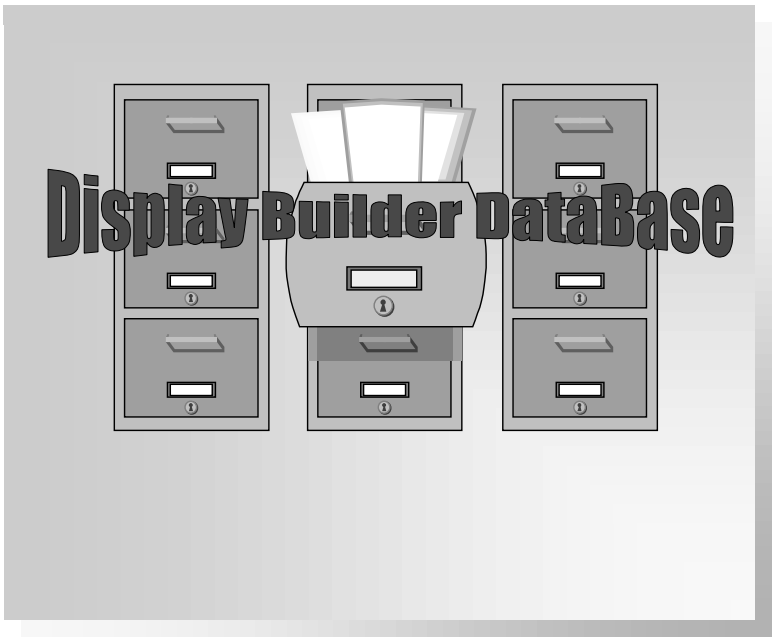
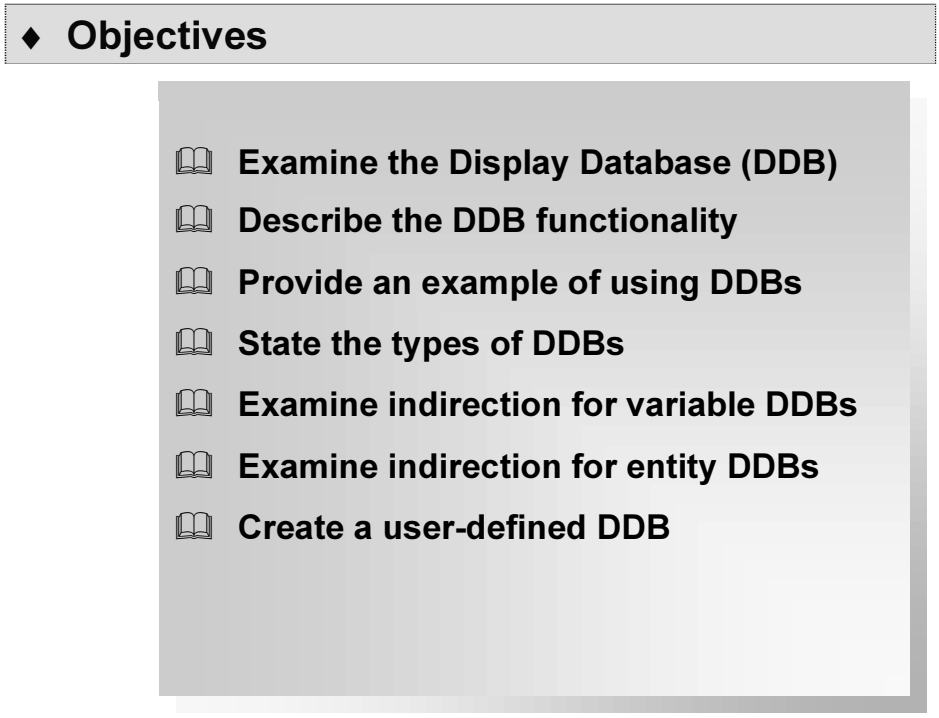

Display Builder DataBase



Objectives

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

- Examine the Display Database (DDB) object in terms of the following:
    - What the Display Database (DDB) object represents
    - Purpose of the Display Database (DDB)
    - Interpret the syntax of the DISPDB object
- Describe how the DDB functionality is supported in GUS and differs from Universal Station DDBs.
- Provide an example application of using DDBs
- State the types of DDBs in terms of
    - Standard DDBS
    - System DDBs
    - User defined DDBs

♦ **Objectives**, continued

- 🕮 **Examine the Display Database (DDB)**
- 🕮 **Describe the DDB functionality**
- 🕮 **Provide an example of using DDBs**
- 🕮 **State the types of DDBs**
- 🕮 **Examine indirection for variable DDBs**
- 🕮 **Examine indirection for entity DDBs**
- 🕮 **Create a user-defined DDB**

Objectives

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

- Examine DDB indirection for variable type DDBs in terms of the following:
  - Describe the GetVar function
  - Give an example of using GetVar for an operator entry
  - List other examples of variable indirection
- Examine DDB indirection for entity type DDBs in terms of the following:
  - Describe the GetEnt function
  - Give an example of using GetEnt for an operator entry
  - List other examples of entity indirection
  - Review an example of double indirection
- Create a user defined DDB
  - Interpret the syntax for a user defined DDB
  - Interpret the differences between Universal Station and GUS user-defined DDBs
  - Load a user-defined DDB

Display Database: Purpose

Introduction

Purpose of Display Database

- Provides support for translated US Displays
- Create a “generic display”

Introduction

As stated in an earlier discussion, the Display Database (DISPDB) in a GUS display represents an object. As an object, it has properties. In everyday terms, the Display Database object properties become available to display builders as a collection of names. One example name is “INT01,” which means “local Display Database Integer #1.” Another example name is “STRING01,” which means “local Display Database string 01.”

Purpose of Display Database

Two reasons for using the Display Database (DDB) are

•The Display Database provides support for translated Universal Station Displays
•The Display Database allows you to create what could be called a “generic display”

Translation support from Universal Stations

The Display Database (DDB) object provides support for migrating or translating displays from the world of Universal Stations to the GUS platform. If you need more information on how DDBs work in a Universal Station, refer to the Actor manual for more information.

Creation of a “generic display”

The Display Database (DDB) object also can be used for the creation of a generic display. A generic display, for the purpose of this discussion, is a display that can be used with multiple sets of similar equipment (e.g., Column #1, Column #2, Column #3, and so on.).

Display Database: Two Types of Data Base

Two Types of Display Database

- Global
- Local

Definition of Global versus Local

Difference between Global and Local

Two types of Display Database

Before reviewing an example application for the Display Database, first examine the two types of Display Database.

- The Global Display Database (DDB)
- The Local Display Database (DDB)

Each of these databases provide useful variables for your GUS Displays.

Global DDB

Values stored in the Global DDB persist as long as a GUS display is called up (or technically, as long as the GUS runtime process, GPB.EXE, is executing). The Global DDB is used to pass information from one GUS display to another GUS display using the InvokeDisplay statement. A global DDB item always has a trailing "g" in its name, as in INT01G, which represents global DDB integer #1.

Local DDB

In a GUS display, the values of local DDB items persist only while a particular display is called up. They are initialized when the display starts. A local DDB item does not have any extra character in its name. For example, INT01 represents local DDB integer #1.

What is the difference between Global and Local DDB?

The values in local DDB items exist only during the life of the display. They are reinitialized whenever a new display is invoked. The values in the global DDB persist across picture invocations. They are initialized when a picture starts up. The global DDB is used to pass data values from one display to a newly invoked display. You could consider the passing of global data in GUS as a ‘one-time, one-way’ copy.

Display Database: GUS vs US Behavior

Global Display Database Behavior

- Different than US
- Global DDB not shared
- Global DDB passed in “InvokeDisplay”

How the Global DDB behavior in GUS differs from US

The global DDB in GUS is quite different than it is in a classic TDC 3000 Universal Station because there can be more than one display running on the GUS. The global DDB in the GUS Display Builder is designed to allow data to be passed between an invoked display and the invoking display when using the InvokeDisplay statement. Note that the global DDB is not shared across displays in the GUS as it is in the Universal Station, rather the global DDB is simply initialized differently when the display is invoked by using InvokeDisplay.

Display Database: Example Application

Example application

- Single display for different views
- Reduces authoring
- Menu invokes single display

Example Application: Generic Display for Similar Equipment

One useful application of the Global DDB is the creation of a single GUS display that can be used to show different views of the same class of process equipment through one display. For example, a site may have four reactors, all of which are similar in appearance and operation. The differences in the reactors are the control points that need to be displayed (i.e., the LCN.point.parameters). Having a single display for all reactors reduces the cost of authoring and maintaining the displays because there is now one display to maintain instead of four.

To clarify this concept, at the end of this section you will complete a lab that represents a site with four reactors. You will create a "reactor menu" that allows an operator to select a reactor of interest. After selecting a particular "reactor" from the menu, a generic reactor display is invoked regardless of the menu selection. However, what's different is that the global DDB values are changed prior to display invocation and, according to the reactor selected, different values are passed to the newly invoked display.

Display Database: Application Approach

Example application approach

- Indirect access
- Menu selection invokes single display
- Script references entities

Example Application Approach

So this approach is summarized as follows. There is only a single Reactor display (named Reactor.PCT) that is capable of displaying the values of the points for any of the four reactors in the plant. Additionally, the following approach is used:

- References to LCN points in the 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, which represent global string variable #1.
- Each of the menu selections that invokes the Reactor display stores the name of the Reactor (e.g., "Reactor 1") into String01G, and stores the names of the data points for the reactor into ENT01G and ENT02G, then the Reactor display is invoked (e.g., Reactor.PCT).
- The script uses an indirect reference to entities. With the SET operator, the data reference is bound at build time. For example, the script statement:

SET DISPDB.ENT01G.external = "FIC100"

Does the following:
DISPDB represents the Display Database (DDB) object
ENT01G represents a global entity of the display DDB object
External represents a string property of the DDB object; here it stores the tagname "FIC100"
SET build time binds ENT01 to FIC100

Display Database: Properties Local, Global

DDB Items are DispDB Object Properties

- Variables naming indicates type
- Use prefix of DispDB

Standard and Global DDB Items

Data Type	Standard Local DDB item	Standard Global DDB item
integer	INT01 - INT256	INT01G - INT256G
float	REAL01 - REAL256	REAL01G - REAL256G
integer[1]	BOOL01 - BOOL256	BOOL01G - BOOL256G
string	STRING01 - STRING256	STR01G - STR256G
point (entity)	ENT01 - ENT256	ENT01G - ENT256G
enumeration[2]	ENM01 - ENM256	ENM01G - ENM256G
variable[3]	VAR01 - VAR256	VAR01G - VAR256G
time	DATIME1 - DATIME256	DATIME1G - DATIME256G
[1] Storage for Boolean values		
[2] Only standard and custom enumerations. (No self defining.)		
[3] See the discussion on Variable-type DispDB items for more details.		

DDB items are properties of the DispDB object

In the Universal Station's Picture Editor, the DDB appeared to the end user as a collection area of built-in variables in a "global name space." In the GUS Display Builder, the DDB variables appear as properties of the DispDB object, DISPDB. For both the US and GUS, the naming of DDB variables indicates their type and contains a number to distinguish the variable. For example, INT01 represents local DDB integer number 1, STRING01 represents local DDB string number 1. Because the DDB represents an object for a GUS display, in order for you to reference a DDB variable requires that you use a prefix of "DispDB". For example, to reference the first integer DDB item in the local DDB, enter DispDB.INT01in your script statement.

DispDB Properties

The following table list the properties of the DispDB object in terms of Local and Global DDB items:

Data Type	Standard Local DDB item names, used for local storage	Standard Global DDB item names, used for global storage
integer	INT01 - INT256	INT01G - INT256G
float	REAL01 - REAL256	REAL01G - REAL256G
integer[1]	BOOL01 - BOOL256	BOOL01G - BOOL256G
string	STRING01 - STRING256	STR01G - STR256G
point (entity)	ENT01 - ENT256	ENT01G - ENT256G
enumeration[2]	ENM01 - ENM256	ENM01G - ENM256G
variable[3]	VAR01 - VAR256	VAR01G - VAR256G
time	DATIME1 - DATIME256	DATIME1G - DATIME256G
[1] Storage for Boolean values		
[2] Only standard and custom enumerations can be stored here. (No self defining.)		
[3] See the discussion on Variable-type DispDB items for more details.		

System DDB : Console, Area

System DDB Items: Console, Area

Current Area/Console Data		
DDB item	Data type	Description
$AL_ENTY	variable	An entity variable that provides the name of a point that is selected on the Alarm Summary, Unit Alarm Summary, Alarm Annunciation, or Organizational Summary displays.
$ARAID	variable	An 8-character string name for the *current* area
$ARADSC	variable	A 24-character string description of the current area.
$CONDSC	variable	A 24-character string description of the current console.
$CONNUM	variable	An integer that represents the *current* console number.
$KEYLEVL	variable	An enumeration that contains the internal keyswitch level for this US. The values are: VIEW, OPR, SUP, and ENGR
$MY_AREA	variable	An integer that represents the area number on which the custom display is running.
$MY_PNA	variable	An integer that represents the node number on which the custom display is running.
$STNNUM	variable	An integer that represents the station number on which the custom display is running.
Other Area/Console Data		
DDB item	Data type	Description
$ARAID01 – $ARAID10	variable	An 8-character string name for each indicated area as configured in the NCF.
$ARADS01 – $ARADS10	variable	A 24-character string description for each indicated area as configured in the NCF.
$CONDS01– $CONDS10	variable	A 24-character string description of the indicated console as configured in the NCF.
$GRPBASE	variable	An entity ID that can be used to access the group definitions

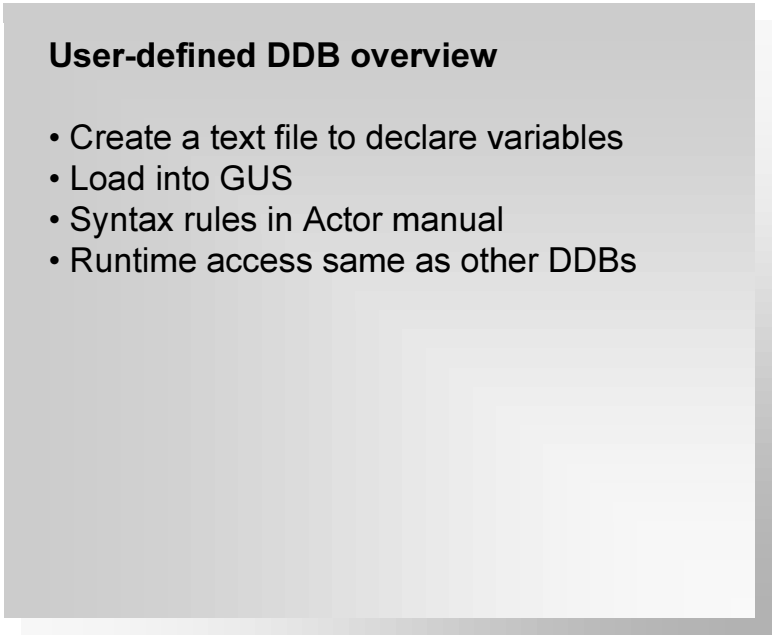
System DDB: Change Zone, Alarm

System DDB Items:

- Change Zone
- Alarm Configuration

Change Zone Data		
DDB item	Data type	Description
$CZ_ENTY	variable	An entity variable for displaying a Change Zone. For more information, refer to Change Zones.
System wide Alarm Configuration Data		
DDB item	Data type	Description
$ALMCOLR	variable	An integer that specifies the alarm priority color option selected in the NCF: 0 = two color option (red, yellow), 1 = 3 color option (user selected colors)
$EALMCLR $HALMCLR $LALMCLR	variable	An integer that specifies the Alarm priority color option selected in the NCF for Emergency, high, and Low priority alarms (1=red, 2=green, 3=yellow, 4=blue, 5=magenta, 6=cyan, 7=white.).
$CONDS01– $CONDS10	variable	A 24-character string description of the indicated console as configured in the NCF.
$REDYEL	variable	An enumeration that contains the minimum alarm priority indicated by the color red. The enumeration set is ALPRIOR. It contains the members LOW, HIGH, and EMERGNCY.

User Defined DDB



User defined DDB items

Unlike the standard DDB, which contains names defined by Honeywell, the user-defined DDB contains variable names defined by the user. To create a user-defined DDB you create a text file. The text contains statements declaring the names and indices of the variables. The text file is then loaded into the GUS during a display build session. All of the Syntax Rules for declaring user-defined DDB variables are found in the LCN Actors Manual, Section A3 User-Defined DDB Variables. A GUS user defined DDB differs from a classic Universal Station's in that:

- There is no "maximum number" of DDB variable names per file.
- There is no "memory limitation" for User-Defined global DDB Variables.

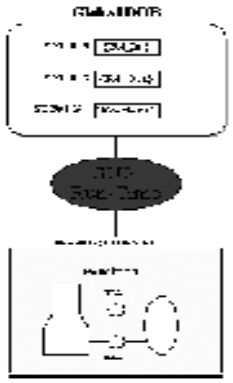
At runtime, all of these user-defined DDB elements are accessible in the same manner as standard DDBs except they are not automatically initialized. For this reason, you must first initialize a user-defined DDB before reading a value from it.

GUS Display Relationship to Global DDB

- **Each display has its own run time process**
- **Each display has its own global DDB**
- **Pass data using InvokeDisplay**
  - global DDB not shared across displays
  - InvokeDisplay initializes global DDB

Relationship Between Displays, GUS Run Time, and Global DDBs

The figure below illustrates the relationship between a display, the GUS run time process (GPB.EXE) 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. In other words, each display has its own global DDB. The global DDB in GUS is quite different than it is in TDC 3000 (TPS3000) because there can be more than one display running on the station. The global DDB in the GUS Display Builder is designed to allow data to be passed between an invoked display and the invoking display when using the InvokeDisplay statement. That is, the global DDB of a display invoked by using InvokeDisplay is initialized to the values in the DDB of the invoking display. Note that the global DDB is not shared across displays in the station, rather it is simply initialized differently when the display is invoked by using InvokeDisplay



Display Database Indirection

To access DDB, use indirection

DispDB Indirection: Two Types

- Variable
- Entity

DISPDB indirection

In order to access the DISPDB, you need to use a scripting technique called indirection. The technique of using DISPDB indirection in a GUS display script is similar to that in a Universal Station Picture Editor. Indirection is a concept of accessing data via references. There are two types of DISPDB indirection:

- Variable
- Entity

Display Database Indirection: Variable

DispDB Variable Indirection:

- Stores reference to “point.parameter”
- Bind at build time for performance
  - Use SET operator (preferred), or
  - Use GetVar
  - GetVar can support runtime operator entry

Variable-type DISPDB indirection

Variable-type DISPDB items are used to contain a reference to a parameter on a particular point. So in that sense a reference is like an alias. For example, if DISPDB.VAR01 is set to reference FIC100.PV, displaying the value of DISPDB.VAR01 is the same as a displaying the value of FIC100.PV. Variable IDs are generally bound at build time. There are multiple ways available for variable type indirection , but the preferred expression is to use a SET operator as follows:

```
SET Dispdb.var01 = LCN.FIC100.pv
```

In this example, the set keyword identifies that a reference to the parameter is being made, and the right hand side of the expression is the object being referenced. In effect, VAR01 is now a substitute for the value of FIC100.PV. More importantly, the SET keyword means that FIC100.PV is build time, not run time, bound, to VAR01.

The GetVar function can also be used to generate a reference that can be stored in a VAR item along with using a SET statement. First, consider the definition of the GetVar function:

```
Dim GetVar(Name as string) as object
```

The argument Name represents the name of the variable, for example FIC100.PV. The function returns an object that a DispDB.Var object can be set to reference.If the argument is a constant string such as “FIC100.PV” the reference is bound at build time. The GetVar function is useful when you wish to access a point.parameter whose name was input from the operator. The following example uses the script Askbox function, which allows an operator to enter a tagname variable string, such as FIC100.PV.(In this case the reference is bound at run time.)

```
GetVar(Askbox$(“EnterVariable”))
```

When setting a VAR item in a SET statement, use the GetVar function on the right-hand side of the assignment to generate the reference to be stored in the VAR item. The following example shows the use of the set statement with VAR items:

```
set DispDB.var01 = GetVar ("FIC100.pv")   ‘ make DispDB.var01 reference FIC100.pv
```

DDB Indirection: Variable Examples

Variable Indirection to point.parameter

Example	Comment
set dispdb.var01 = lcn.A100.pv DispDB.var01 = 3 me.text = DispDB.var01	The statements first set a VAR to A100.pv, then assign a value of "3" to a100.pv. The text object displays the value of a100.pv.
set dispdb.var01 = getvar("A100.num1")	OK
set dispdb.var01 = lcn.A100.num1	OK
set dispdb.var01 = lcn.A100.foo	Buildtime error: Invalid variable name
set dispdb.var01 = display.params.p1	where p1 is of variable type, OK
set dispdb.var01 = display.params.p1	where p1 is of entity type, Runtime error 13 - type mismatch
dim o as object set o = lcn.A100.num1 x = o	OK (but it does not trigger an On Data Change script)
set dispdb.var01 = lcn.A100.num_par(i)	OK - Runtime bound (slower) if i is not constant
set dispdb.var01 = dispdb.var02	OK
set dispdb.var01 = GetVar(askbox("enter variable ID"))	OK - Runtime bound (slower) because operator makes entry at runtime

Display Database Indirection: Entity

DispDB Entity Indirection:

- Stores reference to “point”
- Bind at build time for performance
  - Use SET operator (preferred), or
  - Use GetEnt
  - GetEnt can support run time operator entry

DISPDB Entity-type Indirection

When you require an alias point name in your display, you can substitute entity type DDB variables for the point names. The Entity IDs are now generally bound at build time. Multiple user syntax expressions are available for Entity IDs. The preferred expression, which binds at build time, is as follows:

```
SET Dispdb.ent01 = lcn.FIC100
```

In this form, the set keyword identifies that a reference to an entity is being made, and the right hand side of the expression is the object (in this case, the tagname FIC100) being referenced. The functional form (i.e., GetEnt) of the reference is also provided. The GetEnt function is consistent with the GetVar function. In this form, the lcn identifier string is provided as an argument to a function that returns the referenced object, as follows:

```
SET Dispdb.ent01 = GetEnt("a100")
```

If the argument is a constant string such as “a100” the reference is bound at build time.

If the argument is a variable string as in

```
GetEnt(Askbox$(“EnterEntity”))
```

Then the reference is bound at run time. This example is useful for operator entered tagnames.

DDB Indirection: Entity Examples

Entity Indirection to point (tagname)

Example	Comment
Set dispdb.ent01 = GetEnt("A100")	OK
Set Dispdb.ent01 = lcn.A100	OK
set dispdb.ent01 = lcn.A100.num1	Runtime Error 1058
set dispdb.ent01 = lcn.foo	Buildtime error: Invalid Name
set lcn.A100.ent_par(2) = GetEnt("A100")	OK
set lcn.A100.ent_par(Dispdb.int01) = GetEnt("A100")	OK
dim o as object set o = lcn.A100 x = o.pv	OK (does not trigger a data change script)
set dispdb.ent01 = lcn.A100.ent1	OK
dispdb.ent01 = lcn.A100.ent1	OK (ent01 and ent 1 reference the same entity)
set dispdb.ent01 = display.params.p1	where p1 is entity type, OK
set dispdb.ent01 = display.params.p1	where p1 is variable type, Runtime error 13 - type mismatch
set dispdb.ent01 = dispdb.ent02	OK
set dispdb.ent01 = GetEnt(askbox("enter entity ID"))	OK Runtime bound (slower)
set dispdb.ent01 = lcn.A100.ent_par(i)	OK Runtime bound (slower) if i is not constant

Note: Double indirection is also possible: an entity-type parameter may contain a reference to an entity-type parameter. For example:

```
SET lcn.a100.ent1 = lcn.a101        'A100 has an ent1 parameter of type entity, set it up to reference
                                    'a101.
SET Dispdb.var01 = lcn.a100.ent1    'Set the ent item to reference a100.
me.text=dispdb.ent01.ent1.num10     'The value of a101.num10 is displayed.
```

Note: The assignment syntax for entity IDs is also still supported:

```
Dispdb.ent01 = "a100"
Dispdb.ent01.external = "a100"
```

This syntax is not checked nor bound at build time and will run slower! It is recommended that this form not be used for new displays and that old displays using this form be re-written to use the new expression.

Display Database: User defined DDB

- **Three steps to creating user-defined DDB**
  - 1 Create a text file to declare variables
  - 2 Type in variable statements
  - 3 Load user-defined DDB into display
- **Example syntax**

Variable Name : Variable Type , Index , Disposition ; {Comment}
Example:
DCFI: INTEGER, 100, GLOBAL; {flash index}

Creating a user-defined DISPDB

To create a user defined DDB requires three steps

1. Create a text file with a .DF, "declare file" extension.
2. Type in statements to declare the variables using correct syntax
3. Load the user-defined DDB from the Display pull-down menu.

The User DDB Declare File Syntax is shown in the following table.

Variable Name : Variable Type , Index , Disposition ; { Comment }
Example:
DCFI: INTEGER, 100, GLOBAL; {flash index}

Notes:
1. A statement must begin and end on the same line.
2. More than one statement can appear on a line.
3. Spaces can appear anywhere, except in identifiers such as the Variable Name.

Variable Name
- Up to eight characters (A-Z, a-z, 0-9, and underbar)
- First character must be a letter. Last character cannot be an underbar, and there cannot be two consecutive underbars.

Variable Type	**Synonyms**
Integer	Int, I
Number	Num, Real, R
Boolean	B, Logical, L
String	S
Enum:<data type>	E
Date_Time	Date/Time, Date
Variable	Var, V
Entity_ID	Entity, Ent

Index
Index numbers **100 – 32767** are available for **each type** of global and local user DDB variables.

Disposition	**Synonyms**
Local	Loc, L
Global	Glob, G

Comment
- A-Z, a-z, 0-9, any special characters.
- Must be enclosed in braces { }. Must begin and end on the same line.
- Can appear before and/or after the declare statement.

User-defined DDB: Declare File Syntax

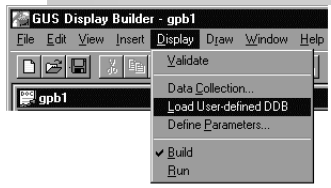
- **Syntax rules in Actors manual apply.**
- **Following rules apply to GUS (not US):**
  - There is an additional overwrite option for modifying a previous DDB declaration statement that contains name duplication. The default value of this option is "False" or "F." Note , however, that this overwrite option cannot be used to duplicate either names in the DDB file or standard DDB names.
  - There is no "maximum number" of DDB variable names per file.
  - There is no "memory limitation" for User-Defined global DDB Variables.
- **Example file declarations**

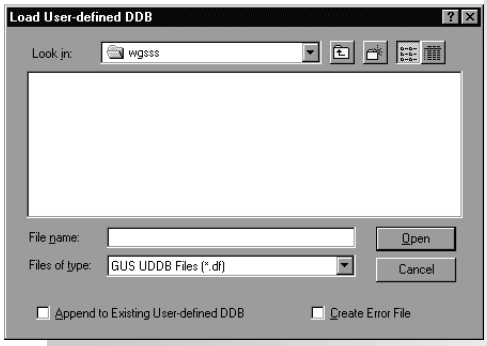
```
SEL_ENT : ENTITY , 200, G ;
my_int: int, 100, local;
REAL01: R, 500, L;
BOOL50: Boolean, 500, L;
XYZZY : STRING, 400, G;
```

Example Declare File

```
{ Note that blank lines are legal }
SEL_ENT : ENTITY , 200, G ;    { Spaces are allowed }
SEL_ENT2: ENT,200,L;           { but not necessary. }

my_int: int, 100, local;       { Lower case is okay. }

REAL01: R, 500, L;             { This will not be recognized }
                               { because it is a duplicate of }
                               { a Standard DDB name. }

BOOL50: Boolean, 500, L;       { This is okay. }
BOOL50: Boolean, 250, L;       { This will not be recognized }
                               { because it is a duplicate name. }

XYZZY : STRING, 400, G; { This is okay. }
PLOVER: STRING, 400, G; { This is an alias of XYZZY. }

My_Date:Date_Time, 700, L; My_Int3:Integer, 5, GLOBAL;
{ This is okay }  My_Enum : ENUM:MODE, 450, LOCAL;
{ All of the above are syntactically correct. }

{ All of the following have a syntax error. }
_Int: INTEGER, 180, G;         { Cannot start with underbar. }
10Real: REAL, 190, Glob;       { Cannot start with numeric. }
STR__5: S, 230, G;             { Consecutive underbars are illegal. }
INTEGER20 : INTEGER, 260, L;   { Name is longer than 8 characters. }
My_Var_1 : VAR, 330, G         { Missing semicolon. }
My_Ent : ENTITY, Loc;          { Missing index. }
My_Bool: LOGICAL, { } 400, G; { Comments within a statement. }
My_Real: REAL, 50, G;          { Index out of range.}
```

User-defined DDB: Loading a file





- **Load from Display>Load User-defined DDB**
- **Initialize DDB at runt ime**
- **Create error file as option**

Example Declare File

You can load a user-defined DDB from the Display>Load User-defined DDB menu item.
A user defined loading dialog box appears. Use the dialog box to navigate and find your user-defined DDB containing a ".df" extension.

At run time, all of these user-defined DDB elements are accessible in the same manner as standard DDBs except they are not automatically initialized. For this reason, users must first initialize a user-defined DDB before reading a value from it.

Create Error File: If this checkbox is selected, an error file with the same path and filename as the ".df" file, but with file extension ".err" will be created. Information about errors encountered during processing of the ".df" file will be placed in the error file.

User-defined DDB: Enumerations

- **Display standard or custom enumerations**
- **Can use enum function**
  - must know set and member names
- **DDB usage**
  - external form not recommended
  - do not assign self-defining enumeration

Example Enumeration Syntax

You can use the Enum function to display all or a portion of a Standard or Custom Enumeration. You must know the LCN SetName and MemberName to use this function.
Example:
Determining the MemberNumber:
me.text = enum("SetName:MemberName").MemberNumber
me.Text = enum("BoxColor:Red").MemberNumber

DDB usage: (Honeywell does not recommend that the .External form be used for DDB)

To access the Internal form use:
DispDB.Enm01.Internal = lcn.point.parameter.Internal
me.text = DispDB.Enm01.Internal

To access the External form use:
DispDB.Enm01.External = lcn.point.parameter.External
me.text = DispDB.Enm01.External

Modifying an LCN enumeration value using an LCN object
lcn.point.parameter = "SetMember"

DDB examples
if DispDB.Enm01.External = "SetMember" Then ...
if DispDB.Enm01.External = "Open" Then ...
if DispDB.Enm01 = lcn.point.param.External Then ...
if DispDB.Enm01.External = lcn.point.param.External Then ...
if DispDB.Enm01.External = lcn.point.param.Value Then ...
if DispDB.Enm01.Internal = lcn.point.param.Internal Then ...

Self DefiningEnumeration
Do NOT assign a Self Defining Enumeration to the DDB