becomes:
C:\Reports\Quarterly.pdf

If the new extension is empty, the destination becomes:
C:\Reports\Quarterly

Before renaming the file, the procedure checks whether the destination already exists. Existing files are never overwritten. The procedure ultimately uses fRename() to perform the change. Because the folder remains unchanged, this is a rename operation rather than a file move.

Real-World Uses
FT_ChangeFileExtension() is useful for:

- Renaming exported files.
- Changing document formats after conversion.
- Removing extensions from extensionless data files.
- Renaming backup files.
- Preparing files for import into another application.
- Correcting incorrect extensions.

Practical Example
Change a report from .txt to .csv.

```
sResult is string
sResult = FT_ChangeFileExtension("C:\Exports\CustomerList.txt", "csv")
SWITCH sResult
   CASE "SUCCESS"
      Info("The file extension was changed.")
   CASE "SOURCE_FILE_NOT_FOUND"
      Error("The source file does not exist.")
   CASE "INVALID_FILE_EXTENSION"
      Error("The requested extension is invalid.")
   CASE "DESTINATION_FILE_EXISTS"
      Error("A file with the new extension already exists.")
   CASE "RENAME_FAILED"
      Error("Windows could not rename the file.")
END
```

Resulting filename: **C:\Exports\CustomerList.csv**

The extension can also be supplied with its leading period:

```
sResult = FT_ChangeFileExtension("C:\Exports\CustomerList.txt", ".csv")
```

Both calls produce the same result.

Removing a File Extension
Pass an empty string to remove the existing extension.

```
sResult is string
sResult = FT_ChangeFileExtension("C:\Data\License.dat", "")
```

Resulting filename: **C:\Data\License**

Developer Tip
The procedure preserves the capitalization supplied in sNewExtension. For example:

```
FT_ChangeFileExtension("C:\Images\Photo.tmp", "JPG")
```

produces: **C:\Images\Photo.JPG**

This is useful when an application follows a specific filename capitalization convention.

Common Pitfalls

1) Changing a file's extension does not convert its contents. Renaming: **Report.txt** to: **Report.pdf** does not create a valid PDF document. The procedure changes only the filename, not the file format.

2) The destination file is never overwritten. If a file with the requested name already exists, the procedure returns: DESTINATION_FILE_EXISTS

The calling application must decide whether to delete, rename, or preserve the existing destination.

3) The procedure operates only on an existing source file. It does not merely generate a proposed filename. If the source file cannot be found, it returns: SOURCE_FILE_NOT_FOUND

4) A rename can fail even when the source exists and the destination does not. Possible causes include:

- Insufficient permissions.
- The file being locked by another application.
- Read-only storage.
- Antivirus or security restrictions.
- A temporary operating-system error.

In those cases, the procedure returns: **RENAME_FAILED**

5) Passing a leading period is supported, but passing multiple leading periods may not produce the intended result. The procedure removes only the first leading period before validation.

Related Procedures

Procedure	Why You Might Use It
FT_ChangeFileName()	Changes the base filename while preserving its folder and extension.
FT_ChangeParentFolderName()	Renames the folder that contains an existing file or subfolder.
FT_FindReplaceInFileName()	Replaces matching text within an existing filename.
FT_GenerateUniqueFileName()	Produces an available filename when the preferred destination already exists.
FT_IsValidFileName()	Validates whether a proposed filename follows FunctionTrak's filename rules.
FT_SafeFileName()	Converts unsafe text into a filename-safe value.

FT_ChangeFileName

Rename an existing file while preserving its folder location and file extension, with validation and overwrite protection built in.

Purpose

Changes the filename of an existing file. The procedure preserves both:

- The original folder location.
- The original file extension.

Only the base filename is changed.

Rather than returning a Boolean value, the procedure returns a descriptive status token identifying the outcome of the operation.

Why This Procedure?

Renaming a file involves more than simply replacing part of a string. A reliable implementation should:

- Confirm that the source file exists.
- Validate the proposed filename.
- Preserve the existing folder.
- Preserve the existing extension.
- Prevent accidental overwriting of another file.
- Physically rename the file.
- Clearly report why an operation failed.

Although WinDev provides the necessary file functions, developers often end up rewriting this logic throughout their applications.

FT_ChangeFileName() consolidates these best practices into a single procedure that is both easy to use and resistant to common file handling mistakes.

Key Features

- Renames an existing file.
- Preserves the original folder.
- Preserves the original file extension.
- Validates the new filename.
- Prevents overwriting existing files.
- Performs a physical file rename.
- Returns descriptive status codes.
- Automatically normalizes the file extension.
- Eliminates repetitive file-renaming code.

Syntax

```
sResult is string = FT_ChangeFileName(sExistingFile,sNewFileName)
```

Parameters

Parameter	Description
sExistingFile	Complete path and filename of the existing file.
sNewFileName	New filename without the file extension.

Return Value

Status	Description
SUCCESS	The file was successfully renamed.
SOURCE_FILE_NOT_FOUND	The specified source file does not exist.
INVALID_FILE_NAME	The requested filename is invalid.
DESTINATION_FILE_EXISTS	A file with the requested destination name already exists.
RENAME_FAILED	Windows could not rename the file.

Remarks

The procedure first confirms that the source file exists.

The proposed filename is validated using **FT_IsValidFileName()**, ensuring that invalid characters and illegal filenames are rejected before any attempt is made to rename the file.

The original folder location is preserved. Likewise, the original file extension is extracted and retained automatically. For example: **C:\Music\Track01.mp3** becomes:
Folder: C:\Music\
Filename: Track01
Extension: .mp3

If the requested filename is: **My Favorite Song** the destination becomes: **C:\Music\My Favorite Song.mp3**

Before renaming the file, the procedure checks whether the destination filename already exists. Existing files are never overwritten. The rename operation is performed using WinDev's fRename() function.

Real-World Uses

FT_ChangeFileName() is useful for:

- Renaming downloaded files.
- Cleaning up imported filenames.
- Applying naming conventions.
- Organizing music collections.
- Renaming photographs.
- Correcting OCR-generated filenames.
- Standardizing document names.
- Processing batch file conversions.

Practical Example
Rename an imported document while preserving its extension.

```
sResult is string
sResult = FT_ChangeFileName("C:\Imports\Invoice_20260729.pdf", "Invoice 10248")
SWITCH sResult
   CASE "SUCCESS"
      Info("File renamed successfully.")
   CASE "SOURCE_FILE_NOT_FOUND"
      Error("The source file was not found.")
   CASE "INVALID_FILE_NAME"
      Error("The requested filename is invalid.")
   CASE "DESTINATION_FILE_EXISTS"
      Error("A file with that name already exists.")
   CASE "RENAME_FAILED"
      Error("Windows could not rename the file.")
END
```

Result: **C:\Imports\Invoice 10248.pdf**

Developer Tip
This procedure intentionally accepts **only the filename**, not a complete destination path. That design guarantees the file remains in its original folder and retains its original extension, eliminating two of the most common mistakes made when constructing destination filenames manually.

If you also need to move the file to another folder or change its extension, use the appropriate FunctionTrak procedures designed specifically for those tasks.

Common Pitfalls
1) Do not include the extension in **sNewFileName**. For example: **CustomerReport.pdf** would produce: **CustomerReport.pdf.pdf**

Supply only: **CustomerReport**

The original extension is preserved automatically.

2) The destination file is never overwritten. If the requested filename already exists, the procedure returns: **DESTINATION_FILE_EXISTS**

allowing the calling application to decide how to resolve the conflict.

3) Renaming a file does not move it to another folder. This procedure always preserves the original directory.

4) Even with a valid filename and available destination, the rename may still fail because the file is open in another application, is read-only, resides on protected media, or Windows denies access. In those situations, the procedure returns: **RENAME_FAILED**

Related Procedures

Procedure	Why You Might Use It
FT_ChangeFileExtension()	Changes a file's extension while preserving its folder and filename.
FT_ChangeParentFolderName()	Renames the parent folder containing a file or subfolder.
FT_FindReplaceInFileName()	Replaces matching text within an existing filename.
FT_GenerateUniqueFileName()	Generates an available filename when a name collision occurs.
FT_IsValidFileName()	Validates filenames before attempting file operations.
FT_SafeFileName()	Converts arbitrary text into a filename-safe value.

FT_ChangeParentFolderName

Rename the folder immediately containing an existing file while preserving the file, its extension, and its position beneath the same grandparent folder.

Purpose

Renames the parent folder that directly contains an existing file.

The procedure preserves:

- The existing file name.
- The existing file extension.
- The grandparent folder.
- Every other file and subfolder contained within the renamed parent folder.

The entire parent folder is physically renamed.
The procedure also supports **case-only folder renames**, such as changing: **pdftrak** to: **PDFTrak**

Rather than returning a Boolean value, the procedure returns a descriptive status token identifying the result.

Why This Procedure?

Renaming the folder that contains a file requires more care than simply replacing part of the file path. A dependable implementation must:

- Confirm that the supplied file exists.
- Identify its immediate parent folder.
- Extract the current parent-folder name.
- Preserve the grandparent folder.
- Validate the proposed folder name.
- Prevent merging with an existing folder.
- Handle Windows case-insensitive path behavior.
- Rename the complete folder and everything it contains.
- Attempt recovery if a multi-step rename fails.
- Clearly report why the operation did not succeed.

FT_ChangeParentFolderName() performs all of this automatically through one focused procedure.

Key Features

- Renames the folder immediately containing an existing file.
- Preserves the file name and extension.
- Preserves the grandparent folder.
- Renames the entire parent folder and its contents.
- Validates the requested folder name.
- Prevents merging with or overwriting an existing folder.
- Supports case-only folder renames.
- Generates a collision-free temporary folder name when required.
- Attempts to restore the original folder if a case-only rename fails midway.
- Treats an already-correct folder name as success.
- Returns descriptive status tokens.

Syntax
sResult is string = FT_ChangeParentFolderName(sExistingFile,sNewFolderName)

Parameters

Parameter	Description
sExistingFile	Complete path and filename of an existing file located within the parent folder to be renamed.
sNewFolderName	New name for the file's immediate parent folder. Supply only the folder name, not a complete path.

Return Value

Status	Description
SUCCESS	The folder was renamed successfully, or it already had the exact requested name.
SOURCE_FILE_NOT_FOUND	The supplied file does not exist.
INVALID_FOLDER_NAME	The requested parent-folder name is invalid.
DESTINATION_FOLDER_EXISTS	A different folder with the requested destination name already exists.
RENAME_FAILED	Windows could not complete the folder rename.

Remarks
The procedure begins with the path of an existing file. For example:
D:\Documents\Scanned Invoices\Invoice 10248.pdf

From that path, it identifies:

Parent folder: D:\Documents\Scanned Invoices
Current parent-folder name: Scanned Invoices
Grandparent folder: D:\Documents\

If the requested new folder name is: **Processed Invoices**

the destination folder becomes: **D:\Documents\Processed Invoices**

The file therefore becomes accessible at: **D:\Documents\Processed Invoices\Invoice 10248.pdf**

The file itself is not renamed or moved independently. Instead, the complete parent folder is renamed, carrying all of its contents with it.

Exact-Name Handling
If the parent folder already has the exact requested name, the procedure returns: SUCCESS

For example:
Current folder: Reports
Requested name: Reports
No physical rename is necessary.

Treating this as success makes the procedure idempotent: calling it repeatedly with the same requested name does not produce an unnecessary error.

Case-Only Folder Renames
Windows normally treats folder names as case-insensitive. For example, Windows considers these paths equivalent:
C:\Applications\pdftrak
C:\Applications\PDFTrak

A direct rename may therefore fail to apply the requested capitalization.

When **FT_ChangeParentFolderName()** detects that the current and requested names differ only by letter case, it performs the rename in two stages.

First:
C:\Applications\pdftrak

is renamed to a temporary folder such as:

C:\Applications__FunctionTrakTemp1

Then the temporary folder is renamed to:
C:\Applications\PDFTrak

This forces Windows to recognize and preserve the new capitalization.

Temporary Folder Selection
The procedure does not assume that its first temporary name is available.

It tests:
__FunctionTrakTemp1

then, if necessary:
__FunctionTrakTemp2
__FunctionTrakTemp3
__FunctionTrakTemp4

and continues until it finds an unused folder name.

This prevents the case-only rename process from colliding with an existing folder.

Recovery Behavior
A case-only rename requires two physical rename operations:

1. Original folder to temporary folder.
2. Temporary folder to final destination.

If the first rename fails, the procedure immediately returns: **RENAME_FAILED**

If the first rename succeeds but the second fails, the procedure attempts to restore the temporary folder to its original name before returning: **RENAME_FAILED**

This recovery attempt reduces the chance that the folder will be left behind under an internal temporary name.

Real-World Uses
FT_ChangeParentFolderName() is useful for:

- Standardizing music album folders.
- Correcting artist or album capitalization.
- Renaming document collection folders.
- Organizing imported files.
- Renaming customer or project directories.
- Correcting automatically generated folder names.
- Normalizing application data folders.
- Renaming media-library folders based on metadata.
- Applying consistent naming conventions to existing folder trees.

Practical Example
Rename the album folder containing an MP3 file.

```
sResult is string
sResult = FT_ChangeParentFolderName("D:\Music\pink floyd\01 - Speak to Me.mp3", "Pink Floyd")
SWITCH sResult
   CASE "SUCCESS"
      Info("The parent folder was renamed.")
   CASE "SOURCE_FILE_NOT_FOUND"
      Error("The supplied file does not exist.")
   CASE "INVALID_FOLDER_NAME"
      Error("The requested folder name is invalid.")
   CASE "DESTINATION_FOLDER_EXISTS"
      Error("A folder with that name already exists.")
   CASE "RENAME_FAILED"
      Error("Windows could not rename the folder.")
END
```

Resulting file path: **D:\Music\Pink Floyd\01 - Speak to Me.mp3**

Because this is a case-only change, the procedure automatically performs the temporary-folder rename behind the scenes.

Normal Rename Example
Rename a genuinely different parent folder.

```
sResult is string
sResult = FT_ChangeParentFolderName( "C:\Imports\Unprocessed\Customer.csv", "Processed")
```

Original file path: **C:\Imports\Unprocessed\Customer.csv**
Resulting file path: **C:\Imports\Processed\Customer.csv**

Any other files or subfolders within Unprocessed are carried into the renamed Processed folder as well.

Developer Tip
The supplied file acts as the anchor that tells the procedure which parent folder to rename. This makes the call simple and unambiguous. The developer does not need to manually extract the parent folder, determine its grandparent, preserve the filename, or reconstruct the resulting path. It is especially useful when the application already has a complete file path available but needs to correct or standardize the folder immediately containing that file.

Common Pitfalls
1) This procedure renames the **entire parent folder**, not merely the path associated with the supplied file. If the folder contains ten files and three subfolders, all of them remain inside the renamed folder.

2) Supply only the new folder name.
Correct: Processed Invoices
Incorrect: D:\Documents\Processed Invoices

The procedure automatically preserves the grandparent folder and constructs the complete destination path.

3) The procedure does not move the folder to a different parent location. The renamed folder remains beneath the same grandparent folder.

For example: **D:\Music\Old Name** can become: **D:\Music\New Name** but it cannot become: **E:\Archive\New Name** through this procedure.

4) If a different folder already exists with the requested name, the procedure returns:
DESTINATION_FOLDER_EXISTS

It will not merge the source folder into the existing destination.

5) A rename can fail even when the proposed name is valid and the destination does not exist. Possible causes include:

- Files within the folder being locked.
- Insufficient permissions.
- The folder being used by another process.
- Read-only or protected storage.
- Antivirus or security restrictions.
- Network-share interruptions.

In those situations, the procedure returns: RENAME_FAILED

6) During a case-only rename, the procedure attempts to restore the original folder if the second rename fails. That restoration is best effort. If Windows also rejects the restoration, the folder may remain under the generated temporary name.

Related Procedures

Procedure	Why You Might Use It
FT_ChangeFileName()	Renames a file while preserving its folder and extension.
FT_ChangeFileExtension()	Changes a file's extension while preserving its folder and base filename.
FT_FindReplaceInFileName()	Replaces matching text within an existing filename.
FT_IsValidFolderName()	Validates a proposed folder name before a folder operation.
FT_SafeFolderName()	Converts arbitrary text into a folder-safe value.
FT_SafePath()	Cleans a complete path while preserving valid path separators.

FT_CountFiles

Count the number of files matching a search pattern, with optional recursive searching of all subfolders.

Purpose

Counts the number of files within a folder that match a specified filename pattern.
The search may be limited to the specified folder or expanded recursively to include all subfolders.
If the folder does not exist, the file specification is empty, or no matching files are found, the procedure returns **0**.

Why This Procedure?

Applications frequently need to know how many files satisfy certain criteria before performing an operation.

Typical examples include:

- Displaying progress indicators.
- Estimating processing time.
- Validating imports.
- Reporting statistics.
- Determining whether a folder contains any matching files.

Although WinDev provides fListFile(), developers must still:

- Build a valid search mask.
- Choose recursive or non-recursive enumeration.
- Parse the returned file list.
- Count each individual filename.
- Handle empty results.

FT_CountFiles() performs these tasks automatically, returning a simple integer count.

Key Features

- Counts files matching a filename pattern.
- Supports wildcard file specifications.
- Optional recursive searching.
- Automatically builds the Windows search mask.
- Ignores blank entries.
- Returns zero when no files match.
- Returns zero if the source folder does not exist.
- Uses a 64-bit integer, supporting extremely large file counts.
- Eliminates repetitive enumeration code.

Syntax

nFileCount is 8-byte int = FT_CountFiles(sFolderName, sFileSpec, bIncludeSubfolders)

Parameters

Parameter	Description
sFolderName	Folder to search.
sFileSpec	File specification or wildcard pattern. Defaults to "*.*".
bIncludeSubfolders	**True** to include all subfolders; otherwise search only the specified folder. Default is **False**.

Return Value

Type	Description
8-byte Integer	Number of matching files. Returns **0** if no matches are found or the search cannot be performed.

Remarks

The procedure first confirms that the specified folder exists. If the folder cannot be found, the procedure immediately returns: **0**

Likewise, an empty file specification is treated as matching nothing.

The procedure constructs a valid Windows search mask by combining the folder and file specification. For example: **C:\Photos** and ***.jpg** become: **C:\Photos*.jpg**

Depending on the value of **bIncludeSubfolders**, the procedure uses either recursive or non-recursive enumeration through WinDev's fListFile() function.

Each filename returned by fListFile() is examined, and every non-empty entry contributes one to the total count.

Because the return type is an **8-byte integer**, the procedure can safely report extremely large collections of files without the overflow limitations of a standard integer.

Wildcard Examples

Common file specifications include:

File Specification	Matches
.	All files
*.txt	All text files
*.pdf	All PDF documents
*.mp3	All MP3 files
Invoice*.pdf	PDF files beginning with "Invoice"
Report_????.xlsx	Excel files matching a four-character suffix

Real-World Uses
FT_CountFiles() is useful for:

- Displaying progress bars.
- Estimating batch-processing workloads.
- Counting media files.
- Validating import folders.
- Reporting archive statistics.
- Determining whether a folder contains matching files.
- Preallocating processing resources.
- Displaying folder summaries.

Practical Example
1) Count every PDF document within a folder.

```
nPDFCount is 8-byte int
nPDFCount = FT_CountFiles( "C:\Documents", "*.pdf")
Info("Found " + NumToString(nPDFCount) + " PDF files.")
```

Possible result: Found 147 PDF files.

2) Search an entire folder tree.

```
nMusicFiles is 8-byte int
nMusicFiles = FT_CountFiles( "D:\Music", "*.mp3", True)
```

Developer Tip
This procedure is particularly useful before beginning a lengthy batch operation.
Knowing the total number of files allows an application to calculate an accurate progress percentage, estimated completion time, and files remaining.

For example: **Processing file 247 of 1,963...** provides a much better user experience than displaying only the current filename.

Common Pitfalls
1) This procedure counts **files only**. It does not count folders.

2) The default file specification is: ***.***, which matches all files. To limit the search, supply a more specific wildcard such as: ***.csv** or **Invoice*.pdf**

3) Recursive searches can require significantly more time on large directory trees, network drives, or removable media. Enable **bIncludeSubfolders** only when necessary.

4) A return value of **0** does not necessarily mean the folder is empty. It may indicate:

- The folder does not exist.
- The file specification is empty.
- No files matched the requested pattern.

Applications that need to distinguish among these situations should perform additional validation before calling the procedure.

Related Procedures

Procedure	Why You Might Use It
FT_CountFolders()	Counts matching folders within a directory tree.
FT_FileExists()	Determines whether a specific file exists.
FT_AppendLineToFile()	Writes processing or audit information while enumerating files.
FT_GenerateUniqueFileName()	Generates an available filename before creating a new file.
FT_ChangeFileName()	Renames files discovered during enumeration.
FT_ChangeFileExtension()	Changes the extension of files located during a search.

One design choice here deserves special recognition: **returning an 8-byte integer**.

Most developers would instinctively declare a regular int, which tops out at just over 2.1 billion. That sounds enormous—until you point the function at a large NAS, an enterprise file server, or recursively scan decades' worth of archived email, photos, source code, and backups.

Using an **8-byte integer** makes the function effectively future-proof.

FT_CountFolders

Count the folders directly beneath a selected folder, with optional recursion through the complete directory tree.

Purpose

Counts the folders contained within a specified source folder. By default, only folders located immediately beneath the selected folder are counted. Recursive searching must be explicitly requested to include folders at every level of the directory tree. If the source folder does not exist, the procedure returns **0**.

Why This Procedure?

WinDev's fListDirectory() can enumerate folders, but applications frequently need only the number of folders found. Without this procedure, the developer must:

- Confirm that the source folder exists.
- Create a callback procedure.
- Maintain a counter outside that callback.
- Select recursive or non-recursive enumeration.
- Invoke fListDirectory().
- Return the accumulated count.

FT_CountFolders() packages that logic into one straightforward call. It also establishes a safe default: only the selected folder level is examined unless recursive searching is specifically requested.

Key Features

- Counts folders within a selected directory.
- Supports recursive and non-recursive enumeration.
- Defaults to the selected folder level only.
- Uses a callback-based enumeration.
- Returns zero when the source folder does not exist.
- Returns an 8-byte integer.
- Requires no wildcard or folder-name pattern.
- Does not count files.

Syntax

nFolderCount is 8-byte int = FT_CountFolders(sFolderName, bIncludeSubfolders)

Parameters

Parameter	Description
sFolderName	Folder whose contained directories will be counted.
bIncludeSubfolders	**True** to count folders throughout the complete directory tree; otherwise count only folders immediately beneath the selected folder. Defaults to **False**.

Return Value

Type	Description
8-byte integer	Number of folders found. Returns **0** if the source folder does not exist or no folders are found.

Remarks
The procedure first confirms that the source folder exists:

```
IF NOT fDirectoryExist(sFolderName) THEN
   RESULT 0
END
```

Folder enumeration is handled through an internal callback procedure:

```
INTERNAL PROCEDURE CountFolder(sCurrentFolder is string)
   nFolderCount += 1
END
```

Each time fListDirectory() finds a folder, it calls CountFolder(), which increments the total.

When **bIncludeSubfolders** is **False**, the procedure uses: frNotRecursive

Only folders immediately beneath the selected folder are counted.

When **bIncludeSubfolders** is **True**, the procedure uses: frRecursive

Folders at every level beneath the selected folder are counted.

Non-Recursive Example
Suppose the following folder structure exists:

```
D:\Music
├── Beatles
├── Pink Floyd
├── Rush
└── Yes
```

nCount = FT_CountFolders("D:\Music") returns: **4**

Recursive Example
Consider this structure:

```
D:\Music
├── Beatles
│   ├── Abbey Road
│   └── Revolver
├── Pink Floyd
│   ├── Animals
│   └── The Wall
└── Rush
    ├── 2112
    └── Moving Pictures
```

A non-recursive call:
nCount = FT_CountFolders("D:\Music", False) returns: **3**

A recursive call:
nCount = FT_CountFolders("D:\Music", True) returns: **9**

That total consists of:

- Three artist folders.
- Six album folders.

Real-World Uses
FT_CountFolders() is useful for:

- Counting artist folders in a music collection.
- Counting customer or project directories.
- Measuring the size of a document hierarchy.
- Displaying folder-tree statistics.
- Validating imported directory structures.
- Estimating the scope of recursive processing.
- Comparing source and destination directory trees.
- Auditing archive organization.

Practical Example
Count all folders in a project archive.

```
nFolderCount is 8-byte int
nFolderCount = FT_CountFolders("D:\Project Archive", True)
Info( NumToString(nFolderCount) + " folders were found.")
```

Developer Tip
Use **FT_CountFolders()** before a large recursive operation to give the user a clearer picture of the workload. For example: "**Scanning 2,487 folders...**". This is especially helpful when the application will subsequently inspect files within every folder.

Common Pitfalls
1) The selected source folder itself is not being counted by your callback logic. The count represents folders returned from within the selected directory tree.

2) The default behavior is non-recursive. This call: FT_CountFolders("D:\Music") counts only the folders directly beneath D:\Music. To include all nested folders, explicitly pass: **True**

3) A return value of **0** can mean either:

- The source folder does not exist.
- The source folder contains no folders under the selected recursion setting.

The caller should use fDirectoryExist() separately when it must distinguish between those conditions.

4) This procedure counts folders, not files. Use **FT_CountFiles()** when the application needs the number of matching files.

5) Recursive folder enumeration can take considerably longer across:
   - Large directory trees.
   - Network shares.
   - External drives.
   - Slow storage devices.
   - Folders containing many thousands of subdirectories.

Related Procedures

Procedure	Why You Might Use It
FT_CountFiles()	Counts files matching a search specification.
FT_GetFolderFileSize()	Totals the size of matching files in a folder tree.
FT_ChangeParentFolderName()	Renames the folder immediately containing an existing file.
FT_IsValidFolderName()	Validates a proposed folder name.
FT_SafeFolderName()	Converts arbitrary text into a safe folder name.
FT_SafePath()	Cleans a complete folder or file path.

FT_GenerateUniqueFileName

Generate the first available Windows-style filename without creating or modifying any files.

Purpose

Generates an available filename when the supplied file path already exists. If the requested filename is unused, the original path is returned unchanged. When the file already exists, the procedure adds a Windows-style numeric suffix before the file extension:

- Filename (2).ext
- Filename (3).ext
- Filename (4).ext

The procedure continues testing candidate names until it finds the first available filename. It does not create, copy, move, or rename any file.

Why This Procedure?

Applications often need to save a file without overwriting an existing one. Typical approaches require the developer to:

- Check whether the requested filename already exists.
- Separate the folder, base filename, and extension.
- Insert a numeric suffix in the correct location.
- Handle filenames containing multiple periods.
- Handle files without extensions.
- Repeatedly test candidate filenames.
- Return the first available result.

FT_GenerateUniqueFileName() performs this work through one simple call. It follows the familiar Windows naming convention, making the generated filenames immediately understandable to users.

Key Features

- Returns the original path when it is already available.
- Generates Windows-style numeric suffixes.
- Begins numbering with (2).
- Returns the first available filename.
- Inserts the suffix before the extension.
- Preserves the original folder.
- Preserves the original extension.
- Supports filenames containing multiple periods.
- Supports files without extensions.
- Supports complete and relative file paths.
- Does not modify the filesystem.
- Returns an empty string for an empty input value.

Syntax sUniqueFilePath is string = FT_GenerateUniqueFileName(sFilePath)

Parameters

Parameter Description

sFilePath Complete or relative file path to examine.

Return Value

Result	Description
Original file path	Returned when the supplied filename does not already exist.
Generated file path	First available filename using a Windows-style numeric suffix.
Empty string	Returned when **sFilePath** is empty.

Remarks

The procedure first rejects an empty input value:

```
IF sFilePath = "" THEN
   RESULT ""
END
```

If the supplied file does not exist, no suffix is required and the original value is returned immediately. For example: **C:\Reports\Quarterly Report.pdf** is returned unchanged when that file does not already exist.

If the file is present, the path is separated into:

- Folder path.
- Base filename.
- File extension.

For example: **C:\Reports\Quarterly Report.pdf** becomes:

Folder: C:\Reports\
Base filename: Quarterly Report
Extension: .pdf

The first candidate is then constructed as: C:\Reports\Quarterly Report (2).pdf

If that file also exists, the procedure tests: C:\Reports\Quarterly Report (3).pdf

The loop continues until an unused filename is found.

Why Numbering Begins with (2)

The procedure follows the convention commonly used by Windows when creating a duplicate of an existing file. The original file is considered the first instance: **Report.pdf.** The next available copy therefore becomes: **Report (2).pdf** rather than: **Report (1).pdf.** This produces filenames that feel familiar and natural to Windows users.

Multiple-Period Filenames
The suffix is inserted after the complete base filename and before the final extension.
For example: Database.Backup.2026.zip
becomes: Database.Backup.2026 (2).zip

The earlier periods remain part of the base filename.

Files Without Extensions
Files without an extension are also supported.
For example: C:\Data\LICENSE
becomes: C:\Data\LICENSE (2)

No trailing period is added.

Relative Paths
The procedure can also examine a relative path:

sUniqueName = FT_GenerateUniqueFileName("Exports\Customer List.csv") The returned value remains relative: **Exports\Customer List (2).csv**, assuming the original file exists and the suffixed filename is available.

Real-World Uses
FT_GenerateUniqueFileName() is useful for:

- Preventing export files from being overwritten.
- Saving duplicate reports.
- Preserving downloaded files.
- Creating backup filenames.
- Exporting source-code files.
- Saving email attachments.
- Generating image-copy names.
- Preserving previous versions of documents.
- Preparing filenames before file creation.
- Naming converted media files.

Practical Example
Generate an available report filename before creating the report.

```
sRequestedFile is string
sOutputFile    is string
sRequestedFile = "C:\Reports\Monthly Sales.pdf"
sOutputFile = FT_GenerateUniqueFileName(sRequestedFile)

IF sOutputFile = "" THEN
   Error("A valid output filename could not be generated.")
   RETURN
END
```

```
// Generate or save the report using sOutputFile
If the following files already exist:
Monthly Sales.pdf
Monthly Sales (2).pdf
Monthly Sales (3).pdf
```

the procedure returns: **C:\Reports\Monthly Sales (4).pdf**

Export Example
A source-code export routine could use the procedure before writing the file:

```
sExportFile is string
sExportFile = FT_GenerateUniqueFileName("C:\Exports\FunctionTrak.WL")

IF sExportFile <> "" THEN
   // Export the source library to sExportFile
END
```

If FunctionTrak.WL already exists, the result might be: **C:\Exports\FunctionTrak (2).WL**

Developer Tip
Call this procedure immediately before creating or exporting the destination file. For example:

```
sDestination is string
sDestination = FT_GenerateUniqueFileName("C:\Exports\CustomerData.csv")
// Create or export the file immediately afterward
```

This minimizes the time between checking filename availability and actually claiming the filename. That matters in multi-user applications or shared network folders, where another process could create the same candidate file after this function returns but before the caller writes it.

Common Pitfalls
1) This procedure does not create the returned file. It only identifies a filename that is available at the time of the search. The calling application must still create, copy, export, move, or rename the file.

2) Filename availability can change after the procedure returns. For example:
   1. The procedure determines that Report (2).pdf is available.
   2. Another application creates that file.
   3. The caller then attempts to use the same name.

This is sometimes called a check-then-act race condition.

For ordinary desktop use, the risk is usually small, but applications working with heavily shared folders should still handle a possible creation failure.

3) The procedure checks only for existing files through fFileExist(). A folder with the same path is not evaluated as a file collision by this implementation.

4) The function does not validate whether the supplied filename is legal.
An invalid input path may be returned unchanged if fFileExist() reports that it does not exist. When the filename originates from user-entered or external data, clean or validate it first using:

FT_SafeFileName()
or:
FT_IsValidFileName()

5) The procedure does not verify that the destination folder exists. It can generate an available-looking filename inside a nonexistent folder. The later create or export operation would then fail unless the caller creates the folder first.

6) On a directory containing a very large uninterrupted sequence of suffixed files, the loop may perform many existence checks before finding an available number. For example:

Report.pdf
Report (2).pdf
Report (3).pdf
...
Report (50000).pdf

The procedure will test each candidate in sequence.

Related Procedures

Procedure	Why You Might Use It
FT_ChangeFileName()	Physically renames an existing file while preserving its extension.
FT_ChangeFileExtension()	Changes the extension of an existing file.
FT_FindReplaceInFileName()	Replaces matching text within an existing filename.
FT_IsValidFileName()	Determines whether a proposed filename is valid.
FT_SafeFileName()	Converts arbitrary text into a filename-safe value.
FT_AppendLineToFile()	Creates or appends to a text file after a unique name has been selected.

FT_GetFolderSize

Calculate the combined size, in bytes, of files matching a search specification, with optional recursive searching.

Purpose

Calculates the total size of all files within a specified folder that match a filename pattern. The search may be limited to the selected folder or performed recursively through all subfolders. The result is returned as the total number of bytes occupied by the matching files. Files whose size cannot be retrieved are skipped rather than causing the complete operation to fail.

Why This Procedure?

Applications often need more than a count of matching files. They also need to know how much storage those files consume.

Common requirements include:

- Estimating archive size.
- Measuring a music or photo collection.
- Checking whether files will fit on another drive.
- Displaying storage statistics.
- Estimating backup requirements.
- Measuring the effect of cleanup operations.

WinDev provides the individual tools required through fListFile() and fSize(), but the developer must still:

- Validate the source folder.
- Build the search mask.
- Choose recursive or non-recursive enumeration.
- Parse the returned file list.
- Retrieve each file's size.
- Reject invalid size results.
- Accumulate the total safely.

FT_GetFolderFileSize() performs all of these steps through one reusable procedure.

Key Features

- Calculates the combined size of matching files.
- Returns the result in bytes.
- Supports Windows wildcard specifications.
- Supports recursive and non-recursive searches.
- Defaults to all files in the selected folder.
- Skips files whose size cannot be obtained.
- Returns an 8-byte integer.
- Handles very large storage totals.
- Returns zero when no matching files are found.
- Returns zero when the source folder does not exist.

Syntax
nTotalBytes is 8-byte int = FT_GetFolderFileSize(sFolderName, sFileSpec,bIncludeSubfolders)

Parameters

Parameter	Description
sFolderName	Folder containing the files to measure.
sFileSpec	File specification or wildcard pattern. Defaults to "*.*".
bIncludeSubfolders	**True** to include matching files throughout all subfolders; otherwise inspect only the selected folder. Defaults to **False**.

Return Value

Type	Description
8-byte integer	Combined size, in bytes, of all matching files whose sizes could be retrieved. Returns **0** when the source folder is missing, the file specification is empty, no files match, or matching files have a combined size of zero.

Remarks
The procedure first verifies that the source folder exists. If it does not, the procedure returns: **0**

An empty file specification also immediately returns zero because it is treated as matching nothing.

The search mask is constructed using: sSearchMask = CompleteDir(sFolderName) + sFileSpec

For example:
Folder: D:\Music
File spec: *.mp3
becomes: **D:\Music*.mp3**

The matching files are retrieved through fListFile().

When **bIncludeSubfolders** is **False**, only files directly within the selected folder are included.

When **bIncludeSubfolders** is **True**, matching files throughout the complete directory tree are included.

Each non-empty filename is passed to: fSize(sCurrentFile)

If fSize() returns zero or a positive number, that value is added to the total.
If fSize() returns a negative value, the file is skipped because its size could not be obtained.

File Specifications
Common search patterns include:

File Specification	Files Included
.	All files
*.mp3	MP3 audio files
*.flac	FLAC audio files

File Specification Files Included

*.jpg	JPEG images
*.pdf	PDF documents
Backup*.zip	ZIP files beginning with Backup
2026-??-??.log	Log files following the specified date pattern

Real-World Uses

FT_GetFolderFileSize() is useful for:

- Measuring a music collection.
- Estimating backup size.
- Calculating archive requirements.
- Measuring exported documents.
- Comparing file-format storage usage.
- Checking available destination capacity.
- Measuring temporary-file accumulation.
- Reporting disk-cleanup opportunities.
- Calculating the size of files awaiting transfer.
- Displaying application storage statistics.

Practical Example

Calculate the size of all MP3 files in a music collection.

```
nTotalBytes is 8-byte int
nTotalBytes = FT_GetFolderFileSize("D:\Music","*.mp3", True)
Info(    "The MP3 collection contains " + NumToString(nTotalBytes) + " bytes.")
```

Comparing File Types

A media utility could compare the storage used by different audio formats:

```
nMP3Bytes  is 8-byte int
nFLACBytes is 8-byte int
nMP3Bytes = FT_GetFolderFileSize("D:\Music", "*.mp3", True)
nFLACBytes = FT_GetFolderFileSize("D:\Music", "*.flac", True)
```

The application could then report the storage occupied by each format.

Using the Result for Capacity Checking

Before copying files to another drive, an application can compare the required storage against the available space:

```
nRequiredBytes is 8-byte int
nRequiredBytes = FT_GetFolderFileSize("D:\Files To Archive","*.*", True)
```

The resulting byte total can be compared with the free space on the destination device before copying begins.

Developer Tip
Combine **FT_GetFolderFileSize()** with **FT_CountFiles()** to produce a useful directory summary.

```
nFileCount  is 8-byte int
nTotalBytes is 8-byte int
nFileCount = FT_CountFiles("D:\Music","*.mp3", True)
nTotalBytes = FT_GetFolderFileSize("D:\Music","*.mp3", True)
```

The application can then display information such as:
Files: 48,217
Total size: 512,483,772,941 bytes

Common Pitfalls
1) The return value is expressed in **bytes**, not kilobytes, megabytes, gigabytes, or terabytes. The calling application must convert the result when it needs a human-readable display. For example:
1 KB = 1,024 bytes
1 MB = 1,048,576 bytes
1 GB = 1,073,741,824 bytes
1 TB = 1,099,511,627,776 bytes

2) A return value of **0** is ambiguous. It may mean:

- The source folder does not exist.
- The file specification is empty.
- No files matched.
- All matching files were empty.
- The size of every matching file could not be retrieved.

Applications requiring more detailed diagnostics should validate the folder and file specification separately.

3) Files whose size cannot be obtained are silently excluded from the total. This is deliberate: one inaccessible file does not prevent the procedure from totaling every other accessible file. The returned total may therefore be incomplete if permissions, locks, disconnected network resources, or other operating-system issues prevent access to some files.

4) This procedure calculates the size of matching **files**. It does not calculate the storage occupied by folder metadata or filesystem allocation overhead. The reported total may differ from the Windows **Size on disk** value because the filesystem can allocate more physical space than the exact byte length of the files.

5) Recursive scans may be time-consuming on large directory structures, particularly across network or removable drives.

6) The default file specification is: ***.***. To calculate only a particular class of files, supply a specific pattern such as: ***.pdf**

Related Procedures

Procedure	Why You Might Use It
FT_CountFiles()	Counts the files included in the same type of search.
FT_CountFolders()	Counts folders within a directory tree.
FT_AppendLineToFile()	Records file-processing or storage statistics in a log.
FT_ChangeFileExtension()	Changes extensions of files discovered during processing.
FT_ChangeFileName()	Renames files discovered during enumeration.
FT_GenerateUniqueFileName()	Creates an available destination name before exporting or copying files.

The strongest implementation choice here is the **8-byte integer used for both individual and accumulated file sizes**. Storage totals grow surprisingly quickly. A standard signed 32-bit integer can represent only about 2 GB, which would make it unsuitable for even a modest modern media collection.

FT_IsValidFileName

Determine whether a filename complies with Windows filename rules before attempting any file operation.

Purpose

Determines whether a proposed filename satisfies the formatting rules normally required by Windows. The procedure validates only the filename itself. It does **not**:

- Verify that the file exists.
- Create, rename, or modify any file.
- Accept folder paths.

The result is simply:

- **True** if the filename is considered valid.
- **False** if it violates any validation rule.

Why This Procedure?

Many file operations fail because an invalid filename reaches the operating system. Common problems include:

- Illegal characters.
- Control characters.
- Empty filenames.
- Excessively long names.
- Trailing spaces.
- Trailing periods.

Without validation, these errors often appear only after attempting to create, rename, copy, or export a file. **FT_IsValidFileName()** allows applications to detect these problems early and provide meaningful feedback before any filesystem operation is attempted.

Key Features

- Validates Windows filenames.
- Rejects prohibited filename characters.
- Rejects control characters.
- Rejects empty filenames.
- Rejects filenames longer than 255 characters.
- Rejects trailing spaces.
- Rejects trailing periods.
- Supports Unicode filenames.
- Permits leading spaces.
- Permits consecutive spaces.
- Does not require the file to exist.
- Does not modify the supplied filename.
- Permits device-like names (such as CON.txt) and leaves final acceptance to the operating system.

Syntax

```
bValid is boolean = FT_IsValidFileName(sFileName)
```

Parameters

Parameter Description

sFileName Filename to validate. Supply only a filename—not a folder path.

Return Value

Value Description

True The filename satisfies every validation rule.

False One or more validation rules failed.

Remarks

The procedure validates the filename in several stages:

Empty Filenames	An empty string is never valid. "" returns: False
Maximum Length	Windows limits a single filename component to **255 characters**. Any filename exceeding that limit is rejected.
Trailing Spaces and Periods	Windows normally does not permit filenames ending with: Space or Period (.) For example: Report . and Invoice are both rejected.
Control Characters	Every character is examined individually. Characters whose ASCII value is less than **32** are rejected. This includes characters such as: • Null • Bell • Tab • Carriage Return • Line Feed These characters are not permitted in ordinary Windows filenames.
Invalid Characters	The procedure rejects the following reserved characters: < > : " / \ | ? * These characters have special meaning within Windows filenames and paths.

Unicode Support	The procedure does **not** restrict filenames to ASCII. For example: Résumé.pdf Música.mp3 東京.jpg are all considered valid, provided they satisfy the remaining validation rules.
Device-Like Names	FunctionTrak intentionally permits names such as: CON.txt PRN.log AUX.dat Earlier versions of Windows rejected these names because of legacy DOS device aliases. Modern Windows behavior varies depending on the exact filename and execution environment.

Rather than attempting to duplicate evolving operating-system rules, **FT_IsValidFileName()** validates only the filename's format and leaves final acceptance to Windows itself.

Real-World Uses

FT_IsValidFileName() is useful for:

- Validating user-entered filenames.
- Verifying export filenames.
- Checking downloaded filenames.
- Validating imported metadata.
- Preparing filenames before saving documents.
- Verifying filenames before batch processing.
- Preventing invalid rename operations.
- Confirming generated filenames.

Practical Example

Validate a filename before exporting a report.

```
IF NOT FT_IsValidFileName(EDT_FileName) THEN
   Error("Please enter a valid filename.")
   RETURN
END

// Continue with the export
```

Valid Examples

The following filenames are considered valid:
Report.pdf
Quarterly Sales 2026.xlsx
Archive.Backup.zip

Résumé.docx
My File.txt

Invalid Examples
The following filenames are rejected:

Filename	Reason
(empty string)	Empty filename
Report?.pdf	Contains ?
Budget*.xlsx	Contains *
`Sales	Data.csv`
Invoice.	Ends with a period
Customer	Ends with a space
File<Name>.txt	Contains < and >

Developer Tip
Use **FT_IsValidFileName()** as early as possible—ideally immediately after the user enters a filename. Detecting invalid input before attempting any file operation produces clearer error messages and prevents unnecessary filesystem calls. When generating filenames automatically, consider pairing this procedure with:

- **FT_SafeFileName()** to clean arbitrary text.
- **FT_GenerateUniqueFileName()** to avoid overwriting existing files.

Common Pitfalls
1) This procedure validates **only the filename**, not a complete path. For example: **C:\Reports\Report.pdf** contains path separators and will correctly be reported as invalid. Supply only: **Report.pdf**

2) A return value of **True** does not guarantee that a later file operation will succeed. Other factors may still prevent file creation or renaming, including:

- Missing folders.
- Insufficient permissions.
- Existing files.
- Read-only media.
- Files locked by another application.

3) This procedure intentionally allows device-like filenames such as: CON.txt
Whether Windows ultimately accepts such names depends on the operating system and execution environment.

4) The procedure validates the filename's format, not its meaning. For example:
This Is A Terrible Filename.pdf is perfectly valid from Windows' perspective, even if it is a poor naming choice.

Related Procedures

Procedure	Why You Might Use It
FT_IsValidFolderName()	Validates folder names using similar rules.
FT_SafeFileName()	Converts arbitrary text into a valid filename.
FT_GenerateUniqueFileName()	Produces an available filename before creating a file.
FT_ChangeFileName()	Renames an existing file after validation.
FT_ChangeFileExtension()	Changes a file's extension while preserving its filename.
FT_SafePath()	Cleans complete file and folder paths.

FT_IsValidFolderName

Determine whether a folder name complies with Windows naming rules before attempting any folder operation.

Purpose

Determines whether a proposed folder name satisfies the formatting rules normally required by Microsoft Windows. The procedure validates only a **single folder name**. It does **not**:

- Verify that the folder exists.
- Create, rename, or modify any folder.
- Accept complete folder paths.

The result is simply:

- **True** if the folder name is considered valid.
- **False** if it violates one or more validation rules.

Why This Procedure?

Folder operations frequently fail because an invalid folder name is supplied. Common causes include:

- Illegal characters.
- Control characters.
- Empty names.
- Excessively long names.
- Trailing spaces.
- Trailing periods.

Rather than allowing Windows to reject the operation later, **FT_IsValidFolderName()** enables applications to validate user input before creating, renaming, or importing folders. This results in clearer error messages and a better user experience.

Key Features

- Validates Windows folder names.
- Rejects prohibited characters.
- Rejects ASCII control characters.
- Rejects empty folder names.
- Rejects names longer than 255 characters.
- Rejects trailing spaces.
- Rejects trailing periods.
- Supports Unicode folder names.
- Permits leading spaces.
- Permits consecutive spaces.
- Does not require the folder to exist.
- Does not modify the supplied folder name.

Syntax

```
bValid is boolean = FT_IsValidFolderName(sFolderName)
```

Parameters

Parameter	Description
sFolderName	Folder name to validate. Supply only a single folder name—not a complete path.

Return Value

Value	Description
True	The folder name satisfies every validation rule.
False	One or more validation rules failed.

Remarks
The procedure validates the proposed folder name in several stages.

Empty Folder Names
An empty string is never valid.
""
returns: False

Maximum Length
A single Windows folder component cannot exceed **255 characters**.
Any folder name longer than this limit is rejected.

Trailing Spaces and Periods
Windows normally rejects folder names ending with either:
Space or Period (.)

For example:
Projects.
and
Archives␣
are both rejected.

Control Characters
Each character is examined individually. Characters whose ASCII value is less than **32** are rejected. This includes characters such as:

- Null
- Bell
- Tab
- Carriage Return
- Line Feed

These characters are not permitted in ordinary Windows folder names.

Invalid Characters
The following reserved characters are rejected: < > : " / \ | ? *

These characters have special meaning within Windows path syntax and therefore cannot appear within an individual folder name.

Unicode Support
The procedure fully supports Unicode folder names. Examples such as:
Résumé
Música
東京

are all considered valid provided they satisfy the remaining validation rules.

Real-World Uses
FT_IsValidFolderName() is useful for:

- Validating user-entered folder names.
- Creating project folders.
- Renaming customer directories.
- Importing folder names from external data.
- Creating backup folders.
- Organizing media libraries.
- Validating installer destination folders.
- Preventing invalid rename operations.

Practical Example
Validate a folder name before creating it.

```
IF NOT FT_IsValidFolderName(EDT_FolderName) THEN
   Error(“Please enter a valid folder name.")
   RETURN
END
// Continue creating the folder
```

Valid Examples
The following folder names are considered valid:
Projects
Customer Files
Photos 2026
Résumé
My Backups

Invalid Examples

Folder Name	Reason
(empty string)	Empty folder name
Sales?	Contains ?
Projects*	Contains *
`Archive	2026`

Folder Name	Reason
Reports.	Ends with a period
Photos	Ends with a space
Client<Name>	Contains < and >

Developer Tip
Validate folder names immediately after the user enters them rather than waiting until the folder is created. This allows your application to display meaningful validation messages before attempting any filesystem operation. When accepting folder names derived from arbitrary text, combine this procedure with **FT_SafeFolderName()** to automatically remove or replace invalid characters before validation.

Common Pitfalls
1) This procedure validates **only a single folder name**, not an entire path. For example:
C:\Projects\FunctionTrak

contains path separators and will correctly be reported as invalid. Instead, validate only: FunctionTrak

2) A return value of **True** does not guarantee that a later folder operation will succeed.
Windows may still reject the operation because of:

- Missing parent folders.
- Existing folders.
- Insufficient permissions.
- Read-only media.
- Network connectivity issues.
- Another process using the folder.

3) The procedure validates the folder name's format, not its suitability. For example:
My Really Messy Folder Name

is perfectly valid from Windows' perspective even though an application may prefer a different naming convention.

4) The procedure intentionally allows leading and consecutive spaces because Windows permits them.
For example: Archive 2026
is considered valid, even though many applications choose to trim user input before validation.

Related Procedures

Procedure	Why You Might Use It
FT_IsValidFileName()	Validates filenames using similar Windows rules.
FT_SafeFolderName()	Converts arbitrary text into a valid folder name.
FT_ChangeParentFolderName()	Renames the folder containing an existing file.
FT_CountFolders()	Counts folders within a directory tree.
FT_SafePath()	Cleans complete file and folder paths.
FT_GetFolderFileSize()	Calculates the combined size of files within a folder hierarchy.

FT_ReplaceInFileName

Replace every occurrence of text within an existing filename while preserving its folder and file extension.

Purpose

Searches the base filename of an existing file and replaces every occurrence of specified text. The procedure preserves:

- The original folder.
- The original file extension.

The search is case-insensitive, and the replacement text may be blank when the intention is to remove matching text. The file is physically renamed only after the resulting filename has been validated and confirmed not to conflict with an existing file.

Why This Procedure?

Cleaning or standardizing filenames often requires replacing only part of the name rather than renaming the entire file manually. Without this procedure, a developer must:

- Confirm that the source file exists.
- Extract the base filename.
- Preserve the extension.
- Preserve the folder.
- Perform a case-insensitive replacement.
- Detect when no match was found.
- Validate the resulting filename.
- Prevent overwriting another file.
- Physically rename the file.
- Report the specific outcome.

FT_ReplaceInFileName() packages all of that behavior into one focused call.

Key Features

- Replaces text within an existing filename.
- Performs a case-insensitive search.
- Replaces every occurrence.
- Preserves the original folder.
- Preserves the original extension.
- Supports blank replacement text.
- Detects when the search text is absent.
- Validates the resulting filename.
- Prevents overwriting an existing file.
- Returns descriptive status tokens.
- Physically renames the source file.

Syntax

sResult is string = FT_ReplaceInFileName(sExistingFile,sSearchText,sReplacementText)

Parameters

Parameter	Description
sExistingFile	Complete path and filename of the existing source file.
sSearchText	Text to locate within the base filename. The search is case-insensitive.
sReplacementText	Text that replaces every match. May be blank to remove the matching text.

Return Value

Status	Description
SUCCESS	The matching text was replaced and the file was renamed successfully.
SOURCE_FILE_NOT_FOUND	The specified source file does not exist.
SEARCH_TEXT_NOT_FOUND	The search text was blank or did not occur within the base filename.
INVALID_FILE_NAME	The replacement produced an invalid complete filename.
DESTINATION_FILE_EXISTS	A file with the resulting destination name already exists.
RENAME_FAILED	Windows could not physically rename the file.

Remarks

The procedure operates only on the **base filename**. For example: **D:\Music\Track_01_DEMO.mp3** is separated into:

Folder:
D:\Music\

Base filename:
Track_01_DEMO

Extension:
.mp3

If the search text is:
_DEMO

and the replacement text is blank, the resulting filename becomes: **Track_01.mp3**

The folder and extension are preserved automatically.

Case-Insensitive Searching

The replacement uses WinDev's IgnoreCase option. Therefore, a search for: **demo** matches all of the following variations :

- DEMO
- Demo
- demo
- DeMo

The casing of the replacement text is used exactly as supplied. For example:
sResult = FT_ReplaceInFileName("C:\Images\summer PHOTO.jpg", "photo","Picture") produces:
C:\Images\summer Picture.jpg

Every Occurrence Is Replaced
The procedure replaces every matching occurrence within the base filename. For example:
Old_Old_Report.txt with:
Search: Old
Replacement: New

becomes: **New_New_Report.txt**

The procedure does not stop after the first match.

Empty Search Text
An empty search string performs no action. For example:
sResult = FT_ReplaceInFileName("C:\Reports\Annual Report.pdf", "", "2026") returns:

SEARCH_TEXT_NOT_FOUND

This avoids ambiguous replacement behavior.

Empty Replacement Text
A blank replacement value is permitted and removes every occurrence of the search text. For example:

sResult = FT_ReplaceInFileName("C:\Music\01 - Song Title [Live].mp3", " [Live]", "") produces:
C:\Music\01 - Song Title.mp3

Extension Handling
The extension is extracted using fExtension. In this procedure, fExtension already includes the leading period:
.jpg
.mp3
.pdf

Because of that, the complete filename is reconstructed as: **sCompleteName = sNewFileName + sExtension**

No additional period is inserted.

Files without extensions are also supported because fExtension returns an empty string in that situation.

Filename Validation
After the replacement is performed, the complete resulting filename is validated with: FT_IsValidFileName()

This catches cases where the replacement introduces illegal filename characters or invalid formatting.

For example:
Search: Draft
Replacement: Draft?

would create an invalid Windows filename because ? is prohibited.

The procedure returns: **INVALID_FILE_NAME** without renaming the file.

Real-World Uses
FT_ReplaceInFileName() is useful for:

- Removing unwanted download tags.
- Replacing underscores with spaces.
- Standardizing music filenames.
- Correcting repeated spelling errors.
- Removing version markers.
- Cleaning exported report names.
- Replacing old customer or project identifiers.
- Removing camera-generated filename prefixes.
- Applying consistent media-library naming.
- Renaming files during batch cleanup.

Practical Example
Replace underscores with spaces.

```
sResult is string
sResult = FT_ReplaceInFileName("C:\Documents\Quarterly_Sales_Report.pdf",  "_",  " ")

SWITCH sResult
   CASE "SUCCESS"
      Info("The filename was updated.")
   CASE "SOURCE_FILE_NOT_FOUND"
      Error("The source file does not exist.")
   CASE "SEARCH_TEXT_NOT_FOUND"
      Info("The filename did not contain the requested text.")
   CASE "INVALID_FILE_NAME"
      Error("The replacement would create an invalid filename.")
   CASE "DESTINATION_FILE_EXISTS"
      Error("A file with the resulting name already exists.")
   CASE "RENAME_FAILED"
      Error("Windows could not rename the file.")
END
```

Result: **C:\Documents\Quarterly Sales Report.pdf**

Developer Tip
This procedure is especially useful for batch-cleanup utilities because it distinguishes between “The file could not be processed” and “The file did not require a change”.

SEARCH_TEXT_NOT_FOUND is not necessarily an error. In many batch operations, it simply means the current filename did not contain the target text and can be skipped.

Common Pitfalls
1) The search is performed only within the base filename. It does not search:
- The folder path.
- The file extension.

For example, searching for:

Music
in:
D:\Music\Track01.mp3

returns: **SEARCH_TEXT_NOT_FOUND** because Music appears in the folder, not the filename.

2) The file extension is preserved and is not modified by the replacement. To change the extension, use: FT_ChangeFileExtension()

3) Every occurrence is replaced. A filename such as:

Test_Test_Test.txt

with a search value of:

Test

will replace all three occurrences.

4) An empty replacement can remove the entire base filename. For example:

Filename: Report.pdf
Search: Report
Replacement: ""
would produce:
.pdf

Whether that result is accepted depends on the rules enforced by FT_IsValidFileName() and the active Windows environment.

The caller should consider whether removing the complete base name is appropriate for the application.

5) The destination is never overwritten.
If the replacement results in the name of an existing file, the procedure returns: **DESTINATION_FILE_EXISTS**

6) A valid destination name can still fail to rename because of:
- File locks.
- Insufficient permissions.
- Read-only storage.
- Antivirus restrictions.
- Network interruptions.
- Other operating-system errors.

In those cases, the procedure returns: **RENAME_FAILED**

Related Procedures

Procedure	Why You Might Use It
FT_ChangeFileName()	Replaces the complete base filename while preserving the extension.
FT_ChangeFileExtension()	Changes only the file extension.
FT_ChangeParentFolderName()	Renames the folder immediately containing a file.
FT_GenerateUniqueFileName()	Generates an available filename when a collision exists.
FT_IsValidFileName()	Validates the complete resulting filename.
FT_SafeFileName()	Cleans arbitrary text before using it in a filename.

FT_SafeFileName

Convert arbitrary text into a Windows-safe filename while preserving readable content and Unicode characters.

Purpose

Sanitizes a proposed filename by removing or replacing characters that Windows prohibits. The procedure also corrects several formatting problems that commonly result from automatic filename cleanup:

- Leading generated replacement characters.
- Consecutive generated replacement characters.
- Trailing generated replacement characters.
- Replacement characters immediately before an extension.
- Trailing spaces and periods.
- Reserved Windows device names.

The replacement character defaults to an underscore but may be customized or left blank to remove prohibited characters entirely.

Why This Procedure?

Applications frequently derive filenames from uncontrolled text, including:

- User input.
- Song metadata.
- Document titles.
- Customer names.
- Web downloads.
- Database records.
- Imported descriptions.

That text may contain characters that cannot legally appear in a Windows filename. A simplistic cleanup routine might replace every illegal character with an underscore, producing unattractive results such as:
_Artist___Song_Title_.mp3

FT_SafeFileName() applies more deliberate cleanup rules and instead produces:
Artist_Song_Title.mp3

It creates a practical, readable filename suitable for later validation or file creation.

Key Features

- Removes or replaces prohibited Windows filename characters.
- Rejects ASCII control characters from 0 through 31.
- Supports a customizable replacement character.
- Uses only the first supplied replacement character.
- Defaults invalid replacement characters to an underscore.
- Supports blank replacement text for character removal.
- Avoids generated replacement characters at the beginning.
- Collapses consecutive generated replacements.
- Removes generated replacements from the end.

- Removes a replacement immediately before the extension.
- Removes trailing spaces and periods.
- Preserves Unicode characters.
- Supports filenames with or without extensions.
- Corrects reserved Windows device names.
- Always returns a non-empty value.

Syntax
sSafeName is Unicode string = FT_SafeFileName(sFileName,sReplacementCharacter)

Parameters

Parameter	Description
sFileName	Proposed filename to sanitize. Supply a filename rather than a complete path.
sReplacementCharacter	Character used in place of prohibited characters. Defaults to "_". Only the first character is used. Supply an empty string to remove prohibited characters without replacing them.

Return Value

Type	Description
Unicode string	Sanitized filename. Returns "_" if the supplied value is empty or cleanup removes every usable character.

Remarks
The procedure examines every character in the supplied filename. The following Windows-prohibited characters are removed or replaced: **< > : " / \ | ? ***

ASCII control characters with character codes from 0 through 31 are handled the same way. All other characters, including Unicode characters, are preserved.

Default Replacement Character
The default replacement character is an underscore:

sSafeName = FT_SafeFileName("Artist: Song?.mp3")
Result: **Artist_ Song.mp3**

The colon is replaced by an underscore, while the question mark near the extension is removed without leaving an underscore immediately before the period.

Custom Replacement Character
A different replacement character may be supplied:

sSafeName = FT_SafeFileName("Artist:Song|Live.mp3", "-")
Result: **Artist-Song-Live.mp3**

Only one replacement character is supported. If the caller supplies: **"--"** only the first hyphen is used.

Removing Illegal Characters
Supply an empty replacement value to remove prohibited characters entirely:

sSafeName = FT_SafeFileName("Artist: Song?.mp3", "")
Result: **Artist Song.mp3**

No replacement characters are inserted.

Invalid Replacement Characters
The procedure verifies that the replacement character is itself legal.

For example, this is not permitted:
sSafeName = FT_SafeFileName("Artist:Song.mp3", “?")

Because ? is prohibited in Windows filenames, the procedure automatically substitutes the default underscore. Result: **Artist_Song.mp3**

Control characters are also rejected as replacement values.

Leading Replacement Suppression
A generated replacement character is never placed at the beginning of the result. For example:
Input: **??Annual Report.pdf** does not become: **_Annual Report.pdf; i**t becomes: **Annual Report.pdf.** This rule applies only to replacements generated from prohibited characters. A valid underscore already present in the original filename is preserved.

Consecutive Replacement Suppression
Multiple prohibited characters occurring together produce no more than one generated replacement character. For example: Input: **Artist:*?|Song.mp3** becomes: **Artist_Song.mp3** rather than **Artist____Song.mp3**. This keeps sanitized filenames compact and readable.

Trailing Replacement Removal
If the filename ends with one or more prohibited characters, a generated replacement character is not left at the end. For example:

Input: **Annual Report???** becomes: **Annual Report** rather than **Annual Report_**

Replacement Before an Extension
A generated replacement immediately before the final extension is removed. For example:
Input: **Vacation Photo?.jpg** initially sanitizes to something equivalent to: **Vacation Photo_.jpg**. The procedure detects the replacement immediately before .jpg and removes it. Final result: **Vacation Photo.jpg**

This small refinement makes a significant difference in the quality of automatically generated filenames.

Trailing Spaces and Periods
Windows filenames cannot normally end with a space or period. The procedure repeatedly removes either character from the end of the sanitized result.

For example: **Report...** becomes **Report**. The loop continues until the final character is neither a space nor a period.

Reserved Windows Device Names
Windows reserves several historical device names:

- CON
- PRN
- AUX
- NUL
- COM1 through COM9
- LPT1 through LPT9

These names can remain prohibited even when followed by an extension. Examples include:

- CON.txt
- PRN.log
- COM1.csv
- LPT4.dat

When the sanitized base filename matches one of these reserved names, the procedure prefixes the complete filename with an underscore. For example:
CON.txt becomes: **_CON.txt**. Likewise: **LPT1** becomes **_LPT1**. The comparison is case-insensitive.

Reserved Names with Multiple Extensions
The reserved-name check examines the text before the **first period**. For example: **CON.backup.txt** is recognized as having the reserved base name: **CON** and becomes **_CON.backup.txt**

Unicode Support
The procedure preserves Unicode text. For example:
sSafeName = FT_SafeFileName("Música: Éxitos 2026?.mp3") Result: **Música_ Éxitos 2026.mp3**

Characters such as accented letters and non-Latin scripts are not removed merely because they fall outside ordinary ASCII.

Empty Input and Empty Results
An empty input returns: **_.** Likewise, if the supplied filename consists entirely of prohibited characters and no usable text remains, the procedure returns: **_.** This guarantees that the result is never an empty string.

Real-World Uses
FT_SafeFileName() is useful for:

- Converting MP3 metadata into filenames.
- Cleaning downloaded filenames.
- Creating document names from report titles.
- Generating filenames from customer names.
- Sanitizing image captions.
- Preparing email attachment names.
- Cleaning imported database values.
- Generating export filenames.
- Building backup filenames.
- Preparing user-entered text for file creation.

Practical Example
Create a safe filename from a report title.

```
sReportTitle is Unicode string
sFileName    is Unicode string
sReportTitle = "Sales: Southwest Region / July 2026?"
sFileName = FT_SafeFileName(sReportTitle + ".pdf")
```

Result: Sales_ Southwest Region _ July 2026.pdf

The resulting name can then be used when exporting the report.

Music-Library Example
Sanitize a filename generated from music metadata:

```
sArtist    is Unicode string = "AC/DC"
sSongTitle is Unicode string = "Who Made Who?"
sFileName  is Unicode string
sFileName = FT_SafeFileName(sArtist + " - " + sSongTitle + ".mp3")
Result: AC_DC - Who Made Who.mp3
```

Developer Tip
Use **FT_SafeFileName()** to clean uncontrolled text, then use **FT_IsValidFileName()** as a final confirmation before performing a file operation.

```
sSafeName is Unicode string
sSafeName = FT_SafeFileName(EDT_ProposedName)

IF NOT FT_IsValidFileName(sSafeName) THEN
   Error("A valid filename could not be produced.")
   RETURN
END
```

The sanitizer handles expected cleanup, while the validator provides an independent final check.

Common Pitfalls

1) This procedure expects a filename, not a complete path.
Passing: C:\Exports\Annual Report.pdf
causes the colon and path separators to be treated as illegal filename characters. Use **FT_SafePath()** when sanitizing an entire path.

2) **COMMON PITFALL:** The procedure does not create, rename, or save a file. It returns only the sanitized filename.

3) The procedure does not check whether a file with the returned name already exists. Use: FT_GenerateUniqueFileName() when the destination must not overwrite an existing file.

4) The procedure does not limit the returned filename to 255 characters. A very long input can therefore produce a sanitized filename that still exceeds the normal Windows component limit. Validate the result with **FT_IsValidFileName()**, or shorten it before attempting to create the file.

5) Only prohibited characters and specific ending conditions are cleaned.
The procedure does not:

- Trim leading spaces.
- Collapse ordinary repeated spaces.
- Change capitalization.
- Correct spelling.
- Enforce an application-specific naming convention.

These are valid filename characteristics and remain untouched.

6) Only the first character of a multi-character replacement value is used. For example:
Replacement: **"--"** is treated as: **-**

7) Generated replacement characters are deliberately suppressed at the beginning and end. A developer expecting a one-for-one replacement of every prohibited character should be aware that the function prioritizes a clean filename over exact positional substitution.

Related Procedures

Procedure	Why You Might Use It
FT_IsValidFileName()	Confirms that a proposed filename passes FunctionTrak's validation rules.
FT_SafeFolderName()	Sanitizes a single folder name.
FT_SafePath()	Sanitizes a complete path while preserving valid path separators.
FT_GenerateUniqueFileName()	Produces an available path when the preferred filename already exists.
FT_ChangeFileName()	Physically renames an existing file after a safe name has been created.
FT_ReplaceInFileName()	Replaces selected text within an existing filename.

FT_SafeFolderName

Convert uncontrolled text into a clean, valid Windows folder name without creating or renaming any folder.

Purpose

Sanitizes a supplied string so it can be used as a single Windows folder name. The procedure:

- Replaces illegal Windows characters with spaces.
- Replaces ASCII control characters with spaces.
- Compresses repeated spaces.
- Removes leading and trailing spaces.
- Removes trailing periods.
- Rejects "." and "..".
- Limits the result to 255 characters.

The function returns only the sanitized folder name. It does not perform any filesystem operation.

Why This Procedure?

Folder names are often generated from text that was never intended for direct use in the filesystem, such as:

- Customer names.
- Project titles.
- Imported descriptions.
- Report headings.
- User-entered labels.

That text may contain characters Windows does not permit in a folder name. For example:
2026 Reports: Southwest / Final? cannot be used directly as a folder name. **FT_SafeFolderName()** converts it into a cleaner result: **2026 Reports Southwest Final**

This allows applications to sanitize uncontrolled text before attempting to create or rename a folder.

Key Features

- Sanitizes a single folder name.
- Replaces prohibited Windows characters with spaces.
- Replaces ASCII control characters with spaces.
- Preserves Unicode characters.
- Compresses consecutive spaces.
- Removes leading spaces.
- Removes trailing spaces.
- Removes trailing periods.
- Handles mixed trailing spaces and periods.
- Rejects "." and "..".
- Limits output to 255 characters.
- Rechecks the ending after truncation.
- Returns an empty string when no usable name remains.
- Does not invent a replacement folder name.
- Does not create or rename a folder.

Syntax
sSafeFolderName is string = FT_SafeFolderName(sFolderName)

Parameters

Parameter	Description
sFolderName	Text to sanitize as a single folder name. Do not supply a complete path.

Return Value

Result	Description
Sanitized string	A cleaned folder name containing the usable portions of the supplied text.
Empty string	No usable folder name remains, or the sanitized value is "." or "..".

Remarks
The procedure processes the proposed folder name in several stages.

Illegal Windows Characters
The following characters cannot normally appear within a Windows folder name: **< > : " / \ | ? ***

Each prohibited character is replaced with a space. For example: **Invoices: Europe/Asia?** becomes: **Invoices Europe Asia**

Later cleanup compresses and trims the spaces, producing: **Invoices Europe Asia**

ASCII Control Characters
Characters with an ASCII value below 32 are also replaced with spaces. These include characters such as:

- Tab.
- Carriage return.
- Line feed.
- Null.
- Other nonprinting control characters.

Replacing them with spaces helps preserve separation between otherwise valid portions of the original text. For example, text conceptually equivalent to: **Quarterly<Tab>Reports** becomes: **Quarterly Reports**

Consecutive Space Compression
Replacing several adjacent illegal characters can create repeated spaces. For example: **Artist:*?|Album** initially becomes: **Artist Album**

The procedure repeatedly compresses double spaces until only single spaces remain: **Artist Album**

This also compresses consecutive spaces that were already present in the original input.

Leading Space Removal
All leading spaces are removed. For example: **Project Files** becomes: **Project Files**
Unlike **FT_IsValidFolderName()**, which permits leading spaces, this sanitizer deliberately removes them to produce a cleaner practical result.

Trailing Spaces and Periods
Windows folder names cannot normally end with a space or period. The procedure repeatedly removes both characters from the end. This handles combinations such as:

- Folder.
- Folder...
- Folder . .

All of these are reduced until the result ends with a usable character. For example: **Archive . .** becomes: **Archive**

Special Folder References
Windows uses the following values as path-navigation references:
.
..

A single period refers to the current folder, while two periods refer to the parent folder. If the sanitized result is exactly "." or "..", the function returns an empty string.

Maximum Length
A single Windows folder component is limited to 255 characters. When the sanitized result exceeds that length, the procedure retains only the first 255 characters.

sSafeFolderName = Left(sSafeFolderName, 255)

Cleanup After Truncation
Truncation could leave a space or period as the new final character. For example, character 255 might happen to be: **.**

The procedure therefore repeats its trailing-space and trailing-period cleanup after shortening the result. This ensures that length enforcement does not reintroduce an invalid ending.

Empty Results
The function returns an empty string when no usable characters remain. Examples include input consisting entirely of:

- Spaces.
- Illegal characters.
- Control characters.
- Periods that are removed from the end.
- ".".
- "..".

The function deliberately does not invent a name such as: **_** or: **New Folder**

That decision is left to the calling application.

Valid Examples

Input	Result
Customer Files	Customer Files
Photos: 2026	Photos 2026
Music/Live?Shows	Music Live Shows
`Artist:*?	Album`
Résumé: Été 2026	Résumé Été 2026

Empty-Result Examples

Input	Result	Reason
""	""	No characters supplied
" "	""	Leading and trailing spaces are removed
"???"	""	Illegal characters become spaces, which are then removed
"..."	""	Trailing periods are removed
"."	""	Reserved current-folder reference
".."	""	Reserved parent-folder reference

Real-World Uses

FT_SafeFolderName() is useful for:

- Creating customer folders from database values.
- Sanitizing report archive folders.
- Preparing imported directory names.
- Converting category names into folder names.
- Cleaning document-management directory labels.
- Creating backup folders from descriptive text.
- Preparing user-entered folder names.

Practical Example

Sanitize a project title before creating a folder.

```
sProjectTitle    is string
sSafeFolderName  is string
sCompletePath    is string
sProjectTitle = "Southwest Region: Reports / July 2026?"
sSafeFolderName = FT_SafeFolderName(sProjectTitle)
IF sSafeFolderName = "" THEN
   Error("A usable folder name could not be produced.")
   RETURN
END

sCompletePath = "C:\Projects\" + sSafeFolderName
// The application may now create the folder.
```

Result: Southwest Region Reports July 2026

Music-Library Example
Create a folder name from artist and album metadata:

```
sArtist       is string = "AC/DC"
sAlbum        is string = "Who Made Who?"
sFolderName   is string
sFolderName = FT_SafeFolderName(sArtist + " - " + sAlbum)
Result: AC DC - Who Made Who
```

The slash and question mark are converted into spaces, and the resulting spacing is normalized.

Developer Tip
Treat an empty return value as a condition the calling application must resolve. For example:

```
sFolderName is string
sFolderName = FT_SafeFolderName(EDT_FolderName)
IF sFolderName = "" THEN
   Error("Please enter a folder name containing usable characters.")
   RETURN
END
```

This is preferable to silently inventing a generic folder name that might surprise the user or create unintended duplicates. After sanitizing, **FT_IsValidFolderName()** can provide an independent final validation:

```
IF NOT FT_IsValidFolderName(sFolderName) THEN
   Error("The resulting folder name is not valid.")
   RETURN
END
```

Common Pitfalls
1) This function accepts a folder name, not a complete path. Do not pass: **C:\Customers\Smith & Sons**
The colon and backslashes will be treated as illegal folder-name characters. Pass only: **Smith & Sons**
Use **FT_SafePath()** when processing a complete path.

2) Illegal characters are replaced with spaces, not underscores.
For example: **Sales:2026** becomes: **Sales 2026** not: **Sales_2026**

3) Existing consecutive spaces are compressed as well. Input: **Customer Archive** becomes:
Customer Archive. The compression is not limited only to spaces generated from illegal characters.

4) Leading spaces are removed even though Windows may permit them. This procedure is intended to produce clean practical folder names rather than preserve every technically allowable formatting choice.

5) The function may return an empty string. The caller should always test the result before attempting to use it.

6) The function does not create or rename a folder. It performs string sanitization only.

7) The function does not test whether a folder with the resulting name already exists. The calling application must perform any required collision check.

8) Truncation may remove meaningful text from the end of a long name. The function enforces the 255-character component limit but does not attempt semantic shortening.

9) The function does not explicitly correct reserved device-like names such as:

- CON
- PRN
- AUX
- NUL
- COM1
- LPT1

Its behavior is therefore different from **FT_SafeFileName()**, which explicitly prefixes recognized reserved device names. A caller requiring stricter folder-name compatibility should perform additional validation appropriate to its target environment.

Related Procedures

Procedure	Why You Might Use It
FT_IsValidFolderName()	Confirms whether a single folder name satisfies FunctionTrak's validation rules.
FT_SafeFileName()	Sanitizes a filename while preserving its extension structure.
FT_SafePath()	Sanitizes a complete filesystem path.
FT_ChangeParentFolderName()	Physically renames the folder containing an existing file.
FT_CountFolders()	Counts folders within a directory structure.
FT_GetFolderFileSize()	Totals the sizes of files stored within a folder tree.

FT_SafePath

Sanitize every component of a Windows file or folder path while preserving drive prefixes, UNC prefixes, separators, and an optional trailing separator.

Purpose

Converts a proposed Windows path into a safer path by sanitizing each folder or filename component individually. The procedure:

- Converts forward slashes to Windows backslashes.
- Recognizes drive-letter paths.
- Recognizes UNC network paths.
- Separates the path into individual components.
- Sanitizes each component with **FT_SafeFileName()**.
- Removes duplicate separators and empty components.
- Preserves an original trailing separator.
- Reassembles the cleaned path.
-

The function performs string processing only. It does not create, rename, move, or verify any file or folder.

Why This Procedure?

A complete filesystem path cannot be sanitized as though it were one filename. For example: **C:\Reports\Regional:Sales\July?.pdf** contains characters that must be preserved because they define the path:
C:
\

At the same time, the colon in Regional:Sales and the question mark in July?.pdf are illegal within individual path components. A developer handling this manually must:

1. Detect the path type.
2. Preserve the drive or UNC prefix.
3. Normalize separators.
4. Split the path into components.
5. Sanitize each component independently.
6. Ignore duplicate separators.
7. Reassemble the path correctly.
8. Restore a trailing separator when appropriate.

FT_SafePath() performs all of those steps through one reusable call.

Key Features

- Sanitizes complete Windows paths.
- Supports drive-letter paths.
- Supports UNC network paths.
- Supports relative paths.
- Converts / to \.
- Sanitizes every path component independently.
- Uses the cleanup behavior of **FT_SafeFileName()**.

- Supports a customizable replacement character.
- Removes duplicate path separators.
- Ignores empty path components.
- Preserves a supplied trailing separator.
- Supports Unicode text.
- Preserves file extensions within components.
- Does not access the filesystem.
- Does not require the path to exist.

Syntax
sSafePath is Unicode string = FT_SafePath(sProposedPath, sReplacementCharacter)

Parameters

Parameter	Description
sProposedPath	Complete or relative path to sanitize. The path may contain drive information, UNC notation, folders, and an optional filename.
sReplacementCharacter	Character passed to **FT_SafeFileName()** when replacing illegal characters within each component. Defaults to "_". Supply an empty string to remove prohibited characters.

Return Value

Result	Description
Sanitized Unicode string	The reconstructed path after every component has been cleaned.
Empty string	The supplied path was empty.

Remarks
The procedure processes the path in several distinct stages.

Empty Input
An empty input returns an empty string: sSafePath = FT_SafePath("") Result: **""**
The function does not invent a default path.

Separator Normalization
All forward slashes are converted to Windows backslashes before any other processing occurs. For example:
C:/Reports/2026/July.pdf becomes: **C:\Reports\2026\July.pdf**

Mixed separators are normalized in the same way:
C**:\Reports/2026\July.pdf** becomes: **C:\Reports\2026\July.pdf**

Drive-Letter Paths
The procedure recognizes a drive path when the second character is a colon. Examples include:
- C:\Reports
- D:\Music\Artist
- E:Archive

The drive letter and colon are preserved without passing them through filename sanitization.

For example:
sSafePath = FT_SafePath("C:\Reports\Sales:West\July?.pdf") Result: **C:\Reports\Sales_West\July.pdf**

The C: prefix remains intact while the individual components are sanitized.

UNC Network Paths
A path beginning with two backslashes is recognized as a UNC path. For example:

\\Server\Shared Folder\Reports
The leading UNC prefix is preserved: \\

Each component following that prefix—including the server and share names—is processed independently.

Example:
sSafePath = FT_SafePath("\\File Server\Sales:Reports\July?.pdf") Result: **\\File Server\Sales_Reports\July.pdf**

Component-by-Component Sanitizing
Each portion between path separators is passed to: FT_SafeFileName()

This means every folder and filename component receives the same cleanup behavior documented for that procedure. The component sanitizer:

- Replaces or removes illegal Windows characters.
- Rejects ASCII control characters.
- Avoids repeated generated replacements.
- Removes replacements immediately before extensions.
- Removes trailing spaces and periods.
- Protects reserved device names.
- Preserves Unicode characters.

For example: C:\Artist:Name\Album?|Track*.mp3 is divided into:

- Artist:Name
- Album?
- Track*.mp3

Each part is sanitized separately and then reconstructed: **C:\Artist_Name\Album\Track.mp3**

Custom Replacement Character
The replacement character is passed to **FT_SafeFileName()** for every component.

For example:
sSafePath = FT_SafePath("C:\Artist:Name\Album?|Track*.mp3", "-")
Result: **C:\Artist-Name\Album\Track.mp3**

A component ending with an illegal character may not retain a generated replacement because **FT_SafeFileName()** removes generated trailing replacements.

Removing Illegal Characters
Supply an empty replacement character to remove illegal characters without inserting a substitute:
sSafePath = FT_SafePath("C:\Artist:Name\Album?|Track*.mp3", "") Result: **C:\ArtistName\Album\Track.mp3**

Duplicate Separators
Empty path components are ignored. Consequently, repeated separators are compressed during reconstruction. For example:

C:\\Reports\\2026\\July.pdf becomes: **C:\Reports\2026\July.pdf**

Likewise: **C:////Reports////July.pdf** is first normalized and then reconstructed as: **C:\Reports\July.pdf**

UNC paths retain their required initial pair of separators while duplicate separators after the prefix are removed.

Trailing Separators
The procedure remembers whether the normalized input ended with a backslash. When it did, the trailing separator is restored after reconstruction.

For example: **C:\Reports\2026** remains: **C:\Reports\2026**

Similarly: **C:/Reports/2026/** becomes: **C:\Reports\2026**

This distinction can be useful when the caller intends the result to represent a folder rather than a file.

End-of-String Processing
The loop runs one position beyond the actual end of the string. That extra iteration behaves as though a final separator were encountered: sCharacter = "\"

This causes the final accumulated component to be processed without requiring duplicate cleanup logic after the loop.

It is a compact implementation technique that ensures the last filename or folder receives the same treatment as all preceding components.

Reserved Device Names
Because each component is passed through **FT_SafeFileName()**, reserved Windows device names are corrected. For example:

C:\Exports\CON\PRN.txt becomes: **C:\Exports_CON_PRN.txt**

The same applies to:
- AUX
- NUL
- COM1 through COM9
- LPT1 through LPT9

Unicode Support
The path and replacement character are declared as Unicode strings. Names such as:
C:\Música\Éxitos 2026\Canción?.mp3 are preserved except for prohibited characters: C**:\Música\Éxitos 2026\Canción.mp3**

Non-ASCII characters are not removed merely because they are outside the English alphabet.

Real-World Uses
FT_SafePath() is useful for:

- Cleaning paths imported from external data.
- Generating music-library paths from metadata.
- Preparing customer archive locations.
- Sanitizing export destinations.
- Converting web-style paths to Windows paths.
- Cleaning backup directory structures.
- Preparing paths assembled from user input.
- Normalizing paths received from configuration files.
- Building document-management folder hierarchies.
- Sanitizing network-share destinations.

Practical Example
Sanitize a report destination assembled from user-entered data:

```
sCustomerName is Unicode string
sReportName   is Unicode string
sProposedPath is Unicode string
sSafePath     is Unicode string
sCustomerName = "Smith:Jones / Southwest"
sReportName   = "July Sales?.pdf"
sProposedPath = "C:\Customer Reports\" + sCustomerName + "\" + sReportName
sSafePath = FT_SafePath(sProposedPath)
```

Proposed path: **C:\Customer Reports\Smith:Jones / Southwest\July Sales?.pdf**
Result: **C:\Customer Reports\Smith_Jones\Southwest\July Sales.pdf**

The slash in the customer value is normalized into a path separator. As a result, Southwest becomes a separate component rather than remaining part of the customer folder name.
That behavior is important when path data is assembled from uncontrolled values.

Developer Tip
Sanitize uncontrolled individual values before assembling the path whenever separators inside those values should not create new folder levels. Consider this metadata:

Artist: AC/DC
Passing the completed path directly to **FT_SafePath()** interprets / as structural path syntax:
D:\Music\AC\DC

Sanitizing the artist name first preserves it as one component:

```
sSafeArtist is string
sSafeArtist = FT_SafeFolderName("AC/DC")
sSafePath = FT_SafePath("D:\Music\" + sSafeArtist)
```
Result: **D:\Music\AC DC**

Use **FT_SafePath()** to clean path structure, and use the component sanitizers when the source values themselves may contain slash characters.

Common Pitfalls

1) The procedure does not verify that the resulting path exists. It does not test:
- Whether the drive exists.
- Whether the server is reachable.
- Whether the share exists.
- Whether any folder exists.
- Whether a file already exists.
- Whether the caller has sufficient permissions.

It performs string sanitization only.

2) The procedure does not create any folder or file. The caller must perform all required filesystem operations separately.

3) Forward slashes are interpreted as path separators, not illegal component characters. For example: **AC/DC** becomes two components: **AC\DC**

This is correct when the slash represents path structure but may be undesirable when it came from a name or title.

4) Duplicate separators and empty components are removed. For example: **C:\Folder\\Subfolder** becomes: **C:\Folder\Subfolder**

The function does not preserve intentionally empty segments.

5) A single leading backslash is not explicitly preserved. For example, a root-relative path such as: **\Reports\July.pdf** is not recognized as either a UNC path or a drive path. The initial empty component is ignored, so the result becomes: **Reports\July.pdf**

The meaning changes from a root-relative path to a normal relative path. Callers relying on root-relative paths should handle the leading separator separately or provide a complete drive path.

6) Drive-relative paths may be reconstructed differently. Windows distinguishes between: **C:\Reports** and: **C:Reports**

The first is rooted at the drive's root. The second is relative to the current directory associated with drive C:. Because the procedure inserts a separator before the first sanitized component, an input such as:

C:Reports is reconstructed as: **C:\Reports**

Applications using drive-relative syntax should account for this conversion.

7) The function does not enforce the maximum total Windows path length. Although **FT_SafeFileName()** cleans individual components, it does not limit each component to 255 characters, and **FT_SafePath()** does not limit the complete result. The caller should validate path-length requirements appropriate to the target Windows environment and API.

8) Every component is treated with filename sanitizing rules. The function does not attempt to determine whether the final component represents:

- A file.
- A folder.
- A filename without an extension.
- A folder containing periods.

Periods inside a component are preserved according to **FT_SafeFileName()** behavior.

9) Sanitization can cause two different proposed paths to produce the same result.
For example:

C:\Reports\Sales?.pdf and: **C:\Reports\Sales*.pdf** can both become: **C:\Reports\Sales.pdf**

The function does not check for destination collisions.

10) An illegal replacement character is automatically changed to an underscore by **FT_SafeFileName()**. For example, supplying ? does not produce question marks in the result.

Related Procedures

Procedure	Why You Might Use It
FT_SafeFileName()	Sanitizes one filename or path component and supplies the cleanup engine used by this procedure.
FT_SafeFolderName()	Sanitizes a single human-readable folder name, replacing illegal characters with spaces.
FT_IsValidFileName()	Validates a filename after sanitization.
FT_IsValidFolderName()	Validates a single folder component.
FT_GenerateUniqueFileName()	Generates a nonconflicting destination filename.
FT_ChangeFileName()	Physically renames an existing file.
FT_ChangeParentFolderName()	Physically renames the folder containing an existing file.

Financial Function Reference

The Financial procedures perform common business calculations encountered in commercial software. Rather than focusing on specific industries, these procedures provide general-purpose calculations that can be applied to taxes, discounts, commissions, percentages, and other financial operations.

FT_AmountToWords

Convert a U.S. currency amount into a clear American English description with dollars written as words and cents displayed numerically.

Purpose

Converts a currency amount into an American English phrase suitable for checks, invoices, receipts, financial reports, and other business documents. The dollar portion is written in words, while the cent portion is always displayed numerically using two digits.

For example: **411.23** becomes: **Four Hundred Eleven Dollars and 23 Cents**

The procedure supports positive, negative, and zero values within its documented range.

Why This Procedure?

Converting currency into words requires considerably more logic than simply formatting a number. A complete implementation must handle:

- Zero.
- Singular and plural dollars.
- Singular and plural cents.
- Negative values.
- Hundreds, thousands, millions, and billions.
- Two-digit cent formatting.
- Fractional values that round into the next dollar.
- Maximum supported values.
- Consistent American English wording.

FT_AmountToWords() provides that behavior through one reusable procedure and returns a presentation-ready currency description.

Key Features

- Converts U.S. dollar amounts into American English.
- Supports positive and negative amounts.
- Supports values through billions.
- Uses an 8-byte integer for the dollar portion.
- Writes the complete dollar portion in words.
- Displays cents numerically using two digits.
- Applies correct singular and plural descriptors.
- Handles cents that round to 100.
- Uses no hyphens in compound numbers.
- Uses "and" only between dollars and cents.
- Returns an empty string for unsupported values.

Syntax

sAmountInWords is string = FT_AmountToWords(nAmount)

Parameters

Parameter Description

nAmount Currency amount to convert into American English words.

Return Value

Result	**Description**
Formatted string	Currency amount expressed using words for dollars and two numeric digits for cents.
Empty string	The supplied amount falls outside the supported range.

Supported Range

The procedure supports amounts from -999,999,999,999.99 through 999,999,999,999.99; Values outside that range return: "". The upper supported dollar amount is just under one trillion dollars.

Remarks

American English Formatting

The procedure uses American English number construction. For example: **411.23** returns: **Four Hundred Eleven Dollars and 23 Cents**. It does not insert "and" within the whole-number portion. Therefore, the result is: **Four Hundred Eleven** rather than: **Four Hundred and Eleven**. The word "and" is reserved exclusively for separating the dollar and cent portions.

Dollar and Cent Separation

The absolute amount is divided into:

- A whole-dollar portion.
- A fractional cent portion.

The whole-dollar amount is obtained using: nDollars = IntegerPart(nAbsoluteAmount). The cents are calculated using: nCents = Round((nAbsoluteAmount - nDollars) * 100, 0). This ensures that the fractional portion is expressed as the nearest whole cent.

Whole-Number Conversion

The dollar portion is converted into words through the internal FunctionTrak helper: FT_InternalWholeNumberToWords(). That helper provides the wording for values such as:

- 0
- 42
- 411
- 1250000

The public procedure then adds the appropriate currency descriptors and cent formatting.

Zero Values

A zero amount returns: Zero Dollars and 00 Cents. Example:
sResult = FT_AmountToWords(0)
Result: Zero Dollars and 00 Cents

The plural forms are used because neither the dollar nor cent amount equals exactly one.

Singular and Plural Dollars
The singular descriptor is used only when the dollar portion equals exactly one.

1.00 returns: One Dollar and 00 Cents. All other dollar values use the plural form: Two Dollars and 00 Cents or One Hundred Dollars and 00 Cents

Singular and Plural Cents
The cent descriptor is singular only when the cent value equals exactly one. 1.01 returns: One Dollar and 01 Cent. Examples using the plural form include:

- One Dollar and 00 Cents
- One Dollar and 02 Cents
- One Dollar and 25 Cents

Two-Digit Cent Formatting
Cents are always displayed using exactly two digits. The procedure uses: NumToString(nCents, "02d")
Examples:

Cent Value	Display
0	00
1	01
5	05
25	25
99	99

This produces consistent financial output.

Negative Amounts
Negative values are converted using their absolute amount, then prefixed with: Negative. For example: **-42.75** returns: **Negative Forty Two Dollars and 75 Cents**

The negative descriptor appears only once, at the beginning of the complete result.

Cents Rounding to the Next Dollar
The procedure explicitly handles cases where the fractional portion rounds to 100 cents. For example, an amount represented internally as: **19.995** may produce: **100** when its fractional portion is multiplied by 100 and rounded. The procedure corrects this by:

1. Increasing the dollar amount by one.
2. Resetting the cents to zero.

The result becomes: **Twenty Dollars and 00 Cents** rather than an invalid result such as: **Nineteen Dollars and 100 Cents**

Compound Number Formatting
Compound numbers are not hyphenated. For example: **42** is expressed as: **Forty Two** not: **Forty-Two**. Similarly: **75** becomes: **Seventy Five**. This style is applied consistently throughout the dollar portion.

The Use of “And”
The word "and" appears only between the dollar and cent portions. Example: **125.45** returns: **One Hundred Twenty Five Dollars and 45 Cents**; it does not return: **One Hundred and Twenty Five Dollars and 45 Cents**. This gives FunctionTrak a consistent American business-document style.

Examples

Amount	**Result**
0.00	Zero Dollars and 00 Cents
1.00	One Dollar and 00 Cents
1.01	One Dollar and 01 Cent
2.05	Two Dollars and 05 Cents
42.75	Forty Two Dollars and 75 Cents
411.23	Four Hundred Eleven Dollars and 23 Cents
-42.75	Negative Forty Two Dollars and 75 Cents

Larger-Value Examples
The procedure supports thousands, millions, and billions through its internal whole-number conversion helper. Examples of the intended format include:

- 1,250.00 becomes One Thousand Two Hundred Fifty Dollars and 00 Cents
- 1,000,001.50 becomes One Million One Dollars and 50 Cents
- 999,999,999,999.99 becomes Nine Hundred Ninety Nine Billion Nine Hundred Ninety Nine Million Nine Hundred Ninety Nine Thousand Nine Hundred Ninety Nine Dollars and 99 Cents

Real-World Uses
FT_AmountToWords() is useful for:

- Printing checks.
- Generating invoices.
- Producing payment receipts.
- Creating purchase orders.
- Preparing contracts.
- Displaying settlement amounts.
- Generating accounting reports.

Range-Validation Example
Always check for an empty result when the value may come from an uncontrolled source:

```
sWrittenAmount is string
sWrittenAmount = FT_AmountToWords(nTransactionAmount)
IF sWrittenAmount = "" THEN
   Error(“The amount falls outside the supported range.")
   RETURN
END
```

Developer Tip
Use the returned empty string as a clear signal that the amount cannot be represented by this procedure.

```
sResult is string
sResult = FT_AmountToWords(nAmount)
IF sResult = "" THEN
   Error("The amount is outside the supported range.")
ELSE
   Info(sResult)
END
```

This is especially important when amounts originate from imported files, database calculations, or user-entered values.

Common Pitfalls
1) The procedure is designed specifically for U.S. currency. It always uses:
- Dollar
- Dollars
- Cent
- Cents

It does not dynamically support:
- Euros.
- Pounds.
- Pesos.
- Canadian dollars.
- Other currencies.

2) The function returns words, not a legally formatted check line. It does not add:
- A currency symbol.
- Asterisks.
- A check-line terminator.
- A fraction such as 23/100.
- Padding to fill a printed field.

Those presentation details remain the caller's responsibility.

3) Cents are displayed numerically rather than written as words. The output is:
Four Hundred Eleven Dollars and 23 Cents
not:
Four Hundred Eleven Dollars and Twenty Three Cents

4) Compound numbers are intentionally not hyphenated. The function returns: **Forty Two** rather than: **Forty-Two**

5) The function uses American placement of "and". It appears only between dollars and cents, not within the whole-number wording.

6) An empty result indicates an unsupported range, not a zero value. Zero returns: Zero Dollars and 00 Cents An empty string means the input exceeded the documented limits.

7) The procedure does not apply currency formatting to the original numeric value. It does not return: $411.23. It returns only the written description.

Related Procedures

Procedure	**Why You Might Use It**
FT_CalculatePercentageOf()	Determines a percentage amount from a supplied base value.
FT_CalculatePercentage()	Determines what percentage one amount represents of another.
FT_CalculateDiscount()	Calculates a discount amount or discounted value.
FT_InternalWholeNumberToWords()	Internal helper used to convert the whole-dollar portion into words.

FT_CalculatePercentageOf

Calculate a specified percentage of a monetary amount and return the result rounded to standard currency precision.

Purpose

Calculates a percentage of a supplied monetary amount. The result is rounded to two decimal places, making it suitable for financial calculations such as:

- Sales tax.
- Discounts.
- Credit card processing fees.
- Commissions.
- Gratuities.
- Service charges.
- Insurance premiums.

The procedure performs only the percentage calculation. It does not format or display the result.

Why This Procedure?

Percentage calculations appear throughout business applications. Although the mathematical formula is simple: Amount × Percentage ÷ 100, financial applications also require consistent currency rounding. Without rounding, calculations may produce values such as: 7.499999999 or 3.14567891 that are inappropriate for invoices, receipts, and payment calculations.

FT_CalculatePercentageOf() performs both the calculation and the required currency rounding in one reusable procedure.

Key Features

- Calculates a percentage of a monetary amount.
- Supports positive values.
- Supports zero values.
- Supports negative values.
- Supports percentages greater than 100%.
- Supports fractional percentages.
- Returns a Currency value.
- Rounds to two decimal places.
- Uses standard financial rounding.
- Performs no formatting.
- Performs no validation of business rules.

Syntax

nResult is currency = FT_CalculatePercentageOf(nAmount,nPercentage)

Parameters

Parameter	Description
nAmount	Monetary amount used as the basis for the calculation.
nPercentage	Percentage rate to apply. Fractional values such as 7.625 are supported.

Return Value

Type	Description
Currency	Calculated percentage amount rounded to two decimal places.

Remarks

Calculation Formula

The calculation performed is: Amount × Percentage ÷ 100

For example: 100.00 × 8.25% returns: 8.25

Currency Rounding

After the calculation, the result is rounded to two decimal places. Internally, the procedure performs: Round(nResult, 2). This ensures that the returned value follows normal financial precision.

Fractional Percentages

Fractional percentage rates are fully supported. For example: 7.625% is calculated correctly.

This is particularly useful for:

- State and local sales taxes.
- Variable processing fees.
- Interest calculations.
- Commission rates.

Percentages Greater Than 100%

The procedure does not restrict the percentage. For example: 150% returns one and one-half times the original amount. Example: 200.00 at: 150% returns: 300.00

Negative Values

Negative amounts are permitted. For example: -100.00 at: 10% returns: -10.00. Negative percentages are also supported. For example: 100.00 at: -15% returns: -15.00. The procedure performs the mathematical calculation exactly as supplied.

Examples

Amount	Percentage	Result
100.00	5	5.00
100.00	7.5	7.50
250.00	8.25	20.63
411.23	7.625	31.35
100.00	150	150.00

Amount Percentage Result

-50.00 10 -5.00

Real-World Uses

FT_CalculatePercentageOf() is useful for:

- Calculating sales tax.
- Determining gratuities.
- Computing discounts.
- Credit card processing fees.
- Insurance premiums.
- Payroll deductions.
- Sales commissions.
- Investment gains.
- Administrative fees.
- Revenue sharing.

Practical Example

Calculate New Mexico sales tax.

```
nSubtotal  is currency = 89.95
nSalesTax  is currency
nSalesTax = FT_CalculatePercentageOf(nSubtotal,7.625)
Result: 6.86
```

Credit Card Fee Example

Calculate a processing fee.

```
nPurchaseAmount is currency = 125.00
nFee is currency
nFee = FT_CalculatePercentageOf(nPurchaseAmount,3.5)
Result: 4.38
```

Tip Calculation Example

Calculate an 18% gratuity.

```
nMealTotal is currency = 62.80
nTip is currency
nTip = FT_CalculatePercentageOf(nMealTotal,18)
Result: 11.30
```

Developer Tip

TIP: This procedure returns only the percentage amount—not the final total.
For example: Subtotal: 100.00, Tax: 8.25
The grand total must still be calculated: nTotal = nSubtotal + nSalesTax

Keeping these calculations separate gives the caller complete control over how the values are displayed and combined.

Common Pitfalls

1) The percentage should be supplied as a percentage, **not** a decimal fraction.
Correct: 7.5 Incorrect:0.075. Supplying 0.075 calculates **0.075%**, not **7.5%**.

2) The procedure returns the percentage amount only. It does **not** return: Amount + Percentage

3) Business-rule validation is intentionally omitted. The procedure permits values such as:

- Negative percentages.
- Percentages greater than 100%.
- Zero percentages.

Whether those values are appropriate depends entirely on the application.

4) The procedure always rounds to two decimal places. If an application requires additional precision—for example, tax accumulation before final rounding—a different calculation strategy may be appropriate.

5) The procedure performs no formatting. The returned value is a Currency variable suitable for further calculations. Formatting as: $31.35 or 31.35% is the responsibility of the caller.

Related Procedures

Procedure	Why You Might Use It
FT_CalculatePercentage()	Determines what percentage one amount represents of another.
FT_CalculateDiscount()	Calculates discounted values from a percentage rate.
FT_AmountToWords()	Converts a monetary amount into American English words.

FT_CalculatePercentage

Determine what percentage one monetary amount represents of another, returning the result rounded to two decimal places.

Purpose

Calculates the percentage that one amount represents of another amount. The result is rounded to two decimal places, making it suitable for financial reporting, analytics, progress calculations, commissions, discounts, and other business applications. The procedure safely handles division-by-zero by returning zero when the total amount is zero.

Why This Procedure?

Business applications frequently need to determine a percentage from two known values. Examples include:

- What percentage of the invoice has been paid?
- What percentage discount was applied?
- What percentage of a budget has been spent?
- What percentage of sales came from a particular product?
- What percentage of an investment has been recovered?

Although the mathematics is straightforward: Partial Amount ÷ Total Amount × 100
a reusable function provides:

- Consistent calculations.
- Standardized rounding.
- Division-by-zero protection.
- Improved readability.

Key Features

- Calculates percentages from two amounts.
- Returns a Real value.
- Rounds to two decimal places.
- Protects against division by zero.
- Supports positive values.
- Supports negative values.
- Supports percentages greater than 100%.
- Performs no formatting.
- Performs no business-rule validation.

Syntax

nPercentage is real = FT_CalculatePercentage(nPartialAmount,nTotalAmount)

Parameters

Parameter	Description
nPartialAmount	Portion of the total amount.
nTotalAmount	Total amount used as the basis for the calculation.

Return Value

Type	Description
Real	Percentage represented by the partial amount, rounded to two decimal places. Returns 0 when the total amount is zero.

Remarks

Calculation Formula

The procedure performs the following calculation: Partial Amount ÷ Total Amount × 100
For example: 25.00 ÷ 100.00 × 100 returns: 25.00

Currency Precision

After calculating the percentage, the result is rounded to two decimal places. Internally, the procedure performs: Round(nResult, 2). This provides consistent results for financial reporting and presentation.

Division by Zero

Division by zero is explicitly prevented. When the total amount equals zero, the procedure immediately returns: 0

For example:
FT_CalculatePercentage(50.00,0) returns: 0

This avoids runtime errors while providing a predictable result.

Percentages Greater Than 100%

The procedure does not limit the calculated percentage. For example: 150.00 out of: 100.00 returns: 150.00. This behavior is appropriate for many business calculations, including revenue growth and budget overruns.

Negative Values

The procedure performs the mathematical calculation exactly as supplied. Examples include:

Partial	Total	Result
-25	100	-25.00
25	-100	-25.00
-25	-100	25.00

Whether negative values are appropriate depends on the application's business rules.

Examples

Partial Amount	Total Amount	Result
25.00	100.00	25.00
50.00	200.00	25.00
75.00	300.00	25.00
125.00	100.00	125.00
0.00	250.00	0.00
50.00	0.00	0.00

Real-World Uses
FT_CalculatePercentage() is useful for:

- Determining invoice payment percentages.
- Measuring budget utilization.
- Calculating completion percentages.
- Determining discount percentages.
- Sales performance reporting.
- Inventory analysis.
- Revenue contribution calculations.
- Commission reporting.
- Financial dashboards.
- Statistical summaries.

Practical Example
Determine what percentage of an invoice has been paid.

```
nInvoiceTotal  is currency = 875.00
nAmountPaid    is currency = 525.00
nPercentPaid   is real
nPercentPaid = FT_CalculatePercentage(nAmountPaid,nInvoiceTotal) Result: 60.00
```

Budget Example
Determine how much of an annual budget has been used.

```
nBudget      is currency = 150000.00
nSpent       is currency = 42875.00
nUsed        is real
nUsed = FT_CalculatePercentage(nSpent,nBudget) Result: 28.58
```

Discount Analysis Example
Determine the percentage discount received.

```
nOriginalPrice   is currency = 199.95
nDiscountAmount  is currency = 30.00
nDiscountPercent is real
nDiscountPercent = FT_CalculatePercentage(nDiscountAmount,nOriginalPrice) Result: 15.00
```

Developer Tip
This procedure complements **FT_CalculatePercentageOf()**. Use **FT_CalculatePercentageOf()** when you know the percentage and need the resulting amount: What is 8.25% of $100.00? Use **FT_CalculatePercentage()** when you know both amounts and need to determine the percentage: $8.25 is what percentage of $100.00? Together, the two procedures eliminate the need to repeatedly write percentage formulas throughout an application.

Common Pitfalls

1) The result is a percentage value—not a decimal fraction. For example: 25 represents: 25% not: 0.25. If a decimal fraction is required, divide the returned value by 100.

2) Returning zero does not always mean the calculated percentage is zero. A return value of zero may also indicate that the total amount supplied was zero and the division was intentionally bypassed. Applications that need to distinguish these situations should test the total amount before calling the procedure.

3) Percentages greater than 100% are valid. For example: 150% simply indicates that the partial amount exceeds the total amount.

4) Negative percentages are permitted. The procedure performs no validation of financial business rules. If negative values are inappropriate for the application, validate the input before calling the procedure.

5) The procedure always rounds to two decimal places.
Applications requiring additional analytical precision should perform their own calculations before applying final presentation rounding.

6) The procedure performs no formatting. The returned value is numeric. Displaying: 28.58% or 28.58 percent is the responsibility of the caller.

Related Procedures

Procedure	Why You Might Use It
FT_CalculatePercentageOf()	Calculates the monetary amount represented by a specified percentage.
FT_CalculateDiscount()	Calculates discounted values from a percentage rate.
FT_AmountToWords()	Converts a currency amount into American English words.

Logic Function Reference

The Logic procedures provide small but frequently used programming utilities that simplify common decision-making and conditional operations. These procedures promote cleaner, more readable code by encapsulating repetitive logic into reusable building blocks.

FT_IIF

Return one of two string values based on the result of a Boolean expression.

Purpose
Evaluates a Boolean condition and returns one of two supplied string values. When the condition evaluates to **True**, the procedure returns the first value. When the condition evaluates to **False**, the procedure returns the second value. The procedure provides a concise alternative to writing a traditional IF...ELSE...END block when selecting between two string values.

Why This Procedure?
Many programming languages include an inline conditional function or operator that simplifies simple decision-making. Without **FT_IIF()**, a developer typically writes:

```
IF bCustomerActive THEN
   sStatus = "Active"
ELSE
   sStatus = "Inactive"
END
```

Using **FT_IIF()**, the same logic becomes:
sStatus = FT_IIF(bCustomerActive, "Active", "Inactive")

This is often easier to read, especially when assigning values within larger expressions or building display strings.

Key Features
- Evaluates a Boolean expression.
- Returns one of two string values.
- Simplifies simple conditional assignments.
- Improves readability.
- Eliminates repetitive IF...ELSE blocks.
- Suitable for display text and report generation.
- Returns either supplied value unchanged.
- Performs no string modification.

Syntax
sResult is string = FT_IIF(bCondition, sTrueValue, sFalseValue)

Parameters

Parameter	Description
bCondition	Boolean expression to evaluate.
sTrueValue	Value returned when the condition evaluates to **True**.
sFalseValue	Value returned when the condition evaluates to **False**.

Return Value

Type Description

String Either **sTrueValue** or **sFalseValue**, depending on the evaluated condition.

Remarks

Boolean Evaluation
The supplied Boolean expression determines which value is returned. If the condition is: True the procedure returns: sTrueValue, Otherwise it returns: sFalseValue. No additional processing is performed.

Returned Values
The selected value is returned exactly as supplied. For example: FT_IIF(True, "Yes", "No") returns: Yes; Likewise: FT_IIF(False, "Yes", "No") returns: No. The procedure does not trim, modify, or translate either value.

Examples

Condition	True Value	False Value	Result
True	"Yes"	"No"	"Yes"
False	"Yes"	"No"	"No"
nCount > 0	"Found"	"Not Found"	Depends on nCount
bRegistered	"Registered"	"Trial"	Depends on bRegistered

Real-World Uses
FT_IIF() is useful for:

- Displaying status text.
- Building report values.
- Selecting captions.
- Choosing button text.
- Displaying Yes/No values.
- Selecting localized strings.
- Building log messages.
- Choosing filenames.
- Creating summary text.
- Simplifying assignments.

Practical Example
Display a customer's account status.

```
sStatus is string
sStatus = FT_IIF(Customer.Active, "Active", "Inactive")
If the customer is active: Active Otherwise: Inactive
```

Report Example
Display whether an invoice has been paid.

```
STC_PaymentStatus = FT_IIF(Invoice.Balance = 0, "Paid", "Outstanding")
```

Developer Tip
Reserve **FT_IIF()** for simple value selection. If each outcome requires multiple statements or significant processing, a traditional IF...ELSE...END block is usually clearer. For example, this is an excellent use:

```
BTN_Save..Caption = FT_IIF(bModified, "Save", "Close")
```

Whereas this is better written as a normal conditional:

```
IF bModified THEN
   SaveRecord()
   RefreshWindow()
   LogChange()
ELSE
   Close()
END
```

Common Pitfalls
1) This version of **FT_IIF()** returns **strings only**. It is not intended for:
   - Numeric values.
   - Currency values.
   - Dates.
   - Booleans.
   - Objects.

If another data type is required, create an overload appropriate for that type.

2) The function returns one of the supplied values exactly as provided. It performs no:
   - Formatting.
   - Translation.
   - Validation.
   - Capitalization.
   - Trimming.

3) Complex logic inside the condition can reduce readability. When the condition becomes difficult to understand, calculate it first:

```
bEligible = Customer.Balance <= 0 AND Customer.Active
sStatus = FT_IIF( bEligible, "Approved", "Pending")
```
This is generally easier to read than embedding a lengthy expression directly in the function call.

FT_IsBetween

Determine whether a value falls within a specified range, regardless of the order in which the range limits are supplied.

Purpose
Determines whether a value falls within an inclusive range. The procedure automatically normalizes the supplied minimum and maximum values, allowing the limits to be specified in either ascending or descending order. The comparison is inclusive, meaning the minimum and maximum values themselves are considered valid results.

Why This Procedure?
Testing whether a value falls within a range is a common programming task. Without **FT_IsBetween()**, developers typically write:

IF nValue >= nMinimum AND nValue <= nMaximum THEN

This assumes the limits are always supplied correctly.

If they are accidentally reversed:
Minimum = 100
Maximum = 50
the comparison fails unexpectedly.

FT_IsBetween() automatically corrects the range before performing the comparison, eliminating an entire class of programming errors.

Key Features
- Tests whether a value falls within a range.
- Automatically normalizes the range limits.
- Supports ascending or descending limits.
- Performs an inclusive comparison.
- Returns a Boolean result.
- Uses Variant parameters for flexibility.
- Eliminates repetitive comparison code.
- Improves readability.

Syntax
bResult is boolean = FT_IsBetween(vValue, vMinimum, vMaximum)

Parameters

Parameter	Description
vValue	Value being tested.
vMinimum	One limit of the range.
vMaximum	The other limit of the range.

Return Value

Type	Description
Boolean	**True** if the value falls within the inclusive range; otherwise **False**.

Remarks

Inclusive Comparison

The comparison includes both limits. Internally, the procedure evaluates:
Value ≥ Lower Limit AND Value ≤ Upper Limit

Therefore:
FT_IsBetween(10, 10, 20) returns: True

Likewise:
FT_IsBetween(20, 10, 20) also returns: True

Only values outside the range return **False**.

Automatic Range Normalization

The procedure determines which supplied value is lower and which is higher before performing the comparison. For example: FT_IsBetween(75, 100, 50) internally becomes: 75 between 50 and 100

The caller does not need to sort or validate the limits beforehand.

Variant Parameters

All three parameters are declared as **Variant** values. This allows the same procedure to work with many comparable data types, including:

- Integers.
- Real numbers.
- Currency values.
- Dates.
- Times.
- DateTime values.
- Strings (using the language's normal comparison rules).

The supplied values should all be compatible with one another.

Examples

Call	Result
FT_IsBetween(25, 10, 50)	True
FT_IsBetween(10, 10, 50)	True
FT_IsBetween(50, 10, 50)	True
FT_IsBetween(75, 10, 50)	False
FT_IsBetween(75, 100, 50)	True
FT_IsBetween(-5, -10, 10)	True

Date Example
Determine whether today's date falls within a reporting period.

```
IF FT_IsBetween(DateSys(),dStartDate,dEndDate) THEN
   Info("Reporting period is active.")
END
```

The order of the supplied dates does not matter.

Currency Example
Determine whether an invoice amount falls within an approval range.

```
IF FT_IsBetween(Invoice.Total,100.00,500.00) THEN
   ProcessAutomatically()
END
```

String Example
Because the parameters are Variants, strings may also be compared.

FT_IsBetween("M", "A", "Z") returns: True

using the normal string comparison rules of the language.

Real-World Uses
FT_IsBetween() is useful for:

- Validating numeric ranges.
- Date-range testing.
- Time-range comparisons.
- Currency validation.
- Temperature limits.
- Inventory thresholds.
- Age validation.
- Score evaluation.
- Report filtering.
- General business-rule testing.

Practical Example
Determine whether a customer's balance qualifies for a discount.

```
IF FT_IsBetween( Customer.Balance, 500.00, 1000.00) THEN
   ApplyPreferredDiscount()
END
```

Because the limits are normalized automatically, the following produces the same result:

```
FT_IsBetween(Customer.Balance, 1000.00, 500.00)
```

Developer Tip
Use **FT_IsBetween()** whenever the intent of the code is "within a range." Compare these two approaches:

IF FT_IsBetween(nScore, 70, 100) THEN

versus:

IF nScore >= 70 AND nScore <= 100 THEN

The first version communicates the programmer's intent immediately while eliminating concerns about accidentally reversing the limits.

Common Pitfalls
1) The comparison is **inclusive**. These all return **True**:
FT_IsBetween(10, 10, 20)
FT_IsBetween(20, 10, 20)

If an exclusive comparison is required, use explicit comparison operators instead.

2) The procedure assumes the supplied values are comparable. For example:
- Integer
- Date
- String

should not be mixed in the same call.

All parameters should represent compatible data types.

3) String comparisons follow the language's normal comparison rules.
The procedure performs no special handling for:
- Natural sorting.
- Locale-specific collation.
- Case-insensitive comparisons.

If specialized comparison behavior is required, normalize the values before calling the procedure.

3) Automatic normalization does not validate business rules. For example: FT_IsBetween(50, 100, 10) correctly becomes a comparison between 10 and 100. If reversing the limits indicates a data-entry error in your application, that validation should occur separately.

Related Procedures

Procedure	Why You Might Use It
FT_IIF()	Returns one of two values based on a Boolean result, often using the outcome of FT_IsBetween().

String Function Reference

The String procedures provide practical tools for manipulating, formatting, validating, and transforming text. These procedures address many of the day-to-day string processing tasks encountered in business applications while handling numerous edge cases consistently.

FT_CleanString

Normalize whitespace and remove non-printable control characters while preserving visible text and Unicode characters.

Purpose

Cleans a string by normalizing whitespace and removing unwanted control characters.

The procedure:

- Converts common whitespace characters into ordinary spaces.
- Removes non-printable ASCII control characters.
- Compresses repeated spaces into a single space.
- Removes leading and trailing whitespace.
- Preserves all visible characters, including Unicode characters and accented letters.

The result is a clean, human-readable string suitable for display, storage, searching, and comparison.

Why This Procedure?

User-entered and imported text frequently contains unwanted whitespace and hidden control characters. Examples include:

- Text copied from web pages.
- Data imported from spreadsheets.
- Email content.
- Text files from different operating systems.
- OCR output.
- Clipboard data.

These sources often contain:

- Tabs.
- Carriage returns.
- Line feeds.
- Non-breaking spaces.
- Multiple consecutive spaces.
- Invisible control characters.

Without cleanup, these characters can produce inconsistent searching, formatting, and comparisons.

FT_CleanString() standardizes the text while preserving its readable content.

Key Features

- Converts common whitespace characters into spaces.
- Removes ASCII control characters.
- Removes the DEL character (ASCII 127).
- Converts non-breaking spaces to ordinary spaces.
- Compresses repeated spaces.
- Removes leading spaces.
- Removes trailing spaces.

- Preserves Unicode characters.
- Preserves accented characters.
- Returns an empty string for empty input.

Syntax
sCleanString is Unicode string = FT_CleanString(sInputString)

Parameters

Parameter	Description
sInputString	String to normalize and clean.

Return Value

Type	Description
Unicode string	Cleaned version of the supplied string.

Remarks
Empty Input
If the supplied string is empty, the procedure immediately returns: "". No processing is performed.

Whitespace Normalization Several common whitespace characters are converted into an ordinary space. These include:

Character	ASCII
Horizontal Tab	9
Line Feed	10
Vertical Tab	11
Form Feed	12
Carriage Return	13
Non-breaking Space	160

For example, text conceptually equivalent to:
John<Tab>Smith becomes: John Smith
Likewise: John<NBSP>Smith becomes: John Smith using an ordinary space.

ASCII Control Characters
All remaining ASCII control characters below character code 32 are removed. The procedure also removes: ASCII 127 (DEL). These characters are discarded completely rather than replaced with spaces. This preserves readable content while eliminating invisible characters that can interfere with searching, sorting, comparisons, and display.

Space Compression
Multiple consecutive spaces are compressed into a single space. For example: John Smith becomes: John Smith. This applies equally to spaces produced from converted tabs, line feeds, and non-breaking spaces.

Leading Spaces
Leading whitespace is removed automatically.
Internally, the procedure initializes its space-tracking flag so that any initial spaces are ignored. For example:
“ Customer Name” becomes: Customer Name.

Trailing Spaces
If the final processed character is a space, it is removed before returning the result. For example:
Customer Name becomes: Customer Name

This includes trailing whitespace that originated as tabs, carriage returns, or non-breaking spaces.

Unicode Preservation
Visible Unicode characters are preserved. Examples include:

- Résumé
- Música
- François
- São Paulo

The procedure removes only unwanted whitespace and control characters. It does not strip accents or convert Unicode text to ASCII.

Examples

Input	**Result**
"John Smith"	"John Smith"
" John Smith "	"John Smith"
"John Smith"	"John Smith"
"John<Tab>Smith"	"John Smith"
"John<CR><LF>Smith"	"John Smith"
"Résumé"	"Résumé"
""	""

Real-World Uses
FT_CleanString() is useful for:

- Cleaning imported text.
- Normalizing user input.
- Preparing searchable text.
- Cleaning OCR output.
- Processing clipboard data.
- Preparing database fields.
- Standardizing log entries.
- Cleaning CSV imports.
- Normalizing configuration values.
- Preparing report output.

Practical Example
Normalize a customer name entered through a data-entry form.
Customer.Name = FT_CleanString(EDT_CustomerName)
If the user enters:
  John<Tab><Tab>Smith
the stored value becomes: John Smith

Import Example
Clean imported spreadsheet data.
```
FOR EACH Customer
   Customer.Company = FT_CleanString(Customer.Company)
END
```

This removes hidden formatting characters that often appear in imported data while preserving the visible text.

Developer Tip
Use **FT_CleanString()** as an early step in data processing before performing comparisons or validation. For example:

```
sCustomer = FT_CleanString(EDT_Name)
IF sCustomer = "" THEN
   Error("Please enter a customer name.")
END
```

Cleaning the input first produces more consistent validation and reduces problems caused by hidden whitespace or control characters.

Common Pitfalls
1) The procedure is intended for ordinary text, not formatted documents. Tabs, carriage returns, and line feeds are intentionally converted into spaces. If line structure must be preserved, use a different approach.

2) Only ASCII control characters are removed. Printable Unicode characters—including accented letters and non-Latin scripts—are preserved exactly as supplied.

3) Consecutive spaces are intentionally collapsed. If multiple spaces carry meaning in the source data, this procedure is not appropriate.

4) Leading and trailing whitespace are always removed. This includes whitespace originally represented by:
- Tabs.
- Line feeds.
- Carriage returns.
- Non-breaking spaces.

5) The procedure does not change capitalization. For example:
john smith remains: john smith. If title case or sentence case is required, consider using:

- FT_ProperCase()
- FT_SentenceCase()

after cleaning the text.

Related Procedures

Procedure	**Why You Might Use It**
FT_ProperCase()	Converts cleaned text into title case.
FT_SentenceCase()	Converts cleaned text into sentence case.
FT_RemoveBlankLines()	Removes empty lines while preserving line-oriented text.
FT_WordWrap()	Formats cleaned text into wrapped lines.

FT_PadString

Pad a string to a specified length using a chosen character and alignment.

Purpose

Pads a string to a specified minimum length by adding a designated padding character to the left, right, or both sides of the string. The procedure supports three padding modes:

- Left
- Right
- Center

If the input string already meets or exceeds the requested length, it is returned unchanged.

Why This Procedure?

Fixed-width strings remain common in many business applications.
Examples include:

- Report generation.
- Text-based exports.
- Console applications.
- Log files.
- Legacy data files.
- Column alignment.

Without **FT_PadString()**, developers often write repetitive loops or concatenate multiple spaces or characters manually. This procedure provides a consistent, reusable solution for all common string-padding requirements.

Key Features

- Pads strings to a specified length.
- Supports left, right, and center alignment.
- Uses any single Unicode character as the padding character.
- Defaults to space padding.
- Automatically ignores additional padding characters beyond the first.
- Returns the original string if no padding is required.
- Handles invalid padding modes gracefully.
- Fully supports Unicode strings.

Syntax

sResult is Unicode string = FT_PadString(sInputString,nLength,sPadCharacter,sPadMode)

Parameters

Parameter	Description
sInputString	String to be padded.
nLength	Desired minimum length of the returned string.
sPadCharacter	Character used for padding. If omitted or blank, a space is used. If more than one character is supplied, only the first character is used.

Parameter	Description
sPadMode	Padding direction. Valid values are "PAD_LEFT", "PAD_RIGHT" (default), and "PAD_CENTER". Invalid values default to right padding.

Return Value

Type	Description
Unicode string	The padded string, or the original string if no padding is required.

Remarks

Requested Length

Padding occurs only when the input string is shorter than the requested length. For example:
Input: "ABC"
Requested Length: 5 returns: ABC·· (where "·" represents a space).

If the requested length is: 3 or 2 the original string is returned unchanged.

Padding Character

Any single Unicode character may be used as the padding character. Examples include:
*
0
-
=
•

If an empty string is supplied, the procedure automatically uses a normal space. If multiple characters are supplied: "--" only the first character is used: -

Padding Modes

The procedure supports three alignment modes.

1. **PAD_RIGHT**
   Padding is added after the string.
   Example: FT_PadString("ABC", 6, ".", "PAD_RIGHT") returns: ABC...
2. **PAD_LEFT**
   Padding is added before the string.
   Example: FT_PadString("ABC", 6, "0", "PAD_LEFT") returns: 000ABC
3. **PAD_CENTER**
   Padding is distributed evenly on both sides.
   Example: FT_PadString("ABC", 7, "-", "PAD_CENTER") returns: --ABC--

When an odd number of padding characters is required, the extra character is placed on the right.

For example: FT_PadString("ABC", 8, "-", "PAD_CENTER") returns: --ABC---

Invalid Padding Modes
If an unrecognized padding mode is supplied, the procedure automatically defaults to: PAD_RIGHT
This ensures a predictable result without generating an error.

Unicode Support
Because the procedure operates on Unicode strings, both the input text and the padding character may contain Unicode characters. For example: FT_PadString("Résumé", 12, "•", "PAD_RIGHT") produces a correctly padded Unicode string.

Examples

Call	Result
FT_PadString("ABC", 5)	"ABC "
FT_PadString("ABC", 6, "0", "PAD_LEFT")	"000ABC"
FT_PadString("ABC", 7, "-", "PAD_CENTER")	"--ABC--"
FT_PadString("ABC", 8, "-", "PAD_CENTER")	"--ABC---"
FT_PadString("ABC", 3)	"ABC"
FT_PadString("ABC", 2)	"ABC"

Real-World Uses
FT_PadString() is useful for:
- Formatting fixed-width reports.
- Aligning console output.
- Building text tables.
- Creating separator lines.
- Formatting account numbers.
- Zero-padding numeric strings.
- Creating labels.
- Generating export files.
- Formatting log entries.
- Improving report readability.

Practical Example
Create a fixed-width customer code.
sCustomerCode = FT_PadString(Customer.ID, 8, "0", "PAD_LEFT")
If the customer ID is: 125 the result becomes: 00000125

Report Example
Center a report heading.
STC_Title = FT_PadString("Monthly Sales", 40, " ", "PAD_CENTER")

The title will appear centered within a 40-character field.

Developer Tip
TIP: Use **FT_PadString()** when formatting output for people, not when storing data.
Padding is ideal for:

- Reports.
- Logs.
- Console displays.
- Text exports.

It is generally unnecessary—and often undesirable—to store padded values in a database.

Common Pitfalls
1) The procedure pads strings only. It does not truncate strings. If the input string is already longer than the requested length, it is returned unchanged.

2) Only the first character of the padding string is used. For example: FT_PadString("ABC", 8, "--") pads using: - not: --

3) Invalid padding modes do not generate an error. Any unrecognized value automatically behaves as: PAD_RIGHT. Check for spelling errors if the alignment is not what you expect.

4) Center padding is intentionally asymmetric when an odd number of padding characters is required. The extra character is always placed on the **right**, ensuring consistent and predictable output.

5) The procedure uses character counts, not display widths. Certain Unicode characters or fonts may occupy different visual widths even though each represents a single character.

Related Procedures

Procedure	Why You Might Use It
FT_CleanString()	Normalize text before padding it for display.
FT_WordWrap()	Format longer text into fixed-width lines.
FT_LineCount()	Determine how many lines a padded block of text contains.

FT_LineCount

Return the number of text lines contained within a string, regardless of the operating system's line-ending convention.

Purpose

Determines the number of text lines contained within a string. The procedure recognizes the three major line-ending conventions:

- Windows (CRLF)
- Unix/Linux (LF)
- Classic Macintosh (CR)

This makes it suitable for processing text originating from virtually any operating system.

Why This Procedure?

Counting text lines is a common requirement when processing:

- Text files.
- Imported data.
- Log files.
- User-entered notes.
- Clipboard contents.
- Report output.

Simply counting occurrences of a particular line-ending sequence is often insufficient because text may originate from different operating systems. **FT_LineCount()** automatically recognizes the most common line-ending formats and returns a consistent line count without requiring the caller to normalize the text first.

Key Features

- Counts text lines in Unicode strings.
- Supports Windows CRLF line endings.
- Supports Unix/Linux LF line endings.
- Supports Classic Macintosh CR line endings.
- Correctly treats Windows CRLF as a single line break.
- Returns zero for an empty string.
- Requires no preprocessing.
- Fully Unicode compatible.

Syntax

```
nLines is int = FT_LineCount(sInputString)
```

Parameters

Parameter	Description
sInputString	String whose lines are to be counted.

Return Value

Type	Description
Integer	Number of text lines contained within the string. Returns 0 when the input string is empty.

Remarks

Empty Strings

An empty string contains no text lines. For example: FT_LineCount("") returns: 0

Line Break Recognition

The procedure recognizes all common line-ending formats:

Operating System	Line Ending
Windows	CRLF (Carriage Return + Line Feed)
Unix / Linux	LF (Line Feed)
Classic Macintosh	CR (Carriage Return)

The caller does not need to determine which format is present.

Windows CRLF Handling

Windows stores line endings as two characters: CR + LF. Although this consists of two characters, it represents only **one** line break. The procedure detects this combination and skips the second character to prevent counting the same line break twice.

Line Count Calculation

Internally, the procedure counts line breaks and then returns: Number of Line Breaks + 1. This reflects how text is normally structured. For example:

Line 1
Line 2
Line 3
contains:

- Two line breaks
- Three text lines

Mixed Line Endings

Because each line-ending type is recognized independently, strings containing mixed formats are processed correctly. Although mixed line endings are uncommon, they can occur when text has been copied or merged from multiple operating systems.

Examples

Input	Result
""	0
"Hello"	1
"Line1<CRLF>Line2"	2
"Line1<LF>Line2<LF>Line3"	3
"Line1<CR>Line2"	2

Real-World Uses

FT_LineCount() is useful for:

- Measuring document size.
- Validating text input.
- Processing imported text files.
- Displaying document statistics.
- Calculating report dimensions.
- Preparing wrapped text.
- Parsing log files.
- Processing clipboard text.
- Importing CSV or configuration files.
- Text editor utilities.

Practical Example

Determine whether a user entered too many lines in a comments field.

```
IF FT_LineCount(EDT_Comments) > 20 THEN
   Error("Comments may not exceed 20 lines.")
END
```

Report Example

Display the number of lines in an imported text file.

```
Info("The document contains " + FT_LineCount(sDocument) + " lines.")
```

Developer Tip

Use **FT_LineCount()** whenever line counts matter instead of searching for a specific line-ending sequence. This avoids assumptions about the operating system that created the text and produces consistent results regardless of whether the source originated on Windows, Linux, or another platform.

Common Pitfalls

1) An empty string returns **0**, not **1**. Although the procedure normally returns the number of line breaks plus one, an explicit check is performed first so that an empty string correctly reports zero lines.

2) Windows line endings consist of two characters. The procedure intentionally counts a CRLF sequence as **one** line break rather than two.

3) A string does not need to end with a line break. For example:
Line 1
Line 2

contains two lines even though the final line has no terminating CR, LF, or CRLF.

4) The procedure counts text lines—not paragraphs or wrapped display lines. Automatic word wrapping performed by an edit control or report is not considered. Only actual line-ending characters are counted.

5) The procedure recognizes only the three standard newline conventions. Other Unicode line-separator characters are not interpreted as line breaks.

Related Procedures

Procedure	**Why You Might Use It**
FT_WordWrap()	Wraps long text into multiple lines before counting them.
FT_RemoveBlankLines()	Removes empty lines from text before determining the final line count.
FT_CleanString()	Cleans imported text prior to additional processing.

FT_ProperCase

Convert text to proper case while intelligently handling apostrophes, contractions, possessives, and Irish surnames.

Purpose

Converts text into proper case by capitalizing the first letter of each word while correcting several common capitalization issues that occur in everyday English. Unlike a simple title-case conversion, the procedure intelligently handles:

- Apostrophe contractions.
- Possessives.
- Irish surnames beginning with **O'**.
- Mixed-case input.

The result is more natural-looking text suitable for names, addresses, titles, and general business applications.

Why This Procedure?

Most programming languages provide a basic title-case function. While useful, these functions typically produce results such as:
JOHN O'BRIEN becoming: John O'Brien (which is correct)
but also: DON'T becoming: Don'T
or CUSTOMER'S becoming: Customer'S

These results are grammatically incorrect.

FT_ProperCase() builds upon standard word capitalization by correcting these common situations automatically.

Key Features

- Converts mixed-case text to proper case.
- Corrects apostrophe contractions.
- Corrects possessive endings.
- Preserves Irish surnames beginning with **O'**.
- Handles Unicode strings.
- Returns an empty string for empty input.
- Preserves punctuation.
- Uses WinDev's native capitalization as its foundation.

Syntax

sResult is Unicode string = FT_ProperCase(sInputString)

Parameters

Parameter	Description
sInputString	Text to convert to proper case.

Return Value

Type	Description
Unicode string	The input string converted to proper case.

Remarks

Initial Capitalization

The procedure first converts the entire string to lowercase before applying WinDev's built-in word-capitalization routine. This establishes a consistent starting point regardless of the original capitalization. For example: jOHN sMITH becomes: John Smith before additional refinements are applied.

Apostrophe Handling

After the initial capitalization, the procedure examines every apostrophe within the string. In most situations, the character immediately following the apostrophe is converted to lowercase. This correctly handles common English contractions such as:

Input	Result
DON'T	Don't
CAN'T	Can't
WE'RE	We're
I'M	I'm
IT'S	It's

Possessive forms are also corrected:

Input	Result
CUSTOMER'S	Customer's
EMPLOYEE'S	Employee's

Irish Surnames

Names beginning with **O'** receive special treatment. When the apostrophe immediately follows an initial **O**, the following letter remains uppercase. Examples include:

Input	Result
O'BRIEN	O'Brien
O'DOHERTY	O'Doherty
O'REILLY	O'Reilly
o'connor	O'Connor

This preserves the traditional capitalization of many Irish surnames.

Unicode Support

Because the procedure operates on Unicode strings, accented and international characters are preserved. Examples include:
JOSÉ O'BRIEN becoming: José O'Brien
and
FRANÇOIS D'ARC becoming: François D'arc

(The apostrophe handling follows the procedure's general capitalization rules.)

Empty Strings
If the input string is empty, the procedure immediately returns an empty string.

Examples

Input	Result
john smith	John Smith
JOHN SMITH	John Smith
jOhN sMiTh	John Smith
DON'T STOP	Don't Stop
CUSTOMER'S ORDER	Customer's Order
O'BRIEN	O'Brien
o'doherty	O'Doherty

Real-World Uses
FT_ProperCase() is useful for:

- Customer names.
- Mailing addresses.
- Contact lists.
- Report titles.
- Form input normalization.
- Imported databases.
- Email subject lines.
- Employee records.
- Document headings.
- CRM applications.

Practical Example
Normalize a customer's name before saving it.

```
Customer.FullName = FT_ProperCase(EDT_FullName)
```

If the user enters: jOHN o'brien the stored value becomes: John O'Brien

Import Example
Clean imported customer records.

```
FOR EACH Customer
   Customer.LastName = FT_ProperCase(Customer.LastName)
END
```

This produces consistently capitalized names regardless of the original data source.

Developer Tip
For the best results, consider cleaning imported text before converting it to proper case.
For example: Customer.Name = FT_ProperCase(FT_CleanString(Customer.Name))

Removing unwanted whitespace and control characters first allows **FT_ProperCase()** to focus solely on capitalization.

Common Pitfalls

1) This procedure is designed primarily for English-language text. Although Unicode characters are fully supported, capitalization conventions vary among languages and cultures.

2) Only the **O'** surname pattern receives special handling. Other naming conventions are intentionally left unchanged.

3) The procedure is intended for proper case—not title case. Words are capitalized according to normal sentence and name conventions rather than formal publishing style.

4) Roman numerals are not preserved. For example: HENRY VIII becomes: Henry Viii. This enhancement is identified as a potential future feature.

5) A user-defined capitalization dictionary is not currently implemented. Specialized capitalization rules for organizations, product names, or family names must be handled separately.

Related Procedures

Procedure	**Why You Might Use It**
FT_SentenceCase()	Converts text to sentence case rather than proper case.
FT_CleanString()	Removes unwanted whitespace before capitalization.
FT_TitleCase() *(if implemented in the future)*	Would provide traditional title-style capitalization for headings.

FT_RemoveBlankLines

Remove empty or whitespace-only lines from a text string while preserving the contents of every nonblank line.

Purpose
Removes all blank lines from a supplied text string.

A line is omitted when it contains no visible content after WinDev's NoSpace(..., sscAll) processing. The procedure recognizes text containing:

- Windows CRLF line endings.
- Unix/Linux LF line endings.
- Classic Macintosh CR line endings.

All retained lines are returned using standard Windows CRLF separators.

Why This Procedure?
Blank lines frequently appear in text originating from:

- User-entered notes.
- Clipboard content.
- Imported text files.
- Email messages.
- Generated reports.
- Data copied from websites.
- Files produced on different operating systems.

Removing those lines manually can be cumbersome, particularly when the source text may use several different newline formats. F**T_RemoveBlankLines()** processes each logical line, removes lines without meaningful content, and rebuilds the remaining text in a consistent format.

Key Features

- Removes completely empty lines.
- Removes lines containing only whitespace recognized by NoSpace().
- Preserves nonblank line content.
- Preserves leading and trailing whitespace on retained lines.
- Supports Windows CRLF line endings.
- Supports Unix/Linux LF line endings.
- Supports Classic Macintosh CR line endings.
- Supports mixed line-ending formats.
- Normalizes retained line separators to CRLF.
- Returns an empty string when no nonblank lines remain.
- Fully supports Unicode text.

Syntax
sResult is Unicode string = FT_RemoveBlankLines(sInputString)

Parameters

Parameter	Description
sInputString	Text from which blank or whitespace-only lines will be removed.

Return Value

Type	Description
Unicode string	Text containing only the retained nonblank lines, separated by Windows CRLF line endings. Returns an empty string when the input is empty or contains no nonblank lines.

Remarks

Empty Input

When the supplied string is empty, the procedure immediately returns: "". No additional processing occurs.

Blank-Line Detection

Each completed line is tested using: NoSpace(sCurrentLine, sscAll). If the result is empty, the line is discarded. Conceptually:

Line 1
<blank line>
Line 2
becomes:
Line 1
Line 2

Lines containing only removable whitespace are treated the same way as completely empty lines.

Retained Line Content

The test for blankness does not replace the original line. When a line contains visible content, the procedure appends: sCurrentLine rather than the result returned by NoSpace(). Consequently, leading and trailing whitespace on a retained line remains unchanged. For example, a line conceptually containing:

··Indented text··

where each · represents a space, remains:

··Indented text··

because the line is nonblank.

Supported Line Endings

The procedure recognizes the three principal newline conventions.

Source	Line ending
Windows	CRLF
Unix/Linux	LF
Classic Macintosh	CR

A Windows CRLF pair is processed as one line ending. After encountering the carriage return, the procedure skips its accompanying line-feed character.

Output Line Endings
Regardless of the source format, retained lines are joined using: CRLF. Therefore, Unix/Linux LF or Classic Macintosh CR input is normalized to Windows-style CRLF output.

Mixed Line Endings
Because carriage returns and line feeds are recognized individually, the procedure can also process text containing mixed newline conventions. For example, a string containing both CRLF and LF separators is rebuilt with consistent CRLF separators.

Final Unterminated Line
A final line does not need to end with CR or LF. After scanning the string, the procedure separately examines any characters remaining in sCurrentLine. When that final line is nonblank, it is added to the result.

No Trailing Line Ending
The procedure inserts CRLF only **between** retained lines. It does not append a CRLF sequence after the final retained line.

Examples
In the examples below, <CRLF>, <LF>, and <CR> represent actual line-ending characters.

Input	**Result**
""	""
"Alpha"	"Alpha"
"Alpha<CRLF><CRLF>Beta"	"Alpha<CRLF>Beta"
"Alpha<LF><LF>Beta"	"Alpha<CRLF>Beta"
"Alpha<CR><CR>Beta"	"Alpha<CRLF>Beta"
"<CRLF>Alpha<CRLF>"	"Alpha"
" <CRLF>Alpha"	"Alpha"
"<CRLF><CRLF>"	""

Real-World Uses
FT_RemoveBlankLines() is useful for:

- Cleaning imported text.
- Compacting customer notes.
- Removing empty rows from generated text.
- Preparing email content.
- Cleaning copied website text.
- Processing log excerpts.
- Normalizing configuration text.
- Preparing report output.
- Cleaning address blocks.
- Standardizing multiline database fields.

Developer Tip
Use **FT_RemoveBlankLines()** when line structure must remain intact. Do not substitute **FT_CleanString()** when the output should remain multiline. For example:
sResult = FT_RemoveBlankLines(sText)
removes blank lines while retaining meaningful line breaks.

By contrast:
sResult = FT_CleanString(sText)
converts recognized newline characters into spaces and produces a single normalized line.

Common Pitfalls
1) The output always uses Windows CRLF line endings. Even when the source uses LF or CR, retained lines are rebuilt using: CRLF. Applications that must preserve the original newline convention will need a different approach.

2) Nonblank lines are not trimmed. Whitespace is removed temporarily only to determine whether a line is blank. The original nonblank line is retained unchanged. For example:
··Customer name··
remains padded with those spaces.

3) The procedure removes blank lines, not duplicate lines. Repeated nonblank lines remain in the result.

4) Visual wrapping does not create logical lines. Text that wraps automatically inside an edit control still represents one line unless actual CR or LF characters are present.

5) The result does not end with CRLF. Even when the source contains a terminating line ending, the returned text ends with the final retained character.

6) A source containing only blank lines returns an empty string. It does not return a single line ending or preserve the original number of empty lines.

Related Procedures

Procedure	Why You Might Use It
FT_LineCount()	Counts the logical lines in a string before or after blank lines are removed.
FT_CleanString()	Converts multiline text into normalized single-line text.
FT_WordWrap()	Divides long text into lines of a specified width.

FT_SentenceCase

Convert text to sentence case by capitalizing the first alphabetic character of the text and the first alphabetic character following ., !, or ?.

Purpose

Converts a Unicode string into sentence case. The procedure first converts the entire string to lowercase. It then capitalizes:

- The first ASCII alphabetic character in the text.
- The first ASCII alphabetic character after a period.
- The first ASCII alphabetic character after an exclamation mark.
- The first ASCII alphabetic character after a question mark.
- The standalone lowercase pronoun i in several common positions.

The original punctuation and spacing are otherwise preserved.

Why This Procedure?

Text entered by users or imported from external sources often arrives in inconsistent capitalization. Examples include:
THIS IS A SENTENCE.
tHIS iS a SENTENCE.
this is a sentence. this is another one.

A basic lowercase conversion removes the inconsistency, but it also removes the capitalization needed at the beginning of each sentence.

FT_SentenceCase() establishes a consistent lowercase baseline and then restores sentence-opening capitalization.

Key Features

- Converts mixed-case text to a lowercase baseline.
- Capitalizes the first alphabetic character.
- Capitalizes after ., !, and ?.
- Skips spaces and punctuation while waiting for the next letter.
- Corrects the standalone English pronoun i in common positions.
- Preserves existing punctuation.
- Preserves existing spacing and line endings.
- Supports Unicode strings.
- Returns an empty string for empty input.

Syntax

sResult is Unicode string = FT_SentenceCase(sInputString)

Parameters

Parameter	Description
sInputString	Text to convert to sentence case.

Return Value

Type	Description
Unicode string	The supplied text converted according to the procedure's sentence-case rules.

Remarks

Lowercase Baseline

The procedure begins by converting the entire source string to lowercase: sInputString = Lower(sInputString)
This ensures that inconsistent capitalization is removed before sentence capitalization is applied. For example:
tHIS IS a TEST.
first becomes:
this is a test.
and is then returned as:
This is a test.

First Alphabetic Character

The flag bCapitalizeNext begins as True. The procedure scans forward until it encounters a character between:
a
and:
z

That character is converted to uppercase. This means leading spaces, digits, quotation marks, and other punctuation may appear before the first letter without preventing capitalization. For example:
  "hello there."
becomes:
  "Hello there."

Sentence Terminators

The procedure treats the following characters as sentence endings:
.
!
?

After encountering one of these characters, it waits for the next lowercase ASCII letter and capitalizes it. For example:
hello. how are you? i am fine!
becomes:
Hello. How are you? I am fine!

Spaces After Punctuation

The next letter does not have to immediately follow the punctuation mark.
For example: **hello. how are you?** becomes: Hello. **How are you?** The original spacing is preserved.

Punctuation After a Sentence Terminator

The procedure continues waiting until it encounters an ASCII lowercase letter.
For example: **hello! "welcome."** becomes: **Hello! "Welcome."**
The quotation mark is retained, and the following letter is capitalized.

Standalone Pronoun "I"
After the main sentence-case conversion, the procedure performs three additional corrections for the standalone lowercase pronoun i. It corrects i when it appears:

- Between ordinary spaces.
- At the beginning followed by a space.
- At the end preceded by a space.

Examples:

Input	Result
you and i agree.	You and I agree.
i agree.	I agree.
you agree and i	You agree and I

Empty Input
When the supplied value is empty, the procedure immediately returns: ""

Examples

Input	Result
hello world.	Hello world.
HELLO WORLD.	Hello world.
hELLO wORLD.	Hello world.
hello. how are you?	Hello. How are you?
stop! do not move.	Stop! Do not move.
you and i agree.	You and I agree.
123 apples are here.	123 Apples are here.
hello.	Hello.
""	""

Real-World Uses
FT_SentenceCase() is useful for:

- Normalizing customer notes.
- Cleaning imported descriptions.
- Formatting comments.
- Standardizing email text.
- Correcting all-uppercase data.
- Preparing report narratives.
- Cleaning survey responses.
- Formatting help text.
- Normalizing database memo fields.
- Improving user-entered prose.

Developer Tip
Clean unwanted control characters before applying sentence case when processing imported text. sText = FT_SentenceCase(FT_CleanString(sText)). Be aware that **FT_CleanString()** converts line breaks to spaces. When multiline structure must remain intact, apply only the cleanup appropriate to the source.

Common Pitfalls
1) The procedure lowers every character before rebuilding sentence capitalization. Existing intentional capitalization is not preserved. For example: NASA launched the Artemis mission. becomes: Nasa launched the artemis mission.

Acronyms, product names, and specialized capitalization require separate correction.

2) Periods inside abbreviations are treated as sentence endings. For example:
dr. smith arrived. becomes: **Dr. Smith arrived.**
That particular result may look acceptable, but other abbreviations can cause unintended capitalization:
the u.s. economy may become: **The u.S. Economy**

Version 1 does not attempt to identify abbreviations.

3) Alphabetic detection is limited to ASCII lowercase letters a through z. The procedure preserves Unicode text, but the capitalization trigger explicitly tests: sCharacter >= "a" AND sCharacter <= "z"
Therefore, an accented or non-Latin letter at the beginning of a sentence may not be recognized as the character to capitalize. For example, a sentence beginning with a lowercase accented character may remain lowercase unless another ASCII letter follows.

4) The pronoun correction recognizes only ordinary-space boundaries. The replacement:
Replace(sResultString, " i ", " I ")
does not correct every grammatically standalone i. Examples that may remain lowercase include:
i,
(i)
"i"
i'm
i'll

The beginning and ending checks also require an ordinary adjacent space.

5) Apostrophe contractions are not specially corrected. Because the entire source is lowered first, contractions such as:
I'M
I'LL
I'D
become:
i'm
i'll
i'd

unless the i happens to match one of the limited standalone-pronoun corrections.

6) A period used as a decimal point is treated as a sentence ending. For example:
the value is 3.14 dollars.
causes the procedure to enter capitalization mode after the decimal point. It will capitalize the next ASCII letter encountered, which may produce an unintended result later in the text.

7) Line endings are preserved, but they do not independently begin a new sentence. Only ., !, and ? set bCapitalizeNext to True. A new line without sentence-ending punctuation does not automatically cause the next line's first letter to be capitalized.

Related Procedures

Procedure	**Why You Might Use It**
FT_ProperCase()	Capitalizes each word while correcting apostrophes and O' surnames.
FT_CleanString()	Normalizes whitespace and removes ASCII control characters.
FT_RemoveBlankLines()	Removes blank lines while preserving meaningful multiline text.
FT_LineCount()	Counts the logical lines contained in a string.

FT_Slugify

Convert text into a lowercase, URL-friendly slug using letters, numbers, Unicode characters, and single hyphen separators.

Purpose

Converts a Unicode string into a compact slug suitable for use in:

- URLs.
- Web page routes.
- Article identifiers.
- Folder or document labels.
- Search-friendly names.
- Human-readable keys.

The procedure converts the input to lowercase, replaces common separators with hyphens, removes decorative punctuation, and eliminates duplicate, leading, and trailing hyphens.

Why This Procedure?

Application data often needs a simplified identifier derived from readable text.
For example: **FunctionTrak User Manual** may need to become: **functiontrak-user-manual**. Similarly: **What's New in Version 2.0?** may need to become: **whats-new-in-version-20**. Writing this conversion repeatedly throughout an application can lead to inconsistent results. **FT_Slugify()** provides a single, predictable rule set for converting ordinary text into clean slug values.

Key Features

- Converts text to lowercase.
- Preserves ASCII letters.
- Preserves digits.
- Preserves Unicode characters above ASCII 127.
- Converts common separators to hyphens.
- Compresses repeated separators into one hyphen.
- Prevents leading hyphens.
- Removes trailing hyphens.
- Removes decorative punctuation.
- Returns an empty string for empty input.
- Fully supports Unicode strings.

Syntax

sSlug is Unicode string = FT_Slugify(sInputString)

Parameters

Parameter	Description
sInputString	Text to convert into a slug.

Return Value

Type	Description
Unicode string	Lowercase slug containing retained letters, numbers, Unicode characters, and single hyphen separators.

Remarks

Lowercase Conversion

The entire input is converted to lowercase before any other processing occurs. For example: **FunctionTrak User Manual** becomes: **functiontrak user manual** before the spaces are replaced with hyphens. The final result is: **functiontrak-user-manual**

Preserved Characters

The procedure retains:

- ASCII letters a through z.
- Digits 0 through 9.
- Characters whose Asc() value is greater than 127.

This means accented and other Unicode characters are preserved rather than transliterated or removed. Examples:

Input	Result
Café Menu	café-menu
Résumé Builder	résumé-builder
Música Latina	música-latina
Über Alles	über-alles

Separator Characters

The following characters are treated as separators:

- Spaces and ASCII control whitespace.
- Forward slash /.
- Backslash \.
- Underscore _.
- Ampersand &.
- Plus sign +.
- Colon :.
- Hyphen -.

Each separator group becomes a single hyphen.
For example: **File / Folder \ Path** becomes: **file-folder-path**
Likewise: **rock & roll + blues** becomes: **rock-roll-blues**

Duplicate Hyphens

Repeated separators never produce repeated hyphens. For example:
alpha / beta___gamma becomes: **alpha-beta-gamma**. The Boolean variable bLastWasHyphen prevents an additional hyphen from being added when the previous output character was already a hyphen.

Leading Hyphens
A separator is appended only when the result already contains text. Therefore:
--- Example becomes: **example** rather than: **---example**

Trailing Hyphens
Any trailing hyphen is removed before the result is returned. For example:
Example --- becomes: **example**

Decorative Punctuation
Characters that are neither retained characters nor recognized separators are discarded. Examples include:

- Periods.
- Commas.
- Apostrophes.
- Quotation marks.
- Parentheses.
- Brackets.
- Exclamation marks.
- Question marks.
- Semicolons.

For example:
What's New? becomes: **whats-new**. The apostrophe and question mark are removed.

Empty Input
When the input string is empty, the procedure immediately returns: ""
Examples

Input	**Result**
FunctionTrak	functiontrak
FunctionTrak User Manual	functiontrak-user-manual
What's New?	whats-new
Version 2.0	version-20
File/Folder\Path	file-folder-path
Rock & Roll	rock-roll
C++ Programming	c-programming
Customer___Record	customer-record
---Hello---	hello
Café Menu	café-menu
""	""

Real-World Uses
FT_Slugify() is useful for:

- Generating page slugs.
- Creating readable URL segments.
- Naming exported documents.

- Creating article identifiers.
- Building blog post routes.
- Producing search-friendly labels.
- Creating category keys.
- Building folder names.
- Generating media identifiers.
- Creating internal aliases from titles.

Developer Tip
Use the slug as a readable identifier, but do not assume it is unique. Different source strings may produce the same result. For example: Version 2.0 and: Version 20 both become: version-20

When uniqueness matters, append an ID, date, hash, or generated suffix.
sSlug = FT_Slugify(Article.Title) + "-" + Article.ID

Common Pitfalls
1) The procedure preserves Unicode characters. This is intentional, but it means the result is not limited to ASCII. Applications requiring ASCII-only URLs will need a separate transliteration step.

2) Decorative punctuation is removed rather than converted to separators.
For example: **one.two** becomes: **onetwo** because the period is discarded. It does not become: **one-two.**
Likewise: **can't** becomes: **cant**

3) Decimal points are removed. For example: **Version 2.0** becomes: **version-20**. This may create ambiguity between 2.0 and 20.

4) Plus signs are treated as separators. For example: **C++** becomes: **c** because the plus signs would create a trailing separator that is later removed.

5) Ampersands are not converted to the word “and.” For example: **Research & Development** becomes: **research-development** not: **research-and-development**

6) Unicode preservation is based on character codes above 127. That preserves accented letters, but it also preserves many non-letter Unicode symbols. The implementation does not specifically determine whether a Unicode character is alphabetic.

7) The procedure does not enforce a maximum length. A very long source string may produce a very long slug. Applications using slugs in databases, URLs, or files should enforce their own length limits.

8) A slug is not automatically safe for every filesystem or web framework. Although the output is substantially simplified, the preserved Unicode characters may still require encoding or additional validation depending on how the slug will be used.

Related Procedures

Procedure	Why You Might Use It
FT_SafeFileName()	Produces a Windows-safe filename rather than a web-style slug.
FT_SafeFolderName()	Produces a readable Windows-safe folder name.
FT_CleanString()	Normalizes whitespace and removes control characters without converting text into a slug.
FT_ProperCase()	Converts text for human-readable display rather than machine-oriented identifiers.

FT_SingleBlankLines

Collapse consecutive blank lines into one blank line while preserving every nonblank line exactly as supplied.

Purpose

Processes multiline text so that no more than one blank line appears between nonblank lines.

The procedure:

- Recognizes empty lines.
- Recognizes lines containing only spaces or TAB characters.
- Collapses multiple consecutive blank lines into one.
- Removes blank lines from the beginning and end of the text.
- Preserves the exact contents of every retained nonblank line.
- Accepts Windows, Unix/Linux, and classic Macintosh line endings.
- Rebuilds the result using WinDev's standard CR line separator.

Why This Procedure?

Multiline text often accumulates excessive blank lines when it is:

- Copied from websites.
- Pasted from email.
- Imported from documents.
- Combined from several sources.
- Entered manually in memo fields.
- Generated through repeated editing.

Removing all blank lines may make the text too dense because meaningful paragraph separation disappears.

Key Features

- Collapses consecutive blank lines into one blank line.
- Treats empty lines as blank.
- Treats space-only lines as blank.
- Treats TAB-only lines as blank.
- Treats lines containing spaces and TABs as blank.
- Preserves nonblank lines exactly.
- Preserves indentation on nonblank lines.
- Removes leading blank lines.
- Removes trailing blank lines.
- Supports Windows CRLF line endings.
- Supports Unix/Linux LF line endings.
- Supports classic Macintosh CR line endings.
- Supports mixed line-ending conventions.
- Normalizes output line endings to WinDev's CR constant.
- Fully supports Unicode strings.

Syntax

sResult is Unicode string = FT_SingleBlankLines(sInputString)

Parameters

Parameter	Description
sInputString	Multiline text whose consecutive blank lines are to be collapsed.

Return Value

Type	Description
Unicode string	Processed text containing no more than one blank line between nonblank lines. Leading and trailing blank lines are removed.

If the input is empty or contains no nonblank lines, the procedure returns an empty string.

Remarks

Empty Input

When the supplied string is empty, the procedure immediately returns: ""

Blank-Line Detection

Each completed line is copied into a temporary value used only to determine whether the line contains visible content. The procedure first removes TAB characters: sTestLine = Replace(sCurrentLine, TAB, ""). It then removes spaces from the temporary value: sTestLine = NoSpace(sTestLine, sscAll). When nothing remains, the line is considered blank. The following are therefore treated as blank lines:

- <empty line>
- <spaces only>
- <TABs only>
- <spaces and TABs>

Preserving Nonblank Lines

The temporary sTestLine value is used only for blank-line detection. When a line contains visible text, the procedure appends the original: sCurrentLine. This preserves the retained line exactly as it appeared in the input. For example, a line containing four leading spaces:

    Indented detail

retains all four spaces.

Trailing spaces on a nonblank line are also retained.

Pending Blank-Line Design

A blank line is not written to the result immediately. Instead, the procedure sets:
bBlankLinePending = True

The blank line is written only when another nonblank line follows. This deferred approach accomplishes three things:

1. Multiple blank lines collapse into one.
2. Leading blank lines are ignored.
3. Trailing blank lines are never written.

One Blank Line Between Paragraphs

When a pending blank line exists and another nonblank line is encountered, the procedure adds: CR + CR

The first CR ends the preceding nonblank line. The second creates the one permitted blank line before the next nonblank line. Without a pending blank line, it adds only: CR. This places the next nonblank line directly beneath the previous one.

Leading Blank Lines
Blank lines appearing before the first nonblank line are discarded.

Trailing Blank Lines
Blank lines appearing after the final nonblank line are also discarded.

Supported Input Line Endings
The procedure recognizes the three principal newline conventions:

Source	Line ending
Windows	CRLF
Unix/Linux	LF
Classic Macintosh	CR

A Windows CRLF pair is treated as one logical line ending. After encountering its carriage return, the procedure skips the accompanying line feed.

Mixed Line Endings
The source text may contain more than one line-ending convention. Each CR, LF, or CRLF sequence completes the current logical line, so mixed text can still be processed correctly.

Output Line Endings
The original line-ending convention is not preserved. All retained lines are rebuilt using WinDev's standard: CR
constant.

Final Unterminated Line
The final line does not need to end with CR or LF. After the main scanning loop finishes, the procedure separately examines any remaining characters in sCurrentLine. If that final line contains visible text, it is added to the result.

No Final Line Ending
The result ends with the final retained character. No CR is appended after the last nonblank line.

Examples
In the following examples, <CR> represents the output line separator and <BLANK> represents one empty line.

Input	Result
""	""
"Alpha"	"Alpha"
"Alpha<CRLF>Beta"	"Alpha<CR>Beta"
"Alpha<CRLF><CRLF>Beta"	"Alpha<CR><CR>Beta"
"Alpha<LF><LF><LF>Beta"	"Alpha<CR><CR>Beta"

Input	Result
"<CRLF><CRLF>Alpha"	"Alpha"
"Alpha<CRLF><CRLF>"	"Alpha"
" <CRLF>Alpha"	"Alpha"
"Alpha<CRLF><TAB><CRLF>Beta"	"Alpha<CR><CR>Beta"
"<CRLF><CRLF>"	""

Real-World Uses
FT_SingleBlankLines() is useful for:

- Cleaning customer notes.
- Normalizing pasted email content.
- Formatting article text.
- Cleaning imported documentation.
- Reducing excessive spacing in reports.
- Preparing multiline database fields.
- Formatting help content.
- Cleaning clipboard text.
- Normalizing correspondence.
- Preserving paragraph structure in generated documents.

Example
Processing Imported Text

```
sImportedText is Unicode string
sImportedText = fLoadText(sFileName)
sImportedText = FT_SingleBlankLines(sImportedText)
```

This is especially useful when the imported file's original operating system or newline format is unknown.

Developer Tip
Choose the procedure according to whether paragraph separation has meaning. Use: FT_RemoveBlankLines(sText) when the output should be compact and every retained line should appear consecutively. Use: FT_SingleBlankLines(sText) when one blank line should remain between paragraphs or logical sections.

Common Pitfalls
1) Nonblank lines are not trimmed. A line is temporarily cleaned only to determine whether it is blank. Its original contents are retained.

2) The procedure does not preserve the original newline format. Windows CRLF, Unix/Linux LF, and classic Macintosh CR input are all rebuilt using WinDev's CR constant.

3) Leading and trailing blank lines are deliberately removed. The procedure permits one blank line only **between** retained nonblank lines.

4) A blank line is not the same as an empty visual display row caused by word wrapping. Only actual CR or LF characters complete a logical line. Automatic wrapping inside an edit control is not examined.

5) The procedure does not remove duplicate nonblank lines. For example:
Alpha
Alpha
remains unchanged.

6) Only TAB characters are explicitly removed before the space test. Other unusual Unicode whitespace characters are not explicitly normalized by this implementation and may cause a visually blank line to be retained.

7) A source containing only blank lines returns an empty string.
It does not return one blank line.

Related Procedures

Procedure	Why You Might Use It
FT_RemoveBlankLines()	Removes all blank lines rather than preserving one.
FT_LineCount()	Counts lines before or after blank-line normalization.
FT_CleanString()	Converts recognized whitespace and line endings into normalized single-line text.
FT_WordWrap()	Introduces logical line endings at a requested text width.

FT_StripHTML

Convert HTML or HTML-like content into readable plain text by removing markup, preserving basic document structure, and decoding common entities.

Purpose

Removes HTML tags from a Unicode string while retaining the readable text contained between those tags. The procedure also:

- Removes HTML comments.
- Converts selected structural tags into line breaks.
- Decodes several common named HTML entities.
- Decodes several common decimal numeric entities.
- Reduces repeated ordinary spaces.
- Collapses excessive blank lines.
- Removes leading and trailing whitespace.

The result is clean plain text suitable for display, storage, searching, reporting, or export.

Why This Procedure?

Simply removing everything between < and > can produce text that is technically tag-free but difficult to read. For example:
First paragraph. Second paragraph.
A basic tag remover might return:
First paragraph.Second paragraph.

FT_StripHTML() recognizes selected structural closing tags and converts them into line breaks:
First paragraph.
Second paragraph.

It also converts encoded text such as: **Tom & Jerry** into: **Tom & Jerry**
This makes the procedure more useful than a simple tag-deletion routine.

Key Features

- Removes ordinary HTML tags.
- Removes HTML comments.
- Handles terminated and unterminated comments.
- Converts
 and
 into line breaks.
- Converts selected closing structural tags into line breaks.
- Decodes common named HTML entities.
- Decodes selected decimal numeric entities.
- Reduces repeated ordinary spaces.
- Collapses excessive blank lines through FT_SingleBlankLines().
- Removes leading and trailing whitespace.
- Preserves Unicode text.
- Returns an empty string for empty input.

Syntax
sPlainText is Unicode string = FT_StripHTML(sInputString)

Parameters

Parameter	Description
sInputString	HTML or HTML-like text to convert into plain text.

Return Value

Type	Description
Unicode string	Plain text with recognized HTML markup removed, selected structural tags converted into line breaks, and supported entities decoded.

An empty input string returns an empty string.

Remarks
Empty Input
When the supplied string is empty, the procedure immediately returns: ""

HTML Comment Removal
Before processing tags, the procedure removes HTML comments enclosed by: <!—
and:
-->
For example:
Before<!-- internal note -->After
becomes:
BeforeAfter
The comment delimiters and all comment content are removed.

Multiple Comments
Comment removal occurs inside a loop, so multiple comments can be removed from the same string.
For example:
A<!-- first -->B<!-- second -->C
becomes:
ABC

Unterminated Comments
When an opening comment marker is found without a matching closing marker, everything from the opening marker to the end of the string is discarded.
For example:
Visible text<!-- unfinished comment
becomes:
Visible text
This prevents unfinished comment content from appearing in the plain-text result.

Ordinary Tag Removal
Characters appearing between < and > are treated as tag content and are not copied to the result.

For example:
The important message
becomes:
The important message
The visible content remains while the and tags are removed.

Structural Tags
Selected tags are converted into line breaks instead of simply disappearing.

Recognized tags include:

Tag	Effect
 	Adds a line break
 	Adds a line break
	Adds a line break
	Adds a line break
	Adds a line break
</tr>	Adds a line break
</h1> through </h6>	Adds a line break

For example:
First line
Second line
becomes:
First line
Second line

Tag Capitalization and Spacing
Tag text is passed through:
Lower(NoSpace(sTag, sscAll))
before comparison.
Consequently, capitalization and spaces inside simple recognized tags do not prevent recognition.
For example, these are treated equivalently:

< br >

After normalization, they match either:
br
or:
br/

Closing Structural Tags
For paragraphs, divisions, list items, table rows, and headings, only the recognized **closing tag** produces a line break.
For example:
Paragraph one Paragraph two
becomes:

Paragraph one
Paragraph two
The opening tags are removed without adding line breaks. The closing tags establish the text boundaries.

Duplicate Structural Line Breaks
Before adding a structural line break, the procedure checks whether the result already ends with CR. This prevents adjacent recognized tags from immediately producing duplicate line endings.
For example:
Text
does not automatically add two consecutive CR characters at that moment.
Final paragraph spacing is subsequently normalized by FT_SingleBlankLines().

Common Named Entities
The procedure decodes the following named HTML entities:

Entity	Result
	Ordinary space
&	&
<	<
>	>
"	"
'	'

For example: **Tom & Jerry** becomes: **Tom & Jerry**

Common Numeric Entities
The procedure also decodes these decimal numeric entities:

Entity	Result
"	"
&	&
'	'
<	<
>	>
	Ordinary space

For example: **"Quoted text"** becomes: **"Quoted text"**

Repeated Spaces
Removing inline tags can leave repeated ordinary spaces.
The procedure repeatedly replaces:
two spaces
with:
one space
until no consecutive pair remains.

For example:
One   two
is normalized to:
One two

This operation affects ordinary space characters throughout the entire result.

Blank-Line Normalization
After tags and entities have been processed, the result is passed to: FT_SingleBlankLines(sResultString)

This ensures that:

- Consecutive blank lines collapse into one blank line.
- Leading blank lines are removed.
- Trailing blank lines are removed.
- Nonblank lines remain intact.
- Output line endings use WinDev's CR constant.

Final Trimming
The final result is processed with: NoSpace(sResultString, sscOutside)
This removes leading and trailing whitespace from the completed plain-text value.

Unicode Preservation
Visible Unicode text outside HTML tags is retained.
For example: ** Résumé de François ** becomes: **Résumé de François**

Examples

Input	**Result**
Hello	Hello
Hello	Hello
One Two	One<CR>Two
One Two	One<CR>Two
One Two	One<CR>Two
Tom & Jerry	Tom & Jerry
5 < 10	5 < 10
"Hello"	"Hello"
<!-- note -->Visible	Visible
Title Text	Title<CR>Text
""	""

In the table above, <CR> represents WinDev's standard line separator.

Real-World Uses
FT_StripHTML() is useful for:

- Displaying email bodies as plain text.
- Cleaning HTML imported from websites.
- Preparing web content for reports.
- Creating searchable text from HTML fields.
- Removing markup from database content.
- Producing plain-text previews.
- Cleaning copied rich-text content.
- Exporting HTML descriptions to text files.
- Preparing content for indexing.
- Converting simple help content into plain text.

Practical Example
Convert an HTML product description into readable plain text.

```
sHTML is Unicode string = "<h2>FunctionTrak</h2>" + "<p>The WinDev function toolbox.</p>" + ...
   "<p>Simple, practical &amp;amp; reusable.</p>"

sDescription is Unicode string
sDescription = FT_StripHTML(sHTML)
```

Result:
FunctionTrak
The WinDev function toolbox.
Simple, practical & reusable.

Email Example
EmailRecord.PlainTextBody = FT_StripHTML(EmailRecord.HTMLBody)
This can provide a readable text version for searching, reporting, or applications that do not display HTML.

List Example
Input:

```
<ul>
   <li>Install FunctionTrak</li>
   <li>Copy the library</li>
   <li>Call the procedures</li>
</ul>
```

Result:
Install FunctionTrak
Copy the library
Call the procedures
The list markers are not generated, but each closing creates a separate line.

Table Example
Input:

```
<table>
<tr><td>Name</td><td>Status</td></tr>
```

<tr><td>FunctionTrak</td><td>Registered</td></tr>
</table>

Result:
NameStatus
FunctionTrakRegistered
The closing </tr> creates row boundaries, but cell boundaries are not converted into spaces or TAB characters.

Developer Tip
Use **FT_StripHTML()** for ordinary display-oriented HTML, email fragments, descriptions, and other relatively simple markup. It is deliberately lightweight. When processing complex or untrusted HTML documents where exact browser-like interpretation is required, use a dedicated HTML parser rather than relying on text scanning alone.

Common Pitfalls
1) This is a lightweight HTML stripper, not a complete HTML parser. HTML can contain malformed markup, quoted > characters, embedded scripts, styles, and other structures that require full parsing rules. This procedure applies a practical character-scanning approach.

2) Script and style content is not removed. The tags themselves disappear, but their enclosed text remains. For example:

<script>alert('Hello');</script>
becomes:
alert('Hello');
Likewise:
<style>body { color: red; }</style>
retains the CSS text.

Applications processing complete web pages may want to remove <script>...</script> and <style>...</style> blocks separately.

3) Structural tags with attributes are generally not recognized as line breaks.
For example:
<br class="clear">
is removed as a tag, but it does not match:
br
or:
br/
and therefore does not generate a line break. Closing structural tags normally do not contain attributes, so tags such as </p> and are still recognized in typical HTML.

4) Opening paragraph and division tags do not create line breaks. Only their recognized closing tags create structural separation.

5) Table cells are joined together. The procedure recognizes </tr> but not </td> or </th> as separators. Consequently:

<td>First</td><td>Second</td>
becomes:
FirstSecond
unless the source includes spacing between the cells.

6) List bullets are not generated.
The closing produces a line break, but the procedure does not prepend bullets, numbers, or indentation.

7) Entity names are case-sensitive. The replacement operations recognize exact lowercase forms such as:
&
An uppercase or mixed-case form is not decoded by the current implementation.

8) Only selected entities are decoded.
Entities such as:
©
€
—
remain unchanged.
Likewise, arbitrary decimal and hexadecimal numeric entities are not decoded.
For example:
©
©
remain in encoded form.

9) Repeated-space normalization does not preserve preformatted spacing. Multiple ordinary spaces are reduced to one, even when they originated inside <pre> content or were deliberately used for alignment.

10) Literal angle brackets may be interpreted as markup. Text such as:
Value < Maximum
contains an opening < without a closing tag marker. Once the procedure enters tag-processing mode, the remaining text may not be copied to the result.
Encode literal angle brackets as < and > before processing HTML-like content.

11) An unterminated ordinary tag discards the remaining visible text. For example:
Visible <strong unfinished text
returns only the portion before the unmatched <.

12) Comments are located using exact comment delimiters.
The procedure searches specifically for:
<!--
and:
-->

Malformed or unusually written comment syntax may not be removed as expected.

13) The procedure does not preserve hyperlinks.
For example:
Example
becomes:

Example
The visible anchor text remains, but the URL in the href attribute is discarded.

FT_TitleCase

Convert text into readable English title case while preserving punctuation, spacing, TABs, and line endings.
Purpose

Converts a Unicode string into title case. The procedure begins by applying **FT_ProperCase()**, then lowercases recognized small words when they appear between the first and last words of the title. Examples of recognized small words include:

a	by	or
an	for	the
and	from	to
as	in	with
at	of	
but	on	

The first and last words remain capitalized even when they appear in the small-word list.

Why This Procedure?
A basic proper-case function capitalizes every word:
The Lord Of The Rings
Formal English title case normally leaves certain short articles, conjunctions, and prepositions lowercase:
The Lord of the Rings
At the same time, those words should remain capitalized when they appear at the beginning or end:
The Road to

FT_TitleCase() applies these practical title-formatting rules without disturbing the source string's separators or punctuation.

Key Features

- Uses **FT_ProperCase()** as its capitalization foundation.
- Lowercases recognized small words inside a title.
- Always preserves capitalization of the first word.
- Always preserves capitalization of the last word.
- Ignores common punctuation when identifying small words.
- Preserves spaces exactly.
- Preserves repeated spaces.
- Preserves TAB characters.
- Preserves line-ending characters.
- Preserves punctuation in the returned title.
- Supports Unicode strings.
- Returns an empty string for empty input.

Syntax
sTitle is Unicode string = FT_TitleCase(sInputString)

Parameters

Parameter	Description
sInputString	Text to convert into title case.

Return Value

Type	Description
Unicode string	The supplied text converted according to the procedure's title-case rules.

Recognized Small Words

The procedure recognizes the following words:

a	an	and	as
at	but	by	for
from	in	into	nor
of	on	or	over
per	the	to	under
up	via	with	yet

These words are lowercased only when they are neither the first nor the last word.

Remarks

Initial Proper-Case Conversion

The procedure first calls: sWorkingString = FT_ProperCase(sInputString)
This establishes the initial capitalization and inherits the behavior of **FT_ProperCase()**, including its handling of:

- Mixed-case input.
- Apostrophe contractions.
- Possessives.
- Irish surnames beginning with O'.

For example: **THE LIFE OF O'BRIEN** first becomes: **The Life Of O'Brien** and is then refined to: **The Life of O'Brien**

Word Counting

Before applying small-word rules, the procedure counts the words in the title. A new word begins after any character whose numeric character code is 32 or lower. This includes:

- Ordinary spaces.
- TAB characters.
- Carriage returns.
- Line feeds.
- Other ASCII control characters.

The total word count allows the procedure to determine whether each word is:

- The first word.
- An interior word.
- The last word.

First Word
The first word is never lowercased by the small-word rules. For example: **the old man and the sea** becomes: **The Old Man and the Sea.** Although the is a recognized small word, its first occurrence remains capitalized because it begins the title.

Last Word
The final word is also never lowercased. For example: **what are you waiting for** becomes: **What Are You Waiting For.** The final word For remains capitalized even though for appears in the small-word list.

Interior Small Words
A recognized small word is converted to lowercase when it appears between the first and last words.
For example: **A TALE OF TWO CITIES** becomes: **A Tale of Two Cities** Similarly: **THE OLD MAN AND THE SEA** becomes: **The Old Man and the Sea**

Punctuation-Aware Comparison
Before checking whether a word belongs to the small-word list, the procedure creates a lowercase comparison value and removes these punctuation characters: . , ! ? : ; ( ) [ ] { } "

The original word is not stripped. The punctuation is removed only from the temporary comparison value.
For example: **War and Peace** recognizes and normally. A punctuated interior word such as: **(and)** is compared as: **and** and the original word is returned as: **(and).** The parentheses remain intact.

Preserved Separators
Words are processed independently, but every real separator is copied back to the result. Consequently, the procedure preserves:

- Single spaces.
- Multiple spaces.
- TABs.
- Carriage returns.
- Line feeds.
- Mixed whitespace.

For example, input containing three spaces between words retains those three spaces in the returned title.

Final-Word Processing
The word-processing loop runs for one extra iteration:
FOR nPosition = 1 TO Length(sWorkingString) + 1
That artificial final iteration acts as a separator so the final accumulated word can be processed even when the source does not end with whitespace. The artificial separator itself is not added to the result.

Empty Input
An empty input string immediately returns: ""

Examples

Input	Result
the lord of the rings	The Lord of the Rings
THE OLD MAN AND THE SEA	The Old Man and the Sea
a tale of two cities	A Tale of Two Cities
war and peace	War and Peace
from here to eternity	From Here to Eternity
what are you waiting for	What Are You Waiting For
the life of o'brien	The Life of O'Brien
""	""

Real-World Uses

FT_TitleCase() is useful for:

- Book titles.
- Article headings.
- Report titles.
- Document names.
- Menu captions.
- Product descriptions.
- Media titles.
- Presentation headings.
- Website page titles.
- Database title fields.
- Music and video catalogs.

Report Example

STC_ReportTitle = FT_TitleCase("sales by region and product category") Result: Sales by Region and Product Category

Preserving Punctuation

sTitle = FT_TitleCase("the good, the bad, and the ugly") Result: The Good, the Bad, and the Ugly

The commas remain in their original positions while the recognized interior small words are lowercased.

Preserving Separators

Input: **THE<TAB>LORD OF THE RINGS**
Result: **The<TAB>Lord of the Rings**

The TAB and repeated spaces are preserved.

Developer Tip

Use **FT_TitleCase()** for headings and titles intended for human display.
Use **FT_ProperCase()** when every word should generally begin with a capital letter, such as personal names:
sPersonName = FT_ProperCase(sName)
sBookTitle = FT_TitleCase(sTitle)

The two procedures are related, but they serve different formatting purposes.

Common Pitfalls

1) The procedure applies an English-language title convention. The built-in small-word list reflects common English articles, conjunctions, and prepositions. Other languages use different capitalization rules.

2) Existing specialized capitalization may be lost. Because **FT_ProperCase()** first converts the source to lowercase, acronyms and deliberately styled names are not preserved. For example: **NASA AND THE FUTURE OF AI** becomes: **Nasa and the Future of Ai.** The procedure does not currently include an acronym dictionary.

3) Title-case rules vary among publishers. Different style guides disagree about:

- Which small words should remain lowercase.
- Whether words after colons should be capitalized.
- How long a preposition must be before it is capitalized.
- How hyphenated compounds should be handled.

This procedure applies its own consistent, practical rule set rather than attempting to reproduce a particular publishing standard.

4) A small word following a colon is not automatically capitalized. For example: **a warning: the final chapter** becomes: **A Warning: the Final Chapter.** Because the is an interior small word, it is lowercased even though some title-case styles would capitalize the first word of a subtitle.

5) Word boundaries are based on whitespace. Punctuation does not divide words for counting purposes. For example, text joined with a slash or hyphen remains one whitespace-delimited word during the first/last-word calculation.

6) The comparison cleanup removes selected punctuation wherever it appears. Although the source comment describes removing surrounding punctuation, the implementation removes each listed punctuation character from the entire temporary comparison value. The original displayed word remains unchanged.

7) Apostrophes are not removed from the comparison value. This is appropriate for names and contractions, but it means an apostrophe-containing token will not match a plain small word.

8) Punctuation-only tokens count as words. Any non-whitespace sequence increases the word count, including a token made entirely from punctuation. This can affect which textual word is considered first or last in unusually formatted input.

9) Small-word matching uses a fixed internal list. Callers cannot currently supply:

- Additional small words.
- A different language list.
- Application-specific capitalization exceptions.
- Acronyms or brand names.

10) Line endings separate words but do not restart title rules. The first word of each new line is not automatically treated as the first word of a new title. The procedure treats the entire string as one title, regardless of embedded line breaks.

Related Procedures

Procedure	Why You Might Use It
FT_ProperCase()	Capitalizes each word and supplies the initial capitalization used by FT_TitleCase().
FT_SentenceCase()	Capitalizes sentence beginnings rather than title words.

FT_WordWrap

Wrap text to a preferred line width without splitting words or destroying paragraph boundaries.

Purpose

Reformats text into lines whose preferred maximum length is specified by the caller.

The procedure:

- Wraps text at word boundaries.
- Never splits a word.
- Leaves an oversized word intact on its own line.
- Combines consecutive nonblank source lines into paragraphs.
- Preserves blank lines as paragraph separators.
- Normalizes spaces, TABs, and control-character separators.
- Removes leading and trailing blank lines.
- Returns output using WinDev's standard CR line ending.

Why This Procedure?

Long text often needs to be reformatted for:

- Reports.
- Plain-text exports.
- Log files.
- Email messages.
- Fixed-width displays.
- Help content.
- Printed documentation.
- Text-based previews.

A simple character-based split can break words:
FunctionTr
ak
A basic line-by-line wrapper may also preserve arbitrary source wrapping that no longer suits the destination.

FT_WordWrap() treats the source as paragraphs, rebuilds each paragraph from its words, and creates new lines according to the requested width.

Key Features

- Wraps text at word boundaries.
- Never divides a word.
- Allows words longer than the requested width.
- Joins consecutive nonblank source lines into one paragraph.
- Preserves paragraph separation.
- Supports Windows CRLF line endings.
- Supports Unix/Linux LF line endings.
- Supports classic Macintosh CR line endings.
- Supports mixed source line endings.
- Reduces repeated spaces and TABs to one space.

- Removes leading whitespace from wrapped lines.
- Removes trailing whitespace from wrapped lines.
- Removes leading and trailing blank paragraphs.
- Returns the original text when the requested width is invalid.
- Uses WinDev's CR constant in the result.
- Supports Unicode strings.

Syntax
sWrappedText is Unicode string = FT_WordWrap(sInputString,nLineWidth)

Parameters

Parameter	Description
sInputString	Text to wrap.
nLineWidth	Preferred maximum number of characters permitted on each output line.

Return Value

Type	Description
Unicode string	Text wrapped to the preferred width without splitting words.

An empty input string returns an empty string. When nLineWidth is less than 1, the original input string is returned unchanged.

Remarks
Empty Input
When the input string is empty, the procedure immediately returns: ""

Invalid Line Width A requested width less than 1 is considered invalid. For example:
sResult = FT_WordWrap(sText, 0) returns the original value of sText without modification. The same applies to negative widths. This avoids attempting to wrap text using an impossible line length.

Preferred Maximum Width
The requested width is a preferred maximum rather than an absolute maximum. The procedure does not split words. If one word is longer than nLineWidth, that word remains intact and occupies its own line. For example: FT_WordWrap("FunctionTrak documentation", 10) returns conceptually:
FunctionTrak
documentation

Although FunctionTrak contains more than ten characters, it is not divided.

Word-Boundary Wrapping
A word is added to the current output line when the resulting length does not exceed the requested width. The calculation includes:
- The existing output line.
- One separating space.
- The new word.

Conceptually: Length(sOutputLine) + 1 + Length(sCurrentWord)

When that total exceeds nLineWidth, the current output line is written and the new word begins the next line.

Exact-Width Matches
A word is added when the completed line length is exactly equal to nLineWidth. For example, with a width of 10:
1234 56789

contains exactly ten characters, including the separating space, and remains on one line.

Paragraph Recognition
Consecutive nonblank source lines are treated as one paragraph. For example:
This is the first source line.
This is the second source line.
is initially combined into one paragraph:
This is the first source line. This is the second source line.

The paragraph is then wrapped according to the requested width. Existing source line endings inside a paragraph are therefore not preserved.

Blank Lines
A blank source line completes the current paragraph. The next nonblank source line begins a new paragraph. This allows paragraph boundaries to survive even though ordinary line endings within a paragraph are discarded.

Whitespace-Only Lines
A source line is tested by:

1. Removing TAB characters.
2. Applying NoSpace(..., sscAll).

A line containing only spaces or TABs is therefore treated as blank.

Leading Blank Lines
Blank lines before the first paragraph are ignored.
For example:

First paragraph.
returns without the leading blank lines.

Trailing Blank Lines
Blank lines after the final paragraph are also removed. The returned value ends with the final visible character rather than a paragraph separator.

Multiple Blank Lines
Once a paragraph has been completed, additional blank source lines do not create additional empty paragraphs.
Consequently, consecutive blank lines are effectively collapsed into a single paragraph boundary.

Space Normalization
During word processing, any character whose numeric value is 32 or lower ends the current word. This includes:

- Spaces.
- TABs.
- Carriage returns.
- Line feeds.
- Other ASCII control characters.

When words are rebuilt, exactly one ordinary space is placed between adjacent words on the same output line. Therefore:
Alpha Beta<TAB>Gamma
becomes:
Alpha Beta Gamma

before wrapping is considered.

Leading and Trailing Spaces
Leading and trailing whitespace within each paragraph is removed naturally during word extraction. The procedure does not append a word separator until a visible word has been found, and it writes no space after the last word on a line.

Supported Source Line Endings
The procedure recognizes:

Source	Line ending
Windows	CRLF
Unix/Linux	LF
Classic Macintosh	CR

A Windows CRLF pair is treated as one logical source line ending.

Output Line Endings
Wrapped output lines use WinDev's standard:
CR
constant.

The source line-ending convention is not preserved.

Final-Paragraph Processing
The procedure appends two line endings to the working copy:
sWorkingString = sInputString + CR + CR

These artificial line endings ensure that the final paragraph is completed even when the original text does not end with CR or LF.

After all text is processed, trailing line endings added during paragraph construction are removed.

Unicode Support
The procedure stores and returns Unicode strings. Unicode letters and symbols inside words are retained. Line-width calculations use WinDev's character-counting behavior rather than the visual display width of individual glyphs.

Real-World Uses
FT_WordWrap() is useful for:

- Formatting report narratives.
- Creating plain-text emails.
- Preparing text-file exports.
- Wrapping log messages.
- Building console-style output.
- Formatting documentation.
- Preparing fixed-width previews.
- Reformatting imported paragraphs.
- Producing printable notes.
- Standardizing database memo fields.

Practical Example
Wrap customer notes to 60 characters for a plain-text report.

```
sReportNotes is Unicode string
sReportNotes = FT_WordWrap(Customer.Notes, 60)
```

The result can be written directly to a text file or displayed in a fixed-width report control.

Reformatting Imported Text
A document may already contain line endings based on a narrow source window:
The customer contacted the office regarding
a recent invoice and requested an updated
copy by email.
Using:
sText = FT_WordWrap(sText, 72)
joins those source lines into one paragraph and wraps them again at the new preferred width.

Developer Tip
Use **FT_WordWrap()** when the text's logical paragraphs matter more than its existing physical line endings. This is particularly useful when imported text was wrapped for a different display width:
sText = FT_WordWrap(sImportedText, 80)

Do not use it when each original nonblank line carries independent meaning, such as:

- Source code.
- Mailing addresses.
- Poetry.
- Tables.
- Lists whose line boundaries must remain fixed.
- Preformatted data.

Common Pitfalls
1) The requested width is not guaranteed when an individual word is too long. Words are never split. A 30-character word remains 30 characters long even when the requested width is 20.

2) Existing nonblank source lines are joined. A line ending does not preserve a hard break unless it participates in a blank-line paragraph boundary. For example:
Name: Scott
Status: Registered

is treated as one paragraph and may become:

Name: Scott Status: Registered

Use a different routine when each source line must remain independent.

3) Repeated spaces and TABs are not preserved. All whitespace separating words is normalized to one ordinary space. The procedure is not suitable for column-aligned or preformatted text.

4) Leading indentation is removed. Indented paragraphs, nested lists, and code examples lose their leading spaces during word reconstruction.

5) The output uses CR, regardless of the source format. CRLF and LF source text are normalized to WinDev's standard CR constant.

6) Width is measured in characters, not visual screen width. Different Unicode characters and proportional fonts may occupy different amounts of display space even when the character counts are identical.

7) Paragraph spacing is normalized. Multiple consecutive blank source lines do not produce multiple distinct empty paragraphs.

8) An invalid width does not return an error. A width below 1 returns the original source text unchanged. Callers requiring validation should test the width before calling the procedure.

Related Procedures

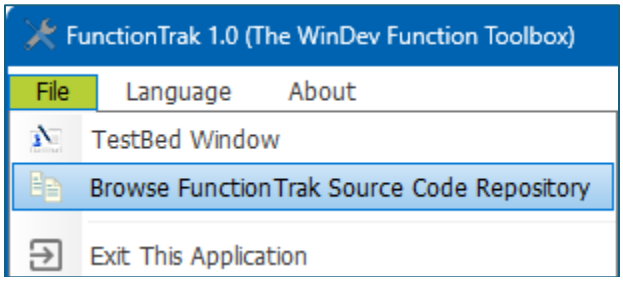
Procedure	Why You Might Use It
FT_LineCount()	Counts the lines produced after wrapping.
FT_SingleBlankLines()	Collapses excessive blank lines without rewrapping paragraph text.
FT_RemoveBlankLines()	Removes all blank lines when paragraph separation is unnecessary.
FT_CleanString()	Produces normalized single-line text instead of wrapped paragraphs.
FT_StripHTML()	Converts HTML into plain text before it is wrapped.

Source Code Repository

The Source Code Repository stores one or more versions of the FunctionTrak Function Library (.WL) within the encrypted FunctionTrak database. Rather than distributing the Function Library as a separate source file, FunctionTrak securely stores each release internally, allowing multiple library versions to be maintained from a single centralized repository.

Registered FunctionTrak users may browse the repository, review available Function Library versions, and export the desired .WL source code file directly to a folder of their choosing. The exported source code may then be imported into an existing WinDev project using WinDev's **Import WLanguage Source Code** feature.

To access the Source Code Repository, select **File → Browse FunctionTrak Source Code** from the FunctionTrak main menu.



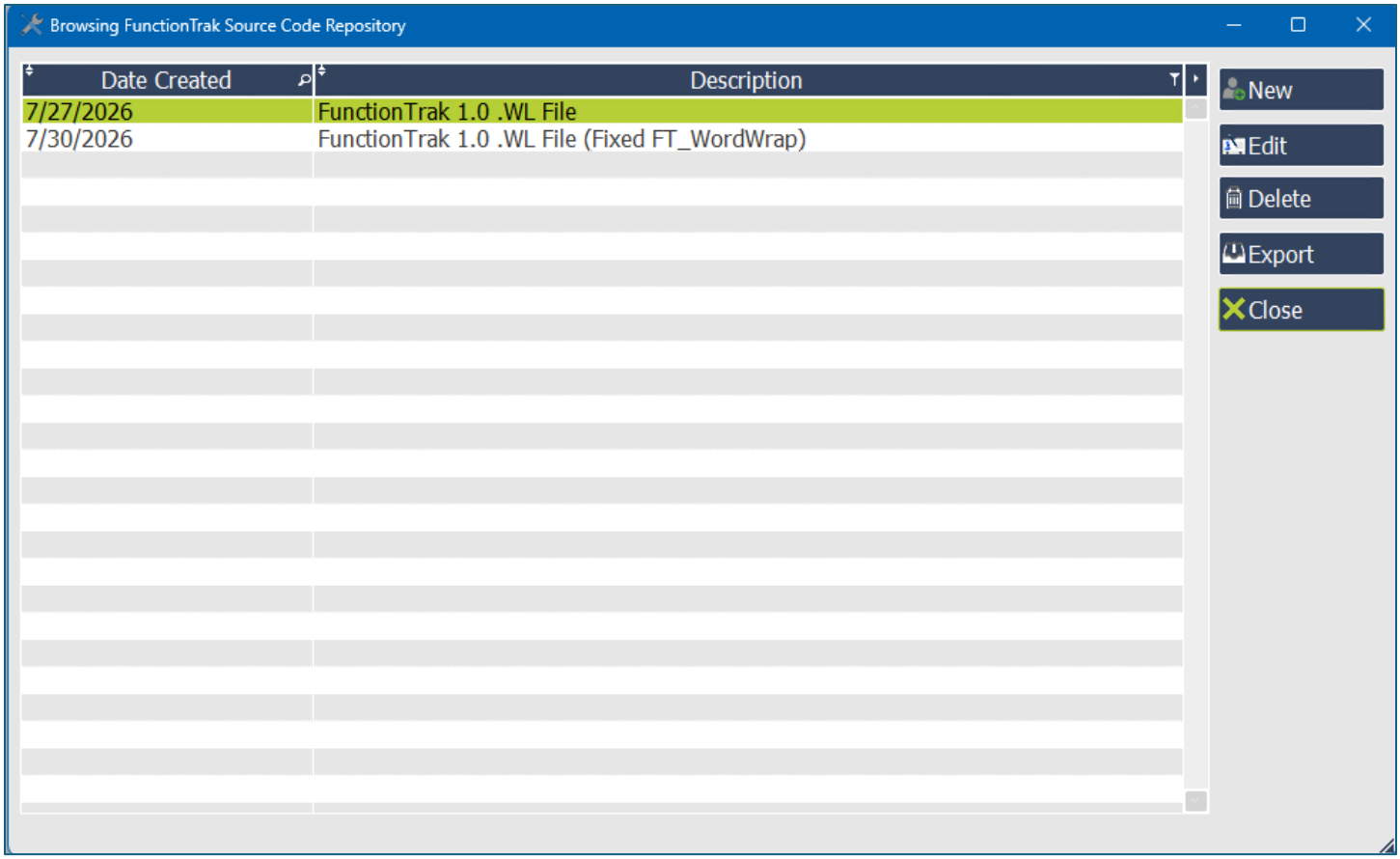
This repository-based approach provides several advantages. Multiple Function Library revisions may coexist within a single database, allowing previously released versions to remain available while newer versions continue to evolve. This simplifies long-term maintenance of existing applications and allows developers to standardize on a particular FunctionTrak release for each project.

As future FunctionTrak releases become available, additional Function Library versions may be added to the repository without affecting earlier releases. Developers may therefore continue maintaining older applications while taking advantage of new FunctionTrak capabilities in future projects.

Browsing the Source Code Repository

The Source Code Repository provides a centralized location for storing one or more versions of the FunctionTrak Function Library (.WL). Each repository record represents a complete Function Library that may be exported and incorporated into one or more WinDev applications.

The repository is designed to simplify Function Library management while providing a historical archive of previous releases. As new FunctionTrak versions are developed, additional Function Libraries may be added to the repository without affecting existing entries. This allows developers to continue supporting previously deployed applications while simultaneously taking advantage of newer FunctionTrak procedures and enhancements in future projects.



Each row within the browse window represents a complete Function Library version. The browse displays the following information:

Date Created

Indicates when the Function Library record was created within the repository.

Description

A brief description identifying the Function Library version, supported WinDev release, or significant enhancements included within that particular library.

The buttons located along the right side of the browse window provide the following functions.

New (Button)

Creates a new Function Library repository record. This function is primarily intended for maintaining future FunctionTrak releases or preserving customized Function Library variations.

Edit (Button)

Opens the selected Function Library record, allowing its descriptive information and embedded source code to be viewed or modified.

Delete (Button)

Removes the selected Function Library from the repository. Because deletion permanently removes that library version from the repository, this function should be used cautiously.

Export (Button)

Exports the selected Function Library to a standard WinDev (.WL) source file. The exported source code may then be imported into any compatible WinDev project.

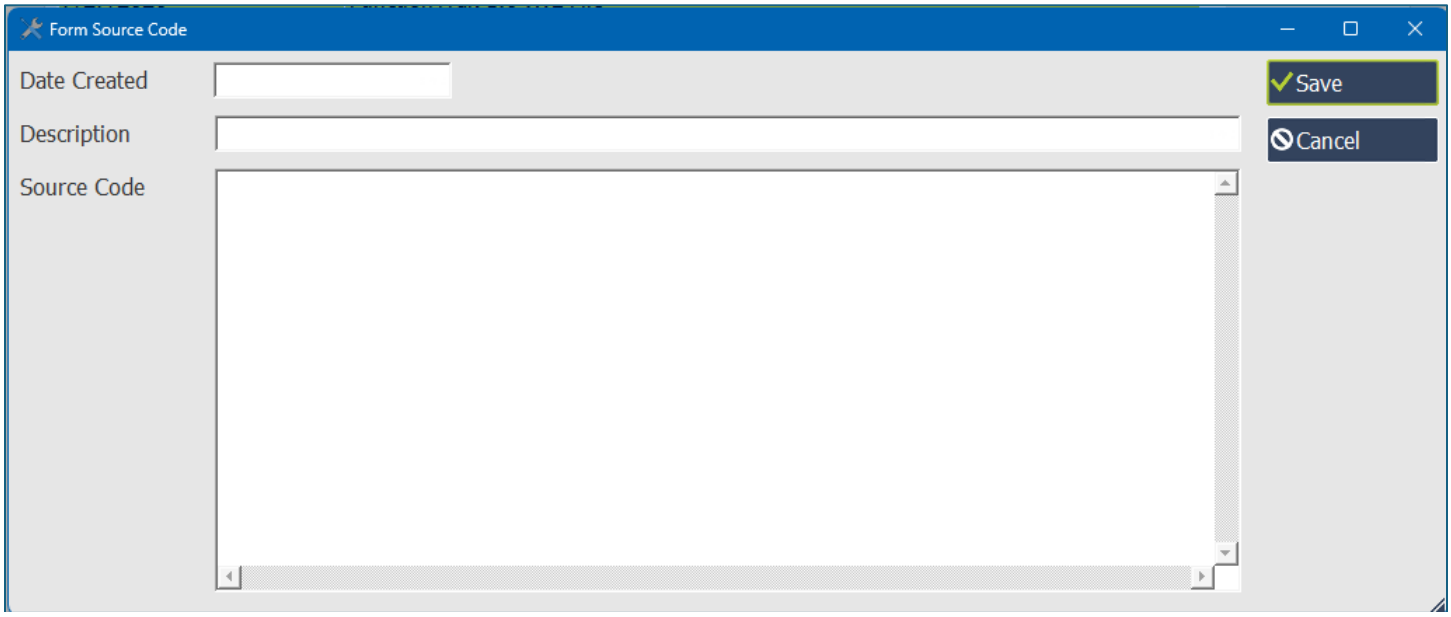
Close (Button)

Closes the Source Code Repository browse window and returns to the FunctionTrak main menu.

Although multiple Function Library versions may exist within the repository, only the selected version is

exported. This allows individual WinDev applications to continue using the Function Library version with which they were originally developed while newer projects may adopt later releases.

Creating a New Function Library
Selecting **New** displays the Function Library maintenance window.



Each Function Library record contains the following information:

Date Created
Specifies the date the Function Library was added to the Source Code Repository. This provides a chronological history of FunctionTrak releases. A popup calendar may be displayed by clicking the calendar button located at the right side of the entry field.

Description
A concise description identifying the Function Library. The description should clearly distinguish one Function Library version from another and may include the supported WinDev release, version number, or a summary of significant enhancements.

Source Code
Contains the complete contents of the FunctionTrak Function Library (.WL) source file. The source code is stored within the encrypted FunctionTrak database, providing a centralized repository for one or more Function Library versions. This field is fully Unicode compatible.

After entering the desired information, click **Save** to store the new Function Library within the Source Code Repository or **Cancel** to abandon the operation without saving any changes.

As additional FunctionTrak releases become available, new Function Library versions may be added without affecting previously stored versions. This approach provides a convenient historical archive while ensuring that older Function Libraries remain available for maintaining existing applications.

Editing Existing Function Libraries
Selecting **Edit** opens the selected Function Library record, allowing its descriptive information and embedded source code to be viewed or modified. This function is primarily intended for maintaining existing Function Library versions, correcting documentation, or incorporating new procedures and enhancements into future FunctionTrak releases. The Function Library maintenance window displays the the same fields as the depicted in the **New** button.

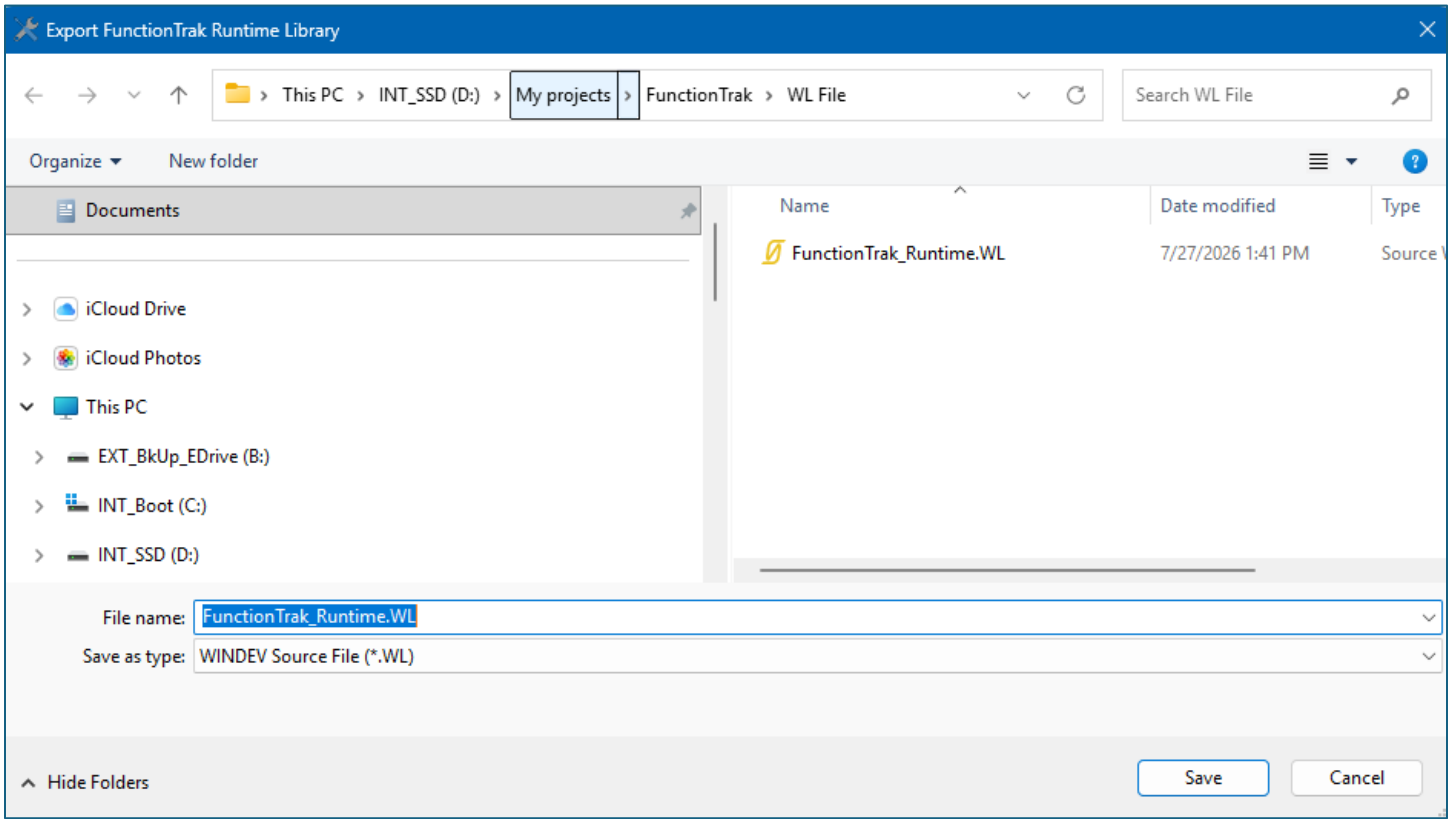
Because developers may build production applications against a particular FunctionTrak release, it is recommended that significant enhancements be stored as new Function Library records rather than replacing an existing production version. Maintaining separate Function Library versions preserves compatibility with existing applications while providing a reliable historical archive of the library's evolution.

As a general practice, released Function Library versions should remain unchanged after distribution. Creating a new repository record for each significant release provides a dependable version history while ensuring that previously released libraries remain available for future maintenance and support.

Exporting a Function Library
The **Export** button writes the selected Function Library from the Source Code Repository to a standard WinDev (.WL) source file. Once exported, the Function Library may be imported into any compatible WinDev application using WinDev's **Import WLanguage Source Code** feature.

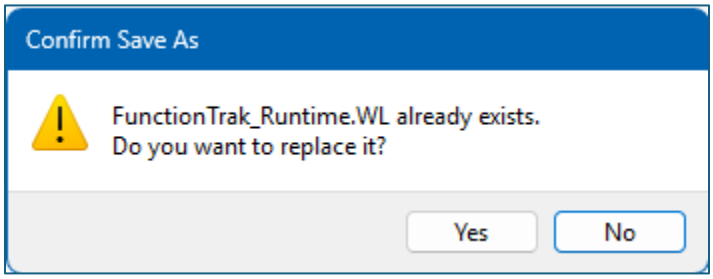
To export a Function Library, first select the desired repository record from the browse window, then click **Export**.

A standard Windows **Save As** dialog is displayed, allowing the destination folder and filename to be specified.

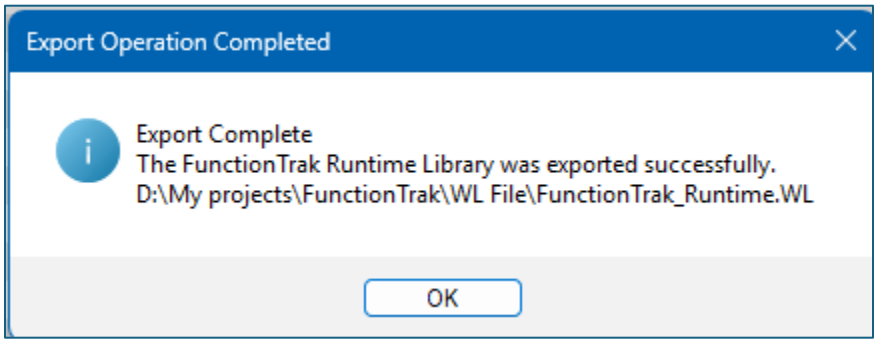


Although any valid filename may be entered, retaining the default filename is recommended whenever practical. Doing so simplifies future Function Library identification and minimizes confusion when maintaining multiple library versions.

After clicking **Save**, FunctionTrak writes the complete Function Library contained within the selected repository record to the specified .WL file. If the destination file already exists, FunctionTrak displays a confirmation dialog before overwriting the existing file:



Upon successful completion, a confirmation message indicates that the Function Library has been exported successfully:



The exported Function Library is a standard WinDev source code file and may be imported directly into any compatible WinDev project using WinDev's built-in **Import WLanguage Source Code** feature.

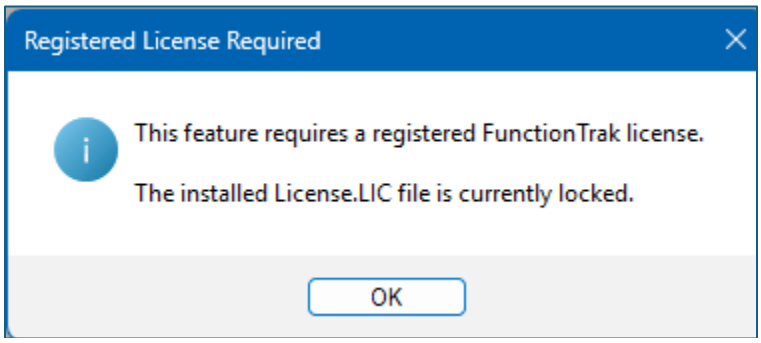
For security purposes, the Function Library is stored within FunctionTrak's encrypted Source Code Repository and is exported only when explicitly requested by a registered FunctionTrak user. This protects the Function Library from casual access while providing developers with a convenient method of obtaining the version required by a particular application.

If multiple Function Library versions exist within the repository, ensure that the desired version is selected before exporting. This allows older applications to continue using the Function Library version against which they were originally developed while newer applications may benefit from subsequent FunctionTrak enhancements.

License Protection

FunctionTrak utilizes **LicenseTrak**, Software by Daughtry's professional software licensing system, to manage product activation and protect the FunctionTrak Function Library from unauthorized distribution.
Rather than relying on executable wrappers, hardware dongles, Internet activation servers, or third-party licensing services, LicenseTrak is implemented entirely in native WinDev code. This approach provides reliable license management while allowing FunctionTrak to execute as a standard WinDev application.
FunctionTrak is available in both **Evaluation** and **Registered** editions. The Evaluation Edition provides unrestricted access to the Function Library demonstrations, documentation, and test environment, allowing developers to fully evaluate the available procedures before purchasing a license.

The principal difference between the Evaluation and Registered editions is access to the **Source Code Repository**. Evaluation users may explore and test every FunctionTrak procedure through the built-in demonstration environment, but access to the Function Library source code remains disabled until a valid registration license has been installed:



Registered users receive a personalized **License.LIC** file that immediately unlocks the Source Code Repository. Once unlocked, the desired Function Library version may be exported as a standard WinDev (.WL) source file and imported directly into existing WinDev applications.

This licensing model allows developers to thoroughly evaluate FunctionTrak before purchasing while protecting the considerable investment required to design, implement, test, and document the Function Library.

Installing Your Registration License
After purchasing FunctionTrak, Software by Daughtry will provide a personalized **License.LIC** file.
To activate the Registered Edition:

1. Close FunctionTrak if it is currently running.
2. Copy the supplied **License.LIC** file into the FunctionTrak installation folder, replacing the existing evaluation license file.
3. Restart FunctionTrak.

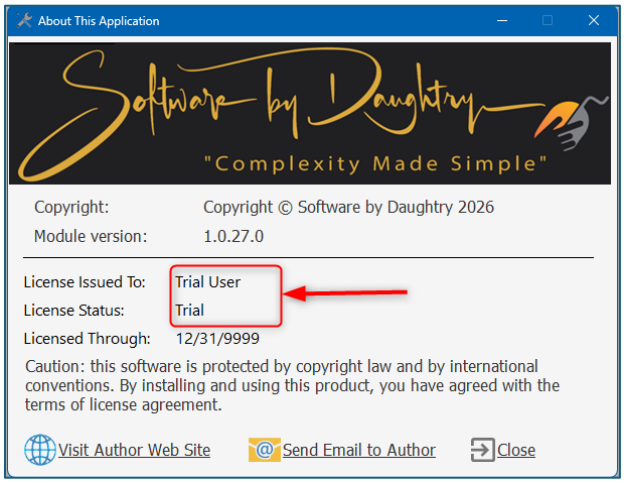
The application automatically detects the new license during startup. No additional activation steps, Internet connection, or registration process is required. Once the registration license has been recognized, the Source Code Repository becomes available and all registered-user functionality is immediately enabled.

Viewing License Information
Current license information may be viewed by selecting: **Help → About FunctionTrak**

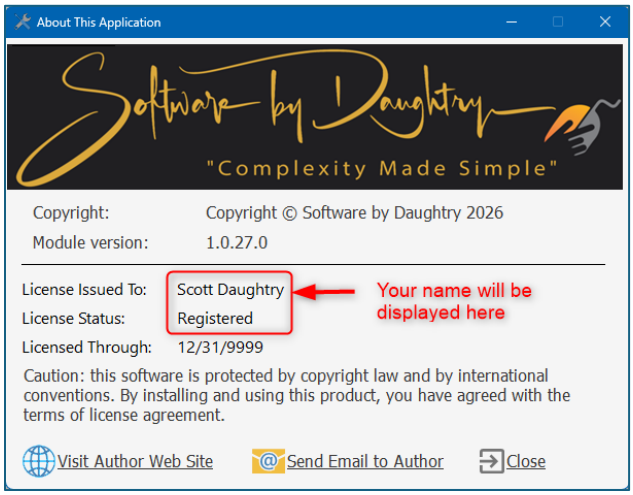
The About window displays licensing information including:

- License Status
- Registered To
- License Type
- Date Created

In trial mode, the About window displays:



In registered mode, the About window displays:



Evaluation Licenses
The Evaluation Edition is intended to provide developers with sufficient time to evaluate the FunctionTrak Function Library before making a purchasing decision. During the evaluation period, all demonstration procedures, documentation, and test examples remain fully functional, allowing each procedure to be exercised with your own data. Evaluation licenses are intentionally restricted from exporting the Function Library source code. This restriction protects the commercial value of FunctionTrak while still allowing a complete evaluation of its capabilities.

Registered Licenses
A Registered license permanently unlocks the FunctionTrak Source Code Repository.
Once registered, developers may:

- Browse all available Function Library versions.
- Export the Function Library as a standard WinDev (.WL) source file.
- Import the library into existing WinDev applications.
- Continue using the exported source code without dependence upon FunctionTrak.

Because the exported library consists entirely of standard WLanguage source code, no runtime DLLs, licensing components, or Internet services are required by applications that incorporate FunctionTrak procedures.

Why LicenseTrak?
FunctionTrak is itself an example of LicenseTrak operating within a real-world commercial application.
By using LicenseTrak to protect FunctionTrak, Software by Daughtry demonstrates the same licensing technology made available to other WinDev developers.

The licensing system protecting FunctionTrak is not a simplified demonstration or special-purpose edition—it is the same native WinDev licensing engine delivered as the LicenseTrak product.

Developers interested in protecting their own WinDev applications are encouraged to explore LicenseTrak, which provides evaluation licensing, permanent registration, subscription licensing, and source-code integration without requiring executable wrappers, cloud activation services, or third-party licensing frameworks. The Software by Daughtry web site (http://www.sdaughtry.com) provides two extensive demo applications and a 180+ page user manual which showcases the ease AND power that LicenseTrak possesses.

Developer Notes

FunctionTrak was created to solve practical programming problems encountered during the development of commercial WinDev applications. Every procedure included within the Function Library originated from a real-world requirement rather than as an academic programming exercise. As you begin incorporating FunctionTrak into your own projects, the following recommendations may help you obtain the greatest benefit from the library.

Import Only What You Need
The exported FunctionTrak Function Library is a standard WinDev (.WL) source code file containing all FunctionTrak procedures. Although importing the complete library is perfectly acceptable, many developers may prefer to copy only those procedures required by a particular application. Because each procedure is self-contained whenever practical, individual functions may easily be reused in existing projects. Keeping only the procedures your application actually uses simplifies long-term maintenance and makes future upgrades easier to manage.

Avoid Modifying Existing Procedures
Although every FunctionTrak procedure is supplied as fully editable source code, modifying existing procedures is generally discouraged. If a procedure no longer behaves as originally documented, future FunctionTrak updates become more difficult to integrate and troubleshooting becomes considerably more complicated. When application-specific behavior is required, consider writing a new wrapper procedure that calls the original FunctionTrak function rather than modifying the original implementation.

Preserve Your Library Version
When beginning a new project, record the FunctionTrak version that was imported into the application. As FunctionTrak evolves, additional procedures may be added and existing procedures may occasionally receive enhancements or bug fixes. Knowing which Function Library version was originally incorporated into a project greatly simplifies future maintenance. The Source Code Repository was specifically designed to preserve historical Function Library versions for this purpose.

Read the Documentation
Although most FunctionTrak procedures are intentionally simple to call, many contain subtle behaviors designed to handle edge cases that commonly occur in production software. For example, procedures that process file names, paths, dates, strings, or financial values often contain special-case logic that may not be immediately obvious from the procedure name alone. Spending a few minutes reviewing the accompanying documentation can often prevent hours of debugging later.

Test With Real Data
Every FunctionTrak procedure may be exercised directly from the built-in demonstration environment. Before integrating a procedure into a production application, experiment with your own data using the FunctionTrak testbed. This provides an excellent opportunity to observe how a procedure behaves with edge cases that may be unique to your application.

Suggestions Are Welcome
FunctionTrak continues to evolve as new programming challenges arise. If you discover a useful utility function that would benefit other WinDev developers, or identify an opportunity to improve an existing procedure, Software by Daughtry welcomes your suggestions. Many of the procedures currently included within FunctionTrak originated from practical problems encountered by developers working on production applications.

A Final Thought
Software development is filled with small problems that consume disproportionate amounts of time. Individually, they may seem insignificant; collectively, they can represent days or even weeks of development effort over the life of a project. FunctionTrak was created to eliminate many of those repetitive programming tasks by providing thoroughly tested, reusable procedures that solve common problems with straightforward, well-documented code. My hope is that FunctionTrak allows you to spend less time reinventing utility functions and more time building the unique features that make your own applications successful.

Happy coding!
Scott Daughtry
Software by Daughtry

Manual Revision History

This section summarizes significant revisions made to the FunctionTrak User Guide. Minor editorial corrections, typographical improvements, and formatting changes that do not affect the technical content of the documentation may not be individually listed.

Version	Date	Description
1.0	July 2026	Initial release of the FunctionTrak User Guide. Documented all FunctionTrak procedures, installation instructions, source code repository, licensing, and application operation.
1.1	*(Future)*	Reserved for future enhancements and documentation updates.

Notes

This User Guide is updated as FunctionTrak evolves. New procedures, enhancements to existing procedures, user interface changes, and significant documentation improvements will be incorporated into future revisions. Software by Daughtry recommends using the User Guide version that accompanies the corresponding FunctionTrak release to ensure that the documentation accurately reflects the functionality available within that version of the Function Library.

Documentation Feedback

Suggestions for improving this User Guide are always welcome. If you discover an error, omission, or explanation that could be made clearer, please contact Software by Daughtry. Constructive feedback from fellow developers plays an important role in the continued improvement of both FunctionTrak and its accompanying documentation.

Support and Registration

If you experience a problem while installing or using FunctionTrak, please contact us using the support email address provided on the Software by Daughtry web site. When requesting assistance, include a detailed description of the issue, the steps necessary to reproduce the problem, and any error messages or screenshots that may help us diagnose the situation. We will make every effort to respond as quickly as possible.

Before performing any troubleshooting or applying software updates, it is strongly recommended that you create a complete backup of your application folder and all associated data files. This simple precaution can help prevent accidental data loss and provide a reliable recovery point should an unexpected problem occur.

FunctionTrak is distributed as a Trial version. Upon receipt of payment, a personalized License.LIC file will be generated and delivered electronically. Simply replace the existing Trial License.LIC file with the registered version to unlock the application and remove the Trial restrictions.

Your personalized license file may be branded with your name or company name, providing a simple means of identifying the licensed owner. All data created or entered while using the Trial version remains fully accessible after registration; no data conversion or reinstallation is required.

Current pricing information, ordering instructions, and product updates are available on the Software by Daughtry web site.