Honeywell

TotalPlant Solution (TPS) System

Display Authoring Tutorial

GU07211
R211
7/00

TotalPlant

Notices and Trademarks

Honeywell

Industrial Automation and Control

Automation College

16404 North Black Canyon Highway

Phoenix, AZ 85053

1-800 852-3211

About This Document

References

The following list identifies all documents that may be sources of reference for material discussed in this publication.

Document Title
Actor's Manual
Button Configuration Data Entry
Control Language/Application Module Reference Manual
Display Builder User's Guide
Display Scripting User's Guide

Contacts

World Wide Web

The following lists Honeywell's World Wide Web sites that will be of interest to our industrial automation and control customers.

Honeywell Organization	WWW Address (URL)
Corporate	*http://www.honeywell.com*
Industrial Automation and Control	*http://www.iac.honeywell.com*
International	*http://www.honeywell.com/Business/global.asp*

Telephone

Contact us by telephone at the numbers listed below.

	Organization	Phone Number	
United States and Canada	Honeywell Inc. Industrial Automation and Control	1-800-343-0228 1-800-525-7439 1-800-423-9883	Sales Service Tech. Support
Asia Pacific	Honeywell Asia Pacific Inc. Hong Kong	(852) 28298298	
Europe	Honeywell PACE Brussels, Belgium	[32-2] 728-2111	
Latin America	Honeywell Inc. Sunrise, Florida U.S.A.	(954) 845-2600	

Symbol Definitions

The following table lists those symbols used in this document to denote certain conditions.

Symbol	Definition
	**ATTENTION:** Identifies information that requires special consideration.
	**TIP:** Identifies advice or hints for the user, often in terms of performing a task.
	**REFERENCE -EXTERNAL:** Identifies an additional source of information outside of the bookset.
	**REFERENCE - INTERNAL:** Identifies an additional source of information within the bookset.
CAUTION	Indicates a situation which, if not avoided, may result in equipment or work (data) on the system being damaged or lost, or may result in the inability to properly operate the process.

Contents

Figures

Guidelines for Performant Displays

Introduction

The guidelines for building performant displays are written in support of the R210 version of the GUS Display Builder. No product anomalies have been addressed in this version of the manual.

Additional Information

References

The following list identifies all documents that may be sources of reference for material discussed in this publication.

REFERENCE - EXTERNAL

Actor's Manual

Button Configuration Data Entry

Control Language/Application Module Reference Manual

Display Builder User's Guide

Display Scripting User's Guide

Building a Display

Keep display density (graphics and data) to a "minimum"

This principle impacts human factors and has system implications. It is imperative to use as few LCN point.parameters as necessary in any display.

Display invocation time depends on the number of parameters being accessed in a display. Design your displays to provide the data necessary to get the job done efficiently with the least system impact.

Design displays utilizing a multiple-window workspace

Because SafeView allows you to configure the layout and behavior of multiple windows on a screen, design your displays with the mind-set that by using multiple windows individual displays can show smaller parts of the process or can be used to perform specific functions (such as, faceplate).

For example, don't build a huge display that shows the entire process and all the data. Instead, create an overview display that shows the whole process with as few parameters as possible. Then build separate displays that show smaller parts of the process or perform specific functions, with as few duplicate parameters as possible.

Use bitmaps judiciously in your displays

Bitmaps take longer to draw than other graphic objects. Use bitmaps judiciously in your displays. You may want to use bitmaps only in "show off" displays. Drawing bitmaps is CPU intensive, but as faster CPUs are available, the negative effect of putting bitmaps in displays will decrease.

Avoid putting animated objects over a bitmap

The entire bitmap is redrawn when any area of the bitmap needs to be redrawn. For this reason, avoid putting animated objects over a bitmap.

When displays start getting complex, name objects

It is a good practice to name objects. It makes script references more meaningful and makes scripts easier to maintain.

It is also possible to view the script on an object by selecting the named object from the pull-down list from any script window. To identify objects having scripts, one application group began the name of objects having scripts with "sc."

Running Displays

Hardware configuration contributes to performance

As in any computing environment, performance is directly related to computing resources such as

- memory
- processor speed
- disk size

Obtain as many computing resources as you can.

Keep PC focused on GUS

Keep the machine focused on GUS as much as possible. Remove unnecessary network connections, background tasks, etc. GUS displays run faster with fewer third-party applications running.

ATTENTION

The first set of display callups takes slightly longer than subsequent display callups.

On PC startup, the Display Manager is invoked, and one or four gpb processes (four if you have Multiple Displays) are started. The initial running of displays using these gpb processes takes longer than subsequent reuse of the gpb processes.

For fastest callup, place display files on a local drive

Display files (.pct files) are fairly large. If you are calling them up over a slow network, this will add significant time. For fastest callup, place display files on a local drive.

For faster callup, use “InvokeDisplay( )” to invoke one display from another one

For faster callup, invoke a display from another display by using the script function “InvokeDisplay( )” rather than invoking the display using the “runpic” command from the command line.

ATTENTION

The SCHEM actor is as performant as “InvokeDisplay( ).”

Accessing Data in Scripts

Use public variables or display parameters instead of DDB items or LCN points

Accessing data stored in public variables or display parameters is faster than accessing data stored in the DDB or LCN. For this reason, use public variables or display parameters whenever possible when scripting a display.

Constraint

If a display is scripted to invoke another display, public variables from the invoking display are not copied to the invoked display; however, global DDB values of the invoking display are copied to the global DDB of the invoked display. Public variables do not take the place of the global DDB.

Considerations when writing scripts using public variables and display parameters:

- You cannot use public variables to store values of variable and entity data types; use the DDB or display parameters to store variable and entity data.
- A change to a public variable will not cause an OnDataChange event, whereas a change to a display parameter will cause an OnDataChange event.
- Public variables cannot be used in a variable expression on a basic object dynamic (rotate, fill, bar, and text value). Display parameters can.

Use the DDB instead of the LCN for temporary storage or passing of data

Accessing data from the DDB is faster than accessing data from the LCN. Avoid using LCN point.parameters for temporary storage of data. Use the DDB for temporary storage of data and/or passing data to a display invoked from another display.

ATTENTION

You must use LCN point.parameters to pass data among running displays.

Use an error handler instead of doing LCN reads for errors

Consider using an error handler instead of doing LCN point.parameter.status reads for handling errors. This especially applies to non-OnDataChange scripts since LCN and DDB reads are immediate in non-OnDataChange scripts.

The BasicScript language element, OnError, allows you to define what action is to be taken when a trappable runtime error occurs.

Use an error handler when accessing multiple servers with OnDataChange

If you include an expression in your OnDataChange script that references data from more than one server, the script will execute as soon as data is returned from the first server. At this time, the variant data type for variables from other servers may still be "unknown," and this will trigger a Type Mismatch error report. You should, therefore, always include an explicit error handler in your script, to prevent displaying these Type Mismatch error messages during normal startup.

Use OnDataChange script with added logic for values you read only once

If some values need to be read once, still use OnDataChange script, and put some logic around those variables so that the code is executed only once. Here is an example:

Sample Script

```
Public once as Boolean
Sub OnDataChange()
   If not once then
      <read parameters and assign to public variables>
      once = TRUE
   end if
   <other script instructions>
End Sub
```

The Boolean “once” will be false when this script is executed for the first time, then be set to true for the duration of the display execution.

Ensure that all dynamic indices are initialized at display startup

Initialize the DDB parameter or scripting variable being used as a dynamic index. Failure to do so results in the default value of the parameter or variable being used as the dynamic index value. This can either cause GUS Runtime to read from or write to the wrong element of an array, or cause a runtime error if the indexed member is not a legal member of the array. Here is an example:

Sample Script

```
Sub OnDataChange()
   me.text = lcn.am_1349(dispdb.int01).[name]
```

```
End Sub
```

Dispdb.int01 was not initialized during display startup, therefore, the dispdb.int01 was equal to its default value of zero. This script raised a Configuration Error since lcn.am_1349(0) was not a legal member of the array.

Be aware that using "SendKeys" can "freeze" your PC

Using the BasicScript language method "SendKeys" in a GUS script will "freeze" your PC for a time when the object in focus is incapable of handling the transmitted keystrokes.

You can use the "SendKeys" method as a way of simulating keyboard input. Therefore, the object in focus must be capable of processing the sent keystrokes to execute the BasicScript language method "SendKeys" without causing an error. If the object in focus cannot process keystrokes, each keystroke sent in the parameter of the "SendKeys" method is handled as a separate error, and this error is "announced" to the user as an audible beep. While the beeps are occurring serially, the PC "freezes." To abort this frozen state, press <Ctrl><Esc>.

Ensure that writing to a display/DDB param doesn't initiate unnecessary actions

In a script, writing to a display parameter or DDB parameter of all data types except entity and variable will cause an OnDataChange event to be sent immediately to all objects that reference the changed parameter in their OnDataChange scripts. The unnecessary execution of OnDataChange scripts reduces performance.

TIP

When a script statement writes to a display parameter or DDB parameter, GUS Runtime does not compare the old value to the new value to determine if the value really changed. It simply marks the parameter as changed and sends out the OnDataChange events to all appropriate objects.

When scripting, declare only one variable per line

When scripting, declare each variable on a separate line. Here is an example:

Sample Script

```
dim LoAllmColor as long
dim HiAlmColor as long
```

Do not declare variables as follows:

Sample Script

```
dim LoAlmColor, HiAlmColor as long
```

Declaring multiple variables on a single line causes erroneous behavior at runtime.

Using the Data Types Entity and Variable in Scripts

How to script indirect references

Read the section "DispDB Indirection" in the *Display Scripting User's Guide* for details on how to script indirect references. This section describes how to script Display Data Base (DISPDB) indirection for variable and entity types. It points out which syntaxes are bound at buildtime and which syntaxes will give better performance at runtime. This section also provides examples that will help you to write scripts that include indirect references.

Use "Enter Parameters" to set entity and variable display parameters at buildtime

Display parameters of type entity and type variable can be set only at buildtime, using the "Enter Parameters" dialog. You cannot set the reference of a display parameter of type entity or variable in a script. If you script "`set display.params.tag = GetEnt("A100")`" where tag is of type entity, you will get an error.

To enter a reference for a display parameter of type entity or variable in the "Enter Parameters" dialog, enter an LCN reference, a DDB reference, or a reference to another display parameter of the same type.

In OnDataChange scripts use "dispdb.<entity>.[name]" to reference a DDB entity

Referencing the name of the entity assigned to a $cz_enty using the scripting syntax "dispdb.[$cz_enty].[name]" is more performant than using the scripting syntax "dispdb.[$cz_enty]" or "dispdb.[$cz_enty].external" in OnDataChange scripts.

Using the scripting syntax "dispdb.[$cz_enty].[name]" in a script statement sets up an entry in the GPB cache and reads the value from cache; whereas referencing the name using "dispdb.<entity>.external" or "dispdb.<entity>" causes an immediate read to the LCN in an OnDataChange script.

ATTENTION

Either the syntax, "`me.text = dispdb.ent01.[name]`" or "`me.text = dispdb.ent01.external`" will cause an immediate read in an OnLeftButtonClick script.

> Note also that even though the above example uses dispdb.[$cz_enty], this performance tip applies for all DDB entities.

Example: Using Performant Syntax

Scenario

In displays using the change zone, you may want to write a conditional statement in an OnDataChange script to determine if there is an entity assigned to dispdb.[$cz_enty], and then script some actions if dispdb.[$cz_enty] is assigned to an entity.

Using "`dispdb.[$cz_enty].[name] <> ""`" will give a configuration error (error number 1052) if there is no entity assigned to dispdb.[$cz_enty]. Here are two scripting options:

Option 1

Instead of scripting as follows:

Sample Script

```
if (dispdb.[$cz_enty].[name]<>"" then,
```

script like this:

Sample Script

```
if dispdb.[$cs_enty].[name].status = HOPC_NO_ERROR
then
   <script actions when $cz_enty is assigned an entity>
else
   <script actions when $cz_enty is NULL>
end if
```

Option 2

Use "`if dispdb.[$cz_enty].[name] <> ""`" in the script and put an error handler in the script to handle the configuration error (1052). Script the action to be taken when the configuration error 1052 occurs (i.e., when the entity is null) in the Error Handler.

Use GetEnt("") to set a Display Data Base entity or variable to NULL

To set a Display Data Base Entity to NULL, use the syntax, "`set dispdb.<entity> = GetEnt("")`".

To set a Display Data Base Variable to NULL, use the syntax "`set dispdb.<variable> = GetVar("")`".

Here is an example:

Sample Script

```
Sub OnLButtonClick()
   set dispdb.[$cz_enty] = GetEnt("")
End Sub
```

In OnDataChange Scripts check for NULL with HOPC_CONFIG..N_ERROR

To determine if an entity in an OnDataChange script is NULL, check if "`dispdb.<entity>.[name]".status = HOPC_CONFIGURATION_ERROR`". This is the most performant way. Here is an example:

Sample Script

```
Sub OnDataChange()
   If dispdb.[$cz_enty].[name].status = HOPC_CONFIGURATION_ERROR
then
      'dispdb.ent01 is NULL
      'script actions when ent01 is NULL
   end if
End Sub
```

How to set a Display Data Base entity = a display param of type entity

To set a Display Data Base entity equal to a display parameter of type entity, use the syntax, "`set dispdb.<entity> = display.params.tag`" where tag is of type entity.

Here is an example:

Sample Script

```
Sub OnLButtonClick()
   set dispdb.ent01g = display.params.tag
   'tag is a display parameter of type entity
End Sub
```

CAUTION	The .external property does not work with the display parameter of type entity.

Examples of scripting that does not work are as follows:

Sample Script that DOES NOT WORK

```
dispdb.[$cz_enty].external = display.params.tag.external
```

and

```
set dispdb.[$cz_enty].external = display.params.tag
```

An example of script that will work is as follows:

Sample Script that DOES WORK

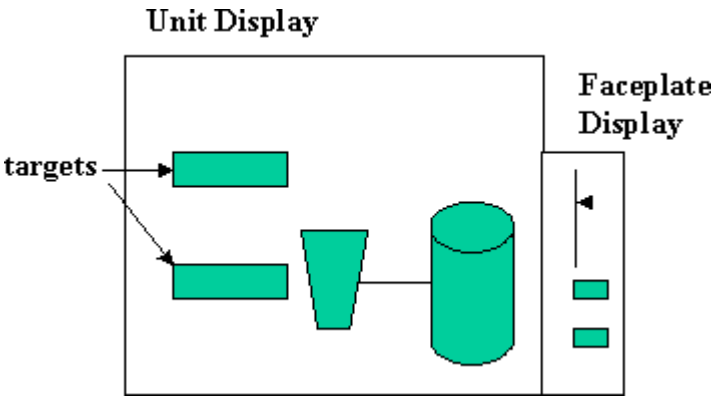
```
dispdb.[$cz_enty].external = display.params.tag.[name]
```

To change an NT registry entity setting, save the internal ID not the entity name

To save an entity to an NT registry setting for the purpose of setting a different entity to this saved setting, save the internal ID (for example, dispdb.ent01.internal) rather than the name of the entity (for example, A100).
Setting the new entity to the saved registry setting containing the internal ID of the entity is more performant.

Example

Consider two displays, a Unit display and a Faceplate display. Selecting a target on the unit display changes the point-of-interest in the Faceplate display. **Unit-Faceplate Displays**



Sample Script (on Target)

```
Sub OnLButtonUp()
   ’indicate if the point-of-’interest has changed
   dim mycount as long
   'set the entity to the 'target’s point
   set dispdb.ent01 = getent("ms_hist1")
   mycount = GetSetting("MyApp", "pointarea", "counter")
   if mycount < 65535 then
   'incr. the counter to indicate a new point-of-interest
      mycount = mycount + 1
   else
      mycount = 0
   end if
   'Save internal ID of the point-of-interest to the registry
   SaveSetting "MyApp", "pointarea", "pointid",
   dispdb.ent01.internal
   'Save the indicator to the registry
   SaveSetting "MyApp", "pointarea", "counter", mycount
End Sub
```

Sample Script (on Faceplate Display)

```
dim mycount as long
Sub OnPeriodicUpdate()
   dim newcount as long
   dim stringid as string
   newcount = GetSetting("MyApp", "pointarea", "counter")
   'is there a new point-of-interest?
   if newcount <> mycount then
      mycount = newcount
   'get the internal ID from the registry
      stringid = GetSetting("MyApp”, "pointarea", "pointid")
      dispdb.[$cz_enty].internal = stringid
   end if
End Sub
```

Scripting Events Properly

OnDataChange

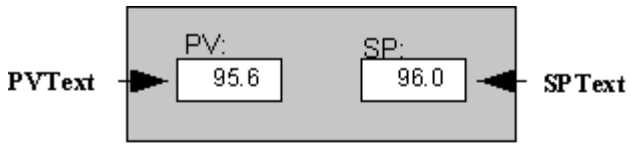
All LCN and DDB reads are from cache (that is, stored, scanned values). All writes are immediate to the LCN and DDB.

Referencing display parameters causes infinite loops

Be aware that you can script an infinite loop when referencing display parameters.

For example, consider a display with two Text objects, PVText and SPText. Each of these Text objects has an OnDataChange script. The OnDataChange script of PVText references display parameters P1 and P2. The OnDataChange script of SPText references display parameters P1 and P3. See the example scripts in the following figure.

Infinite Loop Scripting



Script for PVText

```
Sub OnDataChange
   me.text = lcn.A100.PV
  display.params.P1 = display.params.P1 + display.params.P2
end Sub
```

Script for SPText

```
Sub OnDataChange
   me.text = lcn.A100.SP
  display.params.P2 = display.params.P1 + display.params.P3
end Sub
```

Infinite Loop Scenario

Refer to the previous figure.

Time T1:

The point.parameter lcn.A100.PV changes, causing an OnDataChange event to be sent to the PVText object. The OnDataChange script on the PVText object executes. The PVText's OnDataChange script changes the display parameter P1. Because the OnDataChange script of the SPText object references the display parameter P1, SPText is immediately sent an OnDataChange event.

Time T2:

The OnDataChange script on the SPText object executes in response to the OnDataChange event sent as a result of P1 changing. The SPText's OnDataChange script changes the display parameter P2. Because the OnDataChange script of the PVText object references the display parameter P2, PVText is immediately sent an OnDataChange event. **YOU ARE NOW IN AN INFINITE LOOP**. This OnDataChange "Ping-Pongs" between the PVText and SPText objects.

Guidelines for scripting performant OnDataChange scripts

- Do not solicit user input. This holds up execution of the OnDataChange scripts queued to run on the OnDataChange thread. Solicit user input with a user event such as LButtonClick.
- Avoid writes to the LCN and DDB.
- Use BasicScript variables (private or public) or display parameters instead of LCN parameters or DDB as often as possible. Be careful you do not script infinite loops when using display parameters.
- Do not use a public or private variable as an index value in an OnDataChange script. A change to a public or private variable will not cause an OnDataChange event.
- Do not use the sleep statement.
- Avoid repetitive processing based on a condition. In other words, avoid using the DO Loop" statement or the "While Wend" statement.

OnPeriodicUpdate

All LCN and DDB data reads are from cache, which means they are stored, scanned values. All writes are immediate to the LCN and DDB.

Guidelines for scripting performant OnPeriodicUpdate scripts

- Create an error handler to handle the bad status code "DAS_NO_ERROR_BUT_NO_DATA" that can appear if the LCN data requested does not reach the cache before the cache is read. This can occur because initial data collection requests and OnPeriodicUpdate scripts are started at the same time.
- Avoid writes to the LCN or DDB. Use BasicScript variables (private or public) or display parameters whenever possible.
- Do not solicit user input.
- Do not use a public or local variable as an index value in an OnPeriodicUpdate script. There will be a delay in showing the correct value.
- Do not use the sleep statement.
- Avoid repetitive processing based on a condition. In other words, avoid using the "Do...Loop" statement or the "While...Wend" statement.
- Use OnPeriodicUpdate scripts to do simple animation only.

Use a counter to redefine the "period" of an OnPeriodicUpdate

To redefine the "period" of an OnPeriodicUpdate script, use a counter in the OnPeriodicUpdate script. Do not use a sleep statement to redefine the "period."

It is not recommended that you use a sleep statement in an OnPeriodicUpdate script because an OnPeriodicUpdate script with a sleep statement can slow down the shutdown of a display since the shutdown has to wait until the sleep has terminated.

Here is an example of how to use a counter:

Sample Script

```
Const Period = 10
Dim Cycles as integer
’at startup, "cycles" is automatically initialized to 0
Sub OnPeriodicUpdate()
   if cycles >= Period then
      cycles = 0
      ’put code here to run at the end of each period
   else
      ’period has not yet been reached
      cycles = cycles + 1
   end if
End Sub
```

User events, such as OnLButtonClick

All LCN and DDB reads and writes are immediate.

Guidelines for scripting performant User Event scripts such as OnLButtonClick

- Use BasicScript variables (private or public) as much as possible.
- Do not use the sleep statement.
- Recognize that iterative processing (“For...Next” statement, “For...Each” statement) and repetitive processing based on a condition (“Do...Loop” statement, the “While...Wend” statement) may take time. Following style guide guidelines for a long process, indicate to the user that processing is being done by using a progress indicator.

SeeMsg.Thermometer property in BasicScript for more details.

Recognize that using a dialog box for user input stops all user event processing for that display. This is normal behavior when using a dialog box because a dialog box is modal. In place of a dialog box, consider requesting user input by
(1) using the Honeywell data entry OLE control,
(2) using an embedded display (you may want to show and hide the embedded display based on user action), or
(3) invoking another window (utilize a SafeView workspace).

OnDisplayStartUp

All LCN and DDB reads and writes are immediate.

Guidelines for scripting performant OnDisplayStartUp scripts

- Do not duplicate code that appears in an OnDataChange script. At display startup, a scan of all variables occurs resulting in an OnDataChange event being sent to all objects having OnDataChange scripts. Therefore, the OnDataChange scripts can handle the initial appearance and behavior of the display and its objects except for initializing variables and entities referenced in the display.
- Minimize reads and writes to the LCN and DDB. Preferably, limit data access to initializing display DDB items.
- Do not solicit user input. OnDisplayStartUp processing will be stopped until the input dialog box is closed.
- Use BasicScript variables (private or public) or display parameters as much as possible.
- Do not use the sleep statement.
- Avoid repetitive processing based on a condition. In other words, avoid using the "Do...Loop" statement or the "While...Wend" statement.

OnDisplayShutDown

All LCN and DDB reads and writes are immediate.

Guidelines for scripting performance OnDisplayShutDown scripts

- Be aware that a dialog box invoked in an OnDisplayShutDown script may be covered up by the SafeView placeholder window displayed as a result of closing the display. Because the user is not aware of the hidden dialog, the shutdown script may be stopped.

Cautions on the use of Visual Basic Functions

Visual Basic functions such as the InputBox or AskBox are based on standard Visual Basic functionality. They do not handle NANs, and error handling is minimal.

The InputBox and AskBox are not good choices for process-related entries. For example, unless you add appropriate script, selecting the CANCEL and CLOSE buttons on an AskBox will return a "0" for a numeric entry or a "null" string for a text entry. A similar situation occurs with the InputBox. This could cause a runtime error that closes your display, or even worse, sends a bad value to the process that could cause a problem.

Scripting Animation in a GUS Display

Minimize the use of animation in a GUS display

Animation is achieved by scripting the OnPeriodicUpdate event, which is fired every 0.5 second. Therefore, the OnPeriodicEvent script on each scripted object executes every 0.5 seconds. Having a lot of animation in a display could cause high CPU usage.

When testing a display with animation, monitor the CPU usage of the display

You can monitor the CPU usage of a running display by viewing the Performance page of the Windows NT Task Manager. Because GUS displays run in a multi-window environment, monitoring CPU usage is a necessary task when testing a display. A single display should not use all the available CPU; the performance of the other running displays decreases as a result.

Consider the “frame” for animation instead of “translation-and-rotation of objects”

Using the “frame” approach to animation uses less CPU than the “translation-and-rotation of objects” approach does.

The “frame” approach (using a different drawing in each “frame” to show action) is what movie animators do. Draw different “views” of the object to be animated. In the object’s OnPeriodicUpdate script, changing the visibility (“on” or “off”) of the various “views” gives the appearance of animation to the user of the display.

Rotation of an object is achieved by changing the Angle property of an object. Movement of an object is achieved by changing the TransX and TransY properties of an object. Scripting these properties to achieve animation utilizes more CPU than the “frame” approach does; therefore, minimize the use of these properties to achieve animation in a GUS display.

Using the Change Zone

Ensure that the CZE group is configured correctly

There are two interdependent configuration actions:

- Keep the same collection rate for all variables in the CZE group. When the collection rates in the CZE group differ, erroneous behavior of the Change Zone can occur when a new point-of-interest is displayed.
- Include all the dispdb.[$cz_enty] data references in the CZE group.

Consider deleting unnecessary embedded displays in the Change Zone

Delete embedded displays in the Change Zone that handle point types that are not configured in your TPS system. This will improve display invocation time.

Embedded Display	Point Types Handled
countersub	COUNTAM, COUNTHG
devctlsub	DEVCTL
digcmpnimsub	DICMPNIM, DIOUTNIM, DIINNIM
digcmphgsub	DIGCMPHG, DIGOUTHG, DIGINHG
flagsub	FLAGNIM, FLAGHG, FLAGAM
logicblk	LGBLKHG
procmodsum	PRMODNIM, PRCMODHG
regsub	REGCLNIM, REGAM, REGHG
switchamsub	SWITCHAM
timersub	TIMERNIM, TIMERHG, TIMERAM, REGPVNIM when pvalgid = “TOTALIZER”, REGAM when pvalgid = “TOTALIZER”, DIINNIM when ditype = “ACCUM”

Consider a Display Data Base item, $cz_enty in a custom Change Zone

To improve the performance when invoking the standard change zone for a point or changing the point of interest in the change zone, enhancements have been made in the GUS Application, gpb.exe. These enhancements are concerned with the Display Data Base (DDB) item, $cz_enty. You can utilize dispdb.[$cz_enty] in scripting a custom change zone to take advantage of these enhancements. Following is the list of enhancements.

- When a display having reference(s) to dispdb.[$cz_enty] is validated, the dispdb.[$cz_enty] variables (e.g., dispdb.[$cz_enty].pv) are put in the data collection group named “CZE” and the collection rate of each reference is set to 4 seconds.

ATTENTION

When a previously validated display is validated again, the data collection group assigned to dispdb.[$cz_enty] variables is not changed to the “CZE” group. You may change the group using the Data Collection dialog invoked from “Display/Data Collection” menu item in the GUS Display Builder.Use this admonishment to identify information that requires special consideration. DO NOT use this admonishment to identify information that can be classified as a "tip" or other "nice-to-know" material.

- Assignments to dispdb.[$cz_enty] (e.g., `dispdb.[$cz_enty] = “A100”` or `set dispdb.[$cz_enty] = lcn.A100`) in all scripts (including User Event scripts) are bound at buildtime.
- When the point assigned to dispdb.[$cz_enty] is changed during the running of a display, all variables in the CZE group are updated immediately.

Referencing Data in Subroutines and Functions

Reference subroutine and function data from OnDataChange scripts carefully

During validation, data references in all subroutines and functions are not put in the data collection list to be scanned, and therefore are not part of the GPB cache. This behavior has no effect on the performance or behavior of User Event (e.g., OnLButtonClick), OnDisplayShutDown, or OnDisplayStartUp scripts because all reads are immediate. It could, however, have an effect on the performance and behavior of the OnDataChange and OnPeriodicUpdate scripts.

Consider the following when writing an OnDataChange script:

- A value change to a data reference in a subroutine or function that is invoked from an OnDataChange script will not cause an OnDataChange event to be fired, and therefore the data reference's value will not update in the display.
- If an OnDataChange script is triggered, the subroutines called from within the OnDataChange script will be executed and data will be collected. BUT there may be a performance hit. A subroutine called from an OnDataChange script first checks the GPB cache for the referenced data item. If the LCN or DDB value is referenced in some other object's OnDataChange or OnPeriodicUpdate script, then it will be in the GPB cache and appear in the data collection list. The value will be read from the stored value, BUT if it is not in the GPB cache, then an immediate read of that value will occur. This will decrease performance.

The same considerations apply when writing OnPeriodicUpdate scripts.

Consider using a public script variable when reading same LCN vrbl repeatedly

Consider using a public script variable to store the value of the variable to read the same LCN variable repeatedly within a collection of subroutines. Remember that "reads" in subroutines are immediate so the value must be read from the process network. Reading a public variable into which you have stored the LCN variable is faster.

The same considerations apply when writing OnPeriodicUpdate scripts.

Fine Tuning Performance

Consider the following methodology as an approach to fine tuning performance

Following are the steps that we have used to review displays with respect to performance fine-tuning:

Step	Action
1	Open the display in the GUS Display Builder.
2	View the Collection List. Take note of the number of variables being collected and the collection rate of each variable. Determine if the collection rate is valid for each variable. Collect a variable at the slowest rate possible. If a variable is known to be static, make its collection rate zero. Then the value of the variable will only be collected at startup. This will not improve performance at invocation time, but it will reduce the load on the LCN and HOPC Server while the display is running.
3	Take note of the number of embedded displays in the main display and the number of instances for each embedded display. The easiest way to get this information is to invoke the “Replace Embedded Display” dialog. All the information is provided in this dialog. Then simply cancel the dialog.
4	Repeat steps 1, 2, and 3 for each schematic.
5	From the data gathered, determine which embedded displays are used the most and review them for possible performance enhancements.

Embedded Displays

“Conserve” objects within frequently used embedded displays

Using the fill and line properties of a Text object, it is sometimes possible to remove extra rectangles from an embedded display. This change is minor in the embedded display; however, if a display uses that embedded display repeatedly, the total number of objects is reduced significantly, which can improve performance.

Consider using a data type object display parameter to read or write data

Using the object parameter to write directly to another embedded display’s parameters is fast. It causes the immediate firing of the OnDataChange scripts when a display parameter changes.

Data can also be read and/or written between **embedded displays** using the Display Data Base. This is a less performant method because writing data to a DDB item means an out-of-process write to the DDB that is located in the HOPC Server.

Example

"Controller" Embedded Display

The display author enters the name of the object to be controlled in the display parameter "controllee." In this example, the controllee's name is "Box1." Then, the Controller can write directly to the controllee's display parameter "color" on mouse clicks during runtime.

Controller Writing to Controllee Scripting

Display Parameter	Type	Value	
controlee	object	Box1	Note: This value is written at buildtime.

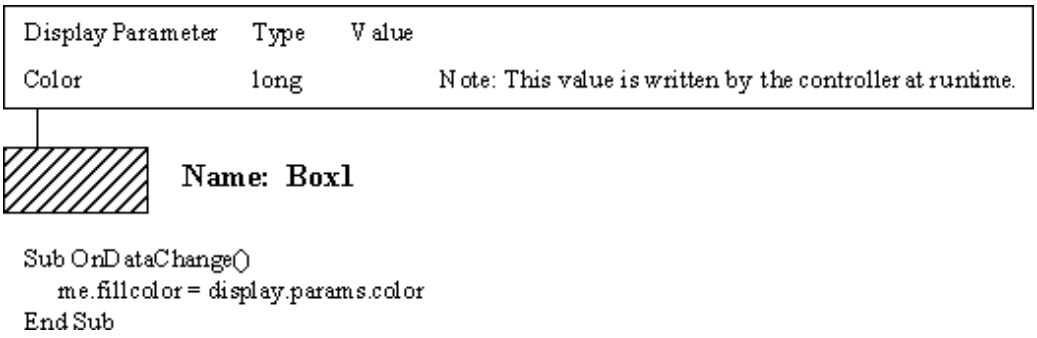
```
OnLButtonClick()
    display.params.controlee.params.color = TDC_CYAN
End Sub

OnRButtonClick()
    display.params.controlee.params.color = makecolor (128,128,128)
End Sub
```

"Controllee" Embedded Display

When the Controller writes to the Controllee's display parameter "color," this immediately fires the OnDataChange event on the "Controllee."

Triggering an OnDataChange Event



Consider using an embedded display as a target/color manager

Embedded displays using the object parameter can be used as a target manager and/or a color manager, which the other embedded displays "know about," and can therefore, read from and write to.

Optimizing GUS Data Collection Groups

Optimizing data collection originated by a GUS node includes minimizing the number of data request messages generated and producing data collection requests that can be processed efficiently by data owners on the LCN and UCN. The goal in is to have performant GUS displays and at same time avoid any unnecessary loading on the LCN/UCN.

Data can be classified by type (String, real, etc), update frequency, location. These "kinds" of data inherently affect the performance of the system. For example, some devices on the UCN return data more quickly then others. Similarly, some types of data can be retrieved faster then others. And, of course, update frequencies relate directly to network loading.

Grouping data correctly using GUS Data Collection Groups will optimize the retrieval of information. The following guidelines and suggested Group layout are presented to help achieve this.

Basic rules for grouping data

Step	Action
1	Certain kinds of data should reside in their own Collection group to localize the effect of gathering that kind of data. For instance, DispDb data can ultimately point to various data owners on the LCN/UCN. Putting these data items into their own group minimizes their effect on other Data Collection groups. This is also true for Collector data. Collector and DispDB data can be combined into one group. The recommended update frequency for this group is 4 seconds or slower. For best system performance, use the slowest collection rate that meets your system requirements.
2	Put non-updating parameters (such as Point and Engineering Unit descriptions, and Alarm limits) into a separate group with an update frequency of 0 (zero). This will cause the data to be retrieved only upon Display startup. The data can be retrieved on demand as well. Do ***not*** use Group 0 for non-updating parameters because the LCNUPDATE 0 command will update ALL groups, not just group 0.
3	Do not modify the update frequency for changezone parameters assigned to the special collection group CZE. The default update frequency for this group is 4 seconds and it should not be changed.
4	For point.parameter references from LCN nodes (data owners) such as AMs, HGs, CGs and NIMs: 1. Determine the LCN node number where the point.parameter resides (this is listed in each point detail display in the parameter NODE_NO). 2. Group parameters by LCN node (or redundant pair). That is, put references associated with each LCN node into a separate Collection groups. Note that the reason for grouping by “node” is that some nodes return data more slowly than others do. For example, if a Data Collection group contains references to both an AM and a CG, the message to the LCN that contains both requests will be delayed until both pieces of data are returned. Because a CG may return data more slowly than an AM node, the effect would be to delay the acquisition of the AM data. 3. In order to accommodate a number of different scan frequencies, do not to increase the total number of scan groups to the point where communication inefficiencies result. The table below provides a Collection Group layout that recombines points and reduces the total number of groups. 4. If there are more then say 20 points that do not change often, put them either in a very slow collection group (for example, 60 seconds), or in a collection group with a zero rate, which means that the points will only be updated on demand. These parameters cannot include alarm limits and point execution status.

Group	Rate (seconds)	Membership
1	4	DispDB and Collectors e.g. ACKSTAT, POINTSTS, POINTCNT, HISTORY, etc
2	0	All non-updating parameters
CZE	4	CZE (Changezone)
3	n	LCNnode xx
4	n	LCNnode yy
5	2n	LCNnode xx
6	2n	LCNnode yy
FST	4	FAST Group

If certain parameters need to be scanned at a faster rate, you have two options:

1. Create a group with ID set to FST. This special group is called the “FAST” group. It has a default scan frequency of 4 seconds. However, when the FAST button on an IKB is pressed, that group will begin to scan at a 1-second rate.

2. Create a separate group for points that need to be scanned more often.

CAUTION	You must use this group judiciously, because scheduling a large number of points for a fast update rate can adversely affect system performance.

Optimizing GUS Data Collection Groups on a UCN

Grouping data access requests by processor type helps to optimize the UCN data gathering efficiency of a custom display. Optimum grouping of points on a particular UCN includes only points belonging to the same type of processor. A typical UCN box (PM, APM, HPM, SMM, LM) has two processor types, Control Processor and Input/Output Processor (IOP).

Control processor point types include the following:

Reg CTL	Reg PV	Digital Composite
Logic	Process Module	Device Control
Array	Flag	Numeric

IOP point types include the following:

Digital Input Digital Output Analog Input Analog Output

In R530, this grouping can be done automatically for schematics built on a Universal Station (US) with the Picture Editor (PE), using the PE OPTIMIZE command. For a GUS display, the grouping is done manually.

The basic rules for grouping UCN data

Using the update rate considerations discussed previously, continue to set up the collection groups as follows:

- Create Demand Scan groups when possible. They do not impose a steady state load on the network or the GUS station.
- For UCN boxes, split groups that have update rates by processor type, control, or IOL.
- Reference points owned by the Control Processor preferentially. Typically, IOP points feed their values to a Regulatory Point in the Control Processor. For example, the PV of an AI Point (AI.PV) and the PV of an AO Point (AO.PV) are typically connected to the PV and OP of a Regulatory Point. Making references to the PV and OP of the Regulatory Point (in the Control Processor) is more performant than making references to the AI and AO Points.

Example

You are creating a display that requests both Control and IOL data from multiple APMs. Some of the data is not updated (descriptors, point names, etc.).

You determine that the best overall system loading is achieved by using two update rates (for example, the standard 4-second update and an 8-second update).

If there are more than 10 parameters in either the xPM control or IOP group that do not change often, put them either in a very slow collection group (such as 60 seconds), or in a collection group with a zero rate. This means that parameters will only be updated on demand. These parameters cannot include alarm limits and point execution status.

Because the numbering of groups is arbitrary within the 0-245 limit allowed by the Display Builder, you could establish the following groups:

Group	Rate (seconds)	Membership
10	4	Control parameters collected every 4 sec
11	4	IOL parameters collected every 4 sec
20	8	Control parameters collected every 8 sec
21	8	IOL parameters collected every 8 sec
30	60	Slowly changing parameters

When the NIM constructs UCN parameter requests, it sorts each group's parameters by UCN node and each APM receives a homogeneous request.

Sample Data Collection Group Layout

If we combine the considerations of the previous 2 sections, we can envision a Data Collection Group layout as shown below.

CAUTION Note that there are over 10 Collection Groups defined. Because of messaging overheads, this may result in a non-performant display. It is anticipated that most displays will NOT require all the types and numbers of Collection Groups that are shown below. In fact, efforts need to be made to minimize both the number of data Collection Groups and the number of "features" built into any one display.

Group	Rate (seconds)	Membership
1	4	DispDB and Collectors e.g. ACKSTAT, POINTSTS, POINTCNT, HISTORY, etc
2	0	All non-updating parameters
CZE	4	CZE (Changezone)
3	n	LCNnode xx
4	n	LCNnode yy
5	2n	LCNnode xx
6	2n	LCNnode yy
10	4	UCN Control parameters collected every 4 sec
11	4	UCN IOL parameters collected every 4 sec
20	8	UCN Control parameters collected every 8 sec
21	8	UCN IOL parameters collected every 8 sec
30	60	Slowly changing parameters
FST	4	FAST Group

Additional Optimization and Tuning

Sizing Up the Display1

Performing the following calculations **for each display** will give you an idea of how performant your system will be.

Step	Action
1	Set Total = 0.
2	Add 0.8 to Total for each Alarm Collector type.
3	If you have history collectors, add 1 to Total. Refer to the TPN Network documentation for a list of Collector types.
4	Add 1 to Total if your have dispdb.entXX references.
5	Add 1 to Total if the display contains a Changezone.
6	Add 1 to Total for each Trace of a Trend object.
7	Add 1 to Total if you use indirection.
8	Add 1 to Total for each Collection group defined in accordance with Sections 1.13 and 1.14.
9	Add 1 to Total if there are more than 300 LCN data references.

If Total is >= 10, your display call up performance may be slow.

It is evident that it is not at all difficult to end up with 10 or more Collection groups and "features" (Collectors, Trends, etc). This will produce a non-performant picture. You must be judicious in selecting the features, the number of data references, and their grouping.

Simplifying the Display

You can simplify the display by performing the following steps:

Step	Action
1	Reducing the number of Collector types, or moving some or all of the alarm and history Collectors out into a second display, if possible.
2	Moving the Trends to a separate display.
3	Moving the history Collectors to a different display.
4	Reducing the number of LCN data owners referenced in the display.
5	Replacing dispdb.ent01 references and indirection references with fixed parameter access, such as lcn.pointname.pv.
6	Reducing the number of Collection groups.
7	Reducing the number of data references in each display.

Sizing Up the Display

Try to recombine points to reduce the number of Collection groups. For instance, if you have two groups referencing the same UCN box and they have been set to scan frequencies that differ by a factor of two (say, 4 sec and 8 sec), it may be more efficient to recombine the points into the Collection group with the faster rate. A method for determining when to do this follows. Consider the following inequality:

$$\frac{20\,R_F}{R_S - R_F} < S$$

where 20 is a constant, R_F is the "faster" scan rate; R_S is the "slower'" scan rate; and S is the number of parameters in the "slower" group.

If this inequality is true, then the groups should be kept separate. Otherwise, they should be combined into the faster group.

Example

Given 2 Collection groups; one at 4 seconds, the another at 8 seconds, and with 50 points in the 8 second group, the calculation would be:

$$\frac{20 * 4}{8 - 4} < 50 \text{ or}$$

$$20 < 50$$

The inequality is true, which means that these 50 points should be left in the 8-second group.

It is also obvious that, for the given collection rates, an 8 second group of less than 20 points should be combined into the 4 second group.

TIP

This recombination is valid as long as the resultant Group size (the numerator) is less than 200. This is because data request messages have a maximum size of 200 references. Groups with more than 200 parameter references result in multiple messages (and the intent here is to reduce the number of data request messages).

Additional UCN group divisions based on your particular network dynamics and load may prove useful. Here are some examples:

- Separate slow parameters (such as temperature and tank level) from fast parameters (such as flow and pressure).
- Separate slow UCN boxes from fast UCN boxes (for example, separate APM and PM from HPM parameters). Do this only if your process warrants it since the creation of additional collection groups can be inefficient.

Cross Screen Actor Functionality on a GUS

This section contains instructions and an example of how you can emulate the cross screen (CROSSCRN) actor functionality available in the TDC on a GUS.

Prerequisites

To complete the procedures listed in this section, the user should be familiar with

- GUS Scripting
- TDC Button Configuration
- AM Custom Point Building
- Custom Data Segment Definition

Overview

The user will create a new script in a GUS picture that will update parameters defined by a Custom Data Segment (CDS) on an AM Custom point. The parameters on the AM Custom point will be used to pass picture name and console station/screen number data to a button configured on a TDC Universal Station to execute the CROSSCRN actor. When correctly implemented, the GUS user will be able to press a button on a GUS graphic and cause a CROSSCRN to execute on TDC.

The following steps are provided as an example only. There are other techniques that could be implemented to achieve the same functionality. This is a “rough” example to provide the user with the basic steps that must be performed to create CROSSCRN functionality. It would be prudent for GUS/TDC users to review the examples and add error checking where appropriate.

Here are the four basic steps that must be performed:

3. Define the Custom Data Segment
4. Build the AM Custom Point
5. Configure a Button on Universal Station
6. Create Script on a GUS Graphic

Define the Custom Data Segment

```
-- This example implements a Custom Data Segment to provide a
-- data access conduit to pass parameters to a button configured
-- on a TDC Universal Station.

CUSTOM

PARAMETER xscrname:STRING
-- Name of picture to send cross screen
VALUE ""
ACCESS OPERATOR
BLD_VISIBLE

PARAMETER scrn_no:NUMBER
-- Screen number to send cross screen
VALUE 0
ACCESS OPERATOR
BLD_VISIBLE

END CUSTOM
```

Note that although "BLD_VISIBLE" is the default, for purposes of documentation it has been included on each parameter in the above example.

Build the AM Custom Point

For this example the following parameters were configured on an AM Custom point.

```
NAME = TDCGUSDA
UNIT = 01
NOPKG = 1
PKGNAME = TDCGUSDA
```

When properly configured, page 5 of the builder will display parameters defined in the CDS that is included as the "package" on this AM Custom point.

Configure the Button on Universal Station

```
CROSSCRN(TRUNC(GS_REAL(TDCGUSDA.SCRN_NO)));
SCHEM(GS_STR(TDCGUSDA.XSCRNAME))
```

The first line in the configured button retrieves the scrn_no parameter from the AM Custom point (TDCGUSDA in the above example), truncates that number to an integer and returns that to the crosscrn actor which tells the station to execute the next actor on the screen number requested.

The second line in the configured button retrieves the xscrname parameter from the AM Custom point (TDCGUSDA in the above example) and returns it as a string to the SCHEM actor that will mount the picture on the screen requested by the crosscrn actor.

Create script on a GUS graphic

Here is an example how to create script on a GUS graphic:

Sample Script

```
Public sta As Integer ‘ variable to be used as station number
Sub OnDisplayStartup()
    ' Get number of THIS Screen/Station
    sta = dispdb.[$stnnum]
End Sub
Sub OnLButtonUp()
    ' Get user to enter desired Picture Name and
    ' Screen/Station number
    lcn.tdcgusda.xscrname = AskBox$("Enter Crosscrn Picture:")
    lcn.tdcgusda.scrn_no  = AskBox$("Enter Crosscrn Number:")
    ' If sending Picture to THIS station just do an
    ' InvokeDisplay
    If lcn.tdcgusda.scrn_no = sta Then
        InvokeDisplay lcn.tdcgusda.xscrname
    Else
    ' If sending picture to some other station, fire the
    ' button that has been configured to run crosscrn script.
        KEY sta, enum("$BUTTONS:AN_CNF40")
    End If
End Sub
```

Explanation of example script

OnDisplayStartup – this routine will run only one time when the user starts the display. All it does is get the station number of the current station and store it into a public variable for later use.

OnLButtonUP – this routine will execute each time the user "presses" the target and clicks the left mouse button. When the left button is released the user will be presented with a dialog box to enter the name of the desired picture to mount. Next the user will be presented with a dialog box to enter the station/screen number to mount the picture on.

After the user has entered both the picture name and screen number a test is performed to determine if this is an actual cross screen request or if the user has requested to mount the picture on the current screen. This test is necessary because of a problem with the cross screen implementation on GUS that causes the station NOT to search for GUS Graphic pictures on the current station when executing the SCHEM actor.

If the request is for the current screen, the script does a simple invoke of the display, which will cause it to be mounted at the station the script is running on. If the request is for some other screen, then the script will fire the button configured for the cross screen and the picture will be mounted on the requested screen.

Limitations

Display names used with this technique must be 8 characters or less or they will be truncated. For example, if the user wanted to mount the picture "blueflower" on screen two the name would be truncated to "blueflow" and an error returned to screen 2.

References

In conjunction with the Prerequisites listed earlier, the following TDC 3000 manuals can be used as an information source:

REFERENCE - EXTERNAL

Control Language/Application Reference Manual

Button Configuration Data Entry

Actor's Manual

Data Access Model

Introduction

From the perspective of a GUS display, there are three methods of reading or writing data to/from the LCN:

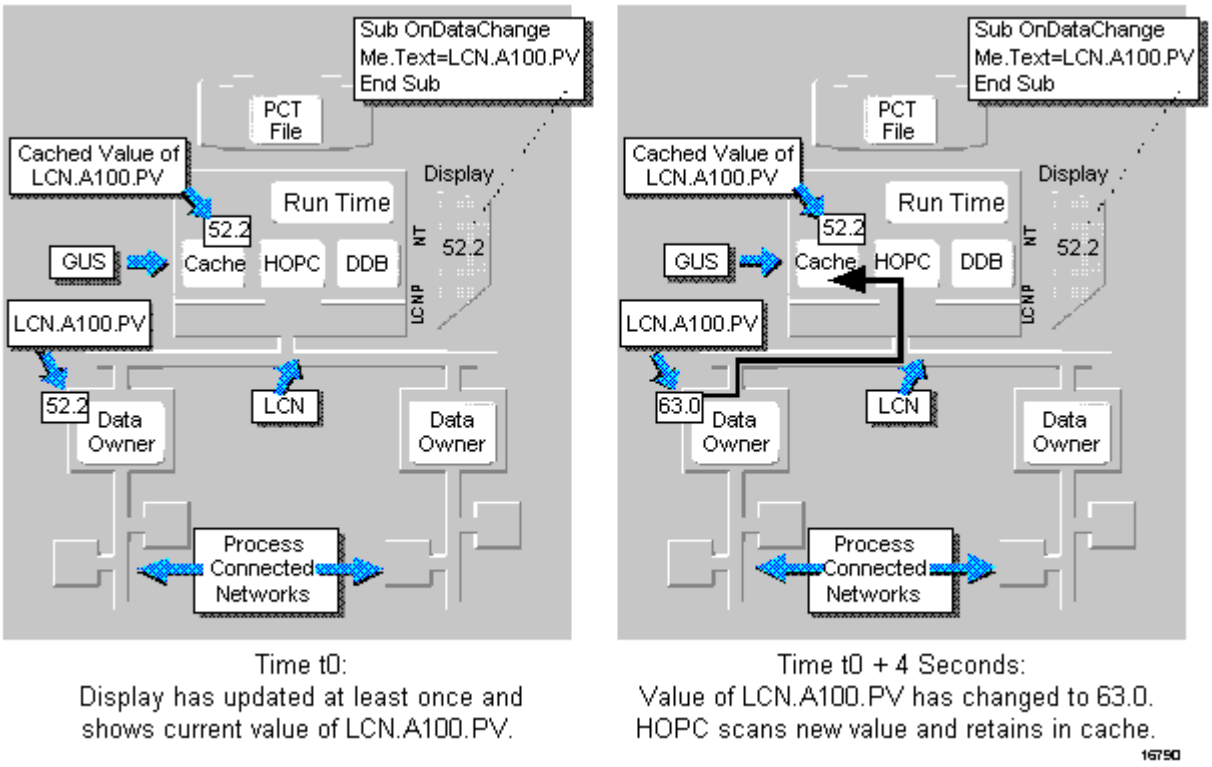
- Scanned (alternately called cached) read
- Immediate read
- Immediate write

Scanned Data

Scanned data is read from the LCN (using a component named HOPC) at a user-specified frequency, and retained in a GUS-resident ***cache*** memory. The GUS Run Time then reads the values from the cache as required.

The figures below illustrate the system operation for a display consisting of a single Text object used to display the value of LCN.A100.PV. At each scan cycle, the newly read value is compared to the previously read value and a DataChange event is generated if the values differ.

T0 and T0+4 Seconds Scanned Data Display Structures

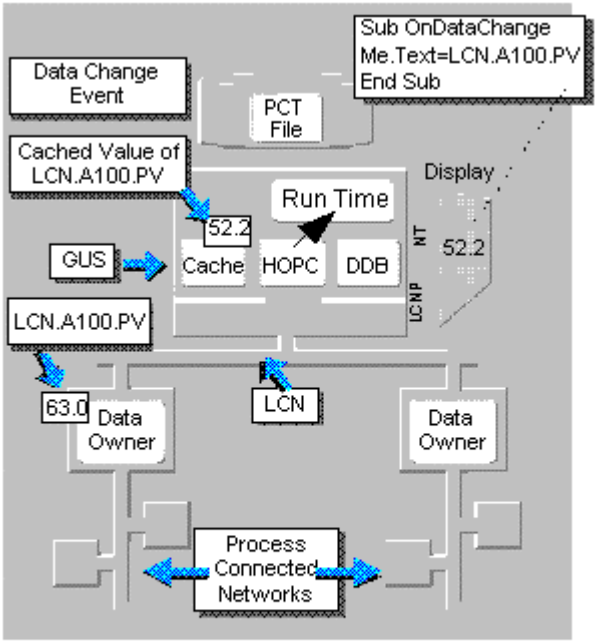


Time t0:
Display has updated at least once and shows current value of LCN.A100.PV.

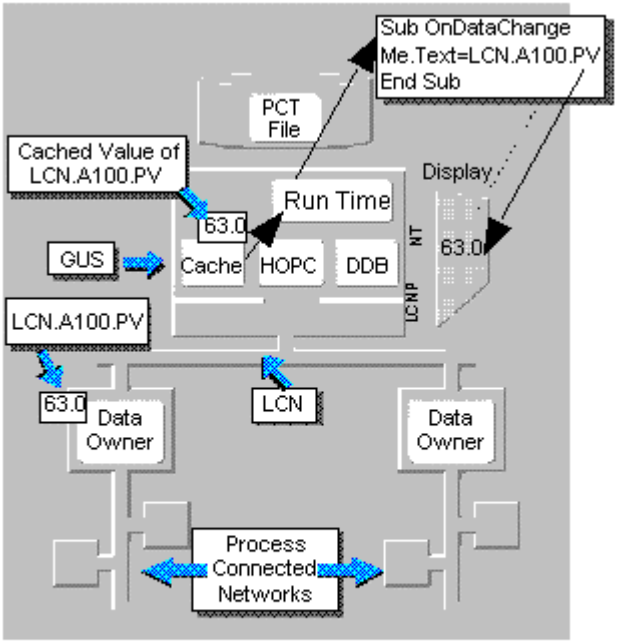
Time t0 + 4 Seconds:
Value of LCN.A100.PV has changed to 63.0.
HOPC scans new value and retains in cache.

16750

T0+4X Seconds Scanned Data Display Structures



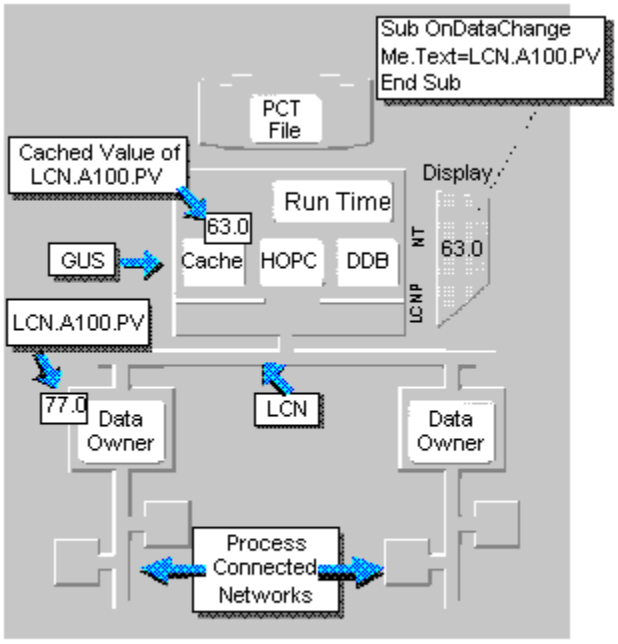
Time t0 + 4.x Seconds:
HOPC detects value changed and generates DataChange event for GUS Run-Time.



Time t0 + 4.x Seconds:
Scripts execute in response to DataChange request, read value from Cache, and display it.

16791

T0+5 Seconds Scanned Data Display Structure



Time t0 + 5 Seconds:
Value of LCN.A100.PV changes to 77.0.

16792

Immediate Read/Write

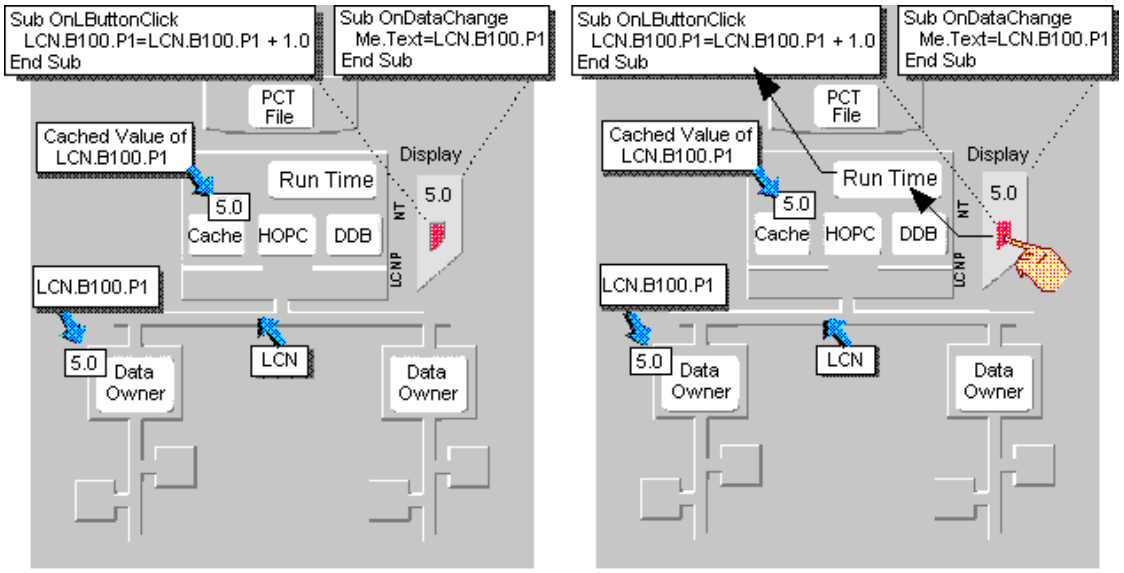
Data referenced by subroutines that are executed in response to a user event will read and write Data directly (immediately) to/from LCN points. These immediate reads and writes do not affect the value of data in the cache.

The figures below illustrate system operation for a display consisting of a Text object used to display the value of LCN.B100.P1 and a rectangle scripted to increment the value of LCN.B100.P1. When the operator clicks on the rectangle, the value of LCN.B100.P1 is read directly from the LCN, incremented, then stored directly back.

TIP

The cached value of LCN.B100.P1 is not affected by this operation.

T0 and T1 Seconds Read/Write Display Structures

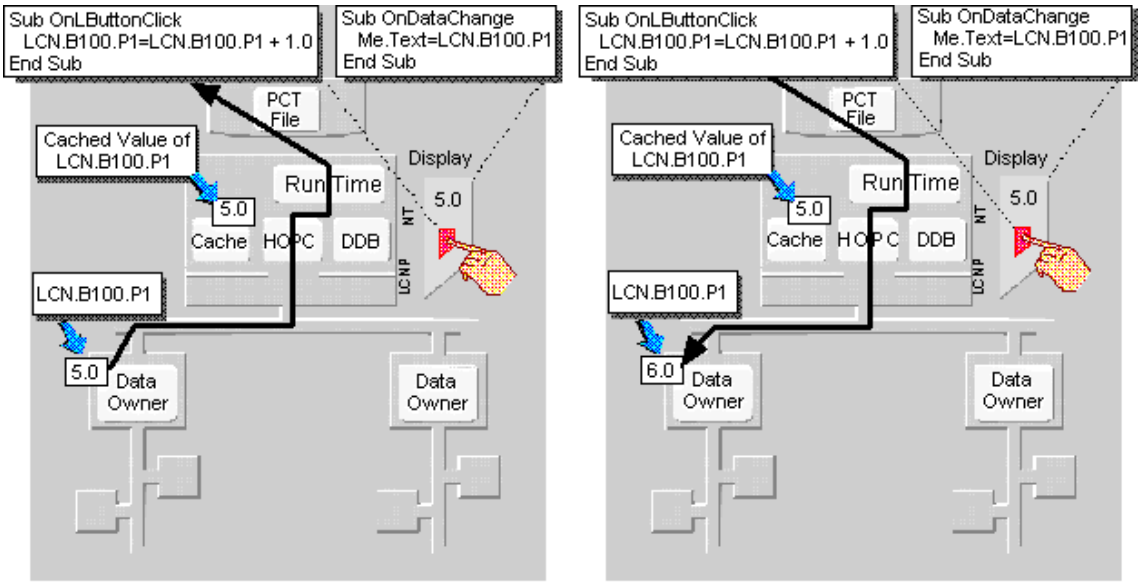


Time t0:
Display shows last scanned value of LCN.B100.P1.

Time t1:
Operator clicks on rectangle generating LButtonClick event. This results in execution of OnLButtonClick subroutine associated with the rectangle.

16793

T2 and T3 Seconds Read/Write Display Structures



Time t2:
GUS Run Time evaluates left hand side of Assignment Statement and does immediate read of LCN.B100.P1.

Time t3:
GUS Run Time evaluates right hand side of Assignment Statement, computes new value, and does immediate write of LCN.B100.P1.

16794

Indexed Access

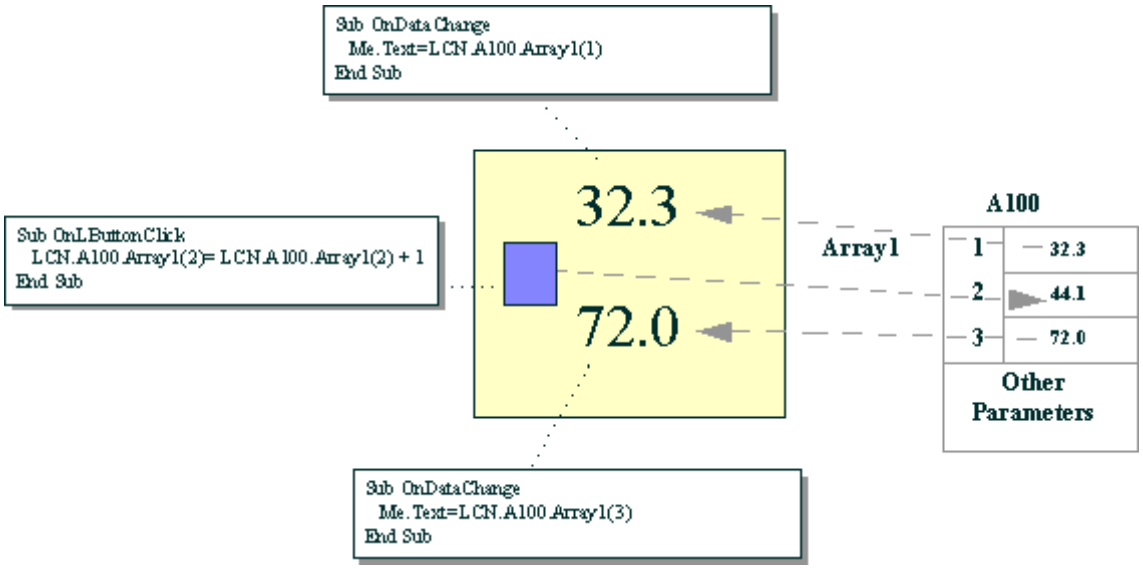
LCN points can have parameters that are arrays of values. You can access each element of the array by using an indexed name form, in which the index denotes the element of interest.

Example: Reading and Writing Indexed Data

The figure below illustrates a display consisting of two Text objects and a rectangle.

The topmost Text object is scripted to display the value of the first element of the array parameter *Array1* of the point *A100*. The bottom-most Text object is scripted to display the value of the third element of the array parameter *Array1* of the point *A100*. The rectangle is scripted to increment the value of the second element of the array parameter *Array1* of the point *A100*.

Reading and Writing Indexed Data



Indirect Access

Indirect access means that a display can reference the parameters of LCN points by "going through" another variable that contains the name of that point.

Indirect access is enabled by two special data types: *Entity* and *Variable*.

Entity data type

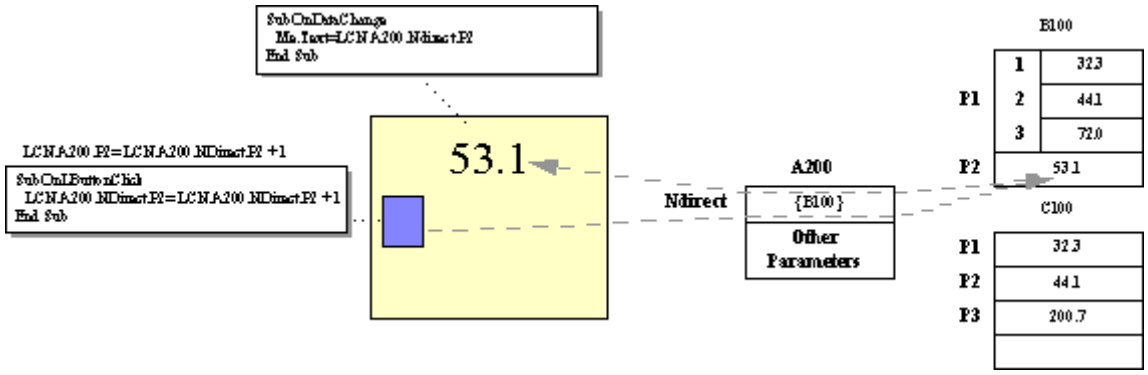
The Entity data type means the data is the *name of an LCN Point.* This allows a display to reference the value of a parameter of the point that is denoted by the name contained in an LCN point.parameters whose data type is entity.

Example: Reading and Writing Indirectly

This sounds complex, but is really rather simple. The figure below shows a display consisting of a Text object and a rectangle. The text reads data indirectly through the parameter *Ndirect* on the point *A200*. The rectangle increments the data indirectly through the parameter *Ndirect* on the point *A200.*

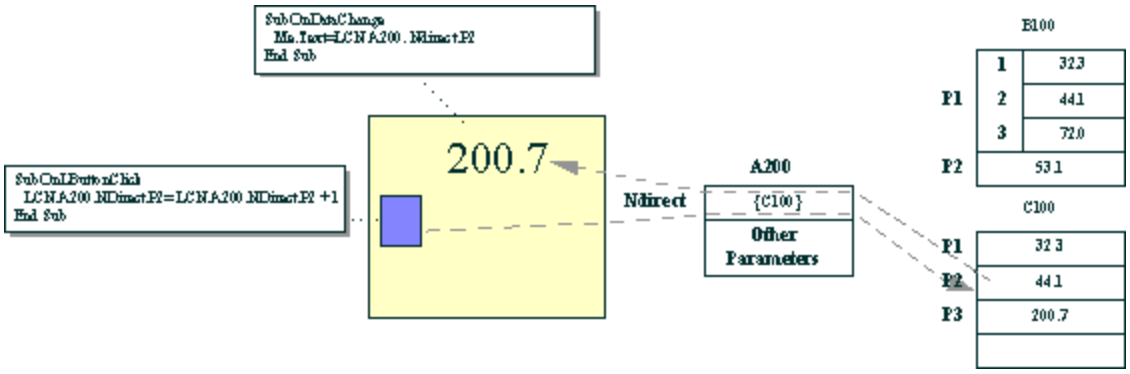
In the figure below, the value of A200.Ndirect is the name of the point B100. Hence the Text object on the display will show the value of B100.P2 and the rectangle will increment the value of B100.P2.

Indirect Access When A200.Ndirect Contains the Name of B100



In the figure below, the value of A200.Ndirect has been changed to contain the name of the point C100. Hence the Text object on the display will show the value of C100.P2 and the rectangle will increment the value of C100.P2.

Indirect Access When A200.Ndirect = Name of C100



External and Internal Names

Each LCN data point has two kinds of names: an internal name and an external name.

The External name is the name used by humans to identify points and is represented by a string (e.g., “A100,” “B100,” ... etc.).

The Internal Name is the name used by the system to identify points, and is represented by a binary number. The internal name has information that enables the system to locate and access the point very rapidly.

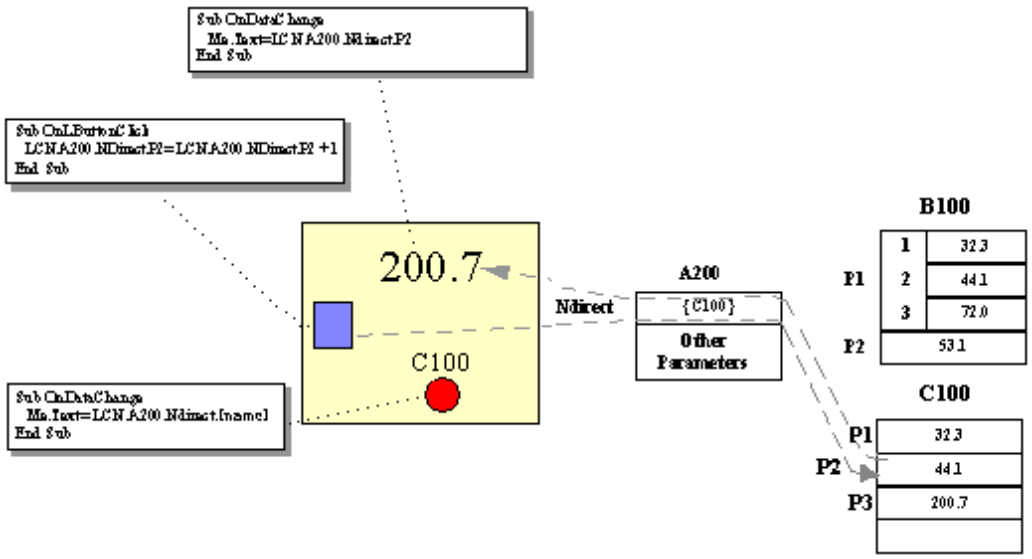
Example: Reading the External Name of a Parameter of Type Entity

Displays can be authored to READ values of parameters of type entity, as illustrated in the figure below.

This figure shows the same display used in the previous examples, with an additional Text object that displays the name of the “point of interest” for the display (i.e., the point the display is presenting).

This is achieved by using the name form A200.NDirect.[name], which directs the GUS display to access the External name form for the point name contained in A200.NDirect.

Displaying the Point Name Contained in A200.NDirect



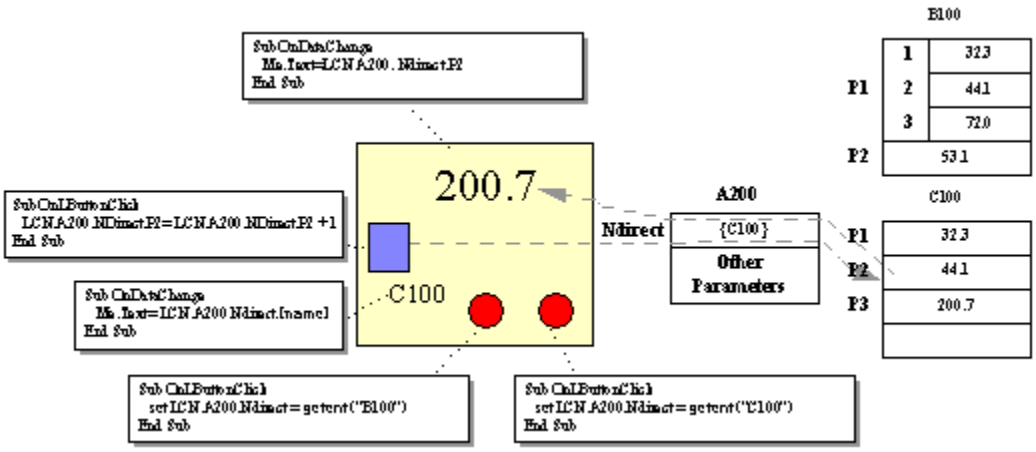
Example: Writing the External Name of a Parameter of Type Entity

Displays can be authored to WRITE to values of parameters of type entity, as illustrated in the figure below.

This figure shows the same display used in the previous examples, with the addition of an Ellipse object that has been scripted to change the “point of interest” for the display (i.e., change the point the display is presenting).

This is achieved by using a statement of the form A200.NDirect.[name]=“point name,” which directs the GUS display to store the External name form for the specified point in A200.NDirect.

Changing the Point Name Contained in A200.NDirect



REFERENCE - INTERNAL

Refer to the subsection Using the Data Types Entity and Variable in Scripts for additional information.

Display Data Base (DDB) Model

In general, there are two types of Display Data Bases:

- the Global Display Data Base
- the Local Display Data Base

Each of these databases consists of a set of variables that provide variables useful in displays.

Global DDB

Values stored in the Global DDB remain intact as long as a GUS Run Time is executing, and can be used to pass information from one executing display to another.

ATTENTION

Information is passed only once when another display is invoked, and only in one direction — from the display currently on the screen to the display being evoked.

Example: Using Global DDB to Reduce Display Authoring and Maintenance

One of the very useful applications of Global DDB is the creation of a single display that can be used to show many different views of the same *class* of process equipment. This example employs a single window.

For example, a site may have several reactors, all of which are similar in appearance and operation; the only difference between the displays of different reactors are the LCN point.parameters. Having a single display for all reactors drastically reduces the cost of authoring and maintaining the displays (one instead of four).

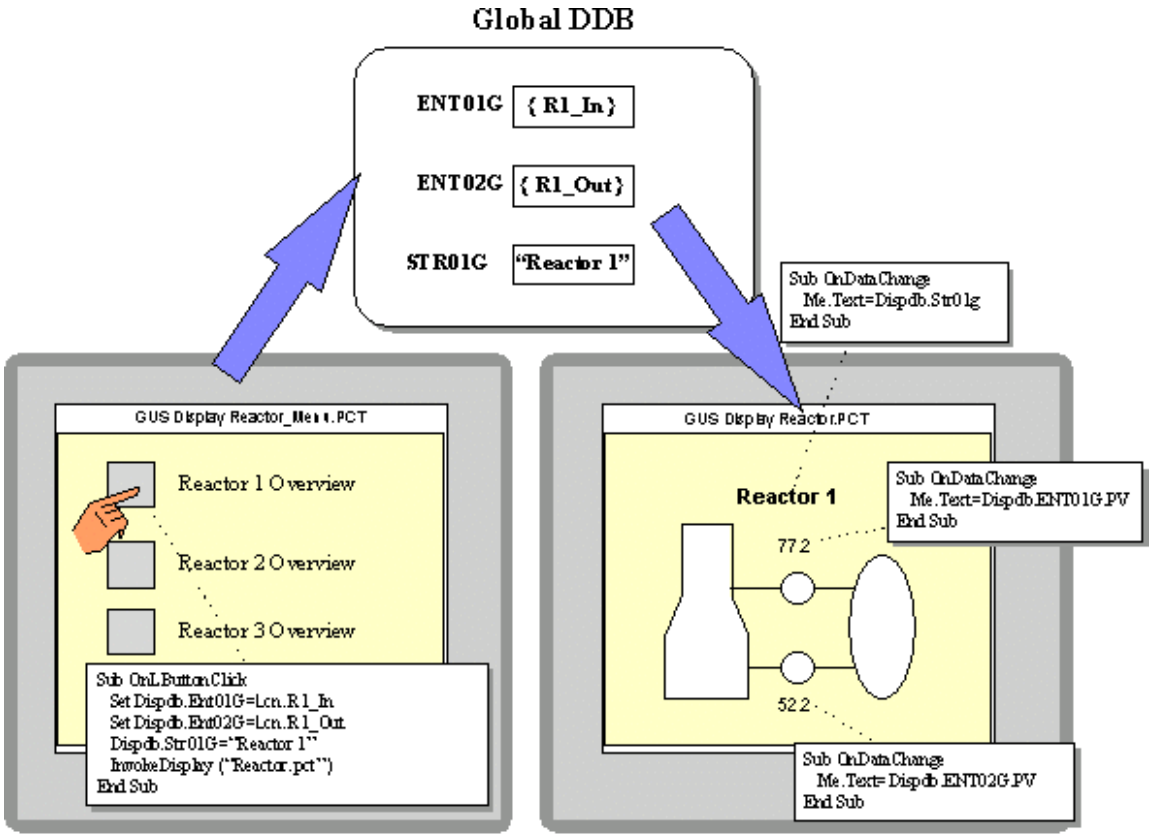
The figures below illustrate the operation at a site that has four reactors. The site has a display allowing the operator to select a reactor of interest. Selecting a reactor results in calling a display for it.

There is only a single Reactor PCT display (named Reactor.PCT) that is capable of displaying the values of the points for any of the four reactors in the plant. This is achieved as follows:

- References to LCN points in Reactor.PCT are all indirect accesses through the DDB variables ENT01G and ENT02G.
- The Name of the Reactor appears at the top of the Reactor display. This is a Text object that displays the value of String01G.
- Each of the rectangles invokes a Reactor display Reactor (e.g., “Reactor 1”), stores the name of the reactor into String01G and the names of the data points for the reactor into ENT01G and ENT02G, and then invokes the Reactor.PCT display.

In this example, we consider a system that does not have the Multiple Displays package, i.e., there is only one display active at any time and calling a Reactor display from the Reactor Menu results in replacing the Reactor Menu with the Reactor display.

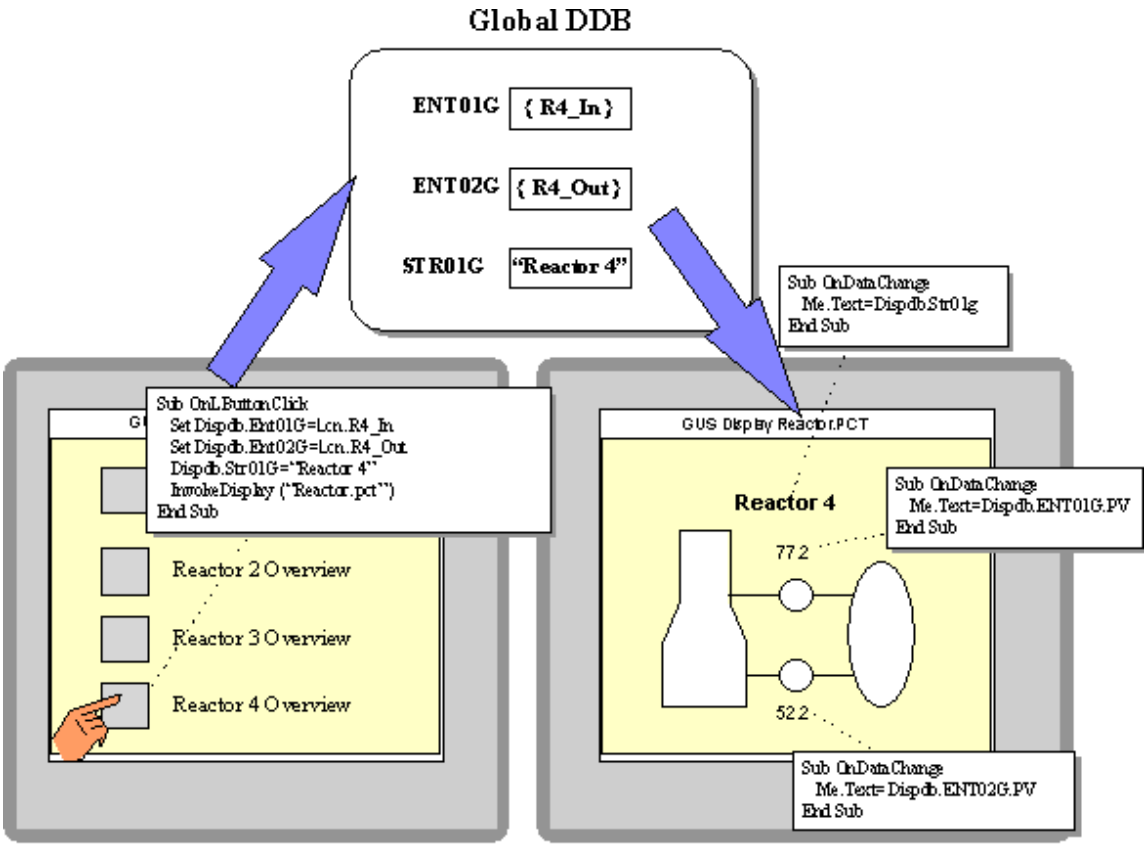
Viewing Reactor 1



Time t0: Operator Invokes Reactor 1 Display

Time t1: Reactor.PCT Runs, Accessing Data Indirectly Through ENT01G and ENT02G

Viewing Reactor 4



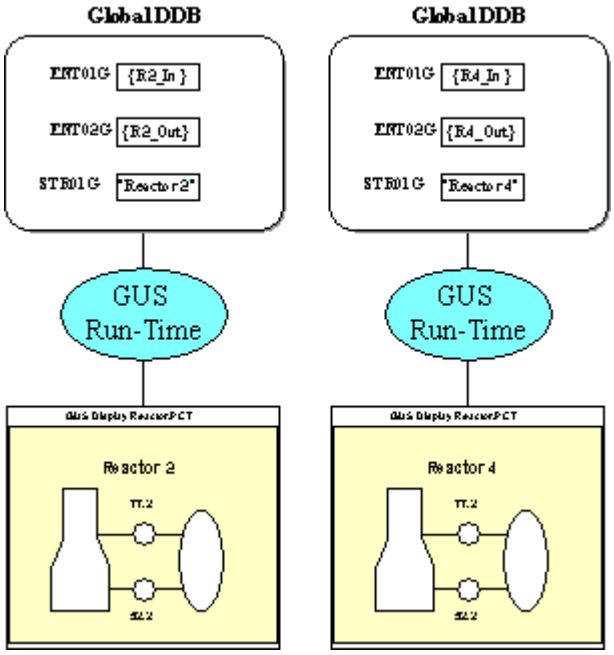
Time t0: Operator Invokes Reactor 4 Display

Time t1: Reactor.PCT Runs, Accessing Data Indirectly Through ENT01G and ENT02G

Relationship between Displays, GUS Run Time, and Global DDBs

The figure below illustrates the relationship between a display, the GUS Run Time process, and Global DDBs. Specifically, each display is rendered by a single GUS Run Time process and each GUS Run Time process has a single Global DDB space.

GUS Run Time and Global DDBs



ATTENTION

Each GUS Run Time process has its own Global DDB space. When one display invokes another display, the invoking display's Global DDB gets copied to the Global DDB of the display being invoked. From that time forward, the Global DDBs are independent. Changing a value in one will not affect the other.

Example: Using Global DDB in a Multiwindow SafeView Workspace

Part 1: Calling the First Reactor Display

The figure below illustrates SafeView Workspace with four windows. Three of the windows are used for GUS displays and one is used for the Native Window. For the purpose of this example, we assume the three windows used for the GUS display are round robin.

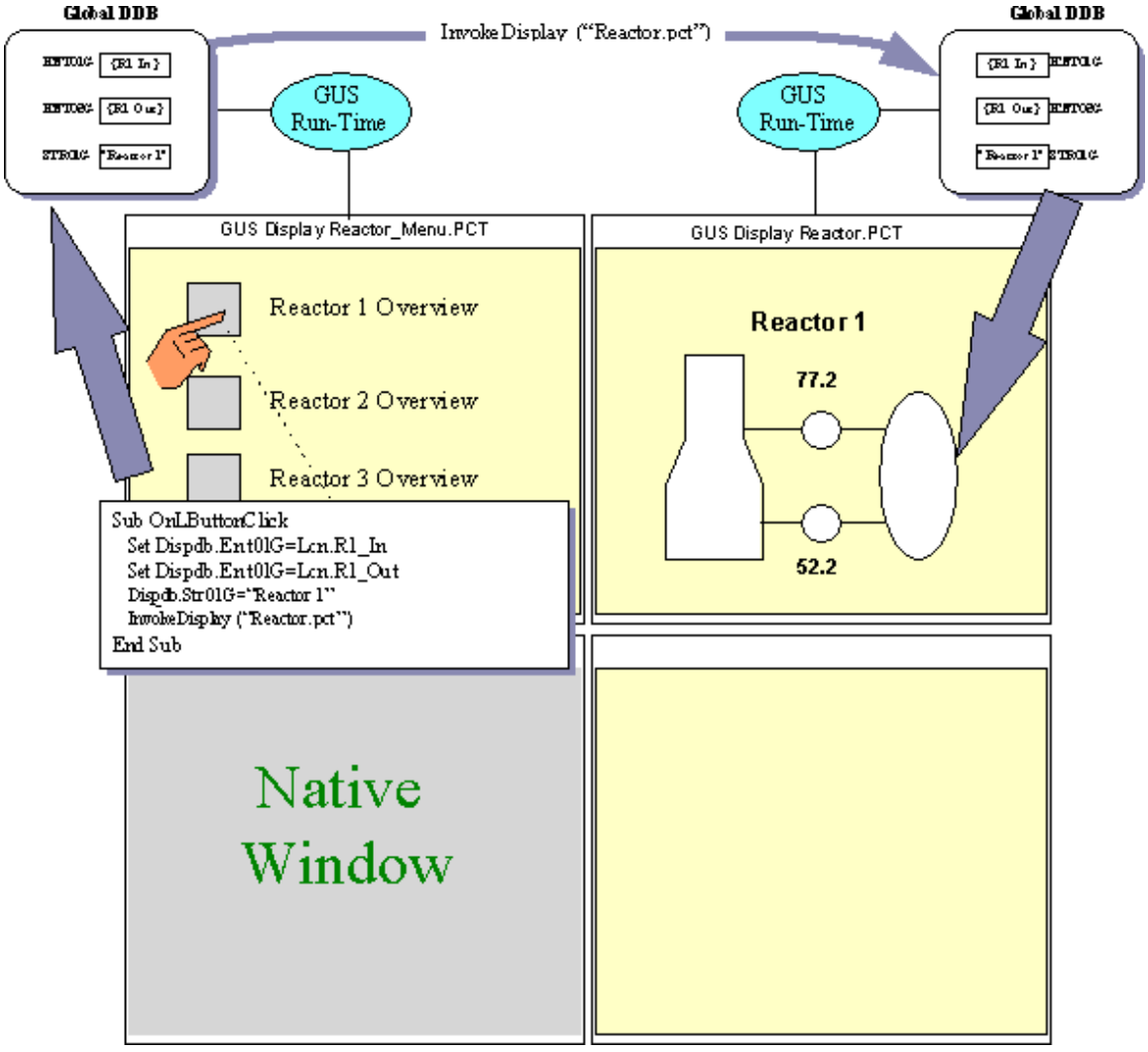
In this figure, the Menu display has been invoked in the upper-left window; the operator has selected the Reactor 1 Overview display.

This selection resulted in executing the OnLButtonClick subroutine for the rectangle. This subroutine stored the names of points for Reactor 1 into the Global DDB Variables ENT01G and ENT02G. The name of the reactor was stored into STR01G.

The InvokeDisplay statement then executes. It copies the Global DDB Values from the Global DDB space associated with the run time executing the Reactor_Menu display to the Global DDB space for the run time associated with the SafeView window to receive the next display. It then invokes the Reactor display in that window. When that display executes, it shows the values for the points in reactor 1, by accessing them indirectly through ENT01G and ENT02G.

In the figure below, the background color of the Native Window has been changed to enhance the visibility for black and white printing.

Part 2: Calling the Next Reactor Display

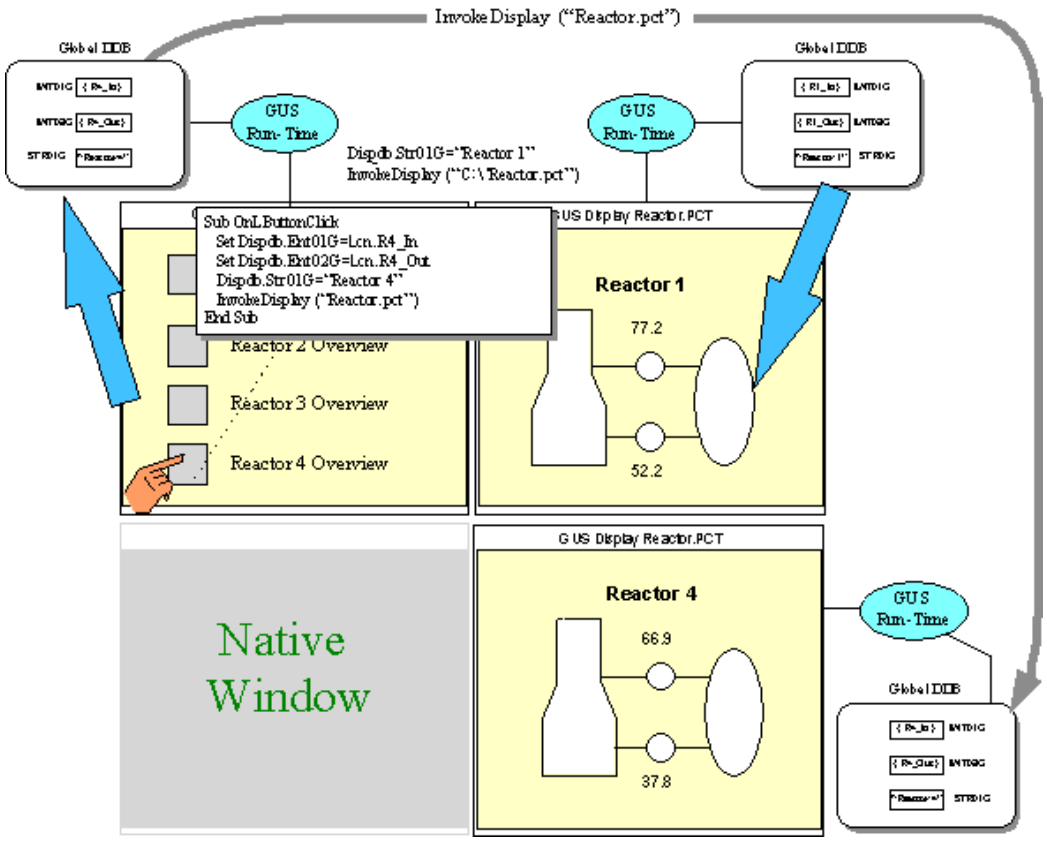


The figure below illustrates the callup of a display for Reactor 4, by selection of the appropriate object in the Reactor_Menu display. The OnLButtonClick subroutine for the

rectangle stored the names of points for Reactor 4 into the Global DDB Variables ENT01G and ENT02G. The name of the reactor was stored into STR01G.

The InvokeDisplay statement then executes. It copies the Global DDB Values from the Global DDB space associated with the run time executing the Reactor_Menu display to the Global DDB space for the run time associated with the SafeView window to receive the next display. It then invokes the Reactor display in that window. When that display executes, it shows the values for the points in reactor 1, by accessing them indirectly through ENT01G and ENT02G.

Calling the Reactor 4 Overview



In the above figure, the background color of the Native Window has been changed to enhance the visibility for black and white printing.

Propagation of System Events

Introduction

Conceptually, events move (or propagate) through the objects in a display, “looking for” a script to service it. When a system event finds a subroutine to handle, the subroutine is executed and the event continues until all objects have been visited.

DisplayStartUp and PeriodicUpdate Events

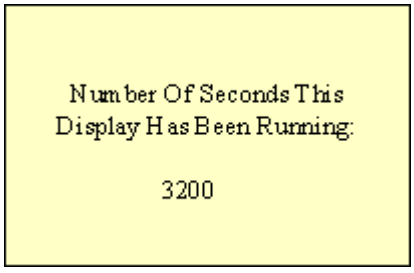
When a display is initially invoked, the GUS Run Time generates a DisplayStartUp event. Subroutines can then be associated with the Display object or any of its member objects to service this event.

As a general rule, subroutines that handle the DisplayStartUp event are used to initialize the display, an Embedded display, or an object.

Example: A Display That Shows the Number of Seconds It Has Been Active

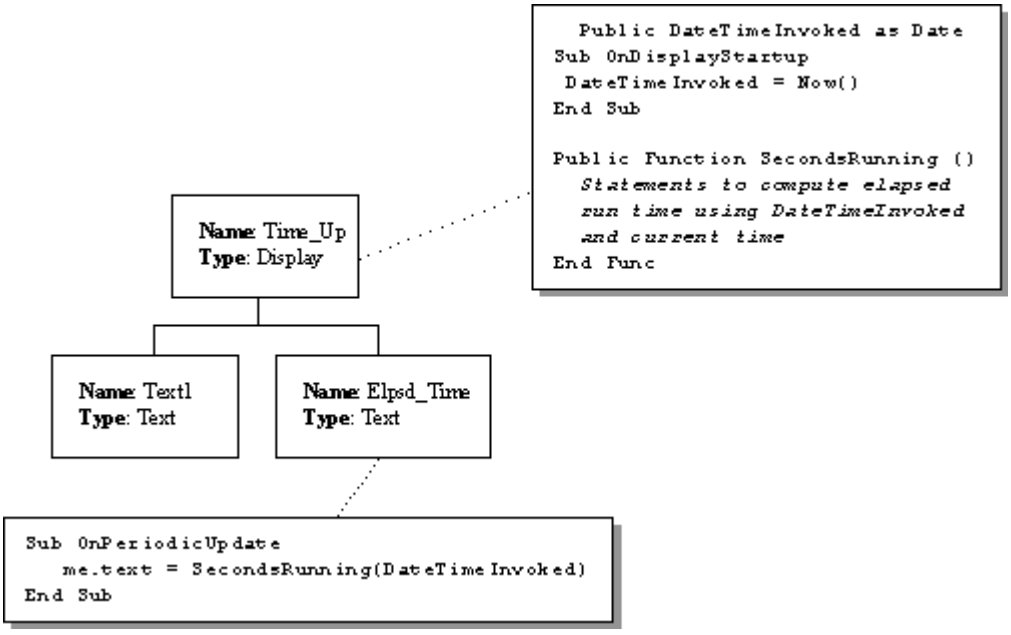
The figures below illustrate a display that shows the amount of time the display has been running and the structure of that display.

Display



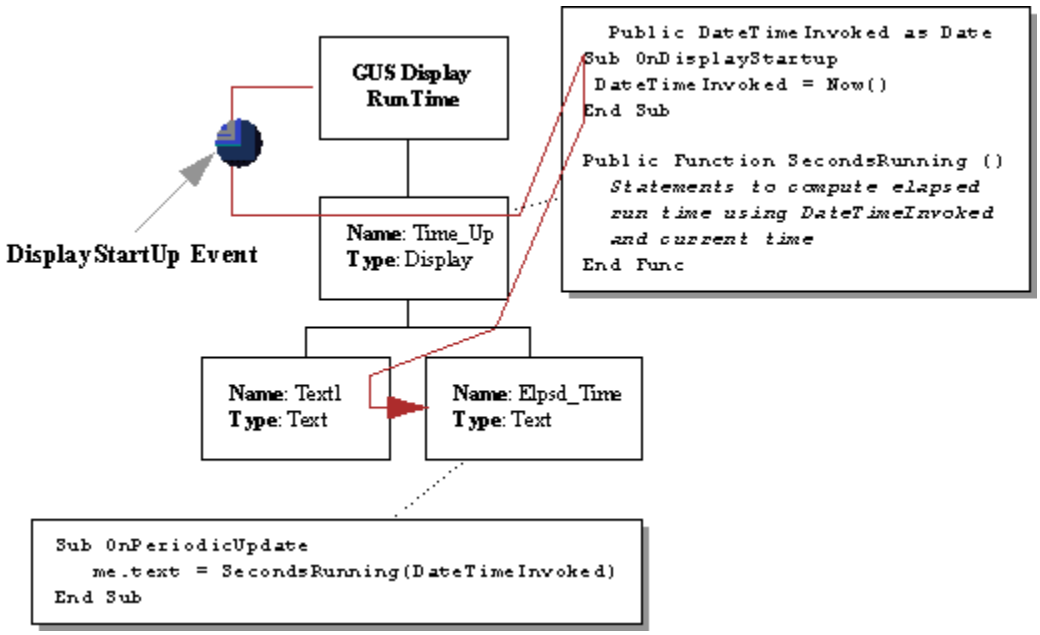
The display consists of three objects: the object representing the display, a Text object displaying the constant text “Number Of Seconds This Display Has Been Running:” (Text1), and a Text object that displays the elapsed time since the display was called, in seconds (Elpsd_Time).

Structure of Display



When the display is initially called, the GUS Run Time generates a DisplayStartUp event and sends it to the display. The figure below shows the path of the DisplayStartUp event as it propagates through the display.

DisplayStartUp Event Path



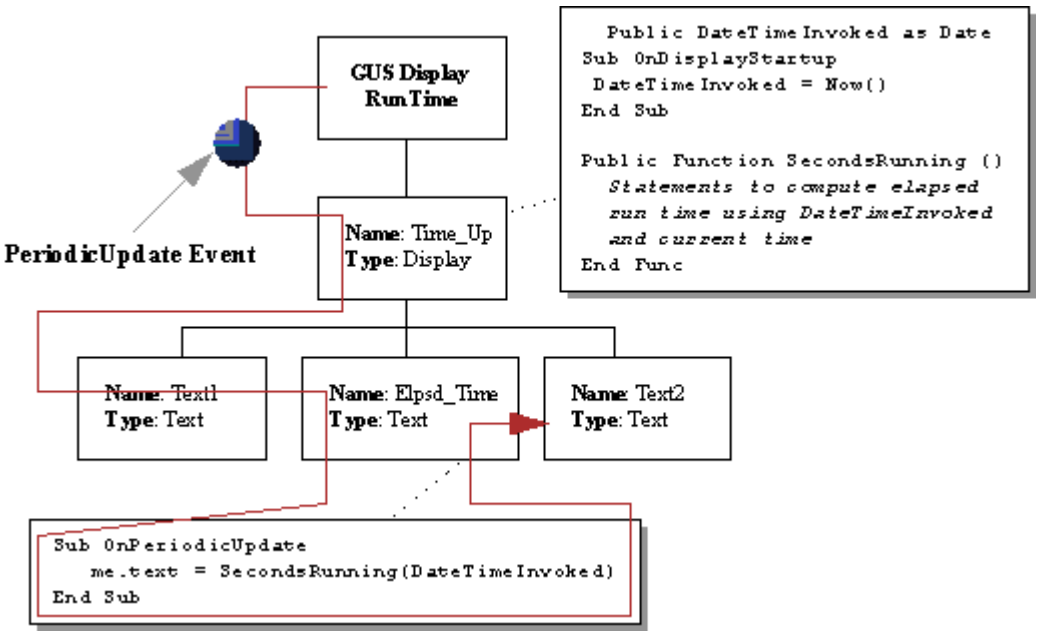
Propagation of the DisplayStartUp event

The procedure is as follows:

Step	Action
1	The DisplayStartUp event visits the Display object, finds the OnDisplayStartUp subroutine, and executes it.
2	This subroutine saves the current date and time in a Global variable.
3	The DisplayStartUp event then visits each object in the display. No other OnDisplayStartUp subroutines exist, so none are executed.

Every half second, the GUS Run Time generates a PeriodicUpdate Event and sends it to the display. The figure below shows the path of the PeriodicUpdate event as it propagates through the display.

Propagation of the PeriodicUpdate Event



The sequence is as follows:

Step	Action
1	The PeriodicUpdate event visits the Display object. Because no subroutine to service this event exists in the script of the Display object, the event continues to propagate.
2	The PeriodicUpdate event visits the Text1 object. Because no subroutine to service this event exists in the script of this object, the event continues to propagate.
3	The PeriodicUpdate event visits the Elpsd_Time object, finds the OnPeriodicUpdate subroutine, and executes it.
4	This subroutine calls a Global Function that calculates the time the display has been running. The result of the function is stored in the Text property of the Elpsd_Time object, displaying it to the operator.
5	The PeriodicUpdate event continues to visit all objects in the display, if there are any.

DataChange Event

When a display that references either LCN point.parameters, or DispDB.parameters, or Display.Params.parameters executes, the GUS Run Time detects when the .parameters change value and sends a DataChangeEvent to the display.

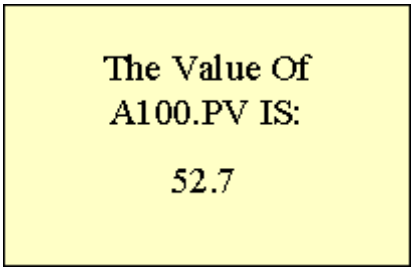
The DataChange event then propagates through the display, visiting each object in search of an OnDataChange subroutine.

A DataChange event is sent to the Display when it is initially invoked (but following the DisplayStartUp event).

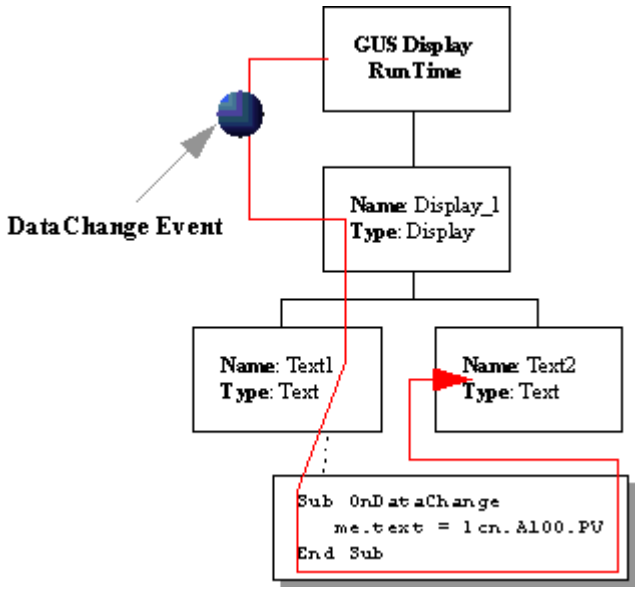
Example: A Display That Shows the Value of A100.PV

The figures below show the display and its structure.

Value of A100.PV



Structure of the Display



Propagation of a DataChange event

The procedure is as follows:

Step	Action
1	The DataChange event visits the Display object. Because no subroutine to service this event exists in the script of the Display object, the event continues to propagate.
2	The DataChange event visits the Text1 object, finds the OnDataChange subroutine and executes it.
3	This subroutine stores the value of A100.PV into the text property of Text1, causing it to be shown on the screen.
4	The DataChange event visits the Text2 object. Because no subroutine to service this event exists in the script of this object, the event continues to propagate.

TIP

In reality, all objects are not searched for an OnDataChange subroutine. When a display is verified, the GUS Display Builder minimizes the time to fire a subroutine in response to the datachange event. However, for the purpose of a model to explain the operation of the runtime, we can think in terms of each object being searched for an OnDataChange subroutine.

DisplayShutDown Event

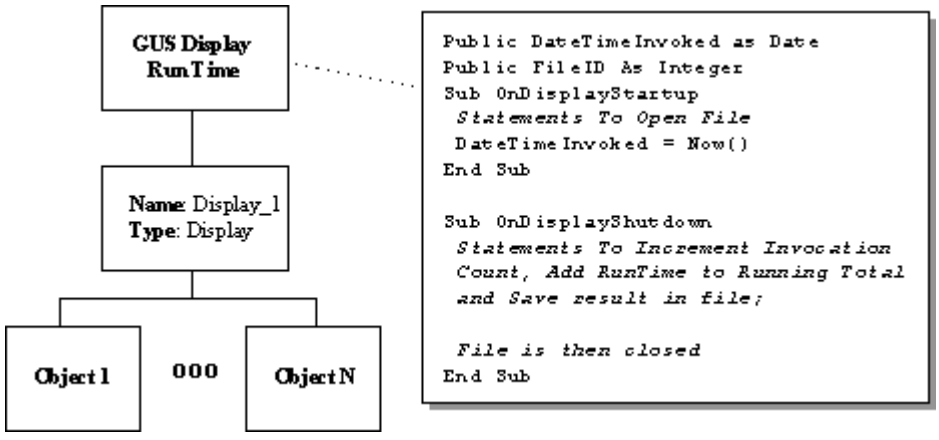
When a Display is closed, the GUS Run Time sends a DisplayShutDown event to the display. The event then visits each object in the display in search of an OnDisplayShutDown subroutine. When such a subroutine is found, it is executed.

In general, subroutines servicing the DisplayShutDown event are used to perform any cleanup required before the display is closed.

Example: A Display That Records Its Usage in a File

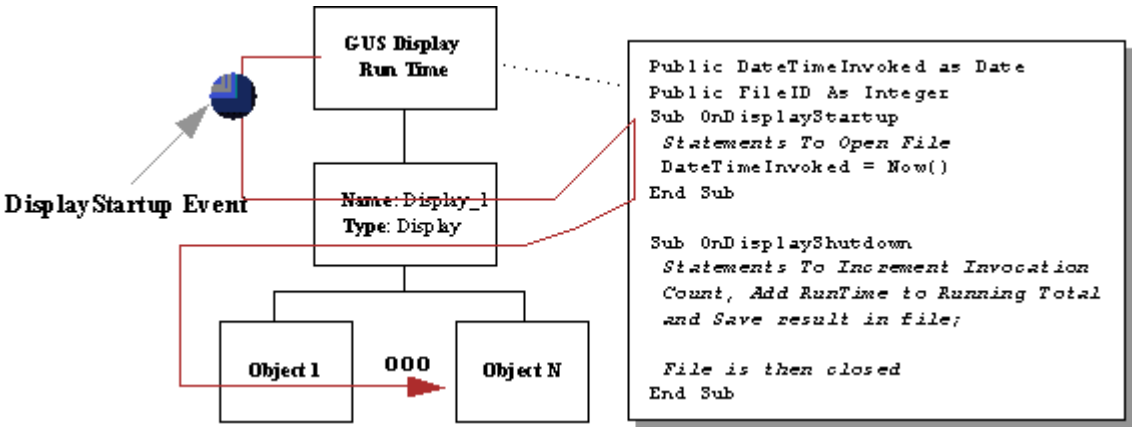
The figure below illustrates the structure of a display that records statistics on its use in a file. Specifically, the display records the number of times it was invoked and accumulates the time that it was active.

Display Structure 1



When the display is invoked, the GUS Run Time generates the DisplayStartUp event and sends it to the display. The OnDisplayStartUp subroutine opens the file, records the File ID in a Public variable, and initializes another public variable with the current time:

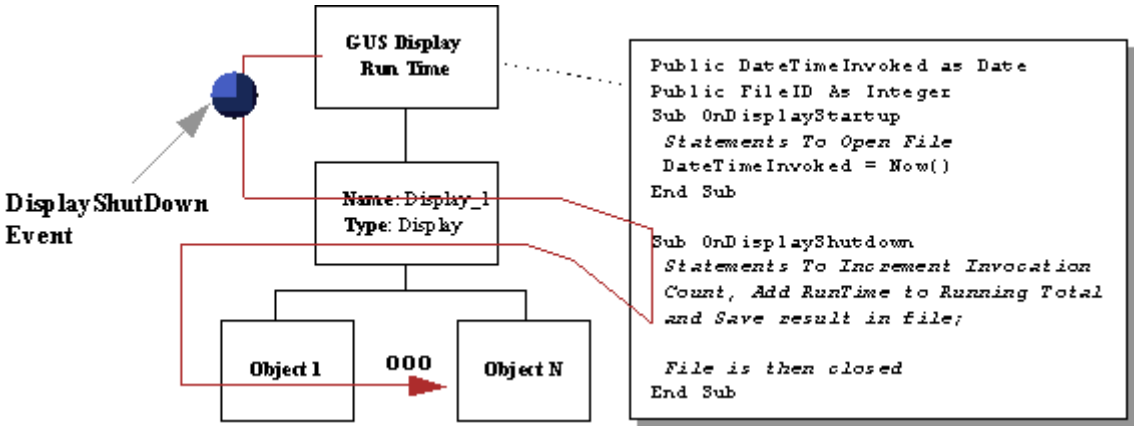
Display Structure 2



When the display is closed, the GUS Run Time generates the DisplayShutDown event and sends it to the display. The OnDisplayShutDown subroutine writes the statistical data to the file (the file is accessed using the Public Variable “File_ID”), then closes it.

When all OnDisplayShutDown subroutines have been run, the GUS Run Time closes the display.

Display Structure 3



Order of Subroutine Execution

The GUS Run Time provides certain guarantees on the order of subroutine execution for system events.

The order of execution guaranteed is outlined below:

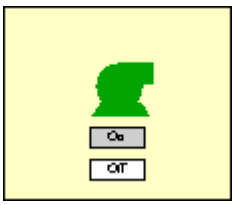
- All DisplayStartUp subroutines for a container object will be executed before the DisplayStartUp subroutines for member objects.
- All DisplayStartUp subroutines for an object (and its members) will be executed before executing any other subroutines (this ensures that the display and all of its objects are initialized before the output of information or execution of subroutines in response to user events).
- For embedded displays, the DisplayStartUp subroutine defined when the display is initially defined will be executed before the DisplayStartUp subroutine defined for the instance of the display.
- All DisplayShutDown subroutines for member objects will be executed before the DisplayShutDown subroutines for a container object.
- For embedded displays, the DisplayShutDown subroutine defined when the display is initially defined will be executed before the DisplayShutDown subroutine defined for the instance of the display.

The GUS Run Time ***does not*** guarantee the order of execution for subroutines servicing other events.

Example: Order of DisplayStartUp Subroutine Execution

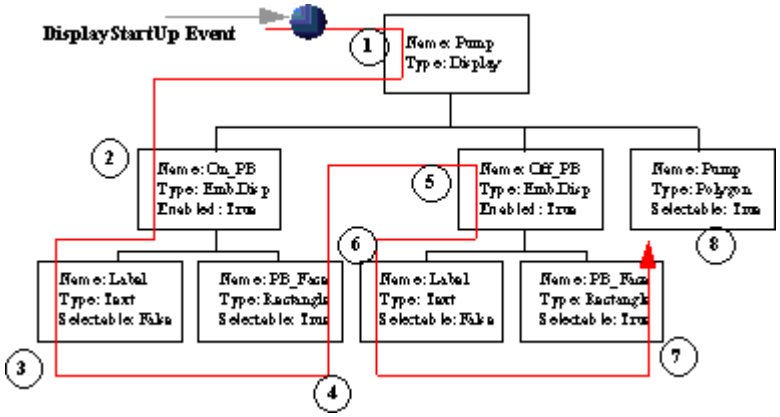
The figures below illustrate a display showing a pump. The display has a couple of embedded displays that act as pushbuttons to turn the pump on and off.

Pump Display

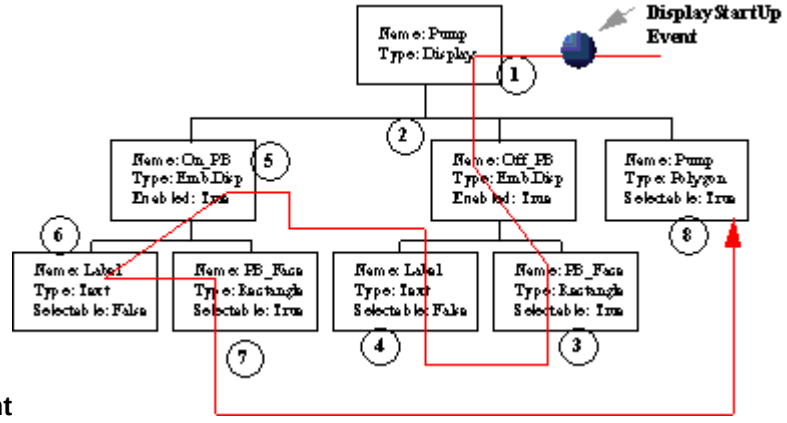


The figure below illustrates the structure of the Pump Display and shows two possibilities of the order of execution for subroutines servicing the DisplayStartUp event. These figures illustrate, that the GUS Run Time only guarantees the order of DisplayStartUp subroutines for container objects (i.e., that the DisplayStartUp subroutine for the container executes before the DisplayStartUp subroutine for the member objects).

One Possible Sequence of Visits to Display Objects by Display StartUp Event



Another Possible Sequence of Visits to Display Objects by Display StartUp

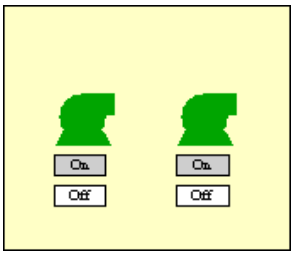


Event

Example: Order of DisplayStartUp Script Execution on an Embedded Display

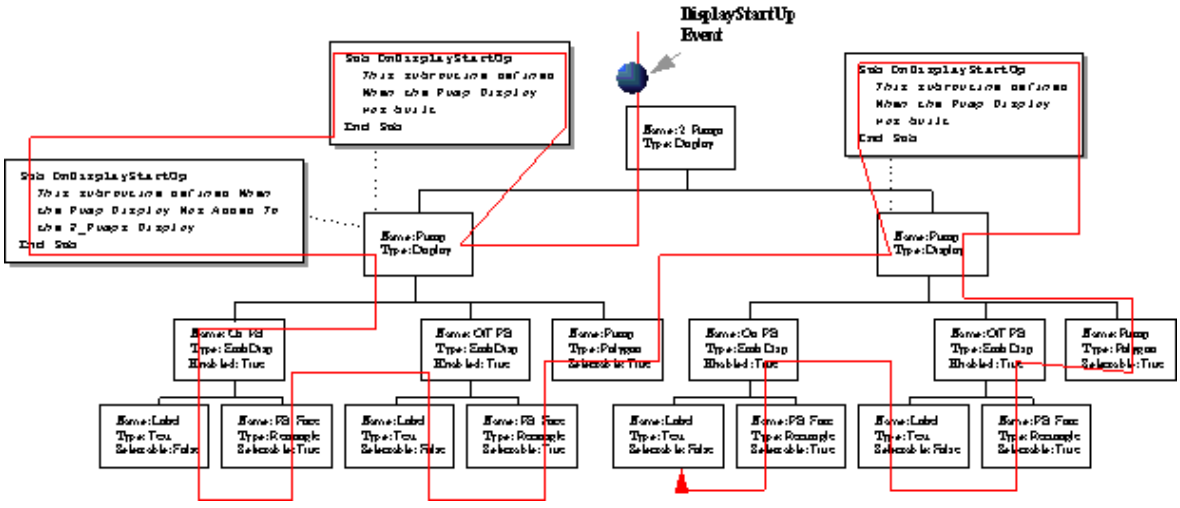
The figure below illustrates a display containing two pump displays as embedded displays.

Display Containing Two Embedded Displays



The figure below illustrates the structure of the 2_Pump Display showing that the OnDisplayStartUp subroutine defined when the Pump Display was constructed executes before the OnDisplayStartUp subroutine defined when the Pump Display was added to the 2_Pump Display.

Another Possible Sequence of Visits to Display Objects by Display StartUp Event



User Events

Introduction

The purpose of a GUS display is to represent a real-world process to an operator and allow the operator to manipulate the process through the display. To achieve this end, objects must be dynamic—objects must be able to alter their appearance, location, size, etc., based on the value of process variables and objects must be capable of responding to operator actions.

Scripting enables the display author to define dynamic Display objects.

Scripting Model

The basic scripting model is described in terms of Display Objects, Events, and Subroutines.

Displays consist of a collection of ***Display Objects*** (for example, a display can have a line, a rectangle, a Text object, etc.). A display is also represented by an object.

When users interact with the display, ***User Events*** are generated. For instance, clicking with the left mouse button generates a LButtonClick event; clicking with the right mouse button generates a RButtonClick event.

Similarly, the GUS Display Run Time generates ***System Events***. For example, the GUS Run Time generates two PeriodicUpdate events per second; when the value of a point.parameter that is referenced by a display changes, a DataChange event is generated.

Objects can be scripted to react to particular kinds of events. Specifically objects can have ***subroutines*** associated with them that will execute in response to a particular kind of event. For example, a rectangle object can have a subroutine that will execute when the user clicks on it with the left button of the mouse (the act of clicking generates a LButtonClick event).

A Display Object may have several subroutines to handle various kinds of Events. The collection of subroutines for a Display Object is called the ***Script*** for that Object.

Conceptually, events propagate through the display “looking for” the appropriate subroutine to handle the event. When the appropriate subroutine is found, it is executed and event propagation stops.

Propagation of User Events

Conceptually, events move (or propagate) through the objects in a display "looking for" a script to service them.

Propagation of user events through the objects in a display is controlled by a well-defined set of rules:

1. User events propagate along the z-axis of the display (from nearest object to farthest object), until a Selectable object is encountered or until all objects in the container have been visited.

2. A user event will result in execution of a subroutine if, and only if, the object is selectable.

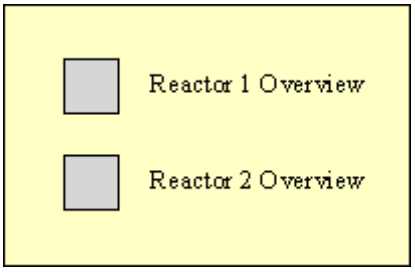
Basic examples of event propagation

This section gives examples illustrating the basic event-propagation concepts.

Example 1: A Menu Display

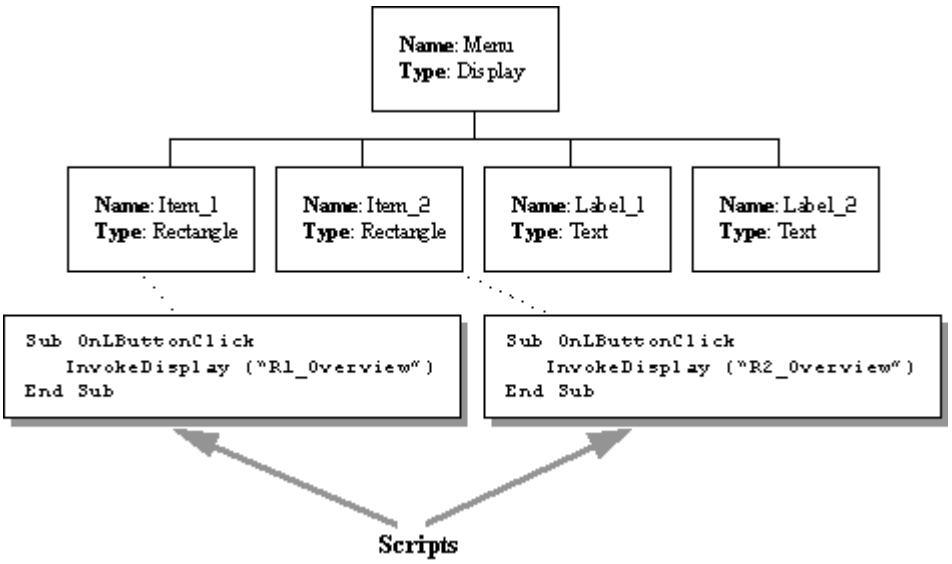
The figure below shows a display that functions as a menu. Clicking on either rectangle with the Left Mouse Button will result in calling another display:

A Menu Display



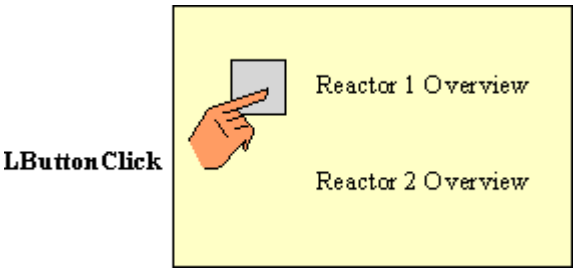
As shown in the figure below, the display consists of five objects (one of which represents the display itself).

Structure Of Menu Display

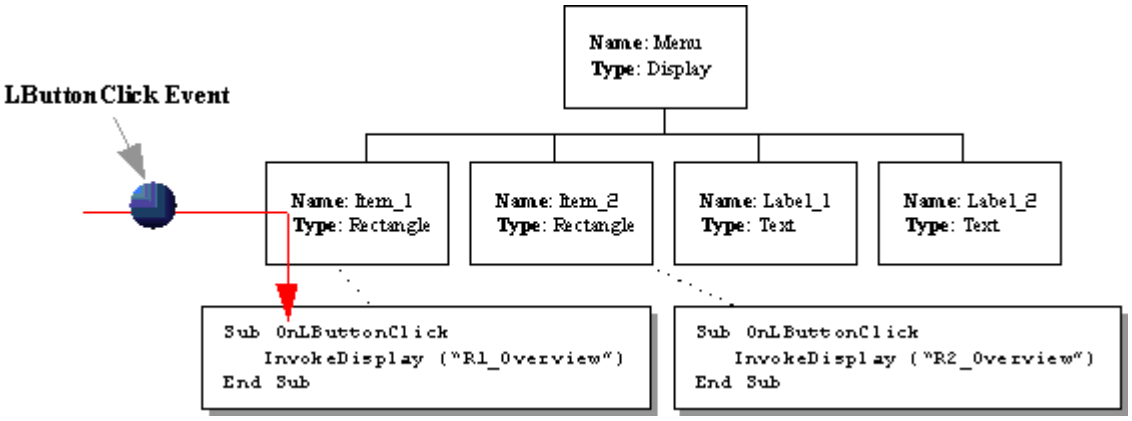


When the user clicks on the upper rectangle with the mouse, a LButtonClick event is generated. This event then propagates through the display looking for a subroutine to handle the event.

Clicking on Rectangle Generates User Event



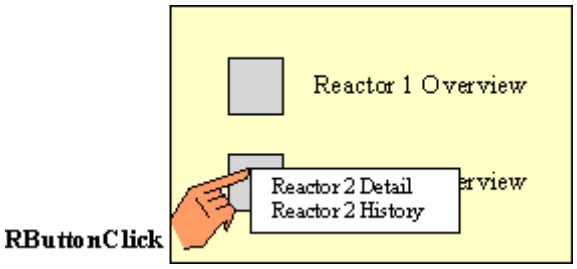
The Event Finds the Subroutine to Handle it and Causes the Subroutine to Execute



Example 2: Objects That Respond to More Than One User Event

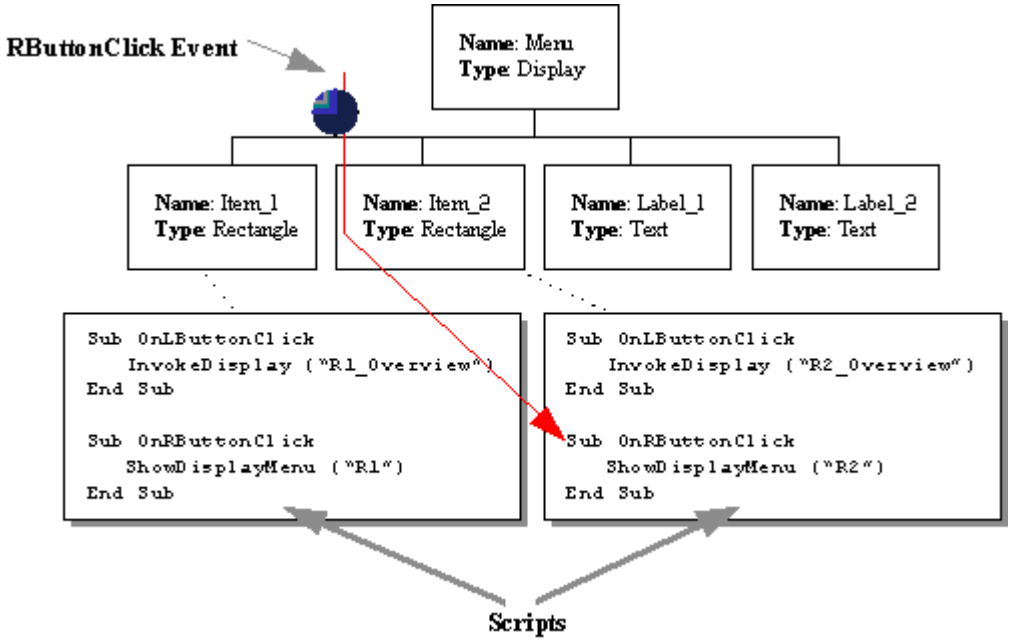
The figure below shows the Menu Display after it has been enhanced to respond to both the LButtonClick and the RButtonClick events. Clicking on either rectangle with the *Right Mouse* Button will display a menu of displays associated with a Reactor. The operator can then select a particular display from the menu:

Enhanced Menu Display



As shown in the figure below, subroutines to handle the RButtonClick event have been added to the rectangles (the GUS Basic for the subroutine to show the display menu is not shown in the figure).

Structure of Enhanced Menu Display

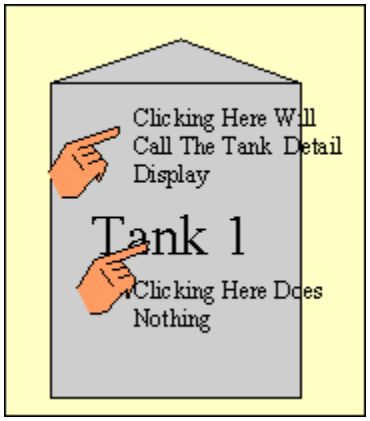


Example 3: A Tank Display

The figure below illustrates a display for a tank. The author of the display scripted the Rectangle representing the body of the tank to respond to a Left button click by calling a more detailed display for the tank.

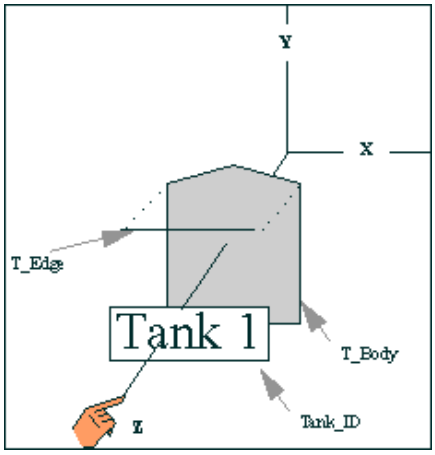
While the display "works" (i.e., it will call the detailed tank display), it does not behave the way the author intended. Specifically, clicking on the Text object used to display the tank name does not result in any user-visible action.

Tank Display



The figure below illustrates the "z" ordering of the objects in the Tank Display (Tank_ID and T_Edge are in front of T_Body).

Z Order of Objects in Tank Display

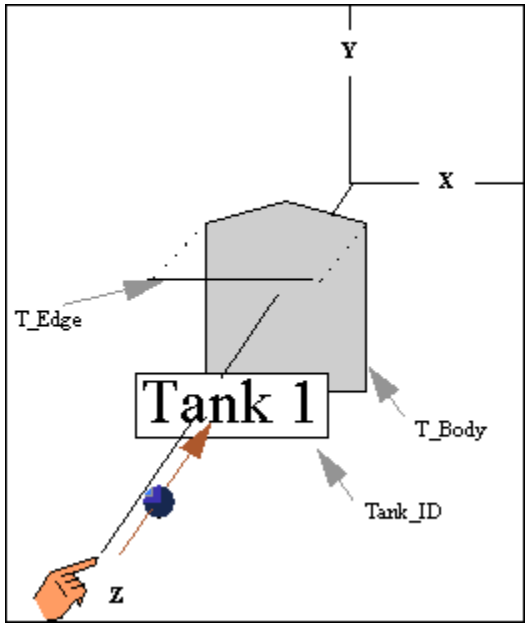


The figure below illustrates the problem in terms of the scripting model.

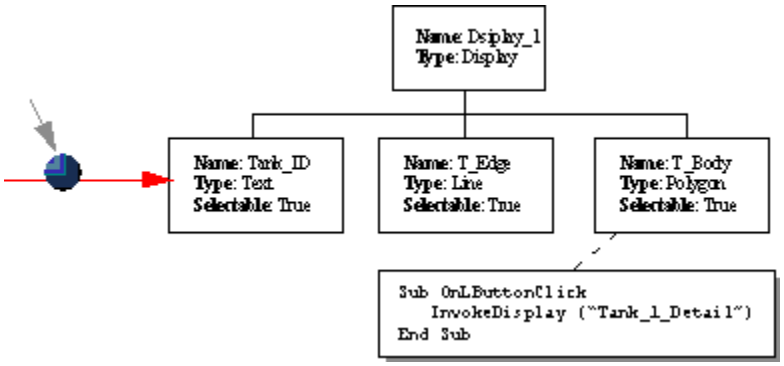
Clicking the Text object used to display the tank name generates a LButtonClick event. Because the Text object is selectable, the LButtonClick event will look for a subroutine on the Text object to execute (Sub OnLButtonClick). Because the Text object has no such subroutine (in fact it has no script at all), the LButtonClick Event stops its propagation and the result is no action.

A similar result would occur if the operator clicked on the line separating the top from the body of the tank.

User Event Stops At Tank_ID Object



Another View of User Event Propagation in Tank Display



This example illustrates the following rule:

- User events propagate along the z-axis of the display (from nearest object to farthest object), until a Selectable object is encountered.

Selectable/Enable property and propagation of user events in simple objects

A subroutine associated with an object will execute only if the object is "selectable." Objects have a property that defines if they are selectable or not. When the value of this property is true, the object is selectable; when the value of this property is false, the object is not selectable.

The name for this property on primitive Display objects (e.g., lines, rectangles, etc.) and on Groups, is ***Selectable***. The name for this property on Embedded Displays is ***Selectable***. The name for this property on OLE Controls is ***Enable***.

On the property sheets, this property is represented by a check box. A check represents the value True and an unchecked box represents the value false.

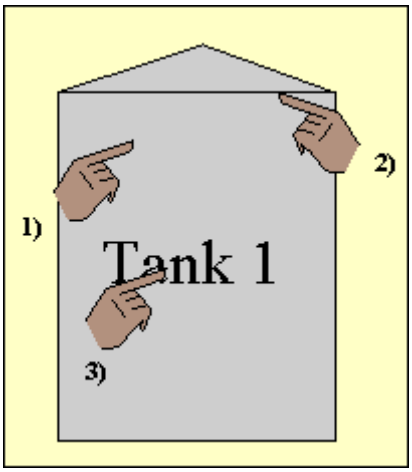
The following example illustrates the use of this property to correct the error in the Tank display.

Example: Correcting the Error in the Tank Display

While the author could replicate the script associated with the tank body (T_Body) to the scripts associated with the Tank_ID and T_Edge, doing so would increase the cost of creating and maintaining the display as well as increasing the probability of error.

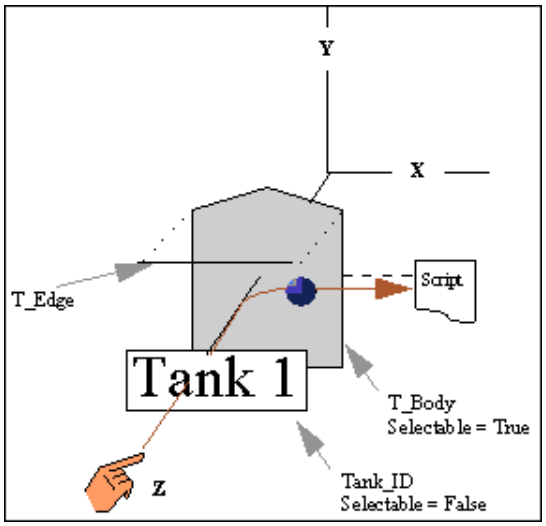
Instead, the author chose to set the value of the Selectable property for the Tank_ID and T_Edge to False. This allows the event to propagate through the objects and continue looking for a subroutine to service it:

Corrected Tank Display

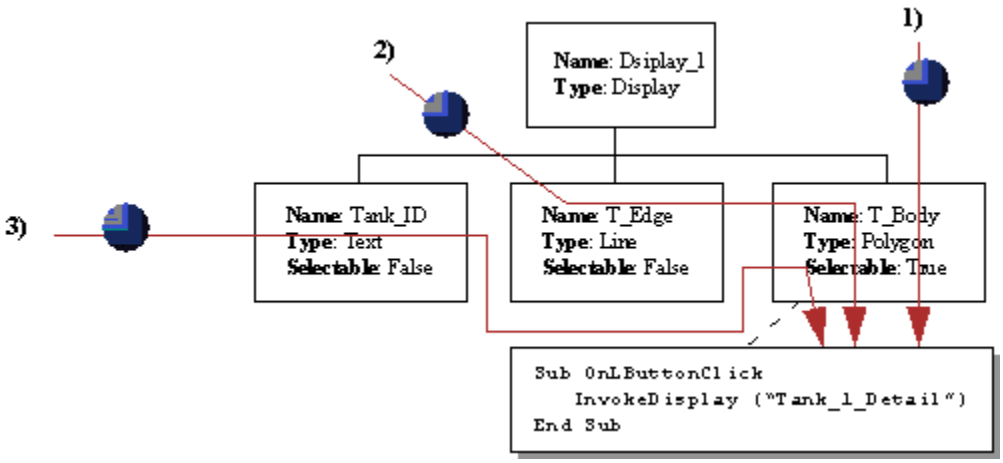


Clicking anywhere on Tank 1 will call the Tank Detail Display.

Z-Order of Objects in Corrected Tank Display



Propagation of Events in Corrected Tank Display



This example illustrates the rule:

- User events propagate along the z-axis of the display (from nearest object to farthest object), until a Selectable object is encountered.

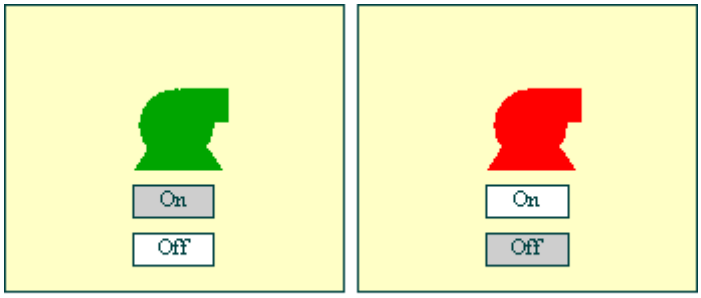
Selectable/Enable property and propagation of user events in embedded displays

Example: A Simple Pump Display

Setting the Selectable property for an Embedded Display to False, results in saving the current value of the Selectable/Enable property for all the member objects in the Embedded Display, then setting their Selectable/Enable property to False. Setting the Selectable property for an Embedded Display to True restores the previously saved value of the Selectable property for the member object.

The figures below illustrate how changing the value of the Selectable property for Embedded Displays at run time can be used.

Changing a Selectable Property at Run Time



t0: Pump On (On Pushbutton Disabled)

t0: Pump Off (Off Pushbutton Disabled)

When the pump is off, the operator can turn it on by selecting the rectangle labeled “On.” Clicking on the rectangle labeled “Off” has no effect. The Selectable property for the embedded display is set to false.

When the pump is on, the operator can turn it off by selecting the rectangle labeled “Off.” Clicking on the rectangle labeled “On” has no effect (i.e., the Selectable property for the embedded display is set to false).

Behavior of the Pump display

The panel display consists of two instances of an embedded display (named pushbutton.pct). The embedded display takes two parameters.

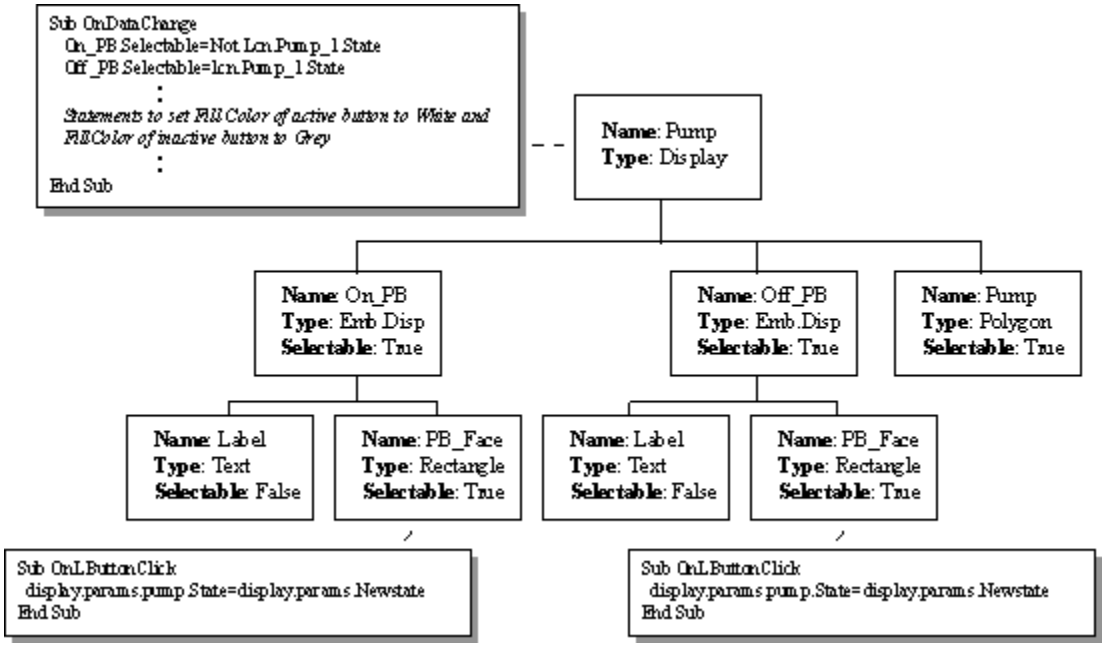
The first parameter is named Pump, and is an Entity type. When the user adds an instance of a pushbutton, the Pump parameter is bound to a point representing a pump.

The second is named NewState and is a Boolean type. When the user adds an instance of a pushbutton, the NewState parameter is bound to True, to turn the pump on, or False, to turn the pump off.

For the purpose of this example, we assume that points representing pumps have a Boolean parameter named "State." The value True means the pump is on and the value False means the pump is off. Storing True to the State parameter for a pump point will turn the pump on and storing False to the State parameter for a pump point will turn the pump off.

The figure below shows the structure of this display:

Structure of Pump Display



PushButton.pct

The embedded display PushButton.pct consists of two objects: a Text object (named Label) and a rectangle (named PB_Face).

The value of the Selectable property for Label is set to False to allow user events to pass through it. The value of the Selectable property for PB_Face is set to True so that the LButtonClick event will execute the subroutine to turn the Pump on or off.

The Pump Display

The Pump Display consists of two instances of the embedded display PushButton.pct and a polygon representing the pump.

An OnDataChange subroutine associated with the Display monitors the point.parameter that indicates if the pump is on or off.

When the pump is on, this script sets the value of the Selectable property for On_PB to False (i.e., `On_PB.selectable =False`), and the value of the Selectable property for Off_PB to True (i.e., `Off_PB.selectable =True`).

When the pump is off, this script sets the value of the Selectable property for On_PB to True (i.e., `On_PB.selectable =True`), and the value of the Selectable property for Off_PB to False (i.e., `Off_PB.selectable =False`).

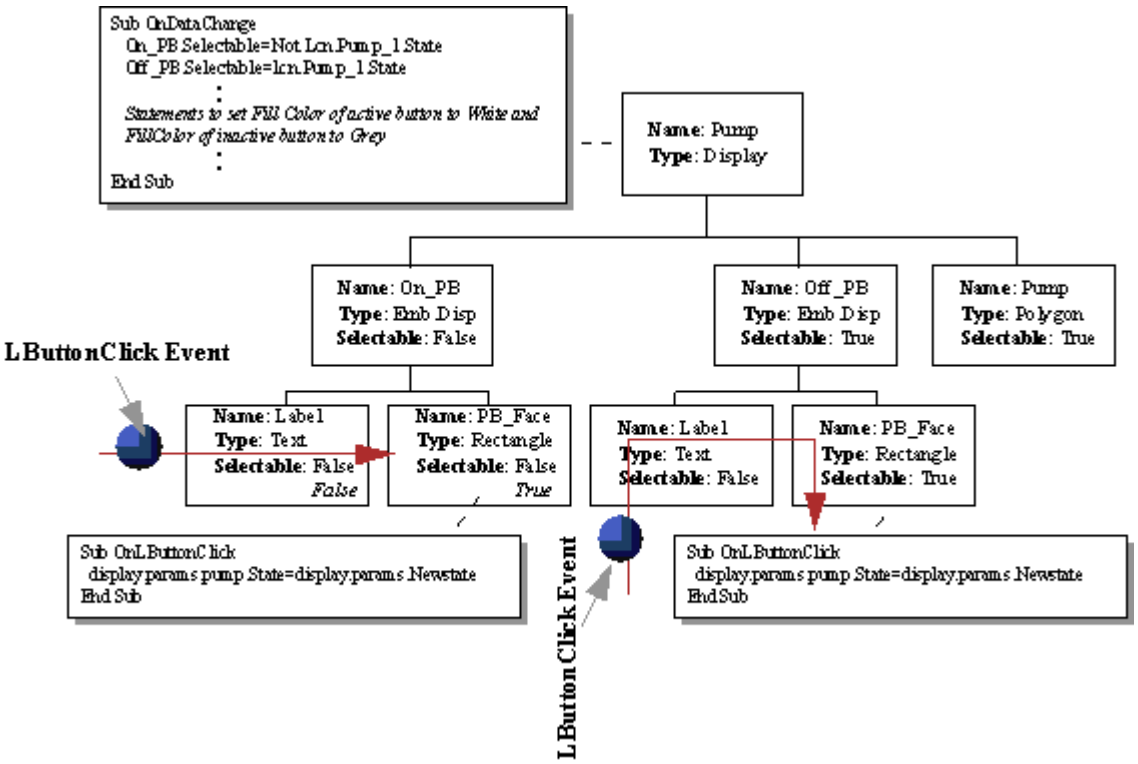
Operation of the Pump display

The figure below illustrates the Pump display when the pump is On. At this time, the OnDataChange subroutine associated with the display has set the value of the Selectable property for On_PB to False. This resulted in first saving the values of the Selectable property for each member of the embedded display (shown under the current value in italic font), then assigning False to the Selectable property for each member object and the Embedded display itself.

Because the Selectable property for the Label in the On_PB embedded display is false, a LButtonClick event passes through it. Because the Selectable property for the PB_Face in the On_PB embedded display is false, a LButtonClick event does not cause execution of the script. Because there are no objects behind the PB_Face object, the event stops.

Similarly, because the Selectable property for the Label in the Off_PB embedded display is false, a LButtonClick event passes through the Label. Because the Selectable property for the PB_Face in the Off_PB embedded display is True, a LButtonClick event results in execution of the script, which turns the pump off.

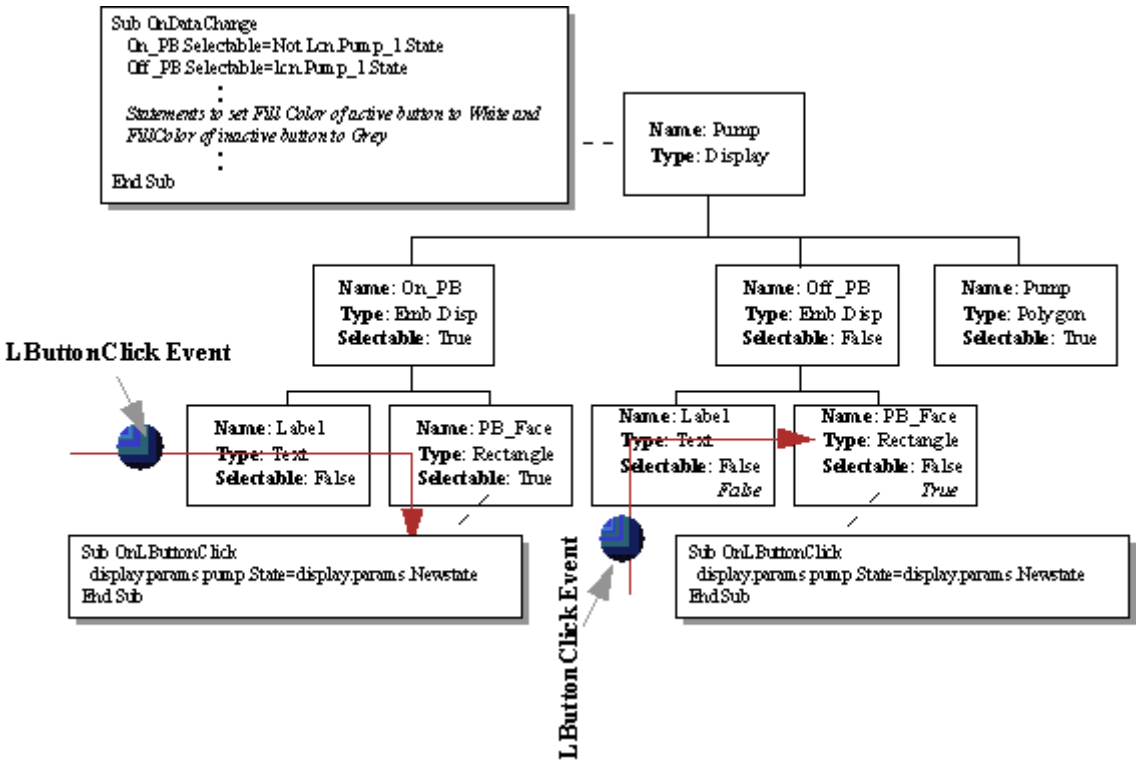
Pump Display After Assigning False to On_PB Embedded Display



Note that in the above display, the initial value of the Selectable property is shown in italics. The value then assigned to the selectable property for each member object and the display itself is shown in normal font.

The figure below illustrates the operation of the Pump display after the Selectable property for the Off_PB embedded display has been set to False.

Pump Display After Assigning True to the Selectable Property of On_PB and False to the Selectable Property of Off_PB

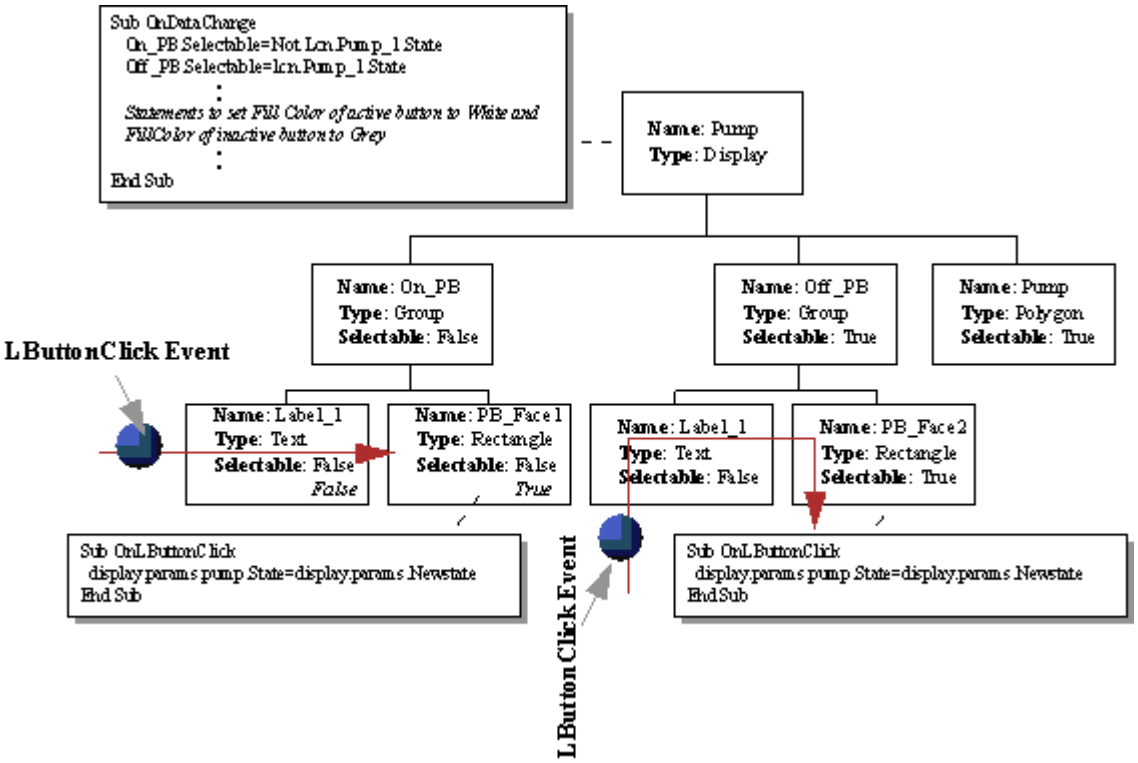


Note that in the above display, the initial value of the Selectable property is shown in italics. The value then assigned to the selectable property for each member object and the display itself is shown in normal font.

Selectable property and propagation of user events in groups

Assigning a value to the Selectable property of a Group results in the same behavior as for embedded displays. The figure below illustrates another way to implement the Pump display, where the embedded displays On_PB and Off_PB are replaced by groups.

Alternate Implementation of Pump Display Using Groups



Note that in the above display, the initial value of the Selectable property is shown in italics. The value then assigned to the selectable property for each member object and the display itself is shown in normal font.

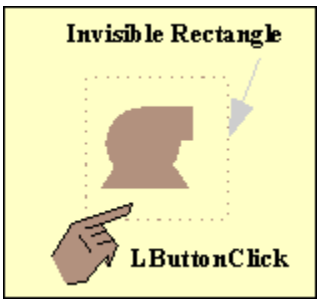
Visibility and selectivity

An object can react to user input independent of its visibility.

Example 1: Using an Invisible Rectangle to Simplify Selection

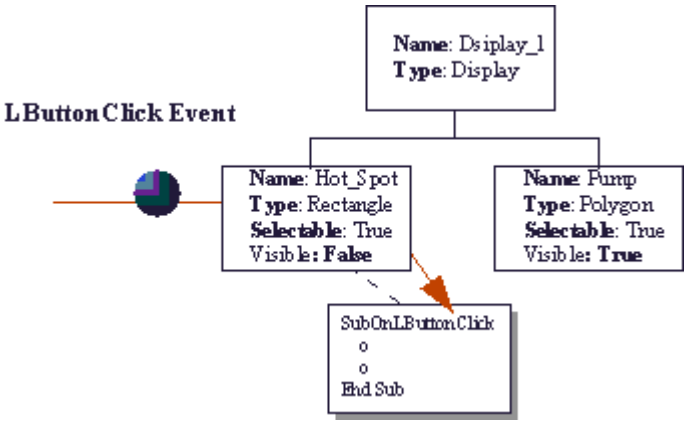
The author of the display in the figure below placed an invisible rectangle over the pump and scripted the rectangle to respond to LButtonClick events. This allows the operator to invoke the detailed display for the pump merely by selecting an area close to the pump.

Invisible Rectangle



The invisible rectangle on the top of pump allows easy selection by the operator.

Display Structure

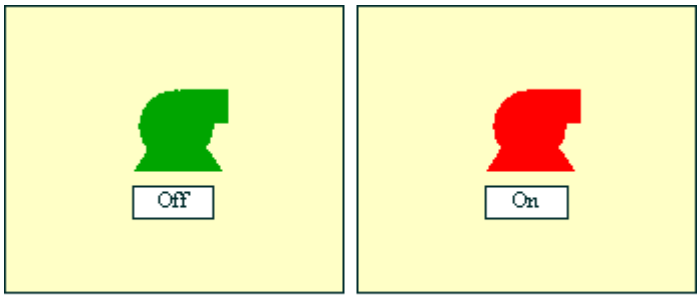


Example 2: Making Objects Conditionally Visible and Selectable

The figure below illustrates a refinement to the Pump display discussed earlier. In this version, the operator sees at most one pushbutton. When the pump is on, the operator sees the pushbutton to turn it off. When the pump is off, the operator sees the pushbutton to turn it on.

The refinement was made by placing On_PB directly over Off_PB and adding a couple of lines to the OnDataChange subroutine to alternate the visibility of these embedded displays.

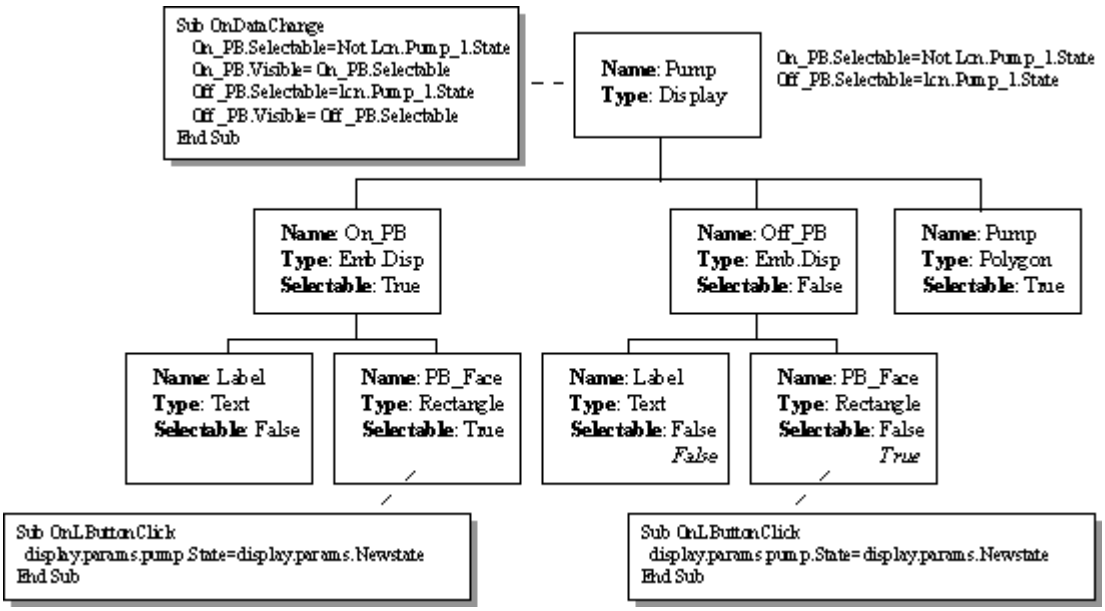
Refinement to Pump Display



t0: Pump On (Off Pushbutton Appears)

t0: Pump Off (On Pushbutton Appears)

Refined Pump Display Structure



Note that in the above display, the initial value of the Selectable property is shown in italics. The value then assigned to the selectable property for each member object and the display itself is shown in normal font.

Scripting Threads

Introduction

Subroutines associated with Display objects can execute concurrently with other subroutines in the display. Concurrent execution is best described in terms of the notion of a ***Thread of execution*** (or just thread).

Conceptually a thread of execution is a sequence of executed scripting statements. All Statements executed by a single thread are executed serially—i.e., first one, then the next, and so on.

Scripting statements executed by two different threads can execute concurrently. Perhaps more importantly, any delay in the execution of statements on one thread does not delay the other.

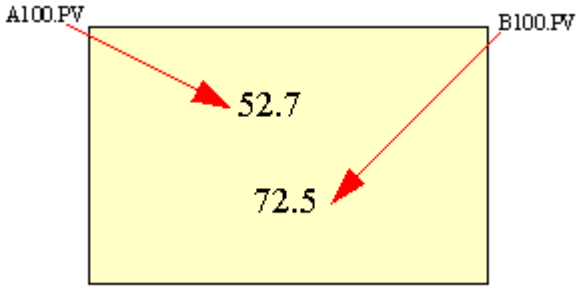
All subroutines in a display are executed by one of three threads, described next.

The Data Change Thread

The Data Change thread executes all of the OnDataChange subroutines for a display. Because a single thread executes each of the OnDataChange subroutines for all objects in a display, the subroutines are executed serially. The GUS Run Time does not guarantee the order in which the OnDataChange subroutines are executed.

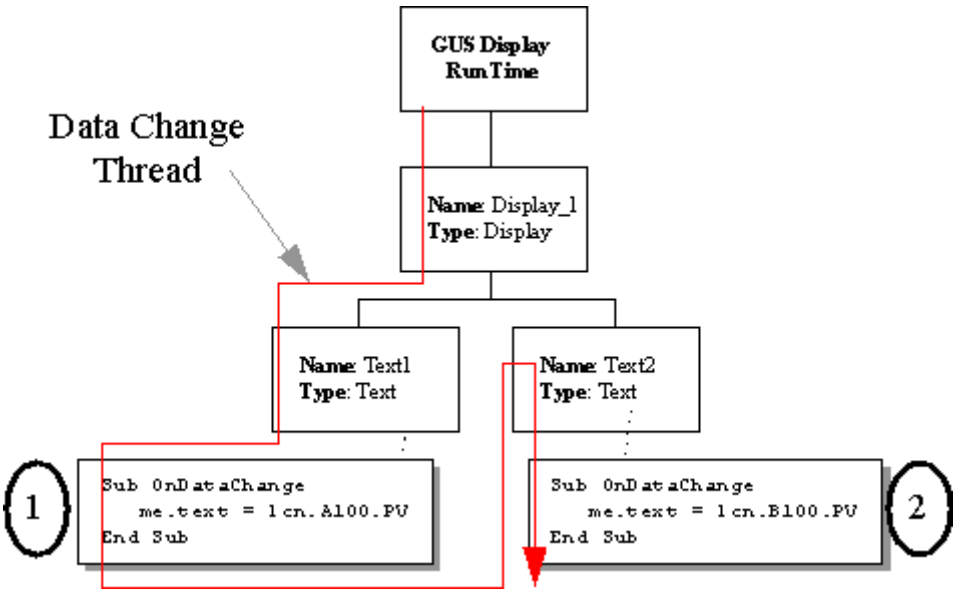
The figure below illustrates a display that shows the values of A100.PV and B100.PV.

Two Point Values



The figure below illustrates the structure of this display and shows one possible order of execution of its OnDataChange subroutines on the Data Change thread for the display.

Display Structure After Execution



The Periodic Update Thread

The Periodic Update thread executes all of the OnPeriodicUpdate subroutines for a display.

Because a single thread executes each of the OnPeriodicUpdate subroutines for all objects in a display, the subroutines are executed serially. The GUS Run Time does not guarantee the order in which the OnPeriodicUpdate subroutines are executed.

The User Interface Thread

The User Interface thread executes all user interface subroutines (e.g., the OnLButtonClick, OnRButtonClick, ... , etc. subroutines) as well as all subroutines for OLE events and the DisplayStartUp and DisplayShutDown events.

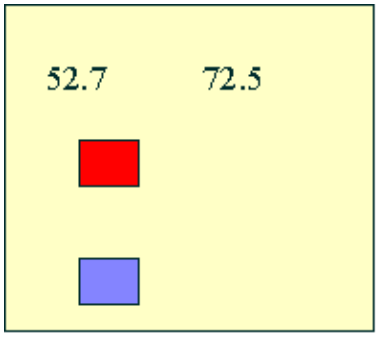
Because a single thread executes each of these subroutines for all objects in a display, the subroutines are executed serially. More importantly, the suspension of this thread will not affect the execution of the other threads in a display.

For example, an object in a display can have an OnLButtonClick subroutine scripted to show a dialog box and await input from the operator. During the time the dialog box is visible, the User Interface thread is suspended. When the user signals completion of input (e.g., clicks on the “OK” button) the User Interface thread resumes and the subroutine continues execution.

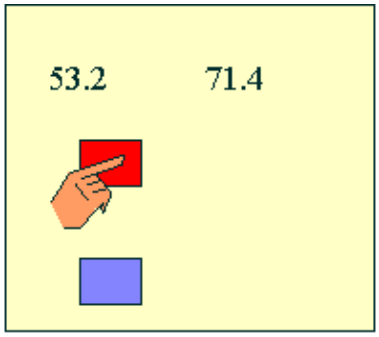
Because other threads execute the OnDataChange and OnPeriodicUpdate subroutines, they will execute even though the User Interface thread is suspended, and the display will continue to update.

The figures below illustrate what happens as a result of three operator clicks:

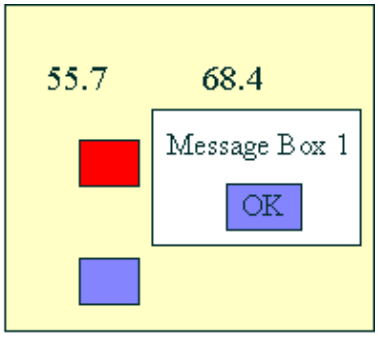
Display Structures After 3 Operator Clicks



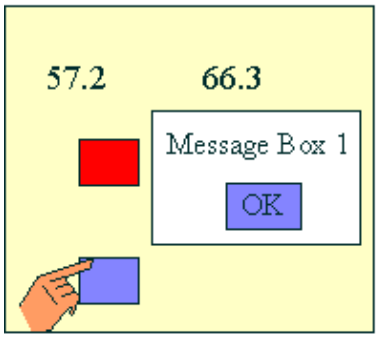
Time t0:Display Updating.



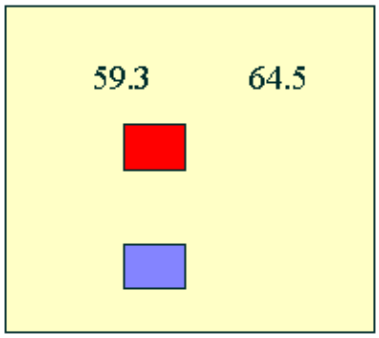
Time t1:Operator Clicks On Rectangle. OnLButtonClick subroutine Displays Message Box; Display Continues To Update.



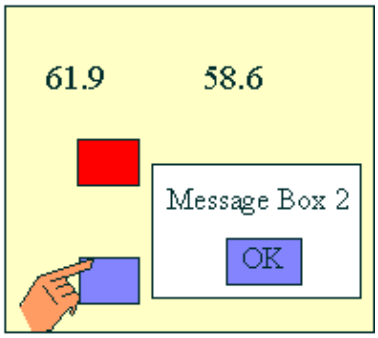
Time t2:Display Continues To Update During Time Message Box. User Interface Thread is suspended.



Time t3:Operator Clicks On Another Rectangle Scripted With An OnLButtonClick Subroutine. The Subroutine does not execute because the User Interface Thread is suspended in the OnLButtonClick Routine for the Top Rectangle.



Time t4:Operator dismisses first Message Box and User Interface Thread continues to execute.



Time t5:Because the User Interface Thread is free, a click on the lower rectangle brings up its message box. Note: the display continued to update throughout the entire interaction.

Threads and Data Access

Writing to DCS data

All writes to DCS data from any script are immediate—the write occurs directly to the DCS point.parameter and script execution is suspended until the write completes.

Reading from DCS data

All reads from DCS point.parameters from any script executed by the User Interface thread are immediate. The read occurs directly from the DCS point.parameter and script execution is suspended until the read completes.

All reads from DCS point.parameters from any script executed by the Data Change or Periodic Update thread, ***read from the data cache***, return the value of the DCS point.parameter at the time of the last scan.

User Interface Guidelines

Introduction

The purpose of this section is as follows:

- to provide general guidelines for creating performant and useful user interfaces using GUS displays and SafeView workspaces, and
- to provide a means of estimating the performance of a user interface built with GUS displays

Performance and LCN Loading

To aid the estimation process, the spectrum of user interface styles has been partitioned into four segments, each representing a commonly used interface style. Because users author GUS displays, there can be a wide range of performance, depending on the particular authoring style, LCN system loading, and other factors.

The table below lists the display callup performance for each of the four user interface styles. Following the table, each of the user interface styles are characterized:

GUS User Interface Style	Display Callup Performance
US Replacement	Native Window: equivalent to TDC
Basic Monitoring and Control	GUS Display: 2 - 5 seconds Native Window: equivalent to TDC
Multi-Level Monitoring and Control	Main Display: 2 - 5 seconds Mid-level Detail: 1 - 2 seconds Point Viewer: 1 - 1.5 seconds Native Window: equivalent to TDC
Wide Area Monitoring and Control	Wide Area Display: 5 - 7 seconds Mid-level Detail: 2 - 5 seconds Low-level Detail: 1 - 2 seconds Point Viewer: 1.-1.5 seconds Native Window: equivalent to TDC

GUS User Interface Styles

US replacement style

This style of user interface uses a GUS to emulate a US using the GUS Native Window.

User Interface Structure

This interface consists of a single screen CPU running a Native Window that is used for:

- Process alarms
- System alarms
- System management information (Console Status, Node Status, etc)
- Group displays
- Detail displays
- Classic schematics

Basic monitoring and control style

This style of user interface uses GUS displays instead of the “classic” TDC schematics. Only one GUS display is active at any time. The GUS Native Window provides access to alarms, system management functions, and classic Group or Detail displays.

User Interface Structure

This interface consists of a single Screen CPU with a single GUS display window that is used for:

- One GUS display containing a Change Zone or other point viewing capability that can reference 200 to 400 parameters
- Native Window, which is used for:
  - Process alarms
  - System alarms
  - System management information (Console Status, Node Status, etc)
  - Group and Detail displays
  - Classic schematics

Multi-level monitoring and control style

This style of user interface uses GUS displays instead of the classic TDC schematics. More than one GUS display can be active at any time. A “Main” display provides the ability to monitor 200 to 400 parameters. Other “mid-level Detail” displays provide a view to 20 to 25 parameters. One to two “Point Viewer” displays (Change Zones) provide the ability to monitor and change the important parameters of two points.

The GUS Native window provides access to alarms, system management functions, and classic Group or Detail displays.

User Interface Structure

This interface consists of a single screen CPU that is used to run several GUS displays such as:

- A Main display of 200 to 400 parameters
- A mid-level Detail display of 20 to 25 parameters
- Up to two Point Viewer displays (Change Zones), displaying 2 to 5 parameters
- A Native Window, that can be used for:
  - Process alarms
  - System alarms
  - System management (Console Status, Node Status, etc)
  - Group and Detail displays
  - Classic schematics

Wide area monitoring and control style

This style of user interface builds on the Multi-Level Monitoring and Control style. Several GUS displays are used and more than one GUS display can be active at one time. A “Wide Area” display provides the ability to monitor 400 to 700 parameters. A “Mid-level Detail” display provides a view to 200 to 400 parameters. A “Low-level Detail” display provides a view to 20 to 25 parameters. A “Point Viewer” display (Change Zones) provides the ability to monitor and change the important parameters of two points.

The GUS Native window provides access to alarms, system management functions, and classics Group or Detail displays.

User Interface Structure

This interface consists of a ***dual*** screen CPU that is used to run several GUS displays such as:

- A Wide Area Overview display of 400 to 700 parameters
- A Mid-level Detail display referencing 200 to 400 parameters showing a “close-up” of a portion of the Wide Area Overview
- A Low-level Detail display of 20 to 25 parameters
- Up to two Point Viewer displays (“Change Zones”) displaying 2 to 5 parameters
- A Native Window, that can be used for:
  - Process alarms
  - System alarms
  - System management (Console Status, Node Status, etc)
  - Group, Detail displays
  - Classic schematics

Point Viewer Displays

Introduction

The Point Viewer display referenced in the previous section is a small display that shows the key parameters for a particular point. This section outlines an approach for creating such displays.

General approach

The general approach is very simple:

- Create a Point Viewer display for each type of LCN point. The point of interest for one of these displays is passed to the display using a Global Display Data Base variable. The display for a particular point type references the appropriate parameters for that point type indirectly, through the DDB.
- The invoking script determines the type of the point and invokes the display to handle that point, passing the internal ID of the point in a Global Display Data Base variable.

For example, assume that you have a naming convention for Point Viewer displays such that the display that handles a particular LCN point type includes the name of the point type in the name of the display. For instance, a display that handles HG Analog Input point types could be named PVD_ANLINHG.PCT; a display that handles HG Analog Output point types could be named PVD_ANLOUTHG.PCT, and so on.

Scripts used to invoke a Point Viewer display would:

- Determine the type of the point to be viewed (called the “point of interest”).
- Store the internal ID of the point of interest in a Global DDB.
- Convert that type into the name of the appropriate display.
- Invoke the display.

Determine the LCN point type

The key to invoking a Point Viewer display for a particular LCN point type quickly is to be able to determine the point type quickly.

The following script fragment illustrates this technique by calling a Point Viewer display for the point A100. This script fragment assumes a public function called “lcnPointType,” which extracts the point type from the internal ID stored in dispdb.ENT01G and returns it as a string. A later section outlines a technique for creating such a function.

Sample Script

```
Sub OnLButtonClick
   dim pntType as string
   ’ Pass point to Point Viewer display
   Set Dispdb.ENTO1G = lcn.A100
   ’ Get the name of the LCN point type for point in
   ENTO1G
   pntType =lcnPointType ()
   ’ Invoke the Point Viewer display for that type
   InvokeDisplay (“PVD_” + pntType + “.PCT”)
End Sub
```

The key to this technique is determining the type of the point of interest. One way to do this is to use the value of the property ent_type. This property returns the type of the point as a string.
The script fragment below contains an example.

Sample Script

```
.
.
type = lcn.A100.ent_type
.
.
```

While this statement will return an indication of the point type, it can take up to 0.5 seconds to execute, compromising call up time of the Point Viewer display. A faster way to obtain the LCN point type is to extract it directly from the internal point ID, avoiding any access to LCN resident Data Owners. Because the LCN point type in the internal ID is represented as an integer, the string form of the point type can be obtained using an array, indexed by the point type obtained from the internal ID.

The following script will extract the integer representation of an LCN point type from an internal ID of an LCN point in dispdb.ENT01G.

Sample Script

```
Public Function lcnEntType () As Integer
   Dim crackEntType As String *
   ’ Put the internal ID in the String
   crackEntType = dispdb.ENT01G.internal
   ’ Extract the byte containing the LCN TYPE and return
   it
   lcnEntType = ascb (mid(crackEntType, 1, 1))
End Function
```

The table below maps the integer type to the string equivalent:

Integer	String
0	NULL
1	POCSTAT
2	HIWAY
3	BOX
4	LIBRARY
5	ANLINHG
6	ANLOUTHG
7	ANLCMPHG
8	DIGINHG
9	DIGOUTHG
10	DIGCMPHG
11	CTLCOUNT
12	REGHG
13	COUNTHG
14	TIMERHG
15	FLAGHG
16	NUMERCHG
17	PRCMODHG
18	LGKBLKHG
19	UNT_ENBL
20	REGAM
21	LOGICAM
22	CUSTOMAM
23	SWITCHAM

Integer	String
24	COUNTAM
25	TIMERAM
26	FLAGAM
27	NUMERCAM
28	BCHHISAM
29	BCHHISPR
30 – 58	Internal use by LCN
59	ACIDP
60	ACCRDP
61	HM_HIST
62	UPIDP
63	UPCRDP
64 - 66	Internal use by LCN
67	BHASMDHG
68 – 70	Internal use by LCN
71	NODE
72	MGLIB
73	ANLINMG
74	ANLOUTMG
75	DIGINMG
76	DIGOUTMG
77	DIGCMPMG
78	REGPVMG
79	REGCTLMG
80	LOGICMG

Integer	String
81	Internal use by LCN
82	FLAGMG
83	NUMERMG
84	TIMERMG
85	PRCMODMG
86	ANLINIPC
87	ANLOTIPC
88	DIGINIPC
89	DIGOTIPC
90	REGPVIPC
91	DIGIPC
92	SPR05IPC
93	SPR04IPC
94	SPR03IPC
95	SPR02IPC
96	SPR01IPC
97	AMG
98 - 104	Internal use by LCN
105	PMBOX
106 – 118	Internal use by LCN
119	ARRMG
120	RSV_ARRMG

Integer	String
121	DEVCTL
122	RSV_DEVCTL
123	CMMG
124	RSV_CMMG

Sample Script

Following is a scripting example that demonstrates some of the performance optimization techniques.

```
'This code is attached to a filled Text object. The display
'has three inline parameters:
'  point [ex) lcn.a100 ]
'  param [ex) PV ]
'  format [ex) ##0.00 ] #'s for digits,
'  0's for placeholders
'
'Inline parameters use the following:
'  \pe()  gives lcn name form  [lcn.a100 becomes a100]
'  \p()   allows use within strings
'  otherwise the parameter is directly substituted
'
Sub OnDataChange()
    '  Store ackstat in a local variable because it is
    '  referenced repeatedly
    dim almstat as string
    'Use an error handler instead of reading .status
    'explicitly
    On Error Goto ODC_error
    'Make sure format isn't empty
    If "-\p(format)-" = "--" Then
       fmt = "###0.0"
    else
       fmt ="\p(format)"
    end If

'Check the alarm status of the point
almstat = collector("ackstat(\pe(point))")

if almstat = "NOALARM" Then
    me.blink = false
    me.visible = true

    'Implement color scheme
    If UCase$("\p(param)") = "OP" Then
       me.textcolor = makecolor(253,253,128)
```

```
        me.fillcolor = makecolor(0,0,0)
    Else
        me.textcolor = makecolor(0,255,255)
        me.fillcolor = makecolor(0,0,0)
    End If
elseif almstat = "UNAKALRM" Then
    me.textcolor = makecolor(0,255,255)
    me.fillcolor = tdc_red
    me.blink = true
elseif almstat = "AKDALRM" Then
    me.textcolor = makecolor(0,255,255)
    me.fillcolor = tdc_red
    me.blink = false
    me.visible = true
end if
me.text = Format$(point.param,fmt)

Exit Sub
ODC_error:
    'Change contents based on error number
    select case err.number
        case HOPC_COMMUNICATION_ERROR
            Me.text = "@@@@@@"
        case else
    'Use this for debugging
    'Msgbox Err.Description + " Error Number is "
    '+CStr(Err.Number)
        me.text = "---"
        me.fillcolor = makecolor(226,0,255)
        me.textcolor = makecolor(0,0,0)
    end select
End Sub
```

Implementing a Display That References HCI Data

Background

Consider the design approach presented in this section to implement an embedded display that references HCI data. HCI server names are global in a display. Therefore, if embedded display A binds HCI.Server1 to “ServerFred” in its startup script, and embedded display B binds HCI.Server1 to “ServerSam” in its startup script, HCI.Server1 will be bound to the server referenced in the last script run. In other words, if the startup script of embedded display A runs last, then HCI.Server1 will be bound to “ServerFred”, and embedded display B will also be bound to “ServerFred” and display data accessed from “ServerFred.”

Design Approach

The following design approach allows the reuse of an embedded display whose instances are bound to different servers.

Functional description of the embedded display

This embedded display shows a point.parameter value accessed from an HCI server.

Designing and implementing an embedded display

Step	Action
1	Define the display parameters shown in the following table:

Display Parameter Name	Data Type
Server	inline
Point	inline
Parameter	inline

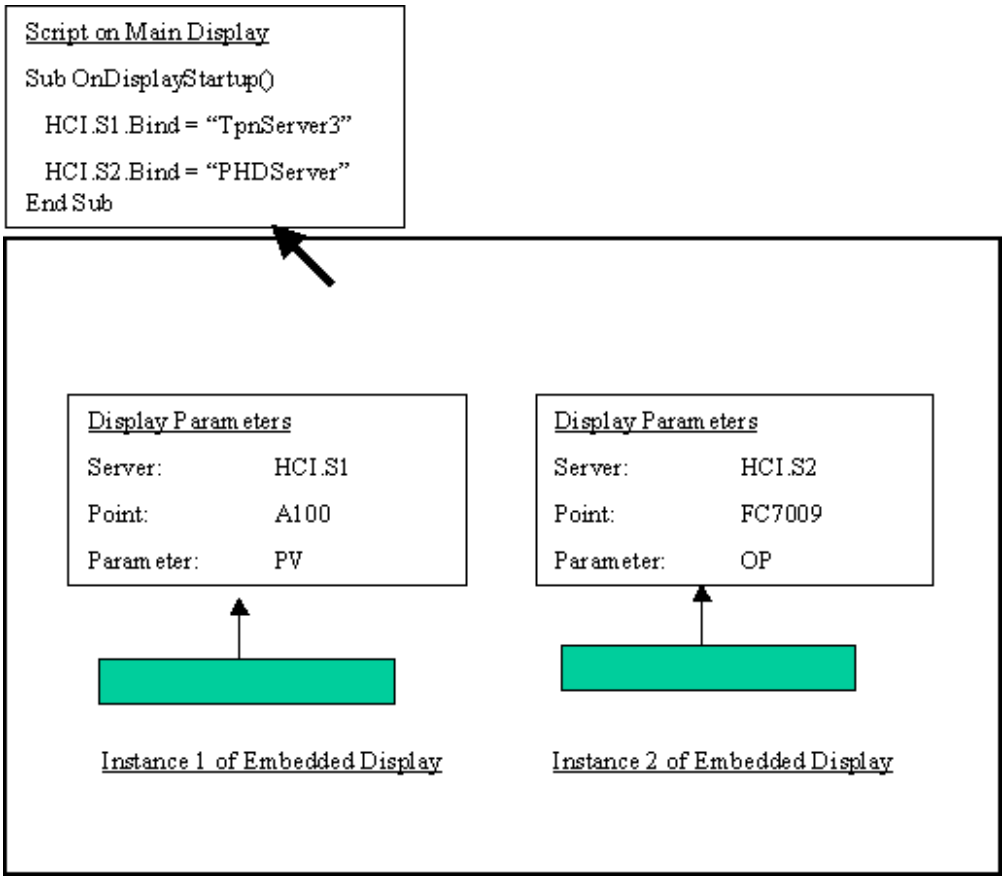
An example of the text object in the embedded display is as follows:

Sample Script

```
Sub OnDataChange()
   'add formatting here if necessary
   Me.text = server.point.parameter
End Sub
```

Reusing the embedded display in a display

Scripting and Display Structure for Reusing an Embedded Display



Developing ActiveX Controls

Configuring Your Control's Project Properties

Provide a user-friendly description for your component

Provide a user-friendly description for your component by entering an appropriate Project Name and Project Description (using the General tab in the Project Properties window for your project, under the VB5 Project menu). GUS currently creates a user-visible name for your control as <Project Name><dot><Control Name> but, in the future, GUS will use the Project Description. The defaults for these are Project1 and UserControl1, so the default entry for your control in the GUS Insert/Control list of controls would be **Project1.UserControl1**. The Project Description will also be displayed in your VB5 Components list.

You can change the name of your control from, say UserControl1, by changing the Name field in the Properties list for this item, not on the Project Properties window. For example, to create a name such as ACMEComponents.AnimatedFeedPump, set the Project Name to "ACMEComponents" and the Control's Name property to "AnimatedFeedPump."

Avoid duplicating your control names

Do not duplicate the names of your controls (using the Project Name and Project Description fields) because they appear as duplicate entries in the list of controls and users will not be able to differentiate between them.

Configure your project for binary compatibility

Indicate that you want binary compatibility for your project by selecting the Binary Compatibility option (using the Component tab in the Project Properties window for your project, under the VB5 Project menu). This guarantees that subsequent versions of your control will "replace" the previous ones.

ATTENTION

You can only select this option after the project has been built. If this option is not set, then future builds of this control will be given a new, different "globally unique identifier." This means that essentially it is a new and different control (VB rewrites this number, but tracks it internally so it knows about the new GUID). If this occurs, then previous GUS displays that have included this control will not load correctly, because the ID of the previous display for this control will no longer be valid.

Note also that the name of the .ocx must match that of the actual .ocx being made as determined by the VB5 File/Make <your Project.ocx> menu item.

It is not necessary to replace or revalidate components

GUS displays automatically utilize updated versions of components built with Binary Compatibility. Therefore, there is no need to "replace" these components or revalidate the displays.

Resizing ActiveX Controls

To accommodate differing screen resolutions as well as dynamic "zoom to fit" automatic resizing, GUS displays are normally scaled. Native GUS picture components are scaled automatically by GUS, but because ActiveX controls always render themselves, they must scale themselves in response to GUS "Resize" method calls. Failure to respond to this request will result in the control either (1) showing too much of itself, or (2) being clipped.

Use UserControl_Resize( ) to make ActiveX controls scale themselves

By default, your VB5 ActiveX control is named, "UserControl," in which case you would provide the "UserControl_Resize( )" method for the control. This is called by the GUS display each time the container (the GUS display) deems it appropriate to inform the control of its new size, for example, when the display is resized. Your method can directly reference its Width and Height properties at this time to obtain this size. It is incumbent upon the control to properly resize its internal components based upon the given width and height.

Do **NOT** adjust your control's size by setting the individual Width or Height properties. Rather, use the Size method as shown in Example 2 below. The former approach is not currently supported.

Example 1

This example demonstrates the resizing of a shape, Shape1, such that it is always appropriately centered within its control as the control resizes. This example uses the Move method.

Sample Script

```
Private Sub UserControl_Resize()
   Shape1.Move Width / 4, Height / 4, Width / 2, Height / 2
End Sub
```

Example 2

This example demonstrates the constraining of a control to a square aspect ratio. You can modify this example to constrain a control such that the control will not resize at all. This may limit the usefulness of the control in scaleable displays.

Sample Script

```
Private Sub UserControl_Resize()
   Dim X, Y As Integer
   X = UserControl.Width
   Y = UserControl.Height
   'Enforce 1-1 Aspect Ratio
   If X < Y then
        X = Y
   ElseIf Y < X Then
       Y = X
   End If
   UserControl.Size X, Y
End Sub
```

Resizing Text

Change the font of your control's text upon Resize to keep it proportional

Microsoft provides a free, useful text-control, called "FlexLabel," which demonstrates text scaling, and can also be used to provide scaleable text within your own controls. Consider using FlexLabel control as a constituent component of your control for scaleable text.

ATTENTION

FlexLabel requires your control to be single-threaded (see your control's Project Properties).

Note also that although possible, it is never necessary to use FlexLabel directly in a GUS display, since GUS native Text objects automatically scale themselves.

Using Ambient and Extender Object Properties Wisely

Do not access Ambient and Extender object property too soon

Visual Basic often includes access to a control's client site (such as, the container) in the default implementations for controls, typically in the UserControl_ReadProperties and/or UserControl_InitProperties events. This is accomplished by accessing the Ambient or Extender object properties. In the current GUS display runtime environment, the control's client site is not available to the control until after these two events have fired. Accessing Ambient or Extender object properties will fail at GUS display load time and produce a "Client Site Not Available" error message.

Delay access of any Ambient or Extender object properties until after the ReadProperties or InitProperties events are fired. A good alternative is to access these properties later, as shown in the UserControl_Show example that follows.

Accessing Ambient or Extender object properties too soon

Sample Script

```
'Load property values from storage
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    m_BackColor = PropBag.ReadProperty("BackColor",
    m_def_BackColor)
    m_ForeColor = PropBag.ReadProperty("ForeColor",
    m_def_ForeColor)
    m_Enabled = PropBag.ReadProperty("Enabled", m_def_Enabled)
    'Set m_Font = PropBag.ReadProperty("Font", Ambient.Font)
    Don't access Ambient object this early!
    m_BackStyle = PropBag.ReadProperty("BackStyle",
    m_def_BackStyle)
    m_BorderStyle = PropBag.ReadProperty("BorderStyle",
    m_def_BorderStyle)
'Caption = PropBag.ReadProperty("Caption", Extender.Name)
'Don't access Extender this early either!
End Sub
```

Using UserControl_Show to access to Ambient object property

Sample Script

```
Private Sub UserControl_Show()
   Font = Ambient.Font
   BackColor = Ambient.BackColor
End Sub
```

Using Ambient and Extender object properties to enhance your control

Use the Ambient object property to make your control track related container properties such as BackColor and Font. Use the Extender object property such as Name.

Provide a UserControl_AmbientChanged event for your control, so it can track changes to changes to its container's ambient properties.

Using Ambient and Extender objects

This example uses the Ambient object to set the font for FlexLabel1 and the background for the control itself. It also uses the Extender.Name property to set the text of FlexLabel1 to the name of the control in the GUS display (as shown in the Edit/OLE Object Name dialog).

Sample Script

```
Private Sub UserControl_Show()
   Set Font = Ambient.Font
   FlexLabel1.Font = Ambient.Font
   BackColor = Ambient.BackColor
   FlexLabel1.Caption = Extender.Name
End Sub
```

Other Ambient Properties

- UserMode (false for buildtime, true for runtime)
- LocaleID - for control internationalization
- DisplayName (always “GPB” in GUS displays)
- ForeColor
- TextAlign

Other Extender Properties

- Visible - whether the GUS component is visible

Creating a Properties Page

There are two compelling reasons why you should create a property page:

- to enable GUS display developers to access your control's build-time properties
- to provide a means to change the Name of your control through its Properties Page

Use Visual Basic Properties Page Wizard

Visual Basic provides a Wizard for creating a Property Page for your control. Use this to provide GUS display developers access to your control’s build-time properties. The Property Page Wizard is an add-in that is accessed using the Add-Ins menu. You may need to first access the Add-In Manager to expose this Wizard. If no property page is provided, GUS still provides a means to access and change the name of your control for that display, through the GUS Edit/OLE Object Name menu option.

Make sure that GUS-scriptable properties are set as runtime-accessible.

Documenting Your ActiveX Control's Components

Tooltips are automatically provided for your control's components when you set the ToolTipText property for these components. These are provided automatically for buildtime properties displayed in the Properties dialog, as well as specific components during runtime. Use tool tips to document the function of your control's buildtime parameters and its runtime-visible components.

Testing Your Controls

Test your constituent controls in GUS before you use them as constituents in your own controls. Some third party controls do not operate correctly in the current GUS environment. This will help you avoid problems that may not be apparent in the Visual Basic environment.

Using PictureClip to Enhance Animation Performance

Visual Basic allows you to use the Microsoft PictureClip component, which provides a means for graphical animation with minimal CPU impact. The PictureClip uses a bitmap graphic comprising a series of equal-sized images which are "clipped" into a separate, animated display graphic. It provides high-performance graphics animation.

Accessing Servers Directly from GUS Scripts

In R120 and later, you can create objects in any scripts and use the objects from any other script. So you do not need to utilize an ActiveX control to gain access to a server object, but rather you can use your server object directly from scripting.

Point Manipulation Keys on the IKB

Introduction

The Point Manipulation Key (PMK) Object is built into all GUS displays. Through the use of scripting, you can create GUS displays that will handle Point Manipulation Key presses (SP, OUT, Raise, Lower, FastRaise, FastLower, MAN, AUTO, NORM, and Enter) from the Integrated Keyboard (IKB) and other Honeywell Engineering keyboards.

The PMK object receives all key presses when a GUS display window has focus. When a PMK is pressed, the PMK object sends an event corresponding to the particular key pressed (for example, pressing the MAN key causes the PMK object to send an OnPMKMan event). The Display object specified as the PMK event handler receives the event.

ATTENTION

The PMK object does not send a key event if the Display object currently defined as the PMK event handler does not have an event handler for the particular key that was pressed. For example, if a Display object called Text1 is the PMK event handler, but does not contain the OnPMKMan sub in its scripting, the PMK object will not generate an event when the MAN key is pressed.

This section provides a detailed example scripting the PMK in a display. Refer to the *Display Scripting User's Guide* for a detailed list and description of PMK Object properties, scriptable events, functions, and methods.

CAUTION	In a display containing the Honeywell Standard ChangeZone (changezone_G120_0.pct), do not set dispdb.[$cz_enty] in the script of an object that uses the PMK object.

You should only set dispdb.[$cz_enty] for Display objects that (1) are designed to invoke the ChangeZone for a specific point and (2) do not use the PMK object should set dispdb.[$cz_enty]. Whenever a valid entity ID is stored in dispdb.[$cz_enty], ChangeZone will make itself the PMK event handler, PMK error handler, and set some of the PMK properties to non-default values.

Enable the PMK

Enable the PMK by setting its entity, eventhandler, and errorhandler properties.

PMK.entity

Set the PMK "entity" property, (for example, PMK.entity), to a valid entity to enable the PMK keys.

PMK.eventhandler

Set the PMK "eventhandler" property, (for example, PMK.eventhandler), to a main Display object or an object in one of its embedded displays in order to enable PMK functionality under certain conditions. Setting this property is optional.

ATTENTION

The display itself or an embedded display can be the eventhandler.

Under certain conditions it is not necessary to set the PMK eventhandler property to enable PMK functionality. When certain PMK properties are set to TRUE, the PMK object will automatically change a parameter value and force GUS Runtime to generate OnDataChange events to display the new value. A set of rules explaining when the eventhandler is not needed follows.

Parameter	Conditions Under Which PMK.eventhandler May Be Null
SP analog value	if pmk.AutoUpdate = True, pmk.AutoDataChange = True AND pmk.AutoWrite = True
OP analog value	if pmk.AutoUpdate = True, pmk.AutoDataChange = True AND pmk.AutoWrite = True
SP digital value	PMK.eventhandler must always be set because you use the ramp keys to "walk" the list of valid values. Provide a listbox to enumerate the valid values and script the rampkey events to "walk" the list.
OP digital value	PMK.eventhandler must always be set because you use the ramp keys to "walk" the list of valid values. Provide a listbox to enumerate the valid values and script the rampkey events to "walk" the list.
MAN, AUTO, NORM	If EnableMode = True and you do not want to "walk" the list of valid modes using the ramp keys.

PMK.errorhandler

Set the PMK "errorhandler" property (for example, PMK.errorhandler) to a main Display object or an object in one of its embedded displays to enable error handling. Because there is no default error handling done by the PMK object, failure to set the errorhandler results in the PMK object "throwing away" all errors generated by PMK functionality. This includes errors from the LCN generated during the ramping of a point.parameter or changing the mode. It is, therefore, strongly recommended that you always set the PMK errorhandler property.

ATTENTION

The display itself or an embedded display can be the eventhandler. The errorhandler does not have to be the same object as the eventhandler.

Scope of the eventhandler and the errorhandler

Be aware that the scope of the eventhandler and the errorhandler is the display in which it is contained.

If the eventhandler or the errorhandler is an object in an embedded display, it can only reference the objects in its embedded display. An eventhandler or errorhandler in the main display can only reference objects in the main display.

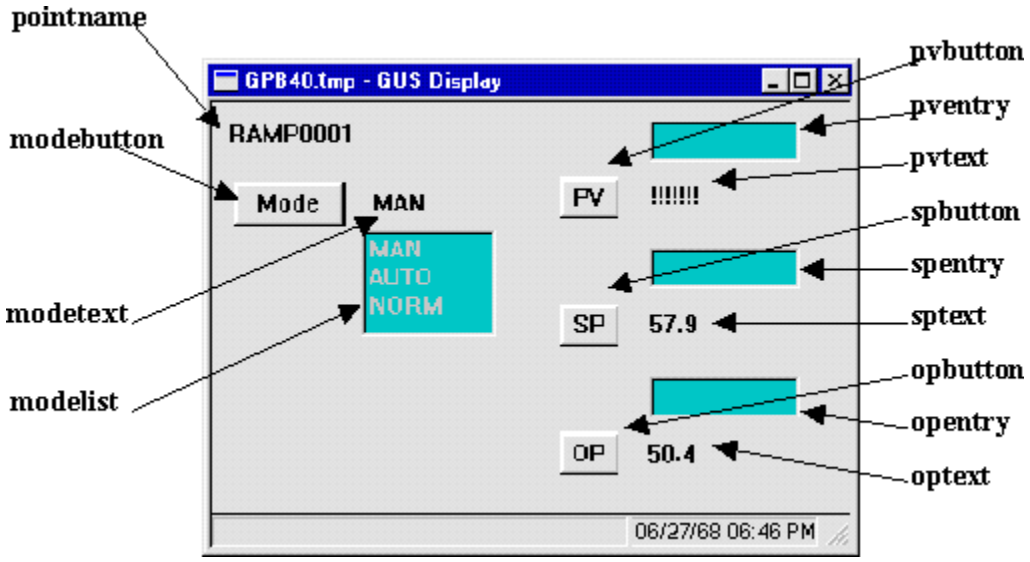
An Example of Scripting the PMK

A sample display that is integrated with the IKB is shown below. The scripts of the display and its objects write to the properties of the PMK object and invoke the PMK's methods to achieve this integration.

PMK Example display

Here is a buildtime view of a sample display, called the "PMK Example" display.

Display Fields



Functional Overview

At display runtime, the PMK Example display above shows the pointname and the current PV, SP, OP, and MODE of the REGAM point ramp0001. The entry boxes for the PV, SP, and OP are invisible. The MODE listbox is invisible.

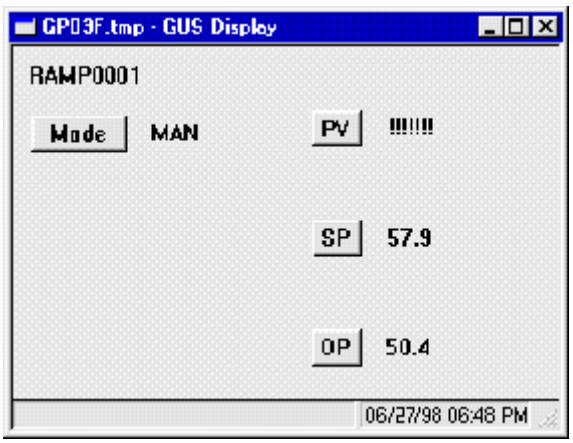
When the operator clicks the PV, SP, OP buttons, the appropriate entry box is displayed and has input focus. Pressing the SP key or the OUT key on the IKB initiates the same behavior for the SP or OP. The IKB rampkeys increment/decrement the SP or OP when the SP or OP entry box, respectively, has focus. The displays show each incremented/decremented value. When the SP or OP entry box loses focus, the rampkeys will no longer ramp the parameter of interest.

Pressing the MAN, AUTO, NORM keys on the IKB changes the mode of the point. Clicking the Mode button on the display shows the list of valid modes for the point with the current mode highlighted and sets the focus to the mode listbox. The IKB raise/lower keys “walk through” the list of valid modes. Pressing the ENTER key selects the highlighted mode as the new mode. Double-clicking the highlighted mode also selects that mode as the new mode.

All errors are handled.

A runtime view of the PMK Example display follows.

PMK Display Example



Implementation notes on the PMK Example display

- The PMK.Key method is called when the operator clicks the SP Button or the OP Button.
- The display sets the PMK entity, EventHandler, and ErrorHandler at display startup, and these properties never change during display execution. The display itself is both the EventHandler and the ErrorHandler.
- The values for AutoDataChange, EnableMode are the default values. AutoDataChange is set to FALSE. The display will refresh the value on the screen after each increment/decrement.
- The display sets AutoWrite to TRUE depending on the parameter of interest. AutoWrite is TRUE when the SP or OP is of interest. SP and OP are analog values for this point, and the desired behavior is to have the PMK object increment/decrement the value. The display sets AutoWrite to FALSE when the MODE is the parameter of interest. This enables the rampkeys to "walk through" the list of mode options in the MODE listbox.
- The ErrorHandler displays the contents of the PMK ErrString property. It is assumed that ErrString contains any error messages sent from the LCN.
- The PV Entry Box, SP Entry Box, and OP Entry Box utilize the OnLostFocus event to set their visibility and selectability to FALSE. The SP Entry Box and OP Entry Box also invoke the PMK.ClearParam method in their OnLostFocus events to clear the PMK parameter, thus preventing erroneous ramping of the OP or SP parameter. The PMK UserData property is used as a flag to determine if the other "rampable" parameter (for example, the OP if the SP is losing focus) now has focus, then the PMK.ClearParam is not invoked.
- When the MAN, AUTO, or NORM key is pressed, focus is set to the pointname object, whether or not the key press is valid. This makes it possible to close any open entry box or listbox.
- The items in the MODE listbox were added at buildtime.
- The timeout timer is activated on the SP entry box or the OP entry box when it becomes visible and gets focus. The timeout timer is reset after every ramp event. This is needed so the entry box does not "timeout" while the operator is ramping the SP or OP.
- Point lcn.ramp0001 is in data collection group number 222.
- The text of the pointname is defined on the value page of its property sheet.

Digital point.parameters

Handle digital point.parameters in the same manner as the PMK Example display handles the mode by providing a list of valid options and enabling the rampkeys to "walk through" the list of options.

Timeout support for the PMK object

Use the timeout properties on the listbox and data entry box as timeout support for the PMK object. The PMK object has no timeout capabilities. You cannot, for example, "timeout" the ability to ramp the SP.

Code for the PMK Example Display

DISPLAY

Sample Script

```
Public Sub DisplayError( valueObject as object, errornumber as
long )
   beep
   select case errornumber
   case HOPC_VALUE_ERROR
         valueObject.text = "--------"
   case HOPC_COMMUNICATION_ERROR
         valueObject.text = "???????"
   case HOPC_CONFIGURATION_ERROR
         valueObject.text = "@@@@@@@"
   case else
         valueObject.text = "!!!!!!!"
   end select
End Sub

Public Sub DisplayValue(Param as string, Value as Variant)
on error goto errorhandler
   select case Param
      case "SP"
         sptext.text = format$(lcn.ramp0001.sp, "#####0.0")
         'reset the timeout timer for the entry box
         spentry.TimeoutActive = true
      'case "PVTV"
      'not a valid parameter for this point, but for some point
      'types, PVTV is used instead of SP
      'case "AVTV"
      'not a valid parameter for this point, but for some point
      'types, AVTV is used instead of SP
      case "OP"
         optext.text = format$(lcn.ramp0001.op, "#####0.0")
         'reset the timeout timer for the entry box
         opentry.TimeoutActive = true
   case else
      err.raise  2001, "Ramping", "Illegal Param from PMK
      Ramp"
```

```
   end select
exit Sub

errorhandler:
msgbox Err.Description, ebinformation, "Display Ramp Value"
end Sub

Sub OnDisplayStartup()
set PMK.entity = lcn.ramp0001
set PMK.ErrorHandler = me
set PMK.EventHandler = me
PMK.UserData = ""
'holds the parameter to be ramped
'The EnableMode, AutoUpdate, AutoWrite properties for this
'display at startup have the default values
'The AutoDataChange property is set to false; this display
'assigns the value in the raise/lower events to the
appropriate Text object
PMK.AutoDataChange = false
End Sub

Sub OnPMKError(ErrCode As Long,ErrString As String)
   msgbox ErrString, ebinformation, "Error"
End Sub
Sub OnPMKAUTO()
   'set focus to cause a clear param if SP or OP
   'had focus
   pointname.setfocus
End Sub
Sub OnPMKMAN()
   'set focus to cause a clear param if SP or OP
   ' had focus
   pointname.setfocus
End Sub
Sub OnPMKNORM()
   'set focus to cause a clear param if SP or OP
   ' had focus
   pointname.setfocus
   End Sub
```

```
Sub OnPMKSP()
on error goto errorhandler
'used as a flag by OnLostFocus event of OPEntry object
   PMK.UserData = "SP"
   spentry.text      = ""
   spentry.visible = true
   spentry.selectable = true
   spentry.setfocus
   'activate the timeout timer
   spentry.TimeoutActive = true
exit sub
errorhandler:
   beep
   pointname.setfocus
   msgbox Err.Description, ebinformation, "Change SP"
End Sub

Sub OnPMKOut()
on error goto errorhandler
'used as a flag by OnLostFocus event of SPEntry object
   PMK.UserData = "OP"
   opentry.text      = ""
   opentry.visible = true
   opentry.selectable = true
   opentry.setfocus
   'activate the timeout timer
   opentry.TimeoutActive = true
   exit sub
errorhandler:
   beep
   pointname.setfocus
   msgbox Err.Description, ebinformation, "Change OP"
End Sub

Sub OnPMKFastLower(Param As String,Value As Variant)
'on error goto errorhandler
   if modelist.visible = TRUE then
      'can't walk the mode list with the Fast keys
      beep
   else
      DisplayValue Param, Value
```

```
   end if
End Sub
Sub OnPMKFastRaise(Param As String,Value As Variant)
   if modelist.visible = TRUE then
      'can't walk the mode list with the Fast keys
      beep
   else
      DisplayValue Param, Value
   end if
End Sub

Sub OnPMKLower(Param As String,Value As Variant)
   if modelist.visible = TRUE then
   'walk down the list of valid modes
      modelist.TimeoutActive = true
      if modelist.ListIndex < (modelist.ListCount - 1) then
         modelist.ListIndex = modelist.ListIndex + 1
      end if
   else
      DisplayValue Param, Value
   end if
End Sub
Sub OnPMKRaise(Param As String,Value As Variant)
   if modelist.visible = TRUE then
   'walk up the list of valid modes
      modelist.TimeoutActive = true
      if modelist.ListIndex > 0 then
         modelist.ListIndex = modelist.ListIndex - 1
      end if
   else
      DisplayValue Param, Value
   end if
End Sub
```

SPTEXT

Sample Script

```
Sub OnDataChange()
   on error goto errorhandler
   me.text = format$(lcn.ramp0001.sp, "#####0.0")
exit Sub

errorhandler:
   DisplayError sptext, err.number
End Sub
```

SPBUTTON

Sample Script

```
Sub OnLButtonClick()
   'set the PMK key to SP (same as
   'pressing the SP key)
   PMK.Key PMK_SP
End Sub
```

SPENTRY

Sample Script

```
Sub OnEnter()
   on error goto errorhandler
   if IsNumeric(me.text) then
      lcn.ramp0001.sp = me.text
   else
      beep
      msgbox "Enter a Valid Numeric", ebinformation, "Valve"
      me.setfocus
   end if
   pointname.setfocus
   me.text      = ""
exit sub

errorhandler:
   beep
```

```
   msgbox Err.Description, ebinformation, "SP Error"
   me.text      = ""
   me.setfocus
End Sub

Sub OnTimeout()
   pointname.setfocus
End Sub

Sub OnLostFocus()
   me.visible = false
   me.selectable = false
   if PMK.UserData <> "OP" then
      PMK.ClearParam
      'neither SP nor OP is in focus for rampin
      PMK.UserData = ""
   end if
End Sub
```

OPTEXT

Sample Script

```
Sub OnDataChange()
   on error goto errorhandler
   me.text = format$(lcn.ramp0001.op, "#####0.0")
exit Sub

errorhandler:
   DisplayError optext, err.number
End Sub
```

OPBUTTON

Sample Script

```
Sub OnLButtonClick()
   'set the key to the OUT button (same as pressing
   'the OUT key)
   PMK.Key PMK_OUT
End Sub
```

OPENTRY

Sample Script

```
Sub OnEnter()
on error goto errorhandler
   if IsNumeric(me.text) then
      lcn.ramp0001.op = me.text
   else
      beep
      msgbox "Enter a Valid Numeric", ebinformation,
      "Valve"
      me.setfocus
   end if
   pointname.setfocus
   me.text      = ""
exit sub

errorhandler:
   beep
   msgbox Err.Description, ebinformation, "OP Error"
   me.text      = ""
   me.setfocus
End Sub
Sub OnTimeout()
   pointname.setfocus
End Sub
Sub OnLostFocus()
   me.visible = false
   me.selectable = false
   if PMK.UserData <> "SP" then
      PMK.ClearParam
      'neither SP nor OP is in focus for rampin
      PMK.UserData = ""
   end if
End Sub
```

MODEBUTTON

Sample Script

```
global modeValue          as string
Sub OnLButtonClick()
   on error goto errorhandler
   'Setting ListIndex will cause that item to be
   highlighted in
   'the listbox.
   modelist.ListIndex = modelist.FindItem(modevalue )
   modelist.visible = true
   modelist.selectable = true
   modelist.setfocus
   modelist.TimeoutActive = true
'Activate timeout timer
exit sub

errorhandler:
   beep
   msgbox Err.Description, ebinformation, "Change Mode"
End Sub
```

MODETEXT

Sample Script

```
global modeValue      as string
Sub OnDataChange()
   on error goto errorhandler
   modevalue = lcn.ramp0001.mode
   me.text = modevalue
   exit sub

errorhandler:
   beep
   DisplayError modetext, err.number
End Sub
```

MODELIST

Sample Script

```
global modattr as string
dim nmode as string
dim nmodattr as string

Sub OnDataChange()
'collect values to be used in Click event
   On Error Resume Next
   nmode = lcn.ramp0001.nmode
   nmodattr = lcn.ramp0001.nmodattr
   modattr = lcn.ramp0001.modattr
   'note the mode list is built at
   'build time for this example
End Sub


Sub OnLButtonDoubleClick()
   dim selectedItem as String
   dim paramstring as string
   on error goto errorhandler
   selectedItem = modelist.List(modelist.ListIndex)
      if selectedItem = "NORM" then
      if nmode <> "NONE" then
         paramstring = "Change MODE to " + nmode
         if ( nmodattr <> modattr ) and ( nmodattr <>
         "NONE" ) then
            paramstring = paramstring  + chr$(13) +
            chr$(10) + "and MODATTR to " + nmodattr
         end if
      else
         beep
         msgbox "Configure normal mode", ebinformation
      end if
   else
      paramstring = "Change MODE to " + selectedItem
      if modattr = "PROGRAM" then
         paramstring = paramstring + chr$(13) + chr$(10) +
         "and MODATTR to OPERATOR"
      end if
```

```
    end if
    Begin Dialog UserDialog 137,198,130,51,"Change Mode"
       OKButton 8,32,40,14
       CancelButton 80,32,40,14
       Text 8,8,116,20,paramstring,.Text1,"Courier New",8
    End Dialog

    dim d as UserDialog
    dim resp as integer

    if selectedItem = "NORM" then
       if nmode <> "NONE" then
          resp = dialog( d, , 30000  )
          if resp = -1 then
             lcn.ramp0001.mode = nmode
             if ( nmodattr <> "NONE" ) then
                lcn.ramp0001.modattr = nmodattr
             end if
             lcn.update 222
          end if
       end if
    else
       resp = dialog( d, , 30000 )
       if resp = -1 then
          lcn.ramp0001.mode = selectedItem
          lcn.ramp0001.modattr = "OPERATOR"
          lcn.update 222
       end if
    end if
    me.visible = false
    me.selectable = false
    exit sub


errorhandler:
    beep
    msgbox Err.Description, ebinformation, "Change Mode"
End Sub
    me.visible = false
    me.selectable = false
```

```
    'set the AutoWrite back to true
    PMK.AutoWrite = true
End Sub

Sub OnLostFocus()
    me.visible = false
    me.selectable = false

    'set the AutoWrite back to true
    PMK.AutoWrite = true
End Sub

Sub OnGotFocus()
    'AutoWrite must be false in order to use the ramp keys to
    'walk the mode list
    PMK.AutoWrite = false
End Sub
```

PVTEXT

Sample Script

```
Sub OnDataChange()
    on error goto errorhandler
    me.text = format$(lcn.multisch.pv, "#####0.0")
    exit Sub

errorhandler:
    DisplayError pvtext, err.number
End Sub
```

PVBUTTON

Sample Script

```
Sub OnLButtonClick()
   pventry.text      = ""
   pventry.visible = true
   pventry.selectable = true
   pventry.setfocus
   'activate the timeout timer
   pventry.TimeoutActive = true
End Sub
```

PVENTRY

Sample Script

```
Sub OnEnter()
   on error goto errorhandler
   if IsNumeric(me.text) then
      lcn.ramp0001.pv = me.text
   else
      beep
      msgbox "Enter a Valid Numeric", ebinformation, "Valve"
      me.setfocus
   end if
   pointname.setfocus
      me.text      = ""
   exit sub

errorhandler:
   beep
   msgbox Err.Description, ebinformation, "PV Error"
   me.text      = ""
   me.setfocus
End Sub

Sub OnTimeout()
   pointname.setfocus
End Sub

Sub OnLostFocus()
   me.visible = false
   me.selectable = false
End Sub
```

Fax Transmittal **Fax No.: (602) 313-4212**

Reader Comments

To: Automation College

From: Name Date:

Title:

Company:

Address:

City: State: Zip:

Telephone: Fax:

Display Authoring Tutorial , Release **R211**, **7/00**

Comments:

You may also call 800-822-7673 (available in the 48 contiguous states except Arizona; in Arizona call 602-313-5558), email us at College.Automation@honeywell.com, or write to:

Honeywell
Industrial Automation and Control
Automation College
16404 North Black Canyon Highway
Phoenix, AZ 85053

Knowledge Building Tools

TotalPlant

Honeywell

Industrial Automation and Control
Honeywell
16404 North Black Canyon Highway
Phoenix, AZ 85053