Needful Things

Warranty Disclaimer

THIS SOFTWARE IS PROVIDED “AS IS”. PROFESSIONAL GRADE MACROS™ (“PGM”) SPECIFICALLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. PGM makes no representation or warranty that this software is fit for any particular purpose.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. In no event shall PGM or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use or inability to use this PGM product, even if PGM has been advised of the possibility of such damages. Some states do not allow the exclusion of incidental or consequential damages, so some of the above may not apply.

GOVERNING LAW AND GENERAL PROVISIONS. This Warranty Disclaimer shall be construed, interpreted, and governed by the laws of the State of Ohio in the United States of America, and any action hereunder shall be brought only in the County of Lake in the State of Ohio. If any provision is found void, invalid, or unenforceable, it will not affect the validity of the balance of this License and Warranty Disclaimer, which shall remain valid and enforceable to its terms. If any remedy hereunder is determined to have failed of its essential purpose, all limitations of liability and exclusion of damages set forth herein shall remain in full force and effect. This Warranty Disclaimer may only be modified in writing signed by you and a specifically authorized representative of PGM. All rights not specifically granted in this statement are reserved by PGM.
Software License

ATTENTION: PLEASE READ THIS DOCUMENT CAREFULLY BEFORE USING THIS PACKAGE. THE INDIVIDUAL OR ENTITY USING THIS PACKAGE (THE "END USER") AGREES TO BE BOUND BY THE TERMS OF THIS LICENSE.

The enclosed computer program(s) and the accompanying documentation are provided to the End-user by Professional Grade Macros ("Licensor") for use only under the following terms. Licensor reserves any right not expressly granted to the End-user. If purchased by retail box, the End-User owns the disk on which the Software is recorded, but Licensor retains ownership of all copies of the Software itself. The End-User assumes sole responsibility for the installation, use and results obtained from use of the Software.

License. End-User is granted a limited, non-exclusive license to install and maintain the Software on ONE COMPUTER PER PURCHASED LICENSE at any time for use only in the End-User's own business.

Restrictions. The End-User may NOT sublicense, assign, or distribute copies of the Software to others. The Software contains trade secrets. The End-User may NOT decompile, reverse engineer, disassemble, or otherwise reduce the Software to a human readable form. THE END-USER MAY NOT MODIFY, ADAPT, TRANSLATE, RENT, LEASE, LOAN, RESELL FOR PROFIT, DISTRIBUTE, OR OTHERWISE ASSIGN OR TRANSFER THE SOFTWARE, OR CREATE DERIVATIVE WORKS BASED UPON THE SOFTWARE OR ANY PART THEREOF.

Protection and Security. The End-User agrees to use its best efforts and to take all reasonable steps to safeguard the Software to ensure that no unauthorized person shall have access thereto and that no unauthorized copy, publication, disclosure or distribution in whole or in part, in any form, shall be made. The End-User acknowledges that the Software contains valuable confidential information and trade secrets and that unauthorized use and/or copying are harmful to Licensor.

Termination. This License is effective until terminated. This License will terminate immediately without notice from Licensor if the End User fails to comply with any of its provisions. Upon termination the End User must destroy the Software and all copies thereof, and the End-User may terminate this License at any time by doing so.

Limited Warranty. Licensor warrants that, for 10 days from the date of initial use (after purchase and registration) by the original End User, the Software shall operate substantially in accordance with the published functional specifications current at the time of shipment. If, during the warranty period, a defect appears, End User shall return the Software to Licensor and Licensor’s only obligation shall be, at Licensor's election, to replace the defective Software or refund the purchase price. The End-User agrees that the foregoing constitutes the End-User's sole and exclusive remedy for breach by Licensor under any warranties made under this Agreement. This warranty does not cover any Software that has been altered or changed in any way by anyone other than Licensor. Licensor is not responsible for problems associated with or caused by incompatible operating systems or equipment, or for problems in the interaction of the Software with software not furnished by Licensor.

No oral or written information or advice given by Licensor or its dealers, distributors, employees or agents shall in any way extend, modify or add to the foregoing warranty.
THE WARRANTY AND REMEDY PROVIDED ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE END-USER ASSUMES ALL RISK AS TO THE SUITABILITY, QUALITY, AND PERFORMANCE OF THE SOFTWARE. IN NO EVENT WILL LICENSOR, OR ITS DIRECTORS, OFFICERS, EMPLOYEES OR AFFILIATES, BE LIABLE TO THE END-USER FOR ANY CONSEQUENTIAL INCIDENTAL, INDIRECT, SPECIAL OR EXEMPLARY DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF DATA OR BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE OR ACCOMPANYING WRITTEN MATERIALS, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

LICENSOR'S LIABILITY TO THE END-USER (IF ANY) FOR ACTUAL DIRECT DAMAGES FOR ANY CAUSE WHATSOEVER, AND REGARDLESS OF THE FORM OF THE ACTION, WILL BE LIMITED TO, AND IN NO EVENT SHALL EXCEED, THE AMOUNT ORIGINALLY PAID TO LICENSOR FOR THE LICENSE OF THE SOFTWARE.

Enhancements. From time to time Licensor may, in its sole discretion, advise the End-User of updates, upgrades, enhancements or improvements to the Software and/or new releases of the Software (collectively, "Enhancements"), and may license the End-User to use such Enhancements upon payment of prices as may be established by Licensor from time to time. All such Enhancements to the Software provided to the End-User shall also be governed by the terms of this License. IN ORDER FOR THE END-USER TO BE ASSURED THAT IT WILL BE ADVISED OF AND LICENSED TO USE ANY ENHANCEMENTS TO THE SOFTWARE, THE END-USER MUST REGISTER THE SOFTWARE.

General. This License will be governed by and construed in accordance with the laws of the State of Ohio in the United States of America ("U.S.A."), and shall inure to the benefit of Licensor and End-User and their successors, assigns and legal representatives. If any provision of this License is held by a court of competent jurisdiction to be invalid or unenforceable to any extent under applicable law, that provision will be enforced to the maximum extent permissible, and the remaining provisions of this License will remain in full force and effect. Any notices or other communications to be sent to Licensor must be mailed first class, postage prepaid, to the following address: Professional Grade Macros, PO Box 5035, Timberlake, OH 44095-5035 U.S.A. This Agreement constitutes the entire agreement between the parties with respect to the subject matter hereof, and all prior proposals, agreements, representations, statements and undertakings are hereby expressly cancelled and superseded. This Agreement may not be changed or amended except by a written instrument executed by a duly authorized officer of Licensor.

Acknowledgment. BY USING THIS PACKAGE, THE END-USER ACKNOWLEDGES THAT IT HAS READ THIS LICENSE, UNDERSTANDS IT, AND AGREES TO BE BOUND BY ITS TERMS AND CONDITIONS. Should you have any questions concerning this License, contact Licensor at the address set forth above.
Trademarks and Copyrights

Copyright © 2001-2003 Professional Grade Macros.

The PGM Functions Library™ is a trademark of Professional Grade Macros, PO Box 5035, Timberlake, Ohio, USA 44095-5035. All Rights Reserved.

This software and documentation is protected by United States copyright laws, and by international treaty provisions.

All other products mentioned throughout this book are either trademarks or registered trademarks of their respective owners and/or companies. They are mentioned throughout this book only to enhance explanations. No mention or use of trade names used in this book are intended to convey endorsement or affiliation with this book, authors, or Professional Grade Macros.

Created and produced in the United States of America … with lots of help from Oz Land.
A Tip of the Hat

Paul Thornett of Pnambic Systems Pty. Ltd. (which he swears is in Australia) has had a big hand in the release of our new PGM Functions Library™. Without him, it would be later than it already is, without a doubt. Floyd and I are fortunate that he came along when he did, and has the talent that he does.

Kevin Heaton at Insight Software Solutions (theoretically in Utah), whose support, insight, help, and knowledge has been above and beyond the call of duty.
What's New with Us?

For those experienced with using our functions, you will see that we have taken a different approach to our PGM Functions Library™ than in the past.

Where we used to force $39 or so out of your death grip ahead of time, you can now use our PGM Functions Library™ free for one month … then give us money. We wanted to keep more in line with what the good folks at Insight Software Solutions do with Macro Express. And due to an untold number of requests and rather, uhm, interesting, albeit not repeatable, phone conversations, we have also instituted a multiple license price table that can be viewed on the Macro Express web site.

We have combined what used to be separate libraries into a single, comprehensive one that takes a lot less maintenance to keep updated. Less maintenance, of course, is strictly theoretical … but it certainly seems like a bright idea. It really does. Truly. Trust us, we are professionals and theoretically know these things.

The Language Extension functions, which used Macro Express as a direct link into VFP, have been left out of this release. Being based on VFP 6.0, and knowing that VFP is now at 8.0, Floyd and I simply did not have time to update them. However, they may be added to the PGM Functions Library™ at a later release level. For those of you that currently have our Language Extension Library, it will work fine with this one.

We have added a `{ PGM Function }` function, which is a master function, used to call other functions in the PGM Functions Library™ that use parameters, passing the parameters to them when called. For example:

```plaintext
Variable Set String %T1% "$\{\text{String - Occurs}\}$, In=Joe Weinpert, For=e, Case=0" // Set a string variable
Write Registry String: "PgmFunction" // Write to the Registry
Macro Run: { PGM Function } // Returns: 3
```

This particular release has been almost completely re-written to take advantage of the newest features of Macro Express 3.4.

We have added categories for String, Math and Science, File Operations, and Developer Tools.
# Table of Contents

**Needful Things** ................................................................................................................................. ii  
  Warranty Disclaimer .............................................................................................................................. ii  
  Software License ................................................................................................................................. iii  
  Trademarks and Copyrights ................................................................................................................. v  

**A Tip of the Hat** ................................................................................................................................. vi  

**What's New with Us?** .......................................................................................................................... vii  

**Table of Contents** .............................................................................................................................. viii  

**First Things First** ............................................................................................................................... 1  
  Installation ........................................................................................................................................ 2  
  Startup ................................................................................................................................................ 11  
  Registration ....................................................................................................................................... 12  
  Support .............................................................................................................................................. 13  
  Uninstall ........................................................................................................................................... 14  

**You Need to Know** ............................................................................................................................ 17  
  Terminology ....................................................................................................................................... 18  
  Overview .......................................................................................................................................... 19  
  How They Work ................................................................................................................................. 20  
  Source Code ..................................................................................................................................... 22  
  Exporting Functions .......................................................................................................................... 28  
  Tips .................................................................................................................................................... 43  
  Speed ................................................................................................................................................. 43  
  File Locking ..................................................................................................................................... 43  
  Caching ............................................................................................................................................ 44  
  Macro Priority ................................................................................................................................. 45  
  Registry Strings ............................................................................................................................... 46  
  X-Mouse .......................................................................................................................................... 46  
  Event Logging ................................................................................................................................. 46  
  Large Fonts ....................................................................................................................................... 46  
  Newsgroup ....................................................................................................................................... 46  

**PGM System Category** ..................................................................................................................... 47  
  Functions .......................................................................................................................................... 48  
  { PGM Function } .............................................................................................................................. 49  
  { PGM Help } ................................................................................................................................... 54  
  { PGM Registration } ........................................................................................................................ 55  
  { PGM Setup } .................................................................................................................................. 57  
  { PGM Bug Report } .......................................................................................................................... 58  
  { PGM } ............................................................................................................................................. 62
Variable Management Category

Overview .......................................................................................................................... 68
Reusable Macro Functions .............................................................................................. 70
The Registry ..................................................................................................................... 75
Saving and Restoring ...................................................................................................... 78
Functions .......................................................................................................................... 79
  { Variables - Save } ........................................................................................................ 80
  { Variables - Restore } ................................................................................................... 81
  { Variables - Save SI } ................................................................................................... 82
  { Variables - Restore SI } .............................................................................................. 83
  { Variables - Save 25 } ................................................................................................. 84
  { Variables - Restore 25 } ............................................................................................. 85
  { Variables - Save DT } ................................................................................................. 86
  { Variables - Restore DT } ............................................................................................ 87
  { Variables - Reset Registry Values } ........................................................................... 88
  { Variables - Reset Current Level } ............................................................................ 89

Program Operations Category .......................................................................................... 91
Overview .......................................................................................................................... 92
The Madness Behind the Method .................................................................................... 93
A History of Events .......................................................................................................... 97
Registry ............................................................................................................................. 99
  Program and Folder Values ........................................................................................ 102
    ProgramName ............................................................................................................ 102
    ProgramFolder ........................................................................................................... 102
    ProgramParameters .................................................................................................. 103
    WindowTitle ............................................................................................................. 103
    ExactMatch? .............................................................................................................. 103
  Launch Values ............................................................................................................... 104
    LaunchState ............................................................................................................. 104
    LaunchDelay ............................................................................................................. 104
    MaxLaunchDelay ....................................................................................................... 104
  Verification Values ...................................................................................................... 105
    VerifyLoops ............................................................................................................... 105
    VerifyDelay ............................................................................................................... 105
    MaxVerifyLoops ........................................................................................................ 105
**Date and Time Category**

Overview .......................................................... 130

Registry .................................................................. 131

Input Values ......................................................... 132

ParameterDecimal1 ........................................... 132
ParameterInteger1 ............................................. 132
ParameterInteger2 ............................................. 132
ParameterString1 ............................................... 133
ParameterString2 ............................................... 133

Return Values ..................................................... 134

ReturnDecimal1 .................................................. 134
ReturnInteger1 ................................................... 134
ReturnString1 ................................................... 134
ReturnError? ..................................................... 134

Default Values .................................................. 135

MonthPosition ................................................... 135
SundayStartsOn ............................................... 135

Functions .......................................................... 136

{ DateTime - Date to Julian } .................................. 138
{ DateTime - Julian to Date } .............................. 139
{ DateTime - Go Months } ................................. 140
{ DateTime - Go Days } ....................................... 142
String Category .............................................................................................................. 175

Overview ...................................................................................................................... 176

Registry ......................................................................................................................... 177

Input Values .................................................................................................................. 179
  ParameterString1 ......................................................................................................... 179
  ParameterString2 ......................................................................................................... 179
  ParameterString3 ......................................................................................................... 180
  ParameterInteger1 ........................................................................................................ 180
  ParameterInteger2 ........................................................................................................ 180
  ParameterInteger3 ........................................................................................................ 181

Return Values ............................................................................................................... 182
  ReturnString1 .............................................................................................................. 182
  ReturnInteger1 ............................................................................................................ 182
  ReturnInteger2 ............................................................................................................ 182
  ReturnError? ................................................................................................................ 183

Parsed Values ............................................................................................................... 184

Functions ...................................................................................................................... 185
  { String - Occurs } ........................................................................................................ 186
  { String - Pad } ............................................................................................................ 188
Math and Science Category ................................................................. 211
Overview ......................................................................................... 212
Registry ........................................................................................... 213
    Input Values ................................................................................ 213
        ParameterDecimal1 .............................................................. 214
        ParameterDecimal2 .............................................................. 214
        ParameterInteger1 .............................................................. 214
        ParameterString1 .............................................................. 214
    Return Values ............................................................................. 215
        ReturnDecimal1 ................................................................. 215
        ReturnInteger1 ................................................................. 215
        ReturnString1 ................................................................. 215
        ReturnError? ................................................................. 215
Functions ...................................................................................... 216
    { Math - Abs } ................................................................. 217
    { Math - Sign } ................................................................. 218
    { Math - Pi } ................................................................. 219
    { Math - Mod } ................................................................. 220
    { Math - Exp } ................................................................. 221
    { Math - Square Root } ........................................................ 222
    { Math - Ceiling } ............................................................... 223
    { Math - Floor } ................................................................. 224
    { Math - Degrees to Radians } ................................................ 225
    { Math - Radians to Degrees } ............................................. 226
    { Math - Decimal to Hex } .................................................. 227
    { Math - Hex to Decimal } .................................................. 228
    { Math - Integer to Hex } .................................................. 229
    { Math - Hex to Integer } .................................................. 230

File Operations Category ................................................................. 231
Overview ......................................................................................... 232
Registry ........................................................................................... 233
    Input Values ................................................................................ 234
        ParameterString1 .............................................................. 234
        ParameterString2 .............................................................. 234
        ParameterString3 .............................................................. 234
        ParameterString4 .............................................................. 234
        ParameterInteger1 .............................................................. 235
    Return Values ............................................................................. 236
        ReturnString1 ................................................................. 236
ReturnString2.................................................................................................................. ......................................... 236
ReturnString3.................................................................................................................. ......................................... 236
ReturnString4.................................................................................................................. ......................................... 236
ReturnError?................................................................................................................... ........................................... 236

Functions ...................................................................................................................... ...................................................... 237
{ File - Backslash Append } ................................................................................................. ......................................... 238
{ File - Backslash Remove } ................................................................................................. ......................................... 239
{ File - Compare } ............................................................................................................ ............................................. 240
{ File - Parse Path Name } ................................................................................................. ............................................. 246
{ File - Unique File Name } ................................................................................................. ............................................. 248
{ File - Validate File Name } ................................................................................................. ............................................. 249
{ File - Validate Path Name } ................................................................................................. ............................................. 251

Developer Tools Category................................................................................................. 253
Functions ...................................................................................................................... ...................................................... 254
{ Utility - Macro Timer } ................................................................................................ 255
{ Utility - Start Timer } ................................................................................................ 257
{ Utility - Get Timer } ................................................................................................ 259

Appendices..................................................................................................................... ......261
Change Log (Book) ........................................................................................................ 262
Change Log (Library)..................................................................................................... 263
Alphabetical Function List ............................................................................................... 264
Function Dependency.................................................................................................. 267
Functions Called ......................................................................................................... 267
Functions Called By..................................................................................................... 276
{ PGM Function } Listings............................................................................................. 284
Functions which can be called with { PGM Function } ................................................. 284
Functions which cannot be called with { PGM Function } ......................................... 287

Index.......................................................................................................................... ...........289
Someone suggested that I leave no blank pages, so …

Two hunters are out in the woods when one of them collapses. He doesn't seem to be breathing and his eyes are glazed. The other man pulls out his phone and calls emergency services.

He gasps to the operator: "My friend is dead! What can I do?" The operator in a calm, soothing voice replies: "Take it easy. I can help. First, let's make sure he's dead."

There is a silence, then a shot is heard.

Back on the phone, the hunter says, "OK, now what?"
First Things First

The difference between "involvement" and "commitment" is like an eggs-and-ham breakfast: the chicken was involved - the pig was committed.

- Unknown
Installation

There are two VERY important things that you need know before you install the PGM Functions Library™.

1. **You must have administrative rights** in order to run the installer, or for that matter, the uninstaller. Information is written to the HKEY_LOCAL_MACHINE hive key in your Windows Registry, and only administrators can do this.

2. **Macro Express version 3.4.0.1 (minimum)** must be installed and working before installing the PGM Functions Library™. The same is true when uninstalling.

You do NOT need to terminate Macro Express in order to install the PGM Functions Library™ (you cannot, however, be in the middle of editing or running a macro!). As a matter of fact, the installer runs two macros of its own during the install process; one before installing the Library and one afterwards. The installer pauses for a few seconds while the macros run. This is normal, and is also true when uninstalling.

One way to start the install process is to use the Run dialog. Browse to wherever you downloaded the setup.exe installer then click on OK. A progress window will display while the files are being extracted to your computer's temporary folder.

On the following page you will see three possible error messages that may be displayed by the installer.

The first message is displayed if you do not have administrative rights to the computer (see number 1 above). As with most software, during installation information about the PGM Functions Library™ is stored in the HKEY_LOCAL_MACHINE hive key of your Registry, where only administrators have access.

The second and third error messages are displayed if Macro Express is not installed on your computer at all or, if it is, then it is the wrong version (see number 2 above).
ERROR

Unable to install the program
Make sure you have ADMINISTRATOR Rights on this PC!

OK

Attention

Error! Could not locate Macro Express.

Macro Express version 3.4 (minimum) needs to be installed and run at least once prior to installing the PGM Functions Library™.

Installation canceled ... click OK.

OK

Attention

Error! Incorrect Macro Express version.

The PGM Functions Library™ requires Macro Express version 3.4 or later! Please update your Macro Express program to the latest version, which may be downloaded from www.macros.com/download.

Installation canceled. You may reinstall after the update ... click OK.

OK
If the installer recognizes that this version of the PGM Functions Library™ was already installed then the following warning message is displayed, otherwise, you will be taken directly to the Welcome dialog.

**Attention**

The 3.4.01 version of PGM Functions Library™ has already been installed!

If you wish to CANCEL this setup, you may do so from the Welcome screen, which will appear next.

If you wish to CONTINUE with this setup, your installation and registration information WILL NOT be overwritten. However, all updates, maintenance releases, and patches WILL be overwritten, so they will need to be reinstalled.

---

**Welcome**

Welcome to PGM Functions Library™ Setup program. This program will install PGM Functions Library™ on your computer.

It is strongly recommended that you exit Windows programs before running this Setup program.

Click **Cancel** to quit Setup and then close any program you have running. Click **Next** to continue with this Setup program.

**WARNING:** This program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

[www.pgmacros.com](http://www.pgmacros.com)
Here you are being asked to read and accept the PGM Functions Library™ license agreement, which must be accepted prior to continuing.
Here you are going to tell the installer where to place the PGM Functions Library™ on your computer. This will be not only where the PGMFL.mex file is found, but also all of the documentation and supporting files that go along with it. It is also where the files needed to uninstall will be placed.
This dialog confirms your PGM Functions Library™ destination choice. If you made a mistake, simply click on the Back button to reenter.
This progress dialog is displayed as the files are being installed on your computer. There are very few files so it should only take a few seconds to run.

Once the files have been installed you will be prompted to load the PGM Functions Library™ into Macro Express, which simply means that the next time you start Macro Express, the PGMFL.mex file will be displayed. It is NOT a requirement that you do this, but rather a matter of convenience.
This is the dialog displayed when the installer is finished. Because a few DLL (Dynamic Link Library) files were installed on your computer, which may have been in use during the install, you may be prompted to restart your computer before using the PGM Functions Library™.

It is also a good idea to review the Readme file that will appear if you leave the Display box checked. It covers what to do next and is shown on the next page.
Please review this file. It instructs you on what to do next.

### Startup
If you chose NOT to load the PGM Functions Library™ into Macro Express during the install process, you can do so in the same manner as any other Macro Express library file (.mex) the next time you start Macro Express.

Click on the **File** menu and then **Open Macro File**. Browse to the location where you installed PGMFL.mex (the PGM Functions Library™) and click OK. That's all there is to it!

### First Thing to Do
You need to run the `{PGM Setup}` function, which creates Registry variables in HKEY_CURRENT_USER required by the PGM Functions Library™. You will find it within the PGM Category. It only takes a few seconds to run, and you will need to do this for each user that logs on to this computer to run the PGM Functions Library™, or use macros developed with it.

### Registration
You must have administrative rights to register the PGM Functions Library™. If you purchased a license when you downloaded the PGM Functions Library™, you can enter it using the `{PGM Registration}` function (also located in the PGM category). Licenses are distributed on a per computer basis. If you want to use the PGM Functions Library™ on more than one computer (and who wouldn't?), then you can purchase a multiple license.
Startup

If you chose not to load the PGM Functions Library™ into Macro Express during the install process, you can do so in the same manner as any other Macro Express library file (.mex). The next time you start Macro Express click on the File menu and then Open Macro File. Now Browse to the location where you installed PGMFL.mex (the PGM Functions Library™) and click OK. That's all there is to it!

The very first thing to do

... is to run the { PGM Setup } function, which creates registry values in your personal area of the Registry (HKEY_CURRENT_USER). These values are required by the PGM Functions Library™. The { PGM Setup } function is located in the PGM System category. Right-click on it and then click on Run Macro Now. It only takes a few seconds to run, and will display progress messages on the screen. This only needs to be done once.

The { PGM Setup } function needs to be run once for each user that logs on to your computer to run the PGM Functions Library™, or use macros developed with it.
Registration

You must have administrative rights to register the PGM Functions Library™.

As you are aware, there is a one month trial period in which to test the PGM Functions Library™. At any point during this time, you can purchase a license from the Macro Express web site.

Licenses are distributed on a per computer basis. If you want to use the PGM Functions Library™ on more than one computer (and who wouldn't?), then you can purchase a multiple license from the same web site.

When you make your purchase, you will receive a license keycode via email similar to the one received when you purchased Macro Express. To enter it, simply run the {PGM Registration} function. Right-click on it and then click on Run Macro Now. The first bit of information to enter is your name, company name, or whichever name you sent to Insight Software Solutions to obtain the licensed registration key. This must be entered exactly as it appears in the email. Case is important here, and so are the spaces.

You will then be prompted to enter the license keycode. This also needs to be entered exactly as it appears in the email. Include the dashes too!

Once both pieces of information are entered and verified, you will be registered, and the information will be displayed. If you make a mistake entering the information, you will be the first to know, and have as many chances as needed to get it right. You can cancel out of the registration process at any time without harming your computer, Macro Express, or the PGM Functions Library™.
Support

We offer free technical support both *during* and after your one month free trial period.

**Email**
Questions can be emailed to support@pgmacros.com. Feel free to attach whatever may help us to help you.

**Fax**
If you prefer to fax us a problem, although we have plenty of our own, then use *(305) 675-3332*, which is an email fax service, i.e. your fax gets converted to email and sent to us automatically.

**{ PGM Bug Report }**
This is a built-in Library function that will gather information about your issue and then email it to us. See the **{ PGM Bug Report }** section within the *PGM System Category* chapter for a detailed explanation.

**Newsgroup**
We have our own newsgroup that can be used to resolve issues. Look under *Tips* within the *You Need to Know* chapter for a more detailed explanation.

**Mail**
Professional Grade Macros
PO Box 5035
Timberlake, Ohio 44095-5035
Uninstall

Just like installing, there are two VERY important things that you need know before you uninstall the PGM Functions Library™.

1. **You must have administrative rights** in order to run the uninstaller. Information is deleted from the HKEY_LOCAL_MACHINE hive key in your Windows Registry, and only administrators can do this.

2. **Macro Express version 3.4.0.1 (minimum)** must be installed and working before uninstalling the PGM Functions Library™.

You do NOT need to terminate Macro Express in order to uninstall the PGM Functions Library™. As a matter of fact, the uninstaller runs a macro of its own during the uninstall process. The uninstaller pauses for a few seconds while the macros run. This is normal.

Begin by accessing the Add or Remove Programs icon located in the Control Panel and then browsing down to the entry for the PGM Functions Library™ then clicking the Remove button.
This error message is displayed if you do not have administrative rights to the computer (see number 1 on the previous page). As with most software, when uninstalling the PGM Functions Library™, information is removed from the HKEY_LOCAL_MACHINE hive key of your Registry, where only administrators have access.

![Error Message](image1)

This next error message is displayed if Macro Express is not installed on your computer, which it needs to perform the uninstall (see number 2 above).

![Attention Message](image2)
This dialog confirms that you do want to uninstall the PGM Functions Library™.

During the uninstall process the progress dialog here will place a check mark next to each step as it is completed. When the Uninstall completed successfully message (at the bottom) appears, simply click on the Close button.
You Need to Know

The most overlooked advantage to owning a computer is that if they foul up, there’s no law against whacking them around a little.

- Porterfield
Terminology

Here is a description for four easily confused terms that we use throughout this book, plus a couple of extra ones for good measure:

**Command** - Any of the 300+ Macro Express native commands.
**Function** - A collection of commands for the express purpose of completing a single task.
**Category** - A collection of like functions.
**Macro** - A complete application consisting of one or more functions.
**Library** - A file of macro applications e.g. a .mex file.
**Machine-Independent Macro** - A macro which will not run on any machine.
**Keyboard** - An instrument used for entering errors into a system.
**Character Density** - The number of very weird people in the office, divided by the floor space.
**Recursive** - See recursive.
Overview

Macro Express contains approximately 300+ built-in, ready-to-use commands, so why in the world would anybody want, or need more? Because we, as users and developers, always want more, and … well, just need more. Even if Macro Express had 1,000 commands, this would still be true. Why? Because, like any other programming language, the commands themselves are just building blocks used in the construction of both simple and complex tasks.

Complex tasks, or macros, can be built from a series of simple tasks (reusable functions), which in turn, are built from the Macro Express commands. The PGM Functions Library™ provides for you some of these simple, but ready-made tasks.

If you need to:

- Easily create reusable functions
- Use multiple functions without overwriting variables
- Time your macros for efficiency
- Reliably launch applications
- Keep an unattended operations log of events
- Do date calculations
- Manipulate strings
- Parse strings
- Search strings
- Count the occurrence of one string in another
- Convert decimal values to hexadecimal strings
- Find square roots
- Compare files and folders
- Parse path names
- Validate file names

… then the PGM Functions Library™ is for you!
How They Work

Plain and simple, every one of our functions is called via the Macro Express Macro Run command. Let us assume, for the moment, that you are building a macro to save data from user input or other application to a file. Some if it is in hexadecimal format and needs to be converted. Normally, you would simply place code in your macro to handle the conversion. Like this:

```plaintext
// Convert a hex string %T10% to a decimal value %D10%
Variable Modify String: Uppercase %T10%
Variable Set Integer %N1% from Length of Variable %T10%
Variable Set Decimal %D10% to 0
Repeat with Variable using %N1%
Variable Modify Integer: %N3% = %N1% - %N2%
Variable Modify String: Copy Part of %T10% to %T1%
Switch (T1)
  Case: A
    Variable Set Decimal %D1% to 10
  End Case
  Case: B
    Variable Set Decimal %D1% to 11
  End Case
  Case: C
    Variable Set Decimal %D1% to 12
  End Case
  Case: D
    Variable Set Decimal %D1% to 13
  End Case
  Case: E
    Variable Set Decimal %D1% to 14
  End Case
  Case: F
    Variable Set Decimal %D1% to 15
  End Case
Default Case
  Variable Modify String: Convert %T1% to decimal %D1%
End Case
End Switch
If Variable %N3% = 0
  Variable Modify Decimal: %D10% = %D10% + %D1%
Else
  Variable Set Decimal %D2% to 1
  Repeat with Variable using %N3%
    Variable Modify Decimal: %D2% = %D2% * 16
  Repeat End
  Variable Modify Decimal: %D3% = %D2% * %D1%
  Variable Modify Decimal: %D10% = %D10% + %D3%
End If
End Repeat End
```
Pretty intense, huh? Sure, this takes care of the problem for this macro, but what about the next macro that needs the same capability? You could, of course, use copy and paste. However, what if the variables used here are used for something else in the macro that you are pasting to? They would all need to be changed. Here’s the solution (you knew it was coming): use our function.

```plaintext
// Convert a hex string to a decimal value
Variable Set String %T1% "e2b5c6" // Hexadecimal
Write Registry String: "ParameterString1" // Pass it
Macro Run: { Math - Hex to Decimal } // Convert it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D1 = 14857670
```

See the Macro Run command? It is calling one of the functions in our Library. You set a value to pass just prior to making the call and then read the results immediately afterwards. In this case %T1% holds the hexadecimal value to be converted, and %D1% is used to read the answer returned. Plain and simple … and ready-made for cutting and pasting!

Because we have no clue as to which variables your functions use, and have no desire to overwrite them when calling one of ours, we save yours before writing ours, and then restore them when finished.

Take another look at the above example. The called { Math - Hex to Decimal } function also uses the %T1% variable to do its work, however, because it saves the current set of variables prior to overwriting them and then restores them at the end, your variables are never changed.
Source Code

PGM Functions Library™ Source Code is available for most of the functions, the exceptions being the non-example functions in both the PGM System and Variable Management categories. When we refer to source code, we mean the password protection feature of Macro Express. The source code library file, PGMFLSource.mex, is identical to the standard library file, PGMFL.mex, except for the passwords, in-line documentation, and comments and may be purchased from the Macro Express web site.

The PGM Functions Library™ contains over 150 functions in 8 different categories, with over 14,000 lines of code. Source code is available for over 130 of these functions, and is a great thing to own. With it you can learn how to create reusable functions, customize how our functions work, investigate questions from people in your department or your clients, or simply satisfy your curiosity.

The source code is heavily commented throughout each function. Here is an example taken from the header section of the { Program - Launch } function:

```c
// PF:{ Program - Launch }
//PN:ProgName,WinTitle,ProgFolder,ProgParams,LaunchState,ExactMatch,LaunchDelay,VerifyLoops,VerifyDelay,ValidateParameters
// PA:PName,WTitle,Phone,PParams,WShow,Exact,LDelay,VLoops,VDelay,VParams
// PT:S,S,SO~.SO~.SO,NO0,NO,NO,NO,NO
//PD:ProgramName,WindowTitle,ProgramFolder,ProgramParameters,LaunchState,ExactMatch?,LaunchDelay,VerifyLoops,VerifyDelay,ValidateParameters?
// PW:  ***** DO NOT CHANGE ANY OF THE LINES ABOVE AND INCLUDING THIS ONE! *****

//-------------------------------------------------------------------------------
// Professional Grade Macros & Pnambic Systems Pty. Ltd.
// Floyd P Watergil - floyd@pgmacros.com
// Paul Thornett - pault1@hotkey.net.au
//-------------------------------------------------------------------------------

// Purpose -
// Launches a program and informs user of success or failure.

// Parameters -
// ProgramName = Name of program to run.
// ProgramFolder = Folder where program is run.
// WindowTitle = Name of program's window.
// ExactMatch? = True or false flag to determine if the passed WindowTitle parameter is an
// exact or partial match. 1 means true and 0 means false. Defaults to partial match (0).
// ProgramParameters = Command line string to pass to program.
// LaunchState = Program window to be normal, minimized, or maximized after launch. Defaults
// to normal.
// LaunchDelay = Number of seconds to wait after launching. Defaults to 0 (zero) seconds.
// VerifyLoops = Number of loops to verify that program has launched, the window title is
// running, and the window is on top. Defaults to 50 loops.
// VerifyDelay = Number of milliseconds to wait between verify loops. Defaults to 500
// milliseconds.
// ValidateParameters? = Flag to check passed parameters for valid data or values. If set to
// zero (the default) then it's assumed that all the parameters are correct and no
```
// validation takes place.

// Returns -
// If no error occurs or program already running:
// ReturnError? = 0
// If an error occurs:
// ReturnError? = 1

// Notes -
// When determining if the passed program file name exists, this function enables the "Use
// Search Path" feature. If a folder name is passed, it’s prepended to the program name so
// pass either a full path (folder and program name), or a separate folder and program name.
// Do not pass both.

// Variable management -
// All variables are saved upon entering and restored prior to exiting. The clipboard is
// not used at all.

// Local variables -
// T1, T2, T3, N2, and N3 = Temporary variables
// T9 = This function name
// T10 = Program name - ProgramName
// T11 = Program folder - ProgramFolder
// T12 = Program file parameters - ProgramParameters
// T13 = Launch state - LaunchState
// T15 = Command string
// T16 = Window title - WindowTitle
// T17 = Full program path and name
// N1 = Error flag - ReturnError?
// N9 = Validate passed parameters - ValidateParameters?
// N11 = Launch delay seconds - LaunchDelay
// N12 = Verify loops - VerifyLoops
// N13 = Verify delay milliseconds - VerifyDelay
// N15 = Exact match flag - ExactMatch?

With source code you can learn how others create real-time, live macro applications by building complex functions with basic Macro Express commands. Learn how to:

- Use the advanced Window Control commands
- Run a complete macro from a variable
- Control program flow using constructs such as Switch/End Switch, and Case/End Case
- Handle If/Else/End If situations, including the use of AND, OR, and XOR
- Use loops in your macros for repeat operations
- Launch, activate, and terminate programs and windows of other applications
- Reposition other windows on your desktop
- Minimize and maximize windows
- Process ASCII text files
- Create modify, and delete variables
You Need to Know PGM Functions Library™

- Set variables from prompts, files, and the clipboard
- Set variables from Window objects, environment space, and other variables
- Set values based on ASCII characters
- Convert variables from one type to another
- Perform arithmetic operations with variables
- Convert file names from Windows to DOS style names
- Work with date and time strings
- Pause and delay macros
- Put a macro in a wait state until an event occurs
- Read and write data to and from your Windows Registry, environment space, and files
- Create and delete Registry keys and values
- Use Ping to determine if you are on-line
- Create prompts and menus for your end users
- Use Macro Express to email messages
- Display messages to the user to give them options

The following category tables list those functions that do, and do not have source code available.

<table>
<thead>
<tr>
<th>Program Operations Category (all source code included)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Program - Launch }</td>
</tr>
<tr>
<td>{ Program Operations Example A }</td>
</tr>
<tr>
<td>{ Program - Focus }</td>
</tr>
<tr>
<td>{ Program Operations Example B }</td>
</tr>
<tr>
<td>{ Program - Terminate }</td>
</tr>
<tr>
<td>{ Program Operations Example C }</td>
</tr>
<tr>
<td>{ Program - Validate Launch Values }</td>
</tr>
<tr>
<td>{ Program Operations Example D }</td>
</tr>
<tr>
<td>{ Program - Validate Verification Values }</td>
</tr>
<tr>
<td>{ Program Operations Example E }</td>
</tr>
<tr>
<td>{ Program - Validate Termination Values }</td>
</tr>
<tr>
<td>{ Program Operations Example }</td>
</tr>
<tr>
<td>{ Program - Toggle Parameter Validation }</td>
</tr>
<tr>
<td>{ Program - Toggle Exact Match }</td>
</tr>
<tr>
<td>{ Program - Toggle Event Logging }</td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Program - Reset }</td>
</tr>
<tr>
<td>{ Program - Clear Error Flag }</td>
</tr>
</tbody>
</table>

* The { Program - Log Event } function includes source code (i.e. no password) in the standard PGMFL.mex library file, so it can be disabled or enabled depending on if you want to use the history feature of the Program Operations category to log events. This will also allow you to change the look and feel of the event log to suit your own needs. Note that event logging can also be enabled or disabled using the { Program - Toggle Event Logging } function.
### Date and Time Category (**all source code included**)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime - Date to Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Go Months</td>
<td></td>
</tr>
<tr>
<td>DateTime - Go Days</td>
<td></td>
</tr>
<tr>
<td>DateTime - Span of Days</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to DOY</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOY</td>
<td></td>
</tr>
<tr>
<td>DateTime - DOY to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Character Month</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Format Input String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Day from Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Day from Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Month from Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Month from Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Year from Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Year from Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Day from Today</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Month from Today</td>
<td></td>
</tr>
<tr>
<td>DateTime - Parse Year from Today</td>
<td></td>
</tr>
<tr>
<td>DateTime - Today to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Today to Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Today to DOY</td>
<td></td>
</tr>
<tr>
<td>DateTime - Today to Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Reset Defaults</td>
<td></td>
</tr>
<tr>
<td>DateTime - Set Defaults</td>
<td></td>
</tr>
</tbody>
</table>
### String Category *(all source code included)*

<table>
<thead>
<tr>
<th>Function</th>
<th>Example Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>String - Occurs</td>
<td>String Example - Occurs</td>
</tr>
<tr>
<td>String - Pad</td>
<td>String Example - Pad</td>
</tr>
<tr>
<td>String - Pad Center</td>
<td>String Example - Pad Center</td>
</tr>
<tr>
<td>String - Pad Left</td>
<td>String Example - Pad Left</td>
</tr>
<tr>
<td>String - Pad Right</td>
<td>String Example - Pad Right</td>
</tr>
<tr>
<td>String - Parse</td>
<td>String Example - Parse</td>
</tr>
<tr>
<td>String - Proper</td>
<td>String Example - Proper</td>
</tr>
<tr>
<td>String - Replicate</td>
<td>String Example - Replicate</td>
</tr>
<tr>
<td>String - Reverse</td>
<td>String Example - Reverse</td>
</tr>
<tr>
<td>String - Search</td>
<td>String Example - Search</td>
</tr>
</tbody>
</table>

### Math and Science Category *(all source code included)*

<table>
<thead>
<tr>
<th>Function</th>
<th>Example Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math - Abs</td>
<td>Math Example - Abs</td>
</tr>
<tr>
<td>Math - Sign</td>
<td>Math Example - Sign</td>
</tr>
<tr>
<td>Math - Pi</td>
<td>Math Example - Pi</td>
</tr>
<tr>
<td>Math - Mod</td>
<td>Math Example - Mod</td>
</tr>
<tr>
<td>Math - Exp</td>
<td>Math Example - Exp</td>
</tr>
<tr>
<td>Math - Square Root</td>
<td>Math Example - Square Root</td>
</tr>
<tr>
<td>Math - Ceiling</td>
<td>Math Example - Ceiling</td>
</tr>
<tr>
<td>Math - Floor</td>
<td>Math Example - Floor</td>
</tr>
<tr>
<td>Math - Degrees to Radians</td>
<td>Math Example - Degrees to Radians</td>
</tr>
<tr>
<td>Math - Radians to Degrees</td>
<td>Math Example - Radians to Degrees</td>
</tr>
<tr>
<td>Math - Decimal to Hex</td>
<td>Math Example - Decimal to Hex</td>
</tr>
<tr>
<td>Math - Hex to Decimal</td>
<td>Math Example - Hex to Decimal</td>
</tr>
<tr>
<td>Math - Integer to Hex</td>
<td>Math Example - Integer to Hex</td>
</tr>
<tr>
<td>Math - Hex to Integer</td>
<td>Math Example - Hex to Integer</td>
</tr>
</tbody>
</table>

### File Operations Category *(all source code included)*

<table>
<thead>
<tr>
<th>Function</th>
<th>Example Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>File - Compare</td>
<td>File Example - Compare</td>
</tr>
<tr>
<td>File - Parse Path Name</td>
<td>File Example - Parse Path Name</td>
</tr>
<tr>
<td>File - Unique File Name</td>
<td></td>
</tr>
<tr>
<td>File - Backslash Append</td>
<td></td>
</tr>
<tr>
<td>File - Backslash Remove</td>
<td></td>
</tr>
<tr>
<td>File - Validate File Name</td>
<td></td>
</tr>
<tr>
<td>File - Validate Path Name</td>
<td></td>
</tr>
</tbody>
</table>
**Developer Tools Category (all source code included)**

- { Utility - Macro Timer }
- { Utility - Start Timer }
- { Utility - Get Timer }

<table>
<thead>
<tr>
<th>Variable Management Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No Source Code Available</strong></td>
</tr>
<tr>
<td>{ Variables - Save }</td>
</tr>
<tr>
<td>{ Variables - Restore }</td>
</tr>
<tr>
<td>{ Variables - Save SI }</td>
</tr>
<tr>
<td>{ Variables - Restore SI }</td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>{ Variables - Reset Registry Values }</td>
</tr>
<tr>
<td>{ Variables - Reset Current Level }</td>
</tr>
</tbody>
</table>

**PGM System Category**

<table>
<thead>
<tr>
<th><strong>No Source Code Available</strong></th>
<th><strong>Source Code Included</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>{ PGM }</td>
<td>{ PGM Function Examples }</td>
</tr>
<tr>
<td>{ PGM Bug Report }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Function }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Function Error }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Help }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Registration }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Setup }</td>
<td></td>
</tr>
<tr>
<td>{ PgmSvr - Process }</td>
<td></td>
</tr>
</tbody>
</table>
You Need to Know PGM Functions Library™

Exporting Functions

Or importing, depending on your point of view! PGM Functions Library™ functions can be exported to other macro application libraries the same as any of your other macros, with one difference: all, or most, of our functions are dependent on other functions to work, so, you must to be sure that, when you export one, you export the supporting functions too. Use the following table to be sure that the proper functions are exported together.

To make things easy for you, each function begins, more or less, with the name of the category it belongs to. However, the placement of functions within categories, when exporting, is for convenience only and has nothing at all to do with whether they work or not. You could place all functions for all applications together in one big category (or no category) and they would all work just fine. We just find it handy to separate them.

Please note that, with the exception of { PGM Help } and { PGM Bug Report }, all functions in the PGM System category MUST be exported in order for any other exported function to work.

<table>
<thead>
<tr>
<th>PGM System Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>You must always export these!</strong></td>
</tr>
<tr>
<td>{ PGM }</td>
</tr>
<tr>
<td>{ PGM Function }</td>
</tr>
<tr>
<td>{ PGM Function Error }</td>
</tr>
<tr>
<td>{ PGM Registration }</td>
</tr>
<tr>
<td>{ PGM Setup }</td>
</tr>
<tr>
<td>{ PgmSvr - Process }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable Management Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To export any of these</strong></td>
</tr>
<tr>
<td>{ Variables - Save }</td>
</tr>
<tr>
<td>{ Variables - Restore }</td>
</tr>
<tr>
<td>{ Variables - Save SI }</td>
</tr>
<tr>
<td>{ Variables - Restore SI }</td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>{ Variables - Reset Registry Values }</td>
</tr>
<tr>
<td>{ Variables - Reset Current Level }</td>
</tr>
<tr>
<td>Program Operation Category</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>{ Program - Launch }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Program - Focus }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Program - Terminate }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Program - Validate Launch Values }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Program - Validate Verification Values }</td>
</tr>
<tr>
<td>{ Program - Validate Termination Values }</td>
</tr>
<tr>
<td>{ Program - Toggle Parameter Validation }</td>
</tr>
<tr>
<td>{ Program - Toggle Exact Match }</td>
</tr>
<tr>
<td>{ Program - Toggle Event Logging }</td>
</tr>
<tr>
<td>{ Program - Reset }</td>
</tr>
<tr>
<td>{ Program - Clear Error Flag }</td>
</tr>
</tbody>
</table>
## Date and Time Category

<table>
<thead>
<tr>
<th>To export any of these</th>
<th>Export these too</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ DateTime - Date to Julian }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Julian Number }</td>
</tr>
<tr>
<td>{ DateTime - Go Months }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ DateTime - Go Days }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ DateTime - Span of Days }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to DOY</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOY</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOY</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>DateTime - DOY to Date</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Julian</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to DOW</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
</tbody>
</table>

**DateTime - Date to DOW**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
</tbody>
</table>

**DateTime - Julian to Character DOW**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
</tbody>
</table>

**DateTime - Date to Character DOW**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Character DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>Function Name</td>
<td>Code Block</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Math - Mod</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>DateTime - Character DOW</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>DateTime - Character Month</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>Math - Mod</td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td>DateTime - Format Input String</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td>{ DateTime - Parse Day from Date }</td>
</tr>
<tr>
<td>DateTime - Parse Day from Date</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Function</td>
<td>Commands</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Julian }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Julian }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Date }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Julian }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Today }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Today }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Today }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td></td>
</tr>
</tbody>
</table>
You Need to Know

<table>
<thead>
<tr>
<th>PGM Functions Library™</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Date }</td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td>{ Math - Mod }</td>
</tr>
</tbody>
</table>

{ DateTime - Today to DOW }

| { Variables - Save DT } |
| { Variables - Restore DT } |
| { Variables - Save 25 } |
| { Variables - Restore 25 } |
| { Program - Log Event } |
| { DateTime - Date to DOW } |
| { DateTime - Evaluate for Leap Year } |
| { DateTime - Validate Date String } |
| { Math - Mod } |

{ DateTime - Today to Character DOW }

| { Variables - Save DT } |
| { Variables - Restore DT } |
| { Variables - Save 25 } |
| { Variables - Restore 25 } |
| { Program - Log Event } |
| { DateTime - Character DOW } |
| { DateTime - Date to Character DOW } |
| { DateTime - Validate Date String } |
| { Math - Mod } |

{ DateTime - Today to DOY }

<p>| { Variables - Save DT } |
| { Variables - Restore DT } |
| { Variables - Save 25 } |
| { Variables - Restore 25 } |
| { Program - Log Event } |
| { DateTime - Date to DOY } |
| { DateTime - Date to Julian } |
| { DateTime - Evaluate for Leap Year } |
| { DateTime - Validate Date String } |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Today to Julian }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ DateTime - Reset Defaults }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>{ DateTime - Set Defaults }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td>String Category</td>
<td>To export any of these</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>{ String - Occurs }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Pad }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Pad Center }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ String - Pad }</td>
</tr>
<tr>
<td>{ String - Pad Left }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ String - Pad }</td>
</tr>
<tr>
<td>{ String - Pad Right }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ String - Pad }</td>
</tr>
<tr>
<td>{ String - Parse }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Proper }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Replicate }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Reverse }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ String - Replicate }</td>
</tr>
<tr>
<td>{ String - Search }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>Math and Science Category</td>
<td>To export any of these</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>{ Math - Abs }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{ Math - Sign }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{ Math - Exp }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{ Math - Square Root }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{ Math - Ceiling }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{ Math - Floor }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{Math - Degrees to Radians}</td>
<td>{Variables - Save DT}</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Save 25}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore 25}</td>
</tr>
<tr>
<td></td>
<td>{Program - Log Event}</td>
</tr>
<tr>
<td></td>
<td>{Math - Pi}</td>
</tr>
<tr>
<td>{Math - Radians to Degrees}</td>
<td>{Variables - Save DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Save 25}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore 25}</td>
</tr>
<tr>
<td></td>
<td>{Program - Log Event}</td>
</tr>
<tr>
<td></td>
<td>{Math - Pi}</td>
</tr>
<tr>
<td>{Math - Decimal to Hex}</td>
<td>{Variables - Save DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Save 25}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore 25}</td>
</tr>
<tr>
<td></td>
<td>{Program - Log Event}</td>
</tr>
<tr>
<td>{Math - Hex to Decimal}</td>
<td>{Variables - Save DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Save 25}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore 25}</td>
</tr>
<tr>
<td></td>
<td>{Program - Log Event}</td>
</tr>
<tr>
<td>{Math - Integer to Hex}</td>
<td>{Variables - Save DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Save 25}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore 25}</td>
</tr>
<tr>
<td></td>
<td>{Program - Log Event}</td>
</tr>
<tr>
<td></td>
<td>{Math - Decimal to Hex}</td>
</tr>
<tr>
<td>{Math - Hex to Integer}</td>
<td>{Variables - Save DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore DT}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Save 25}</td>
</tr>
<tr>
<td></td>
<td>{Variables - Restore 25}</td>
</tr>
<tr>
<td></td>
<td>{Program - Log Event}</td>
</tr>
<tr>
<td></td>
<td>{Math - Hex to Decimal}</td>
</tr>
<tr>
<td><strong>File Operations Category</strong></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>To export any of these</strong></td>
<td><strong>Export these too</strong></td>
</tr>
<tr>
<td>{ File - Compare }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ File - Unique File Name }</td>
</tr>
<tr>
<td>{ File - Parse Path Name }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Unique File Name }</td>
<td>None</td>
</tr>
<tr>
<td>{ File - Backslash Append }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Backslash Remove }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Validate File Name }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Validate Path Name }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
</tbody>
</table>
### Developer Tools Category

<table>
<thead>
<tr>
<th>To export any of these</th>
<th>Export these too</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Utility - Macro Timer }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Utility - Get Timer }</td>
</tr>
<tr>
<td></td>
<td>{ Utility - Start Timer }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ Utility - Start Timer }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Utility - Get Timer }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ Utility - Get Timer }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
</tbody>
</table>
**Tips**

Here are some tips that will help you use PGM Functions Library™ more effectively. Some were submitted by you, the users, and we really appreciate it!

**Speed**

Some things that you can do to speed up your macros when using the *Macro Run* command.

**File Locking**

Take a look at the preferences setup dialog (Options -> Preferences -> Network). Turn File Locking off as shown here. There is an absolutely **HUGE** difference in macro speed with this option turned off, like somebody engaged the afterburners!
Caching
Take another look at the preferences setup dialog (Options -> Preferences -> Caching). Make sure that both of these options are checked. This is space that is set aside in memory for storing functions that have been called. They will be called from here the next time they are used (in the same macro session), rather than read in from your hard drive.
Macro Priority
There is an undocumented Registry value that can be used to give Macro Express more time to run when your processor decides that it is your turn:

\textit{HKEY\_LOCAL\_MACHINE\Software\Insight Software Solutions\Macro Express\AdvOptions\Macro Priority}

When testing the PGM Functions Library™ we change this value to zero to give Macro Express the \textit{most priority} when running a macro because everything we do is centered around this wonderful piece of software. This may, or may not be a good idea for you … we simply do not know because we do not know what the maximum (slowest) value is, or for that matter, what a zero value does to the time allotted for other running programs (other than give the highest priority to the running macro). You could, of course, experiment, and it may help to know that the \textit{default} value is 1,000.

If Macro Express is running when you add or change this value, it must be restarted to take effect. Also, you may create or change the same value in the \textit{HKEY\_CURRENT\_USER} hive key.

Being the utmost authority on Macro Express, here is what Insight Software Solutions says:

\textit{In some situations, Macro Express utilizes too many CPU cycles and the application that is being automated does not run smoothly. The Macro Priority setting causes Macro Express to pause for a moment when running a long macro to allow an application a chance to catch up. The Macro Priority option is a registry setting that cannot be changed from the preferences panel within Macro Express itself. This setting is subject to change in the future.}
Registy Strings

Do not attempt to write *The Rise and Fall of the Roman Empire* to any Registry string value, ours or anybody else’s. Be aware of how much data is contained in a string before you write it to the Registry. If strings are too large, you will receive an error message from your operating system that tells you only the first 16k was written.

X-Mouse

There are various utilities, like *TweakUi*, that allow X-Mouse like behavior in Windows 2000, XP, and possibly others. This means that to activate any window, all you have to do is position the mouse over that window for the amount of time you specified, for example 500 ms. So be careful with your hands when running functions that utilize any of the *Mouse* command functions. Keep them off of the mouse and the keyboard whenever you are running a macro that types-out text. Or you may find that text is being typed into the wrong window!

Event Logging

The `{ Program - Log Event }` function includes source code (i.e. no password) in the standard PGMFL.mex library file, so it can be disabled or enabled depending on if you want to use the history feature of the Program Operations category to log events. This will also allow you to change the look and feel of the event log to suit your own needs. Note that event logging can also be enabled or disabled using the `{ Program - Toggle Event Logging }` function.

Large Fonts

If you have changed your Windows system to use *Large Fonts*, then expect problems with dialogs, not just in the PGM Functions Library™, but in general. In addition to things like not being able to read all the text in a text box, or a window title, we have experienced some pretty weird system behavior at times with Large Fonts. Behaviors that just "go away" when switching back to standard fonts. Which behaviors? Well … weird ones.

Newsgroup

In addition to sponsoring a newsgroup for Macro Express and Keyboard Express, we have one of our own (*Public.PgmFunction3*) for developers and users to openly and freely exchange information, ideas, and solutions centered around the PGM Functions Library™.

To access any of our newsgroups, simply subscribe to them (they are, of course, free!). For example, start Outlook Express and go to Tools -> Accounts -> Add -> News, then follow the wizard instructions. Use *pgmacros.apk.net* for the NNTP server name. There are no login requirements or passwords needed. They are completely free and open.
To invent, you need a good imagination and a pile of junk.

- Thomas Alva Edison
Functions

The functions in this category are accessed by practically every other function within the PGM Functions Library™. Collectively they are the engine that runs the show and are split into two groups: those that are for you to use, and those that handle internal matters and housekeeping.

Functions for your use:
  { PGM Function }  Master function that you can use to call other parameter-driven functions
  { PGM Help }      Context-sensitive help system
  { PGM Registration }  Registers your license and/or displays registration information
  { PGM Setup }*     Variable management handler
  { PGM Bug Report }  Creates trouble report and emails to support@pgmacros.com

Internal use functions:
  { PGM }           System functions handler
  { PGM Function Error }  Error handler for { PGM Function }
  { PgmSvr - Process }  Processor for the Visual Basic DLL library interface

* The { PGM Setup } function is also used internally by the { PGM } function to assist with some of its housekeeping duties.
As mentioned previously, we have added a master \{ PGM Function \} function, which is used to call other functions in the PGM Functions Library™ that use parameters, passing the parameters to them when called. For example:

```
Variable Set String %T1% "String - Occurs", ln=Joe Weinpert, For=e, Case=0" // Set a string variable
Write Registry String: "PgmFunction" // Write to the Registry
Macro Run: \{ PGM Function \} // Returns: 3
```

In order to preserve any macros that you have created using a previous version of the PGM Functions Library™, you can still use the original way of setting the parameter values and then calling the function:

```
Variable Set String %T1% "Joe Weinpert" // Set the string to be searched
Variable Set String %T2% "e" // Set what to search for and count
Variable Set Integer %N1% 0 // Case-insensitive search
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry String: "ParameterString2" // Write T2 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Macro Run: \{ String - Occurs \} // Call the function. Returns: 3
```

Either method works fine, we just wanted to provide for you an alternative way of calling functions containing numerous parameter values that must be set. As you can see from this example, the alternative \{ PGM Function \} method requires much less coding than the original method for the same function.

Not all functions lend themselves to this alternative, \{ PGM Function \} calling method. It is meant only for those functions that have parameters to be set. The \{ Variables - Save \} and \{ Variables - Restore \} functions would not use this method, however, the \{ Program - Launch \} and \{ Program - Focus \} functions would.

How can you tell which is which? There are four ways to tell them apart:

1. Look for a paragraph titled "\{ PGM Function \} Parameters" within the explanation section of the function in question. If it is not there then it has no parameters to pass and cannot be called by \{ PGM Function \}.

2. Create a test macro to call a function with \{ PGM Function \}. If an error is displayed then you know that it cannot be done.

3. Look in the Registry under HKEY_CURRENT_USER\Software\Professional Grade Macros\Pgm Functions. If you see it listed then it can be called with the \{ PGM Function \} function.

4. Look in the \{ PGM Function \} Listings appendix.

For those functions that do lend themselves to being called by \{ PGM Function \}, you will see something like the picture shown on the next page at the beginning of each script if you have purchased source code, but follow along, even if you did not.
PF: Function name
Contains the case-sensitive name of the function.

PN: Parameter names
Contains the full names of parameters.

PA: Parameter abbreviations
Contains the shortened names of parameters.

PT: Parameter types
Contains parameter data types and requirements
- S  Required string
- N  Required integer
- D  Required decimal
- SO Optional string
- NO Optional integer
- DO Optional decimal
- SO[sss]  Initialized string
- NO[nnn]  Initialized integer
- DO[ddd]  Initialized decimal
- SO~  Initialized empty string

PD: Parameter Registry locations
Contains the registry location of parameter

PL: Registry Key location
Used in {PgmSvr - Process} only and not shown, it contains the Registry Key location of the parameter.

PW: Delimiter line
Marks the end of the function header string (***** Warning *****)

Applying this information to what is seen in the example, we learn that {String - Occurs} has three parameters SearchIn or In, which is a required string passed to the ParameterString1 value, SearchFor or For, which is a required string passed to the ParameterString2 value, and finally CaseSensitive? Or Case, an initialized integer, which, if not passed to the ParameterInteger1 value, will be initialized to 0.
So that when \{ PGM Function \} calls the \{ String - Occurs \} functions, it does not have to sort through the header each time to know what to do, the information in PN:, PA:, PT:, and PD: is all blended together and placed in the Registry for each individual function.

When you create a macro like this:

```
Variable Set String %T1% "{ String - Occurs }, In=Joe Weinpert, For=e, Case=0" // Set a string variable
Write Registry String: "PgmFunction"          // Write to the Registry
Macro Run: { PGM Function }            // Returns: 3
```

you are telling \{ PGM Function \} to locate the \{ String - Occurs \} function in the Registry, decipher the stored data, stuff the passed parameters into wherever they are supposed to go, then call the function. In other words, \{ PGM Function \} does all the parameter writing for you.

Some functions contain both required and optional parameters. You don't have to pass optional parameters, but you must pass all required ones. If you miss passing a required parameter, an error will be generated. Also, if you attempt to pass more parameters than the function is looking for, required and optional combined, then an error will be generated.

In order to be as flexible as possible, \{ PGM Function \} can be called with named, abbreviated, or unnamed parameters. You may mix-and-match named and abbreviated parameters together, but not unnamed parameters. They must stand alone and be listed in their correct order, which can be found under the "\{ PGM Function \} Parameters" paragraph within the explanation section of each function. Only unnamed parameters need to be listed in order, both named and abbreviated parameters may be in any order, this is their advantage.
Good:  
Variable Set String %T1% "{ String - Occurs }, Searchln=Joe Weinpert, SearchFor=e, CaseSensitive?=0"

Good:  
Variable Set String %T1% "{ String - Occurs }, CaseSensitive?=0, SearchFor=e, Searchln=Joe Weinpert"

Good:  
Variable Set String %T1% "{ String - Occurs }, In=Joe Weinpert, SearchFor=e, Case=0"

Good:  
Variable Set String %T1% "{ String - Occurs }, Searchln=Joe Weinpert, SearchFor=e"

Good:  
Variable Set String %T1% "{ String - Occurs }, Joe Weinpert, e, 0"

Good:  
Variable Set String %T1% "{ String - Occurs }, Joe Weinpert, e"

Bad:  
Variable Set String %T1% "{ String - Occurs }, e, 0, Joe Weinpert"  - Unnamed parameters out of order

Bad:  
Variable Set String %T1% "{ String - Occurs }, Joe Weinpert, e, 0,"  - Cannot end with a comma

Bad:  
Variable Set String %T1% "{ String - Occurs }, Searchln=Joe Weinpert, e, 0"  - Mixing named and unnamed parameters

Bad:  
Variable Set String %T1% "{ String - Occurs }, Joe Weinpert"  - Too few required parameters

Bad:  
Variable Set String %T1% "{ String - Occurs }, Joe Weinpert, e, 0, 5"  - Too many parameters

Named and abbreviated parameters are case-insensitive. Upper, lower, and mixed case are all fine. SEARCHIN, searchin, and sEaRcHiN are all valid. However, and this is very important, the name of the function that you are calling is case-sensitive and must be entered exactly as it is named.

Good:  
{ String - Occurs }

Bad:  
{ STRING - OCCURS }

Bad:  
{String-Ocurs}

Also, the use of spaces is very flexible. Before and after comma separators, equal signs, etc. are perfectly acceptable. The following command will work fine:

Variable Set String %T1% "{ String - Occurs }, Searchln = Joe Weinpert, SearchFor = e , CaseSensitive? = 0 "

If you are passing string parameters containing either commas ( , ) or equals signs ( = ), then those strings must be wrapped in either single quotes ( ' ), double quotes ( " ), or brackets [ ] . Bear in mind that whichever set of wraps you use, the string cannot contain one of those either.

Good:
"John's book is red"
[John's book is red]
"2 + 3 = 5"

Bad:
'John's book is red'  - Same wrap character that the string contains
2 + 3 = 5  - No wrap character
Knowing when and how to use quoted strings can be tricky. The answer for one situation may not hold true for the next, even in the same command. Take a look at the macro listing here (purposely broken into two lines).

```
Variable Set String %T1% "{ Program - Launch },
excel.exe, Microsoft Excel, c:\program files\microsoft office\office10, c:\my documents\My Spreadsheet.xls"
```

Microsoft Excel will return an error saying something like "c:\my" file cannot be found. Why? Because there are spaces in the command line argument, therefore the string needs to be wrapped in quote marks.

```
Variable Set String %T1% "{ Program - Launch },
excel.exe, Microsoft Excel, c:\program files\microsoft office\office10, "c:\my documents\My Spreadsheet.xls"
```

However, Microsoft Excel will still return the same. Why? Because the parser will remove the quote marks you just placed there thinking that it is simply a quoted string, therefore you need another set of quote marks to prevent the inside ones from being removed.

```
Variable Set String %T1% "{ Program - Launch },
excel.exe, Microsoft Excel, c:\program files\microsoft office\office10, "c:\my documents\My Spreadsheet.xls"
```

Now it will run fine. But that leaves the question "Why does the program path not need quote marks, after all it too contains spaces?" The answer is … I'm not sure. I suspect, however, that Macro Express itself may have something to do with it because in the end, it gets written to the Program Name field in the Macro Express Program Launch command.

The bottom line is that you may need to experiment with quoted strings to find the correct approach to an individual situation.

We know that when developing macros mistakes are made, so there are two types of errors returned by \{ PGM Function \}. The first are those that are generated by this function for things like improper syntax, missing parameters, invalid function names, etc., which will be displayed in a dialog text box stating what went wrong. The second are generated by the actual function that you are calling and are recorded in whichever manner is local to that function.

Speaking of chickens:

```
A waiter asks a patron, "May I take your order, sir?"

"Yes," the patron replies. "I'm just wondering, exactly how do you prepare your chickens?"

"Nothing special, sir. We just tell them straight out that they're going to die."
```
{ PGM Help }

The PGM Functions Library™ has a basic, built-in, context-sensitive help system. By basic, we mean nothing fancy, just good, old-fashioned, html help dialogs, which are very easy to access, navigate, and use. You must have Internet Explorer 4.0 or greater to use { PGM Help }.

Say, for example, that you are working in the script editor and are creating a function, part of which needs to parse a string. Naturally you want to use our { String - Parse } function to do the conversion, but you forgot which parameters to pass and which are returned.

Simply move the editing bar over the Macro Run: { String - Parse } command line, press the { PGM Help } hotkey Ctrl+Alt+Shift+F1 combination, and the help dialog will be displayed. If you think that this picture looks amazingly like this book, you would be absolutely correct.
{ PGM Registration }

You must have administrative rights to register the PGM Functions Library™.

Note that, for your convenience, most of the information you will read here is duplicated within the Registration section of the First Things First chapter.

The purpose of this PGM system function is two-fold: 1) to license the PGM Functions Library™ by registering your user name and keycode that was received when you made your purchase, and 2) to display the licensed registration information for you.

As you are aware, there is a one month trial period in which to test the PGM Functions Library™. At any point during this time, you can purchase a license from the Macro Express web site.

Licenses are distributed on a per computer basis. If you want to use the PGM Functions Library™ on more than one computer (and who wouldn’t?), then you can purchase a multiple license from the same web site.

When you make your purchase, you will receive a license keycode via email similar to the one received when you purchased Macro Express. To enter it, simply run the { PGM Registration } function. Right-click on it and then click on Run Macro Now. The first bit of information to enter is your name, company name, or whichever name you sent to Insight Software Solutions to obtain the licensed registration key. This must be entered exactly as it appears in the email. Case is important here, and so are the spaces.

You will then be prompted to enter the license keycode. This also needs to be entered exactly as it appears in the email. Include the dashes too!
Once both pieces of information are entered and verified, you will be registered, and the information will be displayed. If you make a mistake entering the information, you will be the first to know and have as many chances as needed to get it right. You can cancel out of the registration process at any time without harming your computer, Macro Express, or the PGM Functions Library™. After registering, you can display your license information at any time by simply re-running this function.
{ PGM Setup }

Note that, for your convenience, most of the information you will read here is duplicated within the Startup section of the First Things First chapter.

This PGM system function creates the registry values required by the PGM Functions Library™ in your personal area of the Registry (HKEY_CURRENT_USER). This needs to be the very first function that you run after installing the PGM Functions Library™, as noted earlier in great big red letters. It takes about 10 seconds to run and displays several progress windows as it runs. Here are just two of them:

![Creating File and Folder Registry values.]

Although this only needs to be done once, it needs to be done for each user that will log on to your computer to run, or use, macros developed with the PGM Functions Library™, simply because each user has their own HKEY_CURRENT_USER section of the Registry.

If you forget to run it for a new user, it will run automatically whenever they attempt to run any of the functions in the Library, however, two things may happen 1) the user will receive an incorrect answer from whichever function he or she attempted to run, and 2) the unexpected occurrence of the progress windows that this function displays, may just scare the beejeezes out of them.
{ PGM Bug Report }

The purpose of this function is to help us help you if a problem occurs while using the PGM Functions Library™. With your permission, it gathers some basic information about your particular setup and emails the results, along with a description of your problem.

To use this function you must set your E-Mail Preferences in Macro Express. Please refer to the Help system within Macro Express to learn about these settings. The PGM Functions Library™ depends on them being correct, so it is a good idea to test sending an email manually with the E-Mail Send command after setting your preferences, but prior to using this function.
To email a problem to us, simply run the `{ PGM Bug Report }` function by entering the Ctrl+Alt+Shift+B hotkey combination, or right-click on it and then click on Run Macro Now. The dialog screen for entering a problem description will appear.

![Dialog screen for entering a problem description](image)

Explain as best you can what the problem is so we can recreate and investigate it in order to determine the best resolution for you, then press the **Escape** key.

*Note that if you entered the `{ PGM Bug Report }` function by accident, you can simply stop it in the same manner as you do your other macros: Either right-click on the little guy on the task bar, or by hitting your Abort Macro HotKey, which can be found and/or set in Options -> Preferences -> Playback.*

The function will now begin collecting relevant data that will help resolve your problem. There is a progress window displayed like the one shown here for each stage of the process. The first four stages will take a few seconds to run.

![Progress window for stage 1](image)
Once *PGM Bug Report* has finished the collection process, the data is summarized in a text box for your convenience. And a prompt is displayed asking you what to do.

A file called PgmBugReport.enc has been saved in
  c:\Program Files\Professional Grade Macros\Output

This file has been compressed so that it will not take long to email to us.
It has also been encrypted so as to protect sensitive information contained within.

The file contains the following pieces of information:
  Operating system
  Name of current macro library
  Screen resolution
  Name of current program running
  Name of current folder
  Contents of clipboard
  Names of all currently running visible windows
  Names of all files in your Professional Grade Macros folder and subfolders
  Names of all files in your Macro Express main folder
  Various registry values created by PGM and Macro Express

This information enables Professional Grade Macros support personnel to help resolve your issue.

Is it OK to email this file now using your Internet connection?

[Yes] [No]
You do NOT have to email us immediately. The **PgmBugReport.enc** data file remains intact until the next time this function is run. So, answering No, presents the following dialog.

If, on the other hand, you answer Yes, then the report is emailed, after which **{ PGM Bug Report }** ends.
This PGM system function is for internal, housekeeping duties with the help of \textit{PGM Setup}, of course.
{ PGM Function Error }

Internal error handler for the { PGM Function }. 
There are some processes within the PGM Functions Library™ that are done through Visual Basic. You will find the DLL files that this function processes (through the Window Control %C99% variable) in the <PGM home>\Applications sub-folder. There are currently three plus the executable file:

- **PgmLibFns.dll**: Everything needed for Visual Basic to read from, and write to, the Registry.
- **PgmLibMath.dll**: Math functionality not yet available in Macro Express.
- **PgmLibReg.dll**: Applied Visual Basic Registry functions.
- **PgmSvr.exe**: PGM Functions Library™ / Visual Basic / Macro Express interface.

When the PGM Functions Library™ was first installed on your computer, an area called PGM Server was created in your Registry within the Professional Grade Macros key. This area, or key, contains values which are used to communicate with the **{ PgmSvr - Process }** function.

### PgmSvrAppClass

The name of the PgmSvr.exe application and class needed in order to run the desired process. An application can contain many classes. Currently this is set only within **{ PGM Setup }** and **{ Math - Exp }**.

### PgmSvrCompleted

Used to force **{ PgmSvr - Process }** to wait until PgmSvr.exe has completed processing the current request. Initialized to 0 by **{ PgmSvr - Process }**, set by PgmSvr.exe, and interrogated by **{ PgmSvr - Process }**.
**PgmSvrDebugMode?**
When set to 1, PgmSvr.exe will display some rudimentary messages to aid in debugging. Reset by PgmSvr.exe.

**PgmSvrFunction**
In the same way that an application can contain many classes, so too a class can contain many functions. Where a class does contain more than one user function, this value specifies which function is needed.

**PgmSvrMaxWait**
Defines the maximum number of seconds to wait for the completion of any PgmSvr.exe process. A value of 0 defines a wait for up to 86,400 seconds (24 hours)!

**PgmSvrRetainClass?**
When this indicator is 0, each call to PgmSvr.exe is self-contained - i.e. no values within PgmSvr.exe are retained from one call to the next. When this value is 1, PgmSvr.exe values persist from one call to the next - e.g. as would be needed for array processing. The final call in a series of calls to PgmSvr.exe must reset this indicator to 0. At present, there are no functions that need this capability.

**PgmSvrSleepWait**
A positive value causes PgmSvr.exe to insert a SLEEP of the specified number of seconds after it has completed processing the current request and before it sets the **PgmSvrCompleted** value to 1.

A negative value causes PgmSvr.exe to insert a WAIT of the specified number of seconds after it has completed processing the current request and before it sets the **PgmSvrCompleted** value to 1.

A SLEEP consumes no CPU cycles and causes PgmSvr.exe to become non-responsive.

A WAIT consumes CPU cycles, but allows PgmSvr.exe to remain responsive. This indicator is not normally used.

**PgmSvrStatus**
Check this value when `{PgmSvr - Process}` is finished.

0 - Processing finished without errors.
1 - A problem exists within PgmSvr.exe itself - this should never happen.
2 - The process did not complete before the timeout value was reached.
3 - A previous process, which did not complete because the timeout value was reached, is still running - current process request is rejected (and an error message is shown).

**PgmSvrUnloadWait**
This defines a timeout value in seconds. If PgmSvr.exe is loaded and not used for this number of seconds, PgmSvr.exe unloads itself. Loading PgmSvr.exe takes an appreciable amount of time (on my machine around 1.6 seconds), so this mechanism provides a way of minimizing the number of times PgmSvr.exe must be reloaded.
A snail is returning home late at night and has to cut though a dark alley. As he is passing through, he is mugged by two slugs. Later on at the police station, the officer asks him, "Can you give me a description of the assailants?" The snail ponders this for a moment, and then replies, "Gee, it all happened so fast."
Variable Management Category

The primary purpose of the Data statement is to give names to constants; instead of referring to $Pi$ as $3.141592653589793$ at every appearance, the variable $Pi$ can be given that value with a Data statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of $Pi$ ever change.

- FORTRAN manual for Xerox Computers
Overview

The functions presented in this category are at the root of almost every other function in our Library. They are the foundation for everything we design, develop, build, and market. Without them, our Library, and yours, would not be possible. Period.

Without variables, and a way to manage them, Macro Express would not be an easy, or for that matter, practical, application to develop anything with other than simple, basic macros. Somewhere along the line, as your macros grow in scope and power, you will begin to use variables. And when you do, you will quickly find that you need a way to manage and control them.

Let us first examine the native Macro Express variable environment. It provides us with 99 variables named %T1% through %T99% for storing text strings; 99 variables named %N1% through %N99% for storing integers (whole numbers); and 99 variables named %D1% through %D99% for storing doubles (decimal type numbers). It also provides an additional 99 variables named %C1% through %C99% for storing Window Control variables and an unlimited number of Environment variables, for which you create your own names. Both of these latter types are specialty variables and are of no concern to the Variable Management functions presented here.

All three standard variable types are global in scope, meaning that their values can be changed from anywhere within your macro. To refresh your memory, a macro is a collection of one or more functions linked together via the Macro Run command to perform a specific task. Therefore, %T1% created in one function can be overwritten by %T1% in a different function, which can be changed by another function, ad infinitum. In the native Macro Express variable environment there is only a single %T1%, %N1%, and %D1% variable to be accessed by an unlimited number of functions.

If this sounds confusing, then let me illustrate the problem. We will create a function that sets a string and integer variable each with some value then we will call another function that uses the same two variables, but sets them with different values.

```plaintext
// Main function
Variable Set String %T1% "PGMacros" // T1 initialized to “PGMacros”
Variable Set Integer %N1% to 100 // N1 initialized to a value of 100
Macro Run: Next function // Call the Next function
Text Type: T1=%T1% // T1 is now “Joe Weinpert”
Text Type: N1=%N1% // N1 is now 999

// Next function
Variable Set String %T1% from Prompt // Enter “Joe Weinpert” as the T1 value
Variable Set Integer %N1% from Prompt // Enter 999 as the N1 value
Macro Return // Return to the Main function
```

Once the variables are set with values in another function, those values remain as they are when returning to the main function. This is what makes them global in scope. It simply means that any variable created anywhere in any function can be seen, accessed, changed, and otherwise used, from any other function in the same macro.
To be fair, the two built-in Macro Express commands, **Variable Save** and **Variable Restore** could have been used in the called function to preserve the values of %T1% and %N1%:

```plaintext
// Next function
Variable Save All Variables      // Save variables prior to overwriting
Variable Set String %T1% from Prompt    // Enter “Joe Weinpert” as the T1 value
Variable Set Integer %N1% from Prompt   // Enter 999 as the N1 value
Variable Restore All Variables     // Restore variables prior to returning
Macro Return        // Return to the Main function
```

However, you can only do this once without them being overwritten. In other words, once a set of variables is saved with the **Variable Save** command, they must be restored with the **Variable Restore** command prior to saving another set, or they will be overwritten by the second **Variable Save** command.

Variable Management functions resolve this issue by allowing you to save and restore up to sixteen sets of variables at any time without overwriting the variable set that you are currently using. In other words, you can have sixteen %T1%, %N1%, and %D1% variables stored concurrently in memory. Since Macro Express provides 297 variables, why in the world would anyone want 4,752 of them? Well, we do not. It is not the quantity that is important, but rather how the variable sets themselves are used. Very rarely do we ever use more than a dozen variables in any one of our functions. But we do use the same ones over and over again. It is how reusable functions are created.

Let us say for instance that we use %T9% for the function name in every one of our functions (and we do). Using the Variable Management functions, we can have one function call another, which calls another, and in turn, calls another ... up to sixteen times, or levels deep, before we run out of %T9%. We never again have to worry about which variable gets used in what function. And that is the beauty of the Variable Management functions.

Combining the **Macro Run** command, which is one of the best features of Macro Express, with our Variable Management functions gives you the power to create reusable functions that can be called at any time into any macro.
Reusable Macro Functions

Any repeatable task that you must perform on your computer is a candidate to be a macro. Any repeatable task that your macro must perform is a candidate to be a reusable function. Think of reusable functions as extending the Macro Express built-in commands to suit your requirements. Imagine having an unlimited library of commands, which you have personally designed, available at your fingertips to do almost any imaginable task.

The ability to create reusable functions is the true power of the Variable Management functions. Let us examine the following scenario: Say that your macro is required to retrieve the names of three different folders on your computer - the Windows folder, System folder, and the Temporary folder. Furthermore, you want to format the results so that the drive designator is uppercase, the path is lowercase, and there is no trailing backslash character i.e. C:\windows, C:\windows\system, and C:\windows\temp.

First let us do it without the Variable Management functions. You will find this function, along with the others, in the Examples section of the Variable Management category.

```plaintext
// { Variable Management Example A }

// Retrieve folders into the following variables:
//   T21 = Windows folder
//   T22 = System folder
//   T23 = Temporary folder

// Retrieve the Windows folder
Variable Set String %T10% from Environment Variable
Variable Modify String: Trim %T10%
Variable Modify String: Lowercase %T10%

// Set drive designator to uppercase
Variable Modify String: Copy Part of %T10% to %T11%
Variable Modify String: Delete Part of %T10%
Variable Modify String: Uppercase %T11%
Variable Modify String: Append %T10% to %T11%

// Remove trailing backslash character
Variable Modify String: Copy %T11% to %T12%
Variable Set Integer %N10% from Length of Variable %T12%
Variable Modify String: Copy Part of %T12% to %T13%
If Variable %T13% = "\"
  Variable Modify Integer: Dec (%N10%)
  Variable Modify String: Copy Part of %T11% to %T11%
End If
Variable Modify String: Copy %T11% to %T21%

// Retrieve the System folder
Variable Set String %T10% from Environment Variable
Variable Modify String: Trim %T10%
Variable Modify String: Lowercase %T10%
```
Notice all the duplicated lines of code. This example is a good candidate for a reusable function that accepts a passed folder name, formats it, and then returns it. Here is the same task using Variable Management functions.

```plaintext
// { Variable Management Example B }

// Retrieve folders into the following variables:
// T21 = Windows folder
// T22 = System folder
// T23 = Temporary folder

// Retrieve the Windows folder and format it
Variable Set String %T21% from Environment Variable
Variable Modify String: Trim %T21%
Variable Modify String: Lowercase %T21%
Variable Modify String: Copy Part of %T21% to %T10%
```

Macro Return
Variable Modify String: Save %T21% to Clipboard
Macro Run: { Variable Management Example C }
Variable Set String %T21% from Clipboard

// Retrieve the System folder and format it
Variable Set String %T22% from Environment Variable
Variable Modify String: Save %T22% to Clipboard
Macro Run: { Variable Management Example C }
Variable Set String %T22% from Clipboard

Variable Set String %T23% from Environment Variable
Variable Modify String: Save %T23% to Clipboard
Macro Run: { Variable Management Example C }
Variable Set String %T23% from Clipboard
Macro Return

And here is the formatting function. Notice how the same variables are used in both functions without interfering with each other because they are saved upon entering and restored prior to exiting.

// { Variable Management Example C }

// Save the current variables and read from the clipboard
Macro Run: { Variables - Save }
Variable Set String %T20% from Clipboard
Variable Modify String: Trim %T20%
Variable Modify String: Lowercase %T20%

// Set drive designator to uppercase
Variable Modify String: Copy Part of %T20% to %T21%
Variable Modify String: Delete Part of %T20%
Variable Modify String: Uppercase %T21%
Variable Modify String: Append %T20% to %T21%

// Remove trailing backslash character
Variable Modify String: Copy %T21% to %T22%
Variable Set Integer %N10% from Length of Variable %T22%
Variable Modify String: Copy Part of %T22% to %T23%
If Variable %T23% = ";"
  Variable Modify Integer: Dec (%N10%)
End If

// Write the formatted string to the clipboard and restore variables
Variable Modify String: Save %T21% to Clipboard
Macro Run: { Variables - Restore }
Macro Return

There is no code duplication at all. You have just extended Macro Express by adding a command, in the form of a reusable function, which accepts a folder name, formats it, and then returns it without overwriting any variables.
Now let us take it to the next logical step and do the same task but with a main function (Example D) calling a function (Example E) to format the drive designator, which in turn calls another function (Example F) to remove the trailing backslash character.

    // { Variable Management Example D }

    // Retrieve folders into the following variables:
    //   T21 = Windows folder
    //   T22 = System folder
    //   T23 = Temporary folder

    // Retrieve the Windows folder and format it
    Variable Set String %T21% from Environment Variable
    Variable Modify String: Save %T21% to Clipboard
    Macro Run: { Variable Management Example E }
    Variable Set String %T21% from Clipboard

    // Retrieve the System folder and format it
    Variable Set String %T22% from Environment Variable
    Variable Modify String: Save %T22% to Clipboard
    Macro Run: { Variable Management Example E }
    Variable Set String %T22% from Clipboard

    // Retrieve the Temporary folder and format it
    Variable Set String %T23% from Environment Variable
    Variable Modify String: Save %T23% to Clipboard
    Macro Run: { Variable Management Example E }
    Variable Set String %T23% from Clipboard
    Macro Return

    // { Variable Management Example E }

    // Save the current variables and read from the clipboard
    Macro Run: { Variables - Save }
    Variable Set String %T20% from Clipboard
    Variable Modify String: Trim %T20%
    Variable Modify String: Lowercase %T20%

    // Set drive designator to uppercase
    Variable Modify String: Copy Part of %T20% to %T21%
    Variable Modify String: Delete Part of %T20%
    Variable Modify String: Uppercase %T21%
    Variable Modify String: Append %T20% to %T21%

    // Remove trailing backslash character
    Variable Modify String: Save %T21% to Clipboard
    Macro Run: { Variable Management Example F }

    // Restore variables
    Macro Run: { Variables - Restore }
    Macro Return
// { Variable Management Example F }

// Save the current variables and read from the clipboard
Macro Run: { Variables - Save }
Variable Set String %T21% from Clipboard

// Remove trailing backslash character
Variable Modify String: Copy %T21% to %T22%
Variable Set Integer %N10% from Length of Variable %T22%
Variable Modify String: Copy Part of %T22% to %T23%
If Variable %T23% = "\\"
  Variable Modify Integer: Dec (%N10%)
  Variable Modify String: Copy Part of %T21% to %T21%
End If

// Write the formatted string to the clipboard and restore variables
Variable Modify String: Save %T21% to Clipboard
Macro Run: { Variables - Restore }
Macro Return

Congratulations, you have just created two reusable functions without overwriting any variables, one to format the drive designator and the other to remove a backslash character from any string passed to it.

Nothing to it, is there? Variables are saved and restored in a LIFO (last in, first out) sequence, which simply means that for every { Variables - Save } function you must use a { Variables - Restore } function to retrieve your variables.

On a 750 MHz computer with 256 mb RAM, saving and restoring a set of variables takes about 250 ms depending on the size of the text strings being saved. The Windows Registry allows for strings of up to 16,383 characters in length except in Windows '95, which only allows 4,095.
The Registry

So how do these functions actually work? When the PGM Functions Library™ was first installed on your computer, an area called Variables was created in your Registry within the Professional Grade Macros key. This area, or key, contains sixteen sub-keys numbered 1 through 16. And each one of these contains 297 values, which is enough to store a complete set of Macro Express variables %T1% through %T99%, %N1% through %N99%, and %D1% through %D99%. Pictured is a small portion of this area showing
a mere handful of variables in the section reserved for set number 1. Now look at the Name column in the right hand pane. The values are purposely named to match each Macro Express variable. The Type column tells you the type of variable being stored, REG_SZ for strings, REG_DWORD for integers, and REG_BINARY (not shown) for decimal numbers. And of course the Data column shows the actual value being stored.

The value shown here in the right hand pane (named Current) is a pointer whose sole purpose is to keep track of the last variable set number Saved or Restored, maintaining the integrity of the LIFO stack. Life for this value starts at zero. Each time a Save function is used it is incremented by one and decremented for each Restore function. If you attempt to go past sixteen, the pointer is reset to zero and you start all over again, which, I am sure you realize, results in overwriting the variables you saved sixteen levels ago. For all of the complicated functions and macros we have ever created, we have yet to cascade down past five levels without a Restore. But it is something for you to be aware of none-the-less.

Let us examine this LIFO process in a little more detail. The pointer is sitting at zero. Now you save the variables in Function A. The pointer is incremented and the variables are then written to area 1. Now Function A calls Function B which saves variables also. The pointer is incremented again and the variables are written to area 2. This pattern continues on through to Function D, where, after saving variables and doing whatever else it was designed to do, it restores variables as its last act, which in turn decrements the pointer back to 3. Now you are back to Function C, which does the same thing as function D; restores variables, decrements the pointer (back to 2) and returns to Function B where it was called from. The same pattern of events occurs in Function B and Function A. In the end, all variables have been restored in the opposite manner in which they were saved and the pointer is back at zero.
Whew! Here is an example of what I just said. By the way, the Save and Restore functions handle the incrementing and decrementing of the pointer automatically. It is not something that you need to do or even keep track of, however, it is good to know how it all works, eh?

```cpp
// Pointer begins at zero
Save Function A variables  // Pointer now equals 1
Save Function B variables  // Pointer now equals 2
Save Function C variables  // Pointer now equals 3
Save Function D variables  // Pointer now equals 4
Restore Function D variables // Pointer is back to 3
Restore Function C variables // Pointer is back to 2
Restore Function B variables // Pointer is back to 1
Restore Function A variables // Pointer is back to zero
```
Saving and Restoring

The Save and Restore functions are designed to work as matched pairs. For every Save, you need a Restore. This should be obvious, but if you miss one, you will be Restoring variables that you did not expect. Surprise! The way to handle this is to have the first command line in any function be `{ Variables - Save }` and the last line `{ Variables - Restore }`, which is a matched pair. In other words, and with very few exceptions, variables are saved upon entering a function and restored just prior to exiting. Think in terms of sets.

You can, of course, save and restore variables as many times as you wish in any function. You are not limited to just the beginning and end of a function. Additionally, you do not need to erase variables in the Registry areas after restoring because they are completely overwritten with new values during the next save process. Let us walk through a small example.

```plaintext
Variable Set String %T1% "AaBbCc"   // First, set string variable T1 to be “AaBbCc"
Macro Run: { Variables - Save }    // Now save variables
Variable Set String %T1% "DdEeFf"   // Let us change T1 to be “DdEeFf”
Macro Run: { Variables - Save }    // Save variables again
Variable Set String %T1% "GgHhIi"   // Finally, change T1 to be “GgHhIi”
```

At this point, %T1% has been changed three times in the same function and is currently set to “GgHhIi”. Did you notice that we did not need to clear or erase any of the Registry areas prior to saving? Now let us restore variables.

```plaintext
Macro Run: { Variables - Restore }    // T1 is now “DdEeFf”. Let us restore again
Macro Run: { Variables - Restore }    // T1 is back to the value that we started with, which was “AaBbCc”
```

All the variables are restored. Could this small example have been accomplished using the built-in Macro Express commands `Variable Save` and `Variable Restore`? No, because they are limited to storing and retrieving only a single set of variables at a time. Look and you will see the difference:

```plaintext
Variable Set String %T1% "AaBbCc"   // First, set string variable T1 to be “AaBbCc”
Variable Save All Variables   // Now save variables with the built-in command
Variable Set String %T1% "DdEeFf"   // Let us change T1 to be “DdEeFf”
Variables Save All Variables     // Save variables again
Variable Set String %T1% “GgHhIi”   // Finally, change T1 to be “GgHhIi”
```

At this point, %T1% has been changed three times and is currently set to “GgHhIi”. Now let us restore variables.

```plaintext
Variable Restore All Variables     // T1 is now “DdEeFf” as you would expect, however, let us restore again
Variable Restore All Variables     // T1 is still “DdEeFf”
```

What happened to “AaBbCc”? It was overwritten in memory by the second `Variables Save` command.
Functions

The Variable Management category is composed of ten functions, which handle the saving and restoring of variables to and from the Registry. There are different pairs, or sets, of save and restore variables for various situations. The difference between them is only which of the 297 available variables are dealt with. This was done for the sake of speed.

- **{ Variables - Save }**: Saves all 297 variables T1-T99, N1-N99, and D1-D99
- **{ Variables - Restore }**: Restores all 297 variables
- **{ Variables - Save SI }**: Saves all 198 string and integers T1-T99 and N1-N99
- **{ Variables - Restore SI }**: Restores all 198 string and integers
- **{ Variables - Save 25 }**: Saves the first 25 of all types T1-T25, N1-N25, and D1-D25
- **{ Variables - Restore 25 }**: Restores the first 25 of all types
- **{ Variables - Save DT }**: Saves the first 25 string and integers T1-T25, N1-N25, D1-D25
- **{ Variables - Restore DT }**: Restores the first 25 string and integers
- **{ Variables - Reset Registry Values }**: Resets all variables in all areas to their default empty values
- **{ Variables - Reset Current Level }**: Resets the area pointer back to zero

It was previously mentioned that saving and restoring all 297 variables took only 250 ms to accomplish. This is very fast indeed, however, it is even faster to save and restore fewer variables. If a function is not going to overwrite any decimal variables, why save them? In fact, it makes sense to only save variables that will be overwritten. Although this is technically accurate, it is impractical because of the maintenance involved as you modify the function. So, we have supplied you with a few different Save and Restore functions that work with subsets of the 297 variables.

**{ PGM Function }** does not work with any of the Variable Management functions - all you get is an error message if you try to use it.
{ Variables - Save }

Increments the pointer to the next available variable area in the Registry (1 through 16), then saves all 297 variables (%T1% through %T99%, %N1% through %N99%, and %D1% through %D99%) to that area.

The pointer is a value stored in the Registry, which is incremented each time the { Variables - Save } function is called and decremented with every call to its matching { Variables - Restore } function. Be sure to keep them paired.

Syntax
{ Variables - Save }

Returns
Nothing.

Arguments
None.

Calls
{ PGM }
{ PGM Setup }

Example
The following example saves all 297 variables at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

Macro Run: { Variables - Save } // Save the current variables
Variable Set String %T99% "AaBbCc" // Set T99 to “AaBbCc”
Variable Set Integer %N99% to 100 // Set N99 to 100
Variable Set Decimal %D99% to 67.98 // Set D99 to 67.98
Macro Run: { Variables - Save } // Now save them
Variable Set String %T99% "DdEeFf" // Overwrite T99 with “DdEeFf”
Variable Set Integer %N99% to 200 // Overwrite N99 with 200
Variable Set Decimal %D99% to 12.34 // Overwrite D99 with 12.34
Macro Run: { Variables - Restore } // Now restore them
Text Type: T99=%T99% // T99 is back to “AaBbCc”
Text Type: N99=%N99% // N99 is back to 100
Text Type: D99=%D99% // D99 is back to 67.98
Macro Run: { Variables - Restore } // Restore the original values saved at the start

See Also
{ Variables - Restore }
{ Variables - Restore }

The companion function (be sure to keep them paired) to { Variables - Save }, it restores all 297 variables (%T1% through %T99%, %N1% through %N99%, and %D1% through %D99%) saved in the Registry area (1 through 16) pointed to by the pointer, then decrements the pointer. Er, get the point? If it is decremented down to zero it will remain there until the next Save function is called. In other words, when there is not anything to restore, you could call this function until the cows come home and nothing will change. Did someone mention bovines?

A farmer is milking his cow. As he is milking, a fly comes along and flies into the cows ear. A little bit later, the farmer notices the fly in the milk. The farmer looks up and says, "Hmmm. In one ear, and out the udder."

Syntax

{ Variables - Restore }

Returns

Nothing.

Arguments

None.

Calls

{ PGM }
{ PGM Setup }

Example

The following example saves all 297 variables at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

```
Macro Run: { Variables - Save }  // Save the current variables
Variable Set String %T99% "AaBbCc"  // Set T99 to “AaBbCc”
Variable Set Integer %N99% to 100  // Set N99 to 100
Variable Set Decimal %D99% to 67.98  // Set D99 to 67.98
Macro Run: { Variables - Save }  // Now save them
Variable Set String %T99% "DdEeFf"  // Overwrite T99 with “DdEeFf”
Variable Set Integer %N99% to 200  // Overwrite N99 with 200
Variable Set Decimal %D99% to 12.34  // Overwrite D99 with 12.34
Macro Run: { Variables - Restore }  // Now restore them
Text Type: T99=%T99%  // T99 is back to “AaBbCc”
Text Type: %N99%=%N99%  // N99 is back to 100
Text Type: %D99%=%D99%  // D99 is back to 67.98
Macro Run: { Variables - Restore }  // Restore the original values saved at the start
```

See Also

{ Variables - Save }
{ Variables - Save SI }

Increments the pointer to the next available variable area in the Registry (1 through 16), then saves all 198 string and integer variables (%T1% through %T99% and %N1% through %N99%) to that area. If your function is not using, or overwriting, decimal type variables this function is faster than { Variables - Save }.

The pointer is a value stored in the Registry, which is incremented each time the { Variables - Save SI } function is called and decremented with every call to its matching { Variables - Restore SI } function. Be sure to keep them paired.

Syntax
{ Variables - Save SI }

Returns
Nothing.

Arguments
None.

Calls
{ PGM }
{ PGM Setup }

Example
The following example saves all 198 string and integer variables at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

Macro Run: { Variables - Save SI }  // Save the current variables
Variable Set String %T99% "AaBbCc"  // Set T99 to “AaBbCc”
Variable Set Integer %N99% to 100  // Set N99 to 100
Macro Run: { Variables - Save SI }  // Now save them
Variable Set String %T99% "DdEeFf"  // Overwrite T99 with “DdEeFf”
Variable Set Integer %N99% to 200  // Overwrite N99 with 200
Macro Run: { Variables - Restore SI }  // Now restore them
Text Type: T99=%T99%  // T99 is back to “AaBbCc”
Text Type: N99=%N99%  // N99 is back to 100
Macro Run: { Variables - Restore SI }  // Restore the original values saved at the start

See Also
{ Variables - Restore SI }
{ Variables - Restore SI }

The companion function (be sure to keep them paired) to { Variables - Save SI }, it restores all 198 variables (%T1% through %T99% and %N1% through %N99%) saved in the Registry area (1 through 16) pointed to by the pointer, then decrements the pointer. If it is decremented down to zero it will remain there until the next Save function is called. In other words, when there is not anything to restore, you could call this function until you know what freezes over and nothing will change.

Syntax
{ Variables - Restore SI }

Returns
Nothing.

Arguments
None.

Calls
{ PGM }
{ PGM Setup }

Example
The following example saves all 198 string and integer variables at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

```
Macro Run: { Variables - Save SI }    // Save the current variables
Variable Set String %T99% "AaBbCc"   // Set T99 to “AaBbCc”
Variable Set Integer %N99% to 100    // Set N99 to 100
Macro Run: { Variables - Save SI }    // Now save them
Variable Set String %T99% "DdEeFf"   // Overwrite T99 with “DdEeFf”
Variable Set Integer %N99% to 200     // Overwrite N99 with 200
Macro Run: { Variables - Restore SI }   // Now restore them
Text Type: T99=%T99%                  // T99 is back to “AaBbCc”
Text Type: N99=%N99%                  // N99 is back to 100
Macro Run: { Variables - Restore SI }   // Restore the original values saved at the start
```

See Also
{ Variables - Save SI }
{ Variables - Save 25 }

Increments the pointer to the next available variable area in the Registry (1 through 16), then saves 50 string and integer variables (%T1% through %T25% and %N1% through %N25%) to that area. If you only need to work with a handful of string and integer variables, and if you are not using, or overwriting, decimal type variables, this function is faster than both {Variables - Save} and {Variables - Save SI}. A majority of the functions within our Library make use of this particular one because rarely do reusable functions require more than 25 strings and integers. There are exceptions though!

The pointer is a value stored in the Registry, which is incremented each time the {Variables - Save 25} function is called and decremented with every call to its matching {Variables - Restore 25} function. Be sure to keep them paired.

Syntax
{ Variables - Save 25 }

Returns
Nothing.

Arguments
None.

Calls
{ PGM }
{ PGM Setup }

Example
The following example saves 50 string and integer variables (%T1% through %T25% and %N1% through %N25%) at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

```
Macro Run: { Variables - Save 25 }   // Save the current variables
Variable Set String %T25% "AaBbCc"      // Set T25 to “AaBbCc”
Variable Set Integer %N25% to 100         // Set N25 to 100
Macro Run: { Variables - Save 25 }       // Now save them
Variable Set String %T25% "DdEeFf"       // Overwrite T25 with “DdEeFf”
Variable Set Integer %N25% to 200          // Overwrite N25 with 200
Macro Run: { Variables - Restore 25 }     // Now restore them
Text Type: T25=%T25%                      // T25 is back to “AaBbCc”
Text Type: N25=%N25%                      // N25 is back to 100
Macro Run: { Variables - Restore 25 }     // Restore the original values saved at the start
```

See Also
{ Variables - Restore 25 }
{ Variables - Restore 25 }

The companion function (be sure to keep them paired) to { Variables - Save 25 }, it restores 50 variables (%T1% through %T25% and %N1% through %N25%) saved in the Registry area (1 through 16) pointed to by the pointer, then decrements the pointer. If it is decremented down to zero it will remain there until the next Save function is called. In other words, when there is not anything to restore, you could call this function until politicians stop spending your money and nothing will change.

Syntax

{ Variables - Restore 25 }

Returns

Nothing.

Arguments

None.

Calls

{ PGM }
{ PGM Setup }

Example

The following example saves 50 string and integer variables (%T1% through %T25% and %N1% through %N25%) at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

Macro Run: { Variables - Save 25 }    // Save the current variables
Variable Set String %T25% "AaBbCc"    // Set T25 to “AaBbCc"
Variable Set Integer %N25% to 100    // Set N25 to 100
Macro Run: { Variables - Save 25 }    // Now save them
Variable Set String %T25% "DdEeFf"    // Overwrite T25 with “DdEeFf”
Variable Set Integer %N25% to 200    // Overwrite N25 with 200
Macro Run: { Variables - Restore 25 }   // Now restore them
Text Type: T25=%T25%    // T25 is back to “AaBbCc”
Text Type: N25=%N25%    // N25 is back to 100
Macro Run: { Variables - Restore 25 }   // Restore the original values saved at the start

See Also

{ Variables - Save 25 }
{ Variables - Save DT }

Increments the pointer to the next available variable area in the Registry (1 through 16), then saves only the first 25 of all variable types (%T1% through %T25%, %N1% through %N25%, and %D1% through %D25%) to that area. If you are working with reusable functions that use, or overwrite, only a handful of variables then use this function instead of the full { Variables - Save } function. It is much faster.

The pointer is a value stored in the Registry, which is incremented each time the { Variables - Save DT } function is called and decremented with every call to its matching { Variables - Restore DT } function. Be sure to keep them paired.

Syntax

{ Variables - Save DT }

Returns

Nothing.

Arguments

None.

Calls

{ PGM }

{ PGM Setup }

Example

The following example saves the first 25 of all variable types at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

```
Macro Run: { Variables - Save DT }  // Save the current variables
Variable Set String %T25% "AaBbCc"  // Set T25 to “AaBbCc”
Variable Set Integer %N25% to 100  // Set N25 to 100
Variable Set Decimal %D25% to 67.98  // Set D25 to 67.98
Macro Run: { Variables - Save DT }  // Now save them
Variable Set String %T25% "DdEeFf"  // Overwrite T25 with “DdEeFf”
Variable Set Integer %N25% to 200  // Overwrite N25 with 200
Variable Set Decimal %D25% to 12.34  // Overwrite D25 with 12.34
Macro Run: { Variables - Restore DT }  // Now restore them
Text Type: T25=%T25%  // T25 is back to “AaBbCc”
Text Type: N25=%N25%  // N25 is back to 100
Text Type: D25=%D25%  // D25 is back to 67.98
Macro Run: { Variables - Restore DT }  // Restore the original values saved at the start
```

See Also

{ Variables - Restore DT }
{ Variables - Restore DT }

The companion function (be sure to keep them paired) to { Variables - Save DT }, it restores the first 25 of all variable types (%T1% through %T25%, %N1% through %N25%, and %D1% through %D25%) saved in the Registry area (1 through 16) pointed to by the pointer, then decrements the pointer. Sound familiar? If it is decremented down to zero it will remain there until the next Save function is called. In other words, when there is not anything to restore, you could call this function until peanut butter and jelly are incompatible and nothing will change.

Syntax
{ Variables - Restore DT }

Returns
Nothing.

Arguments
None.

Calls
{ PGM }
{ PGM Setup }

Example
The following example saves the first 25 of all variable types at the beginning of the function and then restores them just prior to the end. In between it manipulates some variables so you can see how they are overwritten and restored.

Macro Run: { Variables - Save DT }   // Save the current variables
Variable Set String %T25% "AaBbCc"  // Set T25 to “AaBbCc”
Variable Set Integer %N25% to 100    // Set N25 to 100
Variable Set Decimal %D25% to 67.98   // Set D25 to 67.98
Macro Run: { Variables - Save DT }   // Now save them
Variable Set String %T25% "DdEeFf"  // Overwrite T25 with “DdEeFf”
Variable Set Integer %N25% to 200    // Overwrite N25 with 200
Variable Set Decimal %D25% to 12.34   // Overwrite D25 with 12.34
Macro Run: { Variables - Restore DT } // Now restore them
Text Type: T25=%T25%                  // T25 is back to “AaBbCc”
Text Type: N25=%N25%                  // N25 is back to 100
Text Type: D25=%D25%                  // D25 is back to 67.98
Macro Run: { Variables - Restore DT } // Restore the original values saved at the start

See Also
{ Variables - Save DT }
{ Variables - Reset Registry Values }

This handy little utility sets the pointer back to zero and then resets all variables in all areas (1 through 16) back to their default zero or empty values. We use it not only for debugging during development, but also within the PGM Functions Library™ installer to create the Variable portion of the Professional Grade Macros Registry key.

Syntax

{ Variables - Reset Registry Values }

Returns

Nothing.

Arguments

None.

Calls

{ PGM }
{ PGM Setup }

Example

The following example resets all variables in the save area back to their zero or empty values. Look, I know it is not much of an example, but there really is not a whole lot to know about this one, now is there?

// Have a lot of garbage accumulated in here?
Macro Run: { Variables - Reset Registry Values } // Now you do not

See Also

{ Variables - Reset Current Level }
{ Variables - Reset Current Level }

This utility sets the pointer back to zero without changing any of the stored variables. Again, we use it for debugging and testing functions during development, so we figured that you could use it too.

**Syntax**

{ Variables - Reset Current Level }

**Returns**

Nothing.

**Arguments**

None.

**Calls**

{ PGM }

{ PGM Setup }

**See Also**

{ Variables - Reset Registry Values }
Well, well …

Two cows were talking in the field. One cow says, "Have you heard about the Mad Cow disease that's going around?"

The other cow answers, "Yeah, makes you glad you're a kangaroo, doesn't it?"
There are two ways to write error-free programs. Only the third one works.
- Unknown
Overview

Have you ever had a problem where launching and focusing a program on one computer works each and every time, but on a different one it does not, unless you change the macro?

Launching a program with Macro Express is a simple task, just use the Program Launch command. If the program is there, it will launch. However, gaining reliable focus of the just-launched program is something else altogether. If you are creating macros for other people, or you have multiple computers, there are too many things that are outside of your control, which can spell success or failure - lack of memory, number of tasks currently running, processor speed, small caches, different operating systems, etc. - all affect your macro’s ability to run.

Efficient settings on one computer may be deficient on another, so how can you be sure that a program will launch and gain focus whichever computer it runs on? You cannot, not without customizing the macro for each computer. Meaning, of course, that you will be managing separate macros for each person or having them adjust the macro you supplied until it runs reliably. As an alternative, you could design for the worst case scenario, which is an awe-inspiring idea for the guy with the worst case computer, but will drive your other users absolutely bananas.

The real solution is to use a complete function in place of the single line Program Launch command. And that my friends, is what the functions in the category are all about. Program Operations functions are designed to launch, focus, and terminate programs, in a reliable manner, on computers with differing capabilities.

But wait, there is more! They can, at your discretion, also keep a log of specific events when attempting to launch a program unattended. This can be used as a debugging tool to determine where exactly the launch failed … and why, or simply as a history log.
The Madness Behind the Method

Here are the steps necessary to reliably launch a program and have it gain focus.

1. Launch the program.
2. Test for a successful launch.
3. Test if the program’s window title is running.
4. Activate the program window title.
5. Test for a successful window activation.

In order to accomplish these steps you have to tell Macro Express more than just which program to launch. What it really needs to know is:

1. Name of the program to launch.
2. Where the program is located.
3. Command line arguments to be passed to the program (if any).
4. Window title of the program.
5. Whether or not to use an exact match for the window title.
6. How the window is to be launched (normal, minimized, or maximized).
7. How many seconds to delay after launching.
8. How many times to test for a successful launch, window title, and window activation.
9. How much time to delay between each test.

Although it seems like a lot of information, it really is not. In fact, except for items 8 and 9, you are already familiar with them. Take a look:
1. Name of the program to launch.
2. Where the program resides on disk.
3. Command line parameters to pass.
4. How the window is to be launched.
5. How many seconds to delay.
Items 8 and 9 cannot be seen in the above pictures because they are not built-in features to Macro Express. They are, however, a very important part of the Program Operations functions. Item 8 is a verification loop counter and item 9 contains a value for delaying each loop through the counter.

The Program Operations functions work in the following manner. They:

Gather the above nine bits of information from Registry variables that you set.
Test them to be sure they make sense and are valid.
Perform whatever operation the information specifies, such as launch, activate, etc.
Verify each phase by looping through a repeat cycle (delaying a certain amount of time prior to the next cycle) until either the phase is successful or the maximum number of loops has been reached. In case of the latter, an error is returned to your calling function so you are not left hanging out to dry. This error can, and should be, tested for so appropriate action can be taken on your part.

Here is an example macro that will launch Microsoft Word. Create three variables to hold the program name, location (home folder), and window title. Write each to the Parameters section of the Registry (see the Registry discussion below). Call the \{Program-Launch\} function then test the error flag when it finishes. If it equals zero then Microsoft Word was launched successfully and has gained focus, otherwise something went wrong.
Notice how we only had to specify three variables %T1%, %T2%, and %T3%? What happened to the other six bits of information? Nothing. There was no need to set any variables for them because they have default values which are preset.

Item 3 is preset to blank string (no command line parameters to pass).
Item 5 is preset to use a partial window title match.
Item 6 is preset to launch the program normally (not minimized or maximized).
Item 7 is preset to a zero launch delay (uses Item 9 instead).
Item 8 is preset to 50 verification loops, or attempts.
Item 9 is preset to a 500 millisecond delay between loops.

The Program Operations functions give you 100% control over what these presets are. You can change them any time you wish, as many times as needed, whenever your particular situation warrants it.

What would this same function look like using { PGM Function }? Much shorter!

See { PGM Function } for details on using this function.
A History of Events

Something else that the Program Operations have is the ability to keep an historical log of important, designated events within a macro. This is especially handy for **unattended operations** when things simply do not go right. If you have an operation that runs unattended, say overnight, and it **inconsistently** fails, you can check the event log to see what is occurring to cause failure.

Here is what the event log, a simple text file, looks like after running the `{Launch Microsoft Word}` function from the previous section. Notice that four events were logged, the call to the `{Program - Launch}` function and three successful verification events that it performed on each phase; **launch**, **activate**, and **focus**.

![Event Log Example]

If Word were already running, the event log would show:

```
6/1/2003 2:35:25 AM { Program - Launch } - winword.exe
```

And if we entered a wrong program name:

```
6/1/2003 2:44:10 AM { Program - Launch } - winword.exe
6/1/2003 2:44:12 AM { Program - Launch } - winword.exe - Fatal error. The c:\program files\microsoft office\office10\winword.exe program file does not exist.
```

Each event is appended to the end of the log and contains a date and time stamp. It even records when I work into the earlier morning, eh? And even though you decide what the rest of the event string consists of, might I suggest that it is a good idea to include the function’s name in it.

Use event logging to not only record when a macro begins and ends, but for anything else in between. Record commands, variables, time, current folders, titles, windows, etc. There is no limit on what you may record. And best of all, recording will not slow your macro significantly, uhm, unless of course you log an event for every line in the macro.
In order to log an event, any event, you need only to write the event text string, say %T1%, to the Parameters section of the Registry (see the Registry discussion below) then call the \{Program - Log Event\} function. For example:

```plaintext
// Log an event.
Variable Set String %T1% "This is my event text string."  // Set T1 to an event string
Write Registry String: "Event"  // Now write to the Registry
Macro Run: { Program - Log Event }  // Run the event log function
```

And this is what gets recoded in the log:

```
6/1/2003 3:15:12 AM This is my event text string.
```
So how do these functions actually work? Or, more to the point, how do you work with them? When the PGM Functions Library™ was first installed on your computer, an area called Parameters was created in your Registry within the Professional Grade Macros key. This area, or key, contains many values, eighteen of which are used to communicate with the Program Operations functions and are shown here. You communicate with them via the Write Registry String, Write Registry Integer, Read Registry String, and Read Registry Integer commands. The Name column, in the right hand pane, is the value name. The Type column is the type of variable being stored, REG_SZ for strings and REG_DWORD for integers. And of course the Data column shows the actual value being stored.

The name of the event log, which defaults to PGM Event.log, is stored in another section of the Professional Grade Macros key called Files. You can, of course change its name to something else entirely, however, its location will always be in the sub-folder called Data, which is located wherever the PGM Functions Library™ was installed.
Substituting the nine steps necessary to reliably launch a program and have it gain focus (shown earlier) with their actual Registry value names will give you a clear picture as to their relationship. Of course, as before, items 8 and 9 cannot be shown in the pictures, however, item 8, the verification loop counter is `VerifyLoops`, and item 9, the value for delaying each loop, is `VerifyDelay`.

1. `ProgramName`
2. `ProgramFolder`
3. `ProgramParameters`
4. `LaunchState`
7. LaunchDelay

If program fails to launch in the specified time:
- Stop Macro
- Continue Macro
- Prompt for Action

4. WindowTitle

If Commands

- If File Exists
- If File Ready
- If File does not Exist
- If File is not Ready
- If Folder Exists
- If Folder does not Exist
- If Window on Top
- If Window not on Top
- If Window Running
- If Window not Running
- If Program on Top
- If Program not on Top
- If Program Running
- If Program not Running
- If Clipboard Text Equals
- If Clipboard Contains
- If Mouse Cursor
- If Macro Enabled
- If Macro Disabled
- If Not Mouse Cursor

Window Title: Microsoft Word
Select
- Exact Match
- Partial Match

5. ExactMatch?
The Program Operations values can be grouped into six categories:

1. Program and folder values.
2. Launch values.
3. Verification values.
4. Return values.
5. Event values.
6. Termination values.

Program and Folder Values

**ProgramName**
The name of the program you want to launch or terminate. You can specify the program name by itself or as a full path name.

```
Variable Set String %T1% "winword.exe" // Set T1 to program
Write Registry String: "ProgramName" // Write T1 to the Registry
```

Or:

```
Variable Set String %T1% "c:\program files\microsoft office\office10\winword.exe" // Set T1 to full path
Write Registry String: "ProgramName" // Write T1 to the Registry
```

**ProgramFolder**
The folder containing the program to be launched. If used, it will be added with the *ProgramName* value internally, therefore, using a full path for the *ProgramName* and a value here is a no-no. You can use a full path for the *ProgramName* value and an empty value here, or a simple program name with the folder specified here.

**Good:**

```
Variable Set String %T1% "c:\program files\microsoft office\office10\winword.exe" // Set T1 to full path
Write Registry String: "ProgramName" // Write T1 to the Registry
```

**Good:**

```
Variable Set String %T1% "winword.exe" // Set T1 to program
Variable Set String %T2% "c:\program files\microsoft office\office10" // Set T2 to folder
Write Registry String: "ProgramName" // Write T1 to the Registry
Write Registry String: "ProgramFolder" // Write T2 to the Registry
```

**Bad:**

```
Variable Set String %T1% "c:\program files\microsoft office\office10\winword.exe" // Set T1 to full path
Variable Set String %T2% "c:\program files\microsoft office\office10" // Set T2 to folder
Write Registry String: "ProgramName" // Write T1 to the Registry
Write Registry String: "ProgramFolder" // Write T2 to the Registry
```
Program Parameters
This is a command line argument string to pass to the program. Most programs allow you to pass data to them on the same line when they are engaged. For instance, Microsoft Word will accept a file name to load when it launches.

Variable Set String %T1% "winword.exe" // Set T1 to program
Variable Set String %T2% "c:\program files\microsoft office\office10" // Set T2 to folder
Variable Set String %T3% "c:\my documents\my word.doc" // Set T3 to command line
Write Registry String: "ProgramName" // Write T1 to the Registry
Write Registry String: "ProgramFolder" // Write T2 to the Registry
Write Registry String: "ProgramParameters" // Write T3 to the Registry

Notice how the command line argument is wrapped in a pair of quote marks as part of the string? It is always a good idea to do this for any command line argument that contains spaces. The outside quote marks that you see are the ones added by Macro Express. As you can see by the other strings, this is normal.

WindowTitle
The window title of the program that you are launching and/or setting focus to.

Variable Set String %T4% "Microsoft Word" // Set T4 to window title
Write Registry String: "Window Title" // Write T4 to the Registry

ExactMatch?
A setting that determines if the WindowTitle value is to be an exact match (true), or partial match (false). 1 = true and 0 = false. The default is 0, which is a partial match.

Variable Set Integer %N4% to 0 // Set N4 to partial match
Write Registry Integer: "ExactMatch?" // Write N4 to the Registry
Launch Values

LaunchState
A program can be launched in one of three states: normal, minimized, or maximized and can gain focus in whichever state that it is launched. In other words, even if a program is minimized to the taskbar when it is launched, it could still be the active, or on-top program. You choose the launch state by setting this value to Normal (the default state), Minimized, or Maximized.

```
Variable Set String %T10% "Normal"    // Set T10 to normal
Write Registry String: "LaunchState"    // Write T10 to the Registry
```

LaunchDelay
The number of seconds to wait after first launching a program. The default value is 0 (zero) because the {Program-Launch} function tests for a successful launch within the verification loop after a program is instructed to launch, therefore this delay is not usually needed. Delays are handled inside the verification loop by the VerifyDelay value. However, this delay value can still be used whenever you wish. A valid entry would range between 0 and the MaxLaunchDelay value of 30 seconds. If a value is entered outside of this range, it is changed back to the default value, but only if the ValidateParameters value is set to true.

```
Variable Set Integer %N10% to 0         // Set N10 to a zero delay
Write Registry Integer: "LaunchDelay"  // Write N10 to the Registry
```

MaxLaunchDelay
This sets the maximum possible LaunchDelay value in seconds. The {Program-Validate Launch Values} uses it to determine if the LaunchDelay value entered is greater than the value here. The default value is 30 seconds.

```
Variable Set Integer %N11% to 30        // Set N11 to max 30 seconds
Write Registry Integer: "MaxLaunchDelay" // Write N11 to the Registry
```
Verification Values

VerifyLoops
The number of loops or attempts to verify that a particular phase was successfully (launch, focus, terminate, etc.). The default value is 50. A valid entry would range between 5 and the MaxVerifyLoops value of 240. If a value is entered outside of this range, it is changed back to the default value, but only if the ValidateParameters value is set to true.

Variable Set Integer %N11% to 50 // Set N11 to 50 loops
Write Registry Integer: "VerifyLoops"  // Write N11 to the Registry

VerifyDelay
The number of milliseconds to wait before cycling through the verify loop again. The default value is 500 milliseconds, or one ½ of one second. A valid entry would range between 50 milliseconds and the MaxVerifyDelay value of 15,000 milliseconds (15 seconds). If a value is entered outside of this range, it is changed back to the default value, but only if the ValidateParameters value is set to true.

Variable Set Integer %N12% to 500 // Set N12 to 500 milliseconds
Write Registry Integer: "VerifyDelay"  // Write N12 to the Registry

MaxVerifyLoops
This sets the maximum possible VerifyLoops value. The Program - Validate Verification Values uses it to determine if the VerifyLoops value entered is greater than the value here. The default value is 240.

Variable Set Integer %N13% to 240 // Set N13 to max 240 loops
Write Registry Integer: "MaxVerifyLoops"  // Write N13 to the Registry

MaxVerifyDelay
Sets the maximum possible VerifyDelay value in milliseconds. The Program - Validate Verification Values uses it to determine if the VerifyDelay value entered is greater than the value here. The default value is 15,000 milliseconds (15 seconds).

Variable Set Integer %N14% to 15000 // Set N14 to max 15000 milliseconds
Write Registry Integer: "MaxVerifyDelay"  // Write N14 to the Registry

ValidateParameters?
A setting that determines if certain values passed to Program Operations functions are to be validated prior to being used within a particular function. 1 = true and 0 = false. The default is 1, which means to validate the values with one of the three Program - Validate... functions, depending on which operation is being performed (launch, focus, or terminate). The following values are validated: LaunchState, LaunchDelay, VerifyLoops, VerifyDelay, and TerminateTimeout.

Variable Set Integer %N15% to 1 // Set N15 to validate values
Write Registry Integer: "ValidateParameters?"  // Write N15 to the Registry
Return Values

**ReturnError?**
This value will tell you if an error has occurred in one of the Program Operations functions. If the value is 1 then an error occurred, and a 0 means that the function ran fine. This value should be tested after a function returns from processing so appropriate action can be taken by your macro. If an error occurs and if event logging is enabled (*LogEvent?*), then the actual error will be contained in the event log.

```
Macro Run: { Program - Launch } // Run the program launch function
Read Registry Integer: "ReturnError?" // Check if there was an error
If Variable %N1% = 1
  Text Box Display: Error! // If so, then prompt the user
End If
```
Event Values

Event
This value contains a string to append to the event log named in the EventLog value, but only if event logging (LogEvent?) is enabled.

```plaintext
// Log function call.
If Macro "( Program - Log Event )" Enabled
    Variable Set String %T9% "( Program - Launch ) - winword.exe"
    Write Registry String: "Event"
    Macro Run: { Program - Log Event }
End If
```

LogEvent?
A setting that determines if event logging is enabled or disabled. 1 = true and 0 = false. The default is 1, which means subsequent events will be appended to the event log.

```plaintext
Variable Set Integer %N16% to 1 // Set N16 to log events
Write Registry Integer: "LogEvents?" // Write N16 to the Registry
```

EventLog
Contains the name of the event log and can be changed at any time, for any reason. The default name is PGM Event.log.

```plaintext
Variable Set String %T10% "PGM Event.log" // Set T10 to the event log name
Write Registry String: "EventLog" // Write T10 to the Registry
```
Termination Values

TerminateTimeout
This is the number of milliseconds to wait after telling a program to terminate for Macro Express to timeout and give an error - only if termination does not occur. The default value is 150 milliseconds, however, the value is not needed because the {Program - Terminate} function tests for a successful termination within the verification loop after a program is instructed to stop. Delays are handled inside the verification loop by the VerifyDelay value. However, this delay value can still be used whenever you wish. A valid entry would range between 0 and the MaxTerminateTimeout value of 1,000 milliseconds (1 second). If a value is entered outside of this range, it is changed back to the default value, but only if the ValidateParameters value is set to true.

```
Variable Set Integer %N10% to 0 // Set N10 to a zero timeout
Write Registry Integer: "TerminateTimeout"    // Write N10 to the Registry
```

MaxTerminateTimeout
This Registry value contains the maximum possible TerminateTimeout value in milliseconds. The {Program - Validate Termination Values} uses it to determine if the TerminateTimeout value entered is greater than the value here. The default value is 1,000 milliseconds.

```
Variable Set Integer %N11% to 1000 // Set N11 to max 1000 milliseconds
Write Registry Integer: "MaxTerminateTimeout " // Write N11 to the Registry
```
Examples

You will find these example functions in the Examples section of the Program Operations category.

Example A - You want to turn event logging off.

```
// { Program Operations Example A }
Variable Set Integer %N1% to 0   // Set N1 to zero
Write Registry Integer: "LogEvents?"   // Turn it off
```

Example B - You want to set all the Program Operations values in the Registry back to their default value.

```
// { Program Operations Example B }
Macro Run: { Program - Reset }  // Run the reset function
```

Example C - You want to be absolutely sure that the Notepad program gets launched and is minimized to the taskbar.

```
// { Program Operations Example C }
Macro Run: { Program - Reset }   // Run the reset function
Variable Set String %T1% "notepad.exe"   // Program name
Variable Set String %T2% "Notepad"   // Window title
Variable Set String %T3% "Minimized"   // Launch state
Write Registry String: "ProgramName"   // Write program name
Write Registry String: "WindowTitle"   // Write window title
Write Registry String: "LaunchState"   // Write launch state
Macro Run: { Program - Launch }   // Launch
Read Registry Integer: "ReturnError?"   // Test for error
If Variable %N1% = 1
  Macro Stop
End If
```

What would this same function look like using `{ PGM Function }`?

```
Macro Run: { Program - Reset }   // Run the reset function
Variable Set String %T1% "{ Program - Launch },notepad.exe, Notepad, , , Minimized"
Write Registry String: "PgmFunction"   // Write it to the Registry
Macro Run: { PGM Function }   // Call the function
Read Registry Integer: "ReturnError?"   // Test for error
If Variable %N1% = 1
  Macro Stop
End If
```
Example D - You have a couple of programs running and Notepad must gain focus.

```
// { Program Operations Example D }
Macro Run: { Program Operations Example C }   // Make sure the Notepad is already launched
// Launch a different program so we can switch focus.
Macro Run: { Program - Reset }   // Run the reset function
Variable Set String %T1% "calc.exe"    // Program name
Variable Set String %T2% "Calculator"    // Window title
Write Registry String: "ProgramName"    // Write program name
Write Registry String: "WindowTitle"    // Write window title
Macro Run: { Program - Launch }   // Launch
Read Registry Integer: "ReturnError?"   // Test for error
If Variable %N1% = 1
  Macro Stop
End If
Window Minimize: "Calculator"    // Minimize to the taskbar
// Set Focus to the Notepad.
Variable Set String %T1% "notepad.exe"    // Program name
Variable Set String %T2% "Notepad"    // Window title
Write Registry String: "ProgramName"    // Write program name
Write Registry String: "WindowTitle"    // Write window title
Macro Run: { Program - Focus }   // Focus
Read Registry Integer: "ReturnError?"   // Test for error
If Variable %N1% = 1
  Macro Stop
End If
```

Same thing using `{ PGM Function }`

```
Macro Run: { Program Operations Example C }   // Make sure the Notepad is already launched
// Launch a different program so we can switch focus.
Macro Run: { Program - Reset }   // Run the reset function
Variable Set String %T1% "{ Program - Launch }, calc.exe, Calculator, , , Minimized"
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }   // Call the function
Read Registry Integer: "ReturnError?"   // Test for error
If Variable %N1% = 1
  Macro Stop
End If
// Set Focus to the Notepad.
Variable Set String %T1% "{ Program - Focus }, notepad.exe, Notepad"
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }   // Call the function
Read Registry Integer: "ReturnError?"   // Test for error
If Variable %N1% = 1
  Macro Stop
End If
```
Example E - You want to launch the Notepad minimized and then wait for 10 seconds before accessing it.

```plaintext
// { Program Operations Example E }  
Macro Run: { Program - Reset }  // Run the reset function
If Program Name "notepad.exe" is running  // If the Notepad is running stop it.
  Variable Set String %T1% "notepad.exe"  // Program name
  Variable Set String %T2% "Notepad"  // Window title
  Write Registry String: "ProgramName"  // Write program name
  Write Registry String: "WindowTitle"  // Write window title
  Macro Run: { Program - Terminate }  // Shut it down
  Read Registry Integer: "ReturnError?"  // Test for error
  If Variable %N1% = 1
    Macro Stop
  End If
End If

Variable Set String %T1% "notepad.exe"  // Program name
Variable Set String %T2% "Notepad"  // Window title
Variable Set String %T3% "Minimized"  // Launch state
Variable Set Integer %N1% to 10  // Delay
Write Registry String: "ProgramName"  // Write program name
Write Registry String: "WindowTitle"  // Write window title
Write Registry String: "LaunchState"  // Write launch state
Write Registry Integer: "LaunchDelay"  // Write launch delay
Macro Run: { Program - Launch }  // Launch
Read Registry Integer: "ReturnError?"  // Test for error
If Variable %N1% = 1
  Macro Stop
End If
Window Restore: "Notepad"  // Normalize after the delay
```

Once again using `{ PGM Function }`

```plaintext
Macro Run: { Program - Reset }  // Run the reset function
If Program Name "notepad.exe" is running  // If the Notepad is running stop it.
  Variable Set String %T1% "{ Program - Terminate }, notepad.exe, Notepad"
  Write Registry String: "PgmFunction"  // Write it to the Registry
  Macro Run: { PGM Function }  // Call the function
  Read Registry Integer: "ReturnError?"  // Test for error
  If Variable %N1% = 1
    Macro Stop
  End If
End If

Variable Set String %T1% "{ Program - Launch },notepad.exe, Notepad, , , Minimized, , 10"
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: { PGM Function }  // Call the function
Read Registry Integer: "ReturnError?"  // Test for error
If Variable %N1% = 1
  Macro Stop
End If
Window Restore: "Notepad"  // Normalize after the delay
```
Functions

The Program Operations category is composed of twelve functions, which handle launching, focusing, and terminating programs in a reliable manner, in addition to having a few helpful utility functions thrown into the mix.

The three primary functions are:

- **{ Program - Launch }**
  - Launches a program and sets focus to it
- **{ Program - Focus }**
  - Sets focus to a program that is already running
- **{ Program - Terminate }**
  - Terminates a running program

There are three functions to validate values:

- **{ Program - Validate Launch Values }**
  - Checks that launch values fall within acceptable limits
- **{ Program - Validate Verification Values }**
  - Checks that the verify values fall within acceptable limits
- **{ Program - Validate Termination Values }**
  - Checks that the termination values falls within limits

Three to toggle features on and off:

- **{ Program - Toggle Parameter Validation }**
  - Toggles the ValidateParameters? value on and off
- **{ Program - Toggle Exact Match }**
  - Toggles the ExactMatch? value on and off
- **{ Program - Toggle Event Logging }**
  - Toggles the LogEvent? value on and off

One to record events:

- **{ Program - Log Event }**
  - Records the event string to the log file.

And two more to reset Registry values:

- **{ Program - Reset }**
  - Resets Registry values back to default values
- **{ Program - Clear Error Flag }**
  - Resets the ReturnError? value back to zero after an error

{ **PGM Function** } only works with the three primary functions **{ Program - Launch }, { Program - Focus }, and { Program - Terminate }**. The rest return an error message if you attempt to use them in this manner.
{ Program - Launch }

Launches a program and sets focus to it. This function uses the following steps to assure that the program will launch, if at all possible, and then gain focus.

1. Reads the passed parameters and validates them.
2. Creates a Macro Express launch command based on the validated parameters.
3. Launches the program.
4. Loops until the launch is confirmed.
5. Creates a Macro Express command to test if the program is running.
6. Loops until running is confirmed by the test command.
7. Creates a Macro Express command to test if the program has gained focus.
8. Loops until focus is confirmed by the test command.

Parameters

ProgramName - Name of program to run.
WindowTitle - Name of program's window.
ProgramFolder - Program's home folder
ProgramParameters - Command line argument string passed to program.
LaunchState - Launch as normal, minimized, or maximized. Defaults to normal.
ExactMatch - WindowTitle to be an exact or partial match. Defaults to partial match.
LaunchDelay - Number of seconds to wait after launching. Defaults to zero seconds.
VerifyLoops - Number of attempts to launch and verify program. Defaults to 50.
VerifyDelay - Number of milliseconds to wait between attempts. Defaults to 500 milliseconds.
ValidateParameters? - Launch and verification parameters are to be validated. Defaults to false.

{ PGM Function } Parameters

ProgName or PName - Required
WinTitle or WTitle - Required
ProgFolder or PHome - Set to empty string if omitted
ProgParams or PParams - Set to empty string if omitted
LaunchState or WShow - Current value if omitted
ExactMatch or Exact - Set to zero if omitted
LaunchDelay or LDelay - Current value if omitted
VerifyLoops or VLoops - Current value if omitted
VerifyDelay or VDelay - Current value if omitted
ValidateParameters or VParams - Current value if omitted

Returns

ReturnError? = 0 if no error occurs or if program is already running.
ReturnError? = 1 if an error occurs.

Calls

{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Program - Log Event }
{ Program - Validate Launch Values }
{ Program - Validate Verification Values }
Example
The following example resets the Program Operations values back to their default state then launches Microsoft Excel without validating parameters, minimized on the taskbar, loads My Spreadsheet.doc workbook, delays for 5 seconds, and then normalizes and centers it on your desktop.

Macro Run: { Program - Reset } // Reset values to their default state
Variable Set String %T1% "excel.exe" // Set program name
Variable Set String %T2% "Microsoft Excel" // Set window title
Variable Set String %T3% "c:\program files\microsoft office\office10" // Set home folder
Variable Set String %T4% "c:\my documents\My Spreadsheet.xls" // Set command line argument
Variable Set String %T5% "Minimized" // Set to launch minimized
Variable Set Integer %N1% to 5 // Set launch delay to 5 seconds
Variable Set Integer %N2% to 0 // Disable validation
Write Registry String: "ProgramName" // Write program name
Write Registry String: "WindowTitle" // Write window title
Write Registry String: "ProgramFolder" // Write home folder
Write Registry String: "ProgramParameters" // Write command line argument
Write Registry String: "LaunchState" // Write launch state
Write Registry Integer: "LaunchDelay" // Write launch delay
Write Registry Integer: "ValidateParameters?" // Write validation state
Macro Run: { Program - Launch } // Launch the package
Read Registry Integer: "ReturnError?" // Test for an error
If Variable %N1% = 1
   Text Box Display: Error! // Prompt user if there is one
End If
Window Restore: "%T2%" // Normalize after the delay
Window Reposition: Center - %T2% // Center on desktop

Same events using { PGM Function }:

Macro Run: { Program - Reset } // Reset values to their default state
Variable Set String %T1% "{ Program - Launch }, excel.exe, Microsoft Excel, c:\program files\microsoft office\office10," "c:\my documents\My Spreadsheet.xls", Minimized, , 5, , , 0" // Set program name, window title, home folder, command line argument, launch minimized, launch delay, validation
Write Registry String: "PgmFunction" // Write PGM Function string to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Integer: "ReturnError?" // Test for an error
If Variable %N1% = 1
   Text Box Display: Error! // Prompt user if there is one
End If
Window Restore: "Microsoft Excel" // Normalize after the delay
Window Reposition: Center - "Microsoft Excel" // Center on desktop

Notice the "c:\my documents\My Spreadsheet.xls"] part of the string? Here's the reason it is formatted in that manner: Windows expects any command line argument containing spaces to be wrapped in double quotation marks, and of course the command line argument in this example is c:\my documents\My Spreadsheet.xls, which contains a space between my and documents, and My and Spreadsheet, therefore, it must be passed as: "c:\my documents\My Spreadsheet.xls". When the { PGM Function } parser sees this, it will think that it is just a string being passed inside quotes and will remove the quote marks, so to prevent this from happening, simply wrap the string in either single quote marks or, as we did … brackets, because they are easier to spot.
Now here is what the same string would look like using named parameters:

```
Variable Set String %T1% "{ Program - Launch }, ProgName=excel.exe, WinTitle=Microsoft Excel, ProgFolder=c:\program files\microsoft office\office10, ProgParams=["c:\my documents\My Spreadsheet.xls"], LaunchState=Minimized, LaunchDelay=5, ValidateParams=0"
```

Remember that named parameters need not be in any particular order to work. Here is the same string using abbreviated named parameters but in a completely different order:

```
Variable Set String %T1% "{ Program - Launch }, VParams=0, WShow=Minimized, PName=excel.exe, PHome=c:\program files\microsoft office\office10, WTitle=Microsoft Excel, PParams=["c:\my documents\My Spreadsheet.xls"], LDelay=5"
```
{ Program - Focus }

Sets focus to a program that is already running. This function uses the following steps to assure that the program gains focus, if at all possible.

1. Reads the passed parameters and validates the data.
2. Creates a Macro Express focus test command based on the validated parameters.
3. Activates the program window.
4. Loops until focus has been confirmed by the test command.

Parameters
ProgramName   - Name of the running program.
WindowTitle   - Name of program’s window.
ExactMatch?   - WindowTitle to be an exact or partial match. Defaults to partial match.
VerifyLoops   - Number of attempts to focus and verify program. Defaults to 50.
VerifyDelay   - Number of milliseconds to wait between attempts. Defaults to 500 milliseconds.
ValidateParameters? - Launch and verification parameters are to be validated. Defaults to true.

{ PGM Function } Parameters
ProgName or PName    - Required
WinTitle or WTitle    - Required
ExactMatch or Exact    - Set to zero if omitted
VerifyLoops or VLoops  - Current value if omitted
VerifyDelay or VDelay  - Current value if omitted
ValidateParams or VParams - Current value if omitted

Returns
ReturnError? = 0 if no error occurs or if program already has focus.
ReturnError? = 1 if an error occurs.

Calls
{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Program - Log Event }
{ Program - Validate Verification Values }

Example
The following example resets the Program Operations values back to their default state then focuses on the already running Microsoft Excel program based on a partial window title match and without validating parameters.

Macro Run: { Program - Reset } // Reset values to their default state
Variable Set String %T1% "excel.exe" // Set program name
Variable Set String %T2% "Microsoft Excel" // Set window title
Variable Set Integer %N1% to 0 // Set partial match
Variable Set Integer %N2% to 0 // Disable validation
Write Registry String: "ProgramName" // Write program name
Write Registry String: "WindowTitle"  // Write window title
Write Registry Integer: "ExactMatch?"  // Write match state
Write Registry Integer: "ValidateParameters?"  // Write validation state
Macro Run: { Program - Focus }
Read Registry Integer: "ReturnError?"  // Test for an error
If Variable %N1% = 1
  Text Box Display: Error!  // Prompt user if there is one
End If

Same events using \{ PGM Function \}:

Macro Run: { Program - Reset }  // Reset values to their default state
Variable Set String %T1% "{ Program - Focus }, excel.exe, Microsoft Excel, 0, , , 0"
Write Registry String: "PgmFunction"  // Write PGM Function string to the Registry
Macro Run: { PGM Function }
Read Registry Integer: "ReturnError?"  // Test for an error
If Variable %N1% = 1
  Text Box Display: Error!  // Prompt user if there is one
End If

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ Program - Focus }, ProgName=excel.exe, WinTitle=Microsoft Excel, ExactMatch=0,
ValidateParams=0"

Remember that named parameters need not be in any particular order to work. Here is the same string using abbreviated named parameters but in a completely different order:

Variable Set String %T1% "{ Program - Focus }, VParams=0, WShow=Minimized, PName=excel.exe, WTitle=Microsoft Excel,
Exact=0"
{ **Program - Terminate** }

Terminates a running program. This function uses the following steps to assure that the program stops, if at all possible.

1. Reads the passed parameters and validates the data.
2. Creates a Macro Express termination test command based on the validated parameters.
3. Terminates the program.
4. Loops until termination has been confirmed by the test command.

**Parameters**

- **ProgramName**   - Name of the running program.
- **TerminateTimeout**  - Number of milliseconds before function times out. Defaults to 150 milliseconds
- **VerifyLoops**   - Number of attempts to focus and verify program. Defaults to 50.
- **VerifyDelay**   - Number of milliseconds to wait between attempts. Defaults to 500 milliseconds.
- **ValidateParameters?**  - Launch and verification parameters are to be validated. Defaults to true.

**Returns**

- **ReturnError?** = 0 if no error occurs or if program is not running.
- **ReturnError?** = 1 if an error occurs.

**Calls**

- { Variables - Save 25 }
- { Variables - Restore 25 }
- { Program - Log Event }
- { Program - Validate Termination Values }

**Example**

The following example resets the Program Operations values back to their default state then terminates the Microsoft Excel program using a timeout value of 200 milliseconds and also validates parameters.

```
Macro Run: { Program - Reset }                   // Reset values to their default state
Variable Set String %T1% "excel.exe"            // Set program name
Variable Set Integer %N1% to200                 // Set timeout value
Variable Set Integer %N2% to 1                  // Enable validation
Write Registry String: "ProgramName"            // Write program name
Write Registry Integer: "TerminateTimeout"      // Write match state
Write Registry Integer: "ValidateParameters?"   // Write validation state
Macro Run: { Program - Terminate }              // Terminate the program
Read Registry Integer: "ReturnError?"           // Test for an error
```
If Variable %N1% = 1
  Text Box Display: Error!  // Prompt user if there is one
End If

Same events using *(PGM Function)*:

  Macro Run: { Program - Reset }  // Reset values to their default state
  Variable Set String %T1% "{ Program - Terminate }, excel.exe, 200, ,, 1"
  Write Registry String: "PgmFunction"  // Write PGM Function string to the Registry
  Macro Run: { PGM Function }  // Call it
  Read Registry Integer: "ReturnError?"  // Test for an error
  If Variable %N1% = 1
    Text Box Display: Error!  // Prompt user if there is one
  End If

Now here is what the same string would look like using *named* parameters:

  Variable Set String %T1% "{ Program - Terminate }, ProgName=excel.exe, Timeout=200, ValidateParams=1"

Remember that named parameters need not be in any particular order to work. Here is the same string using *abbreviated* named parameters but in a completely different order:

  Variable Set String %T1% "{ Program - Terminate }, VParams=1, PName=excel.exe, Time=200"
{ Program - Validate Launch Values }

Validates that the **LaunchState** and **LaunchDelay** values fall within acceptable limits. If not then they are automatically adjusted and saved. When using `{ Program - Launch }`, validation will take place only if the **ValidateParameters?** value is set to 1. However, this function may be run on its own at any time you choose.

If the **LaunchState** value contains "MINIMIZED", "M", or "1" then it is changed to "Minimized". If it is "MAXIMIZED", "X", or "2" then it will be changed to "Maximized". Any other value will be set to "Normal".

If the **LaunchDelay** value is less than zero, or greater than the value stored in **MaxLaunchDelay** (which is 30 seconds) then it is changed to its default value of zero. If it is within the limits, then nothing happens, and the value remains as specified.

**Syntax**

{ Program - Validate Launch Values }

**Parameters**

Although there are no parameters to pass, the following Registry values are automatically accessed by this function:

- **LaunchState** - Launch program as normal, minimized, or maximized. Defaults to normal.
- **LaunchDelay** - Number of seconds to wait after program launches. Defaults to zero seconds.
- **MaxLaunchDelay** - Maximum value allowed for the **LaunchDelay** value. Defaults to 30 seconds.

**Returns**

Nothing. However, if either the **LaunchState** or **LaunchDelay** values are out of bounds, this fact gets recorded in the event log file if logging is enabled.

**Calls**

- `{ Variables - Save 25 }`
- `{ Variables - Restore 25 }`
- `{ Program - Log Event }`
{ Program - Validate Verification Values }

Validates that the **VerifyLoops** and **VerifyDelay** values fall within acceptable limits. If not then they are automatically adjusted and saved. When using any of the three main functions **{ Program - Launch },** **{ Program - Focus },** or **{ Program - Terminate },** validation will take place only if the **ValidateParameters?** value is set to 1. However, this function may be run on its own at any time you choose.

If the **VerifyLoops** value is less than zero, or greater than the value stored in **MaxVerifyLoops** (which is 240) then it is changed to its default value of 50. If it is within the limits, then nothing happens, and the value remains as specified.

If the **VerifyDelay** value is less than zero, or greater than the value stored in **MaxVerifyDelay** (which is 15,000 milliseconds) then it is changed to its default value of 500 milliseconds. If it is within the limits, then nothing happens, and the value remains as specified.

**Syntax**

{ Program - Validate Verification Values }

**Parameters**

Although there are no parameters to pass, the following Registry values are automatically accessed by this function:

- **VerifyLoops** - Number of verification attempts. Defaults to 50.
- **VerifyDelay** - Number of milliseconds to wait between attempts. Defaults to 500 milliseconds.
- **MaxVerifyLoops** - Maximum value allowed for the **VerifyLoops** value. Defaults to 240.
- **MaxVerifyDelay** - Maximum value allowed for the **VerifyDelay** value. Defaults to 15,000 milliseconds.

**Returns**

Nothing. However, if either the **VerifyLoops**, or **VerifyDelay** values are out of bounds, this fact gets recorded in the event log file if logging is enabled.

**Calls**

- { Variables - Save 25 }
- { Variables - Restore 25 }
- { Program - Log Event }
{ Program - Validate Termination Values }

Validates that the `TerminateTimeout` value falls within acceptable limits. If not then it’s automatically adjusted and saved. When using `{ Program - Terminate }`, validation will take place only if the `ValidateParameters?` value is set to 1. However, this function may be run on its own at any time you choose.

If the `TerminateTimeout` value is less than zero, or greater than the value which is stored in `MaxTerminateTimeout` (1000 milliseconds) then it is changed to its default value of 150 milliseconds. If it is within the limits, then nothing happens, and the value remains as specified.

**Syntax**

```
{ Program - Validate Termination Values }
```

**Parameters**

Although there are no parameters to pass, the following Registry values are automatically accessed by this function:

- `TerminateTimeout` - Number of milliseconds before function times out. Defaults to 150.
- `MaxTerminateTimeout` - Maximum value allowed for the `TerminateTimeout` value. Defaults to 1000.

**Returns**

Nothing. However, if its value is out of bounds, this fact gets recorded in the event log file if logging is enabled.

**Calls**

```
{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Program - Log Event }
```
{ Program - Toggle Parameter Validation }

Changes the state of the ValidateParameters? value. If it is enabled then it will be disabled. If it is disabled then it will be enabled. True to false, false to true, etc. 1 = true and 0 = false. The default value is false (do not check values).

This affects whether or not to automatically check the values of some parameters from directly inside the { Program - Launch }, { Program - Focus }, and { Program - Terminate } functions.

If set to true then:

{ Program - Launch } will call the { Program - Validate Launch Values } to check LaunchState and LaunchDelay values and also the { Program - Validate Verification Values } to check VerifyLoops and VerifyDelay values.

{ Program - Focus } will call the { Program - Validate Verification Values } to check VerifyLoops and VerifyDelay values.

{ Program - Terminate } will call the { Program - Validate Termination Values } to check the TerminateTimeout value.

Syntax
{ Program - Toggle Parameter Validation }

Parameters
None.

Returns
Nothing.

Calls
Nothing.
{ Program - Toggle Exact Match }

Changes the state of the *ExactMatch?* value. If it is enabled then it will be disabled. If it is disabled then it will be enabled. True to false, false to true, etc. \( 1 = \text{true} \) and \( 0 = \text{false} \). The default value is *false* (use a partial match).

This affects whether or not the *WindowTitle* value is used as an exact or partial match within the *Program - Launch* and *Program - Focus* functions.

**Syntax**

{ Program - Toggle Exact Match }

**Parameters**

None.

**Returns**

Nothing.

**Calls**

Nothing.
{ Program - Toggle Event Logging }

Changes the state of the $LogEvent?$ value. If it is enabled then it will be disabled. If it is disabled then it will be enabled. True to false, false to true, etc. $1 = true$ and $0 = false$. The default value is $true$ (use event logging).

This value is used by the { Program - Log Event } function to determine that, if called, what should be written to the event log?

Another way to disable event logging is to disable the { Program - Log Event } function (see the discussion below).

**Syntax**

{ Program - Toggle Event Logging }

**Parameters**

None.

**Returns**

Nothing.

**Calls**

Nothing.
{ Program - Log Event }

If the event logging value LogEvent? is enabled then this function records the event string written in the Event value to the log file specified in the EventLog value. Confused?

LogEvent? determines if event logging is enabled or disabled.
Event is the event string that you want written to the event log.
EventLog contains the name of the log file, which defaults to PGM Event.log.

In order to log an event, any event, you need only to write the event text string, say %T1%, to the Event value then call this function. The event is then appended to the end of the log along with a date and time stamp. A blank line can be appended if the first character of the event begins with an asterisk (*).

In addition to using the LogEvent? value, logging can be disabled by simply disabling the { Program - Log Event } function in the Library. We wrap all calls to this function inside an If Macro "{ Program - Log Event }" Enabled / End If structure.

Use event logging to not only record when a macro begins and ends, but for anything else in between. Record commands, variables, time, current folders, titles, windows, etc. There is no limit on what you may record. And best of all, recording will not slow your macro significantly, uhm, unless of course you log an event for every line in the macro.

Syntax
{ Program - Log Event }

Parameters
None.

Returns
Nothing.

Calls
Nothing.

Example
The following example is from the beginning of the { Program - Launch } function. It records the name of the function and also which program is being launched.

// Log function call.
If Macro "{ Program - Log Event }" Enabled
    Variable Set String %T9% "{ Program - Launch } - %T10%"
    Write Registry String: "Event"
    Macro Run: { Program - Log Event }
End If

// Is the event log function enabled?
// Set T9 to function name and target
// Write the event in T9 to the Registry
// Log event to the file
{ Program - Reset }

This utility program resets all of the Program Operations values stored in the Registry back to their default values. It is usually a good idea to call this function prior to calling the { Program - Launch }, { Program - Focus }, or { Program - Terminate } functions to be sure that unwanted values do not interfere with the call. Here's what gets changed:

Program and folder values:
- ProgramName - Erased
- WindowTitle - Erased
- ProgramFolder - Erased
- ProgramParameters - Erased
- ExactMatch? - Set to False (0)

Launch values:
- LaunchState - Set to "Normal"
- LaunchDelay - Set to 0 seconds
- MaxLaunchDelay - Set to 30 seconds

Verification values:
- VerifyLoops - Set to 50 attempts
- VerifyDelay - Set to 500 milliseconds
- MaxVerifyLoops - Set to 240 attempts
- MaxVerifyDelay - Set to 15,000 milliseconds
- ValidateParameters? - Set to False (0)

Return values:
- ReturnError? - Set to False (0)

Event values:
- Event - Erased
- LogEvent? - Set to True (1)

Termination values:
- TerminateTimeout - Set to 150 milliseconds
- MaxTerminateTimeout - Set to 1,000 milliseconds

Syntax

{ Program - Reset }

Parameters
None.

Returns
Nothing.

Calls
Nothing.
{ Program - Clear Error Flag }

This utility program resets the ReturnError? value back to false (0) after an error occurs. Use it to reset just the error flag in those situations where you don't want to change any other values.

Syntax
{ Program - Clear Error Flag }

Parameters
None.

Returns
Nothing.

Calls
Nothing.
Date and Time Category

Time sneaks up on you like a windshield on a bug.

- Jon Lithgow
Overview

We were asked numerous times by quite a few of our clients to create a set of reusable macro functions to handle date and time calculations and here they are. They will allow you to:

- Convert a date string to a Julian number, numeric day-of-week, character day-of-week, or numeric day-of-year.
- Convert today's date to a Julian number, numeric day-of-week, character day-of-week, or numeric day-of-year.
- Convert a Julian number to a date string, numeric day-of-week, character day-of-week, or numeric day-of-year.
- Calculate a new date by adding or subtracting days or months.
- Calculate the difference in days between two dates.
- Evaluate if a year number is a leap year.
- Return a character day-of-week from a numeric day-of-week.
- Return a character month from a numeric month.
- Parse a day number from either a date string, Julian number, or today's date.
- Parse a month number from either a date string, Julian number, or today's date.
- Parse a year number from either a date string, Julian number, or today's date.
- Reformat a user's date input string.
- Validate both date strings and Julian numbers input by users.

The functions are all based on Julian Day numbers, which are not to be confused with Julian Dates, whose calendar ended on October 15th, 1582 when the Gregorian calendar replaced it. The Gregorian calendar is the reformed Julian calendar with the year fixed at 365 days except for leap years which contain 366 days. Leap years are years that are exactly divisible by 4 except for century years which must also be divisible by 400. This is the calendar in common use today. Dates in the Gregorian calendar are called Gregorian dates. Here are some dates and their equivalent Julian Day number.

<table>
<thead>
<tr>
<th>Date</th>
<th>Julian Day Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 1st, 0000</td>
<td>as far back as these functions go 1721120</td>
</tr>
<tr>
<td>January 1st, 0001</td>
<td>first day of the 1st century 1721426</td>
</tr>
<tr>
<td>October 15th, 1582</td>
<td>first day of Gregorian calendar 2299161</td>
</tr>
<tr>
<td>November 16th, 1858</td>
<td>nothing special, just the Julian Day: 2400000</td>
</tr>
<tr>
<td>February 28th, 1900</td>
<td>last day of month, not a leap year 2415079</td>
</tr>
<tr>
<td>January 1st, 1901</td>
<td>first day of the 20th century 2415386</td>
</tr>
<tr>
<td>January 1st, 2001</td>
<td>first day of the 21st century 2451911</td>
</tr>
<tr>
<td>December 31st, 9999</td>
<td>as far into the future as we go 5373484</td>
</tr>
</tbody>
</table>

We were only able to cover a span of (almost) 10,000 years from March 1, 0000 through December 31, 9999. Any dates outside this range will return an error.

All functions have been tested against results obtained from Microsoft Visual FoxPro and Microsoft Visual Basic. We tested hundreds of thousands of dates and numbers to confirm the accuracy of these functions. If we've missed something, and I don't think we have, then please let us know right away.
## Registry

When the PGM Functions Library™ was first installed on your computer, an area called Parameters was created in your Registry within the Professional Grade Macros key along with another area called Date and Time. These areas, or keys, contain many values, eleven of which are used to communicate with the Date and Time functions and are shown in these two pictures. You communicate with them via the **Write Registry String**, **Write Registry Integer**, **Write Registry Decimal**, **Read Registry String**, **Read Registry Integer**, and **Read Registry Decimal** commands. The Name column, in the right hand pane, is the value name. The Type column is the type of variable being stored, REG_SZ for strings, REG_DWORD for integers, and REG_BINARY for decimals. And of course the Data column shows the actual value being stored.
Let’s examine in detail these Registry values and how you use them with the Date and Time functions. They can be grouped into three categories:

1. Input values - You set these.
2. Return values - Set by the functions.
3. Default values - Set by you and read by the functions.

**Input Values**

These are values that you write to the Registry prior to calling any of the Date and Time functions. The value depends on which function you are calling. For example, if you wanted to convert a date string to a Julian number, you would write the string-to-convert to the `ParameterString1` value, call the function, and then read the results from the `ReturnDecimal1` value.

**ParameterDecimal1**

For the Date and Time functions, this is always a Julian Day number to be converted in some manner such as is used by the `{ DateTime - Julian to Date }` and `{ DateTime - Validate Julian Number }` functions.

```
Variable Set Decimal %D1% to 2434389   // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write it to the Registry
Macro Run: { DateTime - Julian to Date }  // Convert it
Read Registry String: "ReturnString1"  // Read the results, T1="19530111"
```

**ParameterInteger1**

Holds an integer value to convert or calculate depending on the function that you are calling. You could convert, for example, a character day-of-week from a day-of-week `integer value` by using the `{ DateTime - Character DOW }` function, or skip a number of days (`integer value`) into the future with the `{ DateTime - Go Days }` function.

```
Variable Set Integer %N1% to 3   // Set N1 as the DOW to convert
Write Registry Integer: "ParameterInteger1"  // Write it to the Registry
Macro Run: { DateTime - Character DOW }  // Convert it
Read Registry String: "ReturnString1"  // Read the results, T1="Wednesday"
```

**ParameterInteger2**

Holds a second integer value depending on the function that you are calling. For example, the `{ DateTime - DOY to Date }` function requires two integer values, one for the year and the other for the day-of-year.

```
Variable Set Integer %N1% to 2000   // Set N1 as the Year
Variable Set Integer %N2% to 60    // Set N2 as the day-of-year
Write Registry Integer: "ParameterInteger1"  // Write N1 to the Registry
Write Registry Integer: "ParameterInteger2"  // Write N2 to the Registry
Macro Run: { DateTime - DOY to Date }  // Calculate it
Read Registry String: "ReturnString1"  // Read the results, T1="20000229"
```
**ParameterString1**

Contains a string value to convert or calculate depending on the function that you are calling. You could convert, for example, a date to a Julian Day number from a formatted date string value by using the `{DateTime - Date to Julian}` function, or skip a number of days from a date (string value) into the future with the `{DateTime - Go Days}` function.

```
Variable Set String %T1% to "20020430"  // Set T1 to a formatted date string
Write Registry String: "ParameterString1"  // Write it to the Registry
Macro Run: { DateTime - Date to Julian }  // Convert it
Read Registry Decimal: "ReturnDecimal1"  // Read the results, D1=2452395
```

**ParameterString2**

Holds a second string value depending on the function that you are calling. An example would be the `{DateTime - Span of Days}` function, which requires two string values, one for each date.

```
Variable Set String %T1% to "20000704"  // Set T1 to a formatted date string
Variable Set String %T2% to "20000222"  // Set T2 to a formatted date string
Write Registry String: "ParameterString1"  // Write T1 to the Registry
Write Registry String: "ParameterString2"  // Write T2 to the Registry
Macro Run: { DateTime - Span of Days }  // Calculate the number of days
Read Registry Integer: "ReturnInteger1"  // Read the results, N1=-133
```
Return Values

The resulting calculations performed by the Date and Time functions are placed in these values. You would read one of them, depending on which function you have called, for your answer. For example, if you wanted to convert a date string to a Julian number, you would write the string-to-convert to the ParameterString1 value, call the function, and then read the results from the ReturnDecimal1 value.

ReturnDecimal1
Contains the resulting calculation of a function call for those that return a Julian Day number such as the \{DateTime - Date to Julian\} function.

Variable Set String %T1% to "20020430" // Set T1 to a formatted date string
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: \{DateTime - Date to Julian\} // Convert it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D1=2452395

ReturnInteger1
Contains the results of any function that returns an integer value such as \{DateTime - Evaluate for Leap Year\} or \{DateTime - Span of Days\} functions.

Variable Set Integer %N1% to 2000 // Set N1 to a year
Write Registry Integer: "ParameterInteger1" // Write it to the Registry
Macro Run: \{DateTime - Evaluate for Leap Year\} // Calculate it
Read Registry Integer: "ReturnInteger1" // Read the results, N1=1 (true)

ReturnString1
Contains the results of any function which returns a string value. The \{DateTime - Julian to Date\} function is a prime example. It returns a formatted date string.

Variable Set Decimal %D1% to 2434389 // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1" // Write it to the Registry
Macro Run: \{DateTime - Julian to Date\} // Convert it
Read Registry String: "ReturnString1" // Read the results, T1="19530111"

ReturnError?
This value will tell you if an error has occurred in one of the Date and Time functions. If the value is 1 then an error occurred, and a 0 means that the function ran fine. This value should be tested after a function returns from processing so appropriate action can be taken by your macro. If an error occurs and, if event logging is enabled (LogEvent?), then the actual error will be contained in the event log.

Macro Run: \{DateTime - Date to Julian\} // Run the conversion function
Read Registry Integer: "ReturnError?" // Check for an error
If Variable %N1% = 1
Text Box Display: Error! // If so, then prompt the user
End If
Default Values

There are two values containing defaults used by these functions. Both of them can be changed any time you wish by using the \{DateTime - Reset Defaults\} and \{DateTime - Set Defaults\} functions.

Macro Run: \{DateTime - Reset Defaults\} // Reset default values
Read Registry Integer: "SundayStartsOn" // N1 = 0
Read Registry Integer: "MonthPosition" // N2 = 1

MonthPosition
Controls the position of the month within a date string. The default position is 1. This affects how the \{DateTime - Format Input String\} function interprets what a user has entered. For example, a user wants to convert a date string like 06/10/2002 to a Julian Day number. If the MonthPosition value is set to 1 then the user means to say June 10th, 2002. If it is set to 2 then the user means October 6th, 2002 instead.

SundayStartsOn
Sunday is assumed to be the first day of the week for these functions. This value sets Sunday to be either 0 (the default) or 1. It affects the results of day-of-week calculations such as those performed by \{DateTime - Date to DOW\}. If set to 0 then Sunday is 0, Monday is 1, Tuesday is 2, etc. Setting it to 1 results in Sunday being 1, Monday 2, Tuesday 3, etc.
Functions

The Date and Time category is composed of thirty-three functions, which fall into six categories as listed here. The first four categories can make use of { PGM Function }. The last two categories cannot.

There are five primary functions:

{ DateTime - Date to Julian } Converts a date string to a Julian Day number
{ DateTime - Julian to Date } Converts a Julian Day number to a date string
{ DateTime - Go Months } Calculates a new date based on the number of months input
{ DateTime - Go Days } Calculates a new date based on the number of days input
{ DateTime - Span of Days } Calculates the span, or number, of days between two dates

Seventy secondary functions:

{ DateTime - Julian to DOY } Converts a Julian Day number to a numeric day-of-year
{ DateTime - Date to DOY } Converts a date to a numeric day-of-year
{ DateTime - DOY to Date } Converts a year and a numeric day-of-year to a date string.
{ DateTime - Julian to DOW } Converts a Julian Day number to a numeric day-of-week
{ DateTime - Date to DOW } Converts a date to a numeric day-of-week
{ DateTime - Julian to Character DOW } Converts a Julian Day number to a character day-of-week
{ DateTime - Date to Character DOW } Converts a date to a character day-of-week

Six support functions:

{ DateTime - Character DOW } Converts a day-of-week number to a character day-of-week
{ DateTime - Character Month } Converts a month number to a character month
{ DateTime - Evaluate for Leap Year } Evaluates a year to determine if it is a leap year
{ DateTime - Format Input String } Converts a date string to a formatted string (YYYYMMDD)
{ DateTime - Validate Date String } Performs a validation check on an input date string
{ DateTime - Validate Julian Number } Performs a validation check on a Julian Day number

Six parsing functions:

{ DateTime - Parse Day from Date } Returns the DD part of a date string as an integer
{ DateTime - Parse Day from Julian } Returns the day-of-month from a Julian date
{ DateTime - Parse Month from Date } Returns the MM part of a date string as an integer
{ DateTime - Parse Month from Julian } Returns the month from a Julian date
{ DateTime - Parse Year from Date } Returns the YYYY part of a date string as an integer
{ DateTime - Parse Year from Julian } Returns the year from a Julian date

Seven that generate a value from today:

{ DateTime - Parse Day from Today } Returns the DD part of today as an integer
{ DateTime - Parse Month from Today } Returns the MM part of today as an integer
{ DateTime - Parse Year from Today } Returns the YYYY part of today as an integer
{ DateTime - Today to DOW } Converts today to a numeric day-of-week
{ DateTime - Today to Character DOW } Converts today to a character day-of-week
{ DateTime - Today to DOY } Converts today to a numeric day-of-year
{ DateTime - Today to Julian } Converts today to a Julian Day number
And two that reset default Registry values:

- { DateTime - Reset Defaults } Resets the Date and Time values back to their default values
- { DateTime - Set Defaults } Interactively set the Date and Time default values

When you see the term "formatted date string" in this section, it refers to a "YYYYMMDD" style, eight character string consisting of a 4-digit year (0000 through 9999), a 2-digit month (01 through 12), and a 2-digit day (01 through 31).

If an individual function can use { PGM Function } then it is so noted. As with many of our functions, if an error occurs you’ll be notified via the ReturnError? value and, if event logging is enabled, the actual error will be recorded in the log file. For the sake of clarity, the examples here do not check for an error, however, it is recommended that you do check, so appropriate action can be taken.
{ DateTime - Date to Julian }

Converts a formatted date string to a Julian Day number.

Parameters
ParameterString1 - Formatted date string

{ PGM Function } Parameters
Date or Dt - Required

Returns
ReturnDecimal1 - Julian Day number
ReturnError? - 0 if no error occurs, or 1 if an error occurs.

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Validate Date String }

Example
The following example converts a formatted date string "20020430" to a Julian Day number 2452395.

Variable Set String %T1% "20020430"     // Set T1 to a formatted date string
Write Registry String: "ParameterString1"    // Write T1 to the Registry
Macro Run: { DateTime - Date to Julian }     // Convert it to a Julian number
Read Registry Decimal: "ReturnDecimal1"    // Read results into D1 (2452395)

Same example using { PGM Function }:

Variable Set String %T1% "{ DateTime - Date to Julian }, 20020430"   // Create PGM Function string
Write Registry String: "PgmFunction"        // Write it to the Registry
Macro Run: { PGM Function }          // Call it
Read Registry Decimal: "ReturnDecimal1"       // Read results into D1 (2452395)

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ DateTime - Date to Julian }, Date=20020430"

Or:

Variable Set String %T1% "{ DateTime - Date to Julian }, Dt=20020430"
{ DateTime - Julian to Date }

Converts a Julian Day number to a formatted date string.

**Parameters**

ParameterDecimal1 - Julian Day number

**PGM Function Parameters**

Julian or Jn - Required

**Returns**

ReturnString1 - Formatted date string
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Validate Julian Number }

**Example**

The following example converts a Julian day number 2434389 to a formatted date string "19530111".

```
Variable Set String %D1% 2434389     // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry
Macro Run: { DateTime - Julian to Date }     // Convert it to a formatted date string
Read Registry String: "ReturnString1"     // Read results into T1 ("19530111")
```

Same example using **PGM Function**:

```
Variable Set String %T1% "{ DateTime - Julian to Date }, 2434389"   // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: { PGM Function }          // Call it
Read Registry String: "ReturnString1"     // Read results into T1 ("19530111")
```

Now here is what the same string would look like using *named* parameters:

```
Variable Set String %T1% "{ DateTime - Julian to Date }, Julian=2434389"
```

Or:

```
Variable Set String %T1% "{ DateTime - Julian to Date }, Jn=2434389"
```


{ Date Time - Go Months }

Calculates a new date based on the number of months input. If the number of months is positive then the date returned will be that many months in the future. If the number of months is negative, it will be that many months in the past.

This function allows that months contain a different number of days and will never overrun the end of the month in the resulting date. For example; adding one month to January 30th will result in February 28th and subtracting two months results in November 30th. Adding one month to January 1st results in February 1st and subtracting two months results in November 1st.

Parameters

- ParameterString1 - Formatted date string (start)
- ParameterInteger1 - Number of months to go into the future or past

Returns

- ReturnString1 - Formatted date string
- ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls

- Variables - Save DT
- Variables - Restore DT
- Program - Log Event
- Math - Mod
- DateTime - Validate Date String
- DateTime - Evaluate for Leap Year

Example

This example adds 3 months to the formatted date string "20020815" (August 15th, 2002), giving us "20021115" (November 15th, 2002).

```
Variable Set String %T1% "20020815" // Set T1 to a formatted date string
Variable Set Integer %N1% to 3 // Set N1 to advance 3 months
Write Registry String: "ParameterString1" // Write date to the Registry
Write Registry Integer: "ParameterInteger1" // Write months to the Registry
Macro Run: { DateTime - Go Months } // Calculate a new date
Read Registry String: "ReturnString1" // Read results into T1 ("20021115")
```
Subtract 2 months from the formatted date string "20030430" (April 30th, 2003), giving us “20030228" (February 28th, 2003).

Variable Set String %T1% "20030430" // Set T1 to a formatted date string
Variable Set Integer %N1% to -2 // Set N1 to skip back 2 months
Write Registry String: "ParameterString1" // Write date to the Registry
Write Registry Integer: "ParameterInteger1" // Write months to the Registry
Macro Run: { DateTime - Go Months } // Calculate a new date
Read Registry String: "ReturnString1" // Read results into T1 ("20030228")

Same two examples using { PGM Function }:

Variable Set String %T1% "{ DateTime - Go Months }, 20020815, 3" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry String: "ReturnString1" // Read results into T1 ("20021115")

Variable Set String %T1% "{ DateTime - Go Months }, 20030430, -2" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry String: "ReturnString1" // Read results into T1 ("20030228")

And using named parameters:

Variable Set String %T1% "{ DateTime - Go Months }, Date=20020815, NumMonths=3"

Or:

Variable Set String %T1% "{ DateTime - Go Months }, Dt=20020815, Go=3"

Second example:

Variable Set String %T1% "{ DateTime - Go Months }, Date=20030430, NumMonths=-2"

Or:

Variable Set String %T1% "{ DateTime - Go Months }, Dt=20030430, Go=-2"
**{ DateTime - Go Days }**

Calculates a new date based on the number of days input. If the number of days is positive then the date returned will be that many days in the future. If the number of days is negative, it will be that many days in the past.

**Parameters**

ParameterString1 - Formatted date string (start)
ParameterInteger1 - Number of days to go into the future or past

**{ PGM Function } Parameters**

Date or Dt - Required
NumDays or Go - Required

**Returns**

ReturnString1 - Formatted date string
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to Julian }
{ DateTime - Julian to Date }

**Example**

Add 69 days to the formatted date string "20020815" (August 15th, 2002), resulting in “20021023” (October 23rd, 2002).

```plaintext
Variable Set String %T1% "20020815"  // Set T1 to a formatted date string
Variable Set Integer %N1% to 69        // Set N1 to advance 69 days
Write Registry String: "ParameterString1"  // Write date to the Registry
Write Registry Integer: "ParameterInteger1"  // Write months to the Registry
Macro Run: { DateTime - Go Days }  // Calculate a new date
Read Registry String: "ReturnString1"  // Read results into T1 (“20021023”)
```

Subtract 69 days from the same string, giving us “20020607” (June 7th, 2002).

```plaintext
Variable Set String %T1% "20020815"  // Set T1 to a formatted date string
Variable Set Integer %N1% to -69       // Set N1 to skip back 69 days
Write Registry String: "ParameterString1"  // Write date to the Registry
Write Registry Integer: "ParameterInteger1"  // Write months to the Registry
Macro Run: { DateTime - Go Days }  // Calculate a new date
Read Registry String: "ReturnString1"  // Read results into T1 (“20020607”)
```
Same two examples using \textit{PGM Function}:

\begin{verbatim}
Variable Set String \texttt{%T1\% } "\{ DateTime - Go Days \}, 20020815, 69"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: \{ PGM Function \}  // Call it
Read Registry String: "ReturnString1"  // Read results into T1 ("20021023")

Variable Set String \texttt{%T1\% } "\{ DateTime - Go Days \}, 20020815, -69"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: \{ PGM Function \}  // Call it
Read Registry String: "ReturnString1"  // Read results into T1 ("20020607")
\end{verbatim}

And using \textit{named} parameters:

\begin{verbatim}
Variable Set String \texttt{%T1\% } "\{ DateTime - Go Days \}, Date=20020815, NumDays=69"
Or:
Variable Set String \texttt{%T1\% } "\{ DateTime - Go Days \}, Dt=20020815, Go=69"

Second example:

Variable Set String \texttt{%T1\% } "\{ DateTime - Go Days \}, Date=20020815, Numdays=-69"
Or:
Variable Set String \texttt{%T1\% } "\{ DateTime - Go Days \}, Dt=20020815, Go=-69"
\end{verbatim}
{ DateTime - Span of Days }

Calculates the span, or number, of days between two dates. If the second date is greater than the first date then the number of days returned will be positive. If the second date is less than the first date then the number of days returned will be negative.

Parameters
ParameterString1  - Formatted date string (start)
ParameterString2  - Formatted date string (end)

{ PGM Function } Parameters
Date1 or Dt    - Required
Date2 or Go   - Required

Returns
ReturnInteger1  - Number of days (positive or negative)
ReturnError?   - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to Julian }

Example
Calculate the number of days between "20000222" (February 22\textsuperscript{nd}, 2000) and "20000704" (July 4\textsuperscript{th}, 2000), resulting in 133 days.

```
Variable Set String %T1% "20000222"     // Set T1 to a formatted starting date string
Variable Set String %T2% "20000704"     // Set T2 to a formatted ending date string
Write Registry String: "ParameterString1"  // Write starting date to the Registry
Write Registry String: "ParameterString2"  // Write ending date to the Registry
Macro Run: { DateTime - Span of Days }     // Calculate the number of days
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (133 days)
```

Reversing the dates will result in -133 days.

```
Variable Set String %T1% "20000704"     // Set T1 to a formatted starting date string
Variable Set String %T2% "20000222"     // Set T2 to a formatted ending date string
Write Registry String: "ParameterString1"  // Write starting date to the Registry
Write Registry String: "ParameterString2"  // Write ending date to the Registry
Macro Run: { DateTime - Span of Days }     // Calculate the number of days
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (-133 days)
```
Same two examples using \textit{PGM Function}:

\begin{verbatim}
Variable Set String \texttt{%T1} \texttt{"\{ DateTime - Span of Days \}, 20000222, 20000704"} \hfill \texttt{// Create PGM Function string}
Write Registry String: \texttt{"PgmFunction"} \hfill \texttt{// Write it to the Registry}
Macro Run: \texttt{( PGM Function )} \hfill \texttt{// Call it}
Read Registry Integer: \texttt{"ReturnInteger1"} \hfill \texttt{// Read results into N1 (133 days)}
\end{verbatim}

\begin{verbatim}
Variable Set String \texttt{%T1} \texttt{"\{ DateTime - Span of Days \}, 20000004, 20000222"} \hfill \texttt{// Create PGM Function string}
Write Registry String: \texttt{"PgmFunction"} \hfill \texttt{// Write it to the Registry}
Macro Run: \texttt{( PGM Function )} \hfill \texttt{// Call it}
Read Registry Integer: \texttt{"ReturnInteger1"} \hfill \texttt{// Read results into N1 (-133 days)}
\end{verbatim}

And using \textit{named} parameters:

\begin{verbatim}
Variable Set String \texttt{%T1} \texttt{"\{ DateTime - Span of Days \}, Date1=20000222, Date2=20000704"}
\end{verbatim}

Or:

\begin{verbatim}
Variable Set String \texttt{%T1} \texttt{"\{ DateTime - Span of Days \}, Dt=20000222, Go=20000704"}
\end{verbatim}

Second example:

\begin{verbatim}
Variable Set String \texttt{%T1} \texttt{"\{ DateTime - Span of Days \}, Date1=20000704, Date2=20000222"}
\end{verbatim}

Or:

\begin{verbatim}
Variable Set String \texttt{%T1} \texttt{"\{ DateTime - Span of Days \}, Dt=20000704, Go=20000222"}
\end{verbatim}
\{ \textbf{DateTime - Julian to DOY} \} \\

Converts a Julian Day number to a numeric day-of-year (120, 200, 365, ...).

\textbf{Parameters} \\
ParameterDecimal1 - Julian Day number

\textbf{PGM Function Parameters} \\
Julian or Jn - Required

\textbf{Returns} \\
ReturnInteger1 - Numeric day-of-year \\
ReturnError? - 0 if no error occurs, or 1 if an error occurs

\textbf{Calls} \\
\{ Variables - Save DT \} \\
\{ Variables - Restore DT \} \\
\{ Program - Log Event \} \\
\{ DateTime - Julian to Date \} \\
\{ DateTime - Date to DOY \}

\textbf{Example} \\
The following example converts a Julian Day number \(2434389\) (January 11\(^{th}\), 1953) to a numeric day-of-year 11.

\begin{verbatim}
Variable Set String %D1% 2434389     // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry 
Macro Run: \{ DateTime - Julian to DOY \}     // Convert it to a numeric day-of-year 
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (day number 11)
\end{verbatim}

Same example using \textit{PGM Function}:

\begin{verbatim}
Variable Set String %T1% "\{ DateTime - Julian to DOY \}, 2434389"     // Create PGM Function string
Write Registry String: "PgmFunction"        // Write it to the Registry 
Macro Run: \{ PGM Function \}              // Call it 
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (day number 11)
\end{verbatim}

Now here is what the same string would look like using \textit{named} parameters:

\begin{verbatim}
Variable Set String %T1% "\{ DateTime - Julian to DOY \}, Julian=2434389"
Or:
Variable Set String %T1% "\{ DateTime - Julian to DOY \}, Jn=2434389"
\end{verbatim}
{ DateTime - Date to DOY }

Converts a date to a numeric day-of-year (120, 200, 365, ...).

Parameters
ParameterString1 - Formatted date string

{ PGM Function } Parameters
Date or Dt - Required

Returns
ReturnInteger1 - Numeric day-of-year
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to Julian }

Example
This example will convert the formatted date string "20020430" (April 30th, 2002) to a numeric day-of-year 120.

Variable Set String %T1% "20020430"     // Set T1 to a formatted date string
Write Registry String: "ParameterString1"    // Write starting date to the Registry
Macro Run: { DateTime - Date to DOY }     // Calculate the day number
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (day number 120)

Same example with { PGM Function }:

Variable Set String %T1% "{ DateTime - Date to DOY }, 20020430"   // Create PGM Function string
Write Registry String: "PgmFunction"        // Write it to the Registry
Macro Run: { PGM Function }              // Call it
Read Registry Integer: "ReturnInteger1"       // Read results into N1 (day number 120)

And using named parameters:

Variable Set String %T1% "{ DateTime - Date to DOY }, Date=20020430"

Or:

Variable Set String %T1% "{ DateTime - Date to DOY }, Dt=20020430"
{ DateTime - DOY to Date }

Converts a year and a numeric day-of-year (120, 200, 365, …) to a formatted date string.

Parameters
ParameterInteger1 - Year
ParameterInteger2 - Numeric day-of-year

{ PGM Function } Parameters
Year or Yr - Required
DayOfYear or Doy - Required

Returns
ReturnString1 - Formatted date string
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Evaluate for Leap Year }

Example
This example will convert the year 2000 and the day-of-year 60 to a formatted date string "20000229" (February 29th, 2000).

Variable Set Integer %N1% to 2000     // Set N1 to the year 2000
Variable Set Integer %N2% to 60      // Set N2 to day 60
Write Registry Integer: "ParameterInteger1"    // Write year o the Registry
Write Registry Integer: "ParameterInteger2"    // Write day to the Registry
Macro Run: { DateTime - DOY to Date }     // Convert to a formatted date string
Read Registry String: "ReturnString1"     // Read results into T1 ("20000229")

Same example with { PGM Function }:

Variable Set String %T1% "{ DateTime - DOY to Date }, 2000, 60"   // Create PGM Function string
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }          // Call it
Read Registry String: "ReturnString1"        // Read results into T1 ("20000229")

And using named parameters:

Variable Set String %T1% "{ DateTime - DOY to Date }, Year=2000, DayOfYear=60"

Or:
Variable Set String %T1% "{ DateTime - DOY to Date }, Yr=2000, Doy=60"
{ DateTime - Julian to DOW }

Converts a Julian Day number to a numeric day-of-week (0, 1, 2, ...).

Parameters
ParameterDecimal1 - Julian Day number

{ PGM Function } Parameters
Julian or Jn - Required

Returns
ReturnInteger1 - Numeric day-of-week
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Julian to Date }
{ DateTime - Date to DOW }

Example
The following example converts a Julian Day number 2434389 (January 11th, 1953) to a numeric day-of-week 0.

Variable Set String %D1% 2434389  // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry
Macro Run: { DateTime - Julian to DOW }  // Convert it to a numeric day-of-week
Read Registry Integer: "ReturnInteger1"  // Read results into N1 (day 0)

Same example using { PGM Function }:

Variable Set String %T1% "{ DateTime - Julian to DOW }, 2434389"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: ( PGM Function )  // Call it
Read Registry Integer: "ReturnInteger1"  // Read results into N1 (day 0)

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ DateTime - Julian to DOW }, Julian=2434389"
Or:
Variable Set String %T1% "{ DateTime - Julian to DOW }, Jn=2434389"
{ **DateTime - Date to DOW** }

Converts a date to a numeric day-of-week (0, 1, 2, ...).

**Parameters**

ParameterString1 - Formatted date string

**{ PGM Function } Parameters**

Date or Dt - Required

**Returns**

ReturnInteger1 - Numeric day-of-week
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ Math - Mod }
{ DateTime - Validate Date String }

**Example**

This example will convert the formatted date string "19530111" (January 11th, 1953) to a numeric day-of-week 0.

```plaintext
Variable Set String %T1% "19530111"       // Set T1 to a formatted date string
Write Registry String: "ParameterString1"   // Write date to the Registry
Macro Run: { DateTime - Date to DOW }        // Calculate the day number
Read Registry Integer: "ReturnInteger1"      // Read results into N1 (day 0)
```

Same example with { **PGM Function** }:

```plaintext
Variable Set String %T1% "{ DateTime - Date to DOW }, 19530111"   // Create PGM Function string
Write Registry String: "PgmFunction"                              // Write it to the Registry
Macro Run: { PGM Function }                                      // Call it
Read Registry Integer: "ReturnInteger1"                         // Read results into N1 (day 0)
```

And using named parameters:

```plaintext
Variable Set String %T1% "{ DateTime - Date to DOW }, Date=19530111"  
Or:
Variable Set String %T1% "{ DateTime - Date to DOW }, Dt=19530111"
```
\{ DateTime - Julian to Character DOW \}

Converts a Julian Day number to a character day-of-week (Monday, Tuesday, …).

**Parameters**

ParameterDecimal1   - Julian Day number

**{ PGM Function } Parameters**

Julian or Jn   - Required

**Returns**

ReturnString1   - Name of the day
ReturnError?   - 0 if no error occurs, or 1 if an error occurs

**Calls**

\{ Variables - Save DT \}
\{ Variables - Restore DT \}
\{ Program - Log Event \}
\{ DateTime - Julian to Date \}
\{ DateTime - Date to Character DOW \}

**Example**

The following example converts a Julian Day number 2434389 (January 11\textsuperscript{th}, 1953) to a character day-of-week Sunday.

```plaintext
Variable Set String %D1% 2434389  // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry
Macro Run: \{ DateTime - Julian to Character DOW \}  // Convert to a character day-of-week
Read Registry String: "ReturnString1"  // Read results into T1 (Sunday)
```

Same example using \{ PGM Function \}:

```plaintext
Variable Set String %T1% "{ DateTime - Julian to Character DOW }, 2434389"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: \{ PGM Function \}  // Call it
Read Registry String: "ReturnString1"  // Read results into T1 (Sunday)
```

Now here is what the same string would look like using *named* parameters:

```plaintext
Variable Set String %T1% "{ DateTime - Julian to Character DOW }, Julian=2434389"
```

Or:

```plaintext
Variable Set String %T1% "{ DateTime - Julian to Character DOW }, Jn=2434389"
```
{ DateTime - Date to Character DOW }

Converts a date to a character day-of-week (Monday, Tuesday, …).

Parameters
ParameterString1 - Formatted date string

{ PGM Function } Parameters
Date or Dt - Required

Returns
ReturnString1 - Name of the day
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to DOW }
{ DateTime - Character DOW }

Example
This example will convert the formatted date string "19530111" (January 11th, 1953) to a numeric day-of-week 0.

Variable Set String %T1% "19530111"     // Set T1 to a formatted date string
Write Registry String: "ParameterString1"    // Write the date to the Registry
Macro Run: { DateTime - Date to Character DOW }   // Convert to a character day-of-week
Read Registry String: "ReturnString1"     // Read results into T1 (Sunday)

Same example with { PGM Function }:

Variable Set String %T1% "{ DateTime - Date to Character DOW }, 19530111"   // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: { PGM Function }            // Call it
Read Registry String: "ReturnString1"          // Read results into T1 (Sunday)

And using named parameters:

Variable Set String %T1% "{ DateTime - Date to Character DOW }, Date=19530111"

Or:

Variable Set String %T1% "{ DateTime - Date to Character DOW }, Dt=19530111"
{ DateTime - Character DOW }

Converts a day-of-week number (0, 1, 2, ...) to a character day-of-week (Monday, Tuesday, ...). This function uses the SundayStartsOn value to determine the beginning of the week.

Parameters
Parameter Integer1 - Day-of-week number

{ PGM Function } Parameters
DayOfWeek or Dow - Required

Returns
ReturnString1 - Name of the day
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
This example will convert the day-of-week number 3 to a character day-of-week "Wednesday" (SundayStartsOn=0) or "Tuesday" (SundayStartsOn=1).

Variable Set Integer %N1% to 3      // Set N1 to day-of-week 3
Write Registry Integer: "ParameterInteger1"    // Write it to the Registry
Macro Run: { DateTime - Character DOW }    // Convert to a character day-of-week
Read Registry String: "ReturnString1"     // Read results into T1 (Wednesday)

Same example with { PGM Function }:

Variable Set String %T1% "{ DateTime - Character DOW }, 3"  // Create PGM Function string
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }        // Call it
Read Registry String: "ReturnString1"       // Read results into T1 (Wednesday)

And using named parameters:

Variable Set String %T1% "{ DateTime - Character DOW }, DayOfWeek=3"

Or:

Variable Set String %T1% "{ DateTime - Character DOW }, Dow=3"
{ DateTime - Character Month }

Converts a month number (1, 2, 3, …) to a character month (January, February, …).

**Parameters**

Parameter Integer 1 - Month number

**{ PGM Function } Parameters**

Month or Mm - Required

**Returns**

Return String 1 - Name of the month

Return Error? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }

{ Variables - Restore DT }

{ Program - Log Event }

**Example**

This example will convert month number 7 to its name "July".

```
Variable Set Integer %N1% to 7          // Set N1 to a month number
Write Registry Integer: "ParameterInteger1" // Write it to the Registry
Macro Run: { DateTime - Character Month } // Convert to a month name
Read Registry String: "ReturnString1"    // Read results into T1 (July)
```

Same example with `{ PGM Function }`:

```
Variable Set String %T1% "({ DateTime - Character Month }, 7)" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry String: "ReturnString1" // Read results into T1 (July)
```

And using named parameters:

```
Variable Set String %T1% "({ DateTime - Character Month }, Month=7"
```

Or:

```
Variable Set String %T1% "({ DateTime - Character Month }, Mm=7"
```
{ DateTime - Evaluate for Leap Year }

Evaluates a year (2002, 2003, …) to determine if it is a leap year.

**Parameters**

Parameter `Integer1` - Year

**{ PGM Function } Parameters**

Year or Yr - Required

**Returns**

Return `Integer1` - 1 if it is a leap year and 0 if it is not
Return `Error?` - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ Math - Mod }

**Example**

Evaluate the year 2000. It will return 1 because it is a leap year.

```plaintext
Variable Set Integer %N1% to 2000 // Set N1 to the year 2000
Write Registry Integer: "ParameterInteger1" // Write year to the Registry
Macro Run: { DateTime - Evaluate for Leap Year } // Evaluate it
Read Registry Integer: "ReturnInteger1" // Read results into N1 (true)
```

Same example with `{ PGM Function }`:

```plaintext
Variable Set String %T1% "{ DateTime - Evaluate for Leap Year }, 2000" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Integer: "ReturnInteger1" // Read results into N1 (true)
```

And using named parameters:

```plaintext
Variable Set String %T1% "{ DateTime - Evaluate for Leap Year }, Year=2000"
```

Or:

```plaintext
Variable Set String %T1% "{ DateTime - Evaluate for Leap Year }, Yr=2000"
```
{ DateTime - Format Input String }

Converts a date string to a formatted date string (YYYYMMDD), which is used throughout the Date and Time functions.

The year portion of any string must be 4-digits in length. 2-digit years will return an error. Additionally, the days and months must contain leading zeroes. For example; both 01/01/2002 and Jan 01, 2002 are correct while 1/1/2002 and Jan 1, 2002 are incorrect and will return an error.

The following date input string formats will be accepted and converted. They can be found within the Macro Express Date command. Note that the MonthPosition value must be set to 2 for three of the input formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
<th>MonthPosition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM/DD/YYYY</td>
<td>03/19/2000</td>
<td></td>
</tr>
<tr>
<td>MM-DD-YYYY</td>
<td>03-19-2000</td>
<td></td>
</tr>
<tr>
<td>MM.DD.YYYY</td>
<td>03.19.2000</td>
<td></td>
</tr>
<tr>
<td>DD/MM/YYYY</td>
<td>19/03/2000</td>
<td>2</td>
</tr>
<tr>
<td>DD-MM-YYYY</td>
<td>19-03-2000</td>
<td>2</td>
</tr>
<tr>
<td>DD.MM.YYYY</td>
<td>19.03.2000</td>
<td>2</td>
</tr>
<tr>
<td>YYYY/MM/DD</td>
<td>2000/03/19</td>
<td></td>
</tr>
<tr>
<td>YYYY-MM-DD</td>
<td>2000-03-19</td>
<td></td>
</tr>
<tr>
<td>YYYY.MM.DD</td>
<td>2000.03-19</td>
<td></td>
</tr>
<tr>
<td>DD MMMM YYYY</td>
<td>19 MAR 2000</td>
<td></td>
</tr>
<tr>
<td>DD Mmmm YYYY</td>
<td>19 Mar 2000</td>
<td></td>
</tr>
<tr>
<td>DD Mmmmmm YYYY</td>
<td>19 March 2000</td>
<td></td>
</tr>
<tr>
<td>MMM DD YYYY</td>
<td>Mar 19 2000</td>
<td></td>
</tr>
<tr>
<td>MMM DD, YYYY</td>
<td>Mar 19, 2000</td>
<td></td>
</tr>
<tr>
<td>MMMM MMM DD YYYY</td>
<td>March 19 2000</td>
<td></td>
</tr>
<tr>
<td>MMMM MMM DD, YYYY</td>
<td>March 19, 2000</td>
<td></td>
</tr>
<tr>
<td>YYYYMMDD</td>
<td>20000319</td>
<td></td>
</tr>
<tr>
<td>DDMmmYYYY</td>
<td>19Mar2000</td>
<td></td>
</tr>
<tr>
<td>DDMMMYYYY</td>
<td>19MAR2000</td>
<td></td>
</tr>
</tbody>
</table>

Parameters
ParameterString1 - Date string to convert

{ PGM Function } Parameters
Date or Dt - Required

Returns
ReturnString1 - Formatted date string
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Validate Date String }
Example
Convert April 30, 2002 to a formatted date string.

Variable Set String %T1% "April 30, 2002" // Set T1 to a date string
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: ( DateTime - Format Input String ) // Convert to a formatted date string
Read Registry String: "ReturnString1" // Read results into T1 ("20020430")

Convert 30/04/2002 to a formatted date string. Note that the *MonthPosition* value must be set to 2.

Variable Set String %T1% "30/04/2002" // Set T1 to a date string
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: ( DateTime - Format Input String ) // Convert to a formatted date string
Read Registry String: "ReturnString1" // Read results into T1 ("20020430")

Same two examples using { PGM Function }:

Variable Set String %T1% "( DateTime - Format Input String ), "April 30, 2002"
Write Registry String: "PgmFunction" // Call it
Macro Run: { PGM Function }
Read Registry String: "ReturnString1" // Read results into T1 ("20020430")

Variable Set String %T1% "( DateTime - Format Input String ), 30/04/2002"
Write Registry String: "PgmFunction" // Call it
Macro Run: { PGM Function }
Read Registry String: "ReturnString1" // Read results into T1 ("20020430")

And using named parameters:

Variable Set String %T1% "( DateTime - Format Input String ), Date="April 30, 2002"
Or:
Variable Set String %T1% "( DateTime - Format Input String ), Dt="April 30, 2002"

Second example:

Variable Set String %T1% "( DateTime - Format Input String ), Date=30/04/2002"
Or:
Variable Set String %T1% "( DateTime - Format Input String ), Dt=30/04/2002"

Notice that "April 30, 2002" is wrapped in its own set of quotes when using the { PGM Function }. This is because of the comma. Without the quotes, the parser would think that you are passing two parameters April 30 and 2002, which would return an error.
**{ DateTime - Validate Date String }**

Performs a validation check on a formatted date string (YYYYMMDD). It is checked for the number of characters that it contains, which must be eight. The year must be between 0000 and 9999, months between 01 and 12, and days between 01 and 31. If the year is 0000 then the month must be greater than, or equal to, 03. And there must be the proper amount of days within a month, which includes leap years.

**Parameters**

**ParameterString1** - Formatted date string

**{ PGM Function } Parameters**

**Date or Dt** - Required

**Returns**

**ReturnError?** - 0 if date is valid, or 1 if it is not, and which is written to the event log

**Calls**

* { Variables - Save DT }
* { Variables - Restore DT }
* { Program - Log Event }
* { DateTime - Evaluate for Leap Year }

**Example**

Test "19810229" (February 29th, 1981) to determine if it is a valid formatted date string, which will result in not being valid (1).

```plaintext
Variable Set String %T1% "19810229" // Set T1 to a formatted date string
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: ( DateTime - Validate Date String ) // Evaluate it
Read Registry Integer: "ReturnError?" // Read results into N1 (not valid)
```

Using the **{ PGM Function }**:

```plaintext
Variable Set String %T1% "( DateTime - Validate Date String ), 19810229" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: ( PGM Function ) // Call it
Read Registry Integer: "ReturnError?" // Read results into N1 (not valid)
```

And using named parameters:

```plaintext
Variable Set String %T1% "( DateTime - Validate Date String ), Date=19810229"
```

Or:

```plaintext
Variable Set String %T1% "( DateTime - Validate Date String ), Dt=19810229"
```
{ DateTime - Validate Julian Number }

Performs a validation check on a Julian Day number, which must be between 172119.9 (March 1st, 0000) and 5373484.3 (December 31st, 9999). For those of you who may be wondering about the decimal points, Julian Day numbers are not whole numbers. A Julian Day number actually begins at noon (.500), rounding does the rest.

Parameters
ParameterDecimal1 - Julian Day number

{ PGM Function } Parameters
Julian or Jn - Required

Returns
ReturnError? - 0 if number is valid, or 1 if it is not, and which is written to the event log

Calls
{ Variables - Save DT }  
{ Variables - Restore DT }  
{ Program - Log Event }  

Example
The following example tests 1720000 to determine if it is valid Julian Day number, which will result in not being valid (1).

```
Variable Set String %D1% 1720000     // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry
Macro Run: { DateTime - Validate Julian Number }   // Convert to a character day-of-week
Read Registry Integer: "ReturnError?"    // Read results into N1 (not valid)
```

Same example using { PGM Function }:

```
Variable Set String %T1% "{ DateTime - Validate Julian Number }, 1720000"   // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: { PGM Function }            // Call it
Read Registry Integer: "ReturnError?"         / /  R e a d  r e s u l t s  i n t o  N 1  ( n o t  v a l i d )
```

Now here is what the same string would look like using named parameters:

```
Variable Set String %T1% "{ DateTime - Validate Julian Number }, Julian=1720000"
Or:
Variable Set String %T1% "{ DateTime - Validate Julian Number }, Jn=1720000"
```
{ DateTime - Parse Day from Date }

Extracts the day-of-month portion of a formatted date string and returns it as an integer.

**Parameters**
ParameterString1 - Formatted date string

**PGM Function Parameters**
Date or Dt - Required

**Returns**
ReturnInteger1 - Numeric day-of-month
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**
- { Variables - Save DT }
- { Variables - Restore DT }
- { Program - Log Event }
- { DateTime - Validate Date String }

**Example**
This example will extract the numeric day-of-month from the formatted date string "20020815" (August 15\(^{th}\), 2002), which will be 15.

Variable Set String %T1% "20020815" // Set T1 to a formatted date string
Write Registry String: "ParameterString1" // Write date to the Registry
Macro Run: { DateTime - Parse Day from Date } // Extract the day-of-month
Read Registry Integer: "ReturnInteger1" // Read results into N1 (day 15)

Same example with **PGM Function**:

Variable Set String %T1% "{ DateTime - Parse Day from Date }, 20020815" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Integer: "ReturnInteger1" // Read results into N1 (day 15)

And using named parameters:

Variable Set String %T1% "{ DateTime - Parse Day from Date }, Date=20020815"

Or:

Variable Set String %T1% "{ DateTime - Parse Day from Date }, Dt=20020815"
{ DateTime - Parse Day from Julian }

Converts a Julian Day number to a formatted date string then extracts the day-of-month portion and returns it as an integer.

Parameters
ParameterDecimal1  - Julian Day number

{ PGM Function } Parameters
Julian or Jn       - Required

Returns
ReturnInteger1     - Numeric day-of-month
ReturnError?       - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }  
{ Variables - Restore DT }  
{ Program - Log Event }  
{ DateTime - Julian to Date }  
{ DateTime - Parse Day from Date }  

Example
This example will extract the numeric day-of-month from the Julian Day number 2452502 (August 15th, 2002), which will be 15.

Variable Set String %D1% 2452502     // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry
Macro Run: { DateTime - Parse Day from Julian }   // Extract the day-of-month
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (day 15)

Same example using { PGM Function }:

Variable Set String %T1% "{ DateTime - Parse Day from Julian }, 2452502"   // Create PGM Function string
Write Registry String: "PgmFunction"         // Write it to the Registry
Macro Run: { PGM Function }           // Call it
Read Registry Integer: "ReturnInteger1"        // Read results into N1 (day 15)

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ DateTime - Parse Day from Julian }, Julian=2452502"

Or:

Variable Set String %T1% "{ DateTime - Parse Day from Julian }, Jn=2452502"
{ DateTime - Parse Month from Date }

Extracts the month portion of a formatted date string and returns it as an integer.

Parameters
ParameterString1 - Formatted date string

{ PGM Function } Parameters
Date or Dt - Required

Returns
ReturnInteger1 - Numeric month
ReturnValue? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Validate Date String }

Example
This example will extract the numeric month from the formatted date string "20020815" (August 15th, 2002), which will be 8.

```
Variable Set String %T1% "20020815"     // Set T1 to a formatted date string
Write Registry String: "ParameterString1"    // Write date to the Registry
Macro Run: { DateTime - Parse Month from Date }   // Extract the month
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (month 8)
```

Same example with { PGM Function }:

```
Variable Set String %T1% "{ DateTime - Parse Month from Date }, 20020815"   // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: { PGM Function }          // Call it
Read Registry Integer: "ReturnValue1"         // Read results into N1 (month 8)
```

And using named parameters:

```
Variable Set String %T1% "{ DateTime - Parse Month from Date }, Date=20020815"
```

Or:

```
Variable Set String %T1% "{ DateTime - Parse Month from Date }, Dt=20020815"
```
{ DateTime - Parse Month from Julian }

Converts a Julian Day number to a formatted date string then extracts the month portion and returns it as an integer.

Parameters
ParameterDecimal1 - Julian Day number

{ PGM Function } Parameters
Julian or Jn - Required

Returns
ReturnInteger1 - Numeric month
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }  
{ Variables - Restore DT }  
{ Program - Log Event }  
{ DateTime - Julian to Date }  
{ DateTime - Parse Month from Date }

Example
This example will extract the numeric month from the Julian Day number 2452502 (August 15th, 2002), which will be 8.

Variable Set String %D1% 2452502 // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1" // Write D1 to the Registry
Macro Run: { DateTime - Parse Month from Julian } // Extract the month
Read Registry Integer: "ReturnInteger1" // Read results into N1 (month 8)

Same example using { PGM Function }:

Variable Set String %T1% "{ DateTime - Parse Month from Julian }, 2452502" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Integer: "ReturnInteger1" // Read results into N1 (month 8)

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ DateTime - Parse Month from Julian }, Julian=2452502"

Or:
Variable Set String %T1% "{ DateTime - Parse Month from Julian }, Jn=2452502"
{ DateTime - Parse Year from Date }

Extracts the year portion of a formatted date string and returns it as an integer.

Parameters
ParameterString1 - Formatted date string

{ PGM Function } Parameters
Date or Dt - Required

Returns
ReturnInteger1 - Year
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Validate Date String }

Example
This example will extract the year from the formatted date string "20020815" (August 15th, 2002), which will be 2002.

Variable Set String %T1% "20020815" // Set T1 to a formatted date string
Write Registry String: "ParameterString1" // Write date to the Registry
Macro Run: { DateTime - Parse Year from Date } // Extract the year
Read Registry Integer: "ReturnInteger1" // Read results into N1 (2002)

Same example with { PGM Function }:

Variable Set String %T1% "( DateTime - Parse Year from Date ), 20020815" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Integer: "ReturnInteger1" // Read results into N1 (2002)

And using named parameters:

Variable Set String %T1% "{ DateTime - Parse Year from Date }, Date=20020815"

Or:

Variable Set String %T1% "{ DateTime - Parse Year from Date }, Dt=20020815"
**{ DateTime - Parse Year from Julian }**

Converts a Julian Day number to a formatted date string then extracts the year portion and returns it as an integer.

**Parameters**
ParameterDecimal1 - Julian Day number

**{ PGM Function } Parameters**
Julian or Jn - Required

**Returns**
ReturnInteger1 - Year  
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**
{ Variables - Save DT }  
{ Variables - Restore DT }  
{ Program - Log Event }  
{ DateTime - Julian to Date }  
{ DateTime - Parse Year from Date }

**Example**
This example will extract the year from the Julian Day number 2452502 (August 15th, 2002), which will be 2002.

```
Variable Set String %D1% 2452502     // Set D1 to a Julian Day number
Write Registry Decimal: "ParameterDecimal1"  // Write D1 to the Registry
Macro Run: { DateTime - Parse Year from Julian }   // Extract the year
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (2002)
```

Same example using **{ PGM Function }**:

```
Variable Set String %T1% "{ DateTime - Parse Year from Julian }, 2452502"   // Create PGM Function string
Write Registry String: "PgmFunction"         // Write it to the Registry
Macro Run: { PGM Function }           // Call it
Read Registry Integer: "ReturnInteger1"        // Read results into N1 (2002)
```

Now here is what the same string would look like using named parameters:

```
Variable Set String %T1% "{ DateTime - Parse Year from Julian }, Julian=2452502"
```

Or:

```
Variable Set String %T1% "{ DateTime - Parse Year from Julian }, Jn=2452502"
```
\{ Date\Time - Parse Day from Today \}

Extracts the day-of-month portion of a today's date and returns it as an integer.

**Parameters**
None.

**Returns**
- \texttt{ReturnInteger1} - Numeric day-of-month
- \texttt{ReturnError?} - 0 if no error occurs, or 1 if an error occurs

**Calls**
- \{ Variables - Save DT \}
- \{ Variables - Restore DT \}
- \{ Program - Log Event \}
- \{ Date\Time - Parse Day from Date \}

**Example**
Extract the numeric day-of-month from today's date (assume it is August 15\textsuperscript{th}, 2002).

\begin{verbatim}
Macro Run: \{ Date\Time - Parse Day from Today \}  // Extract the day-of-month
Read Registry Integer: "ReturnInteger1"        // Read results into N1 (day 15)
\end{verbatim}
{ DateTime - Parse Month from Today }

Extracts the month portion of a today's date and returns it as an integer.

Parameters
None.

Returns
ReturnInteger1 - Numeric month
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Parse Month from Date }

Example
Extract the numeric month from today's date (assume it is August 15th, 2002).

    Macro Run: { DateTime - Parse Month from Today }   // Extract the month
    Read Registry Integer: "ReturnInteger1"            // Read results into N1 (month 8)
{ DateTime - Parse Year from Today }

Extracts the year portion of a today's date and returns it as an integer.

**Parameters**
None.

**Returns**
ReturnInteger1 - Year
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Parse Year from Date }

**Example**
Extract the year from today's date (assume it is August 15\(^{th}\), 2002).

Macro Run: { DateTime - Parse Year from Today } // Extract the year
Read Registry Integer: "ReturnInteger1" // Read results into N1 (2002)
{ DateTime - Today to DOW }

Converts today’s date to a numeric day-of-week (0, 1, 2, …).

Parameters
None.

Returns
ReturnInteger1 - Numeric day-of-week
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to DOW }

Example
Convert today to a numeric day-of-week (assume today is Thursday, May 2, 2002).

Macro Run: { DateTime - Today to DOW }     // Convert today to a numeric day-of-week
Read Registry Integer: "ReturnInteger1"    // Read results into N1 (day 4)
{ DateTime - Today to Character DOW }

Converts today’s date to a character day-of-week (Monday, Tuesday, ...).

Parameters
None.

Returns
ReturnString1 - Name of the day
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to Character DOW }

Example
Convert today to a character day-of-week (assume today is Thursday, May 2, 2002).

Macro Run: { DateTime - Today to Character DOW }   // Convert today to a character day-of-week
Read Registry String: "ReturnString1"          // Read results into T1 (Thursday)
{ DateTime - Today to DOY }

Converts today’s date to a numeric day-of-year (120, 200, 365, …).

**Parameters**

None.

**Returns**

- ReturnInteger1 - Numeric day-of-year
- ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**

- { Variables - Save DT }
- { Variables - Restore DT }
- { Program - Log Event }
- { DateTime - Date to DOY }

**Example**

Convert today to a numeric day-of-year (assume today is Thursday, May 2, 2002).

```
Macro Run: { DateTime - Today to DOY }  // Convert today to a numeric day-of-year
Read Registry Integer: "ReturnInteger1"  // Read results into N1 (day 122)
```
{ DateTime - Today to Julian }

Converts today's date to a Julian Day number.

Parameters
None.

Returns
ReturnDecimal1 - Julian Day number
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ DateTime - Date to Julian }

Example
Convert today to a numeric day-of-year (assume today is Thursday, May 2, 2002).

Macro Run: { DateTime - Today to Julian } // Convert today to a Julian Day number
Read Registry Decimal: "ReturnDecimal1" // Read results into D1 (2452397)
{ DateTime - Reset Defaults }

Resets the *SundayStartsOn* and *MonthPosition* Registry values back to their default values (0 and 1 respectively).

**Parameters**

None.

**Returns**

Nothing.

**Calls**

Nothing.

**Example**

```plaintext
Macro Run: { DateTime - Reset Defaults }  // Reset default values
Read Registry Integer: "SundayStartsOn"  // N1 = 0
Read Registry Integer: "MonthPosition"   // N2 = 1
```
{ DateTime - Set Defaults }

Interactively set the SundayStartsOn and MonthPosition Registry values.

Parameters
None.

Returns
Nothing.

Calls
Nothing.

Example

Macro Run: { DateTime - Set Defaults } // Set default values interactively
I never have found the perfect quote. At best I have been able to find a string of quotations which merely circle the ineffable idea I seek to express.

- Caldwell O'Keefe
Overview

The string functions here enhance what Macro Express has provided for us all. We have found on numerous occasions that we were repeating code within string handling routines for various tasks, so we created reusable functions instead.

- Parse a string into separate words based on any delimiter character, or delimiter string, even whitespace.
- Retrieve the position of one string inside another, from the beginning or the end, or any other point of origin.
- Count the number of occurrences of one string inside another.
- Reverse a string. “Abc” to “cbA”.
- Pad a string with any character, or other string, left, right, or center.
- Convert a string to proper case (capital letters).
- Replicate a character, or string, an unlimited amount of times.

Even though some of these functions appear to do the same thing as the built-in Macro Express commands, each has major differences. For example, where Macro Express allows padding a string on either the left or right sides, these functions will center a string and, if chosen, will use a character other than a space for padding. You can search for a specific occurrence of a string within a string, and not be limited to just the first one. Instead of changing a string to all upper or lower case, each individual word can be capitalized. See the individual function for more differences.
Registry

When the PGM Functions Library™ was first installed on your computer, an area called Parameters was created in your Registry within the Professional Grade Macros key along with another area called Parsed Tokens. These areas, or keys, contain many values, which are used to communicate with the String functions via the Write Registry String, Write Registry Integer, Read Registry String, and Read Registry Integer commands. The Name column, in the right hand pane, is the value name. The Type column is the type of variable being stored, REG_SZ for strings and REG_DWORD for integers. And of course the Data column shows the actual value being stored.
The Parsed Tokens section contains the results of parsing a string with the `{String·Parse}` command.
Let's examine in detail these Registry values and how you use them with the String functions. They can be grouped into three categories:

1. Input values - You set these.
2. Return values - Set by the functions.
3. Parsed values - Used only by the {String - Parse} command to hold results.

**Input Values**

These are values that you write to the Registry prior to calling any of the String functions. The value depends on which function you are calling. For example, if you wanted to count the number of occurrences of one string inside another, you would write the string-to-search to the `ParameterString1` value, the string-to-count to the `ParameterString2` value, then read the results from the `ReturnInteger1` value.

Unlike the Program Operations functions, where each value had a distinct purpose (ProgramName, WindowTitle, etc.), these values are generic placeholders, meaning that they can be, and are, used by many functions to perform many different actions. There is never any guarantee that a particular value will always be used for a particular action or purpose. It really does depend entirely on whichever function you are referencing at the time.

**ParameterString1**

Usually this value contains whatever string you are going to do something to. In other words, the object or target string. You could reverse, for example, an alphabet string using the {String - Reverse} function.

```plaintext
Variable Set String %T1% to "Abcdefg" // Set T1 to an alphabet string
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: { String - Reverse } // Reverse it
Read Registry String: "ReturnString1" // Read the results, T1 = "gfedcbA"
```

**ParameterString2**

Holds a second string value depending on the function that you are calling. An example would be the {String - Occurs} function, which requires two string values, one to search and one to count.

```plaintext
Variable Set String %T1% to "The quick brown fox" // Set T1 to a string to search
Variable Set String %T2% to "fox" // Set T2 to a string to count
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry String: "ParameterString2" // Write T2 to the Registry
Macro Run: { String - Occurs } // Count the occurrences
Read Registry Integer: "ReturnInteger1" // Read the results, N1 = 1
```
ParameterString3
Holds a third string value depending on the function that you are calling. An example would be the \{String - Pad\} function, which requires three string values, one for what to pad, one for how to pad, and one for what to pad with.

\begin{verbatim}
Variable Set String %T1% to "Pnambic"    // Set T1 to a string to pad
Variable Set String %T2% to "B"          // Set T2 to pad both ends
Variable Set String %T3% to "$"          // Set T3 to a pad character
Variable Set Integer %N1% to 11           // Set N1 to a pad length
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry String: "ParameterString2" // Write T2 to the Registry
Write Registry String: "ParameterString3" // Write T3 to the Registry
Macro Run: { String - Occurs }          // Count the occurrences
Read Registry String: "ReturnString1"    // Read the results, T1 = "$$Pnambic$$"
\end{verbatim}

ParameterInteger1
Holds an integer value for size, position, count, etc., depending on the function that you are calling. You could replicate, for example, a string by using the \{String - Replicate\} function.

\begin{verbatim}
Variable Set String %T1% to "Pnambic"    // Set T1 to a string to replicate
Variable Set Integer %N1% to 2            // Set N1 as to how many times
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Macro Run: { String - Replicate }        // Replicate it
Read Registry String: "ReturnString1"    // Read the results, T1 = "PnambicPnambic"
\end{verbatim}

ParameterInteger2
Holds a second integer value depending on the function that you are calling. For example, the \{String - Parse\} function requires two integer values, one to designate the first token and the other for the number of tokens.

\begin{verbatim}
Variable Set String %T1% to "Paul Thornett Pnambic Systems"    // Set T1 to a string to parse
Variable Set String %T2% to " "                                // Set T2 to a space
Variable Set Integer %N1% to 1                                 // Set N1 to the first token
Variable Set Integer %N2% to 4                                 // Set N2 as all the tokens
Write Registry String: "ParameterString1"                     // Write T1 to the Registry
Write Registry String: "ParameterString2"                     // Write T2 to the Registry
Write Registry Integer: "ParameterInteger1"                   // Write N1 to the Registry
Write Registry Integer: "ParameterInteger2"                   // Write N2 to the Registry
Macro Run: { String - Parse }                                 // Parse the string into tokens
\end{verbatim}
ParameterInteger3

Holds a third integer value depending on the function you are calling. For example, the `{String - Search}` function uses three integer values, one for a starting position, one for which occurrence, and one to hold a case-sensitive search flag

- Variable Set String %T1% to "The quick brown fox" // Set T1 to a string to search
- Variable Set String %T2% to "fox" // Set T2 to a string to count
- Variable Set Integer %N1% to 1 // Set N1 to the first character
- Variable Set Integer %N2% to 1 // Set N2 to the first occurrence
- Variable Set Integer %N3% to 0 // Set N3 as case-insensitive
- Write Registry String: "ParameterString1" // Write T1 to the Registry
- Write Registry String: "ParameterString2" // Write T2 to the Registry
- Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
- Write Registry Integer: "ParameterInteger2" // Write N2 to the Registry
- Write Registry Integer: "ParameterInteger3" // Write N3 to the Registry
- Macro Run: {String - Search} // Locate the position
- Read Registry Integer: "ReturnInteger1" // Read the results, N1 = 17
Return Values

The resulting calculations performed by the String functions are placed in these values. You would read one of them, depending on which function you have called, for your answer. For example, if you wanted to reverse a string, you would write the string to the ParameterString1 value, then read the results from the ReturnString1 value.

ReturnString1
Contains the results of any function which returns a string value. The {String - Reverse} function is a prime example.

```
Variable Set String %T1% to "Abcdefg" // Set T1 to an alphabet string
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: { String - Reverse } // Reverse it
Read Registry String: "ReturnString1" // Read the results, T1 = “gfedcba"
```

ReturnInteger1
Contains the results of any function that returns an integer value such as the {String - Occurs} function.

```
Variable Set String %T1% to "The quick brown fox" // Set T1 to a string to search
Variable Set String %T2% to "fox" // Set T2 to a string to count
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry String: "ParameterString2" // Write T2 to the Registry
Macro Run: { String - Occurs } // Count the occurrences
Read Registry Integer: "ReturnInteger1" // Read the results, N1 = 1
```

ReturnInteger2
Contains the results of any function that returns a second integer value such as the {String - Parse} function, which returns both the total number of tokens found and the number which were parsed.

```
Variable Set String %T1% to "Paul Thornett Pnambic Systems" // Set T1 to a string to parse
Variable Set String %T2% to " " // Set T2 to a space
Variable Set Integer %N1% to 1 // Set N1 to the first token
Variable Set Integer %N2% to 4 // Set N2 as all the tokens
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry String: " ParameterString2 " // Write T2 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Write Registry Integer: "ParameterInteger2" // Write N2 to the Registry
Macro Run: { String - Parse } // Parse the string into tokens
Read Registry Integer: "ReturnInteger1" // Read total tokens., N1 = 4
Read Registry Integer: "ReturnInteger2" // Read parsed tokens, N2 = 4
```
ReturnError?
This value will tell you if an error has occurred in one of the String functions. If the value is 1 then an error occurred, and a 0 means that the function ran fine. This value should be tested after a function returns from processing so appropriate action can be taken by your macro. If an error occurs and, if event logging is enabled (LogEvent?), then the actual error will be contained in the event log.

Macro Run: { String - Search } // Locate the position
Read Registry Integer: "ReturnError?" // Check for an error
If Variable %N1% = 1
  Text Box Display: Error! // If so, then prompt the user
End If
Parsed Values

The \{String - Parse\} function can return up to 90 tokens when parsing a string, which are written to the Parsed Tokens key in the Registry (see the picture at the beginning of this section). The values are named \textit{Token1} through \textit{Token90}.

The example here will parse a string, then write the results to \textit{Token1} through \textit{Token4} in the Registry. A text window is shown below with those same values read from the results into variables.

```
Variable Set String %T1% to "Paul Thornett Pnambic Systems"  // Set T1 to a string to parse
Variable Set String %T2% to " "                           // Set T2 to a space
Variable Set Integer %N1% to 1                             // Set N1 to the first token
Variable Set Integer %N2% to 4                             // Set N2 as all the tokens
Write Registry String: "ParameterString1"                  // Write T1 to the Registry
Write Registry String: "ParameterString2"                  // Write T2 to the Registry
Write Registry Integer: "ParameterInteger1"                // Write N1 to the Registry
Write Registry Integer: "ParameterInteger2"                // Write N2 to the Registry
Macro Run: { String - Parse }                              // Parse the string into tokens
Read Registry Integer: "ReturnInteger1"                    // Read total tokens., N1 = 4
Read Registry Integer: "ReturnInteger2"                    // Read parsed tokens, N2 = 4
```
Functions

The string functions here enhance what Macro Express has provided for us all. We have found on numerous occasions that we were repeating code within string handling routines for various tasks, so we created reusable functions instead.

The String category is composed of ten functions. Even though some of these functions appear to do the same thing as the built-in Macro Express commands, each has major differences. For example, where Macro Express allows padding a string on either the left or right sides, these functions will center a string and, if chosen, will use a character other than a space for padding. You can search for a specific occurrence of a string within a string, and not be limited to just the first one. Instead of changing a string to all upper or lower case, each individual word can be capitalized. See the individual function for more differences.

{ String - Occurs } Returns the number of occurrences of a one string inside another string
{ String - Pad } Pads a string on the left or right sides, or centers with a specific pad character
{ String - Pad Center } Centers a string (pads both ends) using a specific pad character
{ String - Pad Left } Pads a string on the left side using a specific pad character
{ String - Pad Right } Pads a string on the right side using a specific pad character
{ String - Parse } Parses an input string based on a character or string delimiter
{ String - Proper } Converts a string to proper case where each word begins with a capital letter
{ String - Replicate } Replicates a string
{ String - Reverse } Reverses a string
{ String - Search } Returns the starting position of one string inside another string

They can all be called by {PGM Function}. As with many of our functions, if an error occurs you'll be notified via the ReturnError? value and, if event logging is enabled, the actual error will be recorded in the log file. For the sake of clarity, the examples here do not check for an error, however, it is recommended that you do check, so appropriate action can be taken.
**{ String - Occurs }**

Returns the number of occurrences of a one string inside another.

**Parameters**
- ParameterString1: String to search in
- ParameterString2: String to count
- ParameterInteger1: Case-sensitive search (0=false 1=true). Defaults to false

**{ PGM Function } Parameters**
- SearchIn or In: Required
- SearchFor or For: Required
- CaseSensitive? Or Case: Set to zero if omitted

**Returns**
- ReturnInteger1: Number of occurrences found
- ReturnError?: 0 if no error occurs, or 1 if an error occurs

**Calls**
- { Variables - Save DT }
- { Variables - Restore DT }
- { Program - Log Event }

**Example**
Count the number of times that the number "0" occurs in the string. The answer is 8.

```plaintext
Variable Set String %T1% "000234062006925010"    // String to search
Variable Set String %T2% to "0"       // String to count
Write Registry String: "ParameterString1" // Write search string to the Registry
Write Registry String: "ParameterString2" // Write count string to the Registry
Macro Run: { String - Occurs }       // Count the occurrences
Read Registry Integer: "ReturnInteger1"   // Read the results, N1 = 8
```

Count the number of times that "Fox" occurs in the string using a case-sensitive search. The answer is 0.

```plaintext
Variable Set String %T1% to "The quick brown fox" // String to search
Variable Set String %T2% to "Fox"        // String to count
Variable Set Integer %N1% to 1           // Case-sensitive
Write Registry String: "ParameterString1" // Write search string to the Registry
Write Registry String: "ParameterString2" // Write count string to the Registry
Write Registry Integer: "ParameterInteger1" // Write case integer to the Registry
Macro Run: { String - Occurs }       // Count the occurrences
Read Registry Integer: "ReturnInteger1"   // Read the results, N1 = 0
```
Same two examples using \{ PGM Function \}:

\begin{verbatim}
Variable Set String %T1% "{ String - Occurs }, 000234062006925010, 0"  // Create PGM Function string
Write Registry String: "PgmFunction"   // Write it to the Registry
Macro Run: { PGM Function }           // Call it
Read Registry Integer: "ReturnInteger1"  // Read the results, N1 = 8

Variable Set String %T1% "{ String - Occurs }, The quick brown fox, Fox, 1"   // Create PGM Function string
Write Registry String: "PgmFunction"   // Write it to the Registry
Macro Run: { PGM Function }           // Call it
Read Registry Integer: "ReturnInteger1"  // Read the results, N1 = 8
\end{verbatim}

Now here is what the same string would look like using \textit{named} parameters:

\begin{verbatim}
Variable Set String %T1% "{ String - Occurs }, SearchIn = 000234062006925010, SearchFor = 0"
Or:
Variable Set String %T1% "{ String - Occurs }, In = 000234062006925010, For = 0"

Second example:

Variable Set String %T1% "{ String - Occurs }, SearchIn = The quick brown fox, SearchFor = Fox, CaseSensitive? = 1"
Or:
Variable Set String %T1% "{ String - Occurs }, In = The quick brown fox, For = Fox, Case = 1"
\end{verbatim}
{ String - Pad }

Pads a string on the left or right side, or centers it, to a specific length using a specified character. The pad character can only be a single character, nothing longer. If you don’t set a pad character then a space will be used for padding.

When centering a string (pad direction = "B" both), if it is determined that the number of pad characters will be an odd value, then the trailing end of the final string will contain the extra pad character.

Parameters
ParameterString1  - String to pad
ParameterInteger1 - Length to pad string to
ParameterString2  - Pad direction; "L" left, "R" right, "B" both (centered)
ParameterString3  - Character to use for padding

{ PGM Function } Parameters
PadString or Str   - Required
FinalLength or Len - Required
PadDirection or Dir - Set to "L" if omitted (pad left)
PadCharacter or Pad - Set to a space if omitted

Returns
ReturnString1   - The input string padded to the specified length
ReturnError?    - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Pad the string "Pnambic" with spaces on the left side to a length of 11 characters.

```
Variable Set String %T1% to "Pnambic"  // String to pad
Variable Set Integer %N1% to 11        // Pad length of 11
Write Registry String: "ParameterString1"  // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1"  // Write N1 to the Registry
Macro Run: { String - Pad }  // Pad it
Read Registry String: "ReturnString1"    // Read the results, T1 = "    Pnambic"
```

Center the string "Pnambic" with spaces to a length of 11 characters.

```
Variable Set String %T1% to "Pnambic"  // String to pad
Variable Set Integer %N1% to 11        // Pad length of 11
Variable Set String %T2% to "B"         // Pad both ends (center string)
Write Registry String: "ParameterString1"  // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1"  // Write N1 to the Registry
Write Registry String: "ParameterString2"  // Write T2 to the Registry
Write Registry Integer: "ParameterInteger2"  // Write N2 to the Registry
Read Registry String: "ReturnString1"  // Read the results, T1 = "    Pnambic"
```
Pad the string "Pnambic" with "$" on the right side to a length of 11 characters.

Variable Set String %T1% to "Pnambic"
Variable Set Integer %N1% to 11
Variable Set String %T2% to "R"
Variable Set String %T3% to "$"
Write Registry String: "ParameterString1"
Write Registry Integer: "ParameterInteger1"
Write Registry String: "ParameterString2"
Write Registry String: "ParameterString3"
Macro Run: { String - Pad }       // Pad it
Read Registry String: "ReturnString1"     // Read the results, T1 = “Pnambic $$$"

Same three examples using { PGM Function }:

Variable Set String %T1% "{ String - Pad }, Pnambic, 11"   // Pad left side
Write Registry String: "PgmFunction"       // Write it to the Registry
Macro Run: { PGM Function }         // Call it
Read Registry String: "ReturnString1"       // Read the results, T1 = “ Pnambic ”

Variable Set String %T1% "{ String - Pad }, Pnambic, 11, B"   // Pad both ends (center string)
Write Registry String: "PgmFunction"       // Write it to the Registry
Macro Run: { PGM Function }         // Call it
Read Registry String: "ReturnString1"       // Read the results, T1 = “ Pnambic ”

Variable Set String %T1% "{ String - Pad }, Pnambic, 11, R, $"   // Pad right side with "$"
Write Registry String: "PgmFunction"       // Write it to the Registry
Macro Run: { PGM Function }         // Call it
Read Registry String: "ReturnString1"       // Read the results, T1 = “Pnambic$$$$”

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ String - Pad }, PadString = Pnambic, FinalLength = 11"   // Pad left side
Or:
Variable Set String %T1% "{ String - Pad }, Str = Pnambic, Len = 11"

Second example:

Variable Set String %T1% "{ String - Pad }, PadString = Pnambic, FinalLength = 11, PadDirection = B"   // Pad both ends
Or:
Variable Set String %T1% "{ String - Pad }, Str = Pnambic, Len = 11, Dir = B"

Third example:

Variable Set String %T1% "{ String - Pad }, PadString = Pnambic, FinalLength = 11, PadDirection = B, PadCharacter = "$"
Or:
Variable Set String %T1% "{ String - Pad }, Str = Pnambic, Len = 11, Dir = B, Pad = $"
{ String - Pad Center }

Centers a string (pads both sides) to a specific length using a specified character. The pad character can only be a single character, nothing longer. If you don't set a pad character then a space will be used for padding. When centering a string, if it is determined that the number of pad characters will be an odd value, then the trailing end of the final string will contain the extra pad character.

Parameters
ParameterString1 - String to pad
ParameterInteger1 - Length to pad string to
ParameterString2 - Character to use for padding

{ PGM Function } Parameters
PadString or Str - Required
FinalLength or Len - Required
PadCharacter or Pad - Set to a space if omitted

Returns
ReturnString1 - The input string centered to the specified length
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ String - Pad }

Example
Center the string "Pnambic" with spaces to a length of 11 characters.

Variable Set String %T1% to "Pnambic" // String to center
Variable Set Integer %N1% to 11 // Pad length of 11
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Macro Run: { String - Pad Center }
Read Registry String: "ReturnString1" // Read the results, T1 = "  Pnambic  

Same example using { PGM Function }:

Variable Set String %T1% "{ String - Pad Center }, Pnambic, 11" // Pad both ends (center string)
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry String: "ReturnString1" // Read the results, T1 = "  Pnambic  

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ String - Pad Center }, PadString = Pnambic, FinalLength = 11" // Pad both ends (center string)
{ String - Pad Left }

Pads a string on the left side to a specific length using a specified character. The pad character can only be a single character, nothing longer. If you don't set a pad character then a space will be used for padding.

Parameters
ParameterString1 - String to pad
ParameterInteger1 - Length to pad string to
ParameterString2 - Character to use for padding

{ PGM Function } Parameters
PadString or Str - Required
FinalLength or Len - Required
PadCharacter or Pad - Set to a space if omitted

Returns
ReturnString1 - The input string padded on the left to the specified length
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ String - Pad }

Example
Pad the string "Pnambic" with spaces on the left side to a length of 11 characters.

```
Variable Set String %T1% to "Pnambic"     // String to pad
Variable Set Integer %N1% to 11      // Pad length of 11
Write Registry String: "ParameterString1"  // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Macro Run: { String - Pad Left }      // Pad it
Read Registry String: "ReturnString1"     // Read the results, T1 = "    Pnambic"
```

Pad the string "Pnambic" with "$" on the left side to a length of 11 characters.

```
Variable Set String %T1% to "Pnambic"     // String to pad
Variable Set Integer %N1% to 11      // Pad length of 11
Variable Set String %T2% to "$"      // Use "$" for padding
Write Registry String: "ParameterString1"  // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Write Registry String: "ParameterString2"  // Write T2 to the Registry
Macro Run: { String - Pad Left }      // Pad it
Read Registry String: "ReturnString1"     // Read the results, T1 = "$$$$Pnambic"
```
Same two examples using \{ PGM Function \}:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Set String %T1%&quot;{ String - Pad Left }, Pnambic, 11&quot;</td>
<td>Pad left side</td>
</tr>
<tr>
<td>Write Registry String: &quot;PgmFunction&quot;</td>
<td>Write it to the Registry</td>
</tr>
<tr>
<td>Macro Run: { PGM Function }</td>
<td>Call it</td>
</tr>
<tr>
<td>Read Registry String: &quot;ReturnString1&quot;</td>
<td>Read the results, T1 = “ Pnambic”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Set String %T1%&quot;{ String - Pad Left }, Pnambic, 11, $&quot;</td>
<td>Pad left side with &quot;$&quot;</td>
</tr>
<tr>
<td>Write Registry String: &quot;PgmFunction&quot;</td>
<td>Write it to the Registry</td>
</tr>
<tr>
<td>Macro Run: { PGM Function }</td>
<td>Call it</td>
</tr>
<tr>
<td>Read Registry String: &quot;ReturnString1&quot;</td>
<td>Read the results, T1 = “$$$$Pnambic”</td>
</tr>
</tbody>
</table>

Now here is what the same string would look like using named parameters:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Set String %T1%&quot;{ String - Pad Left }, PadString = Pnambic, FinalLength = 11&quot;</td>
<td>Pad left side</td>
</tr>
<tr>
<td>Or:</td>
<td></td>
</tr>
<tr>
<td>Variable Set String %T1%&quot;{ String - Pad Left }, Str = Pnambic, Len = 11&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Second example:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Set String %T1%&quot;{ String - Pad Left }, PadString = Pnambic, FinalLength = 11, PadCharacter = $&quot;</td>
<td>Pad with &quot;$&quot;</td>
</tr>
<tr>
<td>Or:</td>
<td></td>
</tr>
<tr>
<td>Variable Set String %T1%&quot;{ String - Pad Left }, Str = Pnambic, Len = 11, Pad = $&quot;</td>
<td></td>
</tr>
</tbody>
</table>
**{ String - Pad Right }**

Pads a string on the right side to a specific length using a specified character. The pad character can only be a single character, nothing longer. If you don't set a pad character then a space will be used for padding.

**Parameters**

ParameterString1  - String to pad
ParameterInteger1  - Length to pad string to
ParameterString2  - Character to use for padding

**{ PGM Function } Parameters**

PadString or Str    - Required
FinalLength or Len  - Required
PadCharacter or Pad - Set to a space if omitted

**Returns**

ReturnString1   - The input string padded on the right to the specified length
ReturnError?    - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ String - Pad }

**Example**

Pad the string "Pnambic" with spaces on the right side to a length of 11 characters.

```plaintext
Variable Set String %T1% to "Pnambic" // String to pad
Variable Set Integer %N1% to 11 // Pad length of 11
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Macro Run: { String - Pad Right } // Pad it
Read Registry String: "ReturnString1" // Read the results, T1 = "Pnambic    
```

Pad the string "Pnambic" with "$" on the right side to a length of 11 characters.

```plaintext
Variable Set String %T1% to "Pnambic" // String to pad
Variable Set Integer %N1% to 11 // Pad length of 11
Variable Set String %T2% to "$" // Use "$" for padding
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Write Registry String: "ParameterString2" // Write T2 to the Registry
Macro Run: { String - Pad Right } // Pad it
Read Registry String: "ReturnString1" // Read the results, T1 = "Pnambic$$" 
```
Same two examples using `{PGM Function}`:

```
Variable Set String %T1% "{ String - Pad Right }, Pnambic, 11"  // Pad right side
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: { PGM Function }  // Call it
Read Registry String: "ReturnString1"  // Read the results, T1 = “Pnambic ”

Variable Set String %T1% "{ String - Pad Right }, Pnambic, 11, $"  // Pad right side with "$"
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: { PGM Function }  // Call it
Read Registry String: "ReturnString1"  // Read the results, T1 = “Pnambic$$$$”
```

Now here is what the same string would look like using named parameters:

```
Variable Set String %T1% "{ String - Pad Right }, PadString = Pnambic, FinalLength = 11"  // Pad right
Or:
Variable Set String %T1% "{ String - Pad Right }, Str = Pnambic, Len = 11"

Second example:

```
Variable Set String %T1% "{ String - Pad Right }, PadString = Pnambic, FinalLength = 11, PadCharacter = $"  // Pad with "$"
Or:
Variable Set String %T1% "{ String - Pad Right }, Str = Pnambic, Len = 11, Pad = "$"
```
**String - Parse**

Parses an input string based on a character, or string delimiter, and returns each token (word or a series of words) as separate variables. Up to 90 tokens can be returned, which are written to the *Parsed Tokens* key in the Registry. The values are named *Token1* through *Token90*.

Parsing is done from either left-to-right, or right-to-left, starting with whichever token you designate in *ParameterInteger1*, and continues until either the number of tokens in *ParameterInteger2* is reached, or the string runs out of tokens, or the maximum number of 90 is reached. The default values for these two settings is zero, which means that you want to parse all tokens, left-to-right, from the beginning of the string. If a value greater than the number of tokens the string contains is entered as the starting number (*ParameterInteger1*) then the largest token number is used instead. If, for example, there are 3 tokens in a string, and you enter 4 as the starting number, then 3 will be substituted.

The character, or string, passed to *ParameterString2* is what the parsing logic uses to separate out the tokens. Parsing the string "The quick brown fox jumps over the lazy dog" with a *space* character as the delimiter will result in nine tokens, as shown on the left, one for each word in the sentence. Parsing the same sentence using an "*o*" as the delimiter, will result in the five strange looking tokens that you see on the right.

Parsing is not limited to a single character. Words, or strings, can also be used. If the above sentence were to be parsed using "*jumps*" as the delimiter, then you would see the tokens as shown to the right here.

The tokens are not changed in any manner after they are parsed. No trimming, changing case, padding, or anything else is done to them.
Parsing is done exactly as the delimiter character is specified. If, for example, you attempt to parse a string like "one two three four five" using a space character you would see the results on the left. What happened? Why is there a blank space between the words? Because there are two spaces, or delimiters, between each word in the string, and not a single space. So, nine tokens were found, not five as the string first appears. How does this happen? After the word one there is a space, which is discarded as the delimiter, leaving one as the first token. The next character is another space, which of course is a delimiter, delimiting a null value. A null value is a token simply by the fact that it is not a delimiter, and in our case, the second token. Next comes the word two and another space delimiter, giving us our third token. This process continues until the last word five. Here we have no trailing delimiter (they are ignored), but five is a token none-the-less, and the last one in our example.

The proper manner to parse a string when dealing with possible multiple spaces, tabs, line feeds, etc., is to use the whitespace delimiter. We consider whitespace to be any character, or series of characters being ASCII 32 or less. Consider this string: "one Sp Tb two Cr Lf Tb three Tb Tb Tb four Sp Sp Sp Sp five". Where Sp = Space, Tb = Tab, Cr = Carriage Return, and Lf = Line Feed.

An attempt to parse it using a single space character would return this:

- Token1: one
- Token2: Tb two Cr Lf Tb three Tb Tb Tb four
- Token3: Null
- Token4: Null
- Token5: five

Not exactly what you want, I would imagine. To remedy this, pass the word whitespace as the delimiter to the ParameterString2 value. Case does not matter:

```
Variable Set String %T1% to "one Sp Tb two Cr Lf Tb three Tb Tb Tb four Sp Sp Sp Sp five"  // String to parse
Variable Set String %T2% to "whitespace"                                            // Whitespace delimiter
Write Registry String: "ParameterString1"                                           // Write Registry
Write Registry String: "ParameterString2"                                           // Write Registry
Macro Run: { String - Parse }                                                   // Parse the string
```

The results:

- Token1: one
- Token2: two
- Token3: three
- Token4: four
- Token5: five
But there is even more to this whitespace feature! You may also specify additional delimiter characters to include with whitespace. For example, say that you wanted to parse a sentence using not only whitespace, but also colons, semi-colons, and commas, simply place them at the end of the word whitespace: whitespace;,. Consider the "one::SpTbtwo::CrLfTbthree;;TbTbTbfour,,SpSpSpSp,,five" string, it has many of these punctuations, in addition to the whitespace characters. Parse it to remove them.

Variable Set String %T1% to "one::SpTbtwo::CrLfTbthree;;TbTbTbfour,,SpSpSpSp,,five" // String to parse
Variable Set String %T2% to "whitespace;;," // Whitespace and more
Write Registry String: "ParameterString1"
Write Registry String: "ParameterString2"
Macro Run: { String - Parse }

The results are the same as before:

Token1: one
Token2: two
Token3: three
Token4: four
Token5: five

What if there were certain characters that you wanted to exclude from the default whitespace set (ASCII 1 through 32) when parsing a string? You can do this! Simply append the exclusion code, X, with the ASCII value of each whitespace character to exclude, 01 through 31 (note that the space character, 32, cannot be excluded) to the end of the word "whitespace" like this: whitespaceX13X10. Consider this string: "oneCrLftwoCrLfthreeTbfourSpfive", which would actual look like this on paper:

one
two
three four five

Now parse it using whitespace, but excluding the Carriage Return and Line Feed characters.

Variable Set String %T1% to "oneCrLftwoCrLfthreeTbfourSpfive" // String to parse
Variable Set String %T2% to "whitespaceX13X10" // Whitespace excluding 13 and 10
Write Registry String: "ParameterString1"
Write Registry String: "ParameterString2"
Macro Run: { String - Parse }

The results are that only three tokens are returned because the CrLf characters were ignored in the parse so they could be retained:

Token1: oneCrLftwoCrLfthree
Token2: four
Token3: five

Both of these whitespace features, inclusion and exclusion, can be combined together to form a single parse. Consider our original string "one::SpTbtwo::CrLfTbthree;;TbTbTbfour,,SpSpSpSp,,five", which contains many punctuations, a carriage return, a line feed, and additional whitespace characters. Parse
it so that only the five words along the carriage return and line feed characters are retained. Note that the exclusion characters must precede the inclusion characters:

```c
Variable Set String %T1% to "one::SpTbTwo::CrLfTbThree;;TbTbTbFour,,SpSpSpSp,,Five"  // String to parse
Variable Set String %T2% to "whitespaceX13X10;;,"  // Exclude CrLf. Include ;;,
Write Registry String: "Parameter String1"  // Write Registry
Write Registry String: "Parameter String2"  // Write Registry
Macro Run: { String - Parse }
```

The results are:

- Token1: one
- Token2: two
- Token3: CrLf
- Token4: three
- Token5: four
- Token6: five

If the string to be parsed contains a trailing delimiter (whatever you specified as `Parameter String2`) it will be truncated and ignored. A leading delimiter is fine. If the string contains one, then the first token returned will be a null token. Also, a leading delimiter is ignored in a reverse (right-to-left) parse routine because, when the string is reversed, it becomes a trailing delimiter.

**Good:**
- Parameter String1 "one,two,three,four,five"
- Parameter String2 ","

**Good:**
- Parameter String1 ",one,two,three,four,five"
- Parameter String2 ","

**Bad:**
- Parameter String1 "one,two,three,four,five,"
- Parameter String2 ","

**Parameters**

- **Parameter String1** - String to parse
- **Parameter String2** - Delimiter character, or string
- **Parameter Integer1** - Token number to parse first:
  - = 0 - Parse starting at the first token found
  - > 0 - Parse starting with the nth token from the left
  - < 0 - Parse starting with the nth token from the right
- **Parameter Integer2** - Number of tokens to parse beginning at the starting token
  - = 0 - Parse all tokens left-to-right starting at the Parameter Integer1 token
  - > 0 - Parse n tokens left-to-right starting at the Parameter Integer1 token
  - < 0 - Parse n tokens right-to-left starting at the Parameter Integer1 token
Here are some different ParameterInteger1 and ParameterInteger2 settings and their affect on returned tokens. ParameterString1 = "abc,def,ghi,jkl" and ParameterString2 = ",":

<table>
<thead>
<tr>
<th>ParameterInteger1</th>
<th>ParameterInteger2</th>
<th>Token1</th>
<th>Token2</th>
<th>Token3</th>
<th>Token4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>abc</td>
<td>def</td>
<td>ghi</td>
<td>jkl</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>abc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>def</td>
<td>ghi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>2</td>
<td>def</td>
<td>ghi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>jkl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-2</td>
<td>jkl</td>
<td>ghi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>jkl</td>
<td>ghi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-4</td>
<td>jkl</td>
<td>ghi</td>
<td>def</td>
<td>abc</td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>ghi</td>
<td>def</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>jlk</td>
</tr>
</tbody>
</table>

{ PGM Function } Parameters

- ParseString or Str - Required
- Delimiter or Dlm - Required
- StartToken or Tok - Set to zero if omitted (all tokens)
- NumTokens or Num - Set to zero if omitted (all tokens)

Returns

- Parsed Tokens
  - Token1 through Token90
- ReturnInteger1
  - Total number of tokens returned (maximum = 90)
- ReturnInteger2
  - Total number of tokens in the input string
- ReturnError?
  - 0 if no error occurs, or 1 if an error occurs

Calls

- { Variables - Save DT }
- { Variables - Restore DT }
- { Program - Log Event }
Example
Parse a string, then write the results to **Token1** through **Token4** in the Registry.

```plaintext
Variable Set String %T1% to "Paul Thornett Pnambic Systems" // Set T1 to a string to parse
Variable Set String %T2% to " " // Set T2 to a space
Variable Set Integer %N1% to 1 // Set N1 to the first token
Variable Set Integer %N2% to 4 // Set N2 as all the tokens
Write Registry String: "ParameterString1" // Write T1 to the Registry
Write Registry String: " ParameterString2 " // Write T2 to the Registry
Write Registry Integer: "ParameterInteger1" // Write N1 to the Registry
Write Registry Integer: "ParameterInteger2" // Write N2 to the Registry
Macro Run: { String - Parse } // Parse the string into tokens
Read Registry Integer: "ReturnInteger1" // Read total tokens., N1 = 4
Read Registry Integer: "ReturnInteger2" // Read parsed tokens, N2 = 4
```

Using **{ PGM Function }**:

```plaintext
Variable Set String %T1% "{ String - Parse }, Paul Thornett Pnambic Systems, ",", 1, 4" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Integer: "ReturnInteger1" // Read total tokens., N1 = 4
Read Registry Integer: "ReturnInteger2" // Read parsed tokens, N2 = 4
```

And using named parameters:

```plaintext
Variable Set String %T1% "{ String - Parse }, ParseString = Paul Thornett Pnambic Systems, Delimiter = ",", StartToken = 1,
NumTokens = 4"
```

Or:

```plaintext
Variable Set String %T1% "{ String - Parse }, Str = Paul Thornett Pnambic Systems, Dlm = ",", Tok = 1, Num = 4"
```
**{ String - Proper }**

Converts a string to proper case where each word begins with a capital letter. This function will capitalize any letter "abcdefghijklmnopqrstuvwxyzßàáâãäåæçèéêëìíîïðñòóôõöøùúûü" if preceded by any whitespace character (ASCII 1 through 32), in addition to changing all others to lower case.

**Parameters**

ParameterString1 - String to convert to proper case

**{ PGM Function } Parameters**

ProperString or Str - Required

**Returns**

ReturnString1 - Input string converted to proper case
ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

**Example**

Converts the "change the color shown (red) to blue, green or yellow. thanks!" string to proper case, resulting in: "Change The Color Shown (Red) To Blue, Green Or Yellow. Thanks!"

Variable Set String %T1% "change the color shown (red) to blue, green or yellow. thanks!" // String to convert
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: { String - Proper } // Convert it
Read Registry String: "ReturnString1" // Read results into T1

Same examples using { PGM Function }:

Notice that we use quotation marks around the string because it contains commas.

Variable Set String %T1% "{ String - Proper }, "change the color shown (red) to blue, green or yellow. thanks!"" // String to convert
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry String: "ReturnString1" // Read results into T1

And using named parameters:

Variable Set String %T1% "{ String - Proper }, ProperString = "change the color shown (red) to blue, green or yellow. thanks!"" Or:

Variable Set String %T1% "{ String - Proper }, Str = "change the color shown (red) to blue, green or yellow. thanks!""
Replicate a string by copying then appending it to itself a specified number of times. Take the string "abc" as an example. Specifying 1 would return the string without changes, "abc". Specifying 2 means that your string will be duplicated once, which will return "abcabc". Also, by specifying a negative number, you can reverse the string. Specifying -1 would return the string "cba". Specifying -2 means that your string will be reversed then duplicated once, which will return "cbacba".

**Parameters**
- **ParameterString1** - String to replicate
- **ParameterInteger1** - Number of times to replicate it
  - > 0 - Replicate the string \(n\) times
  - < 0 - Reverse the string then replicate it \(n\) times
  - = 0 - An error is returned
- **ParameterInteger2** - Where to save the results.
  - 0 = Use \(\text{ReturnString1}\) (default)
  - 1 = Use string variable \(\%\text{T99}\%\)

If the size of the replicated string is going to be fairly large, greater than 16,000 characters, then the function will automatically save it to the \(\%\text{T99}\%\) string variable rather than the \(\text{ReturnString1}\) Registry value, no matter what **ParameterInteger1** is set to because of string length limitations in the Registry.

**Example**
Replicate the string "Abc" 3 times. Results: "Abc Abc Abc".

```
Variable Set String \(\%\text{T1}\%\) "Abc"  // String to replicate
Variable Set Integer \(\%\text{N1}\%\) to 3    // Replicate 3 times
Write Registry String: "ParameterString1"     // Write string
Write Registry Integer: "ParameterInteger1"   // Write count
Macro Run: \{ String - Replicate \}          // Replicate it
Read Registry String: "ReturnString1"         // Read the results, T1 = “Abc Abc Abc”
```
Replicate the string "Abc " 3 times and reverse it. Results: " cbA cbA cbA".

```
Variable Set String %T1% "Abc "       // String to replicate
Variable Set Integer %N1% to -3       // Reverse then replicate 3 times
Write Registry String: "ParameterString1"  // Write string
Write Registry Integer: "ParameterInteger1"   // Write count
Macro Run: { String - Replicate }       // Replicate it
Read Registry String: "ReturnString1"     // Read the results, T1 = “cbA cbA cbA”
```

Replicate the string "Abc " 3 times and save it to %T99%. Results: "Abc Abc Abc ".

```
Variable Set String %T1% "Abc "       // String to replicate
Variable Set Integer %N1% to 3        // Replicate 3 times
Variable Set Integer %N2% to 1        // Save results to T99
Write Registry String: "ParameterString1"  // Write string
Write Registry Integer: "ParameterInteger1"   // Write count
Write Registry Integer: "ParameterInteger2"   // Write destination
Macro Run: { String - Replicate }       // Replicate it
Text Type: %T99%                         // T99 contains the results = “Abc Abc Abc ”
```

Same three examples using `PGM Function`: Notice that we use quotation marks around the string because it contains a space that we want preserved.

```
Variable Set String %T1% "{ String - Replicate }, "Abc ", 3"      // Create PGM Function string
Write Registry String: "PgmFunction"                               // Write it to the Registry
Macro Run: { PGM Function }                                      // Call it
Read Registry String: "ReturnString1"                             // Read the results, T1 = “Abc Abc Abc ”

Variable Set String %T1% "{ String - Replicate }, "Abc ", -3"    // Create PGM Function string
Write Registry String: "PgmFunction"                               // Write it to the Registry
Macro Run: { PGM Function }                                      // Call it
Read Registry String: "ReturnString1"                             // Read the results, T1 = “cbA cbA cbA”

Variable Set String %T1% "{ String - Replicate }, "Abc ", 3, 1"    // Create PGM Function string
Write Registry String: "PgmFunction"                               // Write it to the Registry
Macro Run: { PGM Function }                                      // Call it
Text Type: %T99%                                                 // T99 contains the results = “Abc Abc Abc ”
```

And using named parameters:

```
Variable Set String %T1% "{ String - Replicate }, ReplicateString = "Abc ", ReplicateCount = 3"
Or:
Variable Set String %T1% "{ String - Replicate }, Str = "Abc ", Num = 3"
```

Second example:

```
Variable Set String %T1% "{ String - Replicate }, ReplicateString = "Abc ", ReplicateCount = -3"
Or:
Variable Set String %T1% "{ String - Replicate }, Str = "Abc ", Num = -3"
```
Third example:

```plaintext
Variable Set String %T1% { String - Replicate }, ReplicateString = "Abc", ReplicateCount = -3, AvoidRegistry? = 1

Or:
Variable Set String %T1% { String - Replicate }, Str = "Abc", Num = -3, Reg = 1
```
{ String - Reverse }

Reverses a string. "Abc" becomes "cbA".

Parameters
ParameterString1  - String to reverse

{ PGM Function } Parameters
ReverseString or Rev  - Required

Returns
ReturnString1   - Reversed string
ReturnError?    - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ String - Replicate }

Example
Reverse the string "The quick brown fox jumps over the lazy dog".

Variable Set String %T1% "The quick brown fox jumps over the lazy dog"  // String to replicate
Write Registry String: "ParameterString1"     // Write string
Macro Run: { String - Reverse }          // Reverse it
Read Registry String: "ReturnString1"     // Read the results, T1 = “god yzal eht revo spmuj xof nworb kciuq ehT”

Same example using { PGM Function }:

Variable Set String %T1% "{ String - Reverse }, The quick brown fox jumps over the lazy dog"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: { PGM Function }               // Call it
Read Registry String: "ReturnString1"     // Read the results, T1 = “god yzal eht revo spmuj xof nworb kciuq ehT”

And using named parameters:

Variable Set String %T1% "{ String - Reverse }, ReverseString = The quick brown fox jumps over the lazy dog"

Or:

Variable Set String %T1% "{ String - Reverse }, Str = The quick brown fox jumps over the lazy dog"
{ String - Search }

Returns the location, or starting position, of a string within another string. A zero is returned if the string is not found.

You can search for a specific occurrence of a string, not just the first one. And you can decide whether or not a search is case-sensitive. A search can also be forwards or backwards. If it is backwards then the result (ReturnInteger1) is the position within the string counted from right-to-left. We will use "OneTwoonetwoOneTwo" for the string to search in (ParameterString1) and "one" for the string to search for (ParameterString2) to see how various values can affect your search results.

<table>
<thead>
<tr>
<th>StartPosition</th>
<th>Occurrence</th>
<th>Direction</th>
<th>CaseSensitive?</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>&quot;F&quot;</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>&quot;F&quot;</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>&quot;F&quot;</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&quot;B&quot;</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&quot;F&quot;</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>&quot;F&quot;</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>&quot;B&quot;</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>&quot;B&quot;</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>&quot;B&quot;</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Parameters

 ParameterString1 - String to search in
 ParameterString2 - String to search for
 ParameterInteger1 - Where to begin search in ParameterString1 (string to search)
 ParameterInteger2 - Which occurrence of ParameterString2 (string to search for) to return
 ParameterString3 - Direction of search ("F" = forwards "B" = backwards). Defaults to "F"
 ParameterInteger3 - Case-sensitive search (0 = false 1 = true). Defaults to false

{ PGM Function } Parameters

 SearchIn or In - Required
 SearchFor or For - Required
 StartPosition or Pos - Set to 1 if omitted
 Occurrence or Occ - Set to 1 if omitted
 Direction or Dir - Set to "F" if omitted
 CaseSensitive? Or Case - Set to zero if omitted (false)

Returns

 ReturnInteger1 - Position of string found (zero if not found).
 ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
Example

Find the position of the word "fox" in the string "The quick brown fox".

```plaintext
Variable Set String %T1% to "The quick brown fox"       // String to search
Variable Set String %T2% to "fox"                        // String to find
Write Registry String: "ParameterString1"                 // Write search string
Write Registry String: "ParameterString2"                 // Write find string
Macro Run: { String - Search }                           // Locate position
Read Registry Integer: "ReturnInteger1"                   // Read the results, N1 = 17
```

Find the position of the word "fox" in the string "The quick brown fox" searching backwards.

```plaintext
Variable Set String %T1% to "The quick brown fox"       // String to search
Variable Set String %T2% to "fox"                        // String to find
Variable Set String %T3% to "B"                          // Search backwards
Write Registry String: "ParameterString1"                 // Write search string
Write Registry String: "ParameterString2"                 // Write find string
Write Registry String: "ParameterString3"                 // Write direction string
Macro Run: { String - Search }                           // Locate position
Read Registry Integer: "ReturnInteger1"                   // Read the results, N1 = 3
```

Find the second occurrence of the letter "o" in the string "The quick brown fox".

```plaintext
Variable Set String %T1% to "The quick brown fox"       // String to search
Variable Set String %T2% to "o"                          // String to find
Variable Set Integer %N1% to 2                           // Second science
Write Registry String: "ParameterString1"                 // Write search string
Write Registry String: "ParameterString2"                 // Write find string
Write Registry Integer: "ParameterInteger2"              // Write occurrence
Macro Run: { String - Search }                           // Locate position
Read Registry Integer: "ReturnInteger1"                   // Read the results, N1 = 18
```

Find the first occurrence of the letter "q" in the string "The quick brown fox", searching backwards from the 5th character and using a case-sensitive search.

```plaintext
Variable Set String %T1% to "The quick brown fox"       // String to search
Variable Set String %T2% to "q"                          // String to find
Variable Set Integer %N1% to 5                           // Start at 5th character
Variable Set Integer %N2% to 1                           // First occurrence
Variable Set String %T3% to "B"                          // Search backwards
Variable Set Integer %N3% to 1                           // Case-sensitive
Write Registry String: "ParameterString1"                 // Write search string
Write Registry String: "ParameterString2"                 // Write find string
Write Registry Integer: "ParameterInteger1"              // Write starting position
Write Registry Integer: "ParameterInteger2"              // Write occurrence
Write Registry String: "ParameterInteger3"               // Write direction string
Write Registry Integer: "ParameterInteger3"               // Write Case
Macro Run: { String - Search }                           // Locate the position
Read Registry Integer: "ReturnInteger1"                   // Read the results, N1 = 15
```
Same four examples using *PGM Function*:

**Find the position of the word "fox" in the string "The quick brown fox".**

```plaintext
Variable Set String %T1% "(String - Search ), The quick brown fox, fox"
Write Registry String: "PgmFunction"
Macro Run: { PGM Function }
Read Registry Integer: "ReturnInteger1"
```

**Find the position of the word "fox" in the string "The quick brown fox" searching backwards.**

```plaintext
Variable Set String %T1% "(String - Search ), The quick brown fox, fox, , , B"
Write Registry String: "PgmFunction"
Macro Run: { PGM Function }
Read Registry Integer: "ReturnInteger1"
```

**Find the second occurrence of the letter "o" in the string "The quick brown fox".**

```plaintext
Variable Set String %T1% "(String - Search ), The quick brown fox, o, , 2"
Write Registry String: "PgmFunction"
Macro Run: { PGM Function }
Read Registry Integer: "ReturnInteger1"
```

**Find the first occurrence of the letter "q" in the string "The quick brown fox", searching backwards from the 5th character and using a case-sensitive search.**

```plaintext
Variable Set String %T1% "(String - Search ), The quick brown fox, q, 5, 1, B, 1"
Write Registry String: "PgmFunction"
Macro Run: { PGM Function }
Read Registry Integer: "ReturnInteger1"
```

Now here is what the same examples would look like using *named* parameters:

**Find the position of the word "fox" in the string "The quick brown fox".**

```plaintext
Variable Set String %T1% "(String - Search ), SearchIn = The quick brown fox, SearchFor = fox"
Or:
Variable Set String %T1% "(String - Search ), In = The quick brown fox, For = fox"
```

**Find the position of the word "fox" in the string "The quick brown fox" searching backwards.**

```plaintext
Variable Set String %T1% "(String - Search ), SearchIn = The quick brown fox, SearchFor = fox, Direction = B"
Or:
Variable Set String %T1% "(String - Search ), In = The quick brown fox, For = fox, Dir = B"
```
Find the second occurrence of the letter "o" in the string "The quick brown fox".

Variable Set String %T1% "( String - Search ), SearchIn = The quick brown fox, SearchFor = o, Occurrence = 2"

Or:
Variable Set String %T1% "( String - Search ), In = The quick brown fox, For = o, Occ = 2"

Find the first occurrence of the letter "q" in the string "The quick brown fox", searching backwards from the 5th character and using a case-sensitive search.

Variable Set String %T1% "( String - Search ), SearchIn = The quick brown fox, SearchFor = q, StartPosition = 5, Occurrence = 1, Direction = B, CaseSensitive? = 1"

Or:
Variable Set String %T1% "( String - Search ), In = The quick brown fox, For = q, Pos = 5, Occ = 1, Dir = B, Case = 1"
A man was relaxing with his evening paper, when there was a knock on the door. He opened it, and saw nobody, so he closed the door and went back to his paper. There was another knock, so he opened the door again. This time, he looked down and saw a small snail.

"Mister, could you spare some change?" the snail said. The man picked up the snail, threw him into the bushes, and went back to reading.

A year later, there was another knock at the door. It was the snail.

"What'd you do that for?"
Math and Science Category

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

- John von Neumann (1903-1957)
Overview

Most of what Macro Express provides with the integer and decimal variables is basic four-function calculator stuff. We attempt to enhance this with the Math category functions.

- Convert decimals and integers to hexadecimal values and back again.
- Calculate square roots.
- Remainder calculations (modulo).
- Determine the sign of a value.
- Return absolute values.
- Convert degrees to radians and radians to degrees.
- Calculate floor and ceiling values.
Registry

When the PGM Functions Library™ was first installed on your computer, an area called Parameters was created in your Registry within the Professional Grade Macros key. This area, or key, contains many values, which are used to communicate with the Math and Science functions via the Write Registry String, Write Registry Integer, Write Registry Decimal, Read Registry String, Read Registry Integer, and Read Registry Decimal commands. The Name column, in the right hand pane, is the value name. The Type column is the type of variable being stored, REG_SZ for strings, REG_DWORD for integers, and REG_BINARY for decimals. And of course the Data column shows the actual value being stored.

Let’s examine in detail these Registry values and how you use them with the String functions. They can be grouped into two categories:

1. Input values - You set these.
2. Return values - Set by the functions.

Input Values

These are values that you write to the Registry prior to calling any of the Math and Science functions. The value depends on which function you are calling. For example, if you wanted to find the square root of a number, you would write the number to the ParameterDecimal1 value and read the results from the ReturnDecimal1 value.

Unlike the Program Operations functions, where each value had a distinct purpose (ProgramName, WindowTitle, etc.), these values are generic placeholders, meaning that they can be, and are, used by many functions to perform many different actions. There is never any guarantee that a particular value
will always be used for a particular action or purpose. It really does depend entirely on whichever function you are referencing at the time.

**ParameterDecimal1**

Usually this value contains whatever decimal value you are going to do something to. In other words, the object or target decimal value. For example, you could convert a decimal value to its absolute value using the `{Math - Abs}` function.

```plaintext
Variable Set Decimal %D1% to -123.456 // Set D1 to a negative value
Write Registry Decimal: "ParameterDecimal1" // Write D1 to the Registry
Macro Run: { Math - Abs } // Convert it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D2 = 123.456
```

**ParameterDecimal2**

Holds a second decimal value depending on the function that you are calling. An example would be the `{Math - Mod}` function, which requires two decimal values, a dividend and a divisor.

```plaintext
Variable Set Decimal %D1% to 20 // Dividend
Variable Set Decimal %D2% to 6 // Divisor
Write Registry Decimal: "ParameterDecimal1" // Write dividend
Write Registry Decimal: "ParameterDecimal2" // Write divisor
Macro Run: { Math - Mod } // Get the remainder
Read Registry Decimal: "ReturnDecimal1" // Read the results, D3 = 2
```

**ParameterInteger1**

Contains whatever integer value you are going to do something to. For example, you could convert an integer value to a hexadecimal string using the `{Math - Integer to Hex}` function.

```plaintext
Variable Set Integer %N1% to 61680 // Integer
Write Registry Integer: "ParameterInteger1" // Write it
Macro Run: { Math - Integer to Hex } // Convert it
Read Registry String: "ReturnString1" // Read the results, T1 = F0F0
```

**ParameterString1**

Contains whatever string value you are going to do perform an operation on. For example, you could convert a hexadecimal string to an integer value using the `{Math - Hex to Integer}` function.

```plaintext
Variable Set String %T1% "4F77" // Hexadecimal
Write Registry String: "ParameterString1" // Write it
Macro Run: { Math - Hex to Integer } // Convert it
Read Registry Integer: "ReturnInteger1" // Read the results, N1 = 20343
```
Return Values

The resulting calculations performed by the Math and Science functions are placed in these values. You would read one of them, depending on which function you have called, for your answer. For example, if you wanted to retrieve the value of Pi, you would simply call the \{Math - Pi\} function and read the results in the ReturnDecimal1 value.

Return Decimal1
Contains the results of any function which returns a decimal value. The \{Math - Degrees to Radians\} function is a prime example.

Variable Set Decimal %D1% to 180  // Degrees to convert
Write Registry Decimal: "ParameterDecimal1"  // Write it
Macro Run: \{ Math - Degrees to Radians \}  // Convert it
Read Registry Decimal: "ReturnDecimal1"  // Read the results, D2 = 3.14159265358979

Return Integer1
Contains the results of any function that returns an integer value such as the \{Math - Hex to Integer\} function.

Variable Set String %T1% "4F77"  // Hexadecimal
Write Registry String: "ParameterString1"  // Write it
Macro Run: \{ Math - Hex to Integer \}  // Convert it
Read Registry Integer: "ReturnInteger1"  // Read the results, N1 = 20343

Return String1
Contains the results of any function that returns a string such as the \{Math - Integer to Hex\} function.

Variable Set Integer %N1% to 61680  // Integer
Write Registry Integer: "ParameterInteger1"  // Write it
Macro Run: \{ Math - Integer to Hex \}  // Convert it
Read Registry String: "ReturnString1"  // Read the results, T1 = F0F0

Return Error?
This value will tell you if an error has occurred in one of the Math and Science functions. If the value is 1 then an error occurred, and a 0 means that the function ran fine. This value should be tested after a function returns from processing so appropriate action can be taken by your macro. If an error occurs and, if event logging is enabled (Log Event?), then the actual error will be contained in the event log.

Macro Run: \{ Math - Exp \}  // \(e^x\)
Read Registry Integer: "ReturnError?"  // Check for an error
If Variable %N1% = 1
   Text Box Display: Error!  // If so, then prompt the user
End If
Functions

Most of what Macro Express provides with the integer and decimal variables is basic four-function calculator stuff. We enhance this with ten Math and Science category functions.

This section describes in detail how to use the Math and Science category functions. You will quickly learn that these descriptions are not math lessons, instead, they assume that you know what you are doing and give you information needed to use the functions only.

\begin{itemize}
    \item \{ Math - Abs \} \hspace{1cm} \text{Returns the absolute value of a number}
    \item \{ Math - Sign \} \hspace{1cm} \text{Returns a value based on a number having a sign, no sign or is zero}
    \item \{ Math - Pi \} \hspace{1cm} \text{Returns the value of Pi to 14 decimal places}
    \item \{ Math - Mod \} \hspace{1cm} \text{Returns the remainder of dividing X by Y}
    \item \{ Math - Exp \} \hspace{1cm} \text{Returns the value of } e^x \text{ (e raised to the power of x)}
    \item \{ Math - Square Root \} \hspace{1cm} \text{Returns the square root of a number}
    \item \{ Math - Ceiling \} \hspace{1cm} \text{Returns the nearest whole number that is } \geq \text{ the number passed}
    \item \{ Math - Floor \} \hspace{1cm} \text{Returns the nearest whole number that is } \leq \text{ the passed number}
    \item \{ Math - Degrees to Radians \} \hspace{1cm} \text{Converts degrees to radians}
    \item \{ Math - Radians to Degrees \} \hspace{1cm} \text{Converts radians to degrees}
    \item \{ Math - Decimal to Hex \} \hspace{1cm} \text{Converts a large number to a hexadecimal string}
    \item \{ Math - Hex to Decimal \} \hspace{1cm} \text{Converts a large hexadecimal strings to number}
    \item \{ Math - Integer to Hex \} \hspace{1cm} \text{Converts an integer to a hexadecimal string}
    \item \{ Math - Hex to Integer \} \hspace{1cm} \text{Converts a hexadecimal string to an integer}
\end{itemize}

Except for the \{ Math - Pi \} function, which requires no parameters, they can all be called by \{ PGM Function \}. As with many of our functions, if an error occurs you’ll be notified via the ReturnError? value and, if event logging is enabled, the actual error will be recorded in the log file. For the sake of clarity, the examples here do not check for an error, however, it is recommended that you do check, so appropriate action can be taken.
{ Math - Abs }

Returns the absolute value of a decimal number, which is simply a matter of converting negative values to positives.

**Parameters**
- ParameterDecimal1  - Decimal value

**Parameters**
- Number or Num   - Required

**Returns**
- ReturnDecimal1   - Absolute value

**Calls**
- { Variables - Save DT }
- { Variables - Restore DT }
- { Program - Log Event }

**Example**
Get the absolute value of \(-123.456\)

```plaintext
Variable Set Decimal %D1% to -123.456      // Negative value
Write Registry Decimal: "ParameterDecimal1"    // Write it
Macro Run: { Math - Abs }        // Convert
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = 123.456
```

Using `{ PGM Function }`:

```plaintext
Variable Set String %T1% "{ Math - Abs }, -123.456"    // Create PGM Function string
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }        // Call it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = 123.456
```

Using named parameters:

```plaintext
Variable Set String %T1% "{ Math - Abs }, Number = -123.456"
Or:
Variable Set String %T1% "{ Math - Abs }, Num = -123.456"
```
\{ Math - Sign \}

Returns a value based on whether or not a decimal number has a sign (is negative) or no sign (is positive) or, as a bonus, is simply a zero.

**Parameters**

Parameter

Decimal1 - Decimal value

**{ PGM Function } Parameters**

Number or Num - Required

**Returns**

ReturnDecimal1 - 0 if a zero value, 1 if a positive value, or a -1 if a negative value

**Calls**

\{ Variables - Save DT \}

\{ Variables - Restore DT \}

\{ Program - Log Event \}

**Example**

Determine if the decimal number -123.456 contains a sign.

Variable Set Decimal %D1% to -123.456      // Negative value
Write Registry Decimal: "ParameterDecimal1"    // Write it
Macro Run: \{ Math - Sign \}        // Determine
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = -1

Using { PGM Function }:

Variable Set String %T1% "{ Math - Sign }, -1"     // Create PGM Function string
Write Registry String: "PgmFunction"      // Write it to the Registry
Macro Run: \{ PGM Function \}        // Call it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = 123.456

Using named parameters:

Variable Set String %T1% "{ Math - Sign }, Number = -123.456"

Or:

Variable Set String %T1% "{ Math - Sign }, Num = -123.456"
{ Math - Pi }

Returns the value of $\pi$ to 14 decimal places.

**Parameters**
None

**Returns**
ReturnDecimal1 $\pi$ (3.14159265358979)

**Calls**
Nothing.

**Example**
Get the value of $\pi$.

Macro Run: { Math - Pi }  // Get Pi
Read Registry Decimal: "ReturnDecimal1"  // Read the results, D1 = 3.14159265358979
{ Math - Mod }

Returns the remainder when dividing X (dividend) by Y (divisor) e.g. \( x/y \). A negative number is returned if the dividend is negative. The divisor cannot be zero. The remainder of 20 divided by 6 is 2.

**Parameters**

Parameter Decimal 1 - Dividend (\( x \))

Parameter Decimal 2 - Divisor (\( y \))

**{ PGM Function } Parameters**

Dividend or Div - Required

Divisor or Dvs - Required

**Returns**

Return Decimal 1 - Remainder

Return Error? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }

{ Variables - Restore DT }

{ Program - Log Event }

**Example**

Return the remainder of dividing 20 by 6 ... 20 being the dividend (\( x \)) and 6 being the divisor (\( y \)).

```
Variable Set Decimal %D1% to 20        // Dividend
Variable Set Decimal %D2% to 6        // Divisor
Write Registry Decimal: "ParameterDecimal1" // Write dividend
Write Registry Decimal: "ParameterDecimal2" // Write divisor
Macro Run: { Math - Mod }        // Get the remainder
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D3 = 2
```

Using \( \text{PGM Function} \):

```
Variable Set String %T1% "( Math - Mod ), 20, 6" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = 2
```

Using named parameters:

```
Variable Set String %T1% "( Math - Mod ), Dividend = 20, Divisor = 6"
```

Or:

```
Variable Set String %T1% "( Math - Mod ), Div = 20, Dvs = 6"
```
Returns the value of $e^x$ ($e$ raised to the power of $x$) where $x$ is the passed parameter within the range of -744.44 to +709.782. The value of $e$, which is the base of natural logarithms, is 2.71828182845905.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterDecimal1</td>
<td>Powers of $e$</td>
</tr>
</tbody>
</table>

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power or Pwr</td>
<td>- Required</td>
</tr>
</tbody>
</table>

**Returns**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReturnDecimal1</td>
<td>Value of $e^x$</td>
</tr>
<tr>
<td>ReturnError?</td>
<td>0 if no error occurs, or 1 if an error occurs</td>
</tr>
</tbody>
</table>

**Calls**

- { Variables - Save DT }
- { Variables - Restore DT }
- { Program - Log Event }

**Example**

Return the value of $e^3$.

```
Variable Set Decimal %D1% to 3 // Power of x
Write Registry Decimal: "ParameterDecimal1" // Write it
Macro Run: { Math - Exp } // Calculate it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D2 = 20.085369231877
```

Using **( PGM Function )**:

```
Variable Set String %T1% "{ Math - Exp }, 3" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D1 = 20.085369231877
```

Using named parameters:

```
Variable Set String %T1% "{ Math - Exp }, Power = 3"
```

Or:

```
Variable Set String %T1% "{ Math - Exp }, Pwr = 3"
```
{ Math - Square Root }

Returns the square root of a decimal number. Zero and negative values are not allowed and neither is any value greater than or equal to 1,000,000,000,000,000.

Parameters
ParameterDecimal1 - Decimal value

{ PGM Function } Parameters
Number or Num - Required

Returns
ReturnDecimal1 - Square root
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Get the square root of 144. What could be simpler, eh?

Variable Set Decimal %D1% to 144       // Decimal value
Write Registry Decimal: "ParameterDecimal1"    // Write it
Macro Run: { Math - Square Root }       // Calculate it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D2 = 12

Using { PGM Function }:

Variable Set String %T1% "{ Math - Square Root }, 144"  // Create PGM Function string
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }       // Call it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = 12

Using named parameters:

Variable Set String %T1% "{ Math - Square Root }, Number = 144"

Or:

Variable Set String %T1% "{ Math - Square Root }, Num = 144"
{ Math - Ceiling }

Returns the nearest whole number that is greater than or equal to the passed decimal value. The next whole number greater than 123.456 is 124, from -123.456 is -123, etc.

Parameters
ParameterDecimal1 - Decimal value

{ PGM Function } Parameters
Number or Num - Required

Returns
ReturnDecimal1 - Ceiling
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Get the ceiling value of 123.456.

Variable Set Decimal %D1% to 123.456 // Decimal value
Write Registry Decimal: "ParameterDecimal1" // Write it
Macro Run: { Math - Ceiling } // Calculate it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D2 = 124

Using { PGM Function }:

Variable Set String %T1% "( Math - Ceiling ), 123.456" // Create PGM Function string
Write Registry String: "PgmFunction" // Write it to the Registry
Macro Run: { PGM Function } // Call it
Read Registry Decimal: "ReturnDecimal1" // Read the results, D1 = 124

Using named parameters:

Variable Set String %T1% "( Math - Ceiling ), Number = 123.456"

Or:

Variable Set String %T1% "( Math - Ceiling ), Num = 123.456"
{ Math - Floor }

Returns the nearest whole number that is less than or equal to the passed decimal value. The next whole number less than 123.456 is 123, from -123.456 is -124, etc.

Parameters
ParameterDecimal1 - Decimal value

{ PGM Function } Parameters
Number or Num - Required

Returns
ReturnDecimal1 - Floor
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Get the floor value of 123.456.

Variable Set Decimal %D1% to 123.456       // Decimal value
Write Registry Decimal: "ParameterDecimal1"    // Write it
Macro Run: { Math - Floor }        // Calculate it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D2 = 123

Using { PGM Function }:

Variable Set String %T1% "{ Math - Floor }, 123.456"       // Create PGM Function string
Write Registry String: "PgmFunction"    // Write it to the Registry
Macro Run: { PGM Function }        // Call it
Read Registry Decimal: "ReturnDecimal1"     // Read the results, D1 = 123

Using named parameters:

Variable Set String %T1% "{ Math - Floor }, Number = 123.456"
Or:
Variable Set String %T1% "{ Math - Floor }, Num = 123.456"
{ Math - Degrees to Radians }

Converts degrees to radians. Radians are a unit of angular measure in which the angle of an entire circle is $2\pi$ radians. There are therefore 360° per $2\pi$ radians, equal to $180°/\pi$ or 57.2957795130823° per radian. A right angle is $\pi/2$ radians.

**Parameters**

ParameterDecimal1 - Degrees value

**{ PGM Function } Parameters**

Degree or Deg - Required

**Returns**

ReturnDecimal1 - Radians value

ReturnError? - 0 if no error occurs, or 1 if an error occurs

**Calls**

{ Variables - Save DT }

{ Variables - Restore DT }

{ Program - Log Event }

**Example**

Calculate the radians in $180°$.

```
Variable Set Decimal %D1% to 180        // Degrees
Write Registry Decimal: "ParameterDecimal1"     // Write it
Macro Run: { Math - Degrees to Radians }      // Calculate it
Read Registry Decimal: "ReturnDecimal1"      // Read the results, D2 = 3.14159265358979
```

Using { PGM Function }:

```
Variable Set String %T1% "{ Math - Degrees to Radians }, 180" // Create PGM Function string
Write Registry String: "PgmFunction"                        // Write it to the Registry
Macro Run: { PGM Function }                                // Call it
Read Registry Decimal: "ReturnDecimal1"                    // Read the results, D2 = 3.14159265358979
```

Using named parameters:

```
Variable Set String %T1% "{ Math - Degrees to Radians }, Degree = 180"
Or:
Variable Set String %T1% "{ Math - Degrees to Radians }, Deg = 180"
```
{ Math - Radians to Degrees }

Radians are a unit of angular measure in which the angle of an entire circle is $2\pi$ radians. There are therefore 6.28318530717959 radians in a complete 360° circle.

Parameters
ParameterDecimal1 - Radians value

{ PGM Function } Parameters
Radian or Rad - Required

Returns
ReturnDecimal1 - Degrees value
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Calculate the degrees in 3 radians.

```
Variable Set Decimal %D1% to 3         // Radians
Write Registry Decimal: "ParameterDecimal1"     // Write it
Macro Run: { Math - Radians to Degrees }      // Calculate it
Read Registry Decimal: "ReturnDecimal1"      // Read the results, D2 = 171.887338539247
```

Using { PGM Function }:

```
Variable Set String %T1% "{ Math - Radians to Degrees }, 3"   // Create PGM Function string
Write Registry String: "PgmFunction"       // Write it to the Registry
Macro Run: { PGM Function }               // Call it
Read Registry Decimal: "ReturnDecimal1"    // Read the results, D2 = 171.887338539247
```

Using named parameters:

```
Variable Set String %T1% "{ Math - Radians to Degrees }, Radian = 3"
```

Or:

```
Variable Set String %T1% "{ Math - Radians to Degrees }, Rad = 3"
```
{ Math - Decimal to Hex }

Converts a decimal number to a hexadecimal string. The value to be converted must be a whole number greater than, or equal to zero, which means that negative numbers, or numbers containing decimals, will return an error.

Parameters
ParameterDecimal1 - Decimal value

{ PGM Function } Parameters
Decimal or Dec - Required

Returns
ReturnString1 - Hexadecimal string
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Convert 62,178 to hexadecimal.

```plaintext
Variable Set Decimal %D1% to 62178         // Decimal
Write Registry Decimal: "ParameterDecimal1"      // Write it
Macro Run: { Math - Decimal to Hex }        // Convert it
Read Registry String: "ReturnString1"        // Read the results, T1 = F2E2
```

Using { PGM Function }:

```plaintext
Variable Set String %T1% "{ Math - Decimal to Hex }, 62178"                   // Create PGM Function string
Write Registry String: "PgmFunction"                       // Write it to the Registry
Macro Run: { PGM Function }                        // Call it
Read Registry String: "ReturnString1"                   // Read the results, T2 = F2E2
```

Using named parameters:
```
Variable Set String %T1% "{ Math - Decimal to Hex }, Decimal = 62178"
```
Or:
```
Variable Set String %T1% "{ Math - Decimal to Hex }, Dec = 62178"
```
{ Math - Hex to Decimal }

Converts a hexadecimal string to a decimal number.

Parameters
ParameterString1 - Hexadecimal string

{ PGM Function } Parameters
Hexadecimal or Hex - Required

Returns
ReturnDecimal1 - Decimal value
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example
Convert E2B5C6 to a decimal.

```
Variable Set String %T1% "E2B5C6"             // Hexadecimal
Write Registry String: "ParameterString1"    // Write it
Macro Run: { Math - Hex to Decimal }        // Convert it
Read Registry Decimal: "ReturnDecimal1"      // Read the results, D1 = 14857670
```

Using { PGM Function }:

```
Variable Set String %T1% "{ Math - Hex to Decimal }, E2B5C6"      // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: { PGM Function }                  // Call it
Read Registry Decimal: "ReturnDecimal1"       // Read the results, D1 = 14857670
```

Using named parameters:

```
Variable Set String %T1% "{ Math - Hex to Decimal }, Hexadecimal = E2B5C6"
```

Or:

```
Variable Set String %T1% "{ Math - Hex to Decimal }, Hex = E2B5C6"
```
{ Math - Integer to Hex }

Converts an integer number to a hexadecimal string. Input value must be less than 2,147,483,648.

Parameters
Parameter Integer1 - Integer value

{ PGM Function } Parameters
Integer or Int - Required

Returns
ReturnString1 - Hexadecimal string
ReturnError? - 0 if no error occurs, or 1 if an error occurs

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }
{ Math - Decimal to Hex }

Example
Convert 61,680 to hexadecimal.

```
Variable Set Integer %N1% to 61680         // Integer
Write Registry Integer: "ParameterInteger1"       // Write it
Macro Run: { Math - Integer to Hex }        // Convert it
Read Registry String: "ReturnString1"        // Read the results, T1 = F0F0
```

Using { PGM Function }:

```
Variable Set String %T1% "{ Math - Integer to Hex }, 61680"  // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: { PGM Function }                 // Call it
Read Registry String: "ReturnString1"        // Read the results, T2 = F0F0
```

Using named parameters:

```
Variable Set String %T1% "{ Math - Integer to Hex }, Integer = 61680"
```

Or:

```
Variable Set String %T1% "{ Math - Integer to Hex }, Dec = 61680"
```
\textbf{\{ Math - Hex to Integer \}}

Converts a hexadecimal string to an integer number.

\textbf{Parameters}

ParameterString1 - Hexadecimal string

\textbf{\{ PGM Function \} Parameters}

Hexadecimal or Hex - Required

\textbf{Returns}

ReturnInteger1 - Integer value

ReturnError? - 0 if no error occurs, or 1 if an error occurs

\textbf{Calls}

\{ Variables - Save DT \}
\{ Variables - Restore DT \}
\{ Program - Log Event \}
\{ Math - Hex to Decimal \}

\textbf{Example}

Convert \textbf{4F77} to an integer.

\begin{verbatim}
Variable Set String %T1% "4F77"          // Hexadecimal
Write Registry String: "ParameterString1"       // Write it
Macro Run: \{ Math - Hex to Integer \}        // Convert it
Read Registry Integer: "ReturnInteger1"       // Read the results, N1 = 20343
\end{verbatim}

Using \textit{\{ PGM Function \}}:

\begin{verbatim}
Variable Set String %T1% "{ Math - Hex to Integer }, 4F77"   // Create PGM Function string
Write Registry String: "PgmFunction"          // Write it to the Registry
Macro Run: \{ PGM Function \}               // Call it
Read Registry Integer: "ReturnInteger1"      // Read the results, N1 = 20343
\end{verbatim}

Using \textit{named} parameters:

\begin{verbatim}
Variable Set String %T1% "{ Math - Hex to Integer }, Hexadecimal = 4F77"
\end{verbatim}

Or:

\begin{verbatim}
Variable Set String %T1% "{ Math - Hex to Integer }, Hex = 4F77"
\end{verbatim}
The great thing about a computer notebook is that no matter how much you stuff into it, it doesn’t get bigger or heavier.

- Bill Gates, *Business @ The Speed of Thought*
Overview

The File Operations category includes some functions not available with the standard Macro Express file commands. With them you can:

- Validate file and folder names that the user has input.
- Generate unique file names to be used for your temporary file writing needs.
- Parse path name strings into separate drive, path, file name, and file extension strings.
- Compare files and folders to determine if they are the same or are different.
- Make sure that paths end, or don’t, end with a trailing backslash character.
Registry

When the PGM Functions Library™ was first installed on your computer, an area called Parameters was created in your Registry within the Professional Grade Macros key. This area, or key, contains many values, which are used to communicate with the File Operations functions via the **Write Registry String**, **Write Registry Integer**, **Read Registry String**, and **Read Registry Integer** commands. The Name column, in the right hand pane, is the value name. The Type column is the type of variable being stored, REG_SZ for strings and REG_DWORD for integers. And of course the Data column shows the actual value being stored.

![Registry Editor](image)

Let’s examine in detail these Registry values and how you use them with the String functions. They can be grouped into two categories:

1. Input values - You set these.
2. Return values - Set by the functions.
Input Values

These are values that you write to the Registry prior to calling any of the File Operations functions. The value depends on which function you are calling. For example, if you wanted to make sure that a path ended with a trailing backslash character, you would write the path string to the `ParameterString1` value, then read the results from the `ReturnString1` value.

The values here are generic placeholders, meaning that they can be, and are, used by many functions to perform many different actions. There is never any guarantee that a particular value will always be used for a particular action or purpose. It really does depend entirely on whichever function you are referencing at the time.

**ParameterString1**

Usually this value contains whatever string you are going to do something to. In other words, the object or target string. You could validate, for example, a file name using the `{ File - Validate File Name }` function.

```
Variable Set String %T1% to "MyFileName.txt"     // File name to validate
Write Registry String: "ParameterString1"       // Write it to the Registry
Macro Run: { File - Validate File Name }        // Validate it
Read Registry Integer: "ReturnError?"           // Okay?
```

**ParameterString2**

Holds a second string value depending on the function that you are calling. An example would be the `{ File - Compare }` function, which requires two string values, a source file and a target file.

```
Variable Set String %T1% to "x:\Folder1\File3.txt"   // Set T1 to a source file
Variable Set String %T2% to "x:\Folder2\File3.txt"   // Set T2 to a target file
Write Registry String: "ParameterString1"           // Write T1 to the Registry
Write Registry String: "ParameterString2"           // Write T2 to the Registry
Macro Run: { File - Compare }                      // Compare them
Read Registry String: "ReturnString1"               // Read the results, T1 = “1”
```

**ParameterString3**

Holds a third string value depending on the function that you are calling. An example would be the `{ File - Compare }` function, which uses a string for an output destination.

```
Variable Set String %T1% to "x:\Folder1\File3.txt"   // Set T1 to a source file
Variable Set String %T2% to "x:\Folder2\File3.txt"   // Set T2 to a target file
Variable Set String %T3% to "S"                      // Set output to the screen
Write Registry String: "ParameterString1"           // Write T1 to the Registry
Write Registry String: "ParameterString2"           // Write T2 to the Registry
Write Registry String: "ParameterString3"           // Write T3 to the Registry
Macro Run: { File - Compare }                      // Compare them
ParameterString4
Holds a fourth string value. Again the \{ File - Compare \} function is an example where a string can be set to display certain information in the output.

```plaintext
Variable Set String %T1% to "x:\Folder1"  // Set T1 to a source file
Variable Set String %T2% to "x:\Folder2"  // Set T2 to a target file
Variable Set String %T3% to "S"            // Set output to the screen
Variable Set String %T4% to "I"            // Show only identical files
Write Registry String: "ParameterString1"  // Write T1 to the Registry
Write Registry String: "ParameterString2"  // Write T2 to the Registry
Write Registry String: "ParameterString3"  // Write T3 to the Registry
Write Registry String: "ParameterString4"  // Write T3 to the Registry
Macro Run: { File - Compare }
// Compare them
```

ParameterInteger1
Holds an integer value for size, position, count, etc., depending on the function that you are calling. For example, the \{ File - Parse Path Name \} function can be set to return a long, Window's style path.

```plaintext
Variable Set String %T1% to "c:\Program Files\Macro Express3\MacEdit.exe"  // Path name to parse
Variable Set Integer %N1% to 1                                           // Long (Windows) name
Write Registry String: "ParameterString1"                                // Write it to the Registry
Write Registry Integer: "ParameterInteger1"                              // Write it to the Registry
Macro Run: { File - Parse Path Name }
// Parse it
Read Registry String: "ReturnString1"                                   // T1 = c:
Read Registry String: "ReturnString2"                                   // T2 = \Program Files\Macro Express\MacEdit
Read Registry String: "ReturnString3"                                   // T3 = MacEdit
Read Registry String: "ReturnString4"                                   // T4 = .exe
```
Return Values

The resulting string calculations performed by the File Operations functions are placed in these values. You would read one or more of them, depending on which function you have called, for your answer(s).

ReturnString1
ReturnString2
ReturnString3
ReturnString4
Contains the results of any function which returns string values. In particular, the \{ File - Parse Path Name \} function returns four different strings.

Variable Set String %T1% to "c:\Program Files\Macro Express3\MacEdit.exe" // Path name to parse
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: { File - Parse Path Name } // Parse it
Read Registry String: "ReturnString1" // T1 = c:
Read Registry String: "ReturnString2" // T2 = \Program Files\Macro Express\nRead Registry String: "ReturnString3" // T3 = MacEdit
Read Registry String: "ReturnString4" // T4 = .exe

ReturnError?
This value will tell you if an error has occurred in one of the File Operation functions. If the value is 1 then an error occurred, and a 0 means that the function ran fine. This value should be tested after a function returns from processing so appropriate action can be taken by your macro. If an error occurs and, if event logging is enabled (LogEvent?), then the actual error will be contained in the event log.

Macro Run: { File - Compare } // Compare files
Read Registry Integer: "ReturnError?" // Check for an error
If Variable %N1% = 1
Text Box Display: Error! // If so, then prompt the user
End If
Functions

Finding ourselves needing to constantly validate user input of file and path names, create unique but temporary files, and comparing files, we created the File Operations category consisting of seven functions not available with the standard Macro Express file commands.

- **File - Compare**  Compare files, folders, and sub-folders in five different ways
- **File - Parse Path Name**  Separates a path name into its different components
- **File - Unique File Name**  Returns a unique full path and file name
- **File - Backslash Append**  Adds a trailing backslash to a path name
- **File - Backslash Remove**  Removes any trailing backslashes from a path name
- **File - Validate File Name**  Examines a file name for illegal characters
- **File - Validate Path Name**  Examines a path name for illegal characters

All but the **File - Unique File Name** can use the **PGM Function**. Only the **File - Compare** function actually returns an error. There are no errors to be returned by the other functions. The ReturnError? value, however, is used by both the **File - Validate File Name** and **File - Validate Path Name** functions to answer whether or not the passed names are valid or invalid.
{ File - Backslash Append }

Adds a trailing backslash to a path name (if needed). \texttt{c:\PathName} becomes \texttt{c:\PathName\}.

**Parameters**

ParameterString1 - Path for which to add a backslash

**{ PGM Function } Parameters**

PathName or Path - Required

**Returns**

ReturnString1 - Modified path

**Calls**

{ Variables - Save 25 }

{ Variables - Restore 25 }

{ Program - Log Event }

**Example**

Add a backslash to \texttt{c:\PathName}.

Variable Set String \texttt{%T1%} "c:\PathName" \hfill // Path to modified
Write Registry String: "ParameterString1" \hfill // Write string
Macro Run: { File - Backslash Append } \hfill // Add it
Read Registry String: "ReturnString1" \hfill // Read the results, \texttt{T1 = "c:\PathName\"}

Same example using \{ PGM Function \}:

Variable Set String \texttt{%T1%} "{ File - Backslash Append }, c:\PathName" \hfill // Create PGM Function string
Write Registry String: "PgmFunction" \hfill // Write it to the Registry
Macro Run: { PGM Function } \hfill // Call it
Read Registry String: "ReturnString1" \hfill // Read the results, \texttt{T1 = "c:\PathName\"}

And using named parameters:

Variable Set String \texttt{%T1%} "{ File - Backslash Append }, PathName = c:\PathName"

Or:

Variable Set String \texttt{%T1%} "{ File - Backslash Append }, Path = c:\PathName"
{ File - Backslash Remove }

Removes any trailing backslashes from a path name. c:\PathName\ becomes c:\PathName.

**Parameters**

ParameterString1 - Path from which to remove backslash

**{ PGM Function } Parameters**

PathName or Path - Required

**Returns**

ReturnString1 - Modified path

**Calls**

{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Program - Log Event }

**Example**

Remove the backslash from c:\PathName\.

```
Variable Set String %T1% "c:\PathName\"       // Path to modified
Write Registry String: "ParameterString1"     // Write string
Macro Run: { File - Backslash Remove }         // Remove it
Read Registry String: "ReturnString1"         // Read the results, T1 = “c:\PathName”
```

Same example using { PGM Function }:

```
Variable Set String %T1% "( File - Backslash Remove ), c:\PathName\"          // Create PGM Function string
Write Registry String: "PgmFunction"                                           // Write it to the Registry
Macro Run: { PGM Function }                                                   // Call it
Read Registry String: "ReturnString1"                                        // Read the results, T1 = “c:\PathName”
```

And using named parameters:

```
Variable Set String %T1% "( File - Backslash Remove ), PathName = c:\PathName\"
```

Or:

```
Variable Set String %T1% "( File - Backslash Remove ), Path = c:\PathName\"
```
This function will not work on Win'98 or Win'95.

- Compares two files to determine if their contents are identical or different. This is a one-to-one comparison.
- Compares two folders, and optionally their sub-folders, to determine which files exist in one, but not the other. This is a many-to-many comparison.
- Compares two folders, and optionally, their sub-folders, to determine which files exist in both, and then compares each to determine if their contents are identical or different. This too is a many-to-many comparison.
- Compare a single file to all files in a folder, and optionally, sub-folders to determine which ones are identical and which are not. This is a one-to-many comparison.
- Compare files within a folder, and optionally, sub-folders to determine which ones are identical or different to a single, target file. This is a many-to-one comparison.

File comparison is done by reading the contents of each file into two separate text variables, and then comparing them on a character by character basis. This is very quick, even when dealing with files as large as several megabytes. The comparison results are output to one of four possible destinations:

- To the \texttt{ReturnString1} value in the Registry (default).
- To a specified text file.
- To the \texttt{%T90%} Macro Express string variables.
- To your screen.

**Parameters**

- \texttt{ParameterString1} - Source folder or file name
- \texttt{ParameterString2} - Target folder or file name
- \texttt{ParameterInteger1} - Include sub-folders in comparisons (0=false 1=true). Defaults to false
- \texttt{ParameterString3} - Output destination of results
  - R or \texttt{Empty} = Use \texttt{ReturnString1} (default)
  - F<Filename> = Write output to file name (no space between F and file name)
  - S = Screen display
  - V = Use string variable \texttt{%T90%}

- \texttt{ParameterString4} - Information to include in output (any combination of choices)
  - D = Only list files that are different
  - I = Only list files that are identical
  - S = Only list files within the source folder
  - T = Only list files within the target folder
  - Empty = Use all of the above (\texttt{DIST})

<table>
<thead>
<tr>
<th>ParameterString4</th>
<th>D</th>
<th>I</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{R - ReturnString1}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>\texttt{F - &lt;Filename&gt;}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>\texttt{S - Screen display}</td>
<td>&quot;Different&quot;</td>
<td>&quot;Identical&quot;</td>
<td>&quot;Source only&quot;</td>
<td>&quot;Target only&quot;</td>
</tr>
<tr>
<td>\texttt{V - String %T90%}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
The `ParameterString3` value (OutputTo) returns two different result formats, one for the screen display (S) option and the other for the rest of the options. The screen option displays "words" and the others a "numeric code" based on the options chosen for the `ParameterString4` value (ShowType). In other words, all `ParameterString3` options, except for the screen display option, will have a numeric code 0, 1, 2, or 3 in the results, as shown in the above chart, rather than the word equivalents of "Different", "Identical", "Source only", and "Target only". The reason for this is one of convenience and clarity. When running the function as part of a larger macro application, retrieving results from variables is easier with numeric codes 0, 1, 2, and 3 than with words. On the other hand, when running the function stand-alone, on-the-fly, etc., it would be easier to digest the results displayed on the screen with words rather than codes.

It would be good at this point to look at an actual example to see what these values return. First we need a couple of folders and some files.

```
x:\Folder1
  File1.txt
  File2.txt (different than File2.txt in Folder2)
  File3.txt (same as File3.txt in Folder2)
x:\Folder2
  File2.txt (different than File2.txt in Folder1)
  File3.txt (same as File3.txt in Folder1)
  File4.txt
  File5.txt
```

Now we will set the following values and run it:

- `ParameterString1` (SourceName) to x:\Folder1
- `ParameterString2` (TargetName) to x:\Folder2
- `ParameterString3` (OutputTo) to S (display results to the screen)
- `ParameterString4` (ShowType) to DIST (all comparisons)

```
Variable Set String %T1% to "x:\Folder1" // Source folder name
Variable Set String %T2% to "x:\Folder2" // Target folder name
Variable Set String %T3% to "S" // Results to screen
Variable Set String %T4% to "DIST" // Do all comparisons
Write Registry String: "ParameterString1" // Write it to the Registry
Write Registry String: "ParameterString2" // Write it to the Registry
Write Registry String: "ParameterString3" // Write it to the Registry
Write Registry String: "ParameterString4" // Write it to the Registry
Macro Run: { File - Compare } // Compare
```

The picture of the comparison results output gives us the following information:

- File1.txt is in the source folder only
- File2.txt is in both folders but are different
- File3.txt is in both folders and are Identical
- File4.txt is in the target folder only
- File5.txt is in the target folder only
Now that you have seen the results of comparing everything, let us chart how setting `ParameterString4` (ShowType) to different values affects the comparison results. The matching numeric codes are in parenthesis.

<table>
<thead>
<tr>
<th>ShowType</th>
<th>File1.txt</th>
<th>File2.txt</th>
<th>File3.txt</th>
<th>File4.txt</th>
<th>File5.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIST</td>
<td>Source Only (2)</td>
<td>Different (0)</td>
<td>Identical (1)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>DIS</td>
<td>Source Only (2)</td>
<td>Different (0)</td>
<td>Identical (1)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>DST</td>
<td>Source Only (2)</td>
<td>Different (0)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>IST</td>
<td>Source Only (2)</td>
<td>Identical (1)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>DI</td>
<td>Different (0)</td>
<td>Identical (1)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>DS</td>
<td>Source Only (2)</td>
<td>Different (0)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>DT</td>
<td></td>
<td>Different (0)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>IS</td>
<td>Source Only (2)</td>
<td>Identical (1)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>IT</td>
<td>Identical (1)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>ST</td>
<td>Source Only (2)</td>
<td></td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>D</td>
<td>Different (0)</td>
<td></td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>I</td>
<td>Identical (1)</td>
<td></td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>S</td>
<td>Source Only (2)</td>
<td></td>
<td></td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
<td>Target Only (3)</td>
</tr>
</tbody>
</table>

Picking some rows to examine, take a look at the D row first. You are asking the function to compare files with common names in both folders and display only those that are different. File2.txt is the only one that fits the bill. Now look at the I row. Again, you are asking to compare files with common names, but this time to display only those that are identical. File3.txt is the only one. If you notice the DI row, both files are displayed. They both have common names and are either different or identical to their counterparts in the opposite folder. The other files do not appear in the results simply because there is no matching file to do a comparison with in the opposite folder.

In the S row, you want to display files that are only in the source folder, and not the target folder, and File1.txt is the only one like it. The opposite is true for the T row. A file may exist in the target folder, but not the source folder in order to make it through the comparison. The ST row is asking to reveal which files are not in both folders, because the ones that make it through are either in the source folder or the target folder, but not both.

Something to make note of concerning both the S and T ShowType options: They are only valid if both the source and target names (`ParameterString1` and `ParameterString2`) are folder names and not file names. An error will be returned if you attempt to use either one when using file names.

All of these different comparisons were achieved using a folder name for both the source and target values. They are called many-to-many comparisons (as noted at the start). So what about the one-to-many and many-to-one comparisons?

The one-to-many comparison is achieved by placing a file name in the SourceName value (`ParameterString1`) and a folder name into the TargetName value (`ParameterString2`). Do the opposite for a many-to-one comparison.
Let us run through a one-to-many example. The only difference between this, and the last example is a change to both the `ParameterString1` and `ParameterString4` values:

`ParameterString1` (SourceName) to `x:\Folder1\File3.txt`
`ParameterString2` (TargetName) to `x:\Folder2`
`ParameterString3` (OutputTo) to `S` (display results to the screen)
`ParameterString4` (ShowType) to `DI` (show all file comparisons)

```
Variable Set String %T1% to "x:\Folder1\File3.txt"  // Source file name
Variable Set String %T2% to "x:\Folder2"      // Target folder name
Variable Set String %T3% to "S"               // Results to screen
Variable Set String %T4% to "DI"              // Show all file comparisons
Write Registry String: "ParameterString1"    // Write it to the Registry
Write Registry String: "ParameterString2"    // Write it to the Registry
Write Registry String: "ParameterString3"    // Write it to the Registry
Write Registry String: "ParameterString4"    // Write it to the Registry
Macro Run: { File - Compare }              // Compare
```

The picture of the comparison results output gives us the following information:

File2.txt in the target folder is different than file3.txt in the source folder
File3.txt in the target folder is identical to file3.txt in the source folder
File4.txt in the target folder is different than file3.txt in the source folder
File5.txt in the target folder is different than file3.txt in the source folder

Now that you have seen the results of a one-to-many comparison of file3.txt, let us chart how setting `ParameterString4` (ShowType) to different values (there are only two others) affects results. Again, matching codes are placed inside parenthesis.

<table>
<thead>
<tr>
<th>ShowType</th>
<th>File2.txt</th>
<th>File3.txt</th>
<th>File4.txt</th>
<th>File5.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>Different (0)</td>
<td>Identical (1)</td>
<td>Different (0)</td>
<td>Different (0)</td>
</tr>
<tr>
<td>D</td>
<td>Different (0)</td>
<td></td>
<td>Different (0)</td>
<td>Different (0)</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>Identical (1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Take a look at the **DI** row first. You are asking the function to compare File3.txt in the source folder with every file in the target folder and display the results for each because, naturally, there can be only **two** possible results for each comparison; *different or identical*. The **D** row is the same comparison, but you are displaying only those files in the target folder that are different, while **I** row displays those that are identical, which is only File3.txt.

The many-to-one comparison is opposite. As an example, you could ask the function to compare every file in the source folder to File3.txt in the target folder by simply changing **ParameterString1** and **ParameterString2**.

**ParameterString1** *(SourceName)* to *x:\Folder1*
**ParameterString2** *(TargetName)* to *x:\Folder2\File3.txt*
**ParameterString3** *(OutputTo)* to **S** (display results to the screen)
**ParameterString4** *(ShowType)* to **DI** (show all file comparisons)

The one-to-one comparison is also similar. Ask the function to compare File3.txt in the source folder to the File3.txt file in the target folder by changing **ParameterString1** and **ParameterString2**.

**ParameterString1** *(SourceName)* to *x:\Folder1\File3.txt*
**ParameterString2** *(TargetName)* to *x:\Folder2\File3.txt*
**ParameterString3** *(OutputTo)* to **S** (display results to the screen)
**ParameterString4** *(ShowType)* to **DI** (show all file comparisons)

Leaving **ParameterString4** as shown, will display the results for either comparison. Setting the value to **D** in our example, would return a blank string, because there were no errors, but neither was there anything to display.

**{ PGM Function } Parameters**

- **SourceName or Srce** - Set to empty string if omitted (input via a browse dialog)
- **TargetName or Tgt** - Set to empty string if omitted (input via a browse dialog)
- **IncludeSubfolders? or Subf** - Set to zero if omitted
- **OutputTo or Out** - Set to empty string if omitted
- **ShowType or Show** - Set to empty string if omitted

**Returns**

- **%T90%** - If **ParameterString3** is set to **V**, comparison is saved here
- **Filename** - If **ParameterString3** is set to **F**, comparison is saved to file
- **Screen** - If **ParameterString3** is set to **S**, comparison is written to the screen
- **ReturnString1** - If **ParameterString3** is set to **R**, or is empty, comparison is saved here (default)
- **ReturnValue?** - 0 if no error occurs, or 1 if an error occurs

**Calls**

- **{ Variables - Save DT }**
- **{ Variables - Restore DT }**
- **{ Program - Log Event }**
Example

Compare all the files in one folder to all the files in another and list which ones exist only in the source folder, which ones exist only in the target folder, and for those that exist in both, which are identical and which are different then send the output to a text file. Look in the sub-folders, too!

Variable Set String %T1% to "x:\Folder1"    // Source folder name
Variable Set String %T2% to "x:\Folder2"    // Target folder name
Variable Set String %T3% to "Fx:\out.txt"    // Results to file
Variable Set String %T4% to "DIST"    // Do all comparisons
Variable Set Integer %N1% to 1      // Look in sub-folders
Write Registry String: "ParameterString1"    // Write it to the Registry
Write Registry String: "ParameterString2"    // Write it to the Registry
Write Registry String: "ParameterString3"    // Write it to the Registry
Write Registry String: "ParameterString4"    // Write it to the Registry
Write Registry Integer: "ParameterInteger1"    // Write it to the Registry
Macro Run: { File - Compare }      // Compare

Same example using \{ PGM Function \}:

Variable Set String %T1% "{ File - Compare }, x:\Folder1, x:\Folder2, 1, Fx:\out.txt, DIST"   // Create PGM Function string
Write Registry String: "PgmFunction"            // Write it to the Registry
Macro Run: { PGM Function }               // Call it

Now here is what the same string would look like using named parameters:

Variable Set String %T1% "{ File - Compare }, SourceName = x:\Folder1, TargetName = x:\Folder2, IncludeSubFolders? = 1, OutputTo = Fx:\out.txt, ShowType = DIST"

Or:

Variable Set String %T1% "{ File - Compare }, Srce = x:\Folder1, Tgt = x:\Folder2, Subf = 1, Out = Fx:\out.txt, Show = DIST"
{ File - Parse Path Name }

Separates a path name into its different components; drive, path, file, and extension and returns each in its own value. Network paths (those beginning like \Server) cannot be parsed so use drive mappings instead.

Parameters
ParameterString1 - Path name to parse
ParameterInteger1 - Return long path name (0=false 1=true). Defaults to true

You would normally want to parse a complete path name, however if you omit the path portion then the current folder is assumed to be the correct path. Say that you want to parse the complete path "C:\Program Files\Macro Express3\MacEdit.exe", but you only pass "MacEdit.exe", furthermore your current folder is "C:\temp":

Variable Set String %T1% to "MacEdit.exe" // Path name to parse
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: { File - Parse Path Name } // Parse it
Read Registry String: "ReturnString1" // T1 = C:
Read Registry String: "ReturnString2" // T2 = \temp\nRead Registry String: "ReturnString3" // T3 = MacEdit
Read Registry String: "ReturnString4" // T4 = .exe

Take a good look at the %T2% and %T3% results. Not exactly what you want unless your intention was to simply parse the file name and extension, which of course you can do.

This function also works with folder shortcuts "." (current folder) and ".." (parent folder). They simply get converted to their actual names. The following example assumes that your current folder is "C:\temp" and the file name is "myfile.txt".

Variable Set String %T1% to ".\MyFile.txt" // Path name to parse
Write Registry String: "ParameterString1" // Write it to the Registry
Macro Run: { File - Parse Path Name } // Parse it
Read Registry String: "ReturnString1" // T1 = C:
Read Registry String: "ReturnString2" // T2 = \temp\nRead Registry String: "ReturnString3" // T3 = MyFile
Read Registry String: "ReturnString4" // T4 = .txt

Use the ParameterInteger1 value to have the path portion of the name returned as a long file name (Windows) or a short file name (DOS).

Variable Set String %T1% to "C:\Program Files\Macro Express3\MacEdit.exe" // Path name to parse
Variable Set Integer %N1% to 0 // Short (DOS) name
Write Registry String: "ParameterString1" // Write it to the Registry
Write Registry Integer: "ParameterInteger1" // Write it to the Registry
Macro Run: { File - Parse Path Name } // Parse it
Read Registry String: "ReturnString1" // T1 = C:
Read Registry String: "ReturnString2" // T2 = \Progra~1\Macroe~1\nRead Registry String: "ReturnString3" // T3 = MacEdit
Read Registry String: "ReturnString4" // T4 = .exe
Notice the %T2% variable. It's a short, DOS compatible name (yours may be different). It's important to note that this feature will only work if it's a real folder. If the folder does not exist then this option is ignored.

{ PGM Function } Parameters

PathName or Path - Path name to parse
LongName or Long - Set to 1 if omitted

Returns

ReturnString1 - Drive
ReturnString2 - Path
ReturnString3 - File
ReturnString4 - Extension

Calls

{ Variables - Save DT }
{ Variables - Restore DT }
{ Program - Log Event }

Example

Parse a full path name and return the path as a long Windows name.

```
Variable Set String %T1% to "c:\Program Files\Macro Express3\MacEdit.exe" // Path name to parse
Variable Set Integer %N1% to 1 // Long (Windows) name
Write Registry String: "ParameterString1" // Write it to the Registry
Write Registry Integer: "ParameterInteger1" // Write it to the Registry
Macro Run: { File - Parse Path Name } // Parse it
Read Registry String: "ReturnString1" // T1 = c:
Read Registry String: "ReturnString2" // T2 = \Program Files\Macro Express\MacEdit.exe
Read Registry String: "ReturnString3" // T3 = MacEdit
Read Registry String: "ReturnString4" // T4 = .exe
```

Same example using { PGM Function }:

```
Variable Set String %T1% "{ File - Parse Path Name }, c:\Program Files\Macro Express3\MacEdit.exe" // Create string
Write Registry String: "PgmFunction" // Write it
Macro Run: { PGM Function } // Call it
Read Registry String: "ReturnString1" // T1 = c:
Read Registry String: "ReturnString2" // T2 = \Program Files\Macro Express\MacEdit.exe
Read Registry String: "ReturnString3" // T3 = MacEdit
Read Registry String: "ReturnString4" // T4 = .exe
```

Now here is what the same string would look like using named parameters:

```
Variable Set String %T1% "{ File - Parse Path Name }, PathName = c:\Program Files\Macro Express3\MacEdit.exe"
```

Or:

```
Variable Set String %T1% "{ File - Parse Path Name }, Path = c:\Program Files\Macro Express3\MacEdit.exe"
```
**{ File - Unique File Name }**

Returns a unique full path and file name to the user's temporary folder specified in the %temp% system environment variable. The file name created is derived from a random letter, today's date and the current time, including milliseconds. All temporary files generated with this function contain a .tmp extension. Two full path and file names are returned, a full Windows name and also its shortened DOS name.

**Parameters**
None.

**Returns**
ReturnString1 - Full Windows path and unique filename  
ReturnString2 - Shortened DOS compatible path and filename

**Calls**
{ Variables - Save DT }  
{ Variables - Restore DT }

**Example**
Create a unique file name.

```plaintext
Macro Run: { File - Unique File Name } // Create a unique name  
Read Registry String: "ReturnString1" // Read Windows results into T1  
Read Registry String: "ReturnString2" // Read DOS results into T2

%T1% = C:\Documents and Settings\User\Local Settings\Temp\W20030804185423171.tmp  
%T2% = C:\DOCUME~1\USER\LOCALS~1\Temp\W20030~1.TMP
```
{ File - Validate File Name }

Examines a file name to determine if it contains any of the following illegal characters:

\ - Backslash
/ - Forward slash
: - Colon
* - Asterisk
? - Question mark
" - Double quote mark
< - Less-than symbol
> - Greater-than symbol
| - Pipe symbol

The passed string must be a file name only and not a path name or full path name. FileName.txt is okay, but c:\temp\FileName.txt is not.

Parameters
ParameterString1 - File name to examine

{ PGM Function } Parameters
FileName or File - Required

Returns
ReturnError? - 0 (no errors), file name is valid, or 1 (error), file name is invalid

Calls
{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Program - Log Event }

Example
Determine if File:Name.txt is valid.

Variable Set String %T1% "File:Name.txt" // File name to examine
Write Registry String: "ParameterString1" // Write string
Macro Run: { File - Validate File Name } // Validate it
Read Registry Integer: "ReturnError?" // Read the results, N1 = 1 (invalid)

Determine if FileName.txt is valid.

Variable Set String %T1% "FileName.txt" // File name to examine
Write Registry String: "ParameterString1" // Write string
Macro Run: { File - Validate File Name } // Validate it
Read Registry Integer: "ReturnError?" // Read the results, N1 = 0 (valid)
Same two examples using \texttt{PGM Function}:

\begin{verbatim}
Variable Set String %T1% "\{ File - Validate File Name \}, File:Name.txt"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: \{ PGM Function \}  // Call it
Read Registry Integer: "ReturnError?"  // Read the results, N1 = 1 (invalid)

Variable Set String %T1% "\{ File - Validate File Name \}, FileName.txt"  // Create PGM Function string
Write Registry String: "PgmFunction"  // Write it to the Registry
Macro Run: \{ PGM Function \}  // Call it
Read Registry Integer: "ReturnError?"  // Read the results, N1 = 0 (valid)
\end{verbatim}

And using \textit{named} parameters:

\begin{verbatim}
Variable Set String %T1% "\{ File - Validate File Name \}, FileName = File:Name.txt"
\end{verbatim}

Or:

\begin{verbatim}
Variable Set String %T1% "\{ File - Validate File Name \}, File = File:Name.txt"
\end{verbatim}

Second example:

\begin{verbatim}
Variable Set String %T1% "\{ File - Validate File Name \}, FileName = FileName.txt"
\end{verbatim}

Or:

\begin{verbatim}
Variable Set String %T1% "\{ File - Validate File Name \}, File = FileName.txt"
\end{verbatim}
{ File - Validate Path Name }

Examines a path name to determine if it contains any of the following illegal characters:

- \( <sp> \) - Backslash and space
- :<sp> - Colon and space
- / - Forward slash
- : - Colon (not in second position)
- * - Asterisk
- ? - Question mark
- " - Double quote mark
- < - Less-than symbol
- > - Greater-than symbol
- | - Pipe symbol

Also, if there is a colon in the second position, then the first character must be Aa through Zz. If the passed string contains a filename, then it is assumed to be part of the path and not a filename. Valid path names can contain relative addressing shorthand characters .\ and ..\ For example, ..\temp means the folder named "temp" within the parent directory of where I am right now.

Parameters
ParameterString1 - File name to examine

{ PGM Function } Parameters
PathName or Path - Required

Returns
ReturnError? - 0 (no errors), path name is valid, or 1 (error), path name is invalid

Calls
{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Program - Log Event }

Example
Determine if the local c:\my documents\user\temp string is a valid path name.

Variable Set String %T1% "c:\my documents\user\temp" // Path name to examine
Write Registry String: "ParameterString1" // Write string
Macro Run: { File - Validate Path Name } // Validate it
Read Registry Integer: "ReturnError?" // Read the results, N1 = 0 (valid)
Determine if the network `\server\user\temp` string is a valid path name.

```
Variable Set String %T1% "\server\user\temp"      // Path name to examine
Write Registry String: "ParameterString1"         // Write string
Macro Run: { File - Validate Path Name }          // Validate it
Read Registry Integer: "ReturnValue?"            // Read the results, N1 = 0 (valid)
```

Same two examples using `{ PGM Function }`

```
Variable Set String %T1% "{ File - Validate Path Name }, c:\my documents\user\temp"      // Create PGM Function string
Write Registry String: "PgmFunction"         // Write it to the Registry
Macro Run: { PGM Function }          // Call it
Read Registry Integer: "ReturnValue?"            // Read the results, N1 = 0 (valid)

Variable Set String %T1% "{ File - Validate Path Name }, \server\user\temp"      // Create PGM Function string
Write Registry String: "PgmFunction"         // Write it to the Registry
Macro Run: { PGM Function }          // Call it
Read Registry Integer: "ReturnValue?"            // Read the results, N1 = 0 (valid)
```

And using named parameters:
```
Variable Set String %T1% "{ File - Validate Path Name }, PathName = c:\my documents\user\temp"
Or:
Variable Set String %T1% "{ File - Validate Path Name }, Path = c:\my documents\user\temp"

Second example:
```
Variable Set String %T1% "{ File - Validate Path Name }, PathName = \server\user\temp"
Or:
Variable Set String %T1% "{ File - Validate Path Name }, Path = \server\user\temp"
```
Developer Tools Category

It's easy to cry 'bug' when the truth is that you've got a complex system and sometimes it takes a while to get all the components to co-exist peacefully.

-Doug Vargas
Functions

The Developer Tools category is new to this release of the PGM Functions Library™. It is planned that functions used in the development stage are placed here.

One of the goals of our company is to promote and support the Macro Express community by developing tools that make the task of creating automation macros easier, which is why the PGM Functions Library™ exists. We have found that there are plenty of tools that make programming and development within Macro Express faster, and easier. We are users the same as any other, and having macro tools to create macro programs is just plain slick.

It currently contains three functions all used for timing the speed of your macros during development. More functions will be added as we convert the ones that we created for ourselves to those that others can easily use.

{ Utility - Macro Timer } Time the speed of a whole function or just a section or single line
{ Utility - Start Timer } Programmatically time the speed
{ Utility - Get Timer } Calculates the elapsed time
{ Utility - Macro Timer }

This function gives you the ability to time the speed of a whole function, a partial section, or even just a single line. It should be invoked only via its hotkey and is meant to test speed, not code, therefore, do not run it on untested functions or sections that remain untested. Also, non-contiguous selected lines will not work.

HotKey Activation (as distributed)
Ctrl+Alt+Shift+T

Calls
{ Utility - Start Timer }

How to Use
First you need to decide what you want to test. If it is a whole function, then highlight it within the right-hand pane of the Macro Explorer window and then start this function via its hotkey.

If it is just a section of code to test, then highlight it within the Script Editor window and then start this function via its hotkey.
Macro Express runs so quickly (it can add two integers together 100,000 times in just a single second) that a repeat loop is needed in order to return meaningful results, so a prompt will be displayed once you engage the macro.

![Enter Value dialog]

Note, however, that a **Macro Stop** or **Macro Return** command in your chosen section of code will render the repeat loop useless because either command will, of course, stop the macro when encountered.

Change this value to suit the current situation based on what you want to time. I used 1,000 loops to time the highlighted section shown on the previous page. Take a look at the results below. It took less than 1/3 of a second.

![Get Timer dialog]

Don’t let the **Get Timer** window title throw you. This function is called by **Start Timer** to display the results and unlike the **Start Timer**, timing results will always be displayed in the above dialog when this function is used.
{ Utility - Start Timer }

Resets the timer then records the current time to the nearest millisecond. This function is different from the { Utility - Macro Timer } function in that it is used strictly within a function along with its companion { Utility - Get Timer } function to wrap a section of code that you want to time. Because it is run with the rest of the code, there is no hotkey. Also, there is no repeat loop for the code that you are timing unless, of course, you are actually timing a repeat loop. As before, however, it should not be run with untested code.

Parameters
ElapsedTimeDisplay? - If set to 1 then speed results are displayed in a textbox

{ PGM Function } Parameters
DisplayFlag or Flag - Set to 1 if omitted

Returns
ReturnString1 - (indirectly) Speed results from { Utility - Get Timer }

Calls
{ Variables - Save 25 }
{ Variables - Restore 25 }
{ Utility - Get Timer }

Example
Test the speed of a repeat loop and display the results in a textbox.

Variable Set Integer %N10% to 0     // Set variable to zero
Macro Run: { Utility - Start Timer }     // Begin timer
Repeat Start (Repeat 99999 times)    // Loop 99,999 times
  Variable Modify Integer: Inc (%N10%)    // Increment %N10% by one
Repeat End        // End loop
Macro Run: { Utility - Get Timer }     // Stop timer and display speed results

![Utility - Get Timer](image-url)
Same example using \(\text{PGM Function}\) and placing results in \text{ReturnString1} rather than the textbox.

```plaintext
Variable Set Integer \%N10\% to 0       // Set variable to zero
Variable Set String \%T1\% \{ Utility - Start Timer \}, 0"  // Create PGM Function string
Write Registry String: "PgmFunction"      // Write it to the Registry
Macro Run: \{ PGM Function \}        // Call it (start timer)
Repeat Start (Repeat 99999 times)      // Loop 99,999 times
  Variable Modify Integer: Inc \(%N10\%\)      // Increment \%N10\% by one
Repeat End           // End loop
Macro Run: \{ Utility - Get Timer \}       // Stop timer and display speed results
Read Registry String: "ReturnString1"      // Read the results, \text{T2} = "00:00:000.516"
```

And using \textit{named} parameters:

```plaintext
Variable Set String \%T1\% \{ Utility - Start Timer \}, DisplayFlag = 0"
```

Or:

```plaintext
Variable Set String \%T1\% \{ Utility - Start Timer \}, Flag = 0"
```
{ Utility - Get Timer }

Companion function to { Utility - Get Timer } it calculates the current time and subtracts the start time value resulting in an elapsed time.

Parameters
ElapsedTimeDisplay? - Passed by the { Utility - Start Timer } function. If set to 1 then results are displayed in a textbox (copied from { Utility - Start Timer }).

Returns
ReturnString1 - Speed results (and optionally displays in textbox)

Calls
{ Variables - Save DT }
{ Variables - Restore DT }
{ Math - Mod }

Example
See example in { Utility - Start Timer } description.
You asked for it …

An FBI agent is interviewing a bank teller after the bank had been robbed 3 times successively by the same bandit. The FBI agent asks, "Did you notice anything special about the man?" "Yes," replies the teller, "He was better dressed each time."
Change Log (Book)

2003.10.06 - Original release date.
Change Log (Library)

2003.09.01 - Original release date.
## Alphabetical Function List

### Date and Time Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ DateTime - Character DOW }</td>
<td>Converts a day-of-week number to a character day-of-week</td>
</tr>
<tr>
<td>{ DateTime - Character Month }</td>
<td>Converts a month number to a character month</td>
</tr>
<tr>
<td>{ DateTime - Date to Character DOW }</td>
<td>Converts a date to a character day-of-week</td>
</tr>
<tr>
<td>{ DateTime - Date to DOY }</td>
<td>Converts a date to a numeric day-of-year</td>
</tr>
<tr>
<td>{ DateTime - Date to Julian }</td>
<td>Converts a date string to a Julian Day number</td>
</tr>
<tr>
<td>{ DateTime - DOY to Date }</td>
<td>Converts a year and a numeric day-of-year to a date string.</td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td>Evaluates a year to determine if it is a leap year</td>
</tr>
<tr>
<td>{ DateTime - Format Input String }</td>
<td>Converts a date string to a formatted string (YYYMDD)</td>
</tr>
<tr>
<td>{ DateTime - Go Days }</td>
<td>Calculates a new date based on the number of days input</td>
</tr>
<tr>
<td>{ DateTime - Go Months }</td>
<td>Calculates a new date based on the number of months input</td>
</tr>
<tr>
<td>{ DateTime - Julian to Character DOW }</td>
<td>Converts a Julian Day number to a character day-of-week</td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td>Converts a Julian Day number to a date string</td>
</tr>
<tr>
<td>{ DateTime - Julian to DOY }</td>
<td>Converts a Julian Day number to a numeric day-of-year</td>
</tr>
<tr>
<td>{ DateTime - Julian to DOY }</td>
<td>Converts a Julian Day number to a numeric day-of-year</td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Date }</td>
<td>Returns the DD part of a date string as an integer</td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Julian }</td>
<td>Returns the day-of-month from a Julian date</td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Today }</td>
<td>Returns the DD part of today as an integer</td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td>Returns the MM part of a date string as an integer</td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Today }</td>
<td>Returns the MM part of today as an integer</td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Date }</td>
<td>Returns the YYYY part of a date string as an integer</td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Today }</td>
<td>Returns the YYYY part of today as an integer</td>
</tr>
<tr>
<td>{ DateTime - Reset Defaults }</td>
<td>Resets the Date and Time values back to their default values</td>
</tr>
<tr>
<td>{ DateTime - Set Defaults }</td>
<td>Interactively set the Date and Time default values</td>
</tr>
<tr>
<td>{ DateTime - Span of Days }</td>
<td>Calculates the span, or number, of days between two dates</td>
</tr>
<tr>
<td>{ DateTime - Today to Character DOW }</td>
<td>Converts today to a character day-of-week</td>
</tr>
<tr>
<td>{ DateTime - Today to DOY }</td>
<td>Converts today to a numeric day-of-week</td>
</tr>
<tr>
<td>{ DateTime - Today to Julian }</td>
<td>Converts today to a Julian Day number</td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td>Performs a validation check on an input date string</td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td>Performs a validation check on a Julian Day number</td>
</tr>
</tbody>
</table>

### File Operations Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ File - Backslash Append }</td>
<td>Adds a trailing backslash to a path name</td>
</tr>
<tr>
<td>{ File - Backslash Remove }</td>
<td>Removes any trailing backslashes from a path name</td>
</tr>
<tr>
<td>{ File - Compare }</td>
<td>Compare files, folders, and sub-folders in five different ways</td>
</tr>
<tr>
<td>{ File - Parse Path Name }</td>
<td>Separates a path name into its different components</td>
</tr>
<tr>
<td>{ File - Unique File Name }</td>
<td>Returns a unique full path and file name</td>
</tr>
<tr>
<td>{ File - Validate File Name }</td>
<td>Examines a file name for illegal characters</td>
</tr>
<tr>
<td>{ File - Validate Path Name }</td>
<td>Examines a path name for illegal characters</td>
</tr>
</tbody>
</table>
Math and Science Category

- **Math - Abs**: Returns the absolute value of a number
- **Math - Ceiling**: Returns the nearest whole number that is >= the number passed
- **Math - Decimal to Hex**: Converts a large number to a hexadecimal string
- **Math - Degrees to Radians**: Converts degrees to radians
- **Math - Exp**: Returns the value of \(e^x\) (e raised to the power of x)
- **Math - Floor**: Returns the nearest whole number that is <= the passed number
- **Math - Hex to Decimal**: Converts a large hexadecimal strings to number
- **Math - Hex to Integer**: Converts a hexadecimal string to an integer
- **Math - Integer to Hex**: Converts an integer to a hexadecimal string
- **Math - Mod**: Returns the remainder of dividing X by Y
- **Math - Pi**: Returns the value of Pi to 14 decimal places
- **Math - Radians to Degrees**: Converts radians to degrees
- **Math - Sign**: Returns a value based on a number having a sign, no sign or is zero
- **Math - Square Root**: Returns the square root of a number

PGM System Category

- **PGM**: System functions handler
- **PGM Bug Report**: Creates trouble report and emails to support@pgmacros.com
- **PGM Function**: Master function that you can use to call other parameter-driven functions
- **PGM Function Error**: Error handler for { PGM Function }
- **PGM Help**: Context-sensitive help system
- **PGM Registration**: Registers you license and/or displays registration information
- **PGM Setup**: Variable management handler
- **PgmSvr - Process**: Processor for the Visual Basic DLL library interface

Program Operations Category

- **Program - Clear Error Flag**: Resets the ReturnError? value back to zero after an error
- **Program - Focus**: Sets focus to a program that is already running
- **Program - Launch**: Launches a program and sets focus to it
- **Program - Log Event**: Records the event string to the log file.
- **Program - Reset**: Resets Registry values back to default values
- **Program - Terminate**: Terminates a running program
- **Program - Toggle Event Logging**: Toggles the LogEvent? value on and off
- **Program - Toggle Exact Match**: Toggles the ExactMatch? value on and off
- **Program - Toggle Parameter Validation**: Toggles the ValidateParameters? value on and off
- **Program - Validate Launch Values**: Checks that launch values fall within acceptable limits
- **Program - Validate Termination Values**: Checks that the termination values falls within limits
- **Program - Validate Verification Values**: Checks that the verify values fall within acceptable limits
String Category
- **{ String - Occurs }** Returns the number of occurrences of a one string inside another string
- **{ String - Pad }** Pads a string on the left or right sides, or centers with a specific pad character
- **{ String - Pad Center }** Centers a string (pads both sides) using a specific pad character
- **{ String - Pad Left }** Pads a string on the left side using a specific pad character
- **{ String - Pad Right }** Pads a string on the right side using a specific pad character
- **{ String - Parse }** Parses an input string based on a character or string delimiter
- **{ String - Proper }** Converts a string to proper case where each word begins with a capital letter
- **{ String - Replicate }** Replicates a string
- **{ String - Reverse }** Reverses a string
- **{ String - Search }** Returns the starting position of one string inside another string

Developer Tools Category
- **{ Utility - Get Timer }** Calculates the elapsed time
- **{ Utility - Macro Timer }** Time the speed of a whole function or just a section or single line
- **{ Utility - Start Timer }** Programmatically time the speed

Variable Management Category
- **{ Variables - Reset Current Level }** Resets the area pointer back to zero
- **{ Variables - Reset Registry Values }** Resets all variables in all areas to their default empty values
- **{ Variables - Restore }** Restores all 297 variables
- **{ Variables - Restore 25 }** Restores the first 25 of all types
- **{ Variables - Restore DT }** Restores the first 25 string and integers
- **{ Variables - Restore SI }** Restores all 198 string and integers
- **{ Variables - Save }** Saves 297 variables T1-T99, N1-N99, and D1-D99
- **{ Variables - Save 25 }** Saves the first 25 of all types T1-T25, N1-N25, and D1-D25
- **{ Variables - Save DT }** Saves the first 25 string and integers T1-T25, N1-N25, D1-D25
- **{ Variables - Save SI }** Saves all 198 string and integers T1-T99 and N1-N99
Function Dependency

The *Functions Called* list is a list of what functions are called by a particular function. The *Functions Called By* list is the opposite. It tells you that a particular function is called by which functions. Also, these lists are not the same as what is needed if you export functions. Refer to the *Exporting Functions* section of the You Need to Know chapter. Uhm, confused? Well, take a break. Here is a little dyslexic humor.

Functions Called

<table>
<thead>
<tr>
<th>PGM System Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This function</strong></td>
</tr>
<tr>
<td>{ PGM }</td>
</tr>
<tr>
<td>{ PGM Setup }</td>
</tr>
<tr>
<td>{ PGM Registration }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ PgmSvr - Process }</td>
</tr>
<tr>
<td>{ PGM Function }</td>
</tr>
<tr>
<td>{ PGM Function Error}</td>
</tr>
<tr>
<td>{ PGM Help }</td>
</tr>
<tr>
<td>{ PGM Bug Report }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable Management Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This function</strong></td>
</tr>
<tr>
<td>{ Variables - Save }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Save SI }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore SI }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
</tr>
</tbody>
</table>
### Program Operations Category

<table>
<thead>
<tr>
<th>This function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Program - Launch }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Validate Launch Values }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Validate Verification Values }</td>
</tr>
<tr>
<td>{ Program - Focus }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Validate Verification Values }</td>
</tr>
<tr>
<td>{ Program - Terminate }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Validate Termination Values }</td>
</tr>
<tr>
<td>{ Program - Validate Launch Values }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Program - Validate Verification Values }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Program - Validate Termination Values }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Program - Toggle Parameter Validation }</td>
<td>Nothing</td>
</tr>
<tr>
<td>{ Program - Toggle Exact Match }</td>
<td>Nothing</td>
</tr>
<tr>
<td>{ Program - Toggle Event Logging }</td>
<td>Nothing</td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td>Nothing</td>
</tr>
<tr>
<td>{ Program - Reset }</td>
<td>Nothing</td>
</tr>
<tr>
<td>{ Program - Clear Error Flag }</td>
<td>Nothing</td>
</tr>
</tbody>
</table>
### Date and Time Category

<table>
<thead>
<tr>
<th>This function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ DateTime - Date to Julian }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Julian Number }</td>
</tr>
<tr>
<td>{ DateTime - Go Months }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td>{ DateTime - Go Days }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to Date }</td>
</tr>
<tr>
<td>{ DateTime - Span of Days }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td>{ DateTime - Julian to DOY }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to DOY }</td>
</tr>
<tr>
<td>{ DateTime - Date to DOY }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td>{ DateTime - DOY to Date }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>Function Type</td>
<td>Functions</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to DOW</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Character DOW</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Julian to Date</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOW</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to DOW</td>
<td></td>
</tr>
<tr>
<td>DateTime - Date to Character DOW</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Character DOW</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Character Month</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>Math - Mod</td>
<td></td>
</tr>
<tr>
<td>DateTime - Format Input String</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Date String</td>
<td></td>
</tr>
<tr>
<td>Variables - Save DT</td>
<td></td>
</tr>
<tr>
<td>Variables - Restore DT</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Code</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Program - Log Event</td>
<td></td>
</tr>
<tr>
<td>DateTime - Evaluate for Leap Year</td>
<td></td>
</tr>
<tr>
<td>DateTime - Validate Julian Number</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td>DateTime - Parse Day from Date</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Validate Date String</td>
</tr>
<tr>
<td>DateTime - Parse Day from Julian</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Julian to Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Day from Date</td>
</tr>
<tr>
<td>DateTime - Parse Month from Date</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Validate Date String</td>
</tr>
<tr>
<td>DateTime - Parse Month from Julian</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Julian to Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Month from Date</td>
</tr>
<tr>
<td>DateTime - Parse Year from Date</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Validate Date String</td>
</tr>
<tr>
<td>DateTime - Parse Year from Julian</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Julian to Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Year from Date</td>
</tr>
<tr>
<td>DateTime - Parse Day from Today</td>
<td>Variables - Save DT</td>
</tr>
<tr>
<td></td>
<td>Variables - Restore DT</td>
</tr>
<tr>
<td></td>
<td>Program - Log Event</td>
</tr>
<tr>
<td></td>
<td>DateTime - Validate Date String</td>
</tr>
<tr>
<td>DateTime - Parse Month from Today</td>
<td>Variables - Save DT</td>
</tr>
</tbody>
</table>
### DateTime Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ DateTime - Parse Year from Today }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Today to DOW }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to DOW }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Today to Character DOW }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to Character DOW }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Today to DOY }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to DOY }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Today to Julian }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to Julian }</td>
<td></td>
</tr>
</tbody>
</table>

### String Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ String - Occurs }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Pad }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ String - Pad Center }</td>
<td>{ Variables - Save DT }</td>
</tr>
</tbody>
</table>
Math and Science Category

<table>
<thead>
<tr>
<th>This function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Math - Abs }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Sign }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>Function</td>
<td>Action</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>{ Math - Exp }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Square Root }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Ceiling }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Floor }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Degrees to Radians }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Radians to Degrees }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Decimal to Hex }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Hex to Decimal }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Math - Integer to Hex }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Decimal to Hex }</td>
</tr>
<tr>
<td>{ Math - Hex to Integer }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Hex to Decimal }</td>
</tr>
</tbody>
</table>
### File Operations Category

<table>
<thead>
<tr>
<th>This function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ File - Compare }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Parse Path Name }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Unique File Name }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td>{ File - Backslash Append }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Backslash Remove }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Validate File Name }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ File - Validate Path Name }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
</tbody>
</table>

### Developer Tools Category

<table>
<thead>
<tr>
<th>This function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Utility - Macro Timer }</td>
<td>{ Utility - Start Timer }</td>
</tr>
<tr>
<td>{ Utility - Start Timer }</td>
<td>{ Variables - Save 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore 25 }</td>
</tr>
<tr>
<td></td>
<td>{ Program - Log Event }</td>
</tr>
<tr>
<td>{ Utility - Get Timer }</td>
<td>{ Variables - Save DT }</td>
</tr>
<tr>
<td></td>
<td>{ Variables - Restore DT }</td>
</tr>
<tr>
<td></td>
<td>{ Math - Mod }</td>
</tr>
</tbody>
</table>
Functions Called By

This list is alphabetical for your convenience. Note that if the function you are researching is not listed in the left-hand column, it is not called by any other function.

<table>
<thead>
<tr>
<th>This function</th>
<th>Is called by these</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ DateTime - Character DOW }</td>
<td>{ DateTime - Date to Character DOW }</td>
</tr>
<tr>
<td>{ DateTime - Date to Character DOW }</td>
<td>{ DateTime - Julian to Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to Character DOW }</td>
</tr>
<tr>
<td>{ DateTime - Date to DOW }</td>
<td>{ DateTime - Date to Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to DOW }</td>
</tr>
<tr>
<td>{ DateTime - Date to DOY }</td>
<td>{ DateTime - Julian to DOY }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to DOY }</td>
</tr>
<tr>
<td>{ DateTime - Date to Julian }</td>
<td>{ DateTime - Date to DOY }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Go Days }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Span of Days }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to Julian }</td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td>{ DateTime - DOY to Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Go Months }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td>{ DateTime - Go Days }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to DOY }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Day from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Month from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Year from Julian }</td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Date }</td>
<td>{ DateTime - Parse Day from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Day from Today }</td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td>{ DateTime - Parse Month from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Month from Today }</td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Date }</td>
<td>{ DateTime - Parse Year from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Year from Today }</td>
</tr>
<tr>
<td>{ DateTime - Validate Date String }</td>
<td>{ DateTime - Date to DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Format Input String }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Go Months }</td>
</tr>
<tr>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td></td>
</tr>
<tr>
<td>{ File - Unique File Name }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Bug Report }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Decimal to Hex }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Integer to Hex }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Hex to Decimal }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Hex to Integer }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Date to DOW }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Go Months }</td>
<td></td>
</tr>
<tr>
<td>{ Utility - Get Timer }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Pi }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Degrees to Radians }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Radians to Degrees }</td>
<td></td>
</tr>
<tr>
<td>{ PGM }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Registration }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Reset Current Level }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Reset Registry Values }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore SI }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save SI }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Function Error }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Function }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Registration }</td>
<td></td>
</tr>
<tr>
<td>{ PGM }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Setup }</td>
<td></td>
</tr>
<tr>
<td>{ PGM Registration }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Reset Current Level }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Reset Registry Values }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore SI }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save 25 }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>{ PgmSvr - Process }</td>
<td>{ Math - Exp }</td>
</tr>
<tr>
<td>{ Program - Log Event }</td>
<td>{ DateTime - Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Character Month }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to DOY }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - DOY to Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Format Input String }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Go Days }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Go Months }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Julian to DOY }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Day from Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Day from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Day from Today }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Month from Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Month from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Month from Today }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Year from Date }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Year from Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Parse Year from Today }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Span of Days }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to Character DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to DOW }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to DOY }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Today to Julian }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Date String }</td>
</tr>
<tr>
<td></td>
<td>{ DateTime - Validate Julian Number }</td>
</tr>
<tr>
<td>{ File - Backslash Append }</td>
<td></td>
</tr>
<tr>
<td>{ File - Backslash Remove }</td>
<td></td>
</tr>
<tr>
<td>{ File - Compare }</td>
<td></td>
</tr>
<tr>
<td>{ File - Parse Path Name }</td>
<td></td>
</tr>
<tr>
<td>{ File - Validate File Name }</td>
<td>{ File - Validate Path Name }</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>{ Math - Abs }</td>
<td>{ Math - Ceiling }</td>
</tr>
<tr>
<td>{ Math - Decimal to Hex }</td>
<td>{ Math - Degrees to Radians }</td>
</tr>
<tr>
<td>{ Math - Exp }</td>
<td>{ Math - Floor }</td>
</tr>
<tr>
<td>{ Math - Hex to Decimal }</td>
<td>{ Math - Hex to Integer }</td>
</tr>
<tr>
<td>{ Math - Integer to Hex }</td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ Math - Radians to Degrees }</td>
<td>{ Math - Sign }</td>
</tr>
<tr>
<td>{ Math - Square Root }</td>
<td>{ Program - Focus }</td>
</tr>
<tr>
<td>{ Program - Launch }</td>
<td>{ Program - Terminate }</td>
</tr>
<tr>
<td>{ Program - Validate Launch Values }</td>
<td>{ Program - Validate Termination Values }</td>
</tr>
<tr>
<td>{ Program - Validate Termination Values }</td>
<td>{ Program - Validate Verification Values }</td>
</tr>
<tr>
<td>{ String - Occurs }</td>
<td>{ String - Pad }</td>
</tr>
<tr>
<td>{ String - Pad }</td>
<td>{ String - Pad Center }</td>
</tr>
<tr>
<td>{ String - Pad Left }</td>
<td>{ String - Pad Right }</td>
</tr>
<tr>
<td>{ String - Parse }</td>
<td>{ String - Proper }</td>
</tr>
<tr>
<td>{ String - Replicate }</td>
<td>{ String - Reverse }</td>
</tr>
<tr>
<td>{ String - Search }</td>
<td>{ Program - Launch }</td>
</tr>
<tr>
<td>{ Program - Validate Launch Values }</td>
<td>{ Program - Terminate }</td>
</tr>
<tr>
<td>{ Program - Validate Termination Values }</td>
<td>{ Program - Focus }</td>
</tr>
<tr>
<td>{ Program - Validate Verification Values }</td>
<td>{ Program - Launch }</td>
</tr>
<tr>
<td>{ String - Pad }</td>
<td>{ String - Pad Center }</td>
</tr>
<tr>
<td>String - Pad Left</td>
<td>String - Pad Right</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>String - Parse</td>
<td>Program - Launch</td>
</tr>
<tr>
<td>String - Replicate</td>
<td>String - Reverse</td>
</tr>
<tr>
<td>Utility - Get Timer</td>
<td>Utility - Start Timer</td>
</tr>
<tr>
<td>Utility - Start Timer</td>
<td>Utility - Macro Timer</td>
</tr>
<tr>
<td>Variables - Restore 25</td>
<td>DateTime - Character DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - Character Month</td>
</tr>
<tr>
<td></td>
<td>DateTime - Date to Character DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - DOY to Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Format Input String</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Day from Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Day from Today</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Month from Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Month from Today</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Year from Date</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Year from Today</td>
</tr>
<tr>
<td></td>
<td>DateTime - Reset Defaults</td>
</tr>
<tr>
<td></td>
<td>DateTime - Set Defaults</td>
</tr>
<tr>
<td></td>
<td>DateTime - Today to Character DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - Today to DOY</td>
</tr>
<tr>
<td></td>
<td>DateTime - Today to DOY</td>
</tr>
<tr>
<td></td>
<td>DateTime - Today to Julian</td>
</tr>
<tr>
<td></td>
<td>DateTime - Validate Date String</td>
</tr>
<tr>
<td></td>
<td>File - Backslash Append</td>
</tr>
<tr>
<td></td>
<td>File - Backslash Remove</td>
</tr>
<tr>
<td></td>
<td>File - Compare</td>
</tr>
<tr>
<td></td>
<td>File - Parse Path Name</td>
</tr>
<tr>
<td></td>
<td>File - Validate File Name</td>
</tr>
<tr>
<td></td>
<td>File - Validate Path Name</td>
</tr>
<tr>
<td></td>
<td>Program - Focus</td>
</tr>
<tr>
<td></td>
<td>Program - Launch</td>
</tr>
<tr>
<td></td>
<td>Program - Reset</td>
</tr>
<tr>
<td></td>
<td>Program - Terminate</td>
</tr>
<tr>
<td></td>
<td>Program - Validate Launch Values</td>
</tr>
<tr>
<td></td>
<td>Program - Validate Termination Values</td>
</tr>
<tr>
<td></td>
<td>Program - Validate Verification Values</td>
</tr>
<tr>
<td>PGM Functions Library™ Appendices</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td>{ String - Occurs }</td>
<td></td>
</tr>
<tr>
<td>{ String - Pad }</td>
<td></td>
</tr>
<tr>
<td>{ String - Pad Center }</td>
<td></td>
</tr>
<tr>
<td>{ String - Pad Left }</td>
<td></td>
</tr>
<tr>
<td>{ String - Pad Right }</td>
<td></td>
</tr>
<tr>
<td>{ String - Parse }</td>
<td></td>
</tr>
<tr>
<td>{ String - Proper }</td>
<td></td>
</tr>
<tr>
<td>{ String - Replicate }</td>
<td></td>
</tr>
<tr>
<td>{ String - Reverse }</td>
<td></td>
</tr>
<tr>
<td>{ String - Search }</td>
<td></td>
</tr>
<tr>
<td>{ Utility - Start Timer }</td>
<td></td>
</tr>
<tr>
<td>{ Variables - Restore DT }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to DOW }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to DOY }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Date to Julian }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Evaluate for Leap Year }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Go Days }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Go Months }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to Character DOW }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to Date }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to DOW }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Julian to DOY }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Julian }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Julian }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Julian }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Span of Days }</td>
<td></td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Abs }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Ceiling }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Decimal to Hex }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Degrees to Radians }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Exp }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Floor }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Hex to Decimal }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Hex to Integer }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Integer to Hex }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td></td>
</tr>
<tr>
<td>{ Math - Radians to Degrees }</td>
<td></td>
</tr>
<tr>
<td>Math - Sign</td>
<td>Math - Square Root</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------</td>
</tr>
<tr>
<td>Variables - Save 25</td>
<td>DateTime - Character DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - Date to Character DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - Format Input String</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Day from Today</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Month from Today</td>
</tr>
<tr>
<td></td>
<td>DateTime - Parse Year from Today</td>
</tr>
<tr>
<td></td>
<td>DateTime - Set Defaults</td>
</tr>
<tr>
<td></td>
<td>DateTime - Today to DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - Today to DOW</td>
</tr>
<tr>
<td></td>
<td>DateTime - Validate Date String</td>
</tr>
<tr>
<td></td>
<td>File - Backslash Remove</td>
</tr>
<tr>
<td></td>
<td>File - Parse Path Name</td>
</tr>
<tr>
<td></td>
<td>File - Validate Path Name</td>
</tr>
<tr>
<td></td>
<td>Program - Launch</td>
</tr>
<tr>
<td></td>
<td>Program - Terminate</td>
</tr>
<tr>
<td></td>
<td>Program - Validate Termination Values</td>
</tr>
<tr>
<td></td>
<td>String - Occurs</td>
</tr>
<tr>
<td></td>
<td>String - Pad Center</td>
</tr>
<tr>
<td>{ String - Pad Left }</td>
<td>{ DateTime - Date to DOW }</td>
</tr>
<tr>
<td>{ String - Pad Right }</td>
<td>{ DateTime - Date to DOY }</td>
</tr>
<tr>
<td>{ String - Parse }</td>
<td>{ DateTime - Date to Julian }</td>
</tr>
<tr>
<td>{ String - Proper }</td>
<td>{ DateTime - Evaluate for Leap Year }</td>
</tr>
<tr>
<td>{ String - Replicate }</td>
<td>{ DateTime - Go Days }</td>
</tr>
<tr>
<td>{ String - Reverse }</td>
<td>{ DateTime - Go Months }</td>
</tr>
<tr>
<td>{ String - Search }</td>
<td>{ DateTime - Julian to Character DOW }</td>
</tr>
<tr>
<td>{ Utility - Start Timer }</td>
<td>{ DateTime - Julian to Date }</td>
</tr>
<tr>
<td>{ Variables - Save DT }</td>
<td>{ DateTime - Julian to DOY }</td>
</tr>
<tr>
<td>{ DateTime - Go Days }</td>
<td>{ DateTime - Julian to Day }</td>
</tr>
<tr>
<td>{ DateTime - Go Months }</td>
<td>{ DateTime - Julian to DOW }</td>
</tr>
<tr>
<td>{ DateTime - Julian to DOY }</td>
<td>{ DateTime - Julian to Julian }</td>
</tr>
<tr>
<td>{ Advances Julian Number }</td>
<td>{ Math - Abs }</td>
</tr>
<tr>
<td>{ DateTime - Parse Day from Julian }</td>
<td>{ Math - Ceiling }</td>
</tr>
<tr>
<td>{ DateTime - Parse Month from Julian }</td>
<td>{ Math - Decimal to Hex }</td>
</tr>
<tr>
<td>{ DateTime - Parse Year from Julian }</td>
<td>{ Math - Degrees to Radians }</td>
</tr>
<tr>
<td>{ DateTime - Span of Days }</td>
<td>{ Math - Exp }</td>
</tr>
<tr>
<td>{ DateTime - Validate Julian Number }</td>
<td>{ Math - Floor }</td>
</tr>
<tr>
<td>{ Math - Hex to Decimal }</td>
<td>{ Math - Hex to Integer }</td>
</tr>
<tr>
<td>{ Math - Integer to Hex }</td>
<td>{ Math - Mod }</td>
</tr>
<tr>
<td>{ Math - Radians to Degrees }</td>
<td>{ Math - Sign }</td>
</tr>
<tr>
<td>{ Math - Square Root }</td>
<td>{ Utility - Get Timer }</td>
</tr>
</tbody>
</table>
### Date and Time Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ DateTime - Character DOW }</code></td>
<td>DayOfWeek or Dow</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Character Month }</code></td>
<td>Month or Mm</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Date to Character DOW }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Date to DOW }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Date to DOY }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Date to Julian }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - DOY to Date }</code></td>
<td>Year or Yr</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Evaluate for Leap Year }</code></td>
<td>Year or Yr</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Format Input String }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Go Days }</code></td>
<td>NumDays or Go</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Go Months }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Julian to Character DOW }</code></td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Julian to DOY }</code></td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Julian to DOW }</code></td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Julian to DOY }</code></td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Parse Day from Date }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Parse Day from Julian }</code></td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Parse Month from Date }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td>`{ DateTime - Parse Month from Julian }</td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td>`{ DateTime - Parse Month from Julian }</td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Parse Year from Date }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td>`{ DateTime - Parse Year from Julian }</td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Span of Days }</code></td>
<td>Date1 or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Validate Date String }</code></td>
<td>Date or Dt</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ DateTime - Validate Julian Number }</code></td>
<td>Julian or Jn</td>
<td>Required</td>
</tr>
</tbody>
</table>

### File Operations Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ File - Backslash Append }</code></td>
<td>PathName or Path</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ File - Backslash Remove }</code></td>
<td>PathName or Path</td>
<td>Required</td>
</tr>
<tr>
<td><code>{ File - Compare }</code></td>
<td>SourceName or Srce</td>
<td>Optional - set to &quot;~&quot; if omitted</td>
</tr>
<tr>
<td></td>
<td>TargetName or Tgt</td>
<td>Optional - set to &quot;~&quot; if omitted</td>
</tr>
<tr>
<td></td>
<td>IncludeSubfolders? or SubF</td>
<td>Optional - set to 0 if omitted</td>
</tr>
<tr>
<td></td>
<td>OutputTo or Out</td>
<td>Optional - set to &quot;~&quot; if omitted</td>
</tr>
<tr>
<td></td>
<td>ShowType or Show</td>
<td>Optional - set to &quot;~&quot; if omitted</td>
</tr>
<tr>
<td><code>{ File - Parse Path Name }</code></td>
<td>PathName or Path</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>LongName or Long</td>
<td>Optional - set to 1 if omitted</td>
</tr>
</tbody>
</table>
### PGM Functions Library™ Appendices

#### Math and Science Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameter</th>
<th>Required Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ File - Validate File Name }</td>
<td>FileName or File</td>
<td>- Required</td>
</tr>
<tr>
<td>{ File - Validate Path Name }</td>
<td>PathName or Path</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Abs }</td>
<td>Number or Num</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Ceiling }</td>
<td>Number or Num</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Decimal to Hex }</td>
<td>Decimal or Dec</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Degrees to Radians }</td>
<td>Degree or Deg</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Exp }</td>
<td>Power or Pwr</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Floor }</td>
<td>Number or Num</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Hex to Decimal }</td>
<td>Hexadecimal or Hex</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Hex to Integer }</td>
<td>Hexadecimal or Hex</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Integer to Hex }</td>
<td>Integer or Int</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Mod }</td>
<td>Dividend or Div</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Radians to Degrees }</td>
<td>Radian or Rad</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Sign }</td>
<td>Number or Num</td>
<td>- Required</td>
</tr>
<tr>
<td>{ Math - Square Root }</td>
<td>Number or Num</td>
<td>- Required</td>
</tr>
</tbody>
</table>

#### PGM System Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameter</th>
<th>Optional Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ PgmSvr - Process }</td>
<td>AppClass or Func</td>
<td>- Required</td>
</tr>
<tr>
<td></td>
<td>MaxWait or Wait</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamDec1 or PD1</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamDec2 or PD2</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamDec3 or PD3</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamInt1 or PI1</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamInt2 or PI2</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamInt3 or PI3</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamStr1 or PT1</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamStr2 or PT2</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>ParamStr3 or PT3</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>RetainClass? or Retain</td>
<td>- Optional - set to 0 if omitted</td>
</tr>
<tr>
<td></td>
<td>DebugMode? or Debug</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>SleepWait or Sleep</td>
<td>- Optional</td>
</tr>
<tr>
<td></td>
<td>UnloadWait or Unload</td>
<td>- Optional</td>
</tr>
</tbody>
</table>

#### Program Operations Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameter</th>
<th>Optional Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Program - Focus }</td>
<td>ProgName or PName</td>
<td>- Required</td>
</tr>
<tr>
<td></td>
<td>WinTitle or WTtitle</td>
<td>- Required</td>
</tr>
<tr>
<td></td>
<td>ExactMatch or Exact</td>
<td>- Optional - set to 0 if omitted</td>
</tr>
<tr>
<td></td>
<td>VerifyLoops or VLoops</td>
<td>- Optional - current value if omitted</td>
</tr>
<tr>
<td></td>
<td>VerifyDelay or VDelay</td>
<td>- Optional - current value if omitted</td>
</tr>
<tr>
<td></td>
<td>ValidateDelay or VDelay</td>
<td>- Optional - current value if omitted</td>
</tr>
<tr>
<td></td>
<td>ValidateParams or VParams</td>
<td>- Optional - current value if omitted</td>
</tr>
<tr>
<td>{ Program - Launch }</td>
<td>ProgName or PName</td>
<td>- Required</td>
</tr>
<tr>
<td></td>
<td>WinTitle or WTtitle</td>
<td>- Required</td>
</tr>
<tr>
<td></td>
<td>ProgFolder or PHome</td>
<td>- Optional - set to &quot;~&quot; if omitted</td>
</tr>
<tr>
<td></td>
<td>ProgParams or PParams</td>
<td>- Optional - set to &quot;~&quot; if omitted</td>
</tr>
<tr>
<td></td>
<td>LaunchState or WShow</td>
<td>- Optional - current value if omitted</td>
</tr>
<tr>
<td></td>
<td>ExactMatch or Exact</td>
<td>- Optional - set to 0 if omitted</td>
</tr>
</tbody>
</table>
LaunchDelay or LDelay - Optional - current value if omitted
VerifyLoops or VLoops - Optional - current value if omitted
VerifyDelay or VDelay - Optional - current value if omitted
ValidateParams or VParams - Optional - current value if omitted
{ Program - Terminate }
  ProgName or PName - Required
  TimeOut or Time - Optional - current value if omitted
VerifyLoops or VLoops - Optional - current value if omitted
VerifyDelay or VDelay - Optional - current value if omitted
ValidateParams or VParams - Optional - current value if omitted

String Category
{ String - Occurs }
  SearchIn or In - Required
  SearchFor or For - Required
  CaseSensitive? or Case - Optional - set to 0 if omitted
{ String - Pad }
  PadString or Str - Required
  FinalLength or Len - Required
  PadDirection or Dir - Optional - set to "L" if omitted
  PadCharacter or Pad - Optional - set to if omitted
{ String - Pad Center }
  PadString or Str - Required
  FinalLength or Len - Required
  PadCharacter or Pad - Optional - set to if omitted
{ String - Pad Left }
  PadString or Str - Required
  FinalLength or Len - Required
  PadCharacter or Pad - Optional - set to if omitted
{ String - Pad Right }
  PadString or Str - Required
  FinalLength or Len - Required
  PadCharacter or Pad - Optional - set to if omitted
{ String - Parse }
  ParseString or Str - Required
  Delimiter or Dlm - Required
  StartToken or Tok - Optional - set to 0 if omitted
  NumTokens or Num - Optional - set to 0 if omitted
{ String - Proper }
  ProperString or Str - Required
{ String - Replicate }
  ReplicateString or Repl - Required
  ReplicateCount or Num - Required
  AvoidRegistry? or Reg - Optional - set to 0 if omitted
{ String - Reverse }
  ReverseString or Rev - Required
{ String - Search }
  SearchIn or In - Required
  SearchFor or For - Required
  StartPosition or Pos - Optional - set to 1 if omitted
  Occurrence or Occ - Optional - set to 1 if omitted
  Direction or Dir - Optional - set to F if omitted
  CaseSensitive? or Case - Optional - set to 0 if omitted

Developer Tools Category
{ Utility - Start Timer }
  DisplayFlag or Flag - Optional - set to 1 if omitted
Functions which cannot be called with \{ PGM Function \}

Here is an alphabetical list of the functions that cannot be called with \{ PGM Function \} because they contain no parameters.

**Date and Time Category**
- \{ DateTime - Parse Day from Today \}
- \{ DateTime - Parse Month from Today \}
- \{ DateTime - Parse Year from Today \}
- \{ DateTime - Reset Defaults \}
- \{ DateTime - Set Defaults \}
- \{ DateTime - Today to Character DOW \}
- \{ DateTime - Today to DOW \}
- \{ DateTime - Today to DOY \}
- \{ DateTime - Today to Julian \}

**File Operations Category**
- \{ File - Unique File Name \}

**Math and Science Category**
- \{ Math - Pi \}

**PGM System Category**
- \{ PGM \}
- \{ PGM Bug Report \}
- \{ PGM Function \}
- \{ PGM Function Error \}
- \{ PGM Help \}
- \{ PGM Registration \}
- \{ PGM Setup \}

**Program Operations Category**
- \{ Program - Clear Error Flag \}
- \{ Program - Log Event \}
- \{ Program - Reset \}
- \{ Program - Toggle Event Logging \}
- \{ Program - Toggle Exact Match \}
- \{ Program - Toggle Parameter Validation \}
- \{ Program - Validate Launch Values \}
- \{ Program - Validate Termination Values \}
- \{ Program - Validate Verification Values \}

**Developer Tools Category**
- \{ Utility - Get Timer \}
- \{ Utility - Macro Timer \}
Variable Management Category

{ Variables - Reset Current Level }
{ Variables - Reset Registry Values }
{ Variables - Restore }
{ Variables - Restore 25 }
{ Variables - Restore DT }
{ Variables - Restore SI }
{ Variables - Save }
{ Variables - Save 25 }
{ Variables - Save DT }
{ Variables - Save SI }
Index
(Program Operations Example E), 24, 111
(String - Occurs), vii, xi, 26, 38, 49, 50, 51, 52, 179, 180, 182, 185, 186, 187, 266, 272, 279, 281, 282, 286
(String - Pad), xi, 26, 38, 180, 185, 188, 189, 190, 191, 193, 266, 272, 273, 279, 281, 282, 286
(String - Pad Center), xii, 26, 38, 185, 190, 266, 272, 279, 281, 282, 286
(String - Pad Left), xii, 26, 38, 185, 191, 192, 266, 273, 279, 280, 281, 283, 286
(String - Pad Right), xii, 26, 38, 185, 193, 194, 266, 273, 279, 280, 281, 283, 286
(String - Parse), xii, 26, 38, 180, 185, 202, 203, 204, 205, 266, 273, 279, 280, 281, 283, 286
(String - Proper), xii, 26, 38, 185, 201, 266, 273, 279, 281, 283, 286
(String - Replicate), xii, 26, 38, 180, 185, 202, 203, 204, 205, 266, 273, 279, 280, 281, 283, 286
(String - Reverse), xii, 26, 38, 179, 182, 185, 205, 266, 273, 279, 280, 281, 283, 286
(String - Search), xii, 26, 38, 181, 183, 185, 206, 207, 208, 209, 266, 273, 279, 281, 283, 286
(String Example - Occurs), 26
(String Example - Pad), 26
(String Example - Pad Center), 26
(String Example - Pad Left), 26
(String Example - Pad Right), 26
(String Example - Parse), 26
(String Example - Proper), 26
(String Example - Replicate), 26
(String Example - Reverse), 26
(String Example - Search), 26
(Variable Management Example - Decimals), 27
(Variable Management Example - Integers), 27
(Variable Management Example A), 27, 70
(Variable Management Example B), 27, 71
(Variable Management Example C), 27, 72
(Variable Management Example D), 27, 73
(Variable Management Example E), 27, 73
(Variable Management Example F), 27, 73, 74
(Variables - Reset Current Level), ix, 27, 28, 79, 88, 89, 266, 268, 277, 288
(Variables - Reset Registry Values), ix, 27, 28, 79, 88, 89, 266, 268, 277, 288
(Variables - Restore), ix, 27, 28, 49, 72, 73, 74, 78, 79, 80, 81, 266, 267, 277, 288
(A)
AppClass or Func, 285
Applications, 64
AvoidRegistry? or Reg, 286
(C)
CaseSensitive? or Case, 286
(D)
Data, 76, 99, 131, 177, 213, 233
Date or Dt, 138, 140, 142, 147, 150, 152, 156, 158, 160, 162, 164, 284
Date1 or Dt, 144, 284
Date2 or Go, 144, 284
DayOfWeek or Dow, 153, 284
DayOfYear or Doy, 144, 284
DebugMode? or Debug, 285
Decimal or Dec, 227, 285
Degree or Deg, 225, 285
Delimiter or Dlm, 199, 286
Direction or Dir, 206, 286
DisplayFlag or Flag, 257, 286
Dividend or Div, 220, 285
Divisor or Dvs, 220, 285
ElapseTimeDisplay?, 257, 259
Event, viii, x, 46, 98, 99, 102, 107, 126, 127
EventLog, x, 107, 126
ExactMatch or Exact, 113, 116, 285
ExactMatch?, ix, 22, 23, 103, 112, 113, 116, 117, 124, 127, 265
FileName or File, 249, 285
FinalLength or Len, 188, 190, 191, 193, 286
Help, 58
Hexadecimal or Hex, 228, 230, 285
IncludeSubfolders? or Subf, 244
Integer or Int, 229, 285
LaunchDelay, ix, 22, 23, 104, 105, 111, 113, 114, 115, 120, 123, 127, 286
LaunchDelay or LDelay, 113, 286
LaunchState, ix, 22, 23, 104, 105, 109, 111, 113, 114, 115, 120, 123, 127, 285
LaunchState or WShow, 113, 285
LogEvent?, x, 106, 107, 112, 125, 126, 127, 134, 183, 215, 236, 265
LongName or Long, 247, 284
MaxLaunchDelay, ix, 104, 120, 127
MaxTerminateTimeout, x, 108, 122, 127
MaxVerifyDelay, x, 105, 121, 127
MaxVerifyLoops, ix, 105, 121, 127
MaxWait or Wait, 285
Month or Mm, 154, 284
MonthPosition, x, 135, 156, 157, 173, 174
Number or Num, 217, 218, 222, 223, 224, 285
NumDays or Go, 142, 284
NumMonths or Go, 140, 284
NumTokens or Num, 199, 286
Occurrence or Occ, 206, 286
Output, 240
OutputTo or Out, 244, 284
PadCharacter or Pad, 188, 190, 191, 193, 286
PadDirection or Dir, 188, 286
PadString or Str, 188, 190, 191, 193, 286
ParameterDecimal1, x, xii, 132, 134, 139, 146, 149, 151, 159, 161, 163, 165, 213, 214, 215, 217, 218, 220, 221, 222, 223, 224, 225, 226, 227
ParameterDecimal2, xii, 214, 220
ParameterDecimal3, xii, 214, 220
ParameterDecimal2, x, xi, 132, 148, 180, 181, 182, 184, 195, 198, 199, 200, 202, 203, 206, 207
ParameterDecimal3, x, xi, 181, 206, 207
ParameterString2, x, xi, 49, 133, 144, 179, 180, 181, 182, 184, 186, 188, 189, 190, 191, 193, 195, 196, 197, 198, 199, 200, 206, 207, 234, 235, 240, 241, 242, 243, 244, 245, 246, 247, 249, 251, 252
ParameterString3, xi, xii, 180, 188, 189, 206, 207, 234, 235, 240, 241, 242, 243, 244, 245
ParameterString4, xii, 235, 240, 241, 242, 243, 244, 245
ParamInt1 or PI1, 285
Index

ParamInt2 or PI2, 285
ParamInt3 or PI3, 285
ParamStr1 or PT1, 285
ParamStr2 or PT2, 285
ParamStr3 or PT3, 285
ParseString or Str, 199, 286
PathName or Path, 238, 247, 251, 284, 285
PgmSvrAppClass, ix, 64
PgmSvrCompleted, ix, 64, 65
PgmSvrDebugMode?, ix, 65
PgmSvrFunction, ix, 65
PgmSvrMaxWait, ix, 65
PgmSvrRetainClass?, ix, 65
PgmSvrSleepWait, ix, 65
PgmSvrUnloadWait, ix, 65
Power or Pwr, 221, 285
ProgFolder or PHome, 113, 285
ProgName or PName, 113, 116, 118, 285, 286
ProgramFolder, ix, 22, 23, 96, 102, 103, 113, 114, 127
ProgramName, ix, 22, 23, 96, 102, 103, 109, 110, 111, 113, 114, 116, 118, 127, 179, 213
ProgramParameters, ix, 22, 23, 103, 113, 114, 127
ProperString or Str, 201, 286
Radian or Rad, 226, 285
ReplicateCount or Num, 202, 286
ReplicateString or Repl, 202, 286
RetainClass? or Retain, 285
ReturnDecimal1, x, xi, xii, 21, 132, 133, 134, 138, 172, 213, 214, 215, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 228
ReturnInteger1, x, xi, xii, 133, 134, 144, 145, 146, 147, 149, 150, 155, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 171, 179, 181, 182, 184, 186, 187, 199, 200, 206, 207, 208, 214, 215, 230
ReturnInteger2, x, 182, 184, 199, 200
ReturnString1, x, xi, xii, 132, 134, 139, 140, 141, 142, 143, 148, 151, 152, 153, 154, 156, 157, 170, 179, 180, 182, 188, 189, 190, 191, 192, 193, 194, 201, 202, 203, 205, 214, 215, 227, 229, 234, 235, 236, 238, 239, 240, 244, 245, 247, 248, 257, 258, 259
ReturnString2, xiii, 235, 236, 246, 247, 248
ReturnString3, xiii, 235, 236, 246, 247
ReturnString4, xiii, 235, 236, 246, 247
ReverseString or Rev, 205, 286
SearchFor or For, 50, 186, 206, 286
SearchIn or In, 50, 186, 206, 286
ShowType or Show, 244, 284
SleepWait or Sleep, 285
SourceName or Srce, 244, 284
StartPosition or Pos, 206, 286
StartToken or Tok, 199, 286
SundayStartsOn, x, 135, 153, 173, 174
TargetName or Tgt, 244, 284
TerminateTimeout, x, 105, 108, 118, 122, 123, 127
TimeOut or Time, 286
UnloadWait or Unload, 285
ValidateParameters?, x, 22, 23, 105, 112, 113, 114, 116, 117, 118, 120, 121, 122, 123, 127, 265
ValidateParams or VParams, 113, 116, 118, 285, 286
VerifyLoops, ix, 22, 23, 100, 105, 113, 116, 118, 121, 123, 127, 285, 286
VerifyLoops or VLoops, 113, 116, 118, 285, 286
WindowTitle, ix, 22, 23, 96, 103, 109, 110, 111, 114, 116, 117, 124, 127, 179, 213
Year or Yr, 148, 155, 284