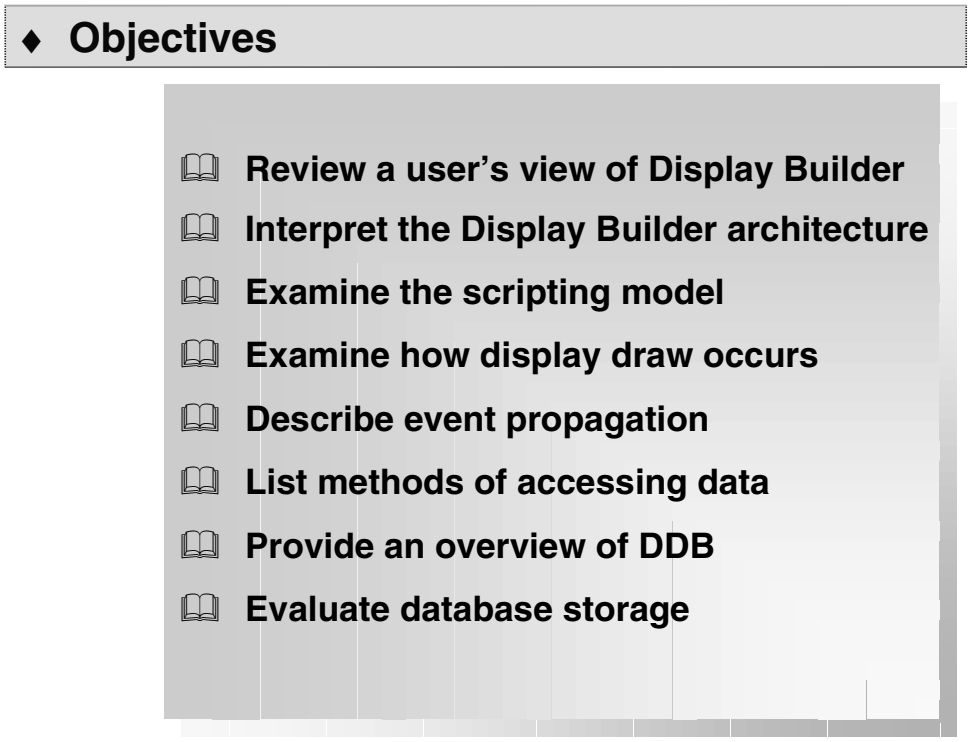
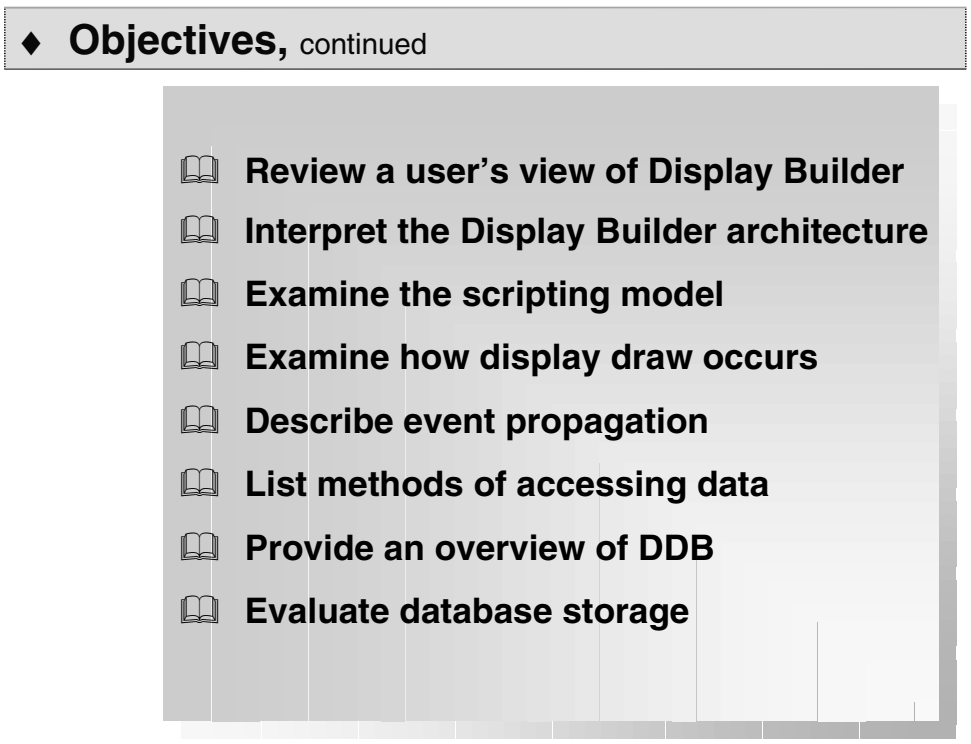

Display Builder Architecture



Objectives

At the end of this module you will be able to do the following:

- Review the user's view of the GUS display builder in terms of three basic things a user needs to know about GUS displays.
- Interpret the Display Builder architecture in terms of the following:
    - The three distinct processes (HOPC, GPB, .pct files) that occur to present a display
    - How display data reads and writes occur
    - What occurs during buildtime display validation
    - What happens during runtime display invocation
- Examine the scripting model for GUS displays in terms of the following
    - System generated events versus user generated events
    - Cached versus immediate reads in scripts
    - Multithreaded script execution
    - Execution of the periodic update thread
    - Execution of the data change thread
    - Execution of the user-interface thread
- Examine how a display draw occurs at invocation and runtime.



Objectives, continued

At the end of this module you will be able to do the following:

- Describe event propagation in terms of the following:
    - Types of events
    - Selectable versus visible properties
    - Enabled versus disabled properties
- List methods of accessing data in terms of the following:
    - Direct, indirect, and collector access of LCN data
    - LCN array parameter access
- Provide an overview of the Display Builder Database (DDB) in terms of the following
    - What the Display Database (DDB) represents
    - Data types supported
    - Global Database behavior
- Evaluate database storage in terms of the following
    - Where data can be stored
    - Data variable behavior comparison
    - Global constants
    - Overview of in-line parameters

User's View of GUS

Introduction

If you know the display model, then you can

- Build performant displays, and
- Not rely on cookbook guidelines.

Review the User's View of GUS

Three essential things:

- GUS displays consist of objects.
- Events are sent to objects.
- Your scripts handle the events sent to objects.

Introduction

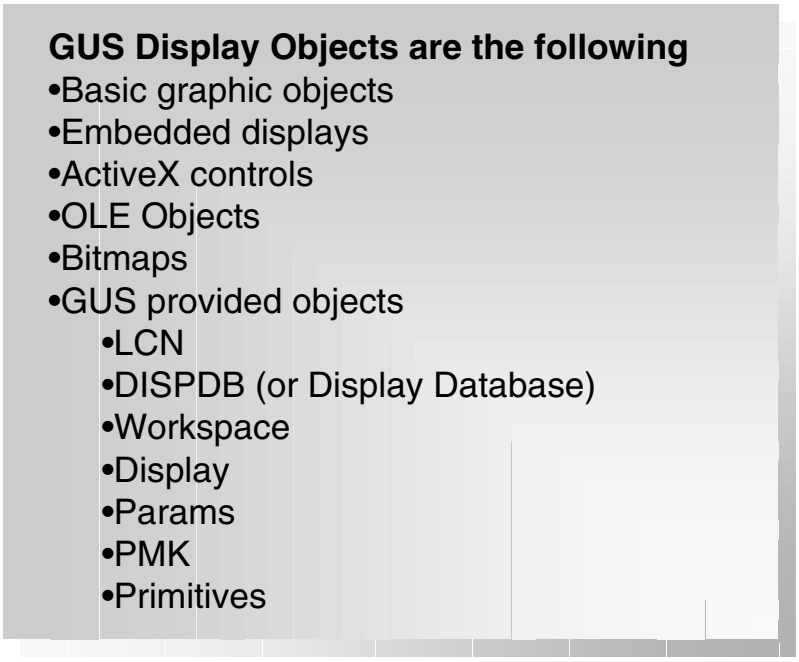
It's important to grasp that having an understanding of the display model lends itself to building performant displays and minimizes reliance on a set of cookbook guidelines.

Review the User's View of GUS

Three essential things a display builder needs to know about the user's view of GUS displays are

- GUS displays consist of objects.
- Events are sent to objects.
- User written scripts handle the events sent to objects.

GUS Display Objects



GUS display objects

GUS display objects consist of the following:

•Basic graphic objects - The rectangle, ellipse, and polygon are just a few examples.

•Embedded displays

•ActiveX controls - These are the Honeywell OLE controls or third-party controls

•OLE Objects - Examples include Word and Excel objects.

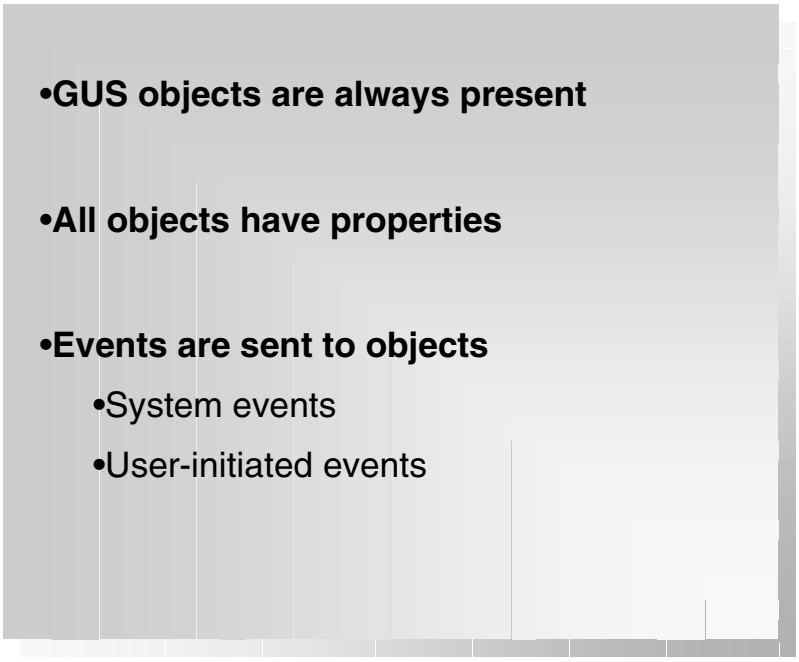
•Bitmaps

•GUS provided objects - GUS objects include the following:

- •LCN - an example of an LCN object is the LCN.tagname.parameter object as in "LCN.FIC100.PV".
- •DISPDB or Display Database - An example of the display database object is DISPDB.ENT01, where DISPDB represents the display database and ENT01 represents an entity such as "FIC100".
- •Workspace - Safeview object.
- •Display - The display itself is an object.
- •Params - The display has parameters that can be user defined in the format "display.params.datatype" (used in parameterized embedded displays).
- •PMK Object - Point Manipulation Object
- •New Primitives - ListBox, DataEntryBox and Button objects located on the toolbar.

•**Note:** Not all objects are visible.

GUS Display Objects



GUS objects are always present

The GUS provided objects – LCN, DISPDB, Workspace, display.params – are always present as visible objects that are used frequently in displays. You do not have to add these objects to a display in order to use them.

All objects have properties

All objects have properties. For example, a rectangle, which is an object on your display, has properties (fillcolor, linewidth, etc.). The DispDB also has properties (INT01, BOOL01, etc.) You access the properties of objects through the property sheets at build time or through scripting at runtime.

Events are sent to objects

What triggers a script to run? First of all, an event is sent to an object. There are two kinds of events:

- System events (OnDataChange, OnPeriodicUpdate, etc.)
- User-initiated events (OnLButtonClick, OnLButtonDoubleClick, etc.)

These events are supplied in the GUS Display Builder.

GUS Display Objects

- **System Event Examples**
  - Display starting up
  - Process value changing
  - Display closing down
  - Display updating periodically
- **User Initiated Event Examples**
  - Operator selects object
- **Events trigger script execution**

System Event Examples

You have probably encountered scripts that made use of system events such as a display starting up (OnDisplayStartup), a process value changing (OnDataChange), a display closing down (OnDisplayShutdown), a periodic display update (OnPeriodicUpdate). These events are triggered or "fired" by the Honeywell system.

Question: Could you consider OnDataChange in effect the same as an OnPeriodicUpdate, as far as scanning process data is concerned?

_____

User Initiated Event Examples

The operator triggers or "fires" an event by selecting a display object. A typical operator-initiated event is the left button click (OnLButtonClick) or the touchscreen event (OnLButtonUp). Other operator events include left button double click (OnLButtonDoubleClick) or right mouse button click (OnRButtonClick)..

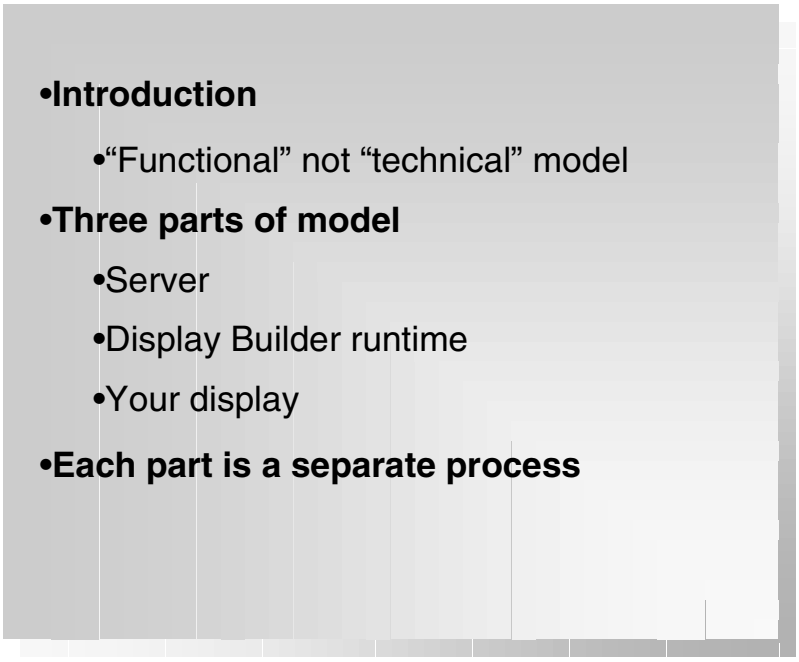
Events trigger the execution of user-written scripts

The events are supplied with the GUS Display Builder and accessed from the event browser while in the script editor window.

Question: What do you think is the most common event used in a script? _____

Note: On Periodic Update was originally used for animation, but animation can cause a performance hit. On Periodic Update is not intended for scanning data. That is the role of On Data Change.

GUS Architectural Model



Introduction

The architectural model is important for you to have as a "functional model." That way you do not have to rely on memorizing the rules stated in the Authoring Tutorial, and in effect you can begin resolving any potential problems yourself. The architectural discussion is a high-level discussion and not one that dwells upon technical details.

Three Parts of the Architectural Model

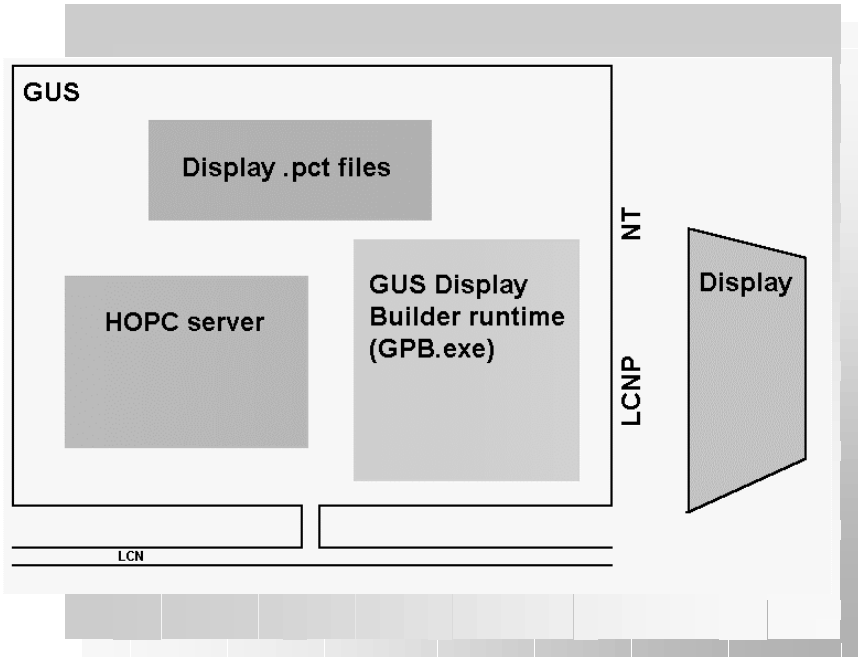
Three parts make up the architecture of the GUS Display Builder:

- The HOPC server (HOPC.exe)
- The GUS Display Builder runtime (GPB.exe)
- The user-built display (Display .pct files).

A simple concept to remember

Each part of the architectural model represents a separate process.

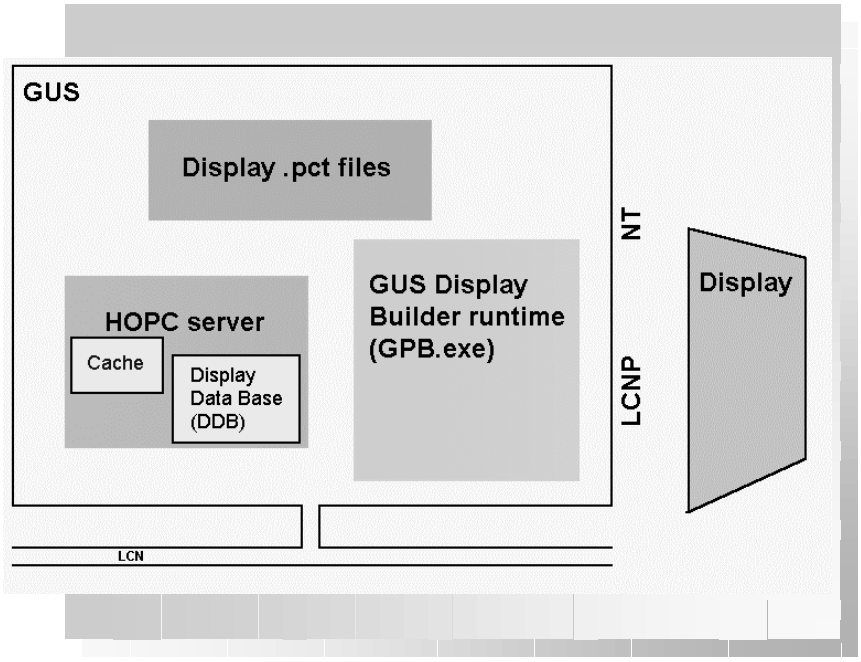
GUS Architectural Model



Our diagram shows each of the three parts as separate boxes

As we go through the discussion of the architectural model, you should note that each part (HOPC, GPB, and .PCT files) is shown in separate boxes because each represents separate processes. The easy rule to remember in display building is that anytime you have to go out of the box you are making an "out-of-process" call. That's more time consuming. But, first, get an overview of what is represented in each box for the HOPC server, GPB, and .pct files.

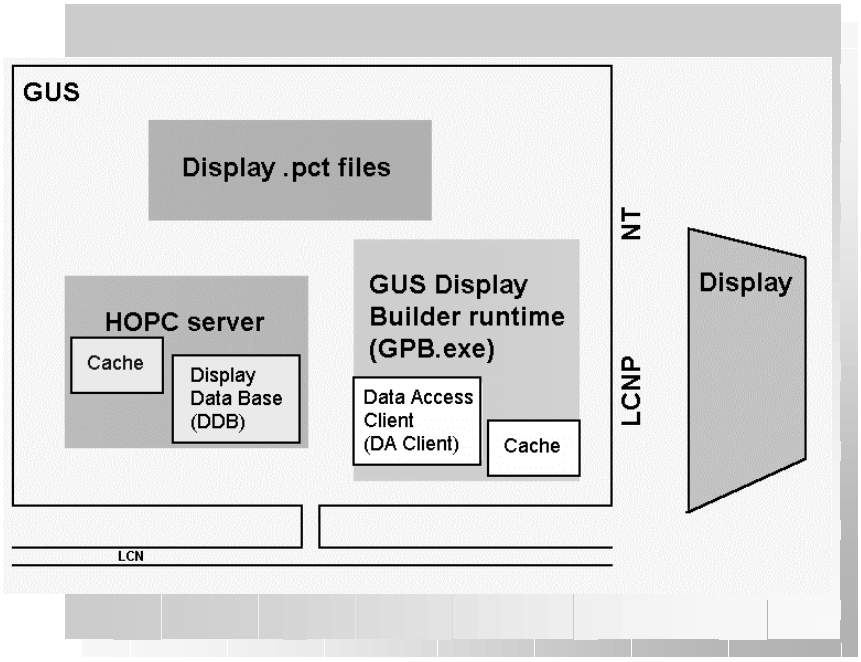
GUS Architectural Model: HOPC



What is the HOPC Server?

Recall that the acronym “HOPC” means “Honeywell OLE for Process Control. “ The HOPC process acts as a data server. It talks to the process network (LCN) and collects the process data. That data is stored in its cache. Note that the HOPC server holds the Display Database (DDB). The DDB contains items such as process variables to be collected for the display, such as FIC100.PV.

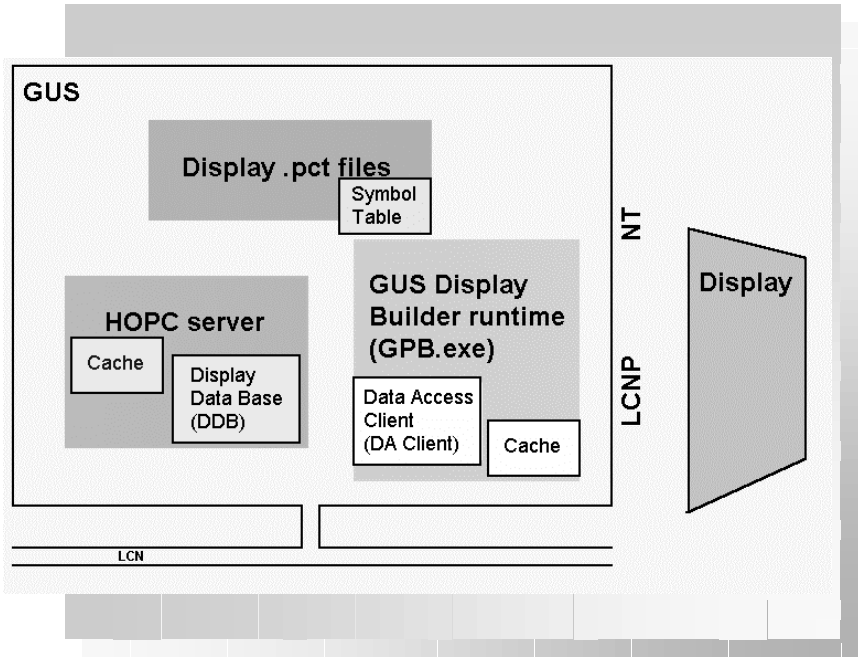
GUS Architectural Model: Display runtime



What is the Display Builder runtime?

The display builder runtime, GPB.EXE, contains a client called the “Data Access (DA) client.” Note that in the GPB box, there is also a cache that stores process data. This is the cache you will want to access for most of your process data. But before discussing that, let’s review the last part of the model - - the displays that you build.

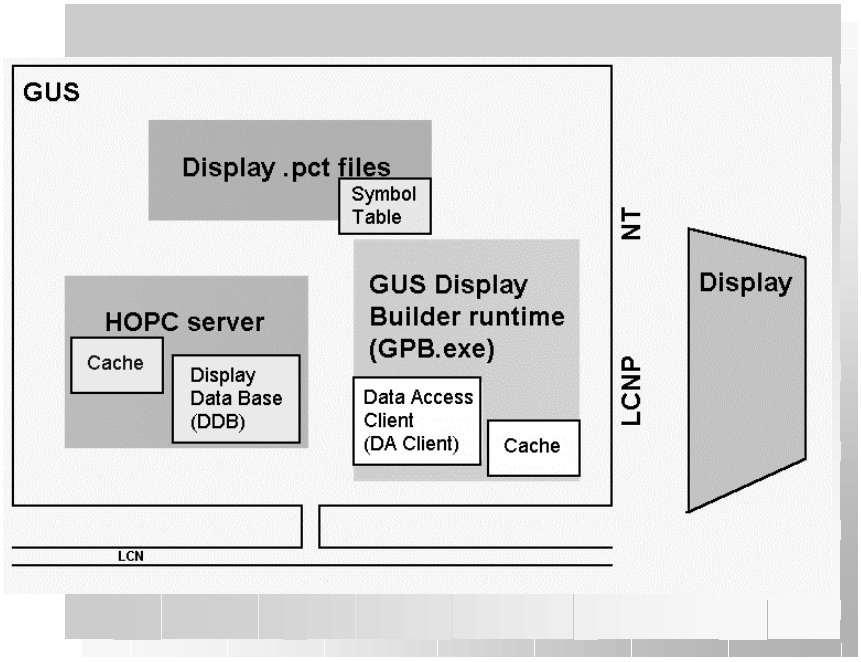
GUS Architectural Model: Display .pct files



The Display .pct files

The display .pct files are the user-built displays that you use to represent your process. At display validation time, a symbol table is created that contains (among other things) information about how often to scan variable such as “FIC100.PV” and what script references are made to it.

GUS Architecture: How HOPC and GPB work



How HOPC works

In HOPC there are two items to be concerned about as a display author. HOPC holds the Display Database (DDB). So, for example, if you have a statement such as DISPDB.INT01 = 5 (where DISPDB represents the Display Database object, and INT01 represent integer #1 of the Display Database object), the value for DISPDB.INT01 = 5 is stored there.

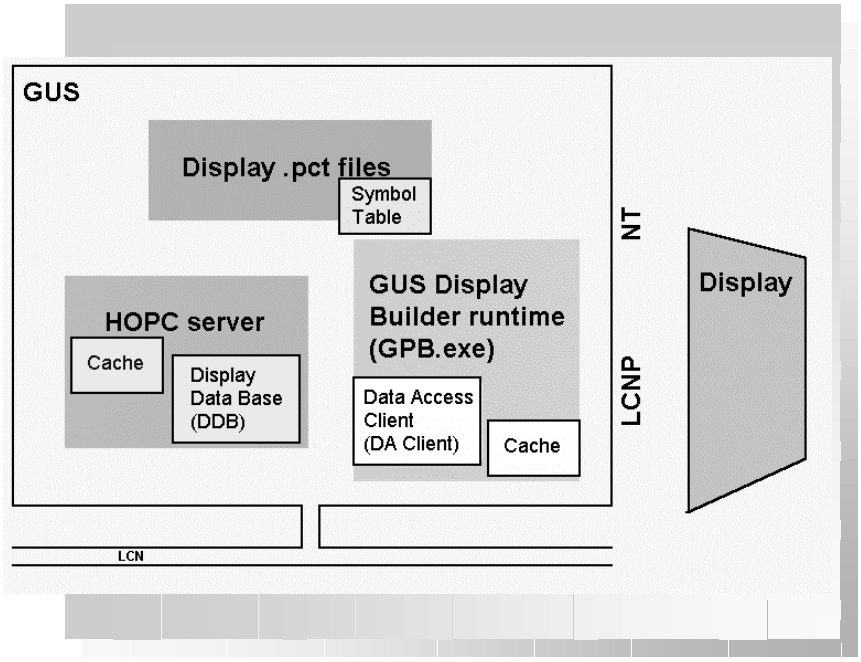
The HOPC server also talks to the LCN. At display startup, the GPB "client" sends down to the HOPC server its variables to be collected. These variables are in a symbol table. HOPC then formats the data, such as creating items called IDBs. HOPC then collects the data from the LCN. In essence, HOPC makes a batch request to the LCN. All values returned from the LCN are then put into HOPC's cache.

How GPB works with HOPC

You'll notice that on the GPB side there is also a cache. So naturally the question arises, what is the difference between the HOPC cache and GPB cache? Most of the time there is no difference at all! Because what HOPC does when it gets a new batch of data is that it compares the "old" value that is currently in cache to the "new" value. If the data has changed, it then sends an OnDataChange event along with the new value for each variable that has changed to GPB.

In summary, HOPC looks for values that have changed and provides the new values. What GPB does is to update its cache with the new values and send out the data change event.

GUS Architecture: Display Performance



How this all relates to display performance

How does this apply to building performant displays? First of all, at display startup, because everything has changed, HOPC is going to send back an "OnDataChange" event for every variable in the display. Therefore, you do not need to duplicate what's in your "OnDataChange" scripts to also be in your "OnDisplayStartup" scripts, because the "OnDataChange" scripts will run right after the "OnDisplayStartup" script(s) at display startup time.

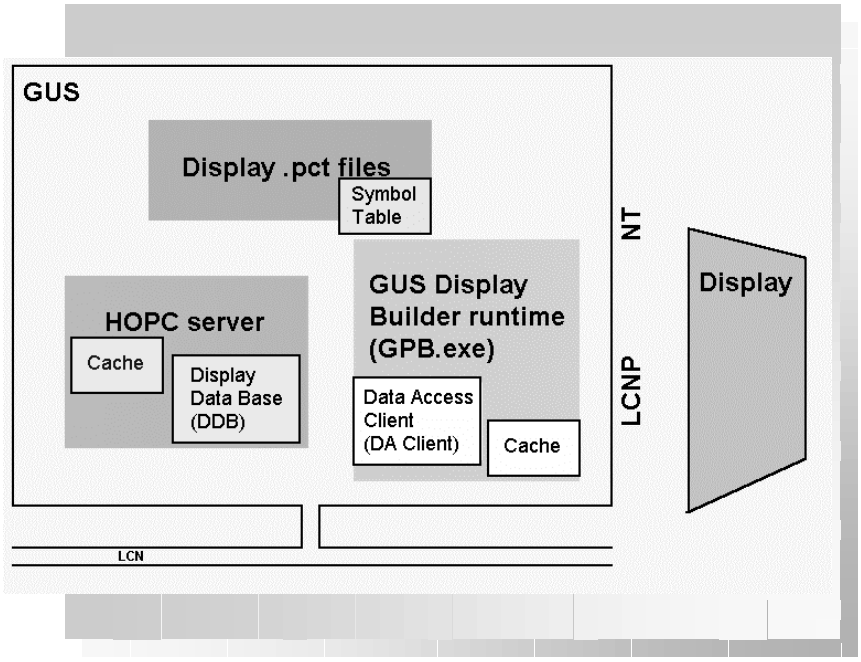
Another reason to avoid duplication in both scripts is that any data in an "OnDisplayStartup" script is an immediate read. So, that leads to another question. What does an "immediate read" mean in terms of our architectural model? An immediate read means that GPB, when it wants the data, must go all the way through HOPC, have HOPC go to the LCN, and then back again to GPB in order to get and display the value. But, the OnDataChange event retrieves its values from GUS Display Builder runtime cache. Because this is an in-process call, it is going to be very fast. As we mentioned earlier, out-of-process calls can be slow.

Therefore, use OnDisplayStartup judiciously, usually for cases where values have to be initialized in a display, for example, DISPDB.ENT01 = "FIC100". As you might expect, there is an exception. If you had scripted:

SET DISPDB.ENT01 = LCN.FIC100

Then that statement (because you have used "SET") is bound at build-time and is not an immediate read.The goal here is to avoid duplicating OnDataChange and OnDisplayStartup events. They will run at the same time. When your display is started, all OnDataChange events are executed.

GUS Architecture: Runtime processing



How Display Builder runtime (GPB.exe) works

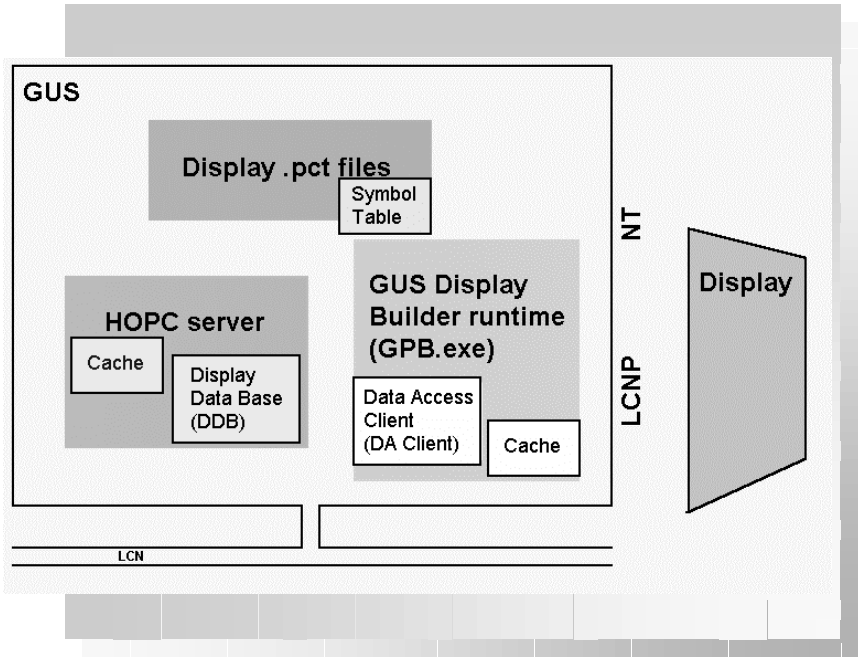
Basically you want the GPB process to run very fast for performant displays. So, as is implied from the architectural model, you want to minimize or avoid going to the HOPC for data. And, you want to minimize going all the way to the LCN with an immediate read through HOPC. You want to go to GPB cache for your display data.

To do that means you need to read most of your values in OnDataChange events, because it will go to GPB cache, which is a very fast read.

Note: Any write is immediate. So, for example, to write to a display database means that it is an out-of-process call. To write to the LCN means a write through HOPC to the LCN and back.

Writes, on the average, can take about a ½ second. Because writes can be time-consuming, you want to do them on a left button click user initiated event. LeftButtonClicks are provided for users who interact with or control process events.

GUS Architecture: Display files processing



How Display .pct files work

Next, review the role of the .pct file. Where does the GPB process get it information? Recall that a .pct file is created with your display objects (rectangles, polygons, etc) and scripts. The .pct files also has a symbol table. The symbol table is built at display validation time.

How you can tell that it is built then is that if you save a file before validation, note its file size; you will see that its file size is larger after display validation.

The symbol table is an extension of the data collection list. In other words, the data collection list is part of the symbol table. The symbol table contains the variables, all data references to the variables, the scan rate(s), and the demand groups. The symbol table also contains what "OnDataChange" scripts reference which variables.

For example, if HOPC sends back to GPB that the output value (OP) for FIC100.OP has changed, then GPB knows what objects to send "OnDataChange" events to.

The update from HOPC cache to GPB cache depends on the scan rate for the variable. That is, if FIC100 is at a scan rate of 10 seconds and TIC100 is a 4 seconds, the caches are updated for FIC100 every 10 seconds and TIC100 every 4 seconds.

In summary, when a picture is validated a symbol table is created. Variables in the .pct symbol table have assigned collection rates (either default or user-assigned). HOPC (the data server between the LCN and the GPB) collects values from the LCN according to the scan rates in the collection list. HOPC then compares these new scanned values (which are put into the HOPC's cache) with the GPB cache. If there is a difference, the GPB cache is updated and an OnDataChange event is generated for the changed variables.

GUS Architecture: Read/Write Data

- **Two methods of reading LCN data**
  - Immediate read
  - Scanned (cached) read
- **One method of writing LCN data**
  - Immediate write
- **Example of reading LCN data**

  IF LCN.FIC100.PV > X
- **Exception:**

  .cachevalue or **.immediate**

Reading and Writing Data from the LCN

As commented on from our previous discussion, there are two methods of reading LCN data

- Immediate read from the LCN
- Scanned (cached) read

but only one method of writing LCN data, an

- Immediate write to the LCN

Example of Immediate Read

An immediate read from the LCN – This read represents a GPB request to the LCN through HOPC. A script statement such as

IF LCN.FIC100.PV > X

causes an immediate read, if the event causing the read is user-initiated, OnDisplayStartup or OnDisplayShutdown.

Exception: On R120 and later, you can force a read from cache using a **.cachevalue** notation in a user event script, such as **OnLButtonUp().** Likewise, you can force an immediate read from the LCN by using a **.immediate** notation in an **OnDataChange()** script.

Caution: Use **.immediate** notation on an exception basis. It can cause the **OnDataChange()** thread to suspend execution while it waits for process data to be returned.

GUS Architecture: Display Validation

Several validation actions occur

- Syntax of each script
- Existence of each referenced parameter
- Certain references are build-time bound
- Symbol table built
- Symbol table stored with .pct file

Symbol table has

- Parameter name
- Internal data reference
- Object script references
- Group and collection rates

What Happens at Display Validation Time?

Several actions occur during display validation

•The syntax of each script in the display is validated.

•The existence of each referenced parameter (LCN, display parameter, DDB) is validated.

•Certain parameter references (entity, variable, inline parameters) are build-time bound (see note).

•Symbol table is built

•Symbol table is stored with the display .pct file.

Symbol table has, but is not limited to:

•Param Name (lcn.tic21941.pv, display.params.tag.pv)

•Internal data reference (if bound at build time -- "SET DISPDB.VAR01 = LCN.TIC21941.PV")

•Object scripts that reference the parameter (OnDataChange…)

•LCN group and collection rate information (Data Collection list)

Note: This is a binding that occurs at build-time. The importance of this will become clearer when we examine how to assign an entity to an entity variable later in the course.

GUS Architecture: Display Invocation

Several invocation actions occur

- GUS Runtime (GPB) reads the display file
- GPB sends the Symbol Table to HOPC Server
- GPB fires OnDisplayStartup events
- HOPC handles immediate reads and writes
- HOPC Server creates the IDB groups
- HOPC Server reads parameters from LCN
- Because first read, HOPC knows that every parameter has changed.
- Because first read, HOPC sends back an OnDataChange event for every parameter.

What Happens at Display Invocation Time?

At display invocation time, the following actions occur:

- GUS Runtime (GPB) reads the display file
- GPB sends the Symbol Table to the HOPC Server
- GPB fires “OnDisplayStartup” events to the display and its objects
- HOPC handles immediate reads and writes from the GPB scripts
- HOPC Server creates the IDB groups (internal database groups used for data access)
- HOPC Server reads the parameters from the LCN
- Because this is the first read, HOPC recognizes that every parameter has changed.
- Because this is the first read, HOPC sends back to GPB an OnDataChange event for every parameter.

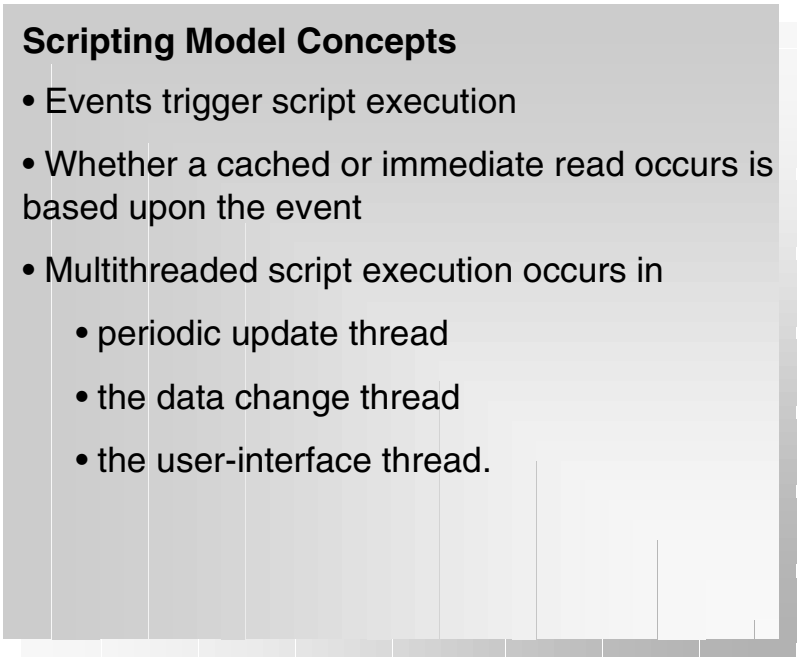
Question : Based upon what you have just learned, if you had a display with slow call-up occurring, what could you change?

Answer (there can be more than one correct answer)

_____

_____

GUS Scripting Model: Concepts



Introduction

Now that you have seen the architectural model for display building, there's one more model to examine to fully understand the GUS Display Builder –the scripting model.

Scripting Model Concepts

The following concepts describe the scripting model for GUS:

- Events trigger script (event-driven execution of script.)
- Whether a cached or immediate read occurs is based upon the event (user-initiated events do immediate reades and OnDataChange events go to GPB cache)
- What multithreaded script execution represents in terms of the
    - periodic update thread
    - the data change thread
    - the user-interface thread.

Note: The GUS Display Builder has benefits that Visual Basic does not provide. For example:
•VB does not have OnPeriodicUpdate or OnDataChange built into it.
•GUS also provides pre-binding and data source management.

GUS Scripting Model: System Events

System events include the following:

- OnPeriodicUpdate
  - Event is fired every 1/2 second
- OnDataChange
  - Only fired when a parameter changes
  - Collection rate sets frequency
- OnDisplayStartup
  - Event fired when the display invoked
- OnDisplayShutdown
  - Event fired when the display closed

System events of the GUS Display Builder

System events make the GUS Display Builder runtime (GPB) different than Visual Basic. The system events include the following:

- OnPeriodicUpdate
  - Event is fired every 1/2 second
- OnDataChange
  - Only fired when a parameter changes
  - Collection rate of parameter determines the frequency that an OnDataChange event for a particular parameter could be fired.
- OnDisplayStartup
  - Event fired when the display is invoked.
- OnDisplayShutdown
  - Event is fired when the display is closed.

Question: If an OnDataChange script references FIC100.pv, and the PV does not change in five minutes, will the script be executed in that five minute interval? _____

Question: How often can an OnDataChange script run? _____

GUS Scripting Model: User Events

User events include the following:

- Mouse Click events
  - OnLButtonDown
  - OnLButtonUp
  - OnLButtonClick
  - OnRButtonClick, etc.
- Integrated keyboard (IKB) events
  - OnPageForward
  - OnDisplayForward
  - OnHelp, etc.

User events of the GUS Display Builder

User events can be considered as “user-fired events” ; they include the following:

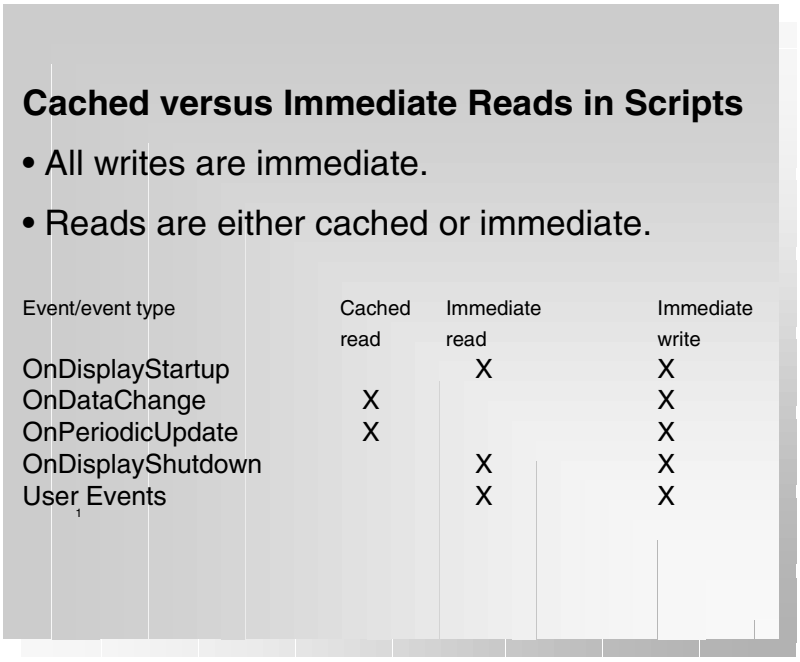
- Mouse Click events, such as (but not limited to)
  - OnLButtonDown
  - OnLButtonUp
  - OnLButtonClick
  - OnRButtonClick
  - PMK Events
- Integrated keyboard (IKB) events, such as (but not limited to)
  - OnPageForward
  - OnDisplayForward
  - OnHelp

Note: Three events in LButtonClick
A left button click by the operator results in firing three events : OnLbuttonDown, OnLButtonUp, OnLButtonClick.

Touchscreen equivalent is LButtonUp
For users with touchscreen displays, lifting a finger from a touchscren target is equivalent to OnLButtonUp.

Note: PMK can be regarded as having special events.

GUS Scripting Model: Cache vs. Immediate



Cached versus Immediate Reads in Scripts

All writes are immediate. Reads are either cached or immediate. The following table describes which event causes an immediate or cached read.

Event or event type	Cached read	Immediate read	Immediate write
OnDisplayStartup		X	X
OnDataChange	X		X
OnPeriodicUpdate	X		X
OnDisplayShutdown		X	X
User Events		X	X

Note: Exceptions for **.immediate** and **.cachevalue** notations are not shown in the above chart.

GUS Scripting Model: Multi-thread Execution

Multithreaded Script Execution

Three threads are executed at display runtime:

- OnPeriodicUpdate thread
- OnDataChange thread
- User Event thread

Key concepts about the threads

- Each of the threads has an event queue
- Scripts run serially in the queue

Multithreaded Script Execution

Finally, the multi-threaded script execution concept is introduced. This, with the rest of the scripting model and architectural model knowledge, will help you make almost all of your display performance decisions. Three threads are executed at display runtime, which are the following:

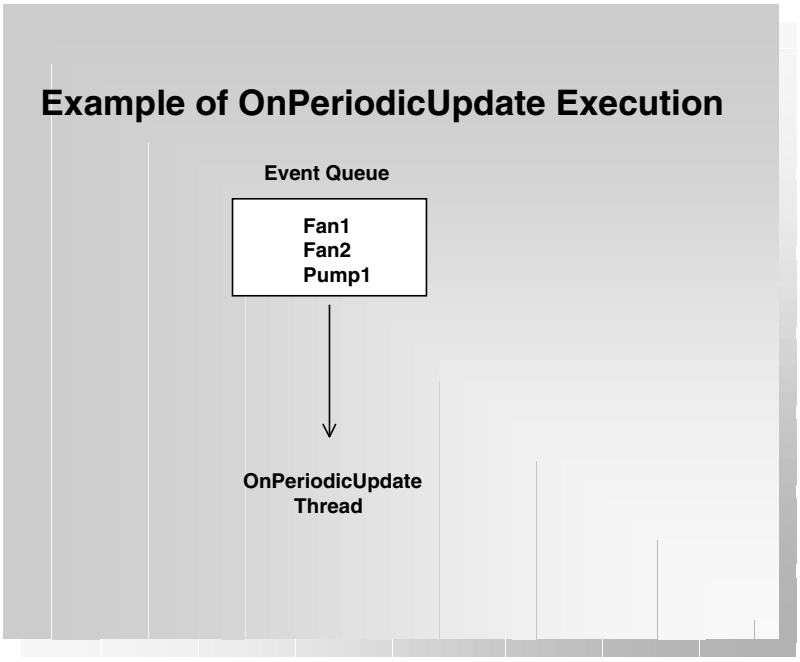
- OnPeriodicUpdate thread
- OnDataChange thread
- User Event thread

Key concepts about the threads

Two key concepts to be aware about the threads

- Each of the threads has an event queue
- Scripts run serially in the queue

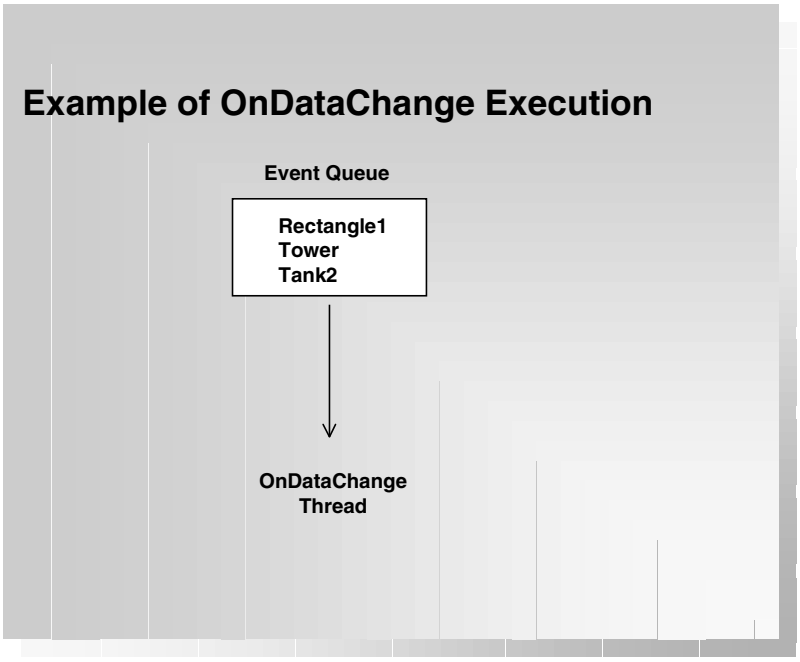
Scripting Model: Periodic Update Thread Example



Example of OnPeriodicUpdate Thread execution

When an event is fired, it is put into the queue to run. Scripts run serially. So, in our example, an event occurred that caused the script for PUMP1 to go into the thread. The script for FAN2 is not run until the script for PUMP1 is executed. Therefore, you want the script for PUMP1 to run fast. Avoid code that causes a script to slow down. Foe example, the script for PUMP1 can be slowed down with Sleep statements, iterative processing (For loops, while loops), or immediate reads or writes that require HOPC to go to the LCN to get the data. Also, user interaction can slow down this type of script. For example, if example, if PUMP1 has a dialog box appear, then that dialog requires user interaction. But, in this case, the dialog stops the rest of the scripts from being executed in the thread. So in this example, FAN2 and FAN1 do not get to run.

Scripting Model: Data Change Thread Example



Example of OnDataChange Thread execution

The OnDataChange thread also runs serially like the OnPeriodicUpdate thread. And, like the OnPeriodicUpdate thread, you want your scripts to run quickly in this thread so don't slow the script down with unnecessary sleep statements, iterative processing, immediate reads/writes/ or user interaction**.**

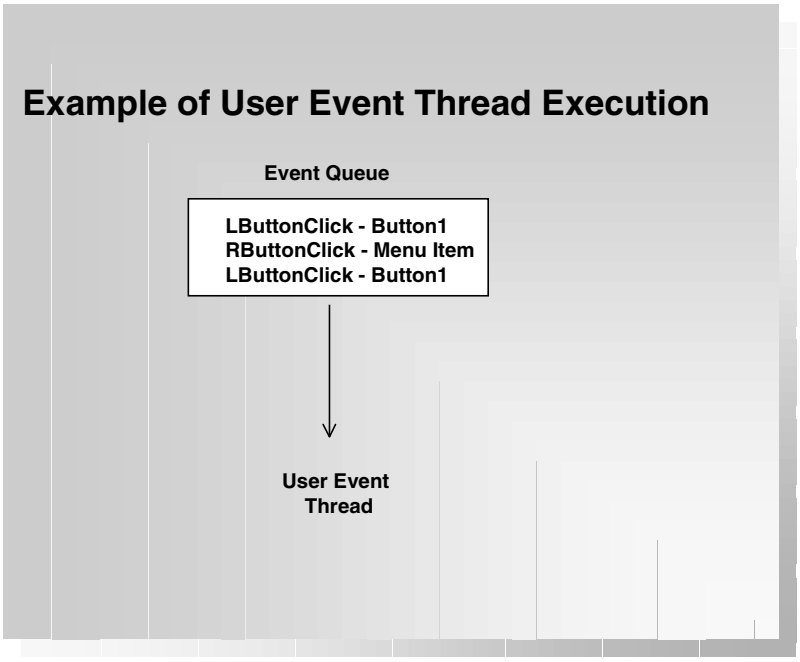
Purpose is to show process data

Because OnDataChange is one of the most common scripts, another concept is worth mentioning. The GUS Display runtime is not intended to be your control system. It's purpose is to show the control system and process data to your operator. In other words, you should not be trying to do control in your OnDataChange scripts. For example, if you have a statement that compares the PV to a number (e.g., If PV>100) then sets the setpoint (then set SP = ##), you are are trying to do control in your display. That's not the purpose of a GUS display. Again, the purpose of a GUS display is to display the process data to your operator and have the operator interact with it.

Question: Suppose the script for TANK2 references TIC100.PV and FIC100.PV. If FIC100.PV changes, an event is fired and TANK2 is in the queue. Now, if TIC100.PV changes while TANK2 is in the queue, will TANK2 be placed in the queue again?

Answer: _____

Scripting Model: User Event Thread Example



Example of User Event Thread execution

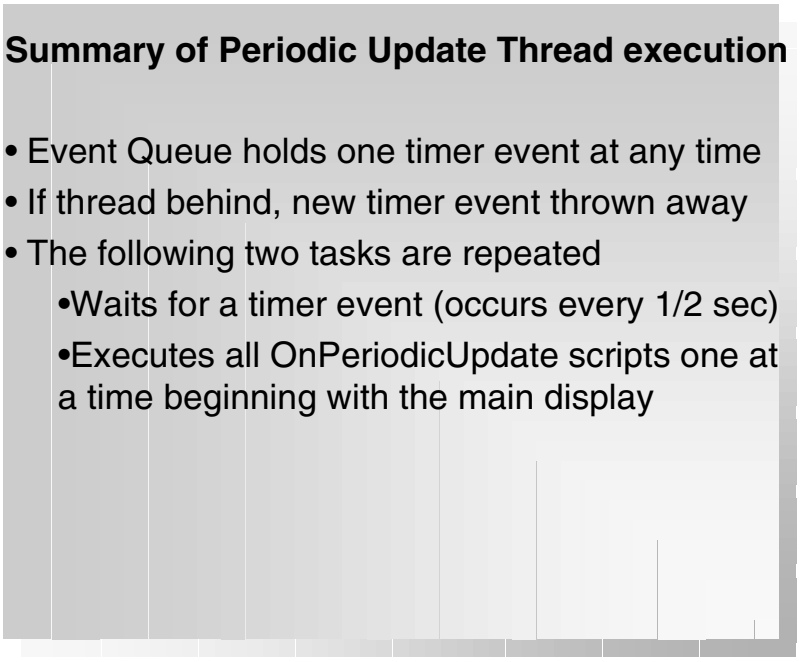
In this thread, the user is in control. This is where you often encounter immediate reads and writes in your scripts because the operator interacts with the system with dialog boxes and other OLE controls. Like the other threads, events can be queued up. But, the difference in this thread is that the user is in charge. It's conceivable to queue up several user items in this thread.

Even though it is acceptable to use dialog boxes with user events, keep in mind that the user event thread will be held up by modal controls. (An example of a modal control is where a dialog box waits for an operator to click the OK or CANCEL button before allowing the next action to happen.)

Why it is important to understand GPB thread execution?

The importance of understanding thread execution is best explained by an example. Assume that an operator is interacting with a dialog box on a display but gets momentarily distracted by a call from the field. In a GUS display, the threads for the OnPeriodicUpdate and OnDataChange are not stopped; they continue to execute. The operator will continue to see current data. Multithreaded script execution provides the benefit of updating the display with current process data.

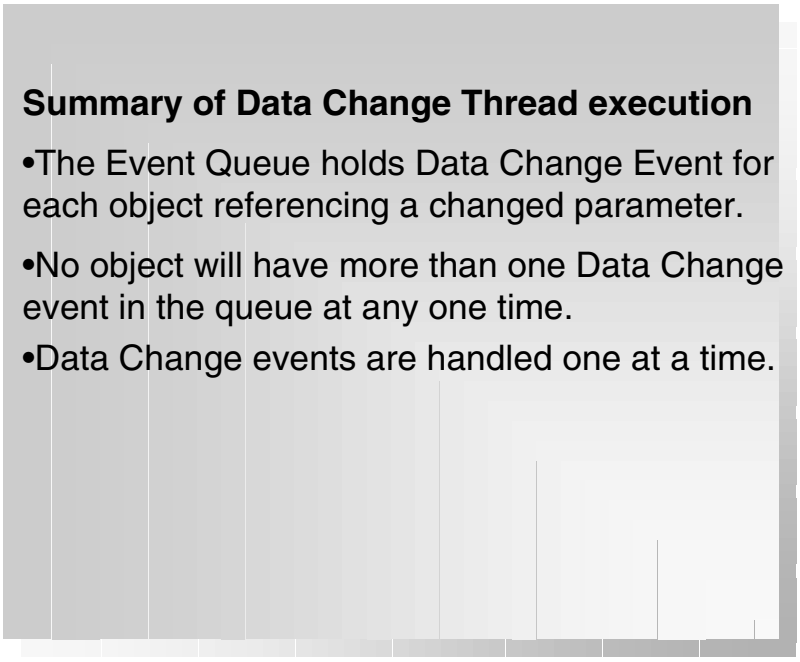
Scripting Model: Periodic Update Execution



Summary of Periodic Update Thread execution

- The Event Queue of the OnPeriodicUpdate Thread can hold only one timer event at any time.
- If the Periodic Update thread falls behind, the “new” timer event is thrown away.
- The following two tasks are repeated by the Periodic Update thread.
    - Waits for a timer event (occurs every 1/2 sec)
    - Executes all the OnPeriodicUpdate scripts one at a time beginning with the main display

Scripting Model: Data Change Execution



Summary of Data Change Thread execution

• The Event Queue of the Data Change Thread holds a Data Change Event for each object that references a changed parameter.

• No object will have more than one Data Change event in the queue at any one time. Duplicate Data Change events are thrown away.

• Data Change events are handled one at a time.

Scripting Model: User Event Execution

Summary of User Event Thread execution

- The Event Queue holds a User Interface Event for each user action.
- User Interface events “stack up” on the queue.
- User Interface events handled one at a time.

Summary of User Interface Event Thread execution

- The Event Queue of the User-Interface Thread holds a User Interface Event for each user action.
- User Interface events simply “stack up” on the queue. Duplicate events are not tossed out - they build up in the event queue.
- User Interface events are handled one at a time.

Display Draw: Introduction

How Draw works in terms of

- Display drawing at display invocation
- Display drawing at display runtime

UpdateDisplay function

Display Draw: Drawing at Display Invocation

When a display is invoked (OnDisplayStartup):

- The OnDisplayStartup scripts are run
- The Data Access client (GPB.exe) sends a message to the HOPC server to start collecting data
- Drawing of static objects begins
- HOPC Server messages GPB only when data collection is complete
- OnDataChange scripts run
- Drawing of dynamic objects begins

Display Drawing at Display Invocation

The following sequence occurs when a display is invoked (OnDisplayStartup):

- The OnDisplayStartup scripts are run.
- The Data Access client in GPB sends the message to the HOPC server to start collecting data.
- Drawing of static objects begins.

Note: At this point in display invocation the display will still show RRRR and TTTT for text objects that require dynamic values because the HOPC server messages GPB only when data collection is complete.

- HOPC Server messages GPB only when data collection is complete
- OnDataChange scripts run
- Drawing of dynamic objects begins

Display Draw: Drawing at Display Runtime

At display runtime, a redraw occurs when:

- Three script threads are idle
- Immediately after a script with user event runs to completion
- If no idle time has occurred within three seconds, a redraw is forced

Display Drawing at Display Invocation

Several rules to be aware of about re-draws and their occurrence:

•Redrawing is done when the three script threads are idle (i.e., Idle means that there is nothing in the queues and no scripts are running) or

•Redrawing occurs immediately after a script handling a user interface event runs to completion. For example, after a LButtonClick event there is a screen re-draw, or

•If no idle time has occurred within three seconds, a redraw is forced.

Display Draw: UpdateDisplay function

What is the UpdateDisplay function?

- Overrides runtime draw
- Forces redraw

Guidelines for use

- Show operator correct object selection
- Use sparingly

Function redraws static object

What is the UpdateDisplay function?

You can override the display runtime draw model using the function “UpdateDisplay.” The UpdateDisplay function allows you, as a display author, to force the display to redraw in your script even though the threads are not in an idle time.

Guidelines for using the UpdateDisplay function

Two guidelines for using this function are

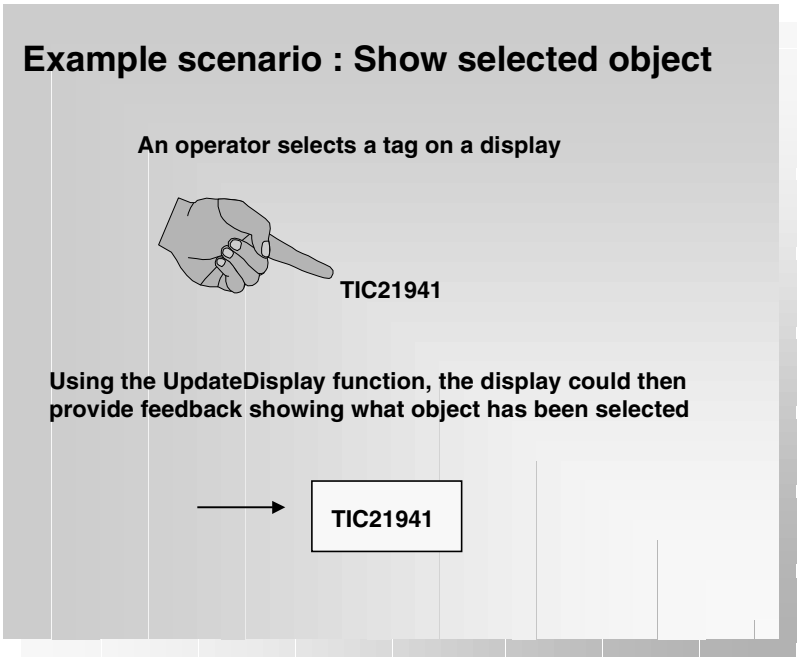
- The purpose of using UpdateDisplay function is that it is to be used in UserEvent scripts to show some color indication to the operator that the correct target is selected and that you may need to do more user input processing in a dialog box.

- Use this function sparingly, it can impact performance.

Note: UpdateDisplay function redraws static object

The redraw that occurs using UpdateDisplay is of static objects. UpdateDisplay does not go out to the process network and get the latest values for the variables.

Display Draw: UpdateDisplay function



Example scenario for the UpdateDisplay function

Assume you have a need to indicate to the operator that the desired display object, say a blue rectangle with a line width of 1, is selected. To show the selection, you could enter a script similar to the one below:

```
Sub OnLbuttonClick()
me.fillcolor = tdc_red
me.linewidth = 5
x= inputbox “Enter tagname”
--rest of script
End Sub
```

In the above script, the object turns red and the line gets wider, but the re-draw does not occur until all the script threads are idle or, as in this case, the user event runs to completion. But, our original purpose was to indicate to the operator that the desired object was selected. In this example, the object did not show a change until it may have been too late for the operator, so the operator may be puzzled as to what change went with what target. So, if its desired to show a selected object to the operator, then before you show the dialog box to an operator, script in the function “UpdateDisplay”. UpdateDisplay forces a redraw of the screen. In our example, the redraw will display a new fillcolor and line width. Our script could look similar to this:

```
Sub OnLbuttonClick()
me.fillcolor = tdc_red
me.linewidth = 5
UpdateDisplay
x= inputbox “Enter tagname”
(rest of script)
End Sub
```

Event Propagation: Overview

Event Propagation based upon

- Whether the event is a system event, on data change event, or user event.
- For user events whether the
  - primitive object is selectable and/or visible,
  - OLE control is enabled or disabled, and
  - object is grouped.

Primitives: Visible versus selectable

Script execution order at display startup

Event Propagation

In general, events move through (or propagate through) objects in a display. Rather than come up with a complex set of rules, this section describes event propagation in terms of

- Whether the event is a system event, on data change event, or user event.

- For user events, whether the primitive object is selectable and/or visible, whether the OLE control is enabled or disabled, and whether the object is grouped.

Primitives: Visible versus selectable

A primitive object, such as a rectangle, can be invisible but still remain selectable. In that case, it is possible to have some scripted actions take place. Just because something is invisible does not mean that you cannot have a scripted user event coded for it.

Script execution order

Question: In what order are display and object scripts run at display startup?

Answer: _____

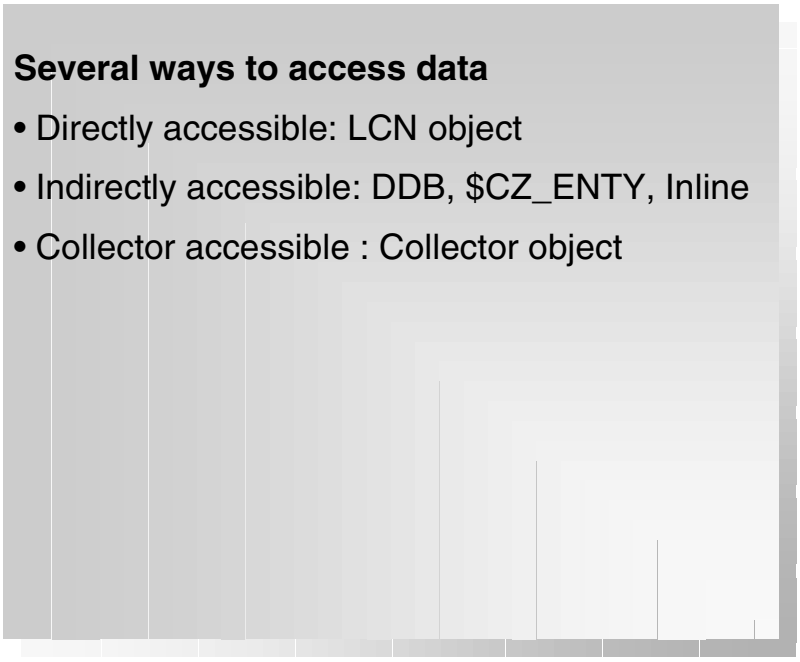
Question: In what order are object scripts run at display startup?

Answer: _____

Question: In what order are display and object scripts run at display shutdown?

Answer: _____

Accessing Data: Overview



Accessing data

TPN data is accessed from GUS displays in several ways:

• Directly accessible, through the use of the LCN object -- LCN.TIC21941.PV

• Indirectly accessible, through the use of the Display Database object, a change zone entity, an inline parameter data type -- DISPDB.VAR01, DISPDB.[$CZ_ENTY], TAG.PV

• As collector data accessible, using the collector object -- COLLECTOR("ACKSTAT(FIC21941)")

Accessing Data: Direct access, LCN object

Direct access

- LCN object example

  Text1.text = LCN.FIC100.PV

- LCN is an object
- Point is an object
- Point has properties

Directly accessing data through the use of the LCN object

In your previous display building class, you saw that you could access LCN data easily using script statements such as the following:

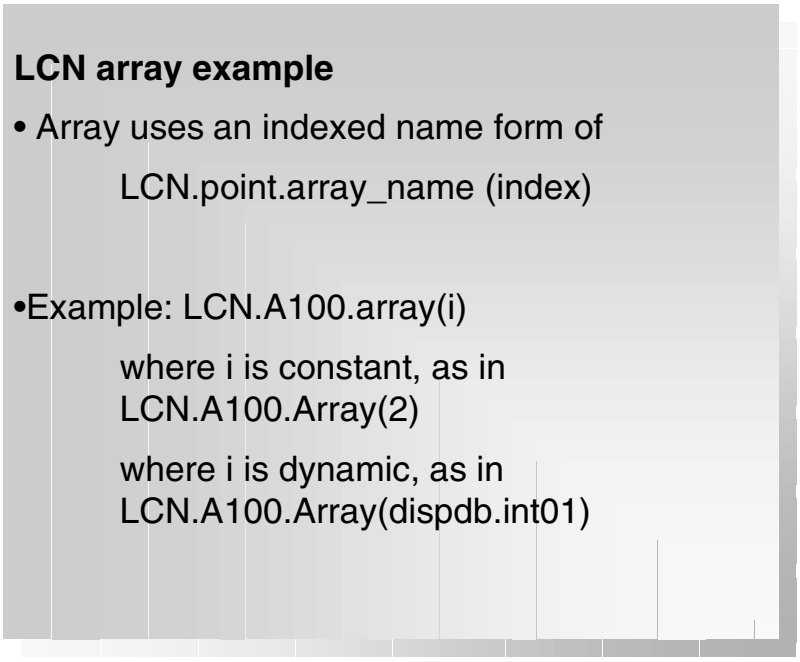
Text1.text = LCN.FIC100.PV

This type of statement is an example of directly accessing process data. Because we have been talking about the GUS Display Builder in terms of objects, it's useful to re-examine and review the example statement in those terms.

First of all, the LCN prefix in "LCN.FIC100.PV" itself is an object. The LCN prefix represents an LCN. FIC100, of course, is a tagname on that LCN. A "tagname" is considered a point in the Display Builder, or more accurately, a "point object". So, in other words, the point object, FIC100, is a directly accessible property of the LCN data object.

The point object (FIC100 in our example) can also have properties. A point's properties are its parameters. FIC100's PV, setpoint (SP), and output (OP) are example parameters. The point's parameters - - PV, SP, OP, and other parameters - - are treated as objects in the GUS Display Builder

Accessing Data: LCN Parameter arrays



LCN points can have parameters that are arrays

LCN points may have parameters which are arrays of values. To access that data means that you use a form of direct access called “indexed access”. Indexed access means that you have a set of values and you want to point to one member in that set of values. Those points are accessed using an indexed name form of

LCN.point.array_name (index)

For example: LCN.A100.array(i)

where (i) may be a constant, as in LCN.A100.Array(2) or

where (i) may be dynamic, as in LCN.A100.Array(dispdb.int01)

Another array example would be the Statetxt() parameter of Digital Composite points --

LCN.FVL21941.STATETXT(0)

LCN.FVL21941.STATETXT(1)

Accessing Data: Indirect Access of LCN data

- **Definition of indirect access:** Reference an object through another variable to get:
  - point name (entity)
  - point.parameter (variable)

- **Ways of indirect access**
  - entity display database (DISPDB) data type,
  - variable display database (DISPDB) data type,
  - a change zone entity, $CZ_ENTY,
  - inline parameter data type

Indirect access of LCN data

Indirect access essentially means that script may reference LCN points and their parameters by "going through' another variable which contains the name of the point (called an entity) or the name of the point.parameter (called a variable).

What are the ways of indirectly accessing LCN data?

Several data types enable indirect access

- an entity display database (DISPDB) data type -- DISPDB.ENT01
- a variable display database (DISPDB) data type -- DISPDB.VAR01
- a change zone entity, $CZ_ENTY -- DISPDB.[$CZ_ENTY]
- inline parameter data type, accessed through the defined parameter -- TAG.PV

Accessing Data: Indirect Access, Entity

- **Example of indirect access of entity :**

  SET dispdb.ent01 = lcn.A100

- **SET versus assignment statement**
  - SET binds at build time
  - Assignment does not build-time bind

Entity type for indirectly accessing LCN data

As an example of indirect access, ,review the statement: SET dispdb.ent01 = lcn.A100 The statement SET is derived from Visual Basic. The example shown is actually the fastest way to get data from the LCN point. The reason that this is that the data is bound at build time, not at runtime. According to the Basic Script reference manual, the set statement "does not duplicate the object being assigned but rather copies a reference of an existing object to an object variable." In everyday terms, the script statement "knows" it needs to go out and get an object's ID, in this case the internal id for the LCN point. What happens is that, at build time, the code has an LCN ID pointer to that object.

Another preferred way of accessing data is: SET DISPDB.ent01 = Get_Ent("FIC100"). This statement is also bound at build time. GET_ENT on the right-hand side of the assignment statement is a GUS Display Builder function.

A less preferred way of accessing data is shown in the following statement: DISPDB.ENT01 = "A100". Although this statement looks very similar to the SET statement, it is what the script considers as an assignment statement. In this case, the data is bound at runtime and will not be as performant as a statement bound at buildtime.

Question: Assume that you used the statement: SET dispdb.ent01 = lcn.A100. Then, for whatever reason, you delete A100 and re-build the point, would you have to re-validate the display?

Answer: _____

Question: Can you think of a reason or display requirement where it is not possible to use the preferred way of accessing data (i.e., the SET statement)?

Answer: _____

Accessing Data: Indirect Access, Variable

- **Example of indirect access of variable :**

  SET dispdb.var01 = lcn.A100.PV

- **Same rules apply:**
  - SET binds at build time
  - Assignment does not build-time bind

Variable type for indirectly accessing LCN data

Another means of indirectly accessing LCN data is the use of a Dispdb variable statement.

For example:

SET DISPDB.VAR01 = LCN.A100.PV

In this example, the point's PV is set to a display database variable. Thus, the same rules apply here as those for the earlier SET examples. (i.e, SET is more performant than an assignment statement because the data is bound at build time).

Accessing Data: Indirect Access, $CZ_ENTY

- **Examples of indirect access in change zone :**

  DISPDB.[$CZ_ENTY] = “A100”

- **Change zone rules:**
  - Only way to get data is to use $CZ_ENTY
  - Build-time bound

Indirectly accessing LCN data via change zone $CZ_ENTY

For display builders who are familiar with building change zones, you know that the way to access data is to use the system entity $CZ_ENTY. So, as an example, to bring a point into the change zone you use a statement similar to the following:

DISPDB.[$CZ_ENTY] = “A100”

The only way you can bring data into a change zone is to use $CZ_ENTY. The statement may give you the impression that it is not as fast as our previous SET examples. However, the GUS Display Builder recognizes this as a change zone entity and binds the data at build time.

Note: Later in the class other ways are shown to access data using $CZ_ENTY

Accessing Data: Indirect Access, Inline

- **Example of indirect access with inline :**
  - "tag" defined as inline parameter for LCN.A100
  - me.text = tag.pv
- **Inline rules:**
  - Defined for embedded displays
  - Enables indirect access for collector data

Indirectly accessing LCN data via inline data type

Inline parameters represent a third data type that is defined for embedded displays. The data types are value (boolean, real, integer, etc.), reference (entity and variable), and the inline data type. Inline parameters cause a text string subsitution at compile time and enable indirect access for collector data, such as the ACKSTAT collector.

As an example of using an inline parameter, you could do the following:

Define "tag" as an inline parameter

Then, in your script for the embedded display, access the inline parameter:

me.text = tag.pv

You do not need the "display.params" syntax when accessing inline parameters.

Note: Later in the class other ways are shown to access data using inline parameters

Display Database: Overview

- **Introduction**

  DISPDB represents Display Database object

- **Two types of Display Database (DISPDB):**
  - Global
  - Local

Overview of the Display Database (DDB)

Earlier we described examples of statements accessing LCN data, such as

SET dispdb.ent01 = lcn.A100

In this example, DISPDB represents the Display Database object.

Two Types of Display Database (DDB)

Two types of Display Database (DDB) exist in the GUS Display Builder:

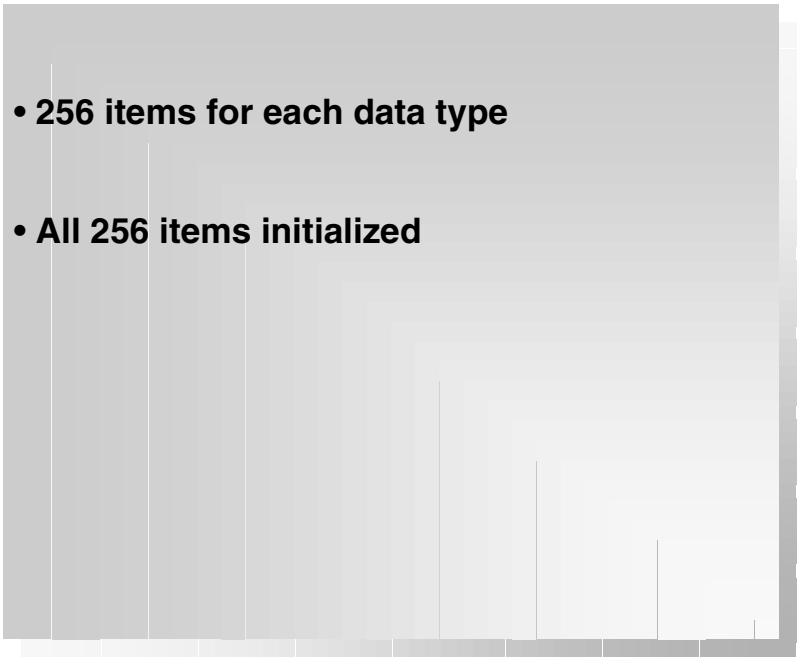
- Global Display Database (DDB), and
- Local Display Database (DDB).

Each GUS display has its own global and local DDB.

Question: What do the terms “global” and “local” usually mean to you?

Answer:_____

Display Database: Number of Data Types



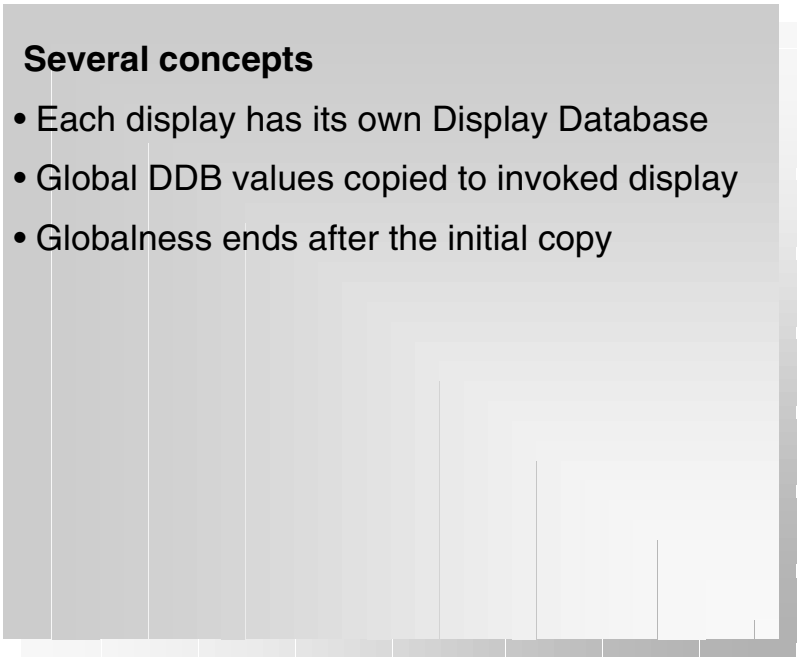
Number of Data Types in the Display Database (DDB)

- There are 256 items for each data type in GUS.
- All 256 items are initialized.

Note: TDC3000 Picture Editor supports 20 items of each data type. GUS extends the number of data types to 256 items.

Early releases of GUS only initialized the first 20 items of each data type. Now all are initialized.

Display Database: Display Runtime



Running Displays using the Display Database (DDB)

Several concepts to be aware of when you have a running GUS display are:

• Each display has its own Display Database (both Global and Local)

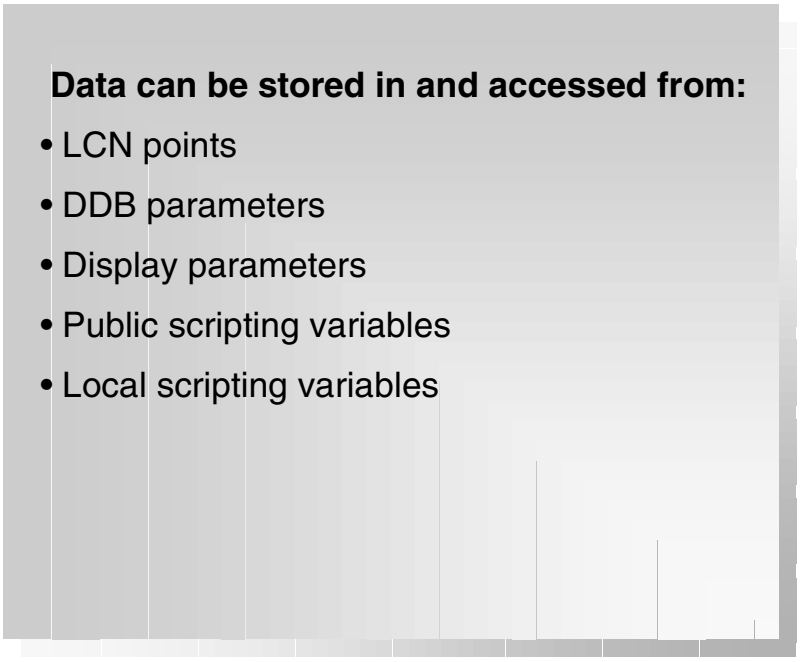
• When one display invokes another display, the values of the Global DDB of the invoking display are copied (or passed) to the Global DDB of the newly invoked display.

• Globalness ends after the initial copy. The global values are only passed one way -- from the invoking display to the invoked display.

As an example -- when a target is selected in a display and that display calls up or invokes a second display, any Global DDBs that were defined in the original display are passed to the second display. It's a "one-shot" pass. If the newly-invoked display then writes to its global DDB, the first display would have no knowledge of the new values. Each display's global DDB operates independently of all others.

Note: After display invocation, the connection between the invoking display and newly invoked display is broken, so the globalness ends after the initial copy. In that sense, the variables that are in the global DDB are not truly global, at least not in the way one normally thinks of a global variable.

Display Data Storage/Access: Summary



In summary, we have seen that data can be stored in and accessed from:

• LCN points (LCN.FIC100.PV)

• DDB parameters (DISPDB.VAR01)

• Display parameters (DISPLAY.PARAMS.TAG.PV or inline TAG.PV)

• Public scripting variables (P = LCN.TIC21941.PV)

• Local scripting variables (L = 24)

Display Data Variable Comparison

Comparison of access, types supported, write times

	Access scope	On Data Change ?	Data types	Write time
LCN points	all displays	Yes	All LCN data types	Slow
DDB parameters	• Display, objects • Global DDB values	Yes	All LCN data types	Slow
Display parameters	Display, its objects, and embedded display instance	Yes	All LCN data types	Fast (in-process)
Public variables	Display and objects	No	Note: use object data type to reference	Fast (In-Process)
Local variables	Subroutine only	No	Note: use object data type to reference	Fast (In-Process)

	Access scope	Cause OnDataChange Event	Data types	Write time
LCN points	all running displays	Yes	Supports all LCN data types	Slow
DDB parameters	• Display and its objects • Global DDB values passed to invoked display	Yes	Supports all LCN data types	Slow
Display parameters	Display, its objects, and the embedded display instance. Exception: Inline parameters can be accessed only in the embedded display that defines them.	Yes	Supports all LCN data types	Fast (in-process) Exception: Entity, variables, and Inline parameters are slow (have to go to LCN).
Public scripting variables	Display and objects that add these variables to their script. Cannot be accessed in basic dynamics	No	Note: use the object data type to reference and entity and a variable	Fast (In-Process)
Local scripting variables	Subroutine only	No	Note: use the object data type to reference and entity and a variable	Fast (In-Process)

As you review the variable types, note that the public and local scripting variables do NOT cause an OnDataChange event (only LCN items do). The Write Time column above refers to either a write to the LCN or to a display object. Writes to the LCN have to go through the HOPC server, thus will take longer than a write to a display object. An example of object data type is shown in the following statements:

```
Dim o as Object
Sub OnDataChange()
Set o = LCN.A100.PV
Me.Text = o
End Sub
```

Display Database: Global Constants

Global constants

•Display Builder does not support global constants

•Ways to achieve global constants:

Option 1

- •Define the constants in the System Registry
- •Use Basic Script function to read the values from the System Registry. Or,

Option 2

- Read in a text file containing the constants

Registry use example:

Color constants in registry; each site different requirements

Global constants

The GUS Display Builder does not support global constants. There is, however, a way to achieve global constants:

- Option 1
  - Define the constants in the System registry
  - Use a Basic Script function to read the values for m the System Registry. (Note: Do not write to the system registry. Or,
- Option 2
  - Read in a text file containing the constants

Accessing the system registry for global constants is faster than reading in a text file of constants.

Registry use example

A company may decide to have color constants in the registry because each site has different color requirements for their process equipment. The registry was then used to store 132 color constants.

Note: The system registry lives in RAM. The more items you add to the registry, like constants, the more memory you use. This could leave less memory for the GUS Display Builder.

ATTENTION: All applications use the registry. Users assume all responsibility for the integrity of the registry. Corrupting the registry can mean a complete NT re-install from scratch.

Display Database: Intro to Inline Parameters

Introduction

Inlines are not like typical display parameters

Purpose of Inline parameters

Access collector data from embedded display

Limitations of Inline parameters

- Only accessed in the defining display
- Errors when the embedded display is validated

Introduction to Inline parameters, an embedded display data type

The following discussion introduces the inline parameter in terms of purpose, limitations, cautions, and how to access, special operators, and accessing TDC collector data.

Purpose of inline parameters

The reason for inline parameters is that they provide a way to parameterize collectors. To have a better idea of what need inline parameters meet, let's digress for a moment with an example scenario. Assume that you have the following script statement:

STAT = COLLECTOR ("ACKSTAT (FIC100)")

The code uses an ACKSTAT collector. If this code was part of an embedded display (i.e., subpicture) there is no way to parameterize which point you wanted to ACKSTAT. This makes the subpicture unusable if this code (the way it is written now) is part of an OnDataChange script. So to support collectors, Honeywell provides a data-type parameter called in line. It's intended to support parameterization of collectors, nothing more; but in practice inline parameters have been used broadly. In summary, inline parameters support a text substitution type of parameter-passing allowing you to use collectors in your embedded displays.

Limitations of inline parameters

•Inline parameters can only be accessed in the defining display, not in the containing display.

•Inline parameters give validation errors when the embedded display is validated.

Caution: There is no build time checking when you use an inline parameter.

Display Database: Access Inline Parameters

Several concepts

- Do not use display.params prefix
- Example
  - Name of inline is “TAG”
  - Value is “LCN.FIC100”
  - Script access PV: me.text = TAG.PV
  - Script validation: me.text = LCN.FIC100.PV

Accessing inline parameters

Inlines are different than typical embedded display parameters. For example, If you had an entity defined as a parameter of data type entity and you had wanted to reference it, you could code the statement as

SET DISPDB.ENT01 = display.params.myentity

where: a display parameter “myentity” is defined as type entity. On the right hand side of the statement, “myentity” is prefixed with “display.params” because it is the parameter of that display that now owns “myentity.” The approach of using “display.params” is true for integers, real and so on EXCEPT for inline parameters. Inline parameter do not have “display.params” in front of the parameter. Recall that inline parameters are mainly use to support collectors, as in the example:

ALM = collector (“ACKSTAT (\pe(TAGPT))”)

In practice what happens is that if you are coding script to access collector data, users feel that they might as well use inlines for other data, as well. For example, let’s revisit accessing a tagname such as FIC100 through an embedded display parameter of “myvalve” which was defined as data type “entity”. Assume that you want to access its tagname and output. The script statements could appear as:

```
me.text = display.params.myvalve .[name]   ‘put name into text object.
me.text = display.params.myvalve .op       ‘put output into text object.
```

For an inline parameter, you would define “myvalve” as a parameter of data type “ inline”. The script statements would then appear as:

```
me.text = myvalve.[name]  ‘put name into text object
me.text = myvalve.op      ‘put output into text object.
```

Again, the advantage that inlines provide is that they can be used with collectors while display.params cannot.

Me.text = collector (“ACKSTAT (\pe(myvalve))”)

Display Database: Inline Operators

Several concepts

- Enables usage in string literals
- Two operators: \p and \pe

- \p(Paramname)
  - Example: str% = “Name:\P(TAGNAME)”
  - Replace tag at validation: “Name:LCN.FIC100”
- \pe(Paramname)
  - Example: alrmstat = collector(“ackstat(\PE(TAGNAME))”)
  - Remove “LCN” at validation: collector(“ackstat(FIC100)”)

Special Operators for Inline parameters

When using inline parameters inside of string literals, you will have to use one of two operators:

- \p
- \pe

\p operator

The \p operator performs a simple substitution at display validation time. The operator has the syntax of \p(paramname). For example:

str% = “Name: \p(tag)”

At display validation, this statement becomes “Name: LCN.FIC100”

\pe operator

The \pe operator removes or “eliminates” the “LCN” prefix from the tagname. The operator has the syntax of \pe(paramname). For example:

alarmstat = collector(“ackstat(\pe(tag))”)

At display validation, this statement becomes alarmstat = collector(“ackstat(FIC100)”)