

# *Welcome to Ghidra Help...*

Ghidra provides context sensitive help on menu items, dialogs, buttons, and tool windows. To access the help, press <F1> or <Help> on any menu item or dialog. If specific help is not available for an item, this page will be displayed.

# Introduction

Ghidra is a highly extensible application for performing software reverse engineering. Ghidra is built upon a completely generic application framework. Application-specific capabilities are provided by small software bundles called plugins, each providing one or more features. This user's guide provides detailed information on how to use both Ghidra's generic and reverse engineering-specific capabilities.

## Intended Audience

This guide is intended for anyone interested in learning how to use Ghidra to reverse engineer a software system.

## Document Scope

The purpose of this document is to describe how to use Ghidra. It does NOT provide information on the software architecture or the programming API.

## Disclaimer

Ghidra is configurable. At any given time, Ghidra capabilities can be added, removed, or even replaced by changing the current set of plugins. Consequently, a feature might not be available as described and the images shown in this document may not exactly match your display.

# Getting Started

## File System Layout

In the directory you choose, a ghidra installation directory named **ghidra\_<version>** is created. The following directory structure will be created under the Ghidra installation directory.

docs	tutorial and on-line help
Extensions	installable extensions for Ghidra, Eclipse and IDA Pro
Ghidra	essential files for running Ghidra
GPL	GPL utility and support programs used by Ghidra
licenses	licenses for non-GPL portions of Ghidra

server	files required to launch and configure the Ghidra server
support	files useful for debugging and configuring Ghidra

## Extensions

There are a number of Ghidra plugins that are not part of the base distribution. They are either experimental, still under development, or contributed by others. These plugins may not have been tested, and therefore may be unstable and are not included in any of the default tools. However, these plugins often contain the more cutting edge features and may be worth considering. They are easily accessible and can be added by [configuring](#) a tool.

## IDA Pro Export

The Ghidra distribution includes a plugin for use with IDA Pro (a commercially available disassembler). The XML plugin is used with IDA Pro to export IDA Pro databases as XML files so that they can be imported into Ghidra. This allows IDA Pro users to migrate their data to Ghidra.

To add the XML exporter plugin to your IDA installation:

- Locate the README file for your version of IDA from the version folders in the <ghidra installation directory>/Extensions/IDAPro folder. The plugin is available for IDA Pro versions 6 and 7. If you are unsure of your IDA version, start IDA and select Help → About program ... from IDA's main menu to display the version.

To export data to Ghidra using the XML plugin, select File → Plugins → Dump database as XML file... from IDA's main menu.

## Starting Ghidra

Launching Ghidra varies depending on the operating system.

### Ghidra on Windows:

Run the `ghidraRun.bat` file located in the Ghidra installation directory.



*One way to run this file is to use the Windows file explorer to locate the `ghidra.bat` file and then simply double click on the file.*

### Ghidra on Linux and macOS:

Run the `ghidraRunshell` script file located in the Ghidra installation

directory.

### Advanced Startup

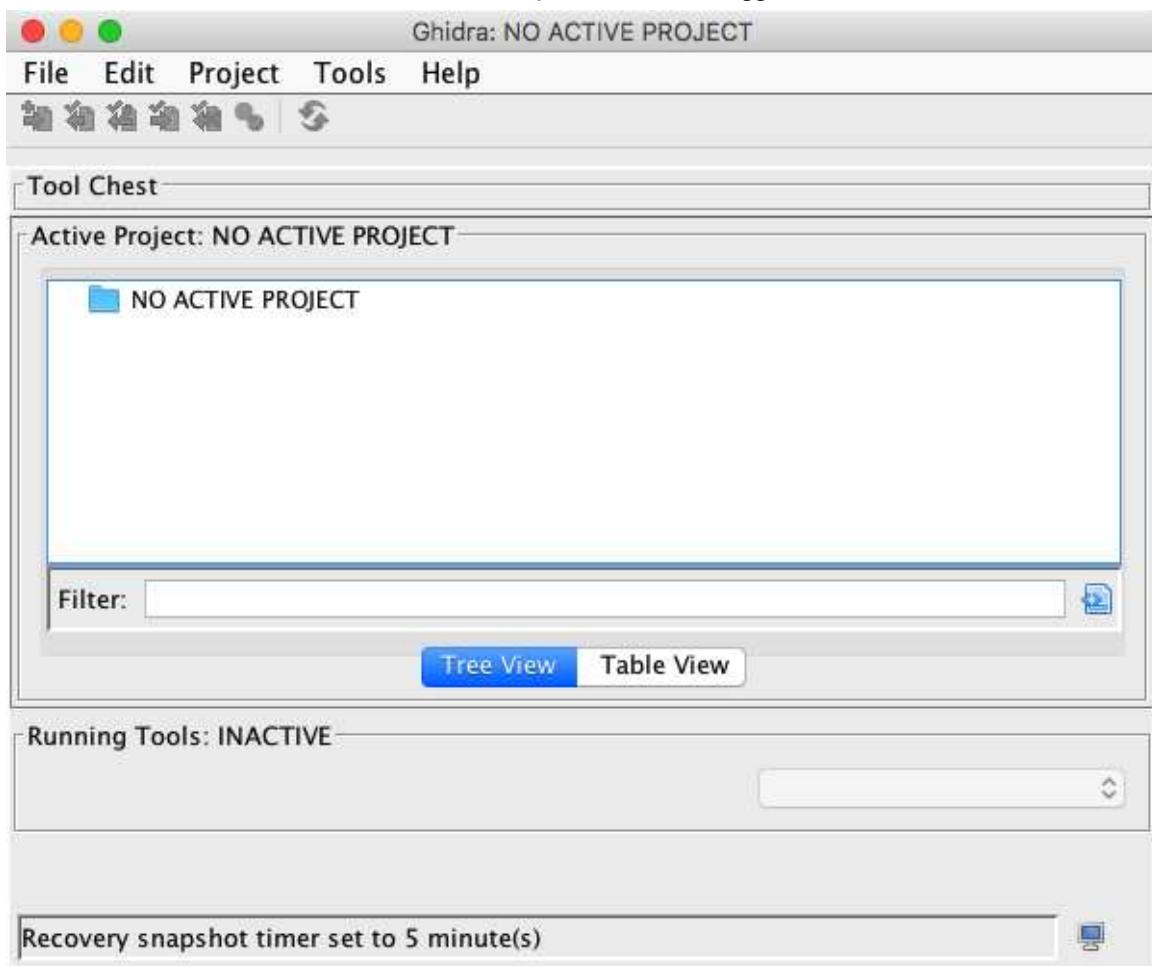
Ghidra provides some Java startup parameters which allow for the usage of advanced features. To use a startup parameter you must open `support/launch.properties` and add the parameter to that file.

For example,

```
VMARGS=-Dfont.size.override=18
```

## Ghidra Overview

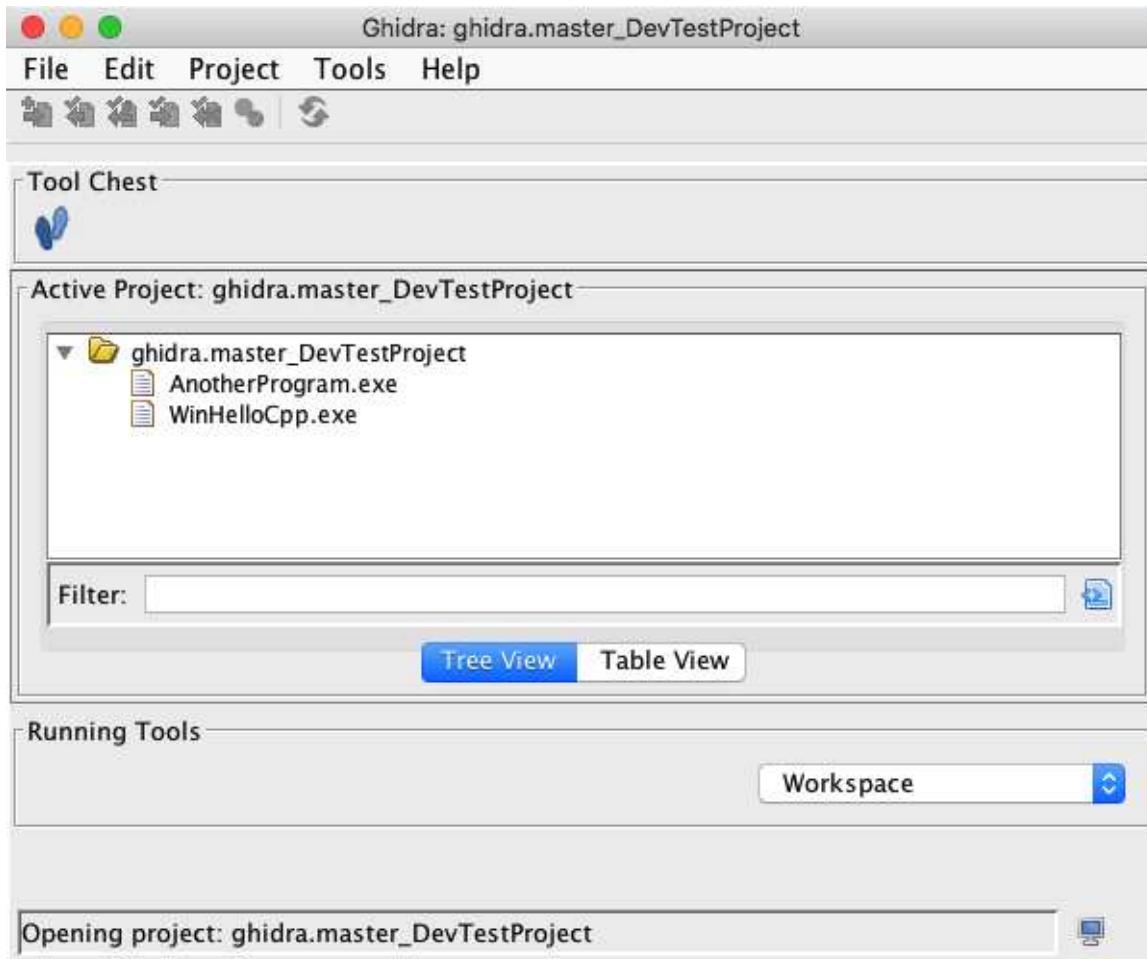
When Ghidra first starts, the [Ghidra Project Window](#) will appear.



Ghidra Front-End with no open project

Ghidra is a project-oriented application and, consequently, all work must be performed in the context of a project. Therefore, the first thing to do is to [create](#) a project or [open](#) an

existing project. Once a project is open, Ghidra will display the folders and data that make up the project along with the user's current set of tools. Of course, newly created projects would not contain any data. Data must be [imported](#) into a Ghidra project before any work can be performed. Importing data into a Ghidra project creates [programs](#) that Ghidra [tools](#) can manipulate.



Ghidra Front-End with an open project

A Ghidra tool is a configuration of plugins that can be used to manipulate programs. When Ghidra is first installed, a default tool –the code browser tool –is created for the user and its icon is displayed in the Tool Chest area of the Ghidra Project Window.

To [run a tool](#), click on its icon in the *Tool Chest*. When a tool is running, a new window will appear for that tool and the tool's icon will be displayed in the *Running Tools* area of the Ghidra Project Window.

Ghidra also supports the concept of *workspaces*. A [workspace](#) is simply a collection of running tools that are visible on the desktop. Users can have multiple workspaces, each with its own set of running tools. Running tools that are not in the current workspace are still running and consuming system resources even though they are not visible.

## Error Dialogs

Errors may occur in Ghidra. An error may be anticipated, or it may be unexpected in

which case it is a programming error. Each type of error is described below.

### General Errors

Whenever an action or operation does not complete as desired, but is an anticipated error such as a user entering a file path that doesn't exist, Ghidra will display an Error dialog explaining the cause of the problem as shown below in the sample error dialog:



### Unexpected Programming Errors

Whenever an action or operation fails in a totally unexpected way, i.e., a programming error, a dialog is displayed as shown below:



The **Details >>>** button expands the dialog to show the details of the java stack trace. (The stack trace is also output to the console.)

# Introduction

Ghidra is a highly extensible application for performing software reverse engineering. Ghidra is built upon a completely generic application framework. Application-specific capabilities are provided by small software bundles called plugins, each providing one or more features. This user's guide provides detailed information on how to use both Ghidra's generic and reverse engineering-specific capabilities.

## Intended Audience

This guide is intended for anyone interested in learning how to use Ghidra to reverse engineer a software system.

## Document Scope

The purpose of this document is to describe how to use Ghidra. It does NOT provide information on the software architecture or the programming API.

## Disclaimer

Ghidra is configurable. At any given time, Ghidra capabilities can be added, removed, or even replaced by changing the current set of plugins. Consequently, a feature might not be available as described and the images shown in this document may not exactly match your display.

# Getting Started

## File System Layout

In the directory you choose, a ghidra installation directory named **ghidra\_<version>** is created. The following directory structure will be created under the Ghidra installation directory.

docs	tutorial and on-line help
Extensions	installable extensions for Ghidra, Eclipse and IDA Pro
Ghidra	essential files for running Ghidra
GPL	GPL utility and support programs used by Ghidra
licenses	licenses for non-GPL portions of Ghidra

server	files required to launch and configure the Ghidra server
support	files useful for debugging and configuring Ghidra

## Extensions

There are a number of Ghidra plugins that are not part of the base distribution. They are either experimental, still under development, or contributed by others. These plugins may not have been tested, and therefore may be unstable and are not included in any of the default tools. However, these plugins often contain the more cutting edge features and may be worth considering. They are easily accessible and can be added by [configuring](#) a tool.

## IDA Pro Export

The Ghidra distribution includes a plugin for use with IDA Pro (a commercially available disassembler). The XML plugin is used with IDA Pro to export IDA Pro databases as XML files so that they can be imported into Ghidra. This allows IDA Pro users to migrate their data to Ghidra.

To add the XML exporter plugin to your IDA installation:

- Locate the README file for your version of IDA from the version folders in the <ghidra installation directory>/Extensions/IDAPro folder. The plugin is available for IDA Pro versions 6 and 7. If you are unsure of your IDA version, start IDA and select Help → About program ... from IDA's main menu to display the version.

To export data to Ghidra using the XML plugin, select File → Plugins → Dump database as XML file... from IDA's main menu.

## Starting Ghidra

Launching Ghidra varies depending on the operating system.

### Ghidra on Windows:

Run the `ghidraRun.bat` file located in the Ghidra installation directory.



*One way to run this file is to use the Windows file explorer to locate the `ghidra.bat` file and then simply double click on the file.*

### Ghidra on Linux and macOS:

Run the `ghidraRunshell` script file located in the Ghidra installation

directory.

### Advanced Startup

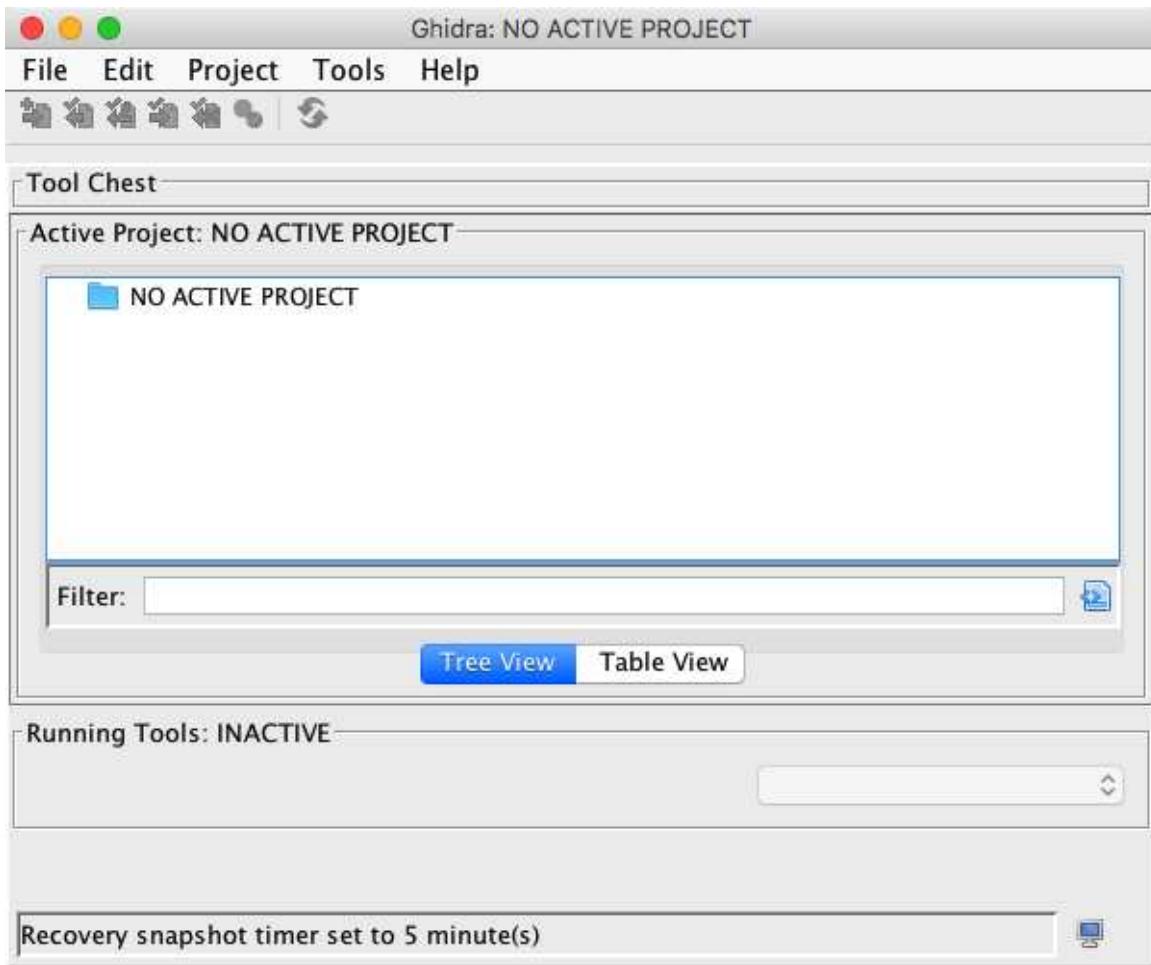
Ghidra provides some Java startup parameters which allow for the usage of advanced features. To use a startup parameter you must open `support/launch.properties` and add the parameter to that file.

For example,

```
VMARGS=-Dfont.size.override=18
```

## Ghidra Overview

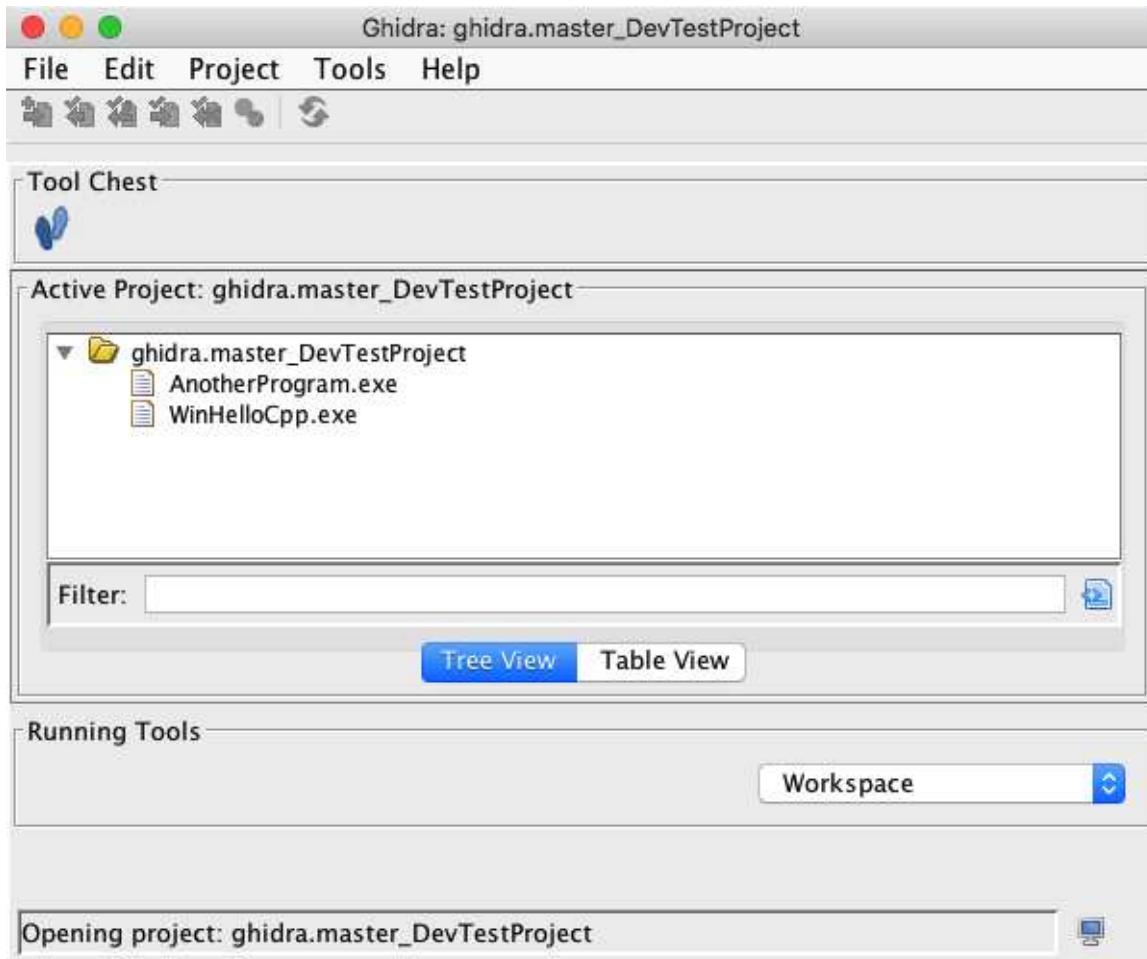
When Ghidra first starts, the [Ghidra Project Window](#) will appear.



Ghidra Front-End with no open project

Ghidra is a project-oriented application and, consequently, all work must be performed in the context of a project. Therefore, the first thing to do is to [create](#) a project or [open](#) an

existing project. Once a project is open, Ghidra will display the folders and data that make up the project along with the user's current set of tools. Of course, newly created projects would not contain any data. Data must be [imported](#) into a Ghidra project before any work can be performed. Importing data into a Ghidra project creates [programs](#) that Ghidra [tools](#) can manipulate.



Ghidra Front-End with an open project

A Ghidra tool is a configuration of plugins that can be used to manipulate programs. When Ghidra is first installed, a default tool –the code browser tool –is created for the user and its icon is displayed in the Tool Chest area of the Ghidra Project Window.

To [run a tool](#), click on its icon in the *Tool Chest*. When a tool is running, a new window will appear for that tool and the tool's icon will be displayed in the *Running Tools* area of the Ghidra Project Window.

Ghidra also supports the concept of *workspaces*. A [workspace](#) is simply a collection of running tools that are visible on the desktop. Users can have multiple workspaces, each with its own set of running tools. Running tools that are not in the current workspace are still running and consuming system resources even though they are not visible.

## Error Dialogs

Errors may occur in Ghidra. An error may be anticipated, or it may be unexpected in

which case it is a programming error. Each type of error is described below.

### General Errors

Whenever an action or operation does not complete as desired, but is an anticipated error such as a user entering a file path that doesn't exist, Ghidra will display an Error dialog explaining the cause of the problem as shown below in the sample error dialog:



### Unexpected Programming Errors

Whenever an action or operation fails in a totally unexpected way, i.e., a programming error, a dialog is displayed as shown below:



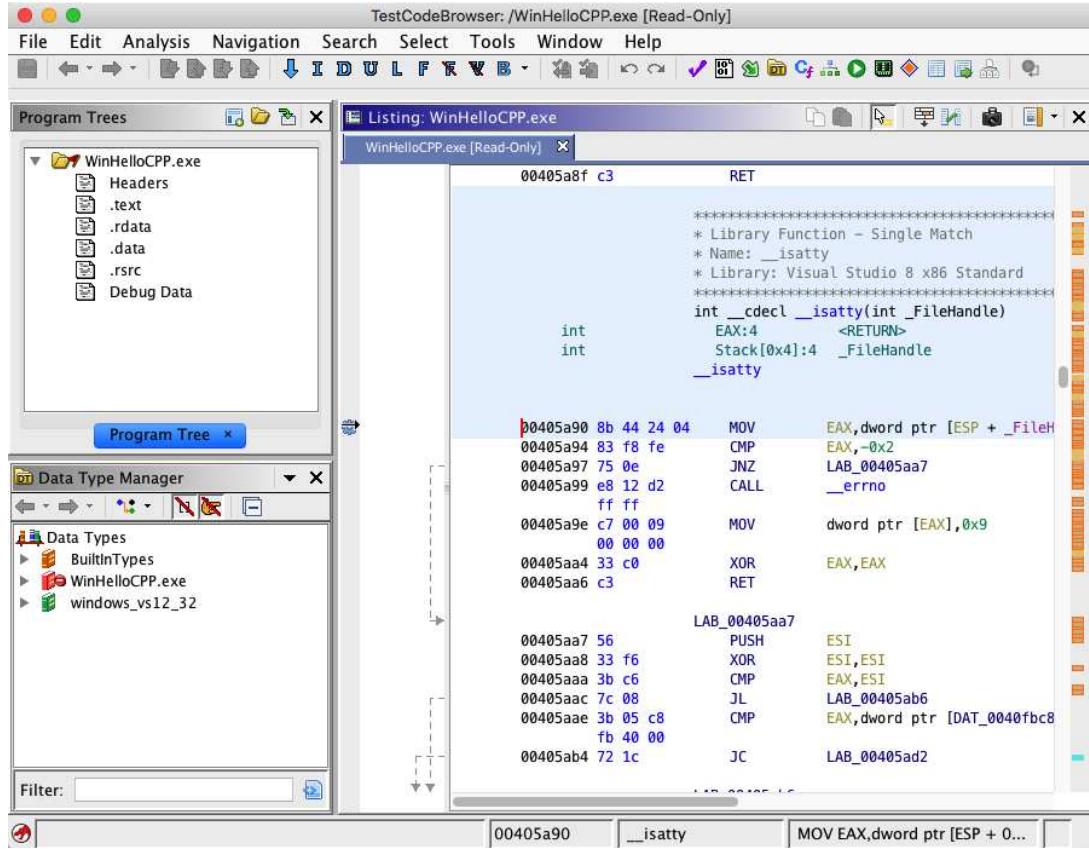
The **Details >>>** button expands the dialog to show the details of the java stack trace. (The stack trace is also output to the console.)

# Docking Windows

Ghidra tools provide various GUI components that allow users to view and manage programs. The Docking Windows feature allows users to customize the layout of these "Dockable" components within a Ghidra tool.

## The Basic Dockable Component

The image below shows a tool with three *Dockable Components*: *Program Trees*, *Listing*, and *Data Type Manager*.



Each component consists of several common parts:

### Title Bar

The title bar is at the top of the component and displays the title of the window. It also serves as a handle for dragging the component to other positions within the tool. The title bar's color indicates whether the component has keyboard focus. When the component has focus, the title bar is blue, otherwise it is gray. Only one component can have focus at a time.

### Local Toolbar

The title bar also serves as a local toolbar. It contains icons for actions that apply only to this component. Hovering the mouse over an icon causes a tool tip to be displayed. The tool tip give a hint as to what the action will do if you select the icon.

### Menu Icon ▾

The menu icon will appear if the dockable component has a local menu. As with the local toolbar, the local menu item applies only to this component. Mouse click on the icon to show the menu.

### Close Icon ✕

Select ✕ to hide the dockable component. The component will be removed until you select the component's name from the tool's [Window](#) menu. Some components, such as search results, are transient and are permanently removed when you close these components.

### Work Area

The work area contains the specific GUI component that is being managed.

## Arranging Components

*Docking Windows* allows users to arrange components into configurations that best fit their needs. There are three ways components can be arranged in a tool:

1. **Docked with other components:** components are side-by-side (or top to bottom) with another component.
2. **Stacked with other components:** components share the same space and tabs are used to display them one at a time.
3. **In their own window:** a component can be placed in its own window.

### Moving Components

Components can be rearranged by dragging them in various ways. To drag a component, press and hold the left mouse button on the title bar of the component to be moved and begin moving the mouse. The mouse cursor will change to indicate what will happen if the mouse button is released at that location.

-  Invalid Location –releasing here will cancel the drag operation.
-  Will move the component to the left of the component that the mouse cursor is over.
-  Will move the component to the right of the component that the mouse cursor is over.
-  Will move the component above the component that the mouse cursor is over.
-  Will move the component below the component that the mouse cursor is over.
-  Will stack the component with the component that the mouse is over, creating a tabbed pane effect.
-  Will place the component in a new window.

 To get the arrow cursors to appear, move the mouse near the inside edge of another component. To get the stack cursor, move the mouse over the middle area of another component. To get the new window cursor, move the mouse over the desktop. The Invalid location cursor will appear when the mouse is over the component being moved.

### Resizing components

Docked components are separated by thin borders. When the mouse is moved over a border which separates two docked components, the cursor will change to a resize  icon. To change the relative size of the components, press the left mouse button while over a border and drag the border in the appropriate direction.

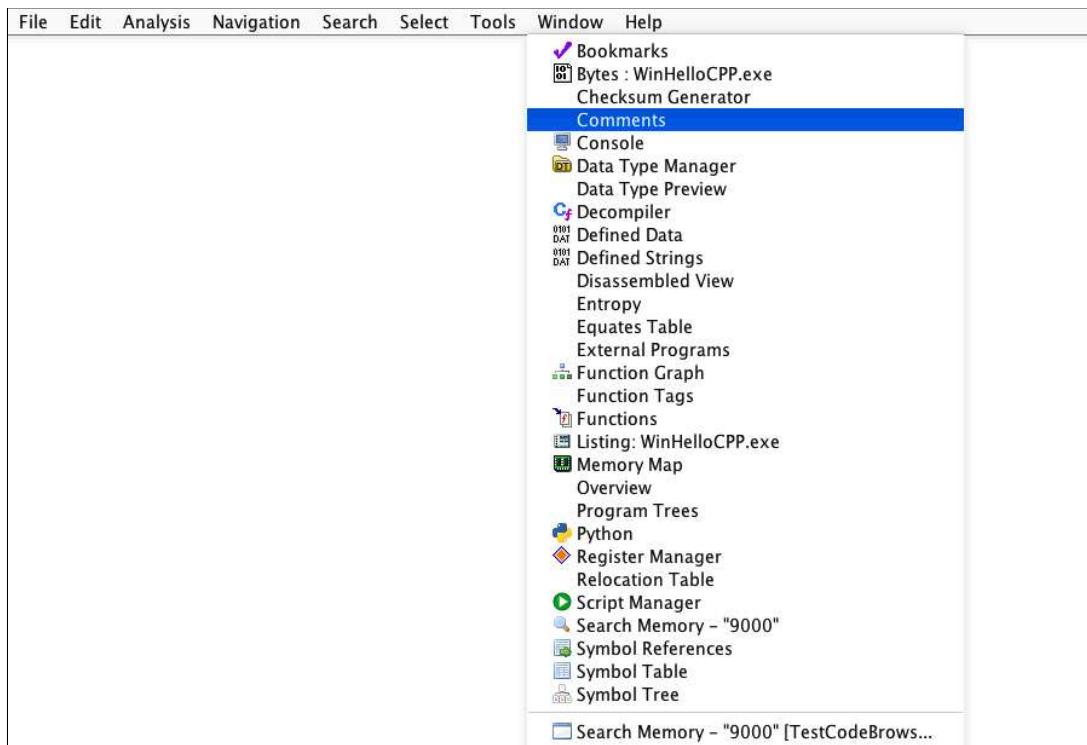
## Renaming Components

**Transient** components (e.g., search windows) can be renamed. Right-click on either the title bar or tab of a transient window and a popup menu item will appear that allows you to change the title of that component. This can be useful when you wish to better identify search results when you have performed many searches.

## Windows Menu

Each component currently loaded in the tool has a corresponding entry in the tool's *Window* menu. The menu item for a component can be used to show a component that is currently not visible. If the component is already showing, it will be brought to the front (if it is behind some other component or windows) and the component will be made active (have keyboard focus). Temporary windows such as search results are shown at the bottom of the menu below the separator bar. Multiple windows of the same type are grouped into sub-menus. For example the following snapshot of a window menu indicates there are several search results windows open.

---



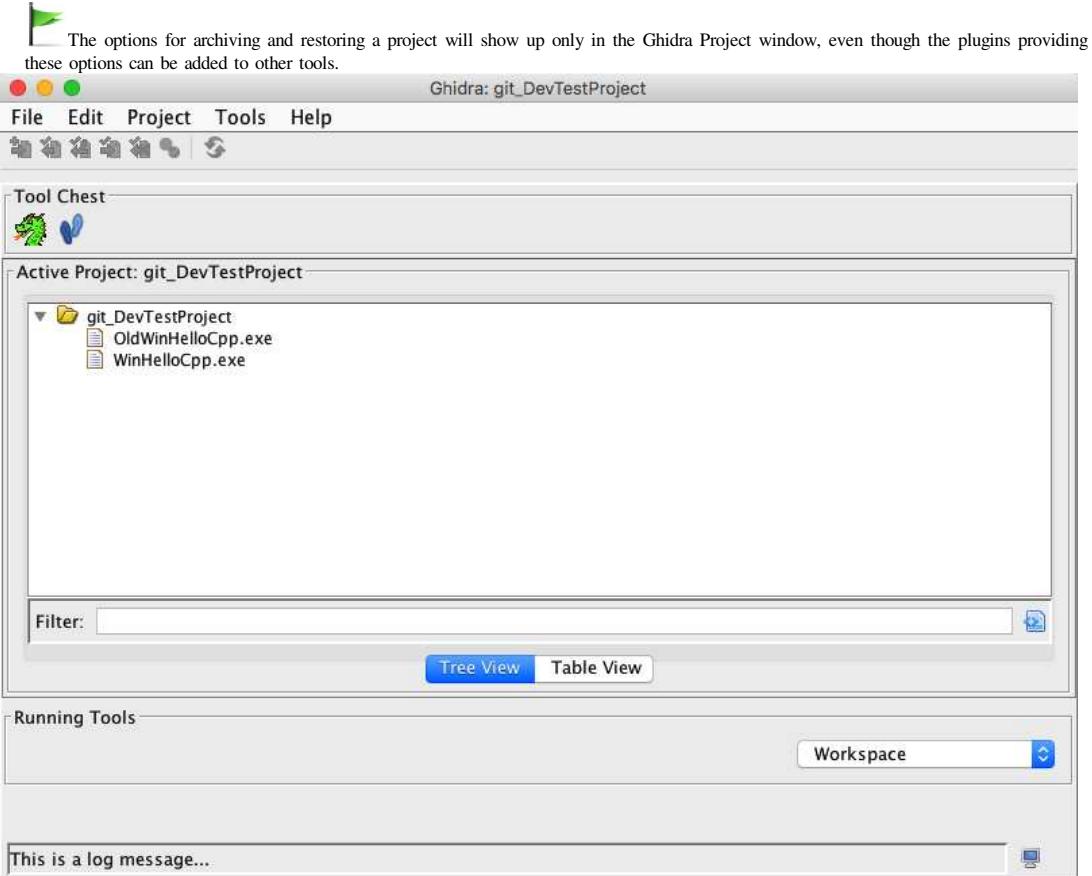
Related Topics:

- [Tool Options](#)
- [Ghidra Tool](#)

## Ghidra Project Window

The Project Window is displayed when you run Ghidra. From this window, you manage your [projects](#), [workspaces](#), [Programs](#), and [tools](#). If you are running Ghidra for the first time, you will need to create a [new project](#) to get started.

The Project Window is a tool and may be [configured](#) with "special" [plugins](#) that provide general capabilities that may be required at a high level. By default, the Project Window contains plugins for [importing](#) and [exporting Programs](#), [archiving a project](#), and [restoring an archived project](#). These plugins may be added to other tools.



The following sections describe the Project Window:

- [Tool Chest](#)
- [Active Project Panel](#)
- [Read-Only Project Data Panel](#)
- [Workspaces](#)
- [Running Tools](#)
- [Project Repository](#)
- [Ghidra Server Connection Status](#)
- [Data Tree](#)
- [Data Table](#)
- [File Icons](#)
- [Console](#)
- [Configure Project Window](#)
- [Edit Menu Options](#)
- [Manage Tools](#)
- [Getting Help](#)
- [Exit Ghidra](#)

### Tool Chest

The Tool Chest shows the tools that you currently have in your <user home dir>/ghidra/ghidra-<version>/tools folder. The tools are placed there when you initially install Ghidra. These tools are always available to your currently open project. See [Ghidra Tool Administration](#) for information on managing tools. The Tool Chest panel is a tool bar with an icon that represents each tool in your Tool Chest.

You can launch a tool by clicking on the icon. You can launch a tool with a Program by dragging a Program file from the [Project data tree](#) and dropping it on the tool icon in the Tool Chest.

## Active Project

The Active Project view shows your programs and datatype archives in a tree view or a table view. The tree view is useful for organizing your files into folders and sub-folders. The table view is useful for sorting all your files on some particular attribute such as size, processor, or modification date. In either view, you open and perform various actions on program files or datatype archives.

### Project Data Tree



The data tree shows all files in the project organized into folders and sub-folders. [Icons for files](#) indicate whether they are under [version control](#) and whether you have the file [checked out](#). Open this view by activating the "Tree View" tab.

### Tree Only Actions

The data tree supports the following operations:

#### *Create New Folder*

To create a new folder,

1. Select a folder that you own.
2. Right mouse click and choose the *New Folder* option.
3. A new node is created in the tree; a cell editor is displayed, containing the default name, New Folder; enter a new name, or the <Escape> key to cancel the editing.



You cannot create a sub-folder of a folder that you do not own.

#### *Copy Folders and Files*

To copy folders and files to another folder that you own,

1. Select a file or folder; you may also select multiple folders and files.
2. Right mouse click and choose the *Copy* option.
3. Select a destination folder.
4. Right mouse click and choose the *Paste* option. For very large files, an "in progress" dialog is displayed. You may cancel the paste operation at any time.

#### *Move Folders and Files*

To move folders and files to another folder that you own,

1. Select a file or folder; you may also select multiple folders and files.
2. Right mouse click and choose the *Cut* option; the icon will change to a dithered image to indicate the cut operation.
3. Select a destination folder.

4. Right mouse click and choose the *Paste* option.



You cannot move a file that is in use.

#### **Drag/Drop for Copy**

You can get the same effect of Copy/Paste using Drag and Drop.

1. Select a folder or file (or multiple folders and files).
2. Hold the Ctrl key down and drag the object to another folder.
3. Drop the object on a folder.



You will not get a valid drop target for folders that you do not own.



If you release the Ctrl key during the drag, the operation changes to a *move* if you are dragging from a folder that you own. Dragging files from another user always results in a *copy*, regardless of whether you hold down the Ctrl key.

#### **Drag/Drop for Move**

You can get the same effect of Cut/Paste using Drag and Drop.

1. Select a folder or file and drag it to a folder that you own.
2. Release the mouse button when you get a valid drop target.



If a folder or file already exists in the destination folder, Ghidra will append a ".copy" to the name to make it unique.

#### **Expand/Collapse**

To expand a folder and all of its descendant folders,

1. Select a folder.
2. Right mouse click and choose the **Expand All** option.

To collapse a folder and all of its descendant folders,

1. Select a folder.
2. Right mouse click and choose the **Collapse All** option.

#### **Project Data Table**

Active Project: git\_DevTestProject

Name	Path	Modified	Proces...	Endian	Ad...	Compi...	Size	Format
OldWinHelloCp...	/	Jun 27, 2018 1...	x86	Little	32	windo...	116411	Portable...
WinHelloCpp.exe	/	Jun 27, 2018 1...	x86	Little	32	windo...	116411	Portable...

Filter:

[Tree View](#) [Table View](#)

The data table shows all files in the project in a table sorted by some attribute of the file. In the example above, the files are sorted on file type. [Icons for files](#) indicate whether they are under [version control](#) and whether you have the file [checked out](#). To open this view, active the "Table View" tab.

#### **Actions for Both the Data Tree and the Data Table**

**Delete**

Deleting folders is a recursive operation, so all descendant folders and files are also deleted. This is a **permanent** operation.  
To delete a folder or file,

1. Select the folder or file or select multiple folders and files.



If you select a folder, then any selections that are descendants of this folder are ignored.

2. Right mouse click and choose the **Delete** option.



The Delete option is disabled for a file that is in use.

**Rename**

To rename a folder or file,

1. Select the folder or file.
2. Right mouse click and choose the **Rename** option.
3. A cell editor is displayed; enter the new name.

Duplicate names are not allowed within the same folder.



You cannot rename a file that is in use.



You cannot rename your project folder.

**Select All**

To select a folder and all of its descendants,

1. Select a folder.
2. Right mouse click and choose the **Select All** option.

**Read-Only**

To mark a file as read only,

1. Select a file.
2. Right mouse click and choose the **Read-Only** option.

The icon for the file is updated to indicate the read only state. When you right mouse click, a check mark shows up in the Read-Only option.



A read-only Program must be saved to a new name if you make changes to it.



You cannot change the read-only state of a file while it is in use.

**Drag File to a Tool**

- To launch a tool with a specific file, drag a file to the tool icon in the Tool Chest.
- To open a file in a *running tool*, drag a file to the tool icon in the [Running Tools](#) tool bar, OR drag the file to the tool window.

**Open a File in the Default Tool**

- To open a file in the tool that was [specified as the "default."](#) double click on the Program that you want to open, OR right mouse click on the file and choose **Open in Default Tool**.

**Open a File With a Specific Tool**

- To launch a tool with a specific file,

1. Select the file.
2. Right mouse click and choose **Open With ➔ <tool name>**.

**Refresh**

The  button on the tool bar refreshes the list of files in the selected folders. This is a way to sync the project folder/file structure with the project repository. The list of files and folders in the Project Data Tree is updated. This button is enabled only for selected folders. You can also refresh folders from a [viewed project](#) or [viewed repository](#).

### About

To [view information about a file](#), right mouse click on the file in the data tree and choose the **About...** option.

### Version Control Actions

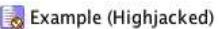
There are numerous actions related to version control. See [Project Repository](#) for details.

### File Icons

The Project Data Tree shows icons for the following types of files:

-  – A [program](#)
-  – A [project data type archive](#) (a data type file stored in the project)

The Project Data Tree shows modifications to these icons for files in the following states:

File Status	Sample Icon in Project Data Tree	Description
<a href="#">Versioned File</a> not checked out.		The program named "Example" is versioned as indicated by the light purple background. It is not checked out , since there is no circle with a check mark. Version 1 is the latest version, as indicated by "(1)"; the version will be the latest version when the file is not checked out.
Versioned File is <a href="#">Checked out</a> exclusively by you.		Version 1 of the program named "Example" is checked out, as indicated by "(1 of 1)"; Version 1 is the latest version. The blue check mark icon indicates that the file is checked out with an <a href="#">exclusive</a> lock. If your project is not associated with a Ghidra Server, you will always have the latest version checked out and the check out will always be exclusive, since the project is not shared.
Versioned File is Checked Out; the project is associated with a Ghidra Server		Version 3 of the program named "Example" is checked out; Version 3 is the latest version on the server, as indicated by "(3 of 3)" and the green circle with a check mark. The asterisk indicates you have changes to the file which have not been checked in yet.
Versioned File is Checked Out; the project is associated with a Ghidra Server. A newer version exists on the server.		Version 2 of the program named "Example" is checked out; a Version 3 has been created since Version 2 was checked out, as indicated by "(2 of 3)" and the magenta circle with a check mark. The asterisk indicates you have changes to the file which have not been checked in yet.
Private File		A program named "Example" is not under version control, exists only on your local machine, and is not visible to other users.
Hijacked File		The private file "Example" exists on your computer, but another user added "Example" to version control, which caused the private file to appear as <i>hijacked</i> , (i.e., the file can be saved "as is" using <a href="#">Save As</a> since you do not have the the file checked out that is on the Ghidra Server.) Hijacked files may also result from a checkout that was <a href="#">terminated</a> . The <i>shared</i>

version of "Example" will not be visible in your project until you [undo the hijack](#). You can also either rename the hijacked "Example", move it to another folder, delete it, or use the [Undo Hijack action](#). Then the shared "Example" will appear in your data tree as a versioned file.

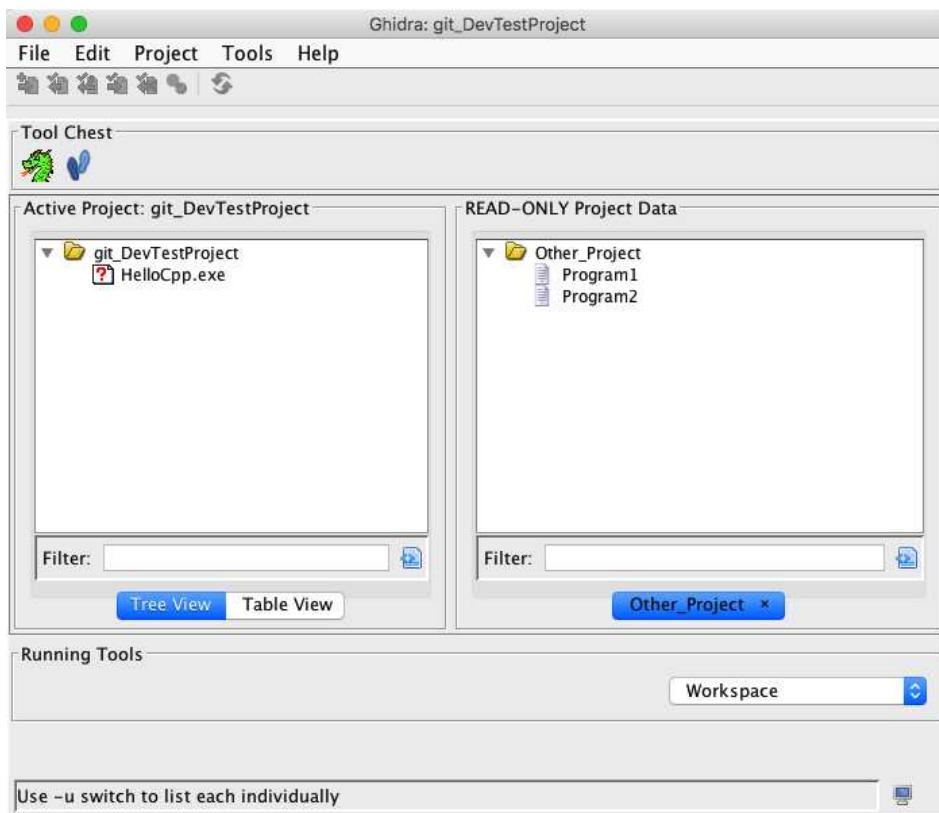
## Read-Only Project Data

You can view data from other Projects or remote Repositories and copy data into your current Project's data folders.

### [View Other Projects](#)

To view the data from another project:

1. Select **Project** → **View Project...**
2. A file chooser is displayed; the default location is the projects folder in the installation folder.
3. Choose a project; the file extension is ".gpr."
4. A new tab for the data tree is created in the "READ-ONLY Project Data" panel in the Project Window, next to the Active Project panel; the tab shows the name of the project.
5. The list of [recent projects menu](#) is updated to include this project.



You can copy and paste folders (via menus or drag and drop) and files from the other view to your folders.



You do not have to hold the Ctrl key down when you drag from the other view since this cannot be a move operation, as this view is always read-only.

### [View a Shared Project](#)

You view a shared project the same way you would a non-shared project; the difference is that when you view the shared project, an attempt is made to connect to the Ghidra Server associated with that project. Depending on the user authentication mode of the Ghidra Server for the other shared project, you may have to enter a password. If the connection

to the Ghidra Server is unsuccessful, then the only files available to you are your [private files](#).

### [View a Repository](#)

To view the data from a server-based repository:

1. Select **Project** ➔ **View Repository...**
2. A repository chooser is displayed; allowing you to specify a Ghidra Server network address and select one of its repositories...
3. Enter the Ghidra Server address and port. The default port is 13100.
4. Click the Refresh button to the right of the host name and port fields. This will connect to the specified Ghidra Server and list available repositories for which you have been granted access. You may be prompted for a password should user authentication be needed.
5. Select the desired repository from the list of those available.
6. Click the **Select Repository** button. A new tab for the data tree is created in the "READ-ONLY Project Data" panel in the Project Window, next to the Active Project panel; the tab shows the URL of the remote repository.
7. The list of [recent\\_projects/repositories menu](#) is updated to include this repository.

### [View Recent](#)

Ghidra maintains a list of Projects and remote Repositories that were recently viewed.

To view a recently opened project or repository,

1. Select **Project** ➔ **View Recent** ➔ <*project path or repository URL*>
2. Select a project or repository from the menu.
  - If the project/repository is not in the view, a new tab is created in the "READ-ONLY Project Data" panel in the Project Window; the tab shows the name of the project or repository URL.
  - If the tab is in the view, then the tab for this project/repository is selected.

### [Close View](#)

To close a view, select **Project** ➔ **Close View** ➔ <*project path/repository URL*>, OR click on the small 'X' on the specific view tab, OR right mouse click on the corresponding view tab and choose the **Close** option.

The tab is removed from the "READ-ONLY Project Data" panel in the Project Window.

### [Close All Read-Only Views](#)

To close all read-only views at once, select **Project** ➔ **Close View** ➔ **Close All Read-Only Views**.

The tabbed pane for read-only Project data is removed from the Project Window.

## Workspaces

A workspace contains a set of [running tools](#), and the tools' opened data. A workspace is analogous to a virtual desktop. When you switch to another workspace, you switch to a different set of running tools. The tools from the other workspace remain running, but are not visible until you switch back to that workspace.

The workspace names are listed in a combo box in the Running Tools panel. Switch to another workspace by choosing a name from the list. The default workspace, named "Workspace," is created in the project.

The workspace state, i.e., [running tools](#), [tool connections](#), [tool configuration](#), etc., is maintained when you [exit Ghidra](#) or [close the Project](#).

- To create a new workspace,

1. Select **Project** ➔ **Workspace** ➔ **Add...**
2. A dialog is displayed; enter a new workspace name. Duplicate workspace names are not allowed.
3. Click on the **OK** button; the newly created workspace becomes the current workspace; the name is added to the list of workspaces in the combo box.

 If you leave "Workspace" as the new workspace name in the dialog and click on **OK**, a one-up number is appended to the name to make it unique.

- To rename the current workspace,

1. Select **Project** ➔ **Workspace** ➔ **Rename...**
2. A dialog is displayed.
3. Enter the new name for the current workspace. Duplicate workspace names are not allowed. The list of workspace names is

updated to reflect the new name.

- To delete the current workspace,

1. Select **Project** → **Workspace** → **Delete...**
2. A dialog is displayed to confirm your delete request.
3. Choose the **Delete** button to delete the workspace.
  - Tools in the workspace are closed.
  - If you made changes to a file that is not open in any other tool, a dialog will be displayed to prompt you to save your changes.
  - The oldest workspace becomes the current workspace. If the deleted workspace was the last one, then the default workspace ("Workspace") becomes the current workspace.

- To switch workspaces,

1. Select **Project** → **Workspace** → **Switch...**
2. Switches sequentially through the list of workspaces (in creation order), wrapping back to the first after the last has been reached.
  - For example: WS1, WS2, WS3, then WS1.
3. If only 1 workspace exists, this action will do nothing.

### Running Tools

The Running Tools panel shows an icon for each tool that is running in the current workspace. Click on the icon to bring that tool forward on your console. To close the tool from the Running Tools panel, right mouse click on the icon for the tool and choose the **Close** option.

To [connect running tools](#), drag one icon onto another icon. Those tools are connected for all [tool events](#).



Tools running in different workspaces may be connected.

### Ghidra Server Connection Status

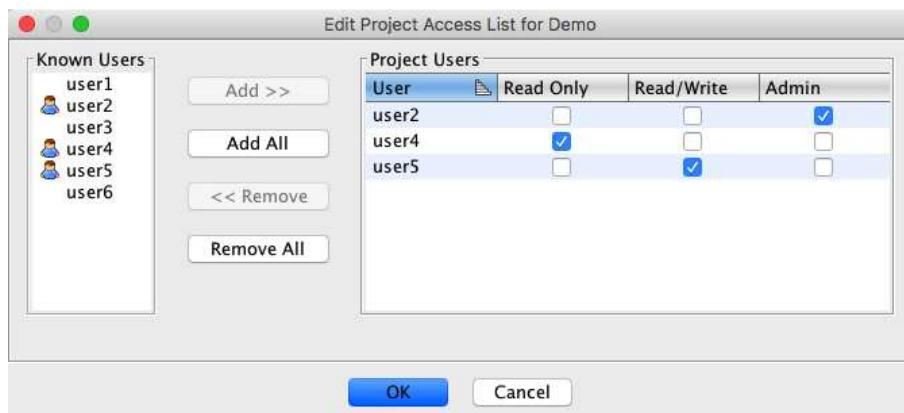
If your project is associated with a Ghidra Server, then below the *Running Tools* panel you will see a connection status panel that shows the name of the [Project Repository](#), your access privileges, and an indication of whether you are currently connected to the Ghidra Server. The [status button](#), indicates that your project repository is connected to the Ghidra Server; the status button, indicates that your project repository is associated with a Ghidra Server but it is not connected to it.



If your project is not associated with a Ghidra Server, then this status panel is empty.

### Edit Project Access List

If your project is [shared](#), the Project menu has an option to edit the [project access list](#). This list controls what users have access to the project and what [privileges](#) the users have. If you have administrative privilege in the project, the option for **Project** → **Edit Project Access List** will be enabled. The dialog displayed when you select this option shows a panel that is the same as the one you see in the [New Project Wizard](#) when you set up the user list for new project. As in the New Project Wizard, this dialog allows you to add and remove users, and change users' privileges in the project.

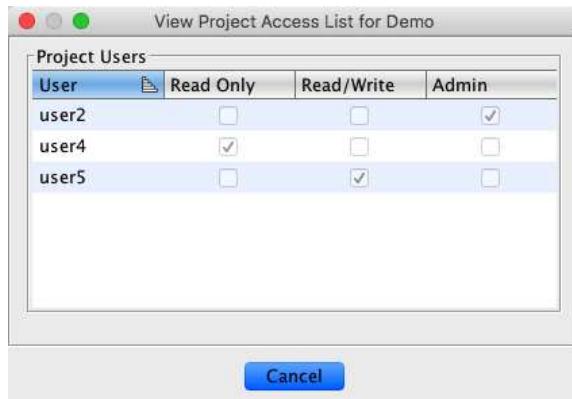


In order for a user to show up in the *Known Users* list, the server administrator must **add** a new user to the Ghidra Server. This is

accomplished from a command shell on the server system using the `svrAdmin` command. Refer to the `server/svrREADME.html` file in the installation directory for use of this administration command.



If the user does not have administrative privilege in the project, the user will not be able to view this full dialog and make edits. Instead, the option for **Project → View Project Access List** will be enabled, which will display the following dialog and allow the user to view the project users and their current access privileges only.



## Change Password

If your project is associated with a Ghidra Server that is using Ghidra password authentication, then the menu item, **Project → Change Password...** will be present. Use this option when you want to change your password. A dialog is displayed to confirm your request, as shown below.



If you select **Continue**, a dialog is displayed for you to enter your new password, and to re-enter your password.



When you initially connect to the Ghidra Server using password authentication, your default password is "`changeme`". The default password expires after 24 hours so you must change your password as soon as possible. If your password expires or if a user forgets their password, the Ghidra Server administrator must **reset** your password. This is accomplished from a command shell on the server system using the `svrAdmin` command. Refer to the `server/svrREADME.html` file in the installation directory for use of this administration command.

## Console

Click on the console icon to display the system console.

Log messages, including the standard output and error streams, are redirected to the console. If you are running Ghidra in development mode (i.e., through Eclipse or some other IDE), you will see standard output and errors in your IDE's console as well as the Ghidra console.

Errors and other informational messages are logged to a file in <user home>/`.ghidra/ghidraUser.log`. Messages are appended to the file

every time you launch Ghidra. Once the log file has reached 500KB in size, however, it will be rolled to a backup file named `ghidraUser.log.0`. Older backup files are similarly rolled to another file with a one-up digit suffix as well. Ghidra stores a maximum of three backup files (`ghidraUser.log.0`, `ghidraUser.log.1`, `ghidraUser.log.2`) at a time. The log files can be used by Ghidra developers for troubleshooting.

The field next to the icon in the Ghidra Project Window shows the last message sent to the console (error messages are in red).

## Getting Help

### Context Sensitive Help

- Ghidra provides context sensitive help that pops up when you hit the <F1> or <Help> key.

To get help on a menu option,

1. Display the menu (either from the tool menu or the popup) that has the option you want help on.
2. Position the mouse pointer over the option.
3. Press the <F1> or <Help> key.
4. The Help Viewer is displayed and shows the appropriate help contents.

- To get help on a dialog or tool window, click somewhere in that window and press the <F1> or <Help> key.

If no specific help exists, then a [default page for Ghidra help](#) is displayed.

### About Ghidra

The [About Ghidra](#) option shows build information about the Ghidra application.

Related Topics:

- [Tool Administration](#)
- [Projects](#)
- [Program](#)
- [Import Program](#)
- [Export Program](#)
- [Welcome to Ghidra Help](#)
- [Create a Shared Project](#)
- [Project Repository](#)
- [Project Info dialog](#)

## GhidraServer

# *Ghidra Server*

The Ghidra Server provides file access to multiple users enabling a team to collaborate on a single effort. It provides network storage for shared project repositories while controlling user access.

## Running the Server

The Ghidra Server capability is incorporated into the standard distribution of Ghidra. Please refer to the *svrREADME.html* file located within the Ghidra installation directory (*<your installationdirectory>/server/svrREADME.html*). This file will provide information regarding the installation, configuration and administration of the Ghidra Server.

Related Topics:

- [Project Repository](#)
- [Creating a Shared Project](#)
- [Project Access List](#)

# About Ghidra

The *About Ghidra* dialog displays general application information regarding Ghidra.

## To view information about Ghidra

- From the menu-bar in tool, click **Help → About Ghidra**



This option is also available from the [\*Ghidra Project Window\*](#)

## Information displayed in the dialog

- Version
  - For example, "8.1"
  - If "Dev" is included in version name, this version is a pre-release build
- Java Version
  - For example, "11.0.1"
  - Version of Java being used to run Ghidra.
- Build date
  - For example, "2019-Jan-18"

## Installed Processors

The *Installed Processors* dialog displays all processors supported by the various language modules currently installed. This list is

quite brief and does not convey the assorted variants which may be implemented for each processor.



This option is also available from the [Ghidra Project Window](#)

## To view the Installed Processors list

- From the menu-bar in tool, click Help → Installed Processors...

## User Agreement

The *User Agreement* dialog displays the conditions that you must agree to in order to use the Ghidra tool. You agreed to these terms when you first ran Ghidra.

## Related Topics

- [About Program File](#)

# Ghidra Projects

A Ghidra project maintains information for a particular reverse engineering (RE) effort. The RE effort involves using tools built with [plugins](#) to analyze target software. Tools are accessible across projects and not associated with any one project in particular. However, the target software is imported into a Ghidra project as [programs](#). The Ghidra project organizes and maintains these programs for multiple users.

The latest release of Ghidra introduces the concept of a [project repository](#) that contains files that are [versioned](#), shared, or private. A shared project is associated with a Ghidra Server that manages the files in the repository. Multiple users in the project can add files, access files on the server and create new versions of the files. Private files are local to your project and have not been added to a project repository. You can still take advantage of the project repository features and not have your project be associated with a Ghidra Server.

You can create subfolders and [import](#) programs into your project. You can also view other data from other projects, or drag the other project data into your current project.

The [Ghidra Project Window](#) shows the active project. The Ghidra Project Window's menu provides access to the following project operations:

- [Create a project](#)
- [Open an existing project](#)
- [Close the project](#)
- [Re-open a project](#)
- [View a recent project](#)
- [Save the project](#)
- [Archive the project](#)
- [Restore an archived project](#)

Related Topics:

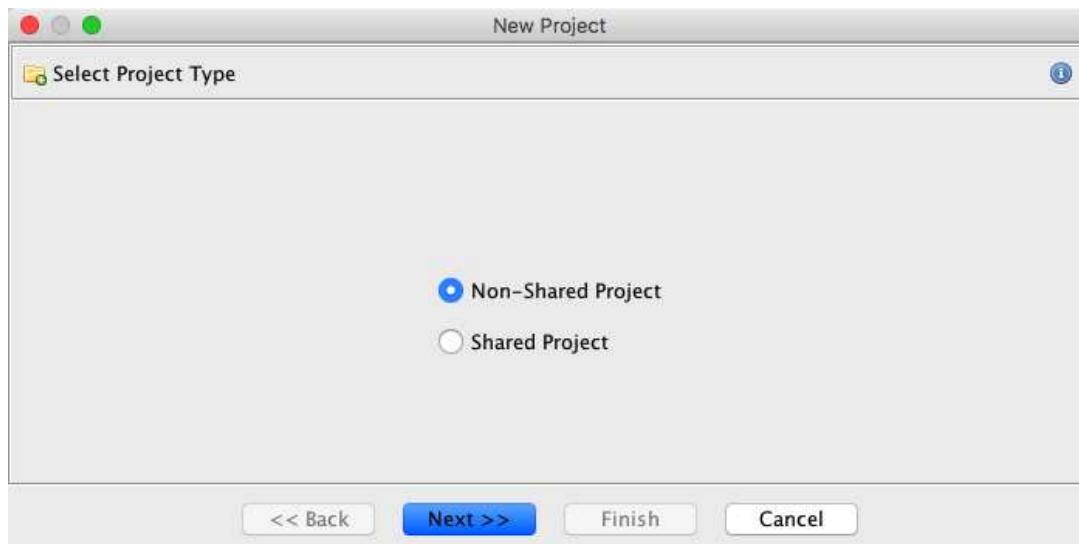
- [Ghidra Programs](#)
- [Ghidra Project Window](#)
- [Import Programs](#)
- [Project Repository](#)

# Creating a Project

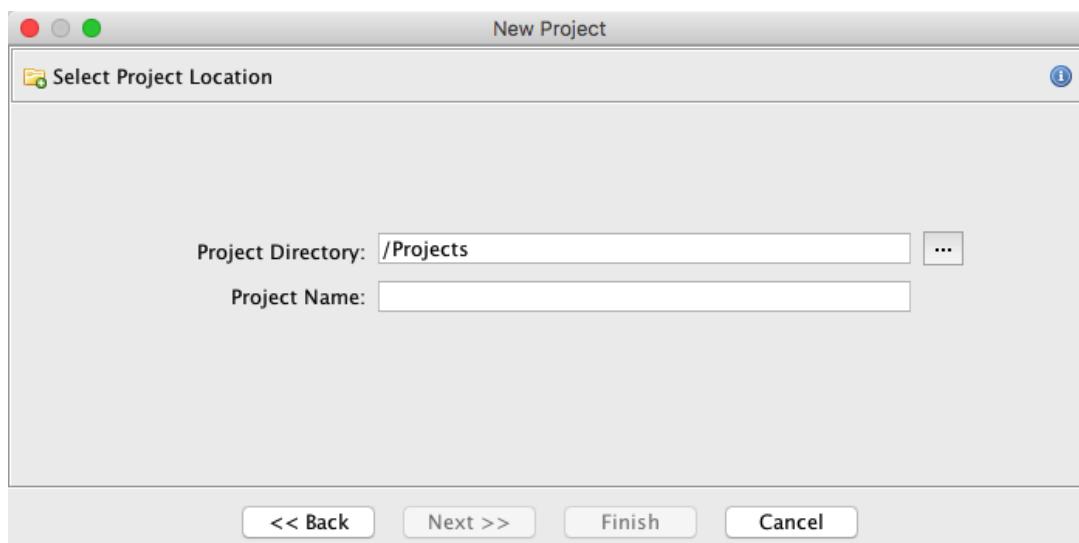
The **New Project** option creates a new project. When you create a new project, Ghidra will close your current project, create the new project, and update the Ghidra Project Window with the newly created project. The *New Project* wizard takes you through the steps to create either a non-shared project or a shared project. The [shared project](#) can be *shared* with others, meaning that the project will be associated with a repository on a server that other users can access.

To create a new non-shared project:

1. Select the **File → New Project...** menu option from the Ghidra Project Window.
2. The first panel of the *New Project* wizard is displayed, where you choose the project type.



3. Leave the *Non-Shared Project* radio button selected to create a project that will not be shared with others.
4. Activate the **Next >>** button.



3. Click on the browse button (...) to display a file chooser; select the directory of where you want your new project to reside. The *Project Directory* field defaults to the last directory that you specified to create a new project.
4. Enter the name of your new project; the **Finish** button is enabled.
5. Activate the **Finish** button. If you have a project opened, Ghidra closes the project and opens the new project.
6. If there is an active project and it has been modified, Ghidra will prompt to save the active project before closing it. From the *Save Modified Files* dialog, click the **OK** button to save changes to the project that is being closed as the new one is opened.

## Creating a Shared Project

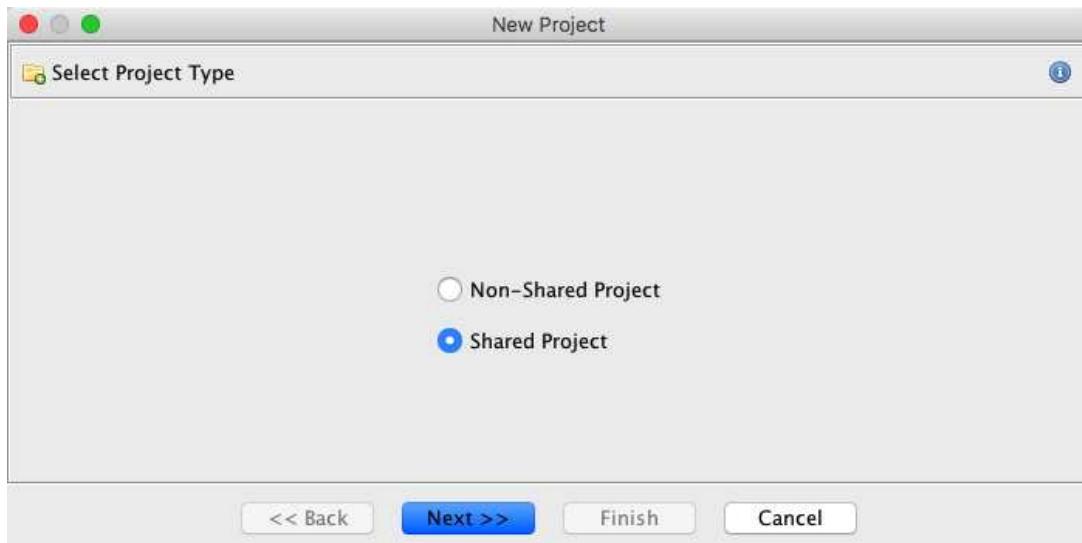


The discussion for this section assumes the following:

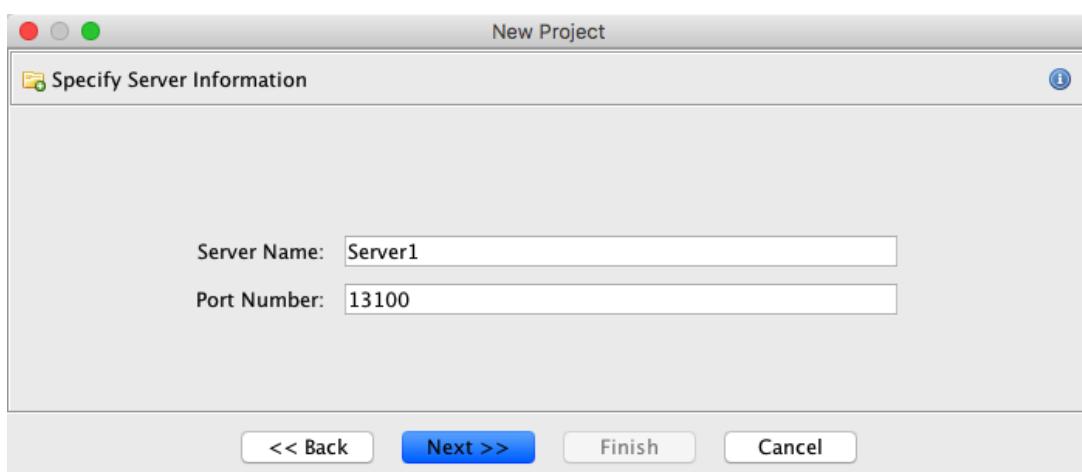
- You have a Ghidra Server that is set up to manage the project repository that you are going to access.
- You have been added to the server's list of known users.
- If you are associating your project with an *existing* project repository, the [Administrator](#) of the project has added you as a user to the project repository.
- You know the name and port number of the Ghidra Server.
- If the Ghidra Server is using PKI Authentication, you have a PKI Certificate.

To create a shared project,

1. Select the **File → New Project...** menu option from the Ghidra Project Window; the first panel of the *New Project* wizard is displayed.
2. Select the *Shared Project* radio button on the *Select Project Type* panel.



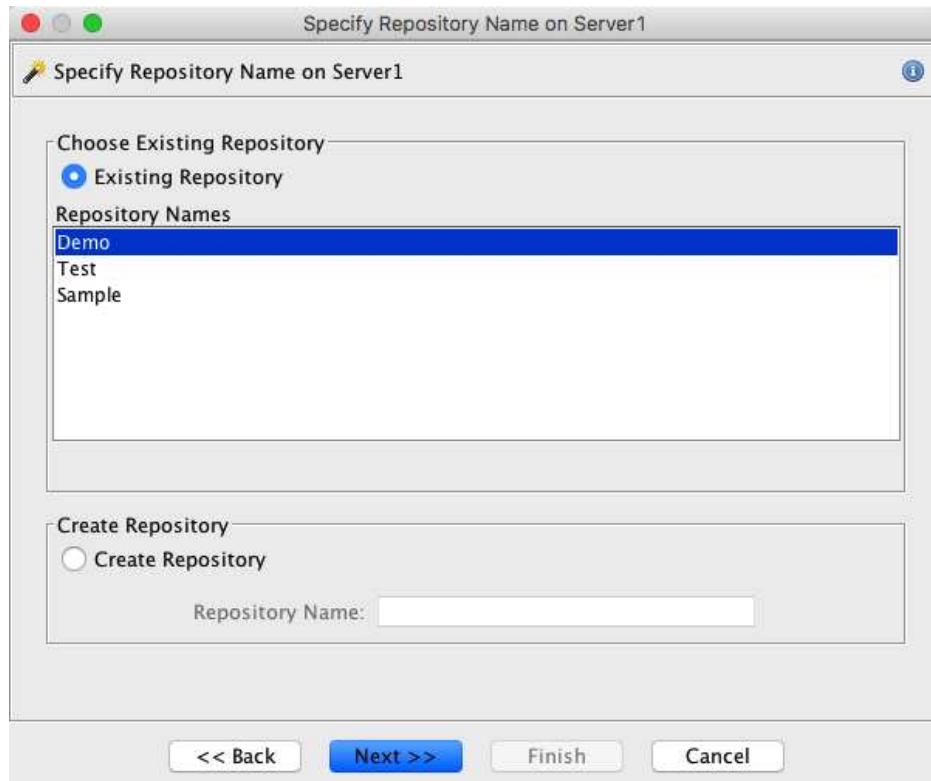
3. Activate the **Next >>** button.



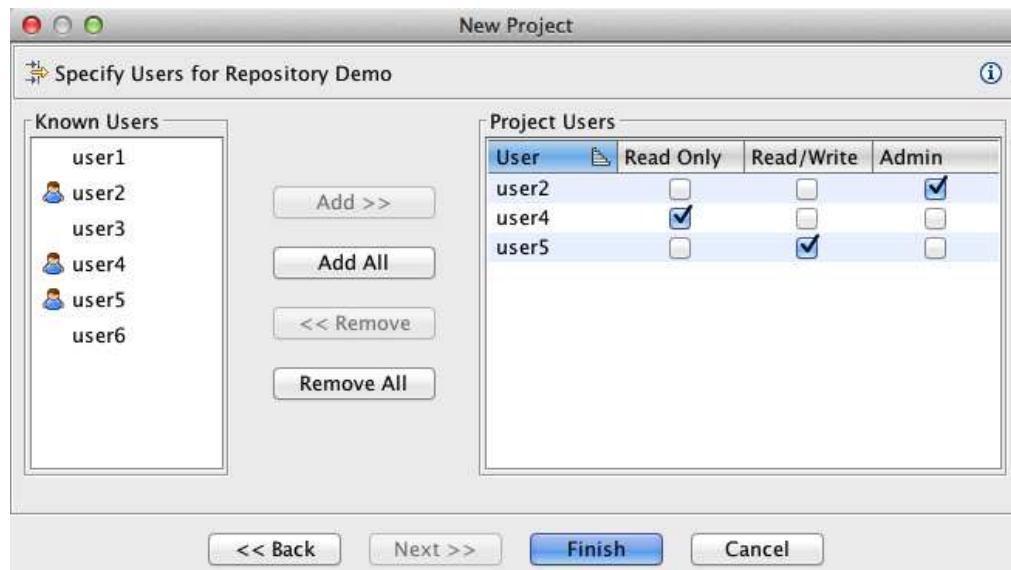
4. Enter the name of the server or the IP address of the server of where the repository resides. (If you had specified server information for accessing another shared project, then the server and port number from that project are used as default values.)
5. Enter the port number. The default port number is 13100.
6. Activate the **Next >>** button.



Refer to the [Troubleshooting](#) page if you fail to connect to the Ghidra Server.



7. In this example, three repositories are listed in the *RepositoryNames* list which reflect all the repositories the user has access to on the currently connected server. This list may differ based upon your login credentials. By default, the *Existing Repository* radio button is selected because of the existing repositories found on the server. Only those repositories you have been granted access to will be included in this list.
  - To use an existing repository, leave the radio button selected, and select a repository name from the list of *RepositoryNames*.
  - To create a new repository, select the *Create Repository* radio button; the *RepositoryName* field becomes enabled. Enter the name of the new repository. This option will be disabled if you have logged into the Ghidra Server in anonymous mode.
8. Activate the **Next >>** button. If you selected the *Create Repository* radio button, then the following sample *Project Access* panel to specify user access is displayed. If you are creating a new shared repository, you are by default the [Administrator](#) in the project. You are not allowed to change your own access. If the server has been configured with anonymous mode enabled, an additional checkbox control will appear allowing you to grant anonymous access to this new repository.



9. Users that are known to the server are listed in the *Known Users* area. Users that are part of the project are marked with the icon. To add users, select users from *Known Users*, select the **Add >>** button. Add all users by selecting the **Add All** button. To remove users, select users in the project, and select the **<< Remove** button. To remove all users (except yourself), select the **Remove All** button.

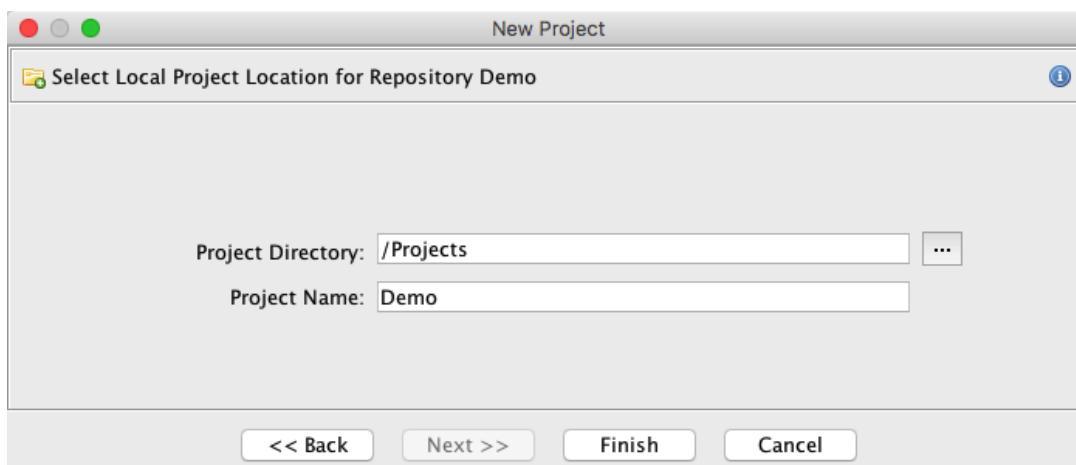
The *Project Users* table on the right side of the dialog indicates user privileges:

- **Read Only** – the user can open programs in read only mode, cannot [check out](#) files from the repository, and cannot [check in](#) files, and cannot [add files](#) to the repository. The read only user may create a local copy of the program; the project repository is not affected.
- **Read/Write** – the user can check out files from the repository and check in files into the repository, and add new files to the repository.
- **Admin** – the user has read/write privileges as well as the ability to add users to and remove users from the project; the Admin can assign administrative privileges to other users. The project must have at least one administrative user. The administrator also can [terminate a check out](#).



If you are creating a project against an *existing* repository, then you will not see the *Project Access* panel as part of the *New Project* wizard. After you have opened the shared project, *and* if you are an administrator in the existing repository, then you can display and modify user privileges by choosing the **Project → Edit Project Access List...** option.

10. Active the **Next >>** button; you must next specify where to create your project that will be associated with the project repository. Typically, you would specify a directory on your local file system.



11. Select the project directory. The *Project Directory* field defaults to the directory that you last specified for a new project. Click on the browse (...) button to bring up a file chooser.
12. The *Project Name* field defaults to the name of the repository that you selected in Step 7. The **Finish** button is enabled when the *Project Directory* and *Project Name* fields have valid entries.
13. Select the **Finish** button to complete the *New Project* process.

Before the new project is opened, if you made changes, you are asked whether to save the old project. If you choose to open the new project, the default tool [Code Browser](#) is in the Tool Chest and no files are associated with the project that you have just created. Your current [workspace](#) is named "Workspace."

Related Topics:

- [Ghidra Projects](#)
- [Close Project](#)
- [Save Project](#)
- [Open Project](#)
- [Archive Current Project](#)
- [Project Repository](#)
- [Edit Project Access List](#)

## Open Project

When you open a Project, your active project is [closed](#); the project identified in the *Open a Ghidra Project* dialog will be opened. If the active project has been modified since it was last saved, Ghidra will prompt you to save the active project before closing it. The project window is restored with any [plugins](#) it had when you last saved the project. Tools that were running when you last saved the project are launched; programs that the tools had open are opened. The last active [workspace](#) from this project now shows up as the active workspace. Other [viewed projects](#) are restored.

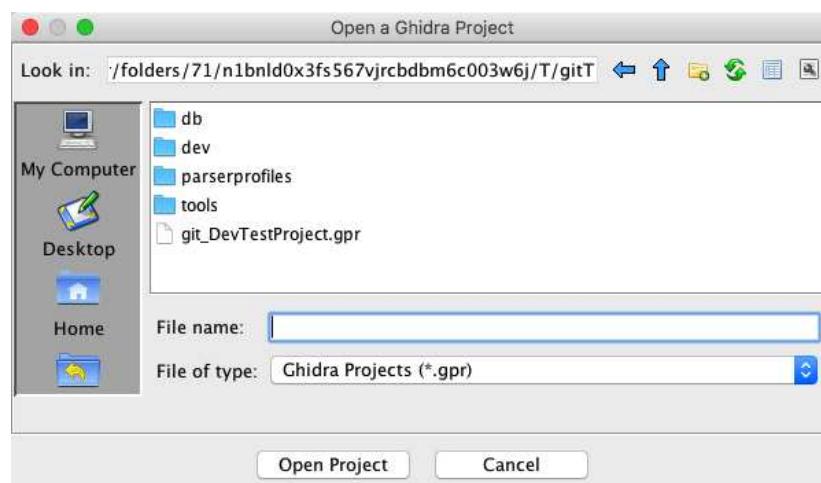


Projects that were created using a release of Ghidra prior to 3.0 can be viewed only. You can drag the data from the old project to your current project.

To open a project, choose it from a list of projects in the default project directory. If the project has been opened before, then the project will appear in the [Reopen](#) list. If the project is being shared by others, it may not reside in the default project directory. Use the browse button, on the *Open a Ghidra Project* dialog, to locate the target project.

To open a project:

- From the Ghidra Project Window, select **File ➔ Open Project....** The *Open a Ghidra Project* dialog appears.

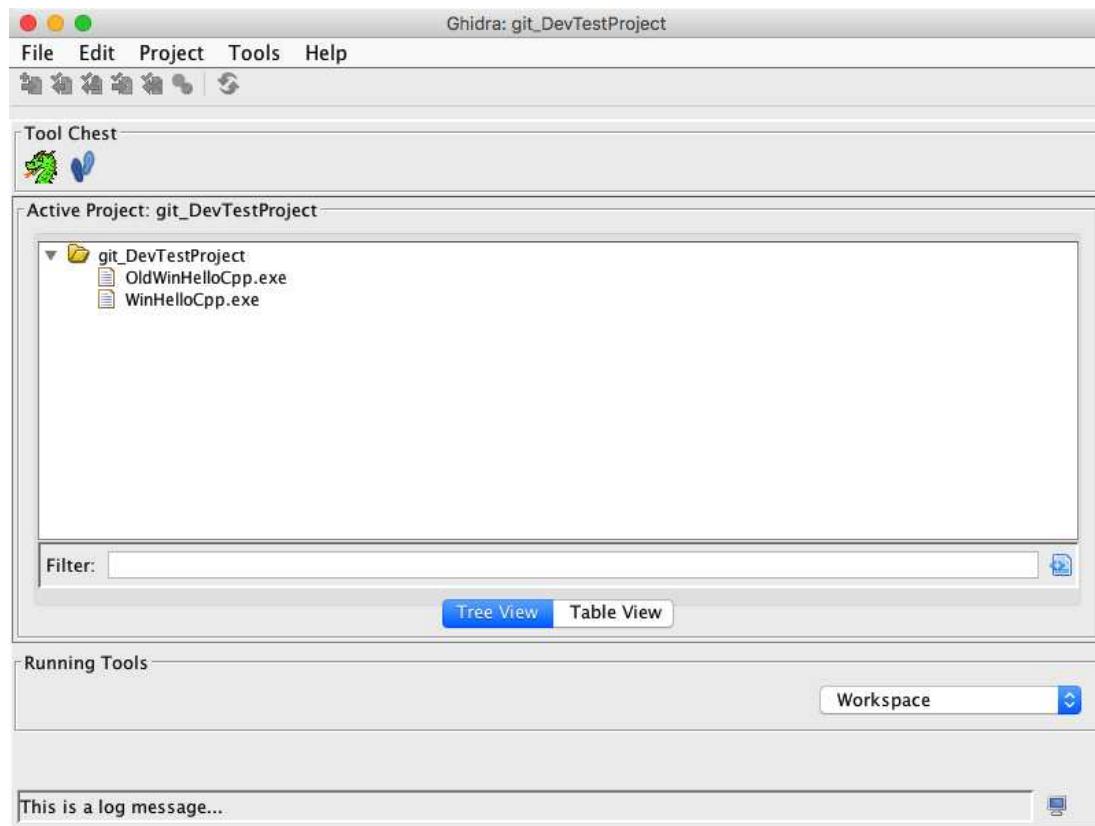


- The dialog box filter defaults to \*.gpr (project file extension). Select the project from the *Open a Ghidra Project* dialog.



The project name can be of any length. The name of the project has the same restrictions that the operating system imposes on file names. A Ghidra Project name must have the **.gpr** extension.

- Click the **Open Project** button. The selected project appears in the Ghidra Project Window. If you are opening a [shared project](#), Ghidra attempts to [connect to the Ghidra Server](#). You may have to enter a password, depending on the type of user authentication the Ghidra Server is using.



If this were a shared project, the connection status button (⌚) would be displayed, indicating that the project was successfully connected to the server. If the project failed to connect to the server, the status button would appear as (⌚). If the server comes up after you have opened the shared project, you can click on status button to attempt to connect to the server. You can still work offline in a shared project, however, you will not be able to do any check outs or check ins.

#### Related Topics:

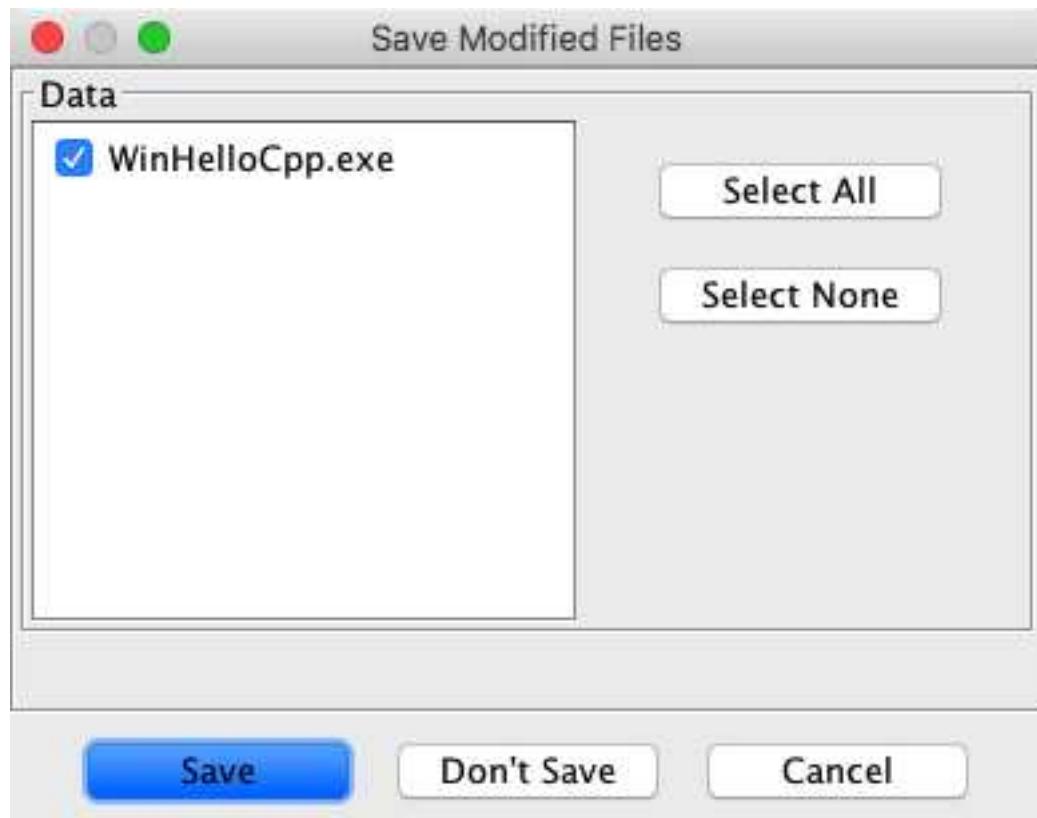
- [Ghidra Projects](#)
- [Close Project](#)
- [Save Project](#)
- [Reopen Project](#)
- [Archive Current Project](#)

# Close Project

Closing a project will return Ghidra to the NO ACTIVE PROJECT state. If you have changed the state of your project (e.g., launched another tool, changed your program, etc.), Ghidra will prompt you to save your project and any changes you have made to your program.

To close a project:

1. Select **File → Close Project** from the menu.
2. If there are unsaved changes, Ghidra will display the *Save Modified Files* dialog to allow you to save any changes. Changed items are selected by default. Deselect items that you do not want saved. Select the **OK** button close the project.



The **Deselect All** button deselects all changed items in your project. If you choose **OK**, your current project state is not

saved and any changes made to your program are lost.

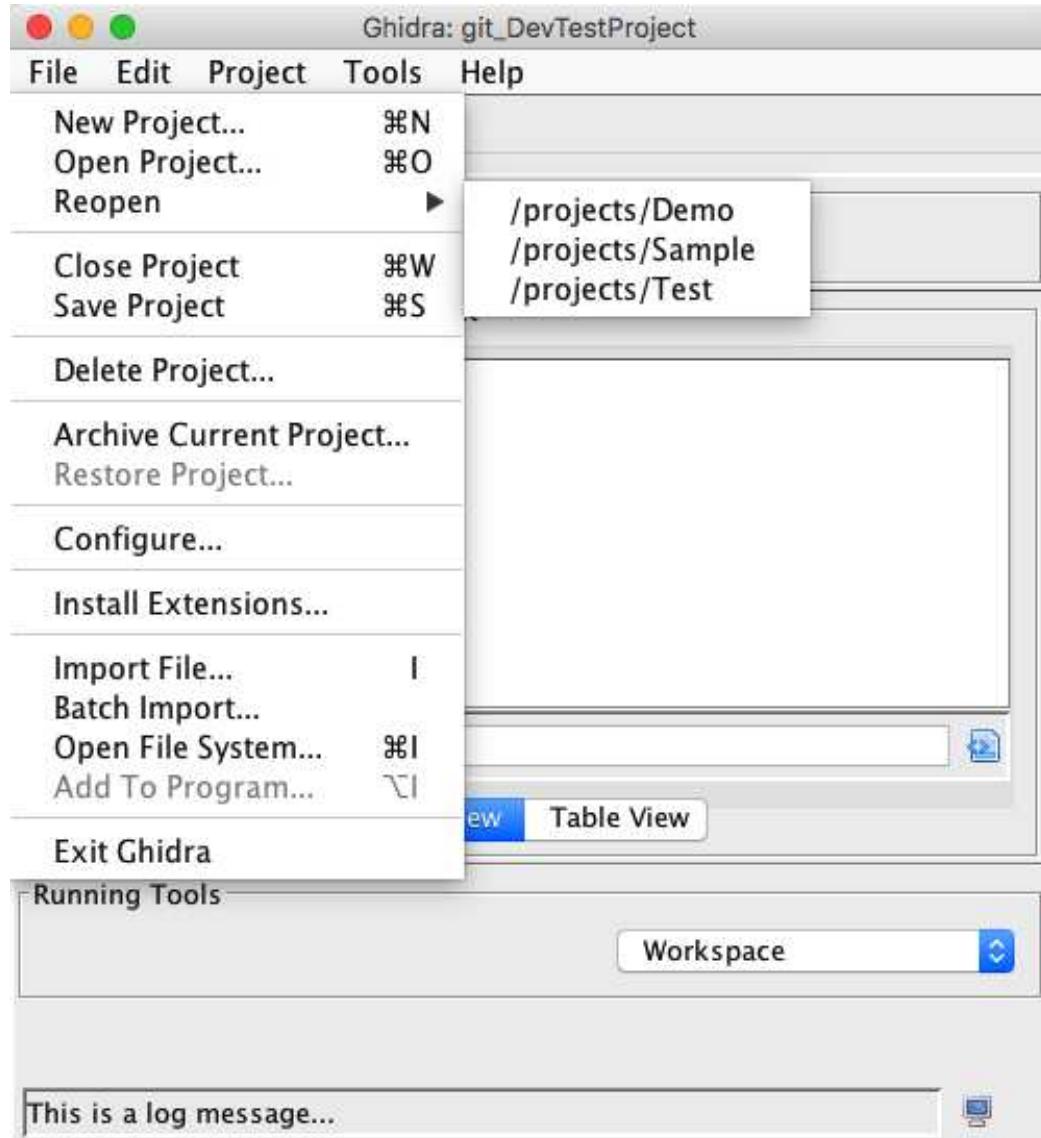
When the project is closed, the Ghidra Project Window indicates NO ACTIVE PROJECT.

**Related Topics:**

- [Ghidra Projects](#)
- [Ghidra Programs](#)
- [New Project](#)
- [Open Project](#)
- [Reopen Project](#)
- [Archive Current Project](#)

# Reopen Project

Ghidra maintains a list of the projects that you recently opened. These are available in the Ghidra Project Window's menu. Reopening a project will close any active project and open the project selected from the Reopen menu. The project that was most recently opened is placed first in the list.



To reopen a project:

- Select **File → Reopen → directorypath/project\_name** where *directory\_path/project\_name* indicates the project from the list that you wish to reopen.

Ghidra will close any active project. It then opens *project\_name* and restores all of the project's configurations.



If the project that you are re-opening is [shared](#), then an attempt is made to connect to the Ghidra Server. If the connection was not successful, you can still access your private files and checked out files. Other files on the server will be unavailable.

#### Related Topics:

- [Ghidra Projects](#)
- [Close Project](#)
- [Save Project](#)
- [Open Project](#)
- [Archive Current Project](#)
- [Project Repository](#)

# Save Project

Saving a Ghidra Project will commit all of the project-related changes that were made since the project was opened or since the project was last saved. Saving a project retains information about:

- The plugins that are being used by the Ghidra Project Window
- The plugins that are being used by each of the running tools
- Options associated with the plugins and/or the tool
- The running tools' sizes and positions on the screen
- Viewed projects
- List of recently opened projects
- The active [workspace](#)

To save the active Ghidra project:

- From the Ghidra Project Window menu, select **File → Save Project**.



You can also save your project as you [exit](#) Ghidra.

Related Topics:

- [Ghidra Projects](#)
- [Ghidra Project Window](#)
- [Code Browser Options](#)
- [Tool Options](#)
- [Exit Ghidra](#)

# Archive Current Project

Archive Current Project will save the entire contents of the currently open project to a file name that you specify.

The archive is a snapshot of the project data at the time of the archival. You must close all running tools before you can begin the archive process.

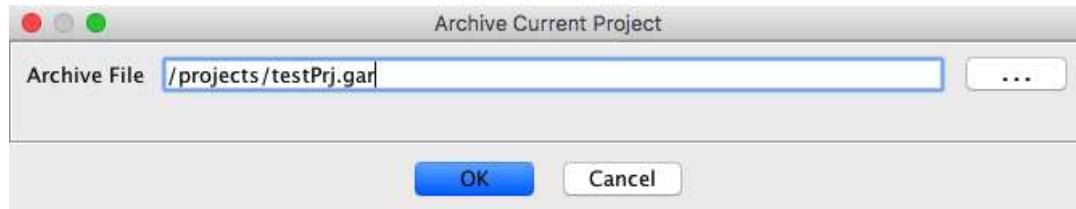
After the archive is complete, the [Ghidra Project Window](#)'s message area will indicate whether the archive succeeded or failed. The archive can be restored by using the [Restore Project](#) operation.

## Why Archive a Project?

- Archiving a project saves it to a file in a format that is compatible between different Ghidra release versions. It also saves off the project data in a way that it can easily be restored at a later date.
- Archiving the project does not remove the project from the Ghidra projects directory. So archiving the project has no impact on further use of the project.
- Archiving periodically to a uniquely named file can simply be used as a way to backup the current state of the project data. The archive file(s) can then be copied to another disk drive or computer system for redundancy.

To archive the current project:

1. Close any tools that are running. (You cannot archive a project that has running tools.)
2. From the [Ghidra Project Window](#), select **File** → **Archive Current Project...**
3. From the *Archive Current Project* dialog, specify the *Archive File* where the project is to be saved. The default location of the archive file is your projects directory.



The file name must end with a '.gar' extension.

4. Click the **OK** button to begin archiving.
5. If the specified archive already exists the *Archive File Exists* dialog is displayed.



Decide whether you want to overwrite the existing archive file. Select **Yes** if you want to overwrite the file. Otherwise, select **No** and enter another filename in the *Archive Current Project* dialog.

6. Your project is saved automatically before the archive process begins.
7. The 'In Progress' dialog is displayed indicating the project is being archived. You can cancel the archive process at any time by clicking on the **Cancel** button in the progress dialog.

## When Archiving Errors Occur

If the archive process encounters an error, archiving will terminate and a message will appear. Also, the failure

is indicated in the [Ghidra Project Window](#)'s message area.

Provided by: *Project Archiver Plugin*

Related Topics:

- [Restore Project](#)
- [Ghidra Projects](#)

# Restore Project

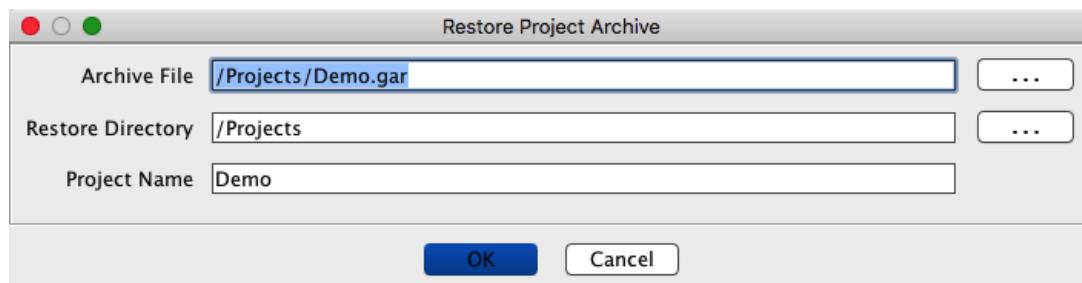
The Restore Project operation will create a new project from an Archived project file. Restoring a project makes it the active project.



You must close your project before you can restore an archived project.

To restore an archived project,

1. If a project is open, close it by selecting **File** ➔ **Close Project** from the [Ghidra Project Window](#) menu.
2. Select **File** ➔ **Restore Project...**



3. The Restore Project Archive dialog is displayed. Fill in the fields to indicate the project to restore and where to restore it.
4.
  - **Archive File:** Specify the full path for the archive file to be restored. Use the browse button ("...") to locate the archive (\*.gar) file.
  - The *RestoreDirectory* and *Project Name* fields are automatically filled in when you use the browse button ("...") to the right of the *Archive File* field to select the archive file.
  - **RestoreDirectory:** The project directory where the new project will be created.
  - **Project Name:** The name of the new project.
4. Press the **OK** button.
5. If the project is being restored to the same name and location as an existing project, the *Project Exists* dialog is displayed, as shown below.



- Specify a different *RestoreDirectory* or a *Project Name* that doesn't exist and try again.
- 6. The 'In Progress' dialog is displayed indicating the archive is restoring. When the restore is complete, this dialog will disappear and the newly restored project appears in the [Ghidra Project Window](#).
  - To cancel the restore operation click on the **Cancel** button. Any files that were created during the restore are removed as a result of the cancellation.

## Restoring a Version 2.x Project

If you restore a project from a version of Ghidra that was 2.x or before, *and* the project contained multiple users, the project is restored with you as the owner of all the files. You will see the folders and data files for the other users that were in the project, but you are the owner.

Provided by: *ArchivePlugin*

Related Topics:

- [Archive Project](#)
- [Ghidra Projects](#)

# Project Repository

Ghidra supports the concept of a *project repository* such that files in the repository can be *versioned*. Versioning allows you to track file changes over the life of the project. The repository supports check out, check in, version history, and viewing what is checked out. When you check in your file, a new version is created. The project repository can be used with or without a [Ghidra Server](#). If the project is associated with a Ghidra Server, the project is *shared*, meaning that the *files in the project* are accessible by multiple users concurrently. The [Project Access List](#) defines which users are allowed to access the shared repository. When you [create a new project](#), you can specify whether it should be associated with a Ghidra Server.

For projects that are not shared, all files and versions are managed locally in your project directory. For the shared project repository, files are maintained on the Ghidra Server. When you check out a file, a file is created locally in your project directory. When you check in a file, it is [merged](#) (if necessary) with the latest version to create a new version. Merging is necessary only for shared project repositories, as with a non-shared project, the version you checked out is always the latest version.

## Connect to the Server

When you open a shared project, Ghidra attempts to connect to the server that is associated with the shared project. Depending on what user authentication mode the server is using, you may have to enter a password. If the server is not running, you are still able to work with your checked out files while you are offline. Other versioned files not checked out are not accessible. When the server comes up, Ghidra will reconnect as necessary. You can also attempt to connect "manually" by selecting the connection status button  on the Ghidra Project Window or on the [Project Info](#) dialog. When the connection is successful, the connection status button changes to .

If you lose the connection to the server after having started Ghidra, shared files not checked out "disappear" from the Ghidra Project Window, as they are unavailable. Private files remain intact and are not affected by the server connection.



You are authenticated only once per Ghidra session; so if you open other project repositories managed by the same Ghidra Server, you will be prompted only once for a password, as required.

## Troubleshooting a Failed Connection

If you fail to connect to the Ghidra Server, check the following:

- Verify that the client machine can "ping" the server machine.
- Verify that you are attempting to login using the correct authentication mode.
- Verify that the Ghidra Server administrator has added you as a user on the server.
- If the server is using Ghidra Password authentication, you may need to have your password reset to "*changeme*" by the Ghidra Server administrator.
- Verify the host name and port of the Ghidra Server.
- Verify that the Ghidra Server is running: from a command window, type

```
telnet [host] [port]
```

If the server is not listening on that port, you will get a failed connection message from `telnet`.

## Version Control

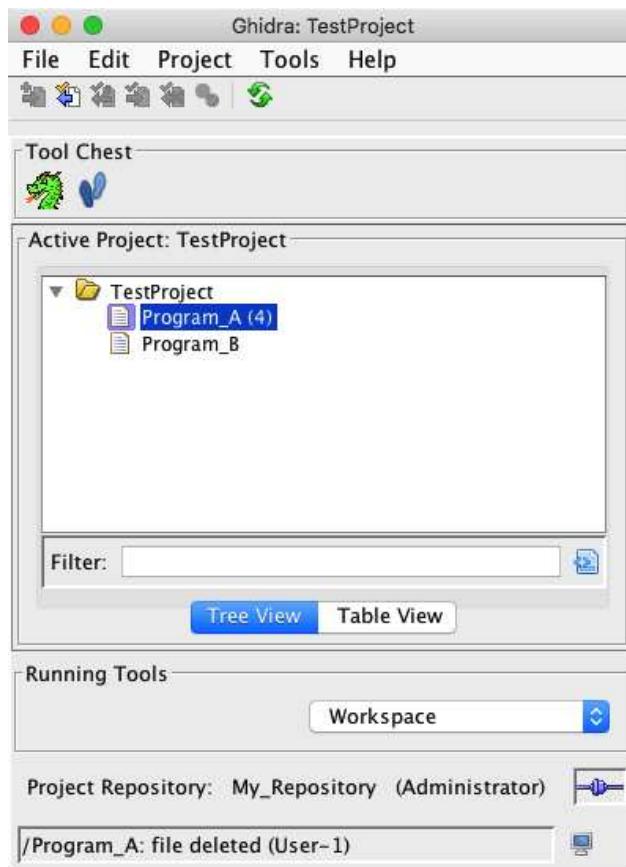
Except for [merging](#), accessing version control features is the same regardless of whether your project repository is shared.

[Add to Version Control](#) 

Add a file (or multiple files at once) to version control by selecting the file in the Ghidra Project Window. You can either click on the tool bar icon, , or right mouse click and choose the **Add to Version Control...** option. A dialog is displayed so that you can add comments about the file.



Leave the checkbox selected for *Keep File Checked Out* so that you do not have check out the file after you have added it to version control. This checkbox will be selected and disabled automatically if you have the file open. The **Apply to All** button allows you to associate the same comment for multiple files that you are adding to version control. After you add the file, the Ghidra Project Window indicates the file's check out state and version.



This image shows that the file "Program\_A" is associated with a shared project (note the blue border on the [file icon](#)), and you are now working with version 1 of 1. The file "Program\_B" has not been added to version control (note the plain icon and no version information). It is considered to be a "private" file. Private files are never visible to other users.

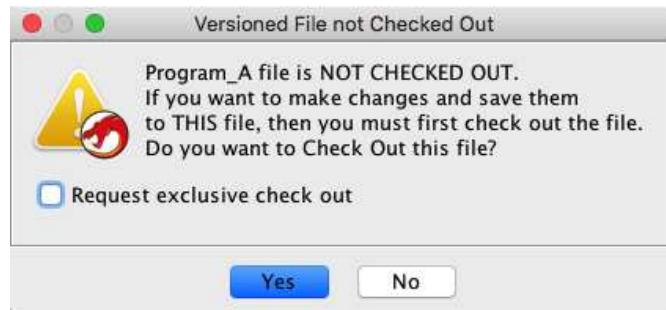
 A normal checkout is indicated by a checkmark with a green background , while an exclusive checkout is indicated by checkmark with a blue background . A checkmark with a red background  indicates that a newer version has been checked-in by another user.

A tool tip on the file (let the mouse pointer hover over it) shows the date the file was checked out, and

the date that it was last modified. An asterisk will appear on the file icon to indicate that changes have been made but not checked in.

### Check Out

To check out a file, select the file in the Ghidra Project Window. You can either click on the check out icon  on the tool bar, or right mouse click on the file and choose the **Check Out...** option.



If your project repository is shared, a dialog is displayed to allow you to request an **exclusive lock** on the file. An exclusive lock is necessary if you plan to [manipulate the memory map](#) in any way, e.g., move or delete memory blocks, [change the program's language](#), etc. An exclusive lock can be granted if no other user has the file checked out. While the exclusive lock exists, no other user can check out the file.



The exclusive lock is implied for a non-shared project repository.

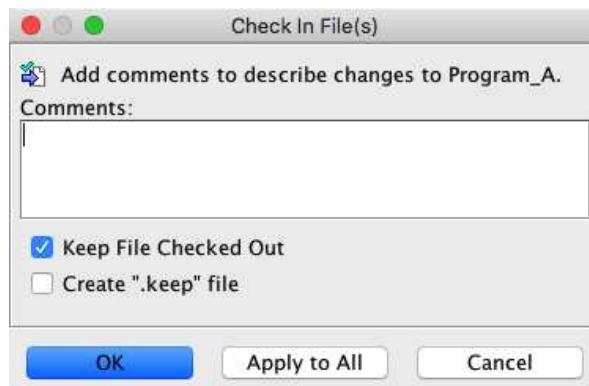
### Check In

After you have made your changes and saved them, you are ready to check in your file. (You cannot check in a file that was not changed.) The check in creates a new version for this file. To check in the file, select the file in the Ghidra Project Window. You can either click on the check in icon  on the tool bar, or right mouse click and select the **Check In...** option.



The  icon is also available from the tool where you have the file opened.

A dialog is displayed so that you can enter comments that describe your changes.



The checkbox for *Keep File Checked Out* is selected and disabled automatically if you still have the file open. If the file is closed and you plan to create more versions, leave the checkbox selected for *Keep File Checked Out*. The checkbox for *Create ".keep" file* is selected by default; this option causes a copy of the file that you are checking in to be created on your local file system.

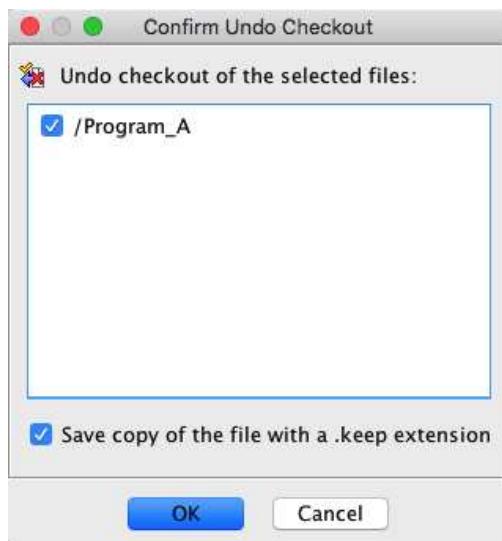
In a shared project repository, when you check in your file, the changes in your file may have to be [merged](#) into the latest version on the server. This will be the case if another user checks in a file since you did a checkout on the file. Under most conditions, the merge will be automatic without any intervention required on your part. However, if you made changes such that a conflict arises, you will have to [resolve the conflict](#) at the time of check in. When another user checks in his file, you will see [navigation markers](#) for changes made since you checked out your file. Potential conflicts are indicated in red. Refer to the [Merge](#) page for more information about merging.

### **Undo Checkout**

You may want to undo your checkout such that you lose all your changes, and your file reverts to the latest version on the server.

To undo a checkout:

1. Close the checked out file.
2. Select the checked out file in the Ghidra Project Window.
3. You can either click on the undo checkout icon  in the tool bar, or right mouse click on the selected file and choose the **Undo Checkout** option. If you had made changes to the file, a dialog is displayed confirm the undo check out.



If the checkbox on the dialog is selected, then a private file is created with a ".keep" extension on the filename. The checkbox is selected by default.

 If you have the file open in the Code Browser when you attempt to undo the check out, you will get an error dialog indicating that the file is in use. You must close the file first, then undo check out.

### **Update**

While you are working on a program in a shared project repository, you may want to periodically update your program to receive any changes made by others who are working in the same shared project repository. Use the **Update...** option to bring your copy into sync with the latest version of the program in the repository. If your changes conflict with those made in the latest version, you will be prompted to [merge](#) changes from the latest version into your program.

Consider this scenario: Suppose you are working on version 4 of 5 of a program. The "5" indicates that there are 5 versions of the program in the repository. The "4" indicates that your working copy of the program is based on the version "4" version that you checked out from the repository. (To see the version numbers for your programs, check the [file's status](#) in the Ghidra Project Window [data tree](#)). When you update, you will update to the latest version in the repository (5). After the update is complete, your file status will show "Version 5 of 5" just as though you had *checked out* version 5. The **Update...** option allows you to have the latest changes applied to your program without your having to check in your file.

To update your current program either select the program in the Ghidra Project Window and click on the update icon  in the tool bar, or right mouse click and choose the **Update...** option. The Update option is only enabled when the latest version number on the server is **greater** than the version that you checked out.



The  icon is also available from the tool where you have the file opened.

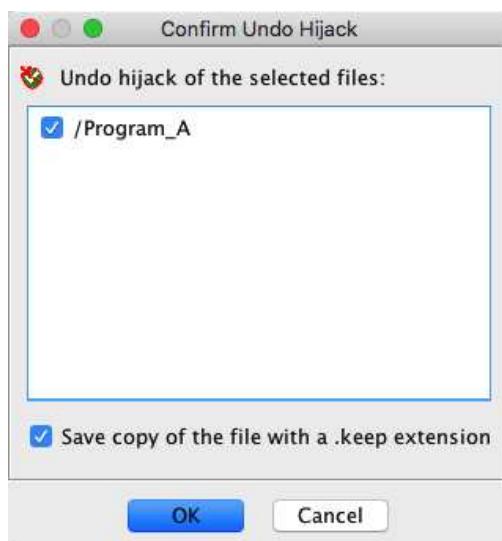


The update action is not applicable in a non-shared project repository.

### **Undo Hijack**

A file becomes "hijacked" when it exists locally as a private file in your project *and* a file of the same name exists in the repository. This will happen when user another adds a file to version control while you have a private file of the same name in your shared project. It can also happen if your checkout of the file is terminated. The [file icon](#) in the Ghidra Project Window changes to indicate that it is hijacked. To undo the hijack:

1. Close the file if you have it open.
2. Right mouse click on the hijacked file(s) in the Ghidra Project Window .
3. Select **Undo Hijack....** The following dialog is displayed to confirm the undo hijack.



Deselect the checkbox next to the name if you do not want to undo the hijack for that file. The checkbox, *Save copy of the file with a .keep extension*, is selected by default; if the checkbox is selected, a .keep file will be created. In this example, you would see **SharedProgram.exe.keep** in your project data tree after you select the **OK** button. The checkbox selection applies to all the files that you have selected for the undo hijack.

### Show Version History

To show the history on any versioned file, right mouse click on the file in the Ghidra Project Window, and select the **Show History...** option. The table shows the date on which the version was created, the user that created it, and the comments describing the version.

Version History for Program_A			
Version	Version Date	User	Comments
3	2019 Feb 26 11:43 AM	User-1	Comment 3
2	2019 Feb 26 11:41 AM	User-1	Comment 2
1	2019 Feb 26 11:40 AM	User-1	Comment 1

**Dismiss**

### [View Version](#)

To view any version in the history, select the version, right mouse click and choose the **Open With** → <tool> where <tool> denotes a menu item for each tool in your tool chest. The version is read only and is opened in the selected tool. The filename shown in the title of the tool indicates the version number, e.g., "SharedProgram.exe@10 [Read Only]" indicates you are viewing version 10 of SharedProgram.exe. You can make changes to the file, but you must save it to a new name.



Other ways to open a specific version in a tool are:

- Drag a version from the *Version History* dialog to a running tool, the running tool's icon, or to a tool icon in the tool chest.
- If you have a [default tool](#) specified, double click on the version that you want to open.
- Choose the **File** → **Open...** option; the [Open Program dialog](#) is displayed; from this dialog you can select a version to open.

### [Delete](#)

You can delete the first and last version if you are the owner, or if you are an [administrator](#) in the project, and if the file is not checked out. If the user who has the file checked out is not available to either undo his checkout or check in his file, the administrator may [terminate the checkout](#) in order to delete the version.

### [View Checkouts](#)

To view a list of who has a file checked out, right mouse click on the file in the Ghidra Project Window, and select the **View Checkouts...** option.

View Checkouts for Program_A					
Checkout Date	Vers...	User	Hostname	Project Name	Location
2019 Feb 26 10...	1	User-1	host1	TestRepo	/path
2019 Feb 26 10...	1	User-1	host1	TestRepo	/path

**Dismiss**

The *Checkout Date* is when you checked out the file; the *Version* is the version number of the file that you have checked out.

If you have administrative privileges in the project repository, you can terminate the checkout. Right mouse click

on the version and choose the **Terminate Checkout** option. A dialog is displayed to confirm the terminate checkout action. The administrator may need to do this if users who have files checked out are no longer working on the project. If your checkout is terminated, the file becomes [hijacked](#).

## Find Checkouts

To view a list of all the files that you have checked out in a folder and all of its subfolders, select a folder, right mouse click and choose **Find Checkouts...**. In the sample image below, all checkouts from the root project directory (pathname of "/") are displayed; one file from the "TestFiles" folder is checked out.

Name	Pathname	Checkout Date	Version
My_Program	/myFolder...	2019 Feb 25 05:54 PM	1
Program_A	/	2019 Feb 25 05:54 PM	4

*Name* is the name of the file; *Pathname* is the complete path to the file. *Checkout Date* is when you checked out the file. *Version* is the version number of the file that you have checked out.

From this dialog, you can [check in](#) your files or [undo your checkout](#). Make a selection in the table, right mouse click and choose **Check In...** or **Undo Checkout**. You can also click on the toolbar icon to check in, or click on the icon to undo the check out.

### Related Topics:

- [Ghidra Server](#)
- [Creating a Shared Project](#)
- [Merging Program Files](#)
- [Open a Version](#)
- [Ghidra File Status](#)
- [Project Information](#)

# Ghidra Programs

A Ghidra [program](#) is an executable unit of software or some group of data. It can be viewed and analyzed within a [Ghidra tool](#). A Ghidra program is stored in a project folder. An assembly [language](#) is associated with a program at the time it is created. The language is used for disassembling bytes into [instructions](#). Each program defines its own [address spaces](#) and [memory](#). Various program elements can be added to the program to further define it as part of the reverse engineering process. Some of the elements that can be defined in the program are labels, references, comments, functions, and data.

A Ghidra program is created by [importing](#) a file into a [Ghidra project](#). This can be accomplished from either the [Ghidra Project Window](#) or any Ghidra tool. Once a program is part of a Ghidra Project, the following actions can be performed:

- [Open](#)
- [Close](#)
- [Rename](#)
- [Save](#)
- [Delete](#)
- [Export](#)
- [About](#)

## Related Topics:

- [Ghidra Projects](#)
- [Ghidra Tools](#)

# Importer

## Introduction

Ghidra can import a variety of different types of files into a Ghidra project as Ghidra "programs." There are separate actions for importing single files, importing multiple files, and importing a file into an existing program. The actions for importing single or multiple files into a new program are available in both the front-end project window or the CodeBrowser tool. The action for adding to an existing program is only available from the Code Browser tool and only if there is a currently open program in the tool.

## Supported Formats

- Common Object File Format (COFF)
- Debug Symbols (DBG)
- Executable and Linking Format (ELF)
- Ghidra Data Type Archive Format
- GZP Input Format
- Intel Hex
- Mac OS X Mach-O
- Module Definition (DEF)
- Motorola Hex
- New Executable (NE)
- Old-style DOS Executable (MZ)
- Portable Executable (PE)
- Preferred Executable Format (PEF)
- Program Mapfile (MAP)
- Raw Binary
- XML Input Format

## File Import Actions

These actions can be used to import one or more files into a Ghidra project. They can be accessed via the File menu in either the Front-end Project Window or the CodeBrowser Tool unless otherwise specified.

### Import File

This action is used to import a single file into Ghidra. If the file is an archive consisting of multiple programs, then this action will bring up the [Batch Importer Dialog](#), otherwise it will use the standard single file [Importer Dialog](#) to complete the import.

*Steps:*

- Invoke the action from the **File ➔ Import File...** menu item.
- Select the file to import using the filechooser that appears.
- Use the [Importer Dialog](#) (or the [Batch Importer Dialog](#) if it is an archive) that pops up to configure the import.
- Press OK to initiate the import.
- A results summary dialog will appear and, if successful, the new program will appear in the project window and if initiated from a CodeBrowser tool, it will be opened in the tool.

### Batch Import

This action is used to import multiple files by selecting a root directory and letting it recursively find programs to import.

*Steps:*

- Invoke the action from the **File ➔ Batch Import...** menu item.
- Use the filechooser dialog that appears to select a root directory for searching for files to import.
- Use the [Batch Importer Dialog](#) that appears to select and configure files for importing.
- Press OK on the dialog to initiate importing the selected files.
- A results summary dialog will appear and, if successful, the new program(s) will appear in the project window. If the action was initiated from the CodeBrowser tool and only a few files were imported, they will be opened in the CodeBrowser tool.

### Open File System

This action is used to open the **File System Browser** which can be used to view the contents of container files (tar, zip, etc.) and import files from within those containers.

*Steps:*

- Invoke the action from the **File ➔ Open File System...** menu item.

- Use the dialog that appears to browse the contents of the container file and import files as desired.

### Add to Program

This action is used to import data from a file into an existing program. The program must be open in the tool to perform this action.

*Steps:*

- Invoke the action from the **File→Add to Program...** menu item.
- Use the filechooser dialog that appears to select a root directory for searching for files to import.
- Use the [Importer Dialog](#) to configure the import.
- Press OK on the dialog to initiate importing the selected files.
- When the import is complete, the currently open program should have additional data in it.

## Other Import Actions

### Import Selection

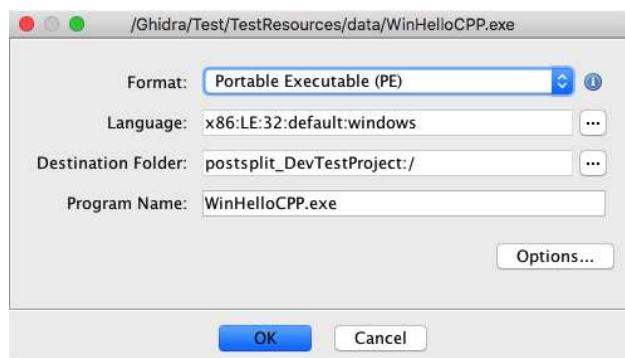
This action is used to import a selection from an open program in the CodeBrowser tool.

*Steps:*

- Make a selection in the Listing window in the CodeBrowser tool.
- Invoke the action by right-clicking and from the popup menu, select the **Extract and Import...** menu item.
- Use the [Importer Dialog](#) to configure the import.
- Press OK on the dialog to initiate importing the selected files.
- When the import is complete, a new program will appear in the project window and also be opened in a new tab in the Listing Window.

## Importer Dialog

When the user initiates a single file import, the **Importer Dialog** is used to configure the import for that file.



### Dialog Fields

- **Format** – This field is a drop-down list containing all the valid [file formats](#) that could be used to import the file. Typically, there are two options available. One for the actual format of the file (if Ghidra could detect it) and other is the **Raw Binary** format, which is always an option regardless of the actual file format and it will simply import the bytes in the file without any interpretation.
- **Language** – This field specifies the language/compiler specification that will be used in the resulting program. Often, this will be automatically detected from the file format. The [Language/Compiler Spec Chooser Dialog](#) can be used to enter or change the language/compiler spec that will be used.
- **Destination Folder** – This field is used to specify the destination folder within the current project for where the newly imported program will be saved. If a folder is selected in the front-end project window, then this field will default to that folder, otherwise the root folder will be the default. The ... will bring up a dialog for changing the destination folder.
- **Program Name** – This field specifies the name for the newly imported program. By default, it will be the name of the imported file with any format specific extension removed (e.g., .xml, .gdf). Path information at the beginning of the this field will be used to create a destination folder in the current project under the root folder specified by the **Destination Folder** field.
- **Options...** – This button will pop up format specific options for the import.

 If this dialog appears as a result of the **Add To Program** action, then the Language, Destination Folder, and Filename fields will be disabled since these values are already determined by the existing program.

## Options

The import options differ depending on the selected format.

### Common Options

These options appear many of the standard executable program formats such as ELF, PE, etc.

#### *Apply Processor Defined Labels*

If this option is on, the importer will create processor labels at specific addresses as defined by the processor specification. This is usually used to label things like the reset vector or interrupt vector.

#### *Anchor Processor Defined Labels*

If this option is on, labels created from the processor specification are **anchored**. This means that if the image base is changed or a memory block is moved, those symbols will remain at the address they were originally placed. If the option is off, the symbols will move with the image base or the memory block.

#### *Create Export Symbol Files*

Creates symbol files for each library used by an executable. The symbol file will contain stack purge information for each exported symbol in the library. The symbol files will be stored in  
<GHIDRA\_INSTALL\_DIR>/Ghidra/Features/Base/data/symbols/<OS>

 When running Ghidra with .symbols files created from an older operating system, you may receive the following warning message:

*Unable to locate [symbol\_name] in [<filepath>.exports]. Please verify the version is correct.*

This warning message indicates which symbols do not exist in the corresponding .exports file. The only information lost by not including these symbols is function purge and comments. If you require this information, manually delete the .exports file and Ghidra will regenerate it.

#### *Load External Libraries*

Recursively resolves the external libraries used by the executable. The entire library dependency tree will be traversed in a depth-first manner and a program will be created for each library. The [external references](#) in these program will be resolved. The "..." button will bring up the [Library Paths Dialog](#)

### COFF Options

COFF format has all the [Common Options](#), plus:

#### *Attempt to link sections located at 0x0*

If selected, sections located at 0x0 will be relocated sequentially in memory. This will avoid conflicts and keeps sections from being ignored.

### ELF Options

ELF format has all the [Common Options](#), plus:

#### *Perform Symbol Relocations*

If selected, Ghidra will attempt to apply the relocations specified in the ELF header.

#### *Image Base*

Specifies the image base to use for importing the memory sections.

***Import Non-loaded Data***

If selected, Ghidra will import ELF sections that don't get loaded into memory when the program is run. These sections will not be stored in a special address space called "other".

***Fixup Unresolved External Symbols***

If selected, Ghidra will attempt to resolve external references against other programs already imported into Ghidra that are in the destination folder for this import.

***Intel Hex Options******BassAddress***

This field is used to specify the start address in memory for where to load the bytes.

***Overlay***

If selected, the bytes will be loaded as an overlay. A new overlay space will be created with the same name as the Block Name.

***Block Name***

This field is used to specify the name of the memory block that will contain the newly imported bytes.

***Mach-O Options***

The Mac OSX Mach-O format has only the [Common Options](#).

***Motorola Hex Options******BassAddress***

This field is used to specify the start address in memory for where to load the bytes.

***Overlay***

If selected, the bytes will be loaded as an overlay. A new overlay space will be created with the same name as the Block Name.

***Block Name***

This field is used to specify the name of the memory block that will contain the newly imported bytes.

***MZ Options***

The MZ format has only the [Common Options](#).

***PE Options***

The PE format has all the [Common Options](#), plus:

***Parse CLI headers (if present)***

If selected, any CLI headers present will be processed.

***Raw Binary Options******Block Name***

The name of the memory block that will contain the raw bytes from the file. By default, it will be the name of the default address space (usually "ram")

***BaseAddress***

This field is the address offset for the block of bytes to be imported. By default, this will be 0.

***File Offset***

This field is the byte offset into the imported file from which to start importing raw bytes. By default, this will be 0.

***Length***

This field is the number of bytes to import. By default, this will be set to the total number of bytes in the imported file.

***Apply Processor Defined Labels***

If this option is on, the importer will create processor labels at specific addresses as defined by the processor specification. This is usually used to label things like the reset vector or interrupt vector.

***Anchor Processor Defined Labels***

If this option is on, labels created from the processor specification are ***anchored***. This means that if the image base is changed or a memory block is moved, those symbols will remain at the address they were originally placed. If the option is off, the symbols will move with the image base or the memory block.

***XML Options***

The XML format is used to load from a Ghidra XML formatted file. The options are simply switches for which types of program information to import.

***Memory Blocks***

Imports memory block definitions (name, start address, length, etc). See [Memory Map](#).

***Memory Contents***

Imports bytes for the memory blocks.

***Instructions***

Imports disassembled instructions. See [Disassembly](#).

***Data***

Imports data types and defined data. See [Data Type Manager](#) and [Data](#).

***Symbols***

Imports user-defined symbols. See [Symbol Table](#).

***Equates***

Imports equate definitions and references. See [Equate Table](#).

***Comments***

Imports comments (pre, post, eol, plate, repeatable). See [Comments](#).

***Properties***

Imports user-defined properties.

***Bookmarks***

Imports [Bookmarks](#).

**Trees**

Imports program organizations (program trees, modules, fragments). See [Program Tree](#).

**References**

Imports user-defined memory, stack, and external references. See [References](#).

**Functions**

Imports functions, stack frames and variables. See [Functions](#).

**Registers**

Imports program context and registers. See [Register Values](#).

**Relocation Table**

See [Relocation Table](#).

**Entry Points**

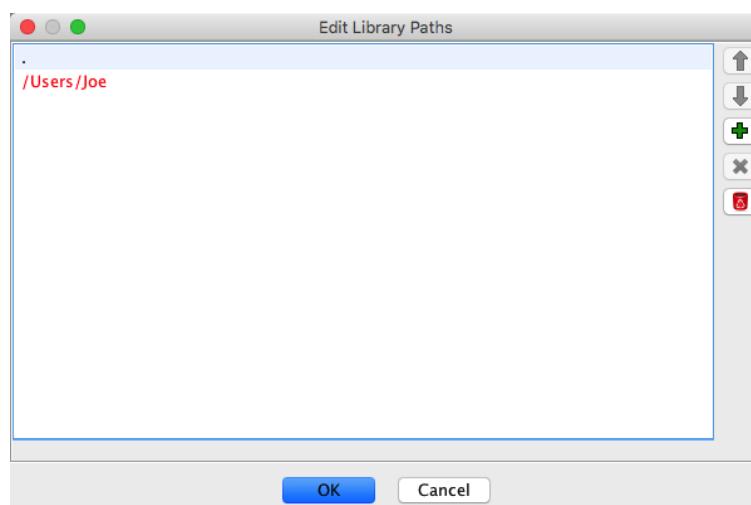
Imports program entry points.

**External Libraries**

See [External Program Names](#).

## Library Search Path

The Library Search Path dialog is used to specify the directories that Ghidra should use to resolve external libraries (e.g.; \*.dll, \*.so) while importing.



### Change the Library Path Search Order

To change the search order of the paths within the list:

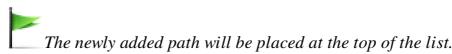
1. Select a path from the list
2. Select the button to move the path **up** in the list
3. Select the button to move the path **down** in the list



*The search order is important when you have different versions of a libraries in different directories. The first directory in the search path that contains a required library is the one that Ghidra will use.*

**Add Library Search Path**

1. Click the  button
2. Select a directory from the file chooser
3. Click the "Select Directory" button

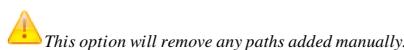
**Remove Library Search Path**

1. Select one or more paths from the list
2. Click the  button

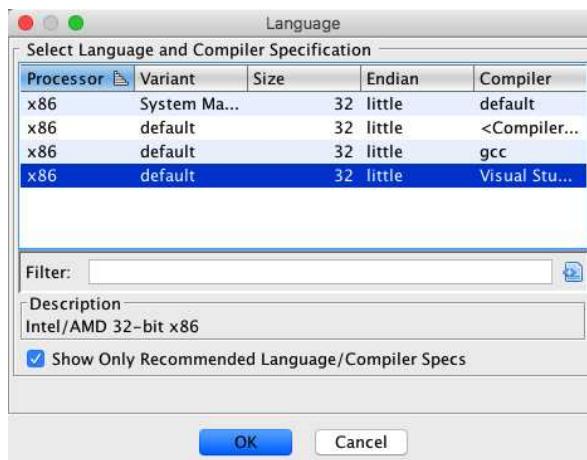
**Reset Library Search Paths**

To reset the paths to the default list:

1. Click the  button
2. Click "Yes" on the pop-up dialog to confirm path reset

**Language and Compiler Specification Dialog**

This dialog is used to specify the Ghidra language (Processor/Compiler Spec) of the program being imported. Certain formats, like "PE", "ELF", or "XML", will usually choose the appropriate language/compiler spec. If not, this dialog can be used to select one or override the default selection.



Each row in the table represents a unique processor/language/compiler spec pair. To select one, simply click on the row and press the **OK** button.

**Table Columns**

- **Processor** – The processor for this selection
- **Variant** – Some processors have different versions of the processor. The primary variant is called "default". Any other variants should have a meaningful name
- **Size** – the size in bits of the processor address space
- **Endian** – the endianess of the processor
- **Compile** – the compiler specification used to build the program

**Filter**

The filter can be used to reduce the number of entries in the table. Only the entries that contain the text in this field will be displayed.

**Description**

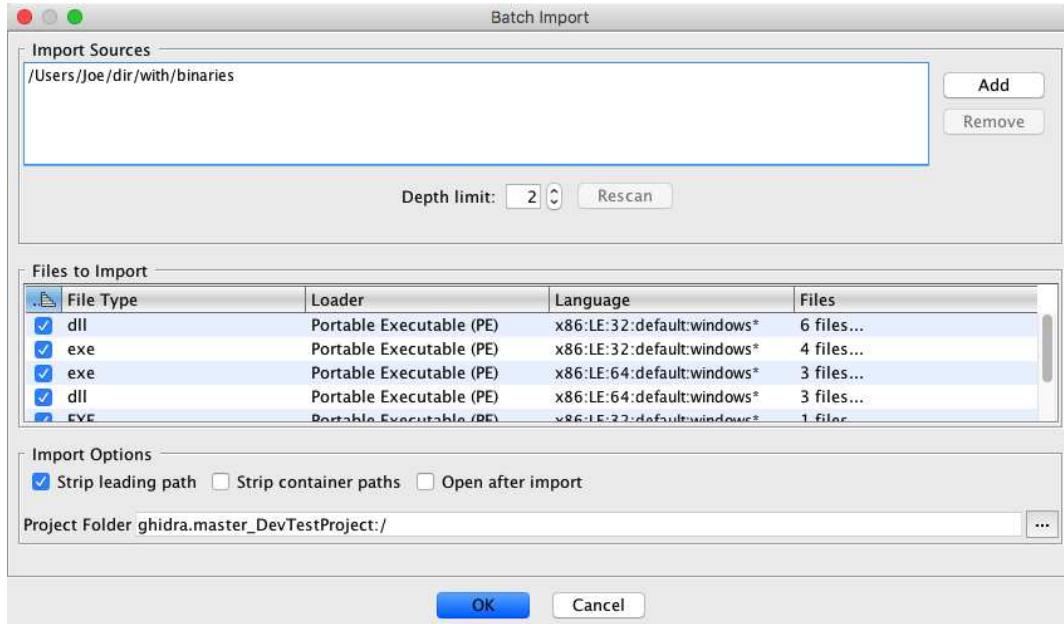
This field shows the currently selected language/compiler spec.

**Show Only Recommended Language/Compiler Specs**

If selected, only the languages suggested by the selected importer format will be shown. Otherwise, all known languages will be shown. Not all importer formats can determine an appropriate language, in which case all the languages will be displayed.

## Batch Import Dialog

The Batch Import Dialog is used to import multiple files at the same time. The files may be individual files in a directory tree, and/or files from an archive file of some sort such as a zip or tar file.



### Import Sources

This section manages a list of folder trees or container files (e.g., zips) to scan for files to import. Initially, this contains the folder or file that was initially selected from the file chooser.

#### *Adding an additional import source folder.*

Pressing the **Add** button will bring up a file chooser for picking an additional folder or file (import source) to search for import files

#### *Removing an import source folder*

Select a folder in the import sources window and press the **Remove** button.

#### *Depth limit*

This field specifies the depth or level of nested containers to search for each of the specified import sources. Note that this is not the level of subfolders to search, but rather the nesting levels of archive type files. (i.e. zips in zips)

#### *Rescan*

This button will rescan the import sources to the current depth for files to import.

### Files to Import

This section displays a table showing the files that were found. Each row represents a set of similar files that can be imported. The table columns are as follows:

- –if checked, this set of files will be imported.
- **File Type** –displays the file extension
- **Loader** –displays the format (Loader) that will be used to import the file.
- **Language** –displays the language that will be used (if applicable). Clicking on this field will pop up a list of acceptable languages to choose from.
- **Files** –displays the number of files in the group. Clicking on this field will pop up a list of the files in the group.

### Import Options

**Strip leading path**

If selected, the newly imported files will not use the relative path of the file when storing the result in the project. Otherwise, the file will be in a corresponding relative path in the project.

**Strip container paths**

If selected, the newly imported files will not use the interior archive path when storing the result in the project. Otherwise, the file will be in a corresponding relative path to the path the file was in its archive.

**Project Destination**

This shows the destination folder in the project that will be the root folder for storing the imported files. Each imported file will be stored in a relative path to that root folder. The relative path is usually the relative path of the file to its import source folder, but can be adjusted with some of the path options described earlier.

Provided By: *Importer Plugin*

# Exporting Files

Ghidra provides an *Exporter* that allows a user to output program information into a file in various formats.

Some of the formats the *Exporter* supports are:

- [ASCII](#)
- [Binary](#)
- [C/C++](#)
- [Ghidra Zip File \(.gjf\)](#)
- [HTML](#)
- [Intel Hex](#)
- [XML Export Format](#)

## Export Action

The export action can be invoked from the front-end project window or the CodeBrowser tool.

### To export from the front-end project window:

#### *Steps:*

- Right-click on the file to export in the tree.
- From the popup menu that appears, select the **Export...** menu item.
- Use the [Exporter Dialog](#) that appears to configure the export.
- Press the **OK** button to perform the export.

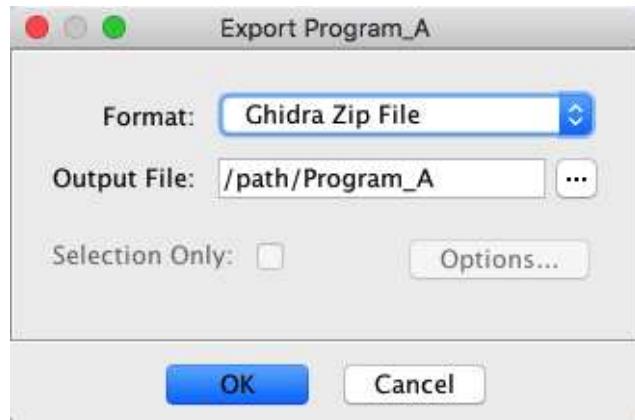
### To export from the CodeBrowser tool:

#### *Steps:*

- Make sure the program to export is the currently open program in the CodeBrowser tool.
- Invoke the action from the **File → Export File...** menu item.
- Use the [Exporter Dialog](#) that appears to configure the export.
- Press the **OK** button to perform the export.

## Export Dialog

The Export Dialog is used to configure the export of the chosen program.



### Dialog Fields

- **Format** –This field is a drop-down list containing all the valid [export file formats](#) that could be used to export the program. By default, the last used format will be auto-selected.
- **Output File** –This field specifies the output file for the export. By default, the output file's name will be the name of the program and the output folder will be the user's home folder or the last folder used for an export if an export has been performed in the current session. Use the "..." button to bring up a file chooser to change the output file.
- **Selection Only** –If this checkbox is selected, then only the areas of the program that are in the current selection will be exported. Obviously, this only applies when exporting from an open program with a selection in the CodeBrowser tool and not when exporting from the front-end project window. Also, not all export formats support partial exports. The GZF format, for example, always exports the entire program since it is really just making an exported copy of the entire program database.
- **Options...** –This button will pop up format specific options for the import.

## Exporters

### Ascii

Creates a plain text representation of the program's listing, similar to what is displayed in the [Code Browser Field Format](#).

#### *Ascii Options*

---

*Advanced*

*Label Suffix* the string to append on

the end of labels

*Comment Prefix* the string to append on

the begining of comments

*Show*

The check-boxes in this panel are used to determine what program elements should be included in the output file. A selected check-box denotes that the

corresponding element will be included in the file. The checkboxes for elements that are present in the program are selected by default.

<b>Comments</b>	Include <i>Pre</i> , <i>Post</i> , <i>EOL</i> , and <i>Plate</i> comments
<b>Properties</b>	Include properties; e.g., <i>Bookmarks</i> , <i>Spacers</i>
<b>Structures</b>	Include <i>Structures</i> and <i>Unions</i> defined on code units
<b>Undefined Data</b>	Include all undefined code units (e.g., "?") or replace with "[BYTES REMOVED]" place-holder
<b>Ref Headers</b>	Include the cross reference header <i>BACK[m,n]</i> or <i>FWD[m,n]:</i> , where <i>m</i> is the number of cross references and <i>n</i> is the number of offcut cross reference; select the <i>Back Refs</i> and/or <i>Forward Refs</i> for the <i>Back/FWD</i> header to show up
<b>Back Refs</b>	Include the list of cross references for each code unit
<b>Forward Refs</b>	Include the list of references to the mnemonic for each code unit
<b>Functions</b>	Include signature and header for each function

#### Width

The text-fields in this panel specify the width, in number of characters, to use when displaying program elements in the output file.



*Setting a width to zero (0) effectively excludes it from the output file.*

## Binary

Creates a binary file containing only the bytes from each memory block in the program. If the program was originally created using the **Binary Importer**, then this exporter allows recreation of the original file.



*Only initialized memory blocks are included in the output file.*

## C/C++

Create a C/C++ file containing all datatypes from the program's [data type manager](#) and all of the functions in the program.

### C/C++ Options



- **Create Header File (.h)** – Select to create a .h file.
- **Create C File (.c)** – Select to create a .c file.
- **Use C++ Style Comments (//)** – Select to use // or /\* style comments.

## Ghidra Zip File (.gjf)

Creates a zip file from a program in your project. You may want to create a zip file so that you can give it to another user who can then [import](#) into their project.

## HTML

Creates a hyper-text representation of the program's listing, similar to what is displayed in the [Code Browser Field Format](#). The HTML output is analogous to the ASCII output, however HTML allows format and hyper-link information to be added to the file. The formatting allows fields to be color-matched to

those in the Code Browser. The hyper-linking allows navigation similar to that supported in the Code Browser.

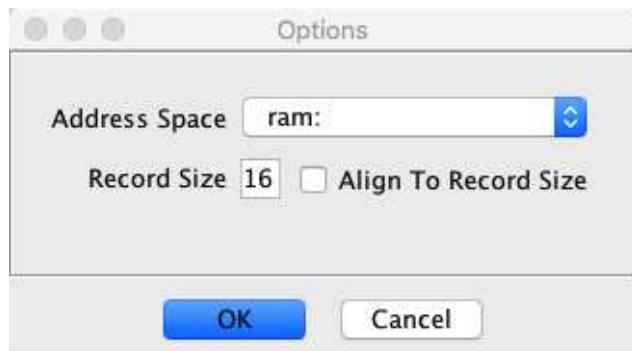


*The HTML Options are identical the [ASCII Options](#).*

## Intel Hex

The Intel Hex format, a printable file representing memory images, was originally designed to program EPROM devices. The Intel Hex exporter creates files in this format which can be used to program these EPROM devices.

### *Intel Hex Options*



- **Address Space** –Specifies which address space to export as Intel Hex format only supports one address space. This option will be initialized to the "default" address space.
- **Record Size** –Specifies the size (in bytes) of each record in the output file. The default 16.
- **Align To Record Size** –If checked, this will ensure that **only** records matching the record size will be output. eg: if you set the record size to 16 but there are 18 bytes selected, you will see only one line of 16 bytes in the output; the remaining 2 bytes will be dropped.

## XML

The XML Exporter creates XML files that conform to Ghidra's Program DTD. You can re-import files in this format using the [XML Importer](#).



*The XML Options are identical the [XML Importer Options](#).*

### Related Topics:

- [Importing Files](#)

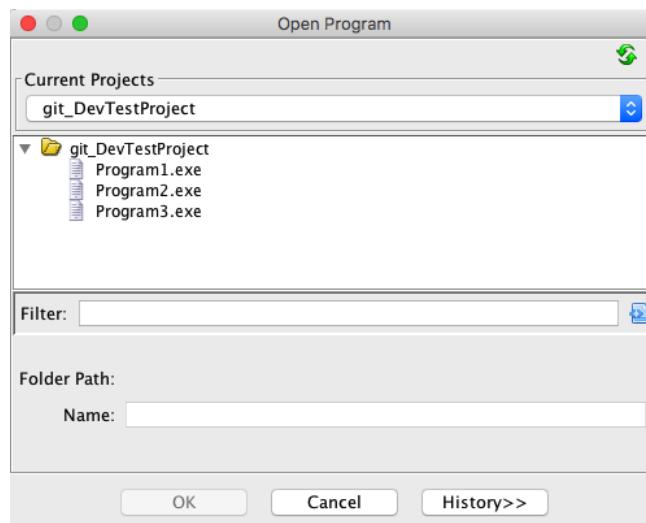
# Opening Program Files

Ghidra Tools can open an existing Ghidra [program](#) file and present it for review and analysis. Programs from other [viewed projects](#) or [viewed repositories](#) may also be opened.

A program can be opened in the currently displayed tool or can be displayed in a [new instance of a Tool](#).

## Opening a Program File in the Current Tool

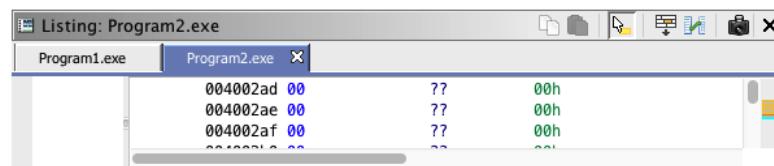
1. To open a program file in the currently displayed tool, select **File ➔ Open...** from the Ghidra Tool's menu.
2. The *Open Program* dialog is displayed.



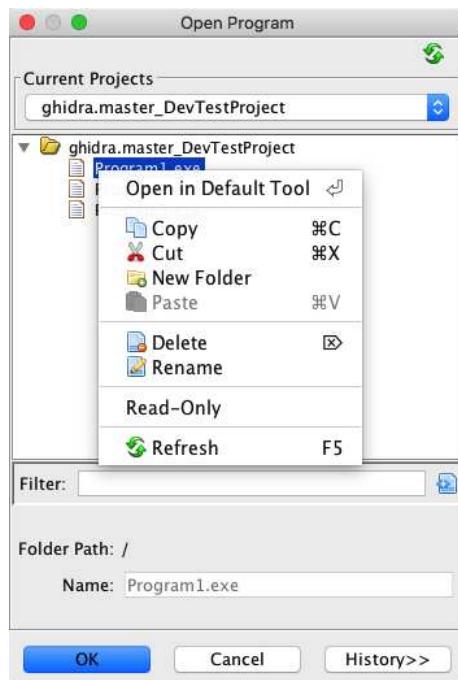
This allows selection of any file that is part of the active project.

3. Select the program file to open.
4. Click the **OK** button *OR* double click on the program to open.

The selected program is opened and displayed in the tool. More than one program can be opened at the same time, but only one of them can be *active* at a time. The Code Browser window shows a tab for each program that you have opened. Select a tab to make that program the active one, as shown in the image below.

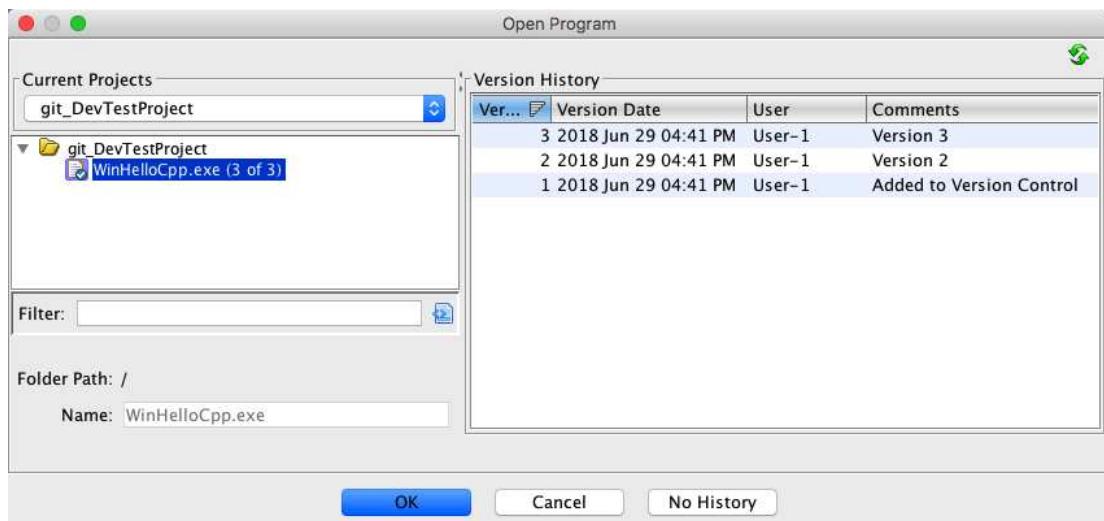


In addition to selecting a file, this dialog can be used to perform some basic directory/file operations. Right click on a program to get the directory/file menu.



### History

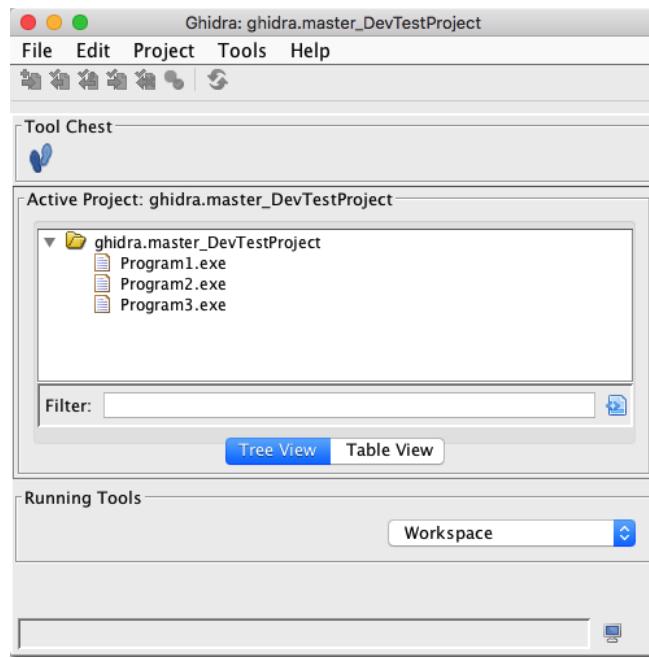
The History button on the *Open Program* dialog expands the dialog to show previous versions of a program (if the selected program is [shared](#)), allowing the user to view a read-only previous version of the program.



The History panel shows all previous versions for the selected program. Each entry shows which user created the version, the date and time the version was created, the version number, and the comment for that version. To open a history file, select it in the Version History table, and press the "OK" button. The version history can be hidden by pressing the "No History" button.

### Opening a Program in a New Tool via Drag and Drop

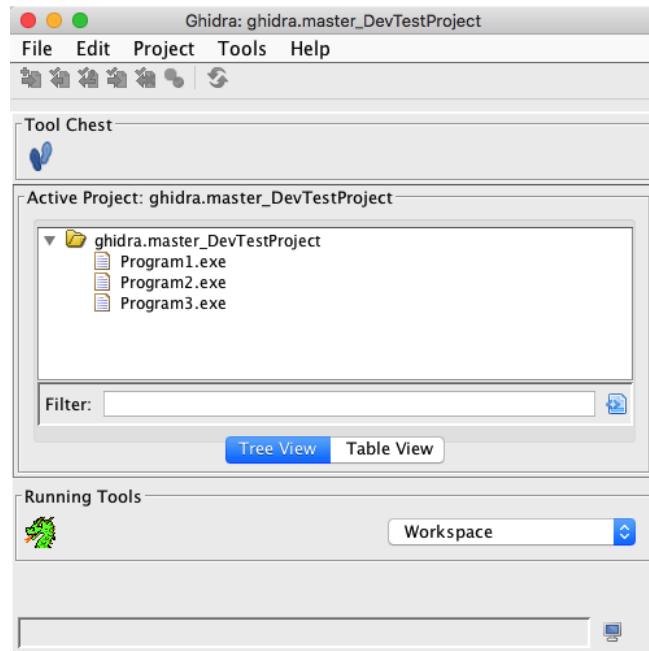
1. Locate the program to open in the Ghidra Project Window.



2. Left mouse press on the program in the tree, drag it to the Tool Chest, and drop it on the desired tool by releasing the left mouse button.

The icon in the Tool Chest indicates the CodeBrowser tool.

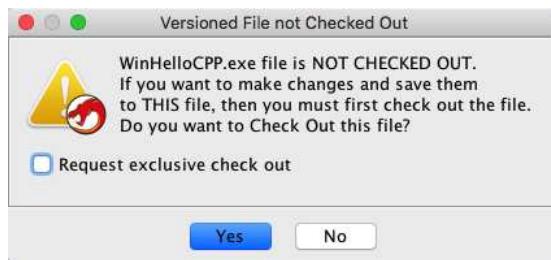
3. A new instance of the tool is launched with the selected program open. The Running Tools area of the Ghidra Project Window now shows the newly launched tool.



Alternatively, programs can be dropped onto running tools (either the icon in the Running Tools area or onto the tool itself). In this case, the program is opened in the existing tool in addition to any programs that are already open.

### Opening a Versioned Program File

If you attempt to open a versioned program file that is not checked out, a dialog is displayed to warn you of this. You will not be allowed to save changes to this file unless you check it out.



If you are working in a [shared project](#), AND if you plan to make drastic changes to memory, e.g., add or remove memory blocks, select the checkbox on the dialog to obtain an [exclusive lock](#) on the program file.

If you choose the "No" option, the program will be opened *read only*, thus you will have to save your changes to *another* filename.

Provided by: *ProgramManagerPlugin*

Related Topics:

- [Ghidra Programs](#)
- [Importing Programs](#)
- [Closing Programs](#)
- [Shared Project](#)

## Closing Programs

Closing a program will remove the program from the Tool. The Tool will remain displayed but will display either another open program or be empty.

If the program to be closed has changes and is open in only one tool, the user will be prompted to save the program. However, if the program is open in multiple tools, the user will not be prompted to save even if changes have been made. In other words, the only time the user will be prompted to save a program that is being closed is when it has been changed and is open only in one tool.

### To Close a Program File

1. From the Tool menu, select **File** ➔ **Close**
2. If changes were made to the program and they haven't been saved yet, the *SaveProgram?* dialog appears.

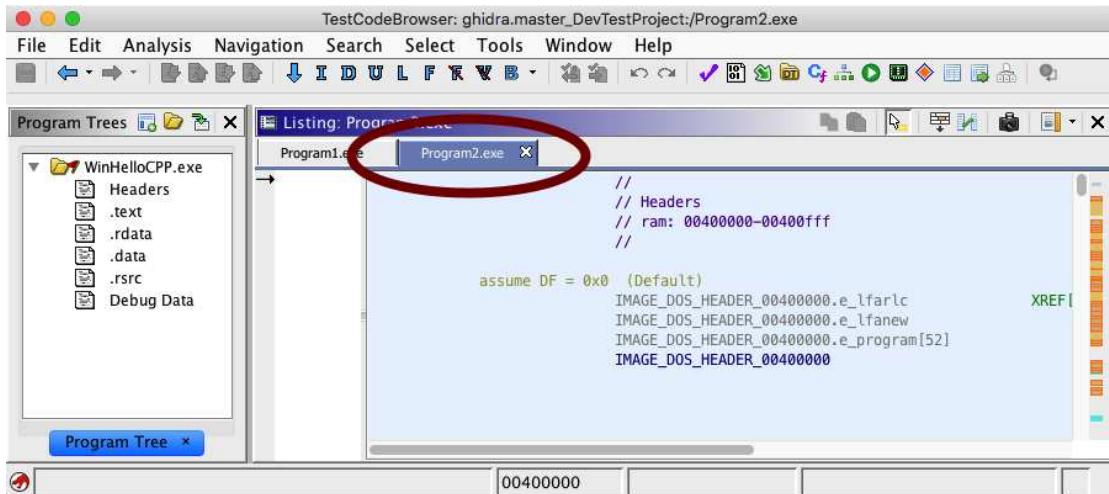


The buttons in the *SaveProgram?* dialog perform the following functions.

- **Save** –Saves the program changes and close the program in the current tool.
- **Don't Save** –Discards any changes that were made to the program and closes it.
- **Cancel** –Leaves the program open in the current tool without any changes being saved.



If the listing window is open and multiple programs are open, the program names are displayed on tabs across the top of the listing window. Programs can be closed by selecting the appropriate tab and pressing the corresponding "x" button.



### To Close All Programs

1. From the Tool menu, select **File** ➔ **Close All**
2. For each program that was changed, the *SaveProgram?* dialog appears.

### To Close All Programs Other Than The Current Program

1. From the Tool menu, select **File** ➔ **Close Others**
2. For each of the other programs that was changed, the *SaveProgram?* dialog appears.

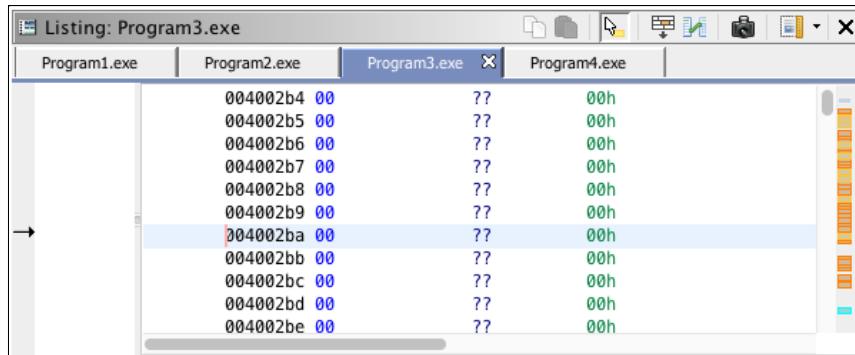
Provided by: *ProgramManager* Plugin

Related Topics:

- [Opening Program Files](#)
- [Saving Program Files](#)

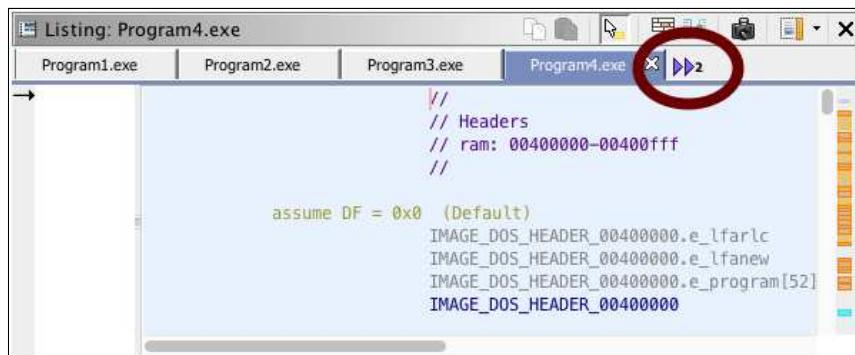
# Navigating Programs

Open programs are represented in Ghidra by tabs at the top of the Listing. You may click on these tabs to change the active program. Also, as described below, there are various actions that allow you to change the active program.



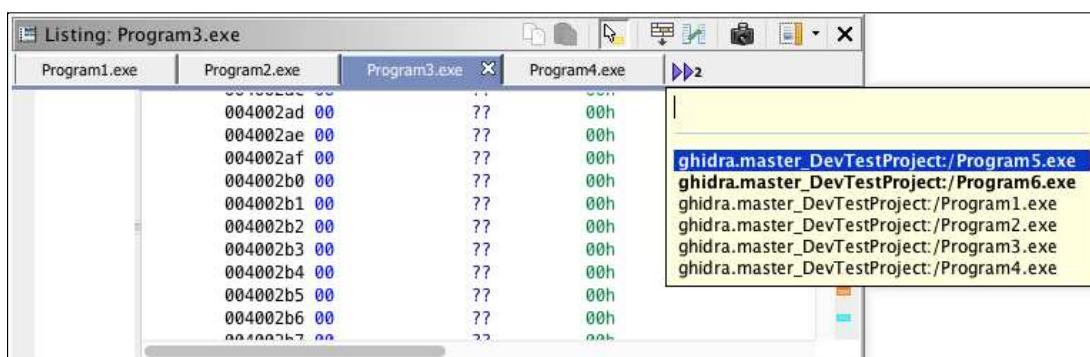
Program Tabs

When there are more open programs than room to display their respective tabs, then a button is displayed to access those programs. This button displays the number of hidden programs.



Program Tabs 'More Button'

To access these hidden programs, click the button to display a menu for selecting programs.



Program Tabs With Popup Window

The popup window displayed allows you to select a tab by clicking a program name, or by using the **up** and **down** **arrow** keys to select a program name and then pressing the **Enter** key to choose the selected program. Also, typing text data will filter the displayed program list. You can close the popup window without making a selection by pressing the **Escape** key or by clicking a component outside of the window.

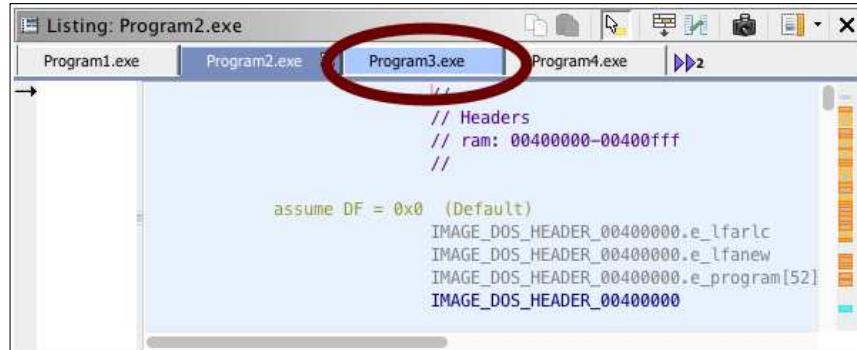
Those programs listed in bold are those that are hidden.

## Navigation Actions

### Go To Next Program and Go To Previous Program Actions

The next and previous actions are only available via keybindings, which by default are **Control-F8** and **Control-F9** for previous and next, respectively. You may change these bindings from the [Key Bindings Options](#).

The next and previous actions allow you to move between open tabs. For example, when the **Go To Next Program** action is executed, the tab to the right of the current tab is highlighted.



*Next Tab Highlighted*

After a brief pause, the highlighted tab will become the active program. Quick, repeated executions of the action will continue to move the highlighted tab to the right.

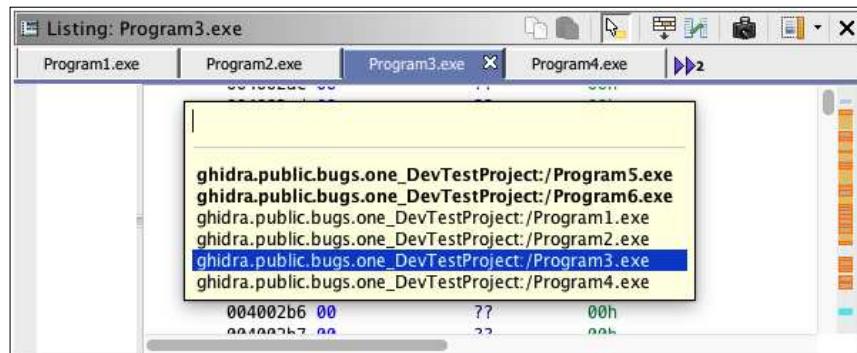
The **Go To Previous Program** action works in the same way as the **Go To Next Program** action, except that it moves the highlight to the left instead of the right.

The operation of tab highlighting varies slightly depending upon the existence of [hidden tabs](#). Without hidden tabs, the highlighting, when starting from the first or last available tab, will wrap around to the other side of the tab list. In this mode, you may highlight tabs with no limits with no program being activated until you stop.

Contrastingly, if there are [hidden tabs](#), then when the first or last tab is highlighted, then the next successive highlight action will trigger the [more tabs button](#) to be executed.

### Go To Program... Action

The **Go To Program...** action will show the program selection popup window when executed. This menu allows you to pick a program to go to.



*Go To Program Popup Window*

To execute this action, from the Tool menu, select **Navigation ➔ Go To Program...**

### Go To Last Active Program Action

The **Go To Last Active Program Action** will activate the last program that was active before the currently active program. Thus, this action is disabled when you do not have a previously active program.

To execute this action, from the Tool menu, select **Navigation ➔ Go To Last Active Program**.

Provided by: *Program Manager* Plugin

Related Topics:

- [Opening Program Files](#)
- [Closing Program Files](#)

# Saving Program

When changes are made to a program, the user must save the program. Otherwise, those changes will be lost. The user can either save the changed program back to the original program file or save the changed program to a new file, leaving the original program file unchanged.

## Save

Saves the changed program back to the original program file. The **Save** option is only enabled when changes have been made. To perform this action:

1. From the Ghidra Tool's menu, select **File ➔ Save**.

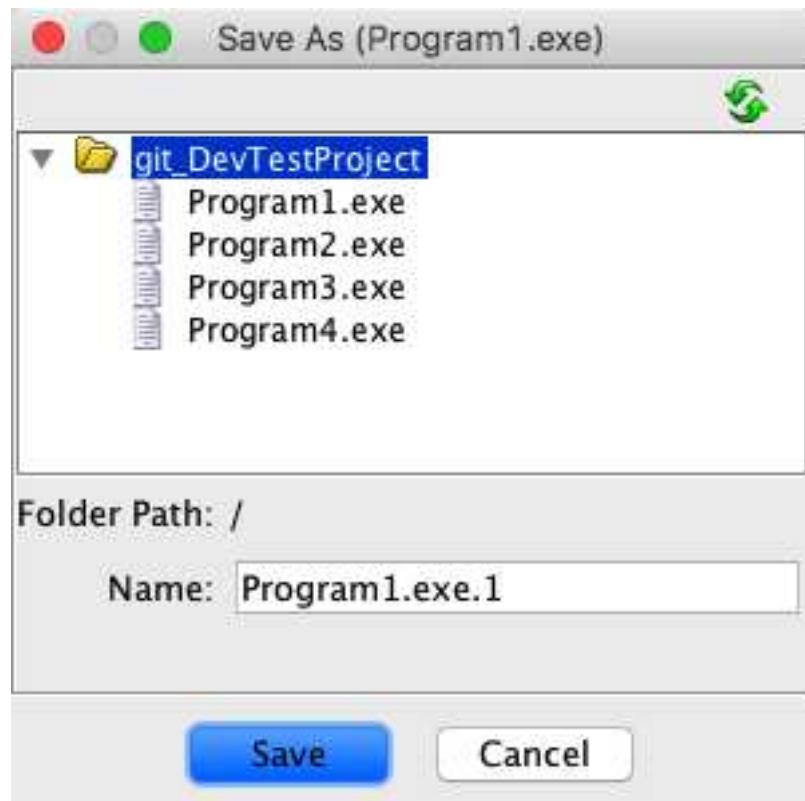
or

Select the Save icon  in the tool bar at the top of the Ghidra Tool.

## Save As

Saves the currently open program to a new program file. This new program file becomes the program that is active in the tool. When selecting **Save As...**, Ghidra will prompt for a filename. To perform this action:

1. From the Ghidra Tool's menu, select **File ➔ Save As...**
  2. The *Save As...* dialog appears.
-



3. Select the folder for saving the program and enter the new *Name* of the program.
4. Click the **Save** button.
5. The program is saved to the new name. This new program is the one now active in the Tool.

If an existing program is selected from the **Save As...** dialog, an overwrite confirmation dialog will be displayed.

## Save All

Saves any currently open programs. If any program has never been saved before, Ghidra will prompt for a filename.

1. From the Ghidra Tool's menu, select **File ➔ Save All**.

Provided by: *Program Manager* Plugin

Related Topics:

- [Open Program](#)
- [Close Program](#)

# About Program

The **About Program...** dialog displays summary information about a program.

## To view information about the currently open (active) program

- From the menu-bar of the tool that has the program open, select **Help ➔ About Program ...**

## To view information about any program in the project window

- Right click on the program and select **About...** from the popup menu.



Figure 1 – About Program

 *Notes:* When viewing the "About" information on a non-open program, you may get a very abbreviated version of the program's information if the program was created with a version of Ghidra before version 4.2. Once a program has been saved with version 4.2 or later, the full "About" information will be available.

## Standard information displayed in the "About Program" dialog

<i>Project File Name</i>	name of the ghidra program file
<i>Last Modified</i>	date the program was last modified
<i>Read-only</i>	whether the program is marked as read-only
<i>Currently Modified</i>	whether the program has been changed
<i>Program Name</i>	name of the program
<i>Language Provider</i>	name of processor language used to disassemble this program
<i>Processor</i>	name of the program's target processor
<i>Manufacturer</i>	name of manufacturer of the program's target processor
<i>Endian</i>	either "big" or "little"
<i>Address Size</i>	size, in bits, of the program's address space
<i>Start Address</i>	minimum address of the program
<i>Ending Address</i>	maximum address of the program

<i># of Memory Blocks</i>	total number of memory blocks in the program
<i># of Instructions</i>	total number of disassembled instructions in the program
<i># of Defined Data</i>	total number of defined data in the program
<i># of Symbols</i>	total number of symbols defined in the program

Below the standard information, the program properties will be displayed. The information displayed here will vary from program to program.

### To close this dialog

1. Click the **OK** button.

Provided by: *About Program* plugin

#### Related Topics

- [About Ghidra](#)
- [Importing Program Files](#)

## Merge Program Files

The Ghidra Server provides file access to multiple users enabling a team to collaborate on a single effort. It provides network storage for [shared project repositories](#) while controlling user access. Together the Ghidra Server and the shared project repository allow multiple users to concurrently:

- Check out the same version of a program file
- Make changes to the program file
- Check the changed program file back into the shared project repository

*Merging* is necessary to integrate changes made to a single version of a program file by multiple users. The following steps show a typical merge scenario:

### Check in / Merge Sequence:

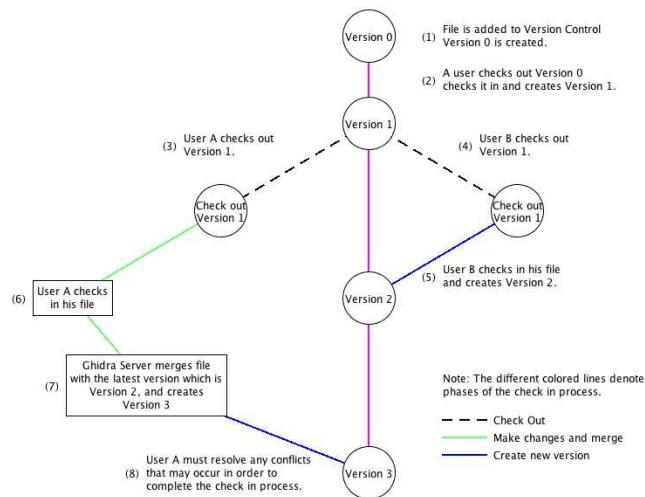
1. A file is added to version control, creating Version 0.
2. A user checks out Version 0, makes changes, and checks it in to create Version 1.
3. **User A** checks out Version 1.
4. **User B** checks out Version 1 and makes changes.
5. **User B** checks in his changed file, creating Version 2 in the project repository. Because no other versions were created since **User B** originally checked out his file, Version 2 can be created automatically. This new version will be the changed file created by **User B** (Step 4). In this case, merging is not involved in the check in process.
6. **User A** completes his changes and checks in the file.
7. During the check in process, the Ghidra Server determines that a new version of the program file has been added since **User A** originally checked out the file (Step 3). The latest version of the file contained in the repository, Version 2, is not the same file that **User A** had checked out, Version 1. A merge is required since a new version of the file was added to the repository after User A checked out the file.



If a new version of a file has been added since the user checked out the file, a merge is required at check in. This is the **only** time a merge is required.

The Ghidra Server starts the merge process by making a copy of Version 2. Changes from **User A**'s checked in file are applied to this copy to create the new file, Version 3. There are two types of merges –[automatic](#) and [manual](#). If changes made by **User A** and **User B** do not conflict [automatic](#), merging is done. If changes made by **User A** and **User B** do conflict, then **User A** must manually [resolve the conflicts](#). Note that the last user checking in changes must resolve conflicts. A conflict results when **User A** and **User B** make the *same type of changes* at a location in the program.

The figure below illustrates this typical scenario where two users check out a file, as described above. The sequence of events are numbered in parentheses in the diagram, and correspond with the **Check in/Merge Sequence** above:



To begin the merge process, select the [Check In](#) option on the Ghidra Project Window. A progress bar is displayed while the merge is in progress. The merge begins with a copy of the latest version of the file from the repository. All changes are applied to this copy. At the conclusion of the merge, this copy along with the applied changes will be saved in the repository, and will become the newest version of the file.

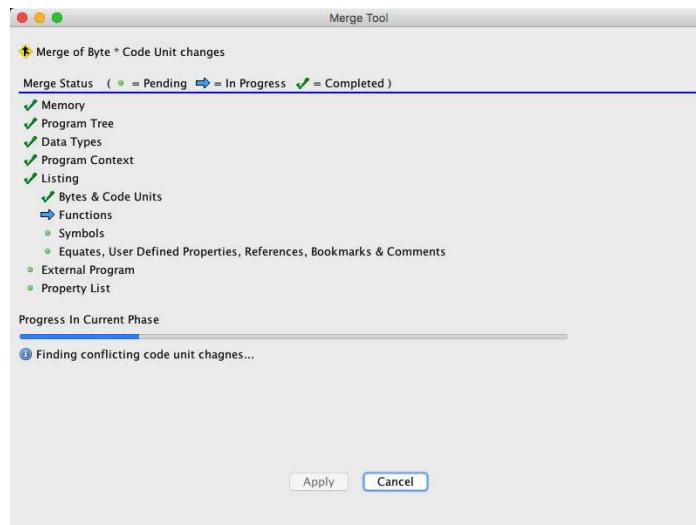
The merge processes program elements in the following order:

- [Memory](#)
- [Program Trees](#)
- [Data Types and Categories](#)
- [Program Context Registers](#)
- [Listing](#)
  - [Code Units](#)
  - [Externals](#)
  - [Functions](#)
  - [Symbols](#)
  - [Equates](#)
  - [User Defined Properties](#)
  - [References](#)
  - [Bookmarks](#)
  - [Comments](#)
- [External Program Names](#)
- [Property Lists](#)

Within each program element an automatic merge is attempted first. If conflicts arise, the user must manually resolve conflicts before moving on to the next program element. For example, the Program Tree merge performs the auto merge, and then requests user input to resolve conflicts, as required. Once the Program Tree has been merged, with all conflicts resolved, the merge of Data Types and Categories begins.

If you cancel the merge at any time, you are canceling the entire *check in* process and your file remains checked out. None of the changes that was made as a result of the merge is applied to the program that you have checked out.

While the auto merge is occurring and conflicts are being determined the merge status is displayed in the merge tool. The following image shows an example of the merge tool when the merge process is auto merging code units.



At this point the Memory, Program Trees, Data Types and Program Context phases have already completed as indicated by the Completed icon, ✓. The Bytes & Code Units sub-phase of the Listing is currently In Progress, ➔. The **Progress In Current Phase** progress bar provides the user with an idea of how much of the phase has completed. The other sub-phases of Listing, the External Programs and Property Lists have not been merged yet as indicated by the Pending icon, ⓘ.

The message below the current phase progress bar gives additional information about what is currently happening in the phase.

At the bottom of the merge tool is a progress bar and message for an immediate task. When multiple of these smaller tasks are combined sequentially they perform the merge of the current phase.

The following sections describe the auto merge process and conflict resolution process in greater detail.

### Auto Merge

For each program element, the *Auto Merge* is the first of the two merges which is performed. The following paragraphs describe how Auto Merging applies to each program element.

#### Memory

Program memory is organized in [memory blocks](#). The only changes to a memory block that can be merged are:

- Name
- Permissions
- Comments

You can make changes to the memory structure (add a block, remove a block, etc.) only when you have checked out the program with an [exclusive lock](#). This restriction prevents drastic changes in memory that could potentially have a major impact on other users. Typically, all users working on the same program should agree on memory structure changes and then have one user make them. An exclusive lock prevents anyone else from checking out the program while the exclusive lock exists. When you check in your changes, merging is not necessary as no other versions were created while you had the program checked out. Your program becomes the new version.

Comments on the memory block are *replaced* in the results program during the merge process. You must resolve all [memory conflicts](#) before the merge process can continue with the next program element, program trees.

#### Program Trees

If you make any change to a [program tree](#), it is marked as changed and may be merged. No further inspection of changes is done. If the tree was not changed in the latest version, then the automatic merge can be done. If the tree was changed in the latest version, this is considered a [conflict](#) that you must resolve before the merge process can continue with the next program elements, data types and categories.

#### Data Types and Categories

[Data types](#) can be created, renamed, edited (contents), or deleted. During merging, newly created data types are added automatically (if a data type with the same name already exists in the parent category, then your new data type is created with a ".conflict" appended to the name). All other changes to data types are merged.

When merging data types, there is only one way that data type conflicts can occur. A conflict results only if in both files the same *type* of change (rename, edit contents, delete) was made to a particular data type. Other than this, no conflicts can occur and the data type you changed can be merged automatically. For example, if you renamed a structure data type and in the latest version the contents of that data type were edited, this will **not** cause a conflict and your change can be applied automatically during the merge process.

[Categories](#) can be renamed, moved, deleted, or restructured (add/remove data types and categories). Note that moving a data type to a different parent category is considered a category change, not a data type change.

As with data types, there is only one way that category conflicts can occur. A conflict results only if, in both files, the same *type* of change (rename, move, delete, restructure) was made to a particular category. Other than this, no conflicts can occur and the category you changed can be merged automatically. For example, if you renamed a category and in the latest version the category was moved to a different parent category, this will **not** cause a conflict and your change can be applied automatically during the merge process.

All [conflicts](#) must be resolved before the merge process can continue with the next program element, Program Context Registers.

#### Program Context Registers

For any of the registers defined by the program's language you can set a register value for an address or range of addresses. If you set a register value or remove a register value at an address, it will automatically change the value during the merge process if the value wasn't changed at that address in the latest version. If your version and the latest version change the register to the same value at an address, it will also get merged automatically.

Automatic merging of all the registers will happen before you are prompted to resolve any conflicting register values.

You must resolve all [register conflicts](#) before the merge process can continue with the next program element, Listing.

#### Listing

Various parts of the program are merged in the Listing merge. These include the bytes, code units, functions, symbols, equates, user defined properties, references, bookmarks, and comments. For each of these parts the auto merge will occur followed by a manual merge of any conflicts.

#### Code Units

If you make any change to the memory bytes or to a code unit, such as creating or clearing an instruction or defining data, the code unit gets marked as changed and may be merged. If you change the memory bytes and the latest version doesn't have memory byte or code unit changes, your byte changes will be merged automatically. If you changed the code unit and it does not conflict with a change in the latest version, the change to the code unit will be merged automatically.

If the latest version had any of the following changes, the code unit is in conflict and must be manually merged:

- The latest version and your version have memory bytes changed to different values
- The latest version changed memory bytes where you changed the code unit or vice versa
- The latest version created or cleared instructions or data differently than you did
- The latest version created, changed, or removed an equate for the code unit you changed or vice versa

You must resolve all [byte and code unit conflicts](#) before the merge process can continue with the next program element, Functions.

#### *External Functions and Labels*

If you make any change to the external locations (external functions or external labels) defined within the program, such as creating, updating, or removing an external location, the external location gets marked as changed and it may be merged.

The following types of changes will get merged automatically:

- Changing the name, external memory address, or data type for an external location as long as the latest version didn't change the same attribute of the external location.
- Changing from an external label to an external function as long as the latest version didn't set the data type for the same external label or remove the label.
- Changing from an external function to a label or removing an external function as long as the latest version didn't change the external function.
- Setting the data type on an external label as long as the latest version didn't also set the data type or change the label into an external function.
- Changing an external function as long as the latest version also didn't make changes to the function.
- Conflicts could be:
  - another variable at the same offsets on the stack frame
  - another register variable with the same named register and first use address.

You must resolve all [external location conflicts](#) before the merge process can continue with the next program element, Functions.

#### *Functions*

If you make any change to the functions defined within the program, such as creating, updating, or removing a function, the entry point of the function gets marked as changed and the function may be merged.

The following types of changes will get merged automatically:

- Adding a function with a body that doesn't overlap any function body in the latest version.
- Removing a function as long as the latest version did not change this function or its stack frame.
- Making changes to the function name, return type, parameters, parameter offset, return type offset, etc. as long as the latest version didn't also change the same part of the function.
- If a variable name conflicts with a variable or symbol of the same name in the function's [namespace](#) of the latest version, that variable will automatically be renamed to a conflict name. This eliminates the name conflict.
- Making changes to existing parameters such as the name, data type, or description if the latest version didn't change the same part of the parameter.
- All local variables will be automatically added if they don't conflict with changes to the latest version.
- Conflicts could be:
  - another variable at the same offsets on the stack frame
  - another register variable with the same named register and first use address.
  - another symbol with the same name in the function's namespace
  - the name, datatype, or comment for a local variable is changed to different values in your version and the latest version.

You must resolve all [function conflicts](#) before the merge process can continue with the next program element, Symbols.

#### *Symbols*

Whenever you add, remove, or rename a symbol, the program gets marked as changed. It also gets marked if you change the primary symbol at an address or if you make an address an external entry point. The symbol phase of the Listing merge will merge labels, namespace symbols, class symbols, and external symbols. Symbols associated with functions, function parameters, and function local variables have already been resolved by the function phase of the Listing merge.

The following types of changes will get merged automatically:

- Any symbol you removed will automatically be removed if the latest version didn't change that symbol.
- Any symbol you added will automatically be added if latest version did not create the same named symbol at that address with a different parent namespace.
- If each program has a symbol named the same within a namespace but at a different address. These are resolved automatically by merging your symbol using a [conflict](#) name. In other words, your symbol will have a .conflict suffix added to its name when it is merged. You are notified at the end of the symbol merge phase of any symbols that were renamed to avoid a conflict.
- Any symbol you renamed will automatically get renamed if the latest version didn't rename that symbol to a different name.
- If you change which symbol is primary, it will be automatically set to primary unless the latest version set a different symbol to primary.
- Entry points that you added or removed will be added or removed automatically.

You must resolve all [symbol conflicts](#) before the merge process can continue with the next Listing merge phase, the Address Based Listing Merge.

#### *Address Based Listing Merge*

During this phase of the Listing merge each of the address based listing elements (the equates, user defined properties, references, bookmarks, and comments) will get merged. First an auto merge will occur for each of these elements. When all of the auto merges are complete, conflict resolution will proceed in address order for each address that has a conflict for any of the address based listing elements. At each address with a conflict you will have to resolve all of the conflicts (equates, user defined properties, references, bookmarks, and comments) before proceeding to the next address with conflicts. The following sub-sections describe the auto merge for each of the address based listing elements.

##### *Equates*

An equate associates a display name with a scalar operand or sub-operand at a particular address in the program. Whenever you create, rename, or remove an equate the program gets marked as changed. An equate conflict can arise between your version and the latest version. The following types of equate changes will get merged automatically:

- If an equate was set on a scalar and the latest version didn't set an equate on that scalar, the equate will be added automatically.
- If an equate's associated value differs between the two versions then there is an equate conflict that will be resolved automatically by changing your version's equate name to a conflict name by appending ".conflict" to your equate name.
- If your version's scalar had its associated equate removed, it will get automatically removed if the latest version didn't change the named equate for that scalar.
- If your version's scalar had its associated equate name changed, it will get automatically renamed if the latest version didn't remove or rename it differently.

If an equate is changed for a scalar operand or sub-operand differently in the latest and your versions, then you must resolve the conflict.

The [equate conflicts](#) will need to be resolved after all of the address based auto merges complete. After the Equates auto merge completes, an auto merge will begin for User Defined Properties.

##### *User Defined Properties*

A user defined property change gets marked in the program whenever the named property gets added, removed or changed at an address. An example property is the [Space](#) property created by the [Format](#) plugin.

A user defined property change will get merged automatically if the latest version didn't change the same named property at the same address in the program. If the named property was changed at an address in your version of the program and in the latest version then a user defined property conflict exists.

The [user defined property conflicts](#) will need to be resolved after all of the address based auto merges complete. After the User Defined Properties auto merge completes, an auto merge will begin for References.

##### *References*

References consist of memory references, variable (stack and register) references, and external references. Changes to these include adding references, changing references, and removing references. The references in your checked out version and the latest version are compared for each of the mnemonic and operands at an address to determine whether a conflict exists or the changes can be automatically merged.

The following types of changes will get merged automatically:

- If you added, changed, or removed a reference for a mnemonic or operand at an address and the latest version didn't change the references.
- If your version and the latest version only have memory references for the mnemonic or operand.
- If your reference is a stack variable reference and the latest version has a stack reference to the same stack frame offset or no references.
- If your reference is a register variable reference and the latest version has a register reference to the same register or no references.
- If your reference is an external reference and the latest version has the same external reference or no references.

If your checked out version and the latest version both have changes to the references for a mnemonic or operand, the references are in conflict whenever your version and the latest version have references that are different types (for example, one has memory references and the other has an external reference).

The [reference conflicts](#) will need to be resolved after all of the address based auto merges complete. After the References auto merge completes, an auto merge will begin for Bookmarks.

### **Bookmarks**

Bookmarks allow you to associate a description with an address in the program. The bookmark has a type and category associated with it. Bookmarks that the user has entered via the GUI are **Note** type bookmarks. Ghidra plugins can also add bookmarks of various other types. For example **Auto Analysis** adds **Analysis** type bookmarks. Only one bookmark of a particular type and category can exist at an address. However, only one **Note** type of bookmark can exist at an address regardless of the category.

Changes to bookmarks include adding a new bookmark, removing a bookmark, changing the description for a bookmark, and changing the category for a note bookmark.

Any changes to the bookmarks will get automatically merged as long as your changes and the latest versions changes do not result in one of the following conflict situations:

- your version and the latest version added or changed a **Note** bookmark resulting in a different category or description for the two versions.
- your version and the latest version added or changed the same bookmark type and category resulting in a different description for the two versions.

The **bookmark conflicts** will need to be resolved after all of the address based auto merges complete. After the Bookmarks auto merge completes, an auto merge will begin for Comments.

### **Comments**

If you make a change to a comment (Plate, Pre, End-of-line, Repeatable, or Post) and the latest version did not have any changes to that type of comment at the same address, the change is automatically accepted. Comment changes include adding, removing, or updating the comment. If the latest version has a change to the same comment then the conflict will need to be resolved manually.

The **comment conflicts** will need to be resolved after all of the address based auto merges complete. After the Comments auto merge completes, the conflict merge will begin for address based listing elements (the equates, user defined properties, references, bookmarks, and comments) at each of the addresses with a conflict.

After all address based conflicts are resolved, the auto merge process continues with the next program element, External Program Names.

### **External Program Names**

Each external reference has an associated external program name. The external name provides an association to the external program. Changing an external program name is marked as a change and may be merged. Changes include adding or removing an external program name, and changing the Ghidra program associated with an external program name. If the change doesn't conflict with a change in the latest version then it will be merged automatically.

A conflict occurs whenever your version removes an external program name and the latest version changes the associated Ghidra program or vice versa. A conflict can also occur if the latest version and your version both change the Ghidra program associated with a particular external program name.

You must resolve all **external program name conflicts** before the merge process can continue with the next program element, Property Lists.

### **Property Lists**

**Property lists** are lists of options that are saved as part of the program. You can view and edit these properties through the **Program Options Dialog**. A property will be merged automatically if it is a new property or it was not changed in the latest version. A **conflict** results when you change a property that was either removed or changed in the latest version.

### **Resolving Conflicts**

For each program element, the **Conflict Resolution Merge** is the second of the two merges which is performed. The first time a conflict is encountered, the merge process displays a Merge Tool. Use this tool to resolve each conflict one at a time. To resolve any conflict, you must choose a radio button or check box depending on the type of conflict. Most conflicts require you to select from the following:

- Latest version (changes from the Latest version that is checked into the repository)
- Checked Out version (your changes)
- Original version (values from the file when you originally checked it out)

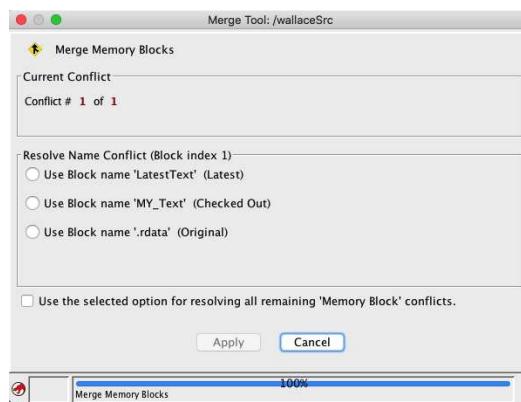
However, there may be other conflict options that let you choose to add or rename the type of item that is in conflict in addition to keeping the one that was checked in the Latest version in the repository. Some conflicts, such as comment conflicts, allow you to check one or both check boxes for the comment conflict. This allows you to keep the Latest comment, your comment, or a combined copy of both the Latest and your (Checked Out) comments.

### **Memory**

All memory conflicts that remain after the **auto\_merge** must be resolved.

The only conflicts on a memory block will be changes to the name, permissions, or comments. A conflict occurs when you change the name of a memory block and another user has changed the name of the same memory block. Similarly, if you changed the permissions or comments on a memory block, and another user has changed the permissions or comments on the same memory block you will have to select which changes to keep.

The image below is the panel that is shown when there is a conflict on the block name. In this sample image, the user who created the latest version had changed the block name to "LatestText" (radio button for *Latest*). You changed the block name to "MY\_Text" (radio button for *Checked Out*). The block name in the original program (the version that you had originally checked out) is shown as ".text" (radio button for *Original*).



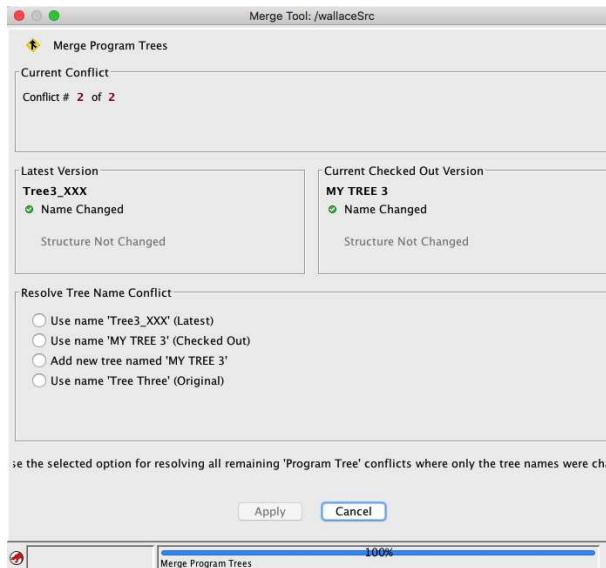
To resolve the conflict, you must select a radio button. After you select a radio button, the **Apply** button becomes enabled. Click on the **Apply** button to continue the merge process. In this example, there was only one conflict in the Memory merge, as indicated by the **Current Conflict** area of the panel. The progress bar on the lower right corner of the Merge Tool indicates progress only for the Memory Block merge, not the entire merge process.

The merge process continues with Program Trees.

### **ProgramTrees**

All program tree conflicts that remain after the **auto\_merge** must be resolved.

If you make any change to a **program tree**, it is marked as changed and may be merged. There is no finer granularity than detecting that the tree changed, so the merge is going to do "all or nothing" on the program tree. For example, if you change the tree (add a folder, rename a fragment, etc.) and another user changed the same tree, you will get a choice to keep your changes, keep the changes from the latest, keep the tree from the original program that you checked out, or rename your tree to a new name. The last option to rename your tree allows you to create your tree in the program that results in the latest version, and not lose anyone else's changes. The image below shows this scenario:



The conflict resolution panel shows what kind of change it is, name change versus a structure change; the checkbox indicates the change in *Latest Version* and *Current Checked Out Version*. In this example, another user changed the name of a program tree to "Tree3\_XXX" (*Latest*). You changed the name of the same program tree to "MY TREE 3" (*Checked Out*). The original name of the tree was "Tree Three" (*Original*). The *Resolve Tree Name Conflict* area of the panel shows radio buttons for each option:

- Use name "Tree3\_XXX" (*Latest*) – select this option to lose your changes and keep what is in the latest version
- Use name "MY TREE 3" (*Checked Out*) – select this option to rename the tree in the program that will become the latest version; the tree is renamed to "MY TREE 3"
- Add new tree named "MY TREE 3" – select this option to create a new tree named "MY TREE 3" in the program that will become the latest version; changes by other users are retained
- Use name "Tree Three" (*Original*) – select this option to lose your changes; "Tree3\_XXX" will be renamed to "Tree Three" which is the name from the program that you had originally checked out

Select an option to resolve the conflict, then click on **Apply**. In this example, the next conflict will be displayed that must be resolved. The progress bar on the lower right corner of the Merge Tool indicates progress only for the Program Tree merge, not the entire merge process.

After all program tree conflicts are resolved, the merge process continues with data types and categories.

#### Data Types and Categories

All data type or category conflicts that remain after the [auto\\_merge](#) must be resolved.

If you make a change (rename, edit contents, delete) to a data type, and the same *type* of change was made in the latest version, then a conflict arises that you must resolve. The merge process for data types and categories handles these conflicts one at a time. The *Current Conflict* area of the panel shows:

*Conflict # x of n*

where *x* is the current conflict number, and *n* is the total number of data type and category conflicts that must be resolved. Data type conflicts are resolved first.

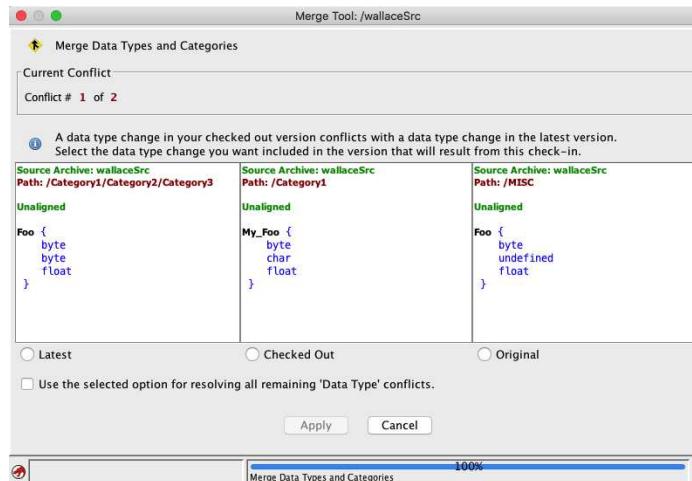
The image below shows the scenario where you:

```
renamed Foo → My_Foo
moved /MISC → /Category1/Category2/Category3
changed undefined → char
```

In the latest version:

```
moved /MISC → /Category1/Category2/Category3
changed undefined → byte
```

The conflict arises because in both versions the file was **moved** and also because the second component was changed to a different data type. The rename of Foo to My\_Foo doesn't cause a conflict. If you had only **renamed** the data type (without **moving** it or **changing** the undefined), there would not have been a conflict.



These are the options to resolve the conflict:

- *Latest* – select this option to keep what is in the latest version (lose your changes)
- *Checked Out* – select this option to apply your changes; (changes in the latest version are lost)
- *Original* – select this option to apply what was in the program that you had originally checked out (your changes and those in the latest version are lost)

Select a radio button to resolve the conflict.

In the Merge Tool window above, the title indicates the project, program and new version number that will result from the merge. Version 5 of "helloProgram" in the "SampleProject" will be created by

the merge. The source archive is indicated for each data type in the conflict window. The "Foo" data type was added to "helloProgram" from the "MyTestArchive" data type archive and is still associated with it in each program version (Latest, Checked Out and Original.)

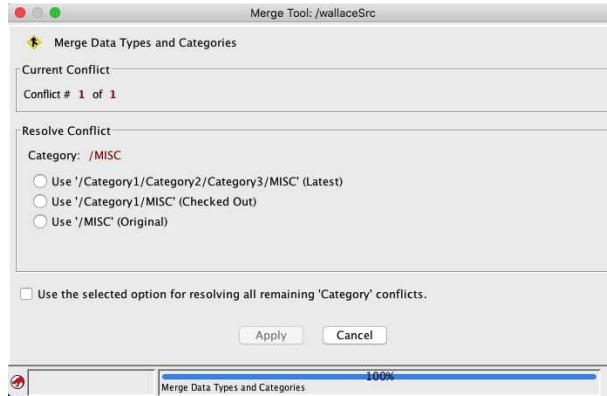
Category conflicts are resolved after data type conflicts. If you make a category change (rename, move, delete, restructure) in your version of the file, and the same type of change was made in the latest version of the file, then a conflict arises that you must resolve.

The image below shows the scenario where you:

**moved /MISC ➔ /Category1/MISC**

In the latest version:

**moved /MISC ➔ /Category1/Category2/Category3/MISC**



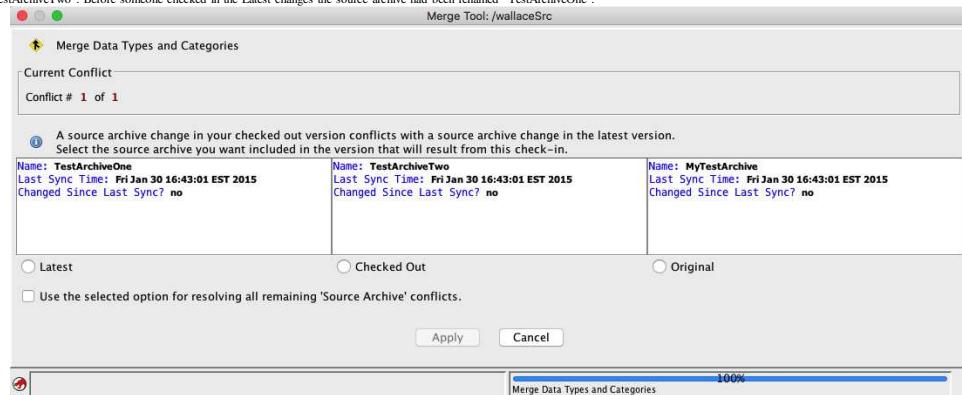
The options to resolve the conflict are:

- *Latest* –select this option to keep what is in the latest version (lose your changes)
- *Checked Out* –select this option to apply your changes
- *Original* –select this option to apply what was in the program that you had originally checked out (your changes and those in the latest version are lost)

Select a radio button to resolve the conflict.

After all data type and category conflicts are resolved, the merge process continues with Program Context Registers.

For data types a source archive conflict can occur if the name of an associated source archive changes to a different name in each of the two programs being merged. In the following, the "helloProgram" in the "SampleProject" (as indicated by the Merge Tool title) has a data type that originated from "MyTestArchive". Before you checked in your last changes to the program, the source data type archive was renamed to "TestArchiveTwo". Before someone checked in the Latest changes the source archive had been renamed "TestArchiveOne".



Name: This is the name of the source archive in that version of the program.

Last Sync Time: This is when the program was last synchronized with the source archive.

Changed Since Last Sync? yes, means that one of the program's data types associated with the source archive has been changed since the last time it was synchronized.

To resolve the conflict, select the radio button associated with the current name of the archive.

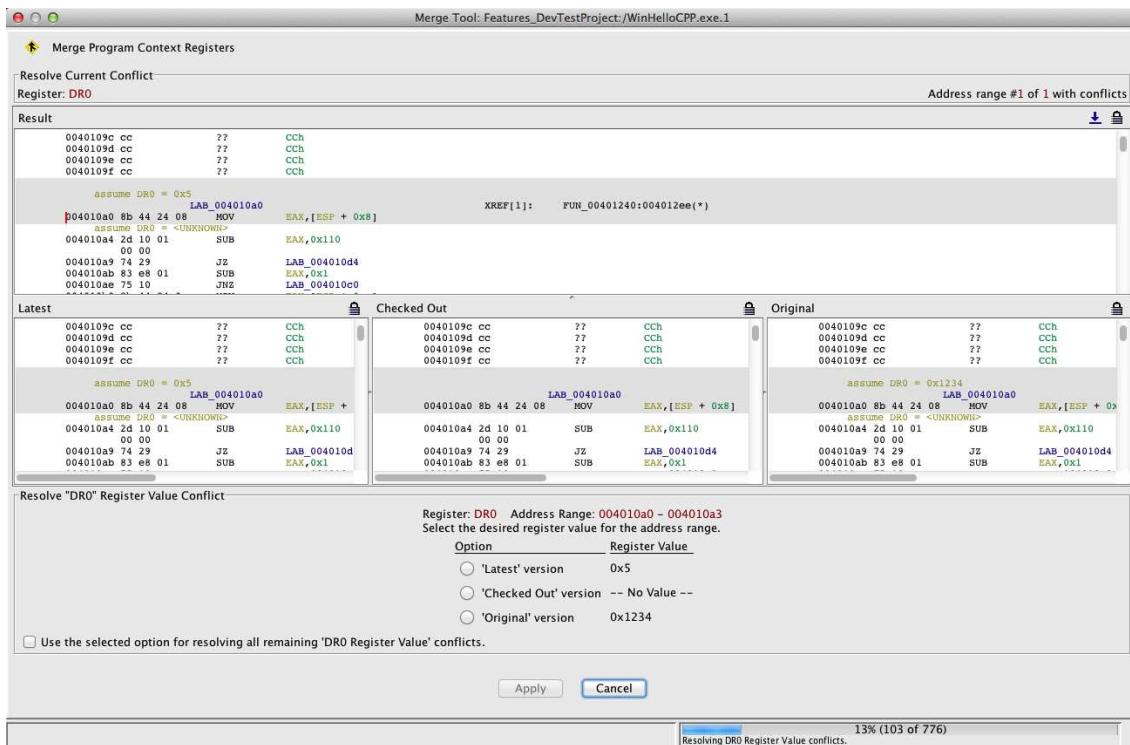
#### Program Context Registers

All register value conflicts that remain after the [auto\\_merge](#) must be resolved.

Register values can be set on an individual address or a range of addresses. When a register's value at an address or address range is changed in both the latest version and your checked out version and the two values are different, this results in a conflict. For each conflict, the merge tool will display the address or address ranges where the two versions conflict.

The conflict window will present you with each register that has value conflicts. For each of these registers you will have to resolve each of the address ranges that has a conflict. The conflict information area at the top of the window indicates the register name. It also indicates which address range you are resolving of all the ranges in conflict for this register.

The following image illustrates a conflict due to the DIRECTION register's value being changed in the latest version and cleared in your checked out version of the program. The address range affected by this particular conflict is from address 01002329 to 0100233b. This is indicated in the conflict resolution area below the program listings. The affected addresses are also indicated by the gray background in the program listings.



**💡** The scrolled listings allow you to see the code units in the different program versions, which may help determine the correct register value to choose. The layout of the Merge Program Context window is very similar to the [Merge Listing](#) window.

The options to resolve the conflict are:

- **Latest** – select this option to keep what is in the latest version (lose your changes)
- **Checked Out** – select this option to apply your changes (overwrites change in the latest version)
- **Original** – select this option to restore the original register values (your changes and those in the latest version are lost)

Select a radio button to resolve the conflict and then press the **Apply** button to continue merging.

After all register conflicts are resolved, the merge process continues with the Listing.

### [Listing](#)

Various parts of the program are merged in the Listing merge phase. These include the bytes and code units, functions, symbols, user defined properties, references, bookmarks, and comments. Listing conflicts must be resolved in the order they are presented before moving to the next conflict and eventually completing the merge process.

#### [Listing Conflict Window](#)

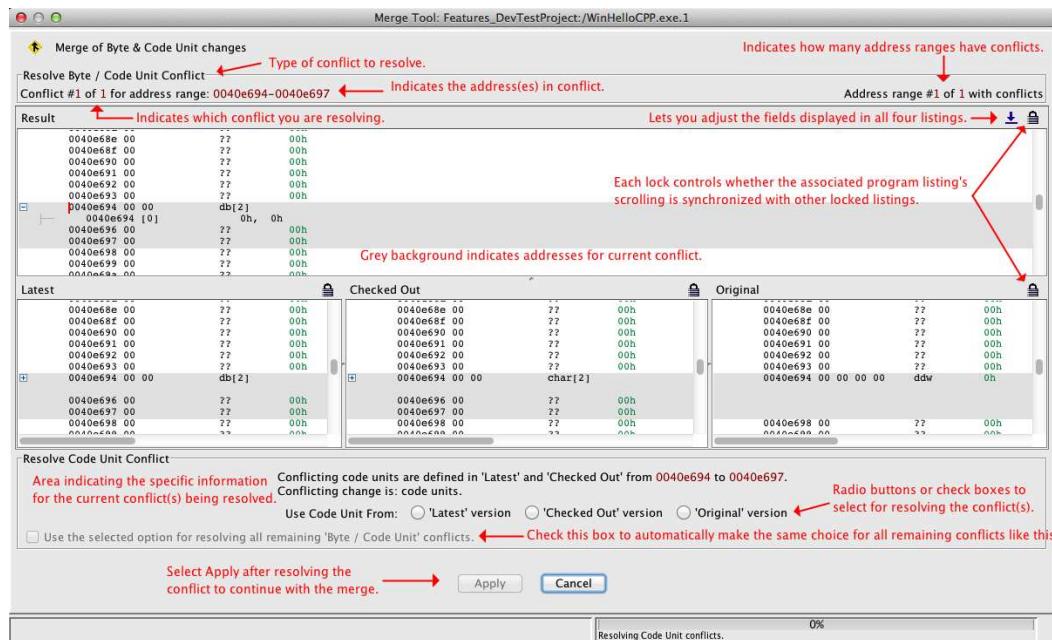
Listing conflicts are presented in a Merge Listing Window. The Merge Listing Window displays four program listings:

Result	The new program version that will result from this merge.
Latest	The latest checked-in version of the program.
Checked Out	Your checked out version of the program to be merged.
Original	The version of the program that you originally checked out.

The following shows the Merge Listing Window and describes (in red text) some of the features of the conflict window.

The **💡 Merge Listing** on the top left side of the window indicates that you are in the Listing phase of the merge. The **Resolve Byte / Code Unit Conflict** area title indicates you are in the Code Unit phase of Listing Merge. This area provides the following information:

- the addresses for the conflict currently being resolved
- number of conflicts at the indicated address(es)
- current conflict
- how many sets of conflicts there are to resolve in this phase of the Listing merge



Each of the listings in the above image displays the associated program (Result, Latest, etc.). This allows you to look at each of the programs involved in the merge whenever you are trying to resolve a Listing conflict. You can use the expand icon in the Result listing's toolbar to adjust the fields that are displayed in all of the listings like in the Code Browser. The *GoTo* is also available for navigating within the listings.

#### Scroll Locks for the Listings

Initially, the four programs scroll together. Each listing has a lock icon associated with it. When the lock icon is displayed for a listing, scrolling that listing or navigating within that listing will also cause all other locked listings to scroll to the same address. When a listing is displaying the unlocked icon , it will not scroll when the other listings scroll. Likewise, scrolling it will not cause any of the other listings to scroll.

#### Code Units

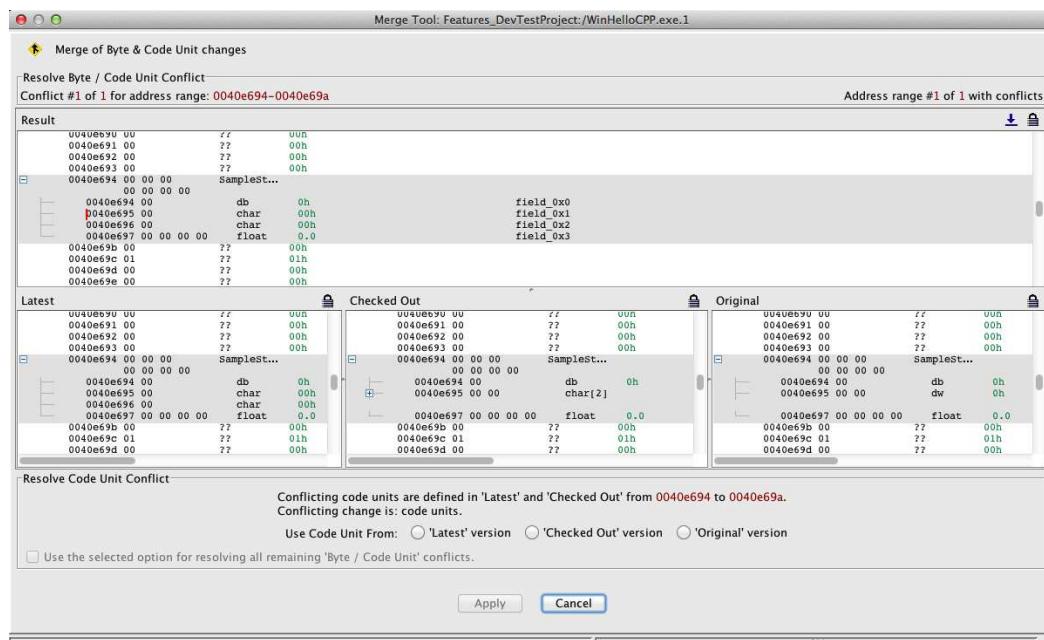
The first type of Listing conflict you may encounter is a byte or code unit conflict. All byte and code unit conflicts that remain after the [auto\\_merge](#) must be resolved.

The following types of changes will result in code unit conflicts that need resolving.

- The same bytes had their values changed to different values in the latest version and your checked out version.
- A byte was changed in one version where a code unit was changed in the other version.
- The code units were changed in your version and the latest version but not to the same new code units.
- For example, latest created an instruction and your version created data at an address. Another example would be that a different structure was applied in your version and the latest version at the same address.
- An instruction was cleared in one version and an equate was set in the other version.
- An instruction had a flow override applied in one version and a reference was added in the other version.

If an instruction change conflicts with a reference change, the references can be viewed by right clicking on an instruction in one of the four listings (Result, Latest, Checked Out, Original) and choosing the [View Instruction Details...](#) action. This will pop up a dialog indicating any fallback or flow override and the references for that instruction.

**Example:** The following image shows a code unit conflict that occurred because you changed the second component of the *SampleStructure* from a word to an array of two characters and the word was changed in the latest version to two separate characters.



Notice that the listings allow you to open and close the structure in the same way that you can in the CodeBrowser by clicking on the + in the listing. The addresses with the gray background in the listing indicate the code unit(s) that you are currently trying to resolve.

Note that you will have already resolved all [data type conflicts](#) by this time; this conflict is due to different data types being applied at the same address. You are resolving which *data type to apply* at the address.

In this case the options to resolve the conflict are:

- *'Latest'* version –select this option to keep the structure containing two separate characters as in the latest version (lose your changes)
- *'Checked Out'* version –select this option to apply your structure containing the 2 character array (overwrites change in the latest version)
- *'Original'* version –select the option to restore the original structure containing the word (your changes and those in the latest version are lost)

Depending on which option you choose, you may end up with a "SampleStructure.conflict" as the data type name.

Select a radio button to resolve the conflict and press the **Apply** button to continue with the merge process.

After all byte and code unit conflicts are resolved, the merge process continues with any Function conflicts in the Listing.

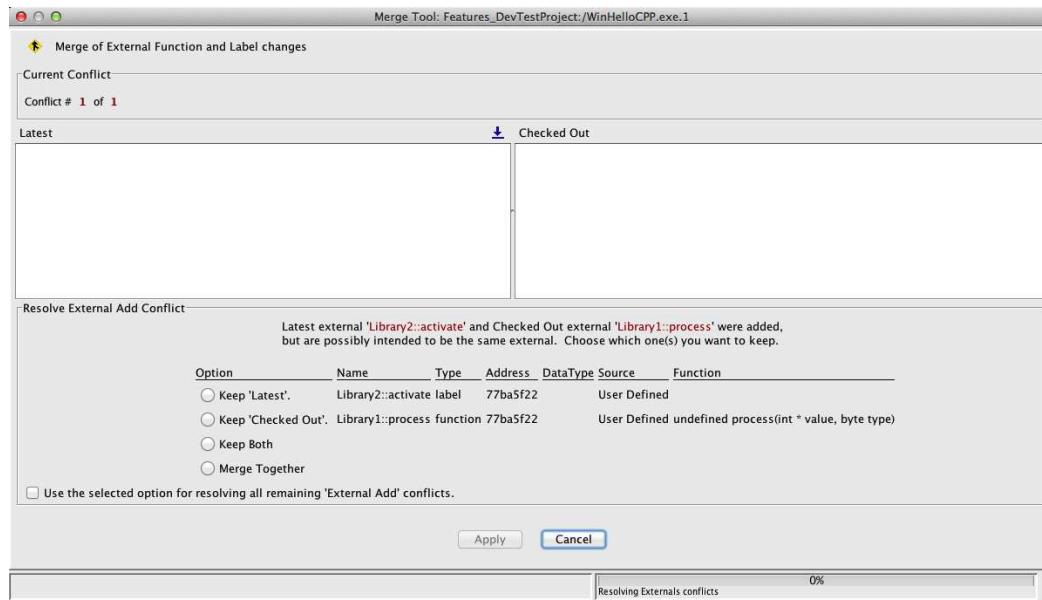
#### *External Functions and Labels*

All external location conflicts that remain after the [auto\\_merge](#) must be resolved.

The following types of changes to external locations will result in conflicts that need resolving between your checked out version and the latest version in the repository.

- If you add an external function or label and it appears to be the same external as one that was added in the latest version. (i.e. It has the same name or is referenced from the same code.)
  - If an existing external function or label is removed in your version and changed in the latest version in any way or vice versa.
  - If you change an external label into an external function and the latest sets the data type on the external label or vice versa.
  - If you change the number of parameters, the parameter storage, or the parameter type (stack versus register) and the latest also changed parameters in any way, then you must select to keep all of your parameters or all of the latest version's parameters. The same is true if the latest version changed the number or type of parameters and you changed parameters in any way.
  - If you change a piece of an external functions information and the latest version changes the same piece of function information to a different value.
  - For example, if one changes the function's name and the other changes its return type, then both changes occur without a conflict. If both change the name, this is a conflict.
  - If you change a piece of stack information and the latest version changes the same piece of stack information to a different value.
- Stack information includes:
- return address offset
  - parameter offset
  - stack purge size
- If you change a piece of parameter information and the latest version changes the same piece of parameter information to a different value.
- Parameter information includes:
- name
  - data type
  - comment
  - offset
  - register

**Example\_1 :** The following image illustrates a possible Add conflict. In this case, an external label was added to the Latest program and an external function was added to the Checked Out program. Both external locations refer to the same external memory address and may be intended as the same external.

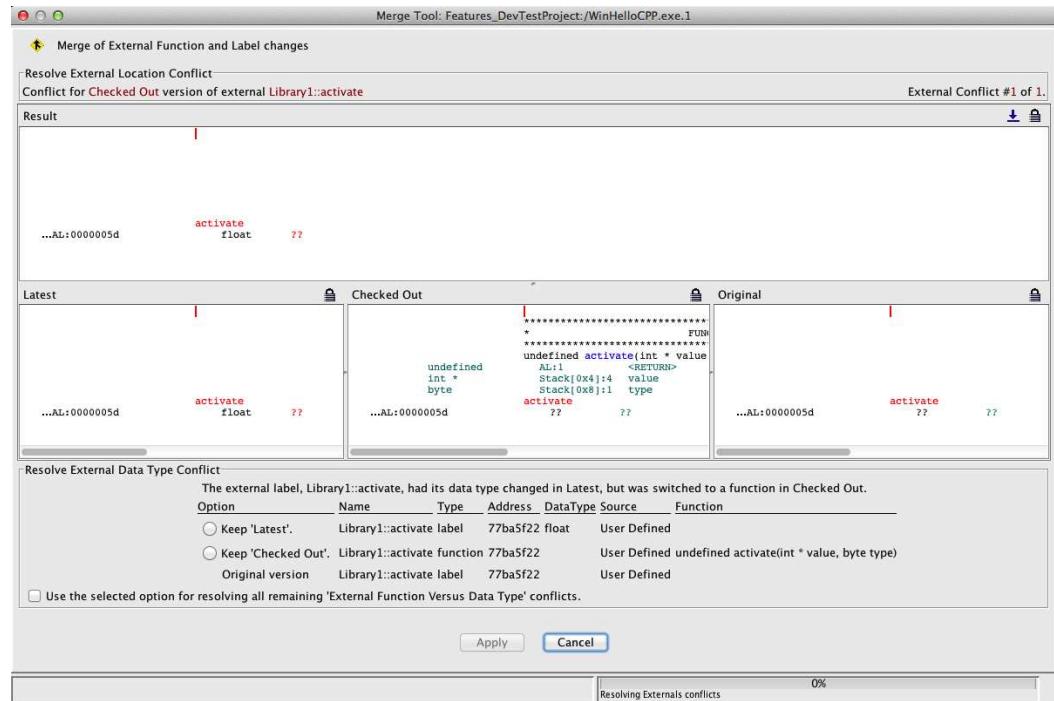


In this case you must select the radio button indicating which external(s) to keep:

- *Keep 'Latest'* will keep the external label.
- *Keep 'Checked Out'* will keep the external function.
- *Keep Both* will keep Both the external label and the external function. If the label and function had the same name, the function would get renamed with a new name containing a conflict extension.
- *Merge Together* will merge the external label and function together. In this case, the result would be an external function, but after pressing the Apply button, you would be prompted with an additional name conflict that you would need to resolve.

Select a radio button and press the **Apply** button to continue with the merge process.

**Example\_2 :** The following image illustrates a conflict due to an external label having its data type set in the latest version while you changed it into a function.

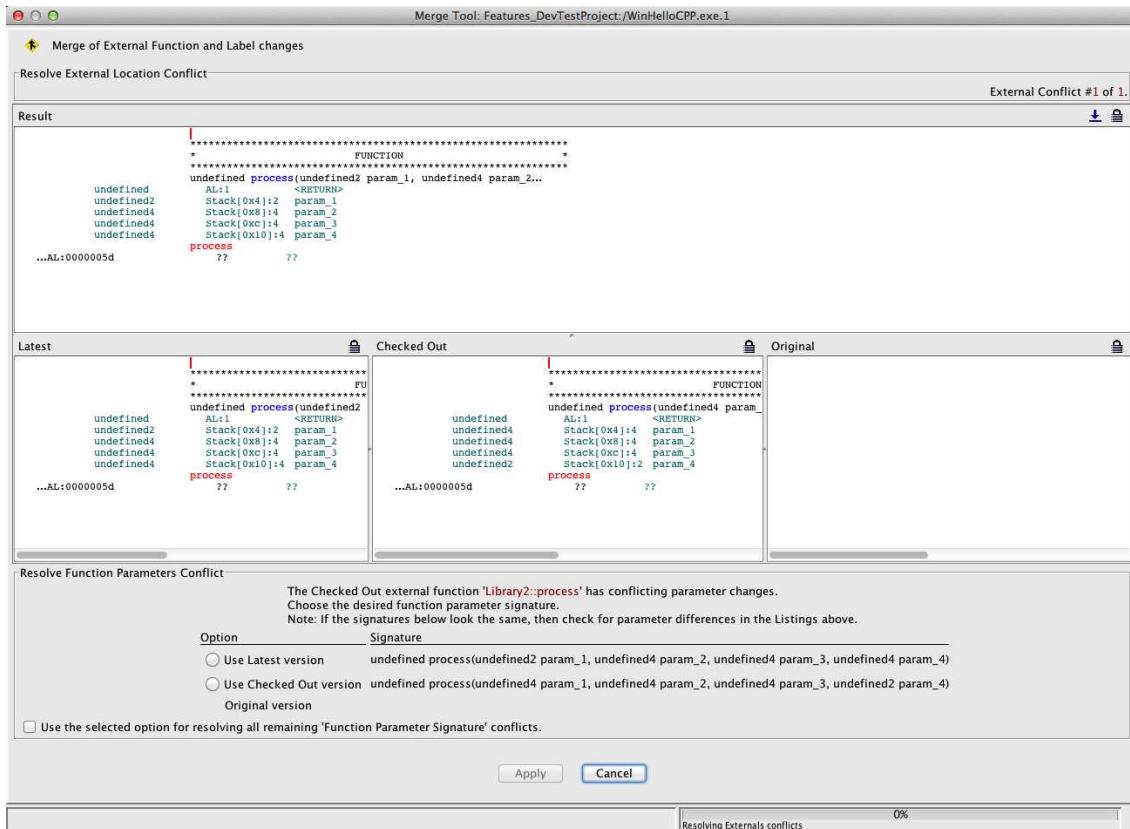


In this case you must select the radio button indicating which change to keep:

- *Keep Latest* will result in the external label with a Float data type and the change to a function is discarded.
  - *Keep Checked Out* will keep the external as a function and the data type change will be discarded.

Select a radio button and press the **Apply** button to continue with the merge process.

**Example\_3 :** The following image illustrates a conflict because you created the same external function as was created in the latest version. However, parameter storage has been defined differently in each version. You must choose which parameter signature and its associated storage should be kept.



In this case the options to resolve the conflict are:

- *Use Latest version*, will use the function parameter storage as it is in the Latest version.
  - *Use Checked Out version* will use the function parameter storage as it is in your Checked Out version.

Select a radio button and press the **Apply** button to continue with the merge process.

After all external label and external function conflicts are resolved, the merge process continues with any regular Function conflicts in the Listing.

#### Functions

All function conflicts that remain after the [auto\\_merge](#) must be resolved.

The following types of changes to functions will result in conflicts that need resolving between your checked out version and the latest version in the repository.

- If you create or change a function causing its body to overlap one or more functions that are defined in the latest version.
  - Functions with bodies that have addresses in common are said to *overlap* if their entry points and bodies are not exactly the same.
  - In this case you must choose which version's function(s) to keep.
- If an existing function is removed in your version and changed in the latest version in any way or vice versa.
- If both versions add a function with the same entry point, but all parts of the function are not the same.
- If you change the number of parameters or the parameter type (stack versus register) and the latest also changed parameters in any way, then you must select to keep all of your parameters or all of the latest version's parameters. The same is true if the latest version changed the number or type of parameters and you changed parameters in any way.
- If you change a piece of function information and the latest version changes the same piece of function information to a different value.

Function information includes:

- name
- body
- return data type

For example, if one changes the function's name and the other changes the return type, then both changes occur without a conflict. If both change the name, this is a conflict.

If you change a piece of stack information and the latest version changes the same piece of stack information to a different value.

Stack information includes:

- return address offset
- parameter offset
- local size
- stack purge size

If you change a piece of parameter information and the latest version changes the same piece of parameter information to a different value.

Parameter information includes:

- name
- data type
- comment
- offset
- register

If you change a piece of local variable information and the latest version changes the same piece of local variable information to a different value.

Local variable information includes:

- name
- data type
- comment
- offset
- register

**Example 1 :** The following image illustrates a function parameter with multiple conflicts. In this case the conflict resolution area below the listings is different than many of the other listing conflict resolution areas. Each row of the table provides you with a separate conflict and a choice that must be made.

Notice in the image that the parameter name and the parameter type are each in conflict. You must choose a radio button on each row to indicate what you want to keep in the version that will get checked in.

Merge Tool: Features\_DevTestProject:/WinHelloCPP.exe.1

Merge of Function changes

Resolve Function Conflict  
Conflict #2 of 2 @ address: 00401130 Address #1 of 2 with conflicts

Result	Checked Out	Original									
<pre>0040112c cc    ??    CCh 0040112d cc    ??    CCh 0040112e cc    ??    CCh 0040112f cc    ??    CCh</pre> <hr/> <pre>***** FUNCTION ***** ***** _cdecl FUN_00401130(undefined4 param_1, int * v...</pre> <pre>pointer _cdecl FUN_00401130(undefined4 param_1, int * v... EAX[4] undefined4 param_1 Stack[0x8]4 value XREF[1]: 0040115C(R) Latest parameter... Stack[0x8]4 param_3 XREF[1]: 00401154(R) Latest parameter... Stack[-0xc]1 local_c XREF[1]: 00401148(R) XREF[1]: 0040114A(*)</pre> <pre>00401130 6a ff PUSH -0x1 00401132 68 ab ae PUSH LAB_0040aea 40 00 00401137 64 al 00 MOV EAX,FS:[0x0]</pre>	<pre>0040112c cc    ??    CCh 0040112d cc    ??    CCh 0040112e cc    ??    CCh 0040112f cc    ??    CCh</pre> <hr/> <pre>***** FUNCTION ***** ***** _cdecl FUN_00401130(undefined4 param_1, int * v...</pre> <pre>pointer _cdecl FUN_00401130(undefined4 param_1, int * v... EAX[4] undefined4 param_1 Stack[0x8]4 value XREF[1]: 0040115C(R) Latest parameter... Stack[0x8]4 param_3 XREF[1]: 00401154(R) Latest parameter... Stack[-0xc]1 local_c XREF[1]: 00401148(R) XREF[1]: 0040114A(*)</pre> <pre>00401130 6a ff PUSH -0x1 00401132 68 ab ae PUSH LAB_0040aea 40 00 00401137 64 al 00 MOV EAX,FS:[0x0]</pre>	<pre>0040112c cc    ??    CCh 0040112d cc    ??    CCh 0040112e cc    ??    CCh 0040112f cc    ??    CCh</pre> <hr/> <pre>***** FUNCTION ***** ***** _cdecl FUN_00401130(undefined4 param_1, int * v...</pre> <pre>void _cdecl FUN_00401130(undefined4 param_1, int * v... Stack[0x8]4 param_1 Stack[0x8]4 param_2 Stack[0x8]4 param_3 Stack[-0xc]1 local_c XREF[1]: 0040115C(R)</pre> <pre>00401130 6a ff PUSH -0x1 00401132 68 ab ae PUSH LAB_0040aea 40 00 00401137 64 al 00 MOV EAX,FS:[0x0]</pre>									
Latest	Checked Out	Original									
<pre>0040112c cc    ??    CCh 0040112d cc    ??    CCh 0040112e cc    ??    CCh 0040112f cc    ??    CCh</pre> <hr/> <pre>***** FUNCTION ***** ***** _cdecl FUN_00401130(undefined4 param_1, int * v...</pre> <pre>pointer _cdecl FUN_00401130(undefined4 param_1, int * v... EAX[4] undefined4 param_1 Stack[0x8]4 value XREF[1]: 0040115C(R) Latest parameter... Stack[0x8]4 type int + Stack[0xc]4 param_3 int undefined1 Stack[-0xc]1 local_c XREF[1]: 00401148(R) XREF[1]: 0040114A(*)</pre> <pre>00401130 6a ff PUSH -0x1 00401132 68 ab ae PUSH LAB_0040aea 40 00 00401137 64 al 00 MOV EAX,FS:[0x0]</pre>	<pre>0040112c cc    ??    CCh 0040112d cc    ??    CCh 0040112e cc    ??    CCh 0040112f cc    ??    CCh</pre> <hr/> <pre>***** FUNCTION ***** ***** _cdecl FUN_00401130(undefined4 param_1, int * v...</pre> <pre>float _cdecl FUN_00401130(undefined4 param_1, int * v... Stack[0x4]4 param_1 Stack[0x4]4 type Stack[0xc]4 param_3 Stack[-0xc]1 local_c XREF[1]: 0040115C(R)</pre> <pre>00401130 6a ff PUSH -0x1 00401132 68 ab ae PUSH LAB_0040aea 40 00 00401137 64 al 00 MOV EAX,FS:[0x0]</pre>	<pre>***** FUNCTION ***** ***** _cdecl FUN_00401130(undefined4 param_1, int * v...</pre> <pre>void _cdecl FUN_00401130(undefined4 param_1, int * v... Stack[0x8]4 param_1 Stack[0x8]4 param_2 Stack[0x8]4 param_3 Stack[-0xc]1 local_c XREF[1]: 0040115C(R)</pre> <pre>00401130 6a ff PUSH -0x1 00401132 68 ab ae PUSH LAB_0040aea 40 00 00401137 64 al 00 MOV EAX,FS:[0x0]</pre>									
Resolve Function Parameter Conflict											
<p>Function: FUN_00401130 EntryPoint: 00401130 Parameter #2 Storage: Stack[0x8]:4</p> <table border="1"> <thead> <tr> <th>Conflict</th> <th>Latest</th> <th>Checked Out</th> </tr> </thead> <tbody> <tr> <td><input type="radio"/> Parameter Name</td> <td><input type="radio"/> value</td> <td><input type="radio"/> type</td> </tr> <tr> <td><input type="radio"/> Parameter Comment</td> <td><input type="radio"/> Latest parameter 2 comment.</td> <td><input type="radio"/> My parameter 2 comment.</td> </tr> </tbody> </table> <p><input type="checkbox"/> Use the selected option for resolving all remaining 'Function Parameter Info' conflicts.</p>			Conflict	Latest	Checked Out	<input type="radio"/> Parameter Name	<input type="radio"/> value	<input type="radio"/> type	<input type="radio"/> Parameter Comment	<input type="radio"/> Latest parameter 2 comment.	<input type="radio"/> My parameter 2 comment.
Conflict	Latest	Checked Out									
<input type="radio"/> Parameter Name	<input type="radio"/> value	<input type="radio"/> type									
<input type="radio"/> Parameter Comment	<input type="radio"/> Latest parameter 2 comment.	<input type="radio"/> My parameter 2 comment.									
<input type="button" value="Apply"/> <input type="button" value="Cancel"/>											

0%

Resolving Function conflicts.

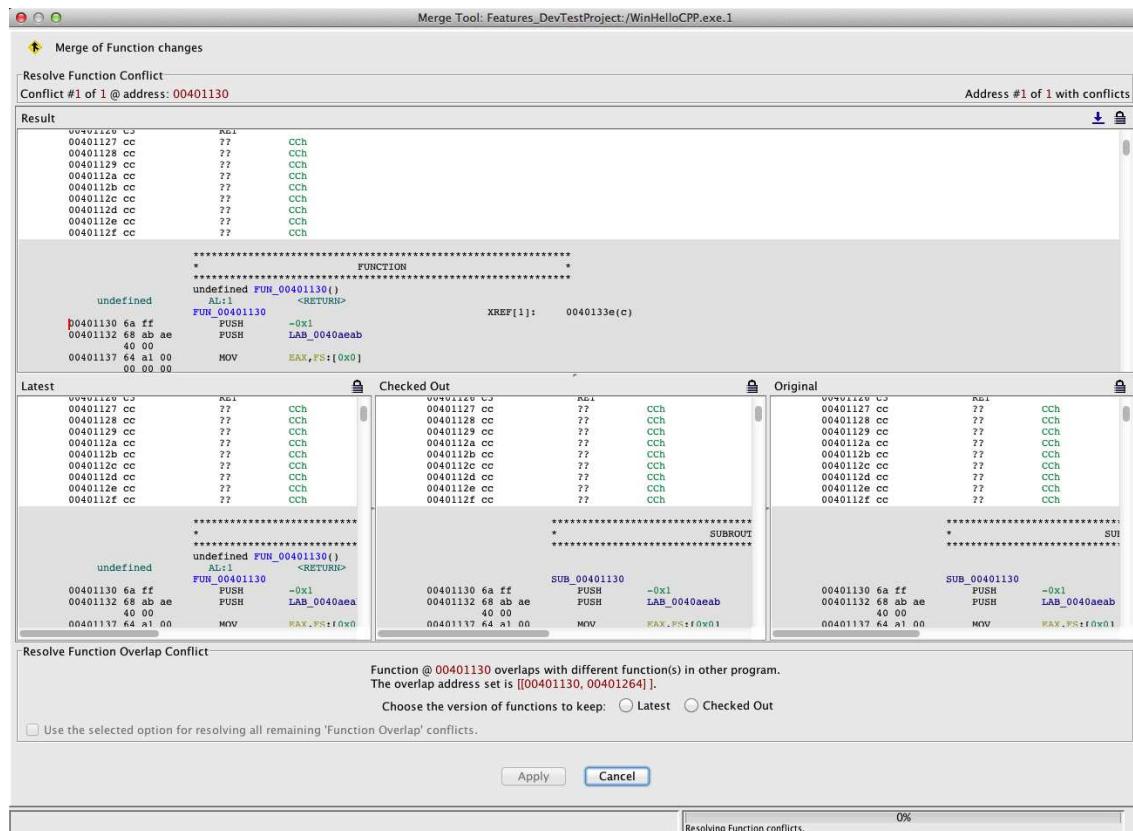
The function FUN\_01002b44 has two conflicts for its second parameter.  
In this case the options to resolve the conflict are:

- Conflict column –indicates what specific part of the function is in conflict.
- Latest column –select the item from this column to keep the latest version's change (lose your changes).
- Checked Out column –select the item from this column to keep your change (overwrites change in the latest version).

Select a radio button on each row to resolve the conflicts and press the **Apply** button to continue with the merge process.

**Example 2 :** The following image illustrates a conflict due to overlapping function bodies. In this case you created a function at address 01001984 and the latest version created a function at address 01001979. The two functions don't have the same entry point, but their bodies overlap. Since function bodies can't overlap, you must choose which version's functions should be kept. The **Resolve**

*Function Overlap Conflict* area indicates the address range(s) that are in conflict. This is also indicated by the gray background in the listings.

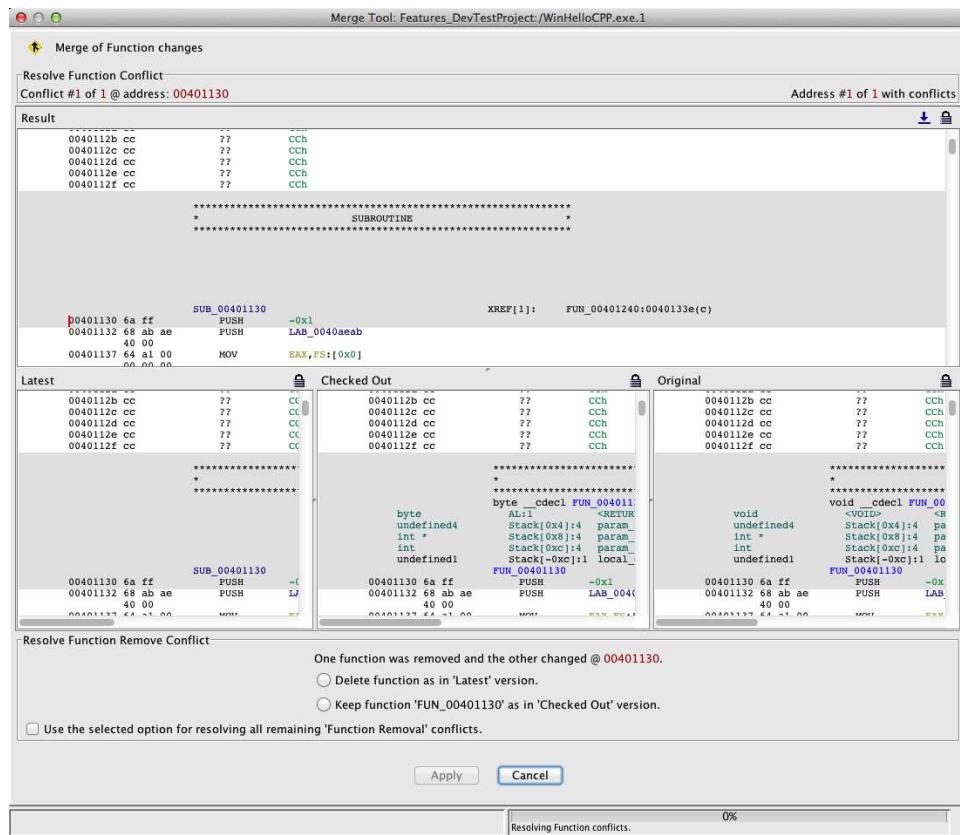


In this case you must select the radio button indicating which version's functions to keep:

- **Latest** –select the radio button in front of Latest to keep the latest version's functions for the indicated address ranges (lose your functions).
- **Checked Out** –select the radio button in front of Checked Out to keep the your functions for the indicated address ranges (overwrites change in the latest version).

Select a radio button and press the **Apply** button to continue with the merge process.

**Example\_3 :** The following image illustrates a conflict because you changed the return type and the function was deleted in the latest version. You must choose whether to keep the function or remove it.



In this case the options to resolve the conflict are:

- Delete function as in 'Latest' version. –select this option to remove the function as in the latest version (lose your changes)
- Keep function 'FUN\_00401130' as in 'Checked Out' version. –select this option to keep the function and your changes to it (overwrites change in the latest version)

Select a radio button and press the **Apply** button to continue with the merge process.

After all function conflicts are resolved, the merge process continues with any Symbol conflicts in the Listing.

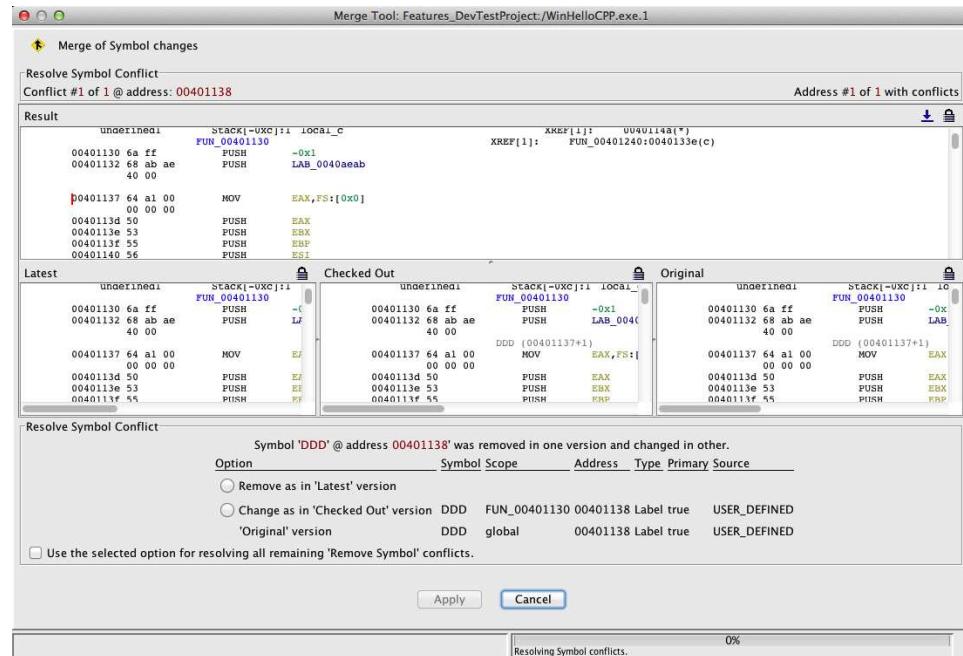
#### Symbols

All symbol conflicts that remain after the `auto_merge` must be resolved.

The symbol phase of the Listing merge will merge labels, `namespace` symbols, class symbols, and external symbols. The following are the types of symbol conflicts that require manual resolution:

- Removed symbol in one version and changed it in other.
- Symbol was renamed differently in each version.
- Same named symbol at the same address in each version, but in different namespaces (symbols with different scope).
- Each version set a different symbol to primary at an address.

**Example 1:** The following image shows a symbol conflict that occurred because you changed the label "DDD" from global scope to function scope (in the FUN\_01004444 function's namespace) and the latest version removed the symbol.



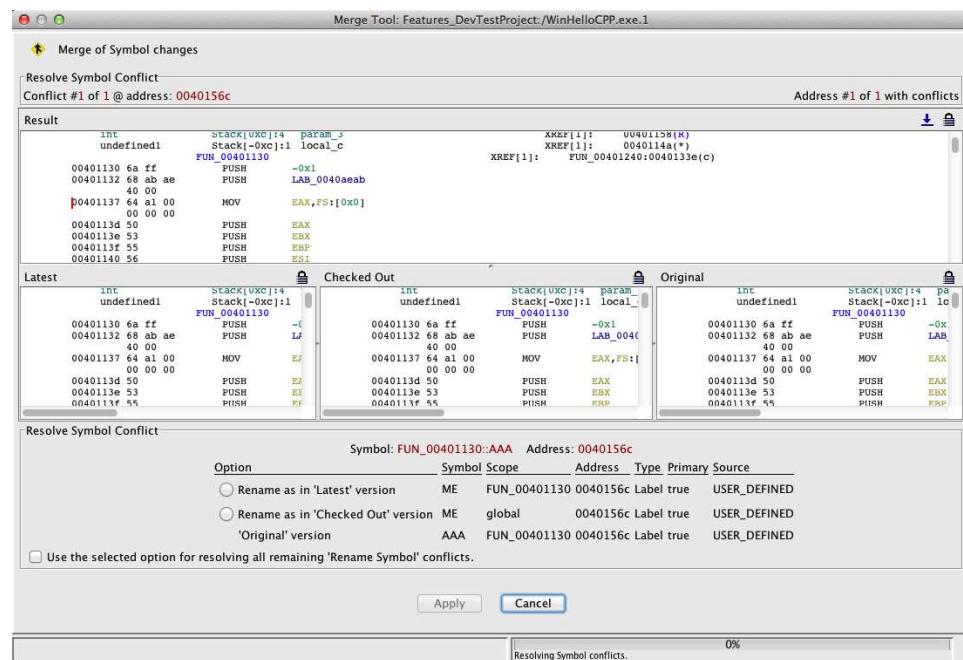
In the **Resolve Symbol Conflict** area you must choose an option to resolve the conflict. Each row shows the information for a different version of the program. The Option column allows you to choose a version change or simply indicates which version's information follows it in the row. The Symbol column gives the symbol name. The Scope gives the namespace containing the symbol. The address indicates the address where the symbol is located. The Type indicates whether the symbol is a label, namespace, class, etc. Primary indicates whether the symbol is the primary symbol at that address in that version.

In this case the options to resolve the conflict are:

- Remove as in 'Latest' version – select this option to leave the label DDD removed (lose your changes)
- Change as in 'Checked Out' version – select this option to change the label DDD to be in the function's scope (overwrites change in the latest version)
- 'Original' version – this is not selectable, but shows the symbol in the original version you checked out.

Select a radio button to resolve the conflict and press the **Apply** button to continue with the merge process.

**Example\_2 :** The following image shows a symbol conflict that occurred because you changed the label "AAA" to "ME" and changed it to global scope, but the latest version only renamed the symbol.

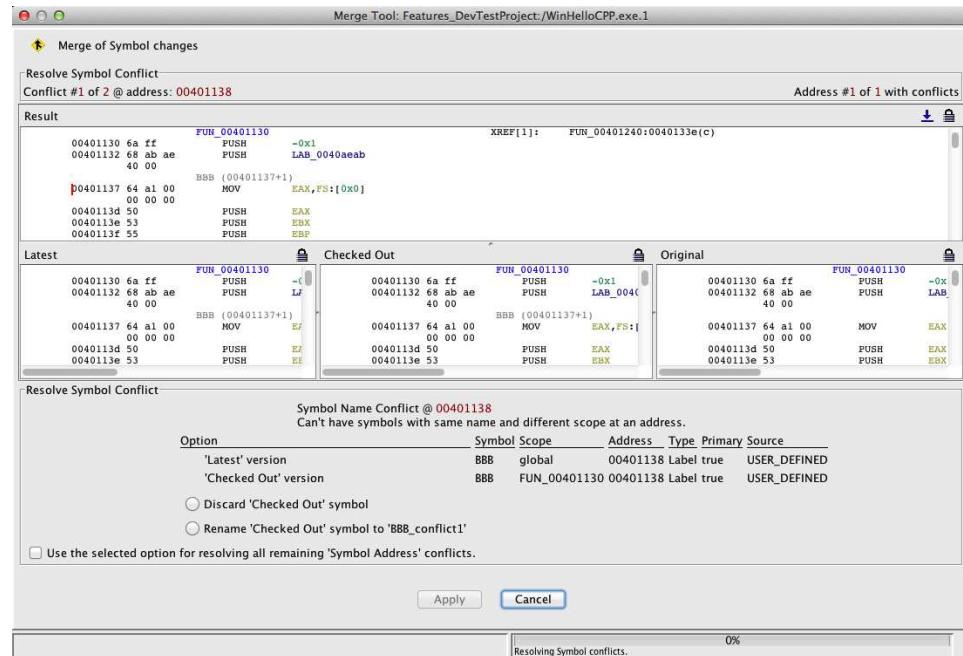


In this case the options to resolve the conflict are:

- Rename as in 'Latest' version – select this option if ME should be in the function's scope (lose your changes)
- Rename as in 'Checked Out' version – select this option if ME should have global scope (overwrites change in the latest version)
- 'Original' version – this is not selectable, but shows the symbol in the original version you checked out.

Select a radio button to resolve the conflict and press the **Apply** button to continue with the merge process.

**Example\_3 :** The following image shows a symbol conflict that occurred because you added the label "Bud" to the function scope (in the FUN\_01004bc0 function's namespace) and the latest version added the same named symbol at the same address, but gave it global scope.



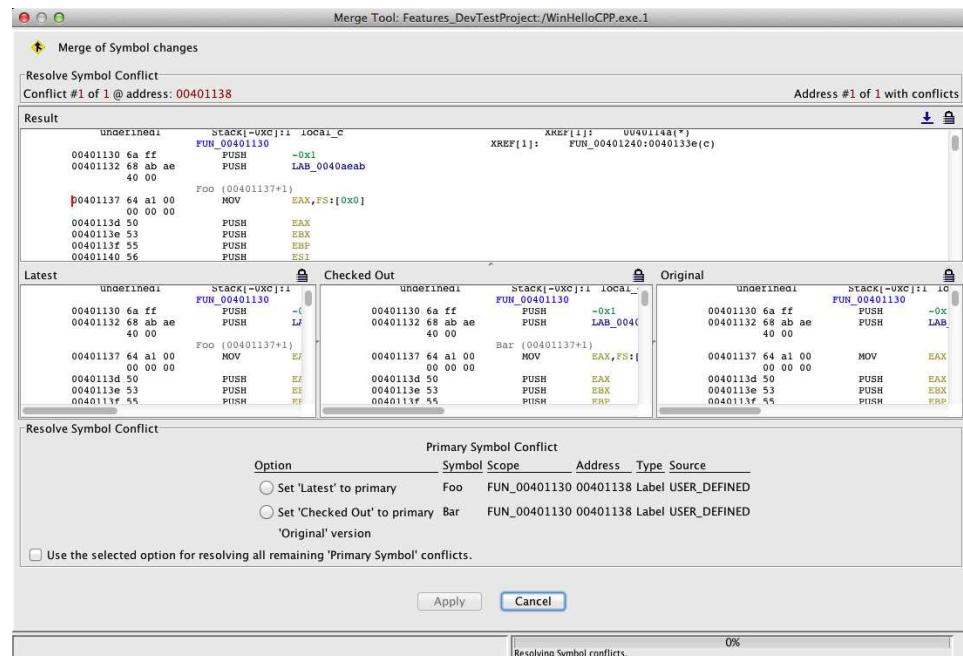
In the *Resolve Symbol Conflict* area you must choose an option to resolve the conflict. Each of the first two rows shows the information for a different version of the program. Below these rows are two radio buttons that let you choose to either discard your symbol or rename it so that its name will not conflict with the one from the latest version.

In this case the options to resolve the conflict are:

- *'Latest' version* – this is not selectable, but shows the symbol in the latest version.
- *'Checked Out' version* – this is not selectable, but shows your symbol.
- *Discard 'Checked Out' symbol* – select this option to not include your symbol (lose your changes).
- *Rename 'Checked Out' symbol to 'Bad\_conflict'* – select this option to rename your symbol to avoid the conflict.

Select a radio button to resolve the conflict and press the **Apply** button to continue with the merge process.

**Example 4 :** The next image shows a symbol conflict about which symbol to make the primary symbol at an address. In the following scenario, you added the label "Bar" and the latest version added the label "Foo" at the same address. Both labels can exist at the same address since they have different names, but they cannot both be the primary symbol. So you must choose which label is the primary one.



In this case the options to resolve the conflict are:

- *Set 'Latest' to primary* – select this option if Foo should be the primary symbol.
- *Set 'Checked Out' to primary* – select this option if Bar should be the primary symbol.
- *'Original' version* – this is not selectable, but shows the symbol in the original version you checked out. In this case there was no label originally.

Notice that both symbol are kept as shown in the Result program's listing.

Select a radio button to resolve the conflict and press the **Apply** button to continue with the merge process.

#### Symbol Merge Information Dialog

At the end of the Symbol merge phase, a *Symbol Merge Information* dialog will appear if any namespaces could not be removed as expected. You might indicate to remove a namespace when it is in conflict with another symbol or it may have been removed by auto merge. However, if another symbol gets placed into that namespace as a result of

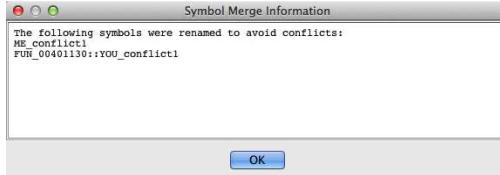
resolving a conflict, the namespace must be retained. The dialog is simply to inform you that this has occurred. The following image illustrates that the namespace called SecondNamespace was not removed since it contained at least one other symbol.



Press the OK button on this dialog to continue with the merge process.

The *Symbol Merge Information* dialog can also have symbol conflict information. If a symbol must be renamed as part of the merge process, you are notified of the new symbol names at the end of the Symbol merge phase.

Conflicting symbols are renamed by adding a suffix of *\_conflict1* and an incremented number to make the name unique. The following image shows two renamed symbols. The first is the global symbol ME which was renamed to ME\_conflict1. The second is the symbol YOU in the FUN\_01001ac3 namespace which was renamed YOU\_conflict1.



Press the OK button to continue with the merge process.

After all symbol conflicts are resolved, the merge process continues with any Address Based Listing Merge conflicts.

#### [Address Based Listing Merge](#)

All of the address based listing elements (equates, user defined properties, references, bookmarks, and comments) has already been auto merged before any of their conflicts are presented for conflict resolution. Conflict resolution will proceed in address order for each address that has a conflict for any of the address based listing elements. At each address with a conflict you will have to resolve all of the conflicts (equates, user defined properties, references, bookmarks, and comments) before proceeding to the next address with conflicts. The following sub-sections describe conflict merging for each of the address based listing elements.

##### [Equates](#)

All equate conflicts that remain after the [auto merge](#) must be resolved.

**Equates** associate a name with a scalar. If your version and the latest version change the equate for a scalar and they are different as a result of the change, then the equates are in conflict and must be resolved manually. Changes include adding an equate to a scalar, removing an equate for a scalar, changing the equate name for a scalar.

If an equate is changed differently in the latest and your versions, then you must resolve the conflict.

Example : The following illustrates an equate conflict on the scalar 0x1 of operand 0 at 01001d0b. The scalar originally had an equate of 01. Each version changed the equate on 0x1 to a new name.

The image below shows the scenario where you:

**changed 01 → PEAR**

In the latest version:

**changed 01 → ORANGE**

The options to resolve the conflict are:

- Keep 'Latest' version PEAR 0x1
- Keep 'Checked Out' version ORANGE 0x1
- 'Original' version 01 0x1

Use the selected option for resolving all remaining 'Equate' conflicts.

**Apply**    **Cancel**

Resolving Equate conflicts.

0%

- Keep 'Checked Out' version – select this option to keep your equate
- 'Original' version – this row simply indicates the original value at the time you checked out the program

Select the radio button for either the latest or checked out version and press the **Apply** button to continue with the merge process.

After all equate conflicts are resolved, the merge process continues with the User Defined Properties.

#### User Defined Properties

All user defined property conflicts that remain after the [auto\\_merge](#) must be resolved.

User defined properties are individual named properties associated with addresses in the program. Individual plugins can create one or more property. All of the conflicts for a named property are presented to you and must be resolved before the next named property with conflicts is presented.

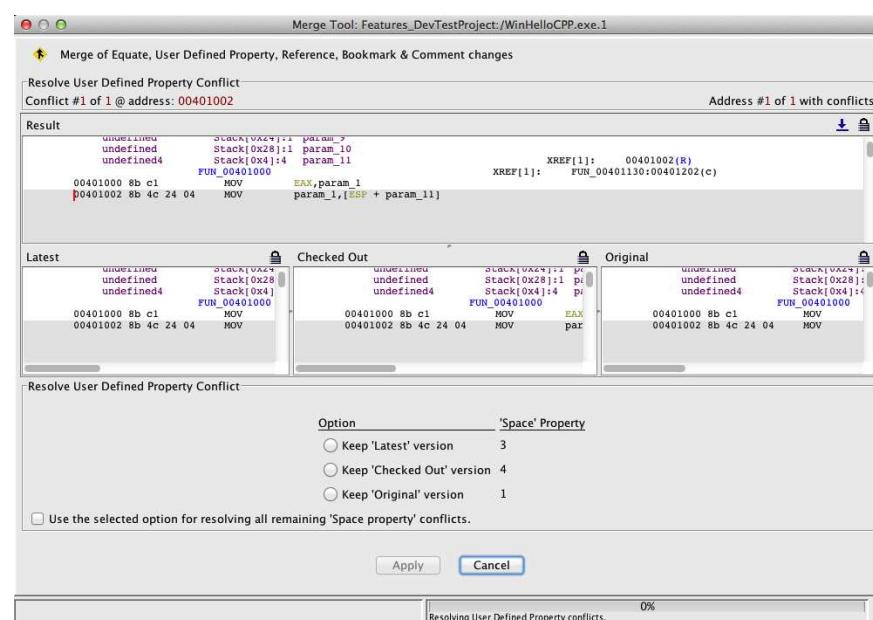
If the named property was changed at an address in your version of the program and changed differently in the latest version then a user defined property conflict exists.

**Example :** The image below shows the scenario where you:

Added a **Space** property with a value of 4.

In the latest version:

Added a **Space** property with a value of 3.



The options to resolve the conflict are:

- Keep 'Latest' version – select this option to keep what is in the latest version (lose your changes)
- Keep 'Checked Out' version – select this option to apply your changes (overwrites change in the latest version)
- Keep 'Original' version – select this option to restore the original value of the property (your changes and those in the latest version are lost)

Select the radio button for the value of the Space property you want to have in the checked in version.

You can resolve the remaining conflicts for the same named property by selecting the checkbox, *Use the selected option for resolving all remaining 'Space' property conflicts.* The same program version option that you selected will be applied to the remaining conflicts for the same named property. In the example above, the 'Check Out' version option would be used for the remaining conflicts for the **Space** property.

When you **Apply** after selecting the radio button and checkbox as illustrated in the example above, your changes (the Checked Out version) would be chosen for the **Space** property at the indicated address and for all remaining addresses that have the **Space** property in conflict. Because you selected the checkbox, you will no longer be prompted to resolve conflicts for that property type.

Selecting the *Use the selected option for resolving all ...* checkbox for the Space property would not affect the resolving of conflicts for any other named property. However, the user can select the checkbox for each different named property with conflicts.

After all user defined property conflicts are resolved, the merge process continues with the References.

#### References

All reference conflicts that remain after the [auto\\_merge](#) must be resolved.

**References** consist of memory references, stack references, register references and external references. The references in your checked out version and the latest version are compared for each mnemonic and operand at an address to determine if a conflict exists.

References conflict if the two versions of the program (your version and the latest version) have a reference at the same address and operand, but they are different types of references (memory, stack, register, external). References can also conflict when they are the same type if both versions changed the reference and one or more parts are now different. The following indicates the parts of a reference that can conflict with another reference of the same type.

Parts of an *external* reference that may conflict:

- external program name
- referenced label
- referenced address
- whether or not it is user defined

Parts of a *stack* reference that may conflict:

- stack offset
- referenced variable name
- first-use address

Parts of a *register* reference that may conflict:

- the register
- referenced variable name
- first-use address

Parts of a *memory* reference that may conflict:

- offset

- label
- reference type (computed call, computed jump, data, read, write, flow, etc.)
- whether or not it is a user reference

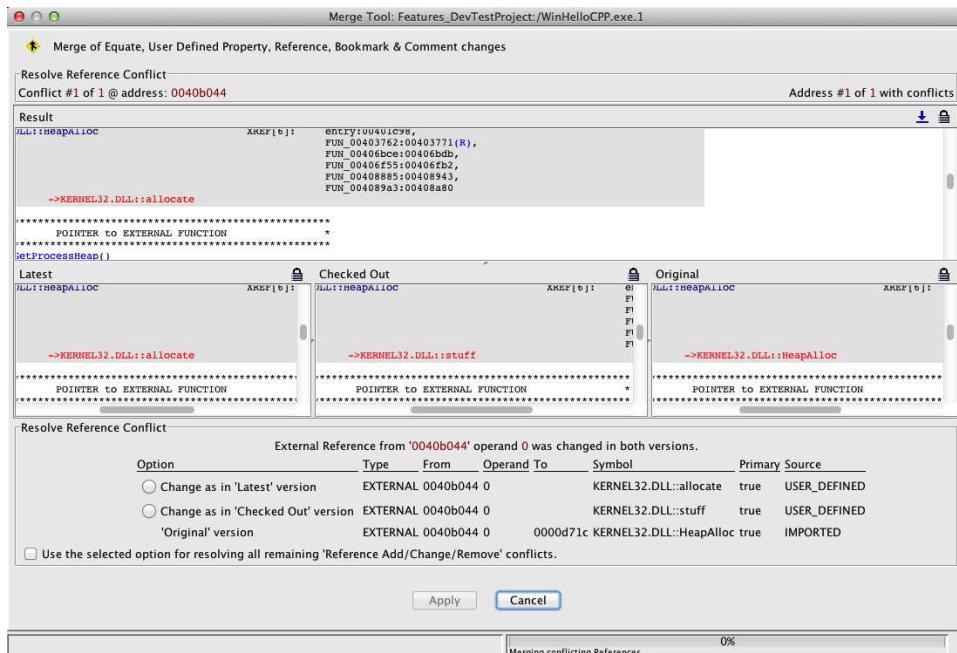
The following types of changes will result in reference conflicts:

- Your version and the latest version each added a different *reference type* (memory, stack, or external).  
If your checked out version and the latest version both have changes to the references for a mnemonic or operand, the references are in conflict whenever your version and the latest version have references that are different types (for example, one has memory references and the other has an external reference.)
- You removed a reference and the latest version changed the same reference or vice versa.
- Your version and the latest version both changed a reference and it now differs in one of the ways they can conflict as indicated above.
- Your version and the latest version both added the same type of reference with the same from and to addresses, but differed such that they *conflict*.
- Your version and the latest version both set the primary reference, but not to the same reference.



The first operand of an instruction is numbered as operand 0.

**Example\_1 :** The following image illustrates an *external reference conflict*. Your version and the latest version both changed the external reference at address 01001000 operand 0. The latest version caused it to become user defined and you changed it to refer to the symbol "buf" instead of "IsTextUnicode".

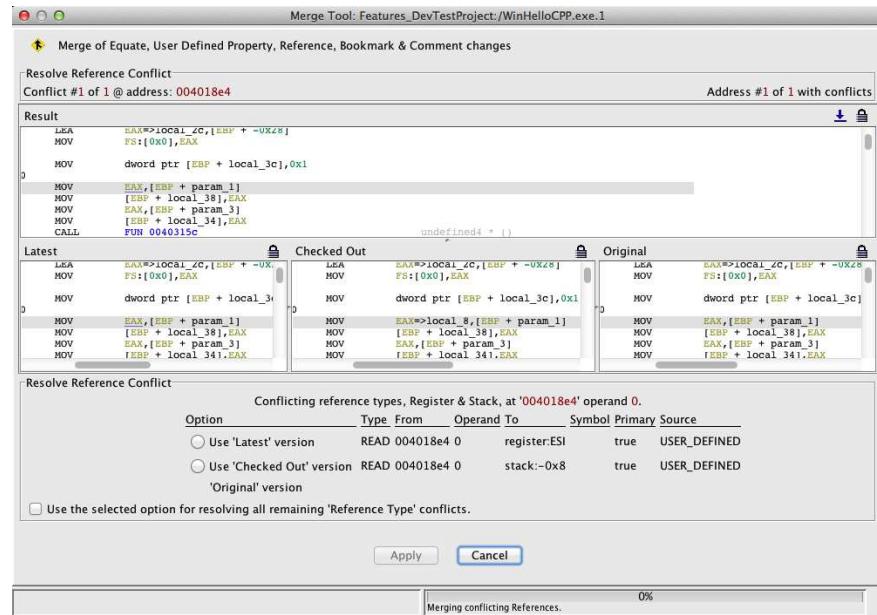


The options to resolve the conflict are:

- *Change as in Latest version* – select this option to keep what is in the latest version (lose your changes). In this case, keep the reference to "IsTextUnicode."
- *Change as in Checked Out version* – select this option to apply your changes (overwrites change in the latest version). In this case keep the reference to "buf."
- *Original* version – This field is not selectable, but is provided to show the value in the Original version that you checked out.

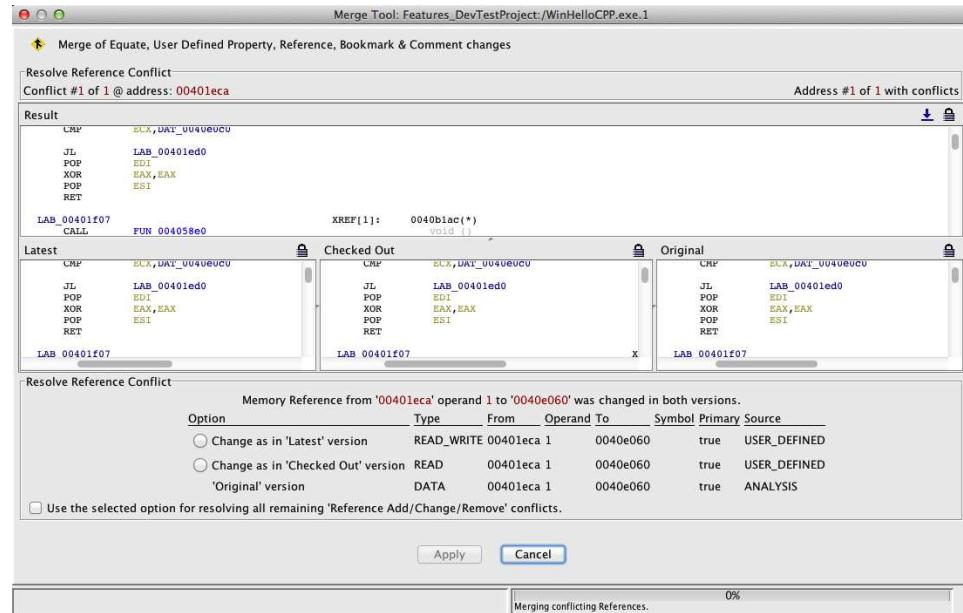
Select the radio button for whether to keep the reference to 0x10 or 0x14 and select the **Apply** button to continue with the merge process.

**Example\_2 :** The following image illustrates *conflicting reference types* being added in your version and the latest version. Possible reference types are memory, stack, register or external. In this case, you added a stack reference and the latest version has a register reference added at address 010018cd operand 0.



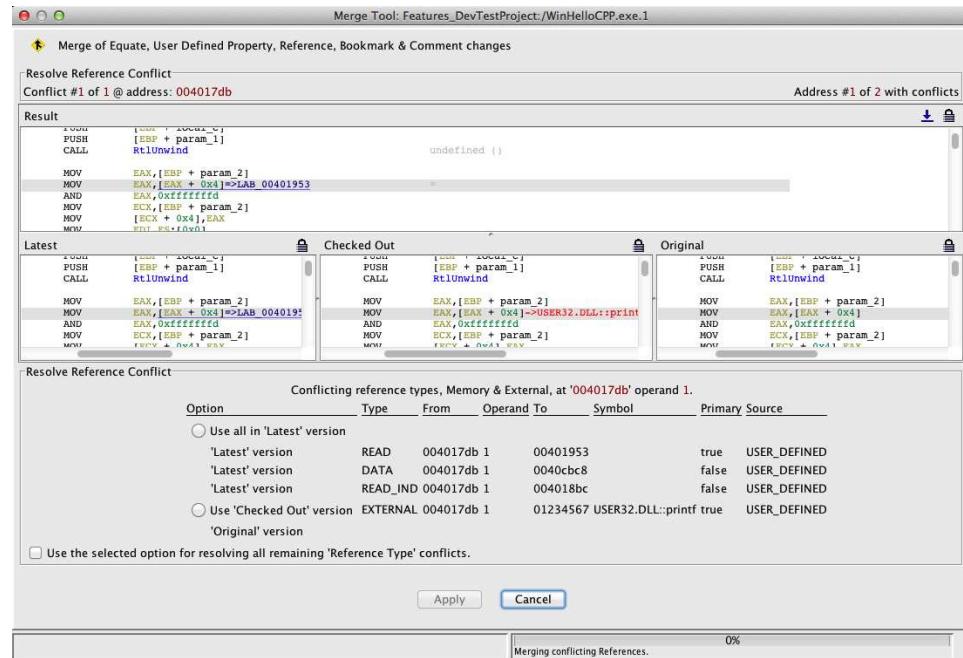
Once again, select the radio button for which reference to keep and select the **Apply** button to continue with the merge process.

**Example 3 :** The next image illustrates a *memory reference conflict* where you changed an existing memory reference from a DATA reference to a WRITE reference and the latest version changed the memory reference to a READ\_WRITE reference.



Select the radio button for which reference to keep and select the **Apply** button to continue with the merge process.

**Example 4 :** The following illustrates a *conflict when different reference types are added* in your version and the latest version. In this case you added an external reference at address 01002510 operand 0 and the latest version added multiple memory references.



The options to resolve the conflict are:

- *Use all in 'Latest' version* – select this option to keep what is in the latest version (lose your changes). This would keep all of the memory references and lose the external reference.
- *Use 'Checked Out' version* – select this option to apply your changes (overwrites change in the latest version). This would keep the external reference and lose all of the memory references.
- *'Original' version* – This field is not selectable, but is provided to show the value in the Original version that you checked out. In this case there was no reference originally.

Select the radio button for which references to keep and press the **Apply** button to continue with the merge process.

After all reference conflicts are resolved, the merge process continues with the Bookmarks.

#### Bookmarks

All bookmark conflicts that remain after the `auto_merge` must be resolved.

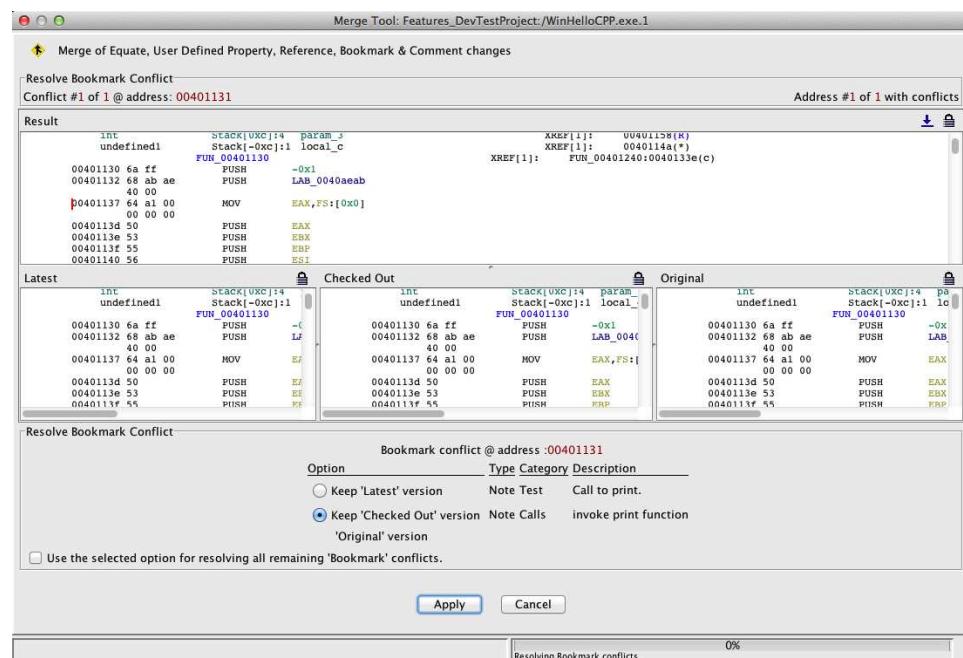
Note bookmarks are in conflict when any of the following occur:

- your version and the latest version both add a **Note** bookmark and the category or description differ.
- your version removes a bookmark and the latest version changes the category or description or vice versa.
- your version and the latest version change the category and/or description so that they no longer match.

All other bookmarks are in conflict when any of the following occur:

- your version and the latest version both add a non-**Note** bookmark with the same category but different description.
- your version removes a non-**Note** bookmark of a specific category and the latest version changes the description for that bookmark type and category or vice versa.
- You cannot directly change the description on a non-**Note** bookmark through regular bookmark editing in Ghidra, but a plugin could change it programmatically.
- your version and the latest version change the description for a bookmark of a particular type and category so that they no longer match.

**Example\_1 :** The next image illustrates a conflict due to your version and the latest version both adding different Note bookmarks at the same address.



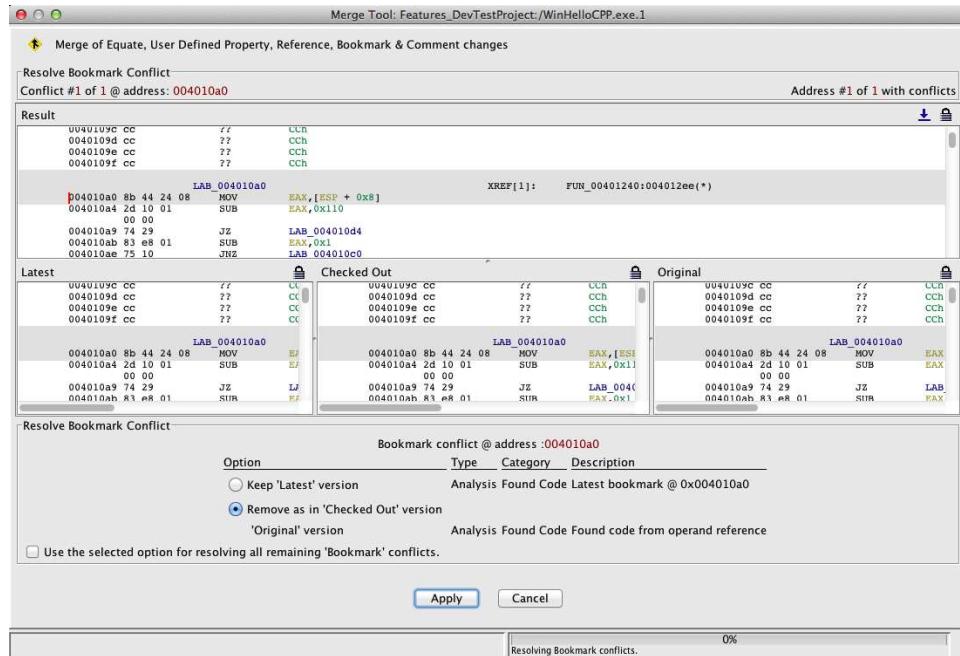
When bookmarks are in conflict, you can:

- Keep 'Latest' version – select this option to keep the bookmark change in the latest version (lose your changes)
- Keep 'Checked Out' version – select this option to apply your bookmark changes at the address (overwrites change in the latest version)
- 'Original' version – This field is not selectable, but is provided to show the value in the Original version that you checked out

Notice that the above bookmarks conflict even though they are not in the same category. This is because only one **Note** bookmark is allowed at an address regardless of its category.

Select either the latest version or your checked out version and then select **Apply** to continue with the merge.

Example\_2 : The next image illustrates an **Analysis** bookmark where another user changed the comment in the latest version, but you removed the bookmark.



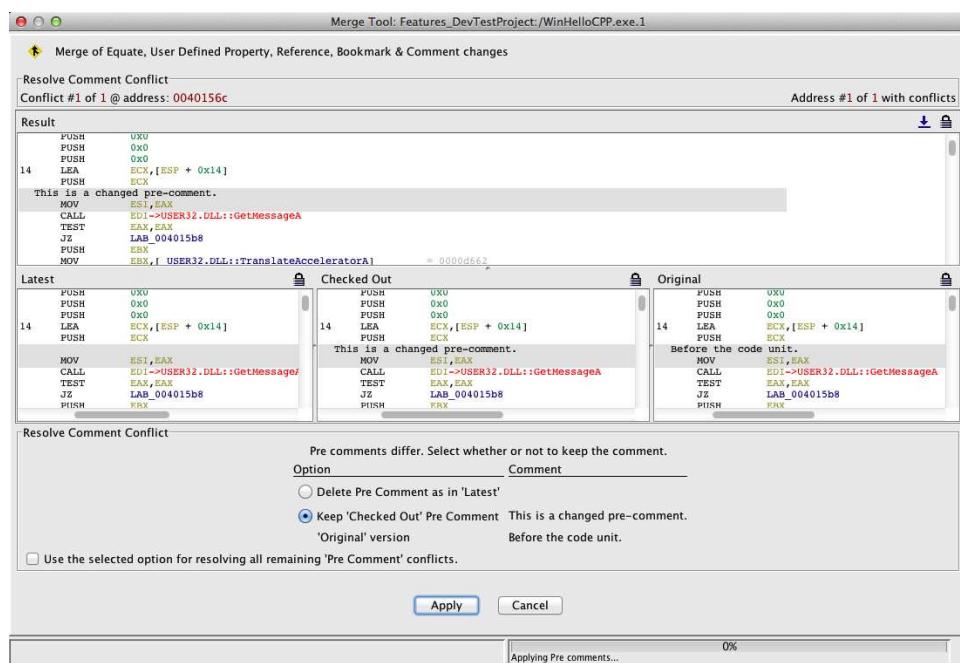
Select *Keep Latest* version to keep the other user's changes or select *Remove as in 'Checked Out'* if you don't want the bookmark in the resulting program. Then select the **Apply** button to continue merging.

#### Comments

All comment conflicts that remain after the [auto\\_merge](#) must be resolved at each conflict address.

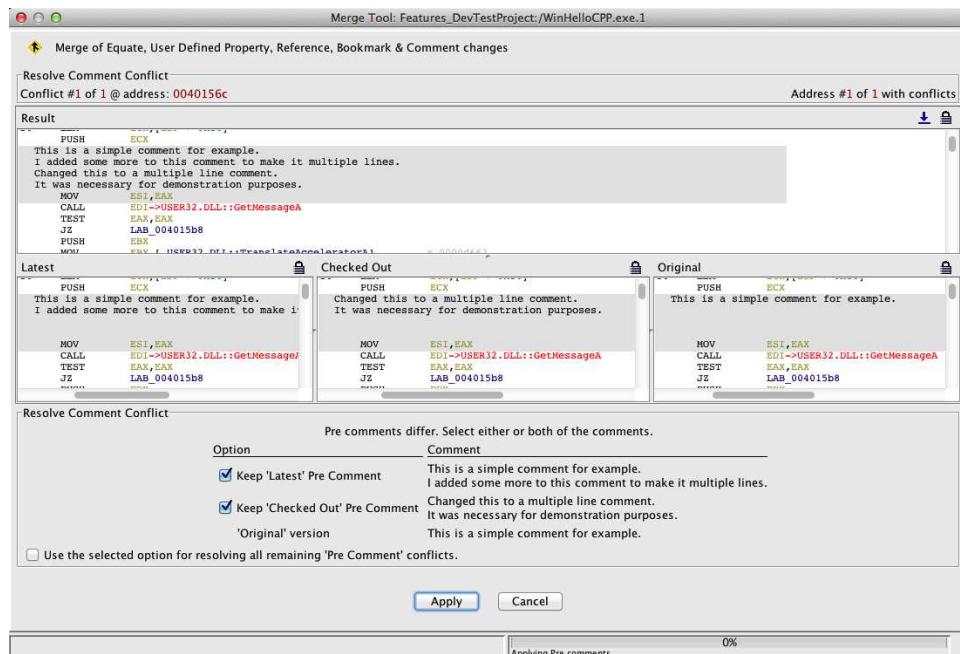
Only comments of the same type and at the same address can conflict with each other. The types of comments are Plate, Pre, End-of-Line, Repeatable, and Post.

If a comment is removed in one version and changed in the other version, you must choose whether to keep or remove it. This scenario is shown in the image below.



If both versions added the same type of comment at an address and the two comments don't match, then you must decide to keep the latest version's comment, your comment, or both comments.

Similarly, if a particular comment was changed in both versions and the two comments no longer match then you must decide to keep the latest version's comment, your comment, or both comments. This scenario is shown in the image below.



If you choose to keep both comments by placing checkmarks in both boxes, your comment is appended to the latest comment with a new line separating them. If one of the comments is contained within the other comment, then the longer comment is kept instead of combining them with a new line separator.

After all address based listing conflicts are resolved, the merge process continues with External Program Names.

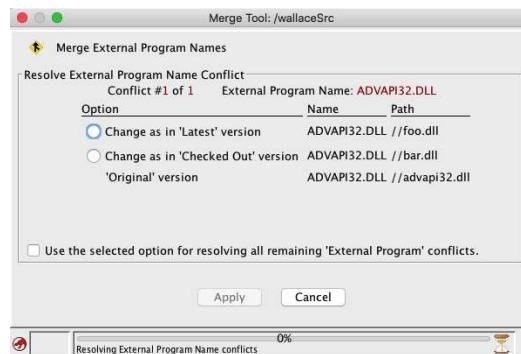
#### External Program Names

All external program name conflicts that remain after the [auto\\_merge](#) must be resolved.

A conflict occurs if :

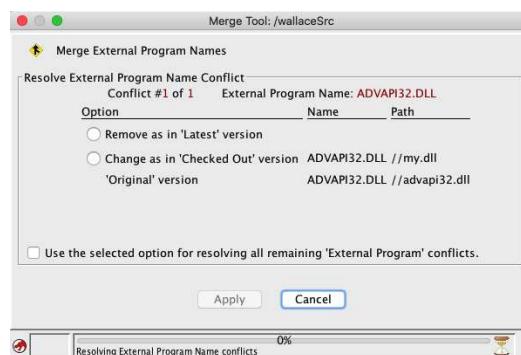
- both versions add the same external program name with different paths
- both versions changed the path for an existing name
- one version removes the external program name and the other redefines the Ghidra program path associated with that name

[Example 1](#) : The following image shows a conflict when the latest and your versions changed the path for an external program name.



Select the version (Latest or Checked Out) of Ghidra program path to associate with the external program name and then select the **Apply** button to proceed with the merge.

[Example 2](#) : The image below shows a conflict due to an external program name being removed in one version and changed in the other.



In the above scenario, the external program name "ADVAPI32.DLL" was removed in the latest version. However, the program path indicating which Ghidra program is associated with the external program name, was modified in your checked out version.

The options to resolve the conflict are:

- *Remove as in 'Latest' version* – select this option to remove the External Program Name definition for ADVAPI32.DLL (lose your changes)
- *Keep 'Checked Out' version* – select this option to apply your changes resulting in a new program path for ADVAPI32.DLL (overwrites change in the latest version)
- *'Original' version* – This field is not selectable, but is provided to show the value in the Original version that you checked out

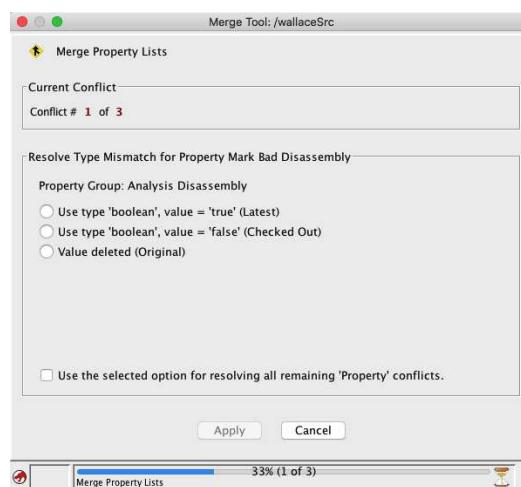
Select the radio button for the desired result and then select the **Apply** button to proceed with the merge.

 When you remove an external program name that is in conflict, it will get added back later as a result of choosing a reference that refers to that external program name.

#### Property Lists

All property list conflicts that remain after the [auto\\_merge](#) must be resolved.

A property list conflict will result when you change a property that was either deleted or changed in the latest version. The image below shows this scenario:



Here, in the *Analysis Disassembly* Property Group, another user changed the *Mark Bad Disassembly* property. You changed the same property. You have following choices:

- Keep the other user's changes (*Latest*), and lose your change
- Keep your changes (*Checked Out*) and lose the change in latest version
- Keep the values from the original version of the program (*Original*), lose your changes and lose changes made in the latest version. In this example, if you choose this option, the property will be deleted.

To resolve the conflict, select a radio button, and then click on the **Apply** button to go to the next property conflict. In this example, there are three property list conflicts (noted in the *Current Conflict* area of this window). The progress bar on the lower right corner of the Merge Tool indicates progress only for the Property List merge, not the entire merge process.

After you resolve all conflicts, the check in process is complete. The Ghidra Server will contain a new version of the program file.

#### Related Topics:

- [Shared Project Repository](#)
- [Check in](#)
- [Memory Blocks](#)
- [Program Trees](#)
- [Data Types and Categories](#)
- [Program Options](#)
- [Property lists](#)
- [Symbols](#)
- [Property Viewer](#)

## Ghidra Tool Administration

A Ghidra Tool is a collection of building blocks, called **Plugins**. You can create tools by combining different Plugins that cooperate with one another to achieve certain functionality. You can add tools to the Tool Chest or configure them to share data and resources with other tools. Ghidra provides a set of Plugins, but you may create your own Plugins to add more functionality to your tools.

### Default Tool

Ghidra provides a default tool, the **Code Browser** that is in your Tool Chest. It has all of the Core plugins already loaded. This tool may be **re-imported** into your tool chest at any time. If the tools exist in your tool chest, then a one-up number is assigned to the name to make them unique. (You will see the name in the tool tip, and in the **Tools** → **Run** menu.) Any tool that you create and add to your Tool Chest is always available to your active Project. However, you should save the tool if you plan to use it for other Ghidra projects.

### Tool Management

The following sections describe tool management:

- [Create Tool](#)
- [Save Tool to Tool Chest](#)
- [Export Tool](#)
- [Import Tool into Tool Chest](#)
- [Configure Tool](#)
- [Run Tool](#)
- [Connect Tools](#)
- [Close Tool](#)
- [Delete Tool](#)

#### Create Tool

Create Tool creates an empty tool (no Plugins). What Plugins you should add is dictated by what you want the tool to do. The following steps can help you create a useful tool:

1. Define your requirements for a tool.
2. Determine whether the existing Plugins meet your requirements. If the Plugins do not provide the functionality, you may need to write your own Plugin that does meet your requirements.
3. Add these Plugins to an existing tool or to a new tool.

To create a New Tool,

1. From the Ghidra Project Window, select the **Tools** → **Create Tool...** option.
2. A new "empty" tool is displayed; the [Configure](#) dialog is displayed.
3. [Configure](#) the tool.
4. Save the tool.

#### Save Tool to Tool Chest

When you save a tool to your Tool Chest, you are saving the tool's configuration such that it is available to your active project. (When you open any project, icons for all tools from your Tool Chest will appear in the Tool Chest panel in the [Ghidra Project Window](#)). An icon for a new tool shows up on the Tool Chest panel of the Ghidra Project Window.



To save a tool,

- From the tool, select **File** → **Save Tool**

To save a tool to a different name, or to change the icon,

1. From the tool, select **File** → **Save Tool As...**
2. Enter a new tool name (the current tool name is shown in the *Tool Name* field by default).
3. Choose an icon from the list of icons ... OR
- Click on the file chooser button (...) to choose a filename from the file system.
4. Click on the **Save** button; if this is a new tool that you are adding to your tool chest, the Tool Chest panel will show the icon for the tool; icon and name changes (tool tip) and are reflected in the Tool Chest panel as well.

#### Export Tool

You can export your tool to an XML file in order to share your tool with other users. You may have configured your tool for a particular area of research or practice; others who are working on a similar problem may find your tool useful.

- To export a tool from the menu:

1. Select **File** → **Export Tool**; a file chooser is displayed.
2. Select a folder and name for the exported file.
3. Click on the **Export** button.

- To export a tool from the icon:

1. In the "Tool Chest", right mouse click on the icon for the tool
2. Select the **Export...** option.

The **Status** area on the Ghidra Project Window indicates whether the export was successful.

 If the tool contains Plugins that are not part of Ghidra (i.e., you wrote new Plugins), you will have to distribute a jar file containing the class files for the Plugins along with the XML file. The recipient can then just place the jar file in one of the [locations](#) where Ghidra will search for available Plugins.

**Import Tool****Import Tool to Tool Chest**

Use the Import Tool option to import an [exported tool](#) into Ghidra. The imported tool will appear in your [Tool Chest](#) with the other tools. If the name of the imported tool already exists, a one-up number is assigned to the newly imported tool to make the name unique.

To import a tool,

1. From the Ghidra Project Window, select **Tools** ➔ **Import Tool to Tool Chest...**
2. Select a file with a ".tool" or ".obj" extension.
3. Click on the **Import** button.

If the import was successful, an icon for the tool is added to the Tool Chest panel. Any errors during the import are reported in the [Status](#) area on the Ghidra Project Window.

 If the tool that you are importing contains Plugins that are not part of Ghidra, you must have the jar file containing the class files for the Plugins. Place the jar file in one of the [locations](#) where Ghidra will search for available Plugins and restart Ghidra.

**Import Default Tools to Tool Chest**

The default tool for the [Code Browser](#) is part of the Ghidra installation, and can be added to your Tool Chest at any time. So if you [delete](#) the default tool from your Tool Chest, you can always recover them by using the [Import Default Tools to Tool Chest](#) option.

To import default tools,

1. From the [Ghidra Project Window](#), select **Tools** ➔ **Import Default...**
2. A dialog is displayed that allows you to selectively import default tools. Select the tools you want to import and then select **OK**.



3. The default tools are added to your Tool Chest; if the default tools already exist, then a one-up number is assigned to them to make the names unique, e.g., `CodeBrowser_1`.

**Run Tool**

To launch a tool from the [Tool Chest](#),

- Click on the icon for that tool in the Tool Chest panel on the Ghidra Project Window, OR
- From the [Ghidra Project Window](#), select **Tools** ➔ **Run Tool** ➔ <tool name>

To launch a tool with a specific Program,

- Drag a Program file from the [data tree](#) in the Ghidra Project Window, and drop it onto a tool icon in the Tool Chest panel. The tool is launched and has the Program opened.

 If you bring up more than one instance of the same tool, the window title shows a one-up number appended to the tool name, e.g., `CodeBrowser(2)`.

**Rename Tool**

To rename a tool from the [Tool Chest](#),

- Right-click on the icon for the tool in the Tool Chest panel in the Ghidra Project Window and select **Rename** from the popup menu.
- Enter the new tool name in the dialog that appears and press the <OK> button.

**Set Tool Associations**

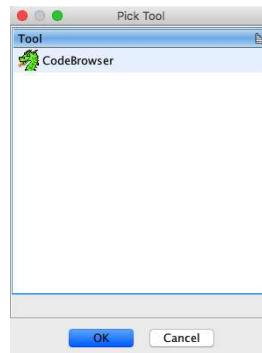
For each type of file in the Front End you can set the tool that Ghidra uses to open that file, the associated tool. As an example, programs in Ghidra are, by default, associated with the [Code Browser](#) tool. The associated tool is the tool that will be chosen by Ghidra to open any file that is double-clicked. Regardless of the current associated tool, you may, at any time, drag a file in the Front End onto any tool in the Tool Chest in order to open the file in that tool.

To change the associated tool for any file type, select **Tools** ➔ **Set Tool Associations...** menu. This will open the [Set Tool Associations Dialog](#)



This dialog shows a list of known content types and the tool that is currently used to open that type when double-clicked in the Front End. If the name and icon of the tool is grayed-out, that means the associated tool is not in your tool chest.

To change a tool association, select the desired **Content Type** in the list and then click the **Edit** button. This button will trigger the [Pick Tool](#) dialog.



This dialog shows a list of all tools that can open the given content type. If you choose a tool from the list that is not in your tool chest, then that tool will be added for you when you close the dialogs.

The **Restore Default** button allows you to restore the tool association to the default setting.

Ghidra uses tool names to store tool associations. This can lead to odd behavior if you rename a custom tool to be the name of a default tool. If your tools exhibit odd behavior, such as being unable to open a file that is considered to be a default type, then you can always delete your tools and [re-import the default tools](#) again.

#### [Close Tool](#)

To close a tool (not the [Ghidra Project Window](#)).

1. Select the **File** → **Close Tool** option, OR
2. Right mouse click on the icon for the running tool and choose the **Close** option.

#### [Program Changes](#)

If you made changes to the Program and not saved them before you closed the tool, then a dialog is displayed to ask whether you want to save your changes.



- Choose the **Save** button to save the Program.
- Choose the **Don't Save** button to close the tool but do not save the changes.
- Choose the **Cancel** button if you do not want to close the tool.

This dialog is displayed when the *last tool is closed that has this Program opened*. So if you have two tools running with the same Program opened and you close one tool, the dialog will not be displayed until you close the second tool.

You cannot close a tool while a background process is running, e.g., Disassembly or Clear. You must first stop the operation, then close the tool.

#### [Tool Configuration Changes](#)

Changes to tools are automatically saved by default. If you open multiple instances of the same tool and make changes to that tool, then Ghidra cannot automatically save the tool for you. In this case, Ghidra will ask you to make a decision regarding saving the changes.

You may configure Ghidra to not automatically save the tool via the Front End options (**Edit**→**Options**→**Tool**→**Automatically Save Tools**).

#### [Delete Tool](#)

To delete a tool from your [Tool Chest](#).

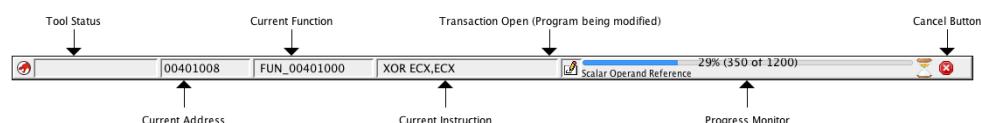
- From the [Ghidra Project Window](#), Select the **Tools** → **Delete Tool** → <tool name> OR,
- Right mouse click on the icon in the Tool Chest and select the **Delete...** option.

A dialog to confirm your delete request is displayed; choose the **Delete** button to remove the tool from your Tool Chest.

You can always get your default tools back into your Tool Chest by selecting the [Import Default Tools to Tool Chest](#) option. However, deleting your own custom tool from your Tool Chest is a **permanent operation**.

#### [Tool Status Components](#)

The tool has a status area to display messages and an area to show progress for a running task, e.g., disassembly. On the right side of the lower portion of the tool, a "write" icon indicates that the program is being modified. The progress bar shows the progress of the running task. The spinning globe indicates that something is happening in the tool. The cancel button allows you to cancel the currently running task. You can still interact with the program while the task is running.



There may be times when a task is "modal" such that no user interaction with the program is allowed until either the task completes or you cancel it. A dialog is displayed to indicate what task is running and to allow you to cancel it. Depending on the task, the dialog may or may not show a progress bar.

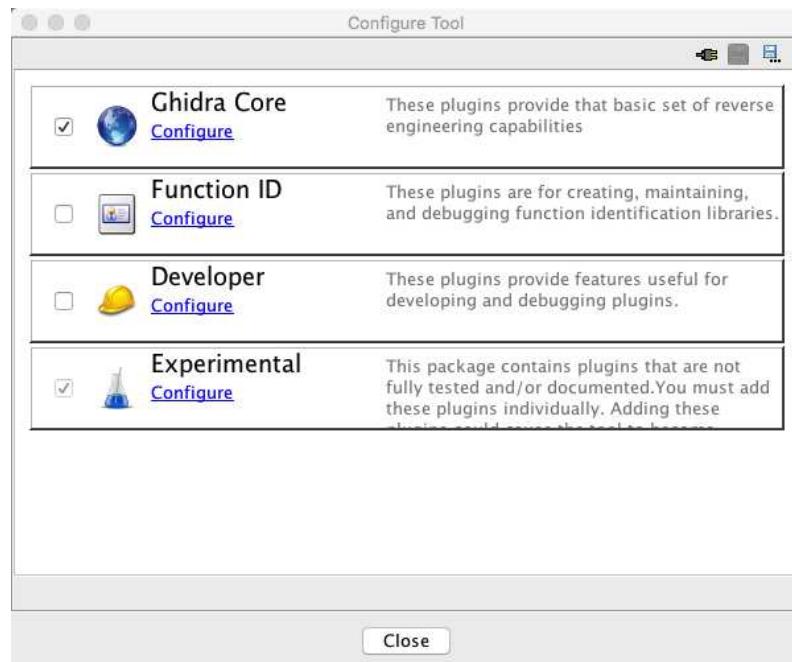


Related Topics:

- [Configure Tool](#)
- [Ghidra Project Window](#)
- [Configure Ghidra Project Window](#)
- [Code Viewer](#)

## Configure Tool

The *Configure Tool* dialog allows you to add/remove plugin packages or individual [Plugins](#) from a tool. To display the *Configure Tool* dialog, select **File ➔ Configure...**. This dialog is also displayed when you [create a new tool](#).



The *Configure Tool* dialog shows a list of plugin packages that can be added to the tool. Clicking the checkbox will add (or remove) all the plugins in the package to the tool. Clicking on the **Configure** link will bring up a dialog for adding individual plugins.



The [Ghidra Project Window's Configure](#) display shows its own list of eligible front-end plugins, as only certain plugins may be added to the Project Window.

### Saving

Save changes to your tool by clicking on the icon in the dialog's toolbar; [save your tool to a different name](#) by clicking on the .

### Configuring All Plugins

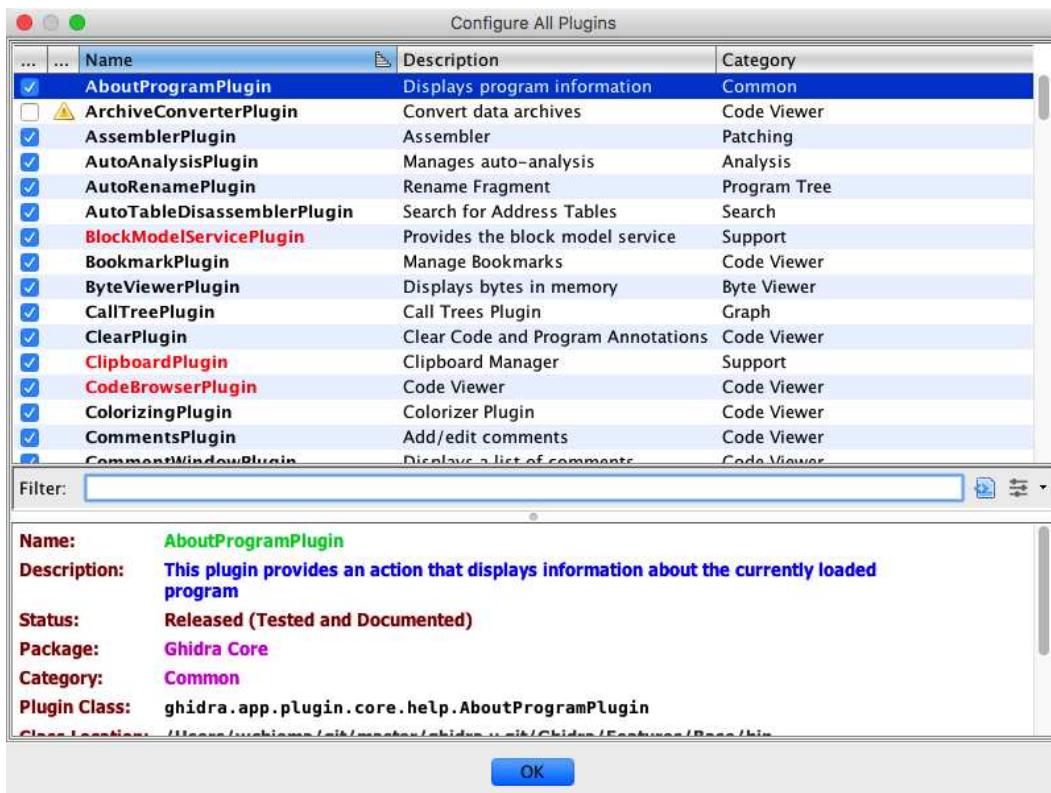
To Configure all plugins regardless of package, select the icon in the dialog's toolbar and the *Configure Plugins* dialog will appear with all plugins in its plugin table.



The **Experimental** package can't be added as a package. Experimental plugins must be added individually.

### Configure Plugins Dialog

Selecting the *Configure* link for a package will bring up the the *Configure Plugins Dialog*.



The dialog has two parts: A table at the top of the dialog that shows all the plugins in the package and an information window at the bottom that shows details about an individual plugin.

### Plugin Table

The plugin table shows the following information for each plugin:

- Checkbox to indicate whether the plugin is in the tool;
- A status icon:
  - none – the plugin is good. It has help and been reasonably tested.
  - – the plugin is useable, but has not been fully tested and/or not documented.
  - – the plugin is under development and may not be usable at all. Not included with production distribution.
- Plugin name: the name is displayed in red when some other plugin depends on this plugin;
- Short description of the plugin;
- Category for where the plugin belongs functionally, e.g., it works in the context of a Code Browser or Byte Viewer, etc.

The *Search Filter* allows you to narrow the list of plugins displayed in the table. Only those plugins whose name or description contains the string that you enter as the filter will be displayed. As you type, the table is updated to reflect the filter.

### Information Window

When you select a row in the table, the scrolled window below the table shows more information about the plugin and any contact information that the author supplied, e.g., author's name, organization, etc. The *Dependencies* section lists the class names of the plugins that depend on the selected plugin due to some service that it provides. The *Class Location* indicates from where the java classes are being loaded.

### Related Topics:

- [Create Tool](#)
- [Configure the Ghidra Project Window](#)
- [Save Tool to Tool Chest](#)
- [Ghidra Tool Administration](#)

## Tool Options Dialog

Each tool has an *Options* dialog that shows options in a tree format. When you click on the node, the associated options appear in a panel to the right of the tree. At a minimum, the tree has a node for key bindings and tool options . Select the node to show the corresponding options that you can change. Plugins may provide their own options, in which case new nodes in the tree or new options for the tool may show up. Options provide a flexible way for changing plugin behavior or functionality.

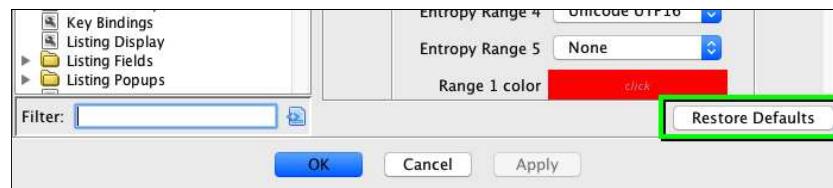


The Tool Options dialog has a filter text field that can be used to quickly find options relating to a keyword. Any options names or descriptions that contain the keyword text will be displayed.

To display the *Options* dialog, select **Edit→Tool Options...** from the tool menu.

### Restoring Default Settings

You can restore any currently selected options panel to its default settings by pressing the **Restore Defaults** button at the bottom of the options panel. Use caution when executing this action, as it cannot be undone.



### Key Bindings

You can create a new "hot key" for a Plugin's action or modify the default key binding. The hot key (or accelerator) that you add can be used to execute the action with a keystroke combination.



You can also change key bindings from within Ghidra by pressing **F4** while the mouse is over any toolbar icon or menu item. Click [here](#) for more info.

The *Key Bindings* panel has a table containing the *Action Name*, *Key Binding*, and *Plugin Name*. You can sort the columns in ascending or descending order. (By default, the *Action Name* column is sorted in ascending order.) The Plugin name is the name of the plugin supplying the action.

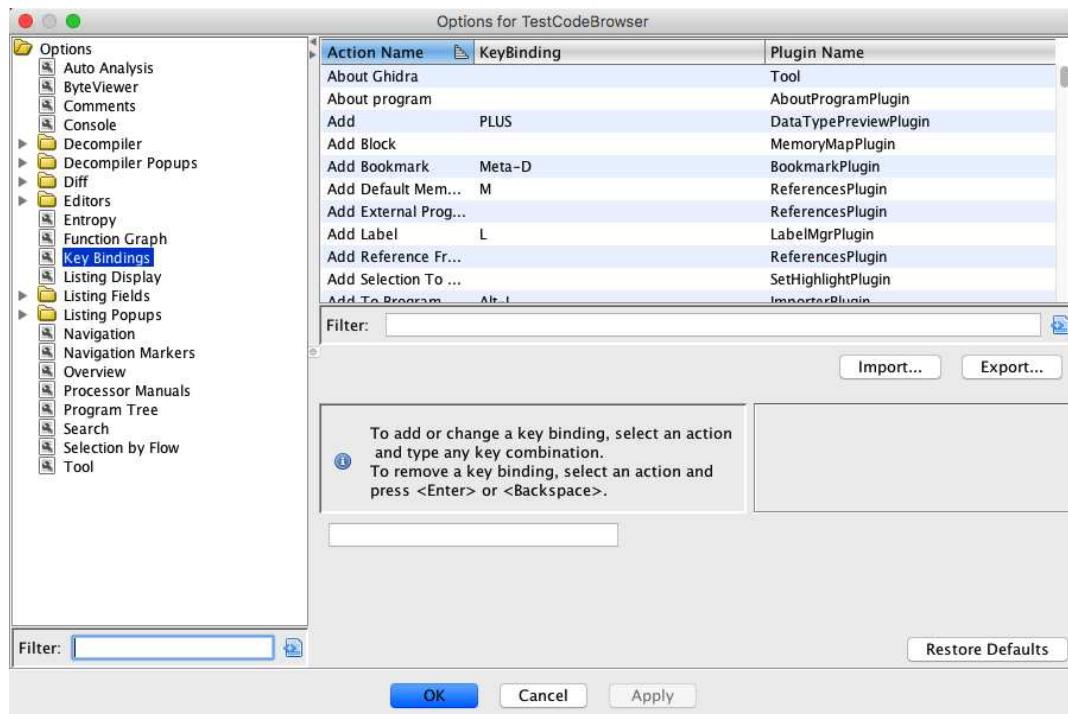
- Click on the category header to change the sort order.
- Change the order of the columns by dragging the column header to another position in the table.
- The text field below the table captures keystroke combinations entered.
- If an action has a description to explain what it does, it will be displayed below the text field.



The table entries are not editable.

The display below shows the key bindings panel for the *Project Window*. Using the Key Bindings Options panel works the same as for a regular Ghidra Tool.





### Change a Key Binding

To change the Key Binding,

1. Select **Edit ➔ Tool Options...** from the main menu.
2. Select the **Key Bindings** node in the options tree.
3. Select an action name to either set a key binding or change the existing key binding.
4. Click in the text field and type the key or keystroke combination (e.g., Ctrl x).
  - When a key is mapped to multiple actions, the action name is listed below the text field.
5. Click on the **OK** or **Apply** button.

When a key is mapped to multiple actions, and more than one of these actions is valid in the current context (i.e., the action is enabled), then a dialog is displayed for you to choose what action you want to perform.

To avoid the extra step of choosing the action from the dialog, do not map the same key to actions that are applicable in the same context.

### Remove a Key Binding

To remove a Key Binding,

1. Select **Edit ➔ Tool Options...** from the main menu.
2. Select the **Key Bindings** node in the options tree.
3. Select an action name for the key binding that you want to remove.
4. Click in the text field for the key binding.
5. Press the <Enter> or <Backspace> to clear it.
6. Click on the **OK** or **Apply** button.

### Import Key Bindings

To import a Key Binding,

1. Select **Edit ➔ Tool Options...** from the main menu.
2. Select the **Key Bindings** node in the options tree.
3. Press the **Import...** button.
4. On the warning dialog, press the **Yes** button to import key bindings or the **No** button to cancel the process.
5. On the file chooser dialog, choose a previously exported file from which to import key bindings.
6. Press **OK** to import the key bindings.

Importing key bindings will override your current key bindings settings. It is suggested that you [export your key bindings](#) before you import so that you may revert to your previous settings if necessary.

After importing you must save your tool (**File ➔ Save Tool**) if you want your changes to persist between tool invocations.

### Key Binding Short-Cut

A key binding can be applied to any menu item or toolbar icon. For example:

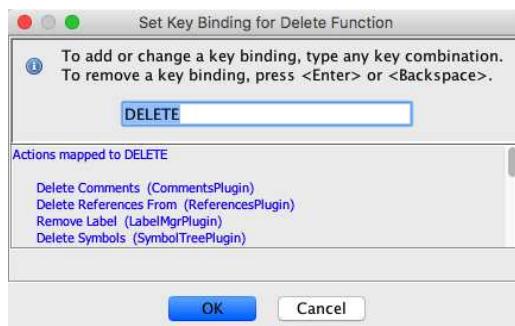
File ➔ Close  
Data ➔ Cycle ➔ Cycle: Float, Double

Apply key bindings to menu items or icons that are frequently accessed. To do this:

1. Display a menu item.
2. Place the cursor on the menu item or let the mouse hover over an icon on the toolbar.

 This menu item or icon will be associated with the Key Binding. When the Key Binding key is used, this menu item or action associated with the icon will be applied.

3. Press the <F4> key to display **Set Key Binding** dialog:



4. Enter a key combination in the **Set Key Binding** dialog. The panel below the text field that accepts the key input shows the other actions that are mapped to the key. These are potential collisions if these actions are enabled at the same time. Press **OK** to change the key binding.
5. The key combination that is entered in this dialog will be the key binding for the menu item.

#### Export Key Bindings

To export a Key Binding,

1. Select **Edit ➔ Options...** from the Tool menu.
2. Select the **Key Bindings** node in the options tree.
3. Press the **Export...** button.
4. If you have made changes, then you will be prompted to apply those changes before continuing.
5. On the file chooser dialog, choose a file to which to export key bindings.
6. Press **OK** to export the key bindings.

## Tool

*Tool* is a default node in the options tree that shows up in each tool's options window. The *Tool* panel defines the options for the Tool. The table below lists the basic options. Plugins may add their own options to the *Tool* options. If a tool does not have a plugin that uses an option, the option will not show up on the *Tool* panel. For example, the Ghidra Project Window does not have plugins that use the Max Go to Entries, Search Limit, or Subroutine Model so these options will not appear on the *Tool* panel. If an option has a description, it will show up in the description panel below the tree when you pass the mouse pointer over the component in the options panel.

Tool Options	
Option	Description
Docking Windows On Top	Selected means to show each undocked window on top of its parent tool window; the undocked window will not get "lost" behind its parent window. Unselected means that the undocked window may go behind other windows once it loses focus. Use the Windows menu to make the undocked window visible.
Max Goto Entries	Number of past entries to keep in the <a href="#">Go to Address or Label</a> dialog
Search Limit	Numeric: The maximum number of search hits to allow; for example, when you <a href="#">search for text</a> in the Program, the search is stopped when the number of matches exceeds this limit, or the max number of addresses displayed while doing a <a href="#">Go To</a> for an entry that has multiple matches.
Subroutine Model	Sets the default subroutine model. This setting is mainly used when creating call graphs. See <a href="#">Block Models</a> for a description of the valid Models.
Use C-like Numeric Formatting for Addresses	Selected means to attempt to interpret the value entered in the <a href="#">Go To dialog</a> as a number as follows: <ul style="list-style-type: none"> <li>• interpret the value as a hex number if it starts with "0x"</li> <li>• interpret the value as an octal number if it starts with "0"</li> <li>• interpret the value as a binary number if it ends with a "b"</li> </ul>

To change Tool Options,

1. From the tool, select **Edit ➔ Tool Options...**
2. Select the *Tool* node in the options tree.
3. Change the value for the option.
4. Click on the **OK** or **Apply** button.

## Tool

Some **Tool** options can only be set from the [Front End](#). Some of those are described below.

Tool Options	
Option	Description
<b>Swing Look and Feel</b>	This controls the appearance of the UI widgets for things such as colors and fonts. Each operating system provides a different default Look and Feel. Some of these work better than others.
Use Inverted Colors	<p>This is a <b>prototype</b> feature that allows the user to invert each color of the UI. Doing this effectively creates a Dark Theme, which some users find less visually straining.</p> <p> As a prototype feature, this feature has many known issues, including:</p> <ul style="list-style-type: none"> <li>● Pre-generated content, such as images, icons and help files will have inverted colors.</li> <li>● Some color combinations will be difficult to read</li> </ul>

### Related Topics:

- [Go to Address or Label](#)
- [Subroutine Model](#)

# Tool Connections

Ghidra Tools that share data and interact dynamically are said to be *connected*. Tools are connected via *tool events*. Tools generate events when you:

- open or close a Program
- move the cursor to a different location in the Program
- make a [selection](#) in the Program (drag the mouse over some addresses).

A main reason to connect tools is to have one tool track Program location or selections in another tool.

A tool may generate events that only another tool of this type can process, e.g., the Code Browser in one tool produces and consumes an edit event that is appropriate for a Code Browser in another tool. Thus, tools can be connected for only those events that they have in common.

Tool connections are directional. You can connect two Tools A and B in a *single direction* such that Tool A generates an event consumed by Tool B; however, Tool A will not consume any events generated by Tool B. You can also connect Tools A and B such that the connections are *bi-directional*, where Tool A generates an event consumed by Tool B, and Tool B generates an event consumed by Tool A.

The following sections describe the ways to connect tools.

## Automatic Tool Connection

You automatically connect tools when you:

- drag the icon for one running tool onto another running tool, OR
- drag the icon from the [Tool Chest](#) onto the icon for the running tool.

The tools are connected in both directions for all events (all Tool A events will be consumed by Tool B, and all Tool B events will be consumed by Tool A). For example, if you move the cursor in Tool A, the location change is reflected in Tool B. Conversely, if you move the cursor in Tool B, the location change is reflected in Tool A.

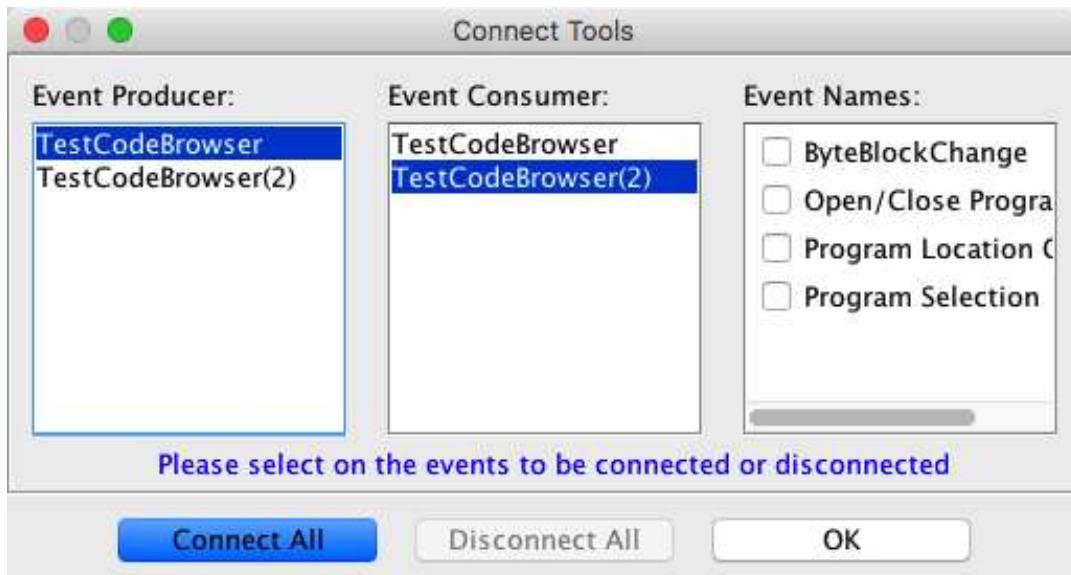


Drag and drop tool icons is the quickest way to connect two tools.

## Manual Tool Connection

### [Connecting Tools](#)

You manually connect tools by selecting the **Tools** → **Connect Tools...** option from the [Ghidra Project Window](#).



- To connect two tools in one direction for a specific event,

1. Select a tool in the *Event Producer* list (for example Tool A).
2. Select a tool in the *Event Consumer* list (for example Tool B).
3. Select the check box for the event of interest in the *Event Names* list.

Only the consumer tool will respond to the event generated by the producer tool.

- To connect two tools in both directions for a specific event,

1. Connect the tools in one direction as described above.
2. Select Tool B as the event producer and Tool A as the event consumer.
3. Select the check box for the event of interest in the *Event Names* list.



When you select a check box for an event, the connection is established immediately. The **OK** button dismisses the dialog, as does selecting the **X** button. There is no "cancel" associated with connecting tools.



The manual connection is useful only if you want to control specific events between the tools. From the *Connect Tools* dialog, the easiest way to connect all tools for all events is to click on the **Connect All** button. This action will connect tools in both

directions, just as though you had dragged and dropped the tool icons to do the [automatic connection](#). Conversely, disconnect all tools for all events by clicking on the **Disconnect All** button.



### **Some Notes on Connecting Tools:**

1. Because of the basic premise of connections, connected Tools should have the same program [opened](#). For example, tracking program locations between two tools each showing a different program will probably not be very useful.
2. Tools running in different [workspaces](#) may be connected. Even though tools may not be visible because they are not in your active workspace, they can still be connected to tools that are running in other workspaces.

## **Disconnect Tools**

- To disconnect tools for a specific event,
  1. Select a tool in the *Event Producer* list (for example Tool A).
  2. Select a tool in the *Event Consumer* list (for example Tool B).
  3. Turn off the check box for the event of interest.

The consumer tool will no longer respond to that event.

- To disconnect two tools in both directions for a specific event,

1. Disconnect the tools in one direction as described above.
2. Select Tool B as the event producer and Tool A as the event consumer.
3. Turn off the check box for the event of interest in the *Event Names* list.

- To disconnect all tools for all events, click on the **Disconnect All** button.



When you deselect a check box for an event, the connection is terminated immediately.

# Edit Plugin Path

The Plugin Path is a preference that indicates where Ghidra should search for Java classes outside of the standard installation locations. User developed Java classes can be used to extend Ghidra with additional [plugins](#) and [data types](#).



All plugins discovered by Ghidra can be displayed in the [Configure](#) dialog for all tools. All known data types are shown in the [Data Type Manager](#) display.



The *User Plugin Paths* list shows the paths in the order to be searched. Each path is either a directory path or a jar file path. If the path is a

directory, then only class files in that directory will be used (not jar files within that directory). If the path is a jar file, then classes within the jar file will be used.

The *User Plugin Jar Directory* shows the directory that contains jar files to search.



In addition to the above, Ghidra also searches in the installation directory, in the `<home>/ .ghidra/.ghidra-<version>/plugins` directory, if it exists.

The directories noted above, as well as any found jar files, are added to Ghidra's classpath. The search order of these paths is:

1. Jar files in *User Plugin Jar Directory* (Plugin Path preference)
2. Jar files in the Ghidra plugins installation directory
3. *User Plugin Paths* from the Plugin Paths preference

## Editing Plugin Paths



After you make a change to the plugin path, you must restart Ghidra to see the effects.

### Add a Plugin Path

To add a Plugin Path,

1. From the Ghidra Project Window, select **Edit → Plugin Path...**
2. The *Edit Plugin Path* dialog is displayed; in the *Directory or Jar File Name* field
  - Select the **Add Jar...** or **Add Dir...** button to choose either a jar file or directory from the file system.
  - Locate and select the appropriate jar file or directory within the file chooser dialog.
  - Select the **Add Jar**, or **Add Dir** button within the file chooser dialog.
3. Select the **Apply** or **OK** button from the *Edit*

*Plugin Path* dialog.

- **Apply** applies the changes and leaves the dialog up.
- **OK** applies the changes and dismisses the dialog.

## Change the Search Order

To change the search order of the paths within the User Plugin Path list,

1. Select a path from the User Plugin Paths list.
2. Select the  button to move the path up in the list; select the  to move the path down in the list.



The search order is important when you have different versions of a plugin in different jar files. The first class that is loaded is the one that you will be using when you run Ghidra.

## Set the User Plugin Jar Directory

- To set the User Plugin Jar Directory,
  1. Enter the absolute directory path in the *User Plugin Jar Directory* field, OR click on the ... button to choose a directory from the file system.
  2. Select the **Apply** or **OK** button.
    - **Apply** applies the changes and leaves the dialog up.
    - **OK** applies the changes and dismisses the dialog.

## Remove Paths

- To Remove an existing jar from the Plugin Path,

1. From the Ghidra Project Window, select **Edit**→  
**Plugin Path...**
2. Select a **User Plugin Path**.
3. **Click the Remove Button.**
4. **Click Apply or OK.**

● To Remove the User Plugin Jar Directory from the Plugin Path,

1. Clear the *User Plugin Jar Directory* field.
2. Select the **Apply** or **OK** button.

- **Apply** applies the changes and leaves the dialog up.
- **OK** applies the changes and dismisses the dialog.



When you click on the **Apply** or **OK** button, your preferencesfile in your <home>/.ghidra/.ghidra-<version> folder is updated immediately.



If you have a tool that was built with Plugins that came from the paths that you removed, you will get an error message listing each Plugin that could not be found when you re-open the project or when you launch that tool.

## Related Topics:

- [Ghidra Project Window](#)
- [Configure Ghidra Project Window](#)
- [Configure Tool](#)
- [Manage Data Types](#)
- [Built in Data Types](#)
- [Plugins](#)



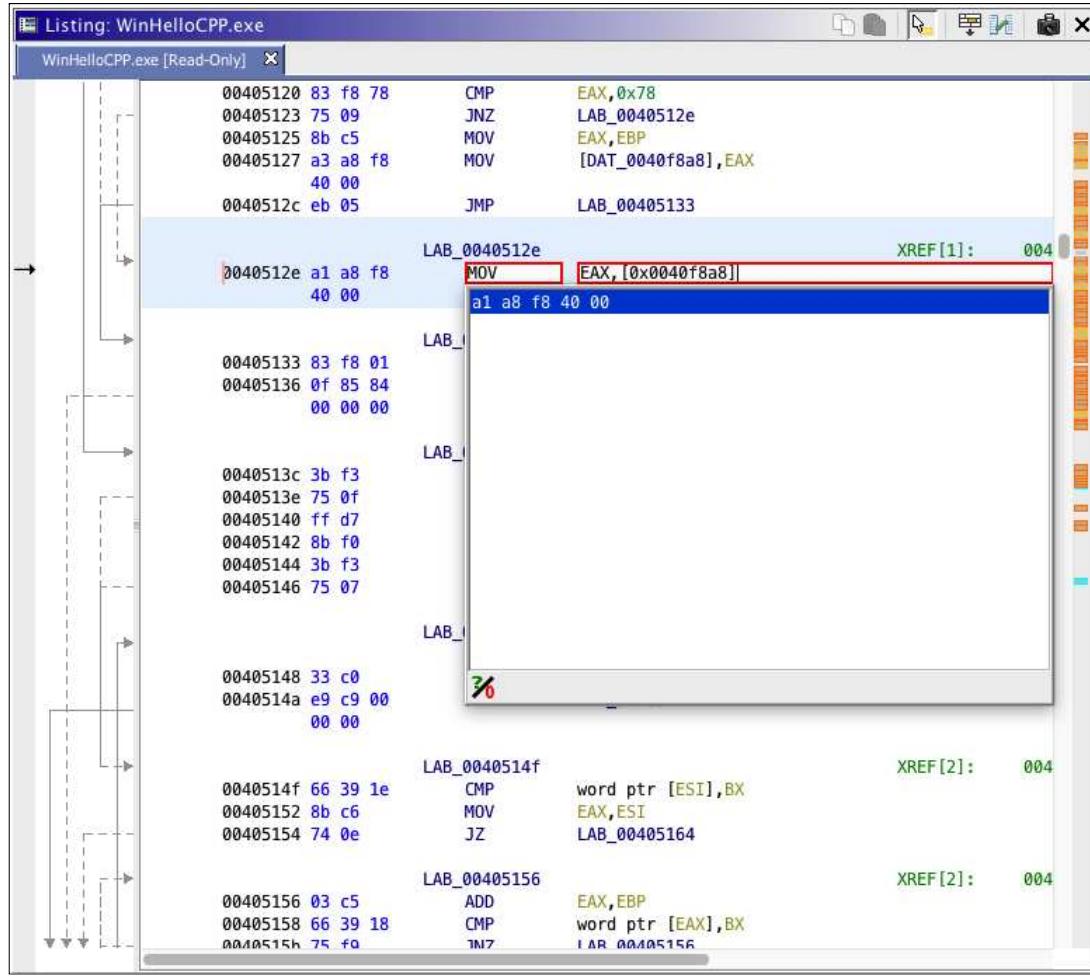
# ***Ghidra Functionality***

This section provides help on specific functionality that has been included in this installation.

Ghidra is a highly extensible application and the set of capabilities will change depending on which optional modules have been included in the installation . Options modules can install new sections into this part of the table of contents.

# Assembler

The **Patch Instruction** action will allow you to edit the current assembly instruction in the listing. The first time you use the action, it may take a moment to prepare the assembler for your processor. You can then edit the text of the instruction, optionally replacing it altogether. As you edit, a content assists will provide completion suggestions. It will present a list of byte sequences when the text comprises a complete instruction. Activating a completion suggestion will input that text at your cursor. Activating a byte sequence will complete the action, replacing the instruction at your cursor. Pressing ESC or clicking outside of the assembly editor will cancel the action.



Assembly Editor

To edit assembly, select **Patch Instruction** from the [Listing View](#) context menu or press CTRL-SHIFT-G on the instruction to modify. Click the plus button below the content assist to exhaust the undefined bits.

Ghidra's assembler is based on the same SLEIGH modeling that powers the disassembler. This offers some nice benefits:

- There is no need for an external tool chain.
- The assembler and disassembler share the same mnemonic syntax.
- Most Ghidra-supported processors are also supported by the assembler.
- Processors added to Ghidra automatically get an assembler.

Keep in mind, the above list is in an ideal world. We are in the process of improving the assembly engine and processor modules to eventually support assembly for all of Ghidra's processors. In the meantime, we test several popular processors and assign a performance rating to each. The possible ratings are:

1. **Platinum:** Our automated tests did not find any errors. This offers the best possible user experience.
2. **Gold:** You will rarely encounter an error. You will find it very useful.
3. **Silver:** You may occasionally encounter an error, but the assembler is still usable. You will likely find it useful with occasional frustration.
4. **Bronze:** You are likely to encounter errors, but there are enough working instructions that the assembler is useful. You may find it

useful, but it will probably be frustrating.

**5. Poor:** You are likely to encounter severe errors, and there are few instructions that assemble. You may or may not find it useful, but we consider it unusable.

**6. Unrated:** The processor is not tested, or the test failed before a rating could be assigned. You might get lucky, but don't count on it.

As of this release, our tested processors fall under Platinum, Gold, or Poor.

- Platinum

- 68000:BE:32:default
- AARCH64:BE:64:v8A
- AARCH64:LE:64:v8A
- ARM:BE:32:v7
- ARM:LE:32:v7
- avr8:LE:16:extended
- MIPS:BE:32:default
- MIPS:BE:32:micro
- MIPS:BE:32:R6
- MIPS:BE:64:64-32addr
- MIPS:BE:64:R6
- MIPS:BE:64:default
- pa-risc:BE:32:default
- PowerPC:BE:32:default
- PowerPC:BE:64:A2-32addr
- PowerPC:BE:64:A2ALT-32addr
- PowerPC:BE:64:default
- sparc:BE:32:default
- sparc:BE:64:default
- SuperH4:BE:32:default
- SuperH4:LE:32:default
- TI\_MSP430X:LE:32:default
- x86:LE:32:default

- Gold

- x86:LE:64:default

- Bronze

- avr32:BE:32:default

- Poor

- dsPIC30F:LE:24:default

Provided by: *Assembler* plugin

Related Topics:

- [Listing View](#)

# The Byte Viewer

The Byte Viewer displays bytes in memory in various formats, e.g., Hex, Ascii, Octal, etc. The figure below shows the Byte Viewer plugin in a separate window from the [default tool](#), the Code Browser.

Addresses	Hex
00400000	4d 5a 90 00 03 00 00 00 00 04 00 00 00 ff ff 00 00
00400010	b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00
00400040	0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
00400050	69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
00400060	74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
00400070	6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 00 00
00400080	15 e3 c7 92 51 82 a9 c1 51 82 a9 c1 51 82 a9 c1
00400090	76 44 d4 c1 40 82 a9 c1 76 44 c4 c1 0e 82 a9 c1
004000a0	92 8d f4 c1 56 82 a9 c1 51 82 a8 c1 3d 82 a9 c1
004000b0	76 44 c7 c1 7d 82 a9 c1 76 44 d5 c1 50 82 a9 c1
004000c0	76 44 d1 c1 50 82 a9 c1 52 69 63 68 51 82 a9 c1
004000d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000e0	00 00 00 00 00 00 00 00 50 45 00 00 4c 01 04 00

Start: 00400000    End: 0041cedf    Offset: 00000000    Insertion: 00400000

To show the Byte Viewer, select the icon, , on the Code Browser toolbar, OR, choose the **Window ➔ Bytes: ...** menu.

The following paragraphs describe the Byte Viewer.

## Data Formats

This section describes the formats that Ghidra provides by default. Each format is an instance of a `DataFormatModel` interface, so any [new formats that you provide](#) will automatically show up in the *Byte Viewer Options* dialog that lists the data formats that may be added to your view. To add or remove a data format view from the tool, press the  icon to bring up the *Byte Viewer Options* dialog. Select the formats that you want and press the **OK** button.

### Hex

The Hex view shows each byte as a two character hex value. [Change the group size](#) for the Hex format to show the bytes grouped in that size. When you add the Byte Viewer plugin to a tool and then open a program, the Hex view is automatically displayed by default.

This view supports byte [editing](#).

### Ascii

The Ascii view shows each byte as its equivalent Ascii character. For those bytes that do not represent an Ascii character, the format shows it as a tic (".").

This view supports byte [editing](#).

## Address

The Address view displays a tic (".") for all bytes whose formed address does not fall within the range of memory for the program. For those addresses that can be formed and are in memory, the view shows the symbol, So if you go to that address in the [Code Browser](#), and [make a Pointer data type](#), the address pointed to is in memory. Conversely, if you go to a "tic" address in the Code Browser and make a pointer, the address pointed to is not in memory (the operand is rendered in red).

This view does not support [editing](#).

## Disassembled

The Disassemble view shows a "box" () symbol for each address that has undefined bytes. For those addresses that are [instructions](#) or [defined data](#), the view shows a tic ("."). With this view, you can easily see what areas of the program have been disassembled.

This view does not support [editing](#).

## HexInteger

This format shows four byte numbers represented as an eight digit hex number.

This view supports [editing](#). When a byte is changed, all four bytes associated with this address are rendered in [red](#) to denote the change.

## Integer

This view shows four byte numbers represented in decimal format.

This view does not support [editing](#).

## Octal

The octal view shows each byte as a three character octal value.

This view supports [editing](#).

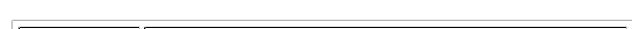
## Binary

The binary view shows each byte as an eight character binary value.

This view supports [editing](#).

## Status Fields

The labels below the scroll pane that contains the views shows the following information:



Start	The minimum address of Memory
End	The maximum address of Memory
Offset	Displayed in decimal, the number of bytes added to each block of memory that is being displayed. This number is calculated when you set the <a href="#">alignment address</a> or the number of bytes per line.
Insertion	The address of your current cursor location

## Editing Memory

To enable byte editing,

1. Toggle the Enable/Disable Edit toolbar button  so that it appears pushed-in.
2. Click in a view that supports editing, e.g., Hex or Ascii
3. The cursor changes to red to indicate that this view can be edited.

Changing bytes is allowed only if your cursor is at an address that does not contain an instruction. If you attempt to change a byte of an instruction, an "editing not allowed" message is displayed in the status area of the tool.

Changed bytes are rendered in **red**. This color can be changed via the [Byte Viewer Edit Options](#) by double-clicking on the [Edit Color](#) field.

Undo the edit by hitting the Undo button () on the tool. The byte reverts to its original value. Redo your edit by hitting the Redo button () .

To turn off byte editing, click the Enable/Disable Edit toolbar button  so that it no longer appears pushed-in.

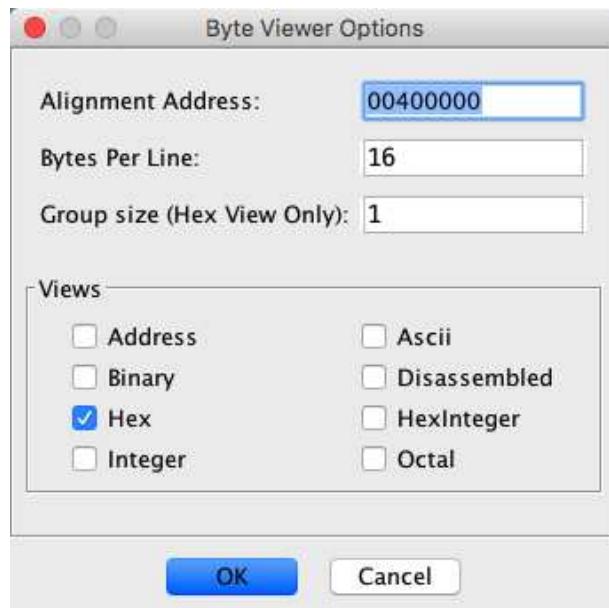
 If you have two Byte Viewers running, you can [connect](#) the two tools for the "Byte Block Edit" event so that when you make changes in one Byte Viewer, the other will reflect those changes in red.

## Cursor Colors

The format view that currently has focus shows its cursor in **magenta**. (Cursor colors can be changed via the [Options](#) dialog) If the byte editing is enabled and the view that is in focus supports editing, then the cursor is **red**.

## Byte Viewer Options:

The *Byte Viewer Options* dialog can be used to add and remove views, set the *AlignmentAddress*, set the number of *bytes per line*, and set the *group size* to be used by the *hex* view. To launch the *Byte Viewer Options* dialog, press the  icon on the Byte Viewer toolbar.



### Alignment Address

The alignment address specifies what address should appear in column 0. Any address can be specified, but the address will be normalized to be near the program's minimum address. This enables you to view bytes in an offset manner and to identify patterns in the bytes. Changing the alignment address affects the [offset](#), which is the column that would display the bytes for address 0 if it existed. The offset is affected by both the alignment address and the bytes per line. The offset is displayed as a label below the scroll pane containing the views.



Sometimes you might see a byte pattern such that you want all the bytes to line up in the first column of the display. Consider the cursor position in the image below. If you want to see the fourth column of bytes (values of 00) to appear in the first column, you would enter an alignment address of 0040b003, as indicated by your cursor position.

Addresses	Hex
0040b000	c2 d6 00 00 b6 d6 00 00 cc d6 00 00 00 00 00 00 00 00
0040b010	c2 db 00 00 b2 db 00 00 a2 db 00 00 8c db 00 00
0040b020	7c db 00 00 6e db 00 00 5c db 00 00 4a db 00 00
0040b030	02 d9 00 00 e2 d6 00 00 ee d6 00 00 00 d7 00 00
0040b040	0c d7 00 00 1c d7 00 00 28 d7 00 00 3a d7 00 00
0040b050	4c d7 00 00 64 d7 00 00 7c d7 00 00 90 d7 00 00
0040b060	a4 d7 00 00 c0 d7 00 00 de d7 00 00 f2 d7 00 00
0040b070	04 d8 00 00 18 d8 00 00 26 d8 00 00 32 d8 00 00
0040b080	40 d8 00 00 4a d8 00 00 62 d8 00 00 72 d8 00 00
0040b090	88 d8 00 00 98 d8 00 00 b0 d8 00 00 c2 d8 00 00
0040b0a0	d0 d8 00 00 dc d8 00 00 ec d8 00 00 1c d9 00 00
0040b0b0	34 d9 00 00 4e d9 00 00 64 d9 00 00 7e d9 00 00
0040b0c0	90 d9 00 00 9e d9 00 00 b6 d9 00 00 c4 d9 00 00
0040b0d0	d2 d9 00 00 e0 d9 00 00 fa d9 00 00 0a da 00 00
0040b0e0	20 da 00 00 3a da 00 00 42 da 00 00 4e da 00 00

Start: 00400000 End: 0041cedf Offset: 00000000 Insertion: 0040b003

The result of setting the alignment address to 0040b003 is shown below. The calculated offset is 13, the number of bytes added to each memory block to create a new alignment. The first line of the display shows the "remainder" bytes of 16 (bytes per

line) divided by 13, the offset. If you were to put your cursor on the starting byte of the first line, you would see that your insertion point is 0040b000, in this example.

Bytes : WinHelloCPP.exe								X
Addresses	Hex							
0040b000	c2 d6 00							
0040b003	00 b6 d6 00 00 cc d6 00 00 00 00 00 c2 db 00							
0040b013	00 b2 db 00 00 a2 db 00 00 8c db 00 00 7c db 00							
0040b023	00 6e db 00 00 5c db 00 00 4a db 00 00 02 d9 00							
0040b033	00 e2 d6 00 00 ee d6 00 00 00 d7 00 00 0c d7 00							
0040b043	00 1c d7 00 00 28 d7 00 00 3a d7 00 00 4c d7 00							
0040b053	00 64 d7 00 00 7c d7 00 00 90 d7 00 00 a4 d7 00							
0040b063	00 c0 d7 00 00 de d7 00 00 f2 d7 00 00 04 d8 00							
0040b073	00 18 d8 00 00 26 d8 00 00 32 d8 00 00 40 d8 00							
0040b083	00 4a d8 00 00 62 d8 00 00 72 d8 00 00 88 d8 00							
0040b093	00 98 d8 00 00 b0 d8 00 00 c2 d8 00 00 d0 d8 00							
0040b0a3	00 dc d8 00 00 ec d8 00 00 1c d9 00 00 34 d9 00							
0040b0b3	00 4e d9 00 00 64 d9 00 00 7e d9 00 00 90 d9 00							
0040b0c3	00 9e d9 00 00 b6 d9 00 00 c4 d9 00 00 d2 d9 00							
0040b0d3	00 e0 d9 00 00 fa d9 00 00 0a da 00 00 20 da 00							
Start: 00400000 End: 0041cedf Offset: 13 Insertion: 0040b003								

### Set Bytes Per Line

The bytes per line indicates how many bytes are displayed in one line in a view. The default value is 16.



All formats shown must be able to support the new value. For example, since the HexInteger and Integer formats show bytes in groups of four, the bytes per line must be a multiple of four. If a selected format cannot support a value for the bytes per line, an error message will appear and the **OK** button will be disabled.

### Set Group Size

The group size is the number of bytes that the Hex view shows as a "unit." For example, a group size of two means to show two bytes grouped together with no spaces.

### View Selection

Each potential view is listed as a checkbox. Select the checkboxes corresponding to the views to be shown. Red text indicates a view cannot be displayed since it doesn't support the specified number of bytes per line.

### Reorder Views

The various views in the ByteViewer can be reordered by dragging the view header to the left or right of its current position. The view positions are swapped.

### Writing Your Own Format Plugin

To supply your own format to be added to the list of views displayed in the Byte Viewer,

1. Write an implementation of the `ghidra.app.plugin.core.format.DataFormatModel` interface, which determines the format of how the bytes should be represented.
2. Edit your [Plugin path](#) to include your class files if you are running Ghidra in production mode versus development mode; in development mode, you will have to add your class files to your classpath in your development environment.
3. Restart Ghidra.

Provided by: *Byte Viewer Plugin*

Related Topics:

- [Byte Viewer Options](#)
- [Pointer data types](#)
- [Code Browser](#)
- [Configure Tool](#)
- [Select Bytes](#)

# Configuration Options for the Byte Viewer

The *Byte Viewer* panel in the Options dialog for the Byte Viewer tool has configurable items for colors and font. The Byte Viewer options determines the colors, font, and header configuration. To view the options, select **Edit**→**Tool Options** and then select the *Byte Viewer* node in the Options tree.

## Colors and Font

The Byte Viewer displays can display each of the following in a user-defined color and font.

Block Separator Color	Used to render gaps in memory, i.e., separators between memory blocks
Current View Cursor Color	Used in the view that has focus
Cursor Color	Used in the views that do not have focus
Edit Cursor Color	Used to indicate changed bytes in memory; this color is used as the cursor color in those views that support editing
Font	Font specified for all views
Highlight Cursor Line	Highlights the line containing the cursor

Highlight Cursor Line Color	The color of the highlight for the line containing the cursor
Non-Focus Cursor Color	The cursor color when the Byte Viewer is not focused

- To change the color,

1. From the Byte Viewer menu select **Edit → Tool Options**.
2. Select the *Byte Viewer* node; the panel of options appears to the right of the tree.
3. Double-click on the color bar for the color option of choice.
4. A color chooser is displayed; select a different color, and click **OK**.
5. Click **OK** or **Apply** on the Options dialog.

- To change the font,

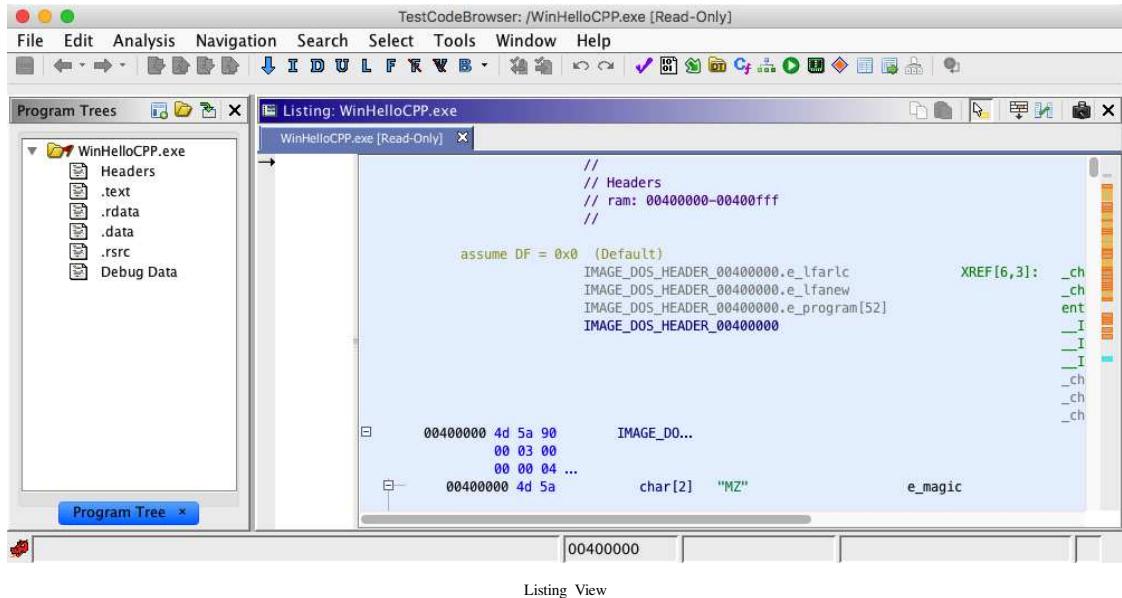
1. From the Byte Viewer menu select **Edit → Options**.
2. Select the *Byte Viewer* node; the panel of options appears to the right of the tree.
3. Click on the *Font* button.
4. A font editor is displayed; select a font, size, and/or style, and click **OK**.
5. Click **OK** or **Apply** on the Options dialog.

#### Related Topics:

- [Tool Options](#)
- [Key Bindings](#)
- [Byte Viewer](#)

## Listing View

The Listing View is the main windows for displaying and working with a program's instruction and data.



### Main Display

In the Code Browser tool displayed above, the Listing is shown to the right of the [Program Tree](#). The Listing is currently showing a snippet of code for "WinHelloCPP.exe". Code is made up of various elements such as addresses, bytes, mnemonics, and operands. The Listing uses *fields* to display these elements. The overall layout of the Listing can be changed by adjusting the size and position of the fields using the [Browser Field Formatter](#).

### The View

The Listing can either display an entire program or a subset of a program. The *view* determines the set of addresses that are displayable in the Listing. By default, the view contains the entire program, but other Ghidra components can restrict the view to some subset of the program. For example, the [Program Tree](#) can be used to restrict the view to a module or fragment. If the view is larger than the screen size, a vertical scroll bar will appear that allows the display to be scrolled anywhere within the view.

### Cursor

The Listing maintains a cursor location. Even though the Listing never actually changes a program, it plays an important role for plugins that do. Many Ghidra plugins operate at a specific address or even on a specific field on an instruction at an address. These plugins use the Listing's cursor position to determine the appropriate "program location".

The cursor can be moved by using any of the "arrow" keys or by clicking the mouse over any appropriate location.

### Selection

The Listing also maintains a selection (shown using a green shaded area). Some plugins operate on a particular address. Other plugins can operate over a range of addresses. For these plugins, the selection is used to determine the range of address on which to operate.

A selection can be set by either dragging the mouse while holding down the left mouse button or by pressing the <shift> key while moving the cursor with the "arrow" keys.

Pressing the <Ctrl> key down while clicking the mouse will either add to the selection if that line is not already in the selection or it will remove that line from the selection.

Clicking the mouse anywhere (without dragging) will cause the selection to go away.

### Highlight

Since the selection is transient (it goes away easily), the Listing also has the concept of a highlight. The highlight is similar to the selection, but it stays around until it is explicitly cleared. To create a highlight, first create a selection and then use the [Highlight](#) menu to convert the selection to a highlight. Later the highlight can be converted back into a selection using the same menu. At that point an action can be applied to the selection.



Highlights are not saved across program sessions. To achieve a lasting color effect, you can [color the Listing background](#).

### Background Color

See [Listing Background Colors](#)

## Navigation

The Listing provides built-in navigation functionality for addresses and labels. Double-click on any address, label or reference markup within the Listing and it will attempt to reposition to that location if it exists in the program. If the address exists in the program but not in the current view, the view is expanded to include the address. If the location refers to an external location and it is associated with another program, that program may be opened and positioned within the CodeBrowser. See the tool [Edit Tool Options](#) dialog for control over certain Navigation behaviors.

In the XRef field, sometimes there are too many addresses to display so the field will display "[more]" to indicate that one or more cross-reference addresses are not shown.



Double-clicking on the "[more]" text will cause a [Location References Dialog](#) to appear. Also, double-clicking on the XREF header text (`XREF [n]`) will too show this dialog.

This dialog lists all the Xref addresses, any labels that are at that address and a preview of the instruction at that address. Clicking on any row in the table will cause the browser to navigate to that address. Double-clicking will navigate and dismiss the dialog.

## Keyboard Controls

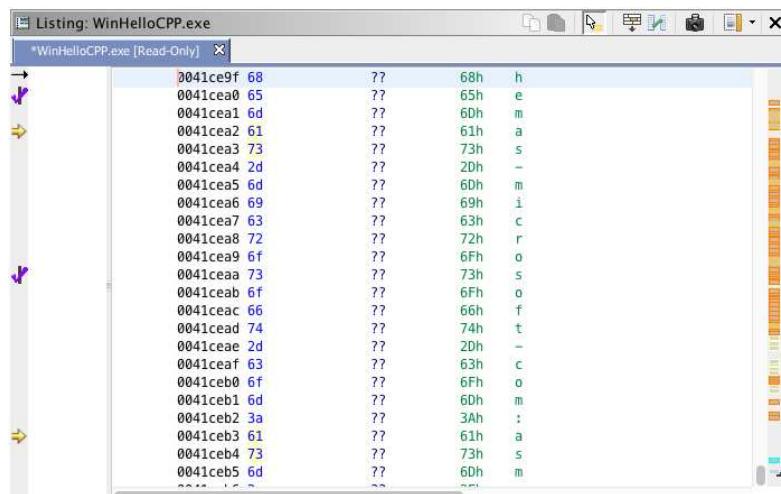
The following key mappings are used by the Listing:

<Home>	Move the cursor to the beginning of the line.
<End>	Move the cursor to the end of the line.
<Ctrl><Home>	Move the cursor and the display to the top of the program.
<Ctrl><End>	Move the cursor and the display to the bottom of the program.
<Page Up>	Move the display to the previous page.
<Page Down>	Move the display to the next page.
<Up Arrow>	Move the cursor up one line, scroll the display if necessary.
<Down Arrow>	Move the cursor down one line, scroll the display if necessary.
<Left Arrow>	Move the cursor to the left one position. If at the beginning of a line, move the cursor to the last position of the previous line.
<Right Arrow>	Move the cursor to the right one position. If at the end of a line, move the cursor to first position of the next line.
<Shift>	Press shift when using the mouse scroll wheel to perform horizontal scrolling. This will only work when the horizontal scroll bar is visible.

You can disable horizontal scrolling in the Listing (and all other Field Panels) via the Tool Options at `Edit → Tool Options... → Listing Fields → Mouse → Horizontal Scrolling`

## Markers

Markers are used to indicate special locations within a program. These location can either be specified by the user (Bookmarks) or by various Ghidra plugins (Search, Analysis, Changes, etc). There are two types of markers: Margin Markers and Navigation Markers.



## Markers

### Margin Marker

Margin Markers appear on the left side of the Listing and are used to indicate locations within the currently displayed code. There are two types of Margin Markers – **Point Markers** and **Area Markers**. **Point Markers** are used to indicate individual addresses (Ex: bookmarks, search results, breakpoints, etc) and are displayed using icons. **Area Markers** are used to indicate a range of addresses (Ex: areas in the program that have been changed) and are displayed using a colored vertical line.

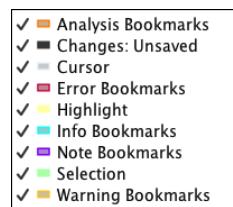
### Navigation Marker

Navigation Markers appear on the right side of the listing and are used to indicate locations within the overall view. These markers provide an overview of all the markers and an easy way to navigate to them. Navigation Markers are displayed as colored bars where each color represents a different type of marker. To see the color assignments bring up the [Marker Popup Menu](#) as described below.

To navigate the browser to the actual location of a marker, left-click on the marker. The browser will be repositioned to display the location represented by that marker.

The Navigation Marker area is divided into two vertical areas. The left area is used to display Point Markers while the right area displays Area Markers. This prevents the Point markers from being hidden by the Area markers.

You can control what is displayed in the Navigation Marker area by right-mouse clicking in the margin; a popup menu is displayed, as shown in the image below. Turn markers on and off by clicking on the appropriate menu item.



Marker Popup Menu

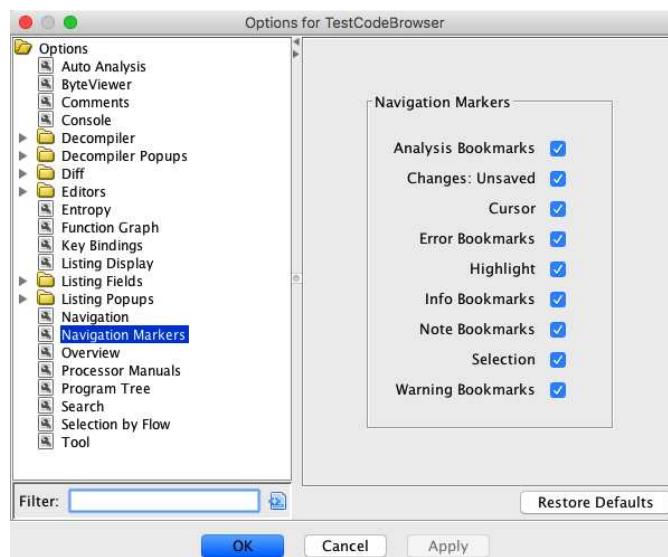
The markers shown in the Marker Popup Menu are described below:

Analysis Bookmarks	<a href="#">Bookmarks</a> inserted by the auto analysis process. Indicates information about where functions and <a href="#">address tables</a> were created.
Changes: Conflicting	Areas where you and others have made conflicting changes.
Changes: Latest Version	Areas where changes exist in the latest version made by another user.
Changes: Not Checked-In	Areas where you have made changes since your last check-out.
Changes: Unsaved	Areas where you have made changes to the Program.
Error Bookmarks	Areas where the disassembler created <a href="#">Error bookmarks</a> due to encountering bad instructions.
Highlight	Areas that are highlighted (non-transient selections).
Info Bookmarks	Bookmarks inserted by a plugin to indicate a location of interest.
Note Bookmarks	Bookmarks inserted by the user.
Register Values	Areas where the selected register has defined values. Appears only when the Register Manager Window is visible.
Selection	Areas that are selected (transient).



The marker popup menu is updated appropriately when some plugin adds/removes new marker types.

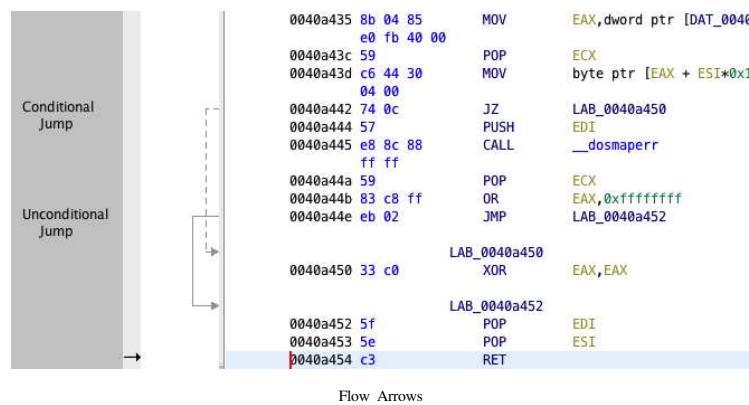
Another way to control the display of Navigation Markers is to set the options in Navigation Markers panel of the [Edit Options](#) dialog, as shown below. Select the *Navigation Markers* node in the Options tree. Click the appropriate checkbox to turn the marker on or off.



Navigation Marker Options

### Flow Arrows

The flow arrows graphically illustrate the flow of execution within a function. They appear as arrows on the left side of the Listing display indicating source and destinations for jumps. Conditional jumps are indicated by dashed lines; unconditional jumps are indicated by solid lines. Flow lines are bolded when the cursor is positioned at the source of the jump.



Flow Arrows

#### Selecting Flow Arrows

By default, flow arrows are only shown when either the start or end address is visible. Thus, as you scroll the Listing, you will see flow arrows appear and disappear, as the addresses the arrows are based upon appear and disappear in the listing.



You can select an arrow keep it from disappearing as you scroll the Listing. To select an arrow, simply click it with the mouse. Selected arrows appear green in color.

#### Disabling Flow Arrows

The panel displaying the flow arrows can be resized by dragging the border between it and the main code browser panel. To hide the flow arrows, simply drag the border until it is no longer visible. The arrows automatically disable themselves when they are no longer visible.

#### Navigating Flow Arrows

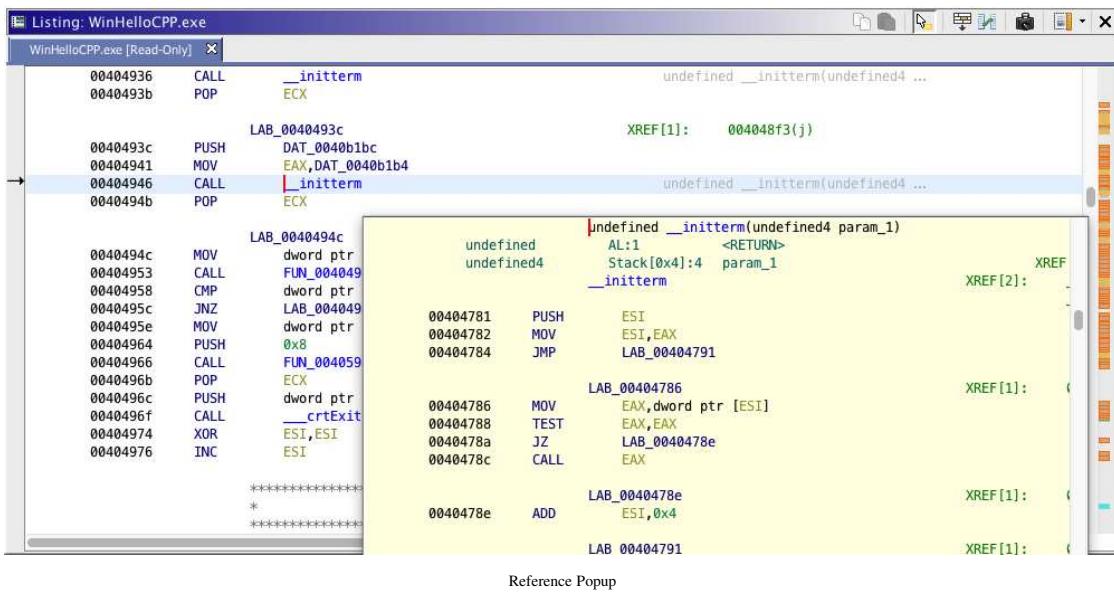
You can double-click a flow arrow to navigate to its end point. Also, if the cursor is at the end point address, then you can double-click to navigate to the start point of the arrow. Each double-click will go to the opposite end of the arrow, whenever the Listing cursor is at an endpoint.

#### Mouse Hover

The Listing includes the capability of displaying popup windows when the user hovers over a particular field. This occurs whenever a plugin has additional information that it wants to display about that field. The popup window disappears when the user moves the mouse off of the window or field. Some example popup windows that can be displayed: *Reference Popups*, *Truncated Text Popups*, and *Data Type Popups*.

#### Reference Popups

Reference popups are displayed whenever the mouse is hovered over a memory reference. A Reference popup containing a Listing window is displayed showing the code referred to by the reference. The Listing popup uses the exact same formatting and layout as the primary Listing except for the background color, which is displayed in a unique color to distinguish the popup. You can click and scroll within the Listing popup, however, you cannot follow references within the reference popup. The reference popup can be quite useful for quickly inspecting a series of calls without having to navigate within the Listing or to quickly examine data located in another part of the program's memory. The figure below shows an example of a reference popup.



Reference Popup

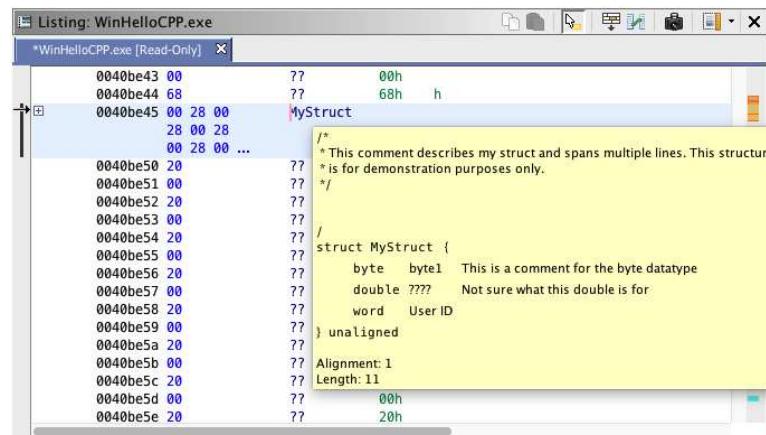
You can change the size of the popup can be changed by editing options. Choose **Edit** → **Tool Options...**, click on the *Listing Popups* node in the Options tree. Select the *Reference Code Viewer* node to display the height and width values of the popup. Edit the values and click on the **OK** or **Apply** button.

#### Truncated Text Popups

Truncated text popups are displayed whenever the mouse is hovered over a field where the size of the text exceeds the field size and is therefore truncated. A tooltip-like window is displayed showing the entire contents of the truncated field (provided the contents fits on the screen). This is particularly useful for long strings and data within a program that cannot be fully displayed. The figure below shows an example of a truncated text popup.

#### Data Type Popups

The data type popup is shown whenever the user hovers over a the mnemonic field where a data type has been set. The figure below shows an example of a data type popup.



Data Type Popup

#### Address Popup

The address popup is shown whenever the user hovers over an address. It shows the relationship between the hovered address and the base of memory and the containing memory block. For addresses in functions, the function offset is also shown; for addresses within a complex data (structure, array, etc.), the offset from the base of that data is shown. Also, if the byte value for the address can be traced back to the original imported file, then the filename and offset for that location is displayed

#### Function Name Popup

Displays the hovered symbol's parent namespace for symbols that are inside of functions.

#### Scalar Popup

Displays the hovered scalars as 1-, 2-, 4-, and 8-byte values, each in decimal, hexadecimal, and as ASCII character sequences.

### Disabling Mouse Hover

Because the mouse hover popup windows can sometimes get in the way, a mechanism for quickly disabling them is included. Simply click the icon illustrated above to toggle mouse hover mode. This icon appears in the top right corner of the Listing's toolbar whenever a mouse hover plugin is loaded. When clicked, the icon will change to indicate the new state.

Clicking the icon has no effect when all of the hover services are disable in Ghidra's options. A reminder message will be shown in the event that you attempt to enable popups from this icon and all hover services are disabled.



**Mouse Hover Enabled**

**Mouse Hover Disabled**

### Opening/Closing Structures and Arrays

Structures and arrays consist of smaller component data types. By default, the contents of structures and arrays are not displayed. There is a Open/Close control (+/-) that can be used to display or hide the contents of structures and arrays.

To see the contents of a Structure or Array, select on the + icon.

Closed	<table border="1"> <tr><td>0040be40 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be41 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be42 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be43 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be44 68</td><td>??</td><td>68h h</td></tr> <tr><td>0040be45 00 28 00</td><td>MyStruct</td><td></td></tr> <tr><td>  28 00 28</td><td></td><td></td></tr> <tr><td>  00 28 00 ...</td><td></td><td></td></tr> <tr><td>0040be50 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be51 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be52 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be53 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be54 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be55 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be56 20</td><td>??</td><td>20h</td></tr> </table>	0040be40 20	??	20h	0040be41 00	??	00h	0040be42 20	??	20h	0040be43 00	??	00h	0040be44 68	??	68h h	0040be45 00 28 00	MyStruct		28 00 28			00 28 00 ...			0040be50 20	??	20h	0040be51 00	??	00h	0040be52 20	??	20h	0040be53 00	??	00h	0040be54 20	??	20h	0040be55 00	??	00h	0040be56 20	??	20h
0040be40 20	??	20h																																												
0040be41 00	??	00h																																												
0040be42 20	??	20h																																												
0040be43 00	??	00h																																												
0040be44 68	??	68h h																																												
0040be45 00 28 00	MyStruct																																													
28 00 28																																														
00 28 00 ...																																														
0040be50 20	??	20h																																												
0040be51 00	??	00h																																												
0040be52 20	??	20h																																												
0040be53 00	??	00h																																												
0040be54 20	??	20h																																												
0040be55 00	??	00h																																												
0040be56 20	??	20h																																												

To hide the contents of a Structure or Array, select on the -icon.

Open	<table border="1"> <tr><td>0040be40 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be41 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be42 20</td><td>??</td><td>20h</td></tr> <tr><td>0040be43 00</td><td>??</td><td>00h</td></tr> <tr><td>0040be44 68</td><td>??</td><td>68h h</td></tr> <tr><td>0040be45 00 28 00</td><td>MyStruct</td><td></td></tr> <tr><td>  28 00 28</td><td></td><td></td></tr> <tr><td>  00 28 00 ...</td><td></td><td></td></tr> <tr><td>    0040be45 00</td><td>db</td><td>0h byte1</td></tr> <tr><td>    0040be46 28 00 28 00 28</td><td>double</td><td>6.675391337775248E-308 7???</td></tr> <tr><td>    00 28 00</td><td></td><td></td></tr> <tr><td>    0040be4e 20 00</td><td>dw</td><td>20h User ID</td></tr> <tr><td>    0040be50 20</td><td>??</td><td>20h</td></tr> <tr><td>    0040be51 00</td><td>??</td><td>00h</td></tr> <tr><td>    0040be52 20</td><td>??</td><td>20h</td></tr> <tr><td>    0040be53 00</td><td>??</td><td>00h</td></tr> <tr><td>    0040be54 20</td><td>??</td><td>20h</td></tr> <tr><td>    0040be55 00</td><td>??</td><td>00h</td></tr> <tr><td>    0040be56 20</td><td>??</td><td>20h</td></tr> </table>	0040be40 20	??	20h	0040be41 00	??	00h	0040be42 20	??	20h	0040be43 00	??	00h	0040be44 68	??	68h h	0040be45 00 28 00	MyStruct		28 00 28			00 28 00 ...			0040be45 00	db	0h byte1	0040be46 28 00 28 00 28	double	6.675391337775248E-308 7???	00 28 00			0040be4e 20 00	dw	20h User ID	0040be50 20	??	20h	0040be51 00	??	00h	0040be52 20	??	20h	0040be53 00	??	00h	0040be54 20	??	20h	0040be55 00	??	00h	0040be56 20	??	20h
0040be40 20	??	20h																																																								
0040be41 00	??	00h																																																								
0040be42 20	??	20h																																																								
0040be43 00	??	00h																																																								
0040be44 68	??	68h h																																																								
0040be45 00 28 00	MyStruct																																																									
28 00 28																																																										
00 28 00 ...																																																										
0040be45 00	db	0h byte1																																																								
0040be46 28 00 28 00 28	double	6.675391337775248E-308 7???																																																								
00 28 00																																																										
0040be4e 20 00	dw	20h User ID																																																								
0040be50 20	??	20h																																																								
0040be51 00	??	00h																																																								
0040be52 20	??	20h																																																								
0040be53 00	??	00h																																																								
0040be54 20	??	20h																																																								
0040be55 00	??	00h																																																								
0040be56 20	??	20h																																																								

Structures within Structures, Unions and Arrays can be opened in the same manner.

### Actions for Expanding/Collapsing Data

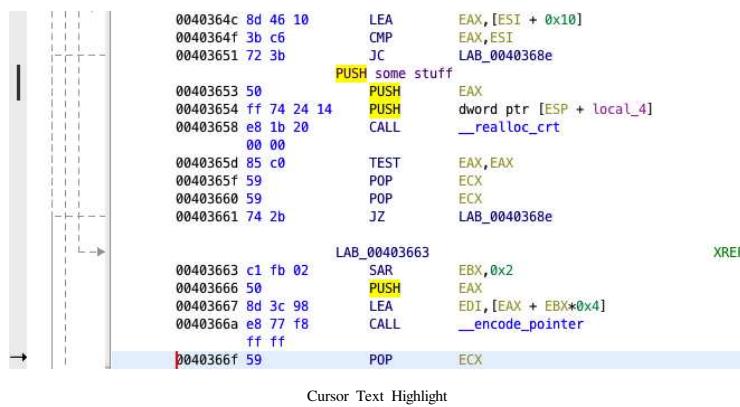
You may also expand and collapse data structures from the right-click popup menu via the **Expand All Data**, **Collapse All Data**, **Expand All Data In Selection**, **Collapse All Data In Selection**, and **Toggle Expand/Collapse Data** actions respectively. These actions are only available when the popup menu is activated under certain conditions. The actions are described in detail below:

- **Expand All Data** –This action will expand a data element and all its children recursively starting with the data at the cursor location. This action will only appear when the cursor is on a data that is expandable.
- **Collapse All Data** –This action will recursively collapse the outer most expandable data element containing the data at the cursor location. This action will only appear when the cursor is on an expandable data element or any of its child elements at any level.
- **Expand All Data In Selection** –This action will search the current selection and find all expandable data elements and then recursively expand each such data element and all its children. This action is available whenever there is a selection.
- **Collapse All Data In Selection** –This action will search the current selection and find all expandable data elements and then recursively collapse each such data element and all its children. This action is available whenever there is a selection.
- **Toggle Expand/Collapse Data** –This action will toggle the expanded/collapsed state of an expandable data element. If it is expanded, it will collapse it and if it is collapsed, it will expand it. Additionally, if the cursor is on a non-expandable data element that is contained in a parent expandable data element, then the parent element will be collapsed. This action is available whenever the cursor is on an expandable data element at any level or when the cursor is on a data element inside another data element.

The **Expand All Data** and **Collapse All Data** actions behave differently. The **Expand All Data** recursively opens the data starting at the location of the popup menu. The **Collapse All Data** action, on the other hand, always works on the **outermost** data structure, regardless of where in the containing structure the popup menu was activated.

### Cursor Text Highlight

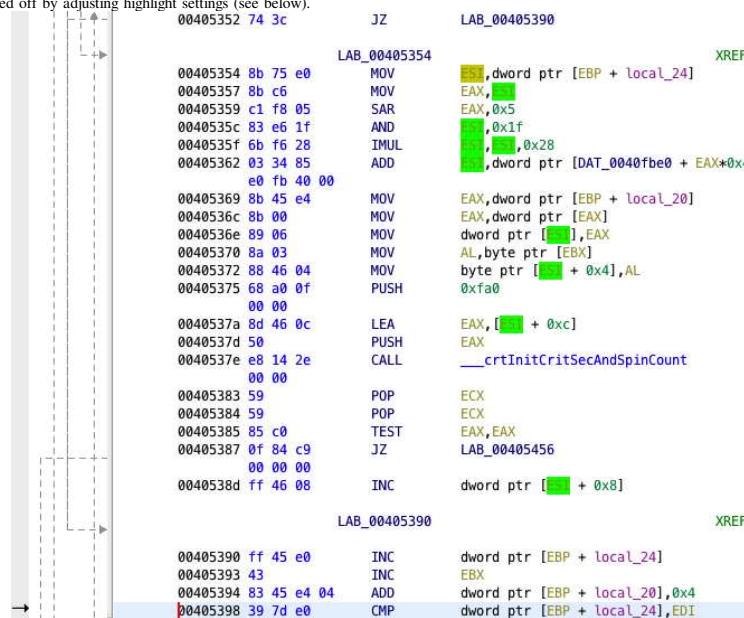
The Listing is comprised of many fields which are used to display the individual elements of a program (address, bytes, operand, etc). Clicking the middle mouse button on a field places the cursor in that field and causes the text under the cursor to become highlighted along with all other occurrences of that text. For example, in the figure below, the cursor is on the "PUSH" mnemonic at address 0x01004072. Notice that every "PUSH" on the screen (not just the mnemonics) is also highlighted. Specifically, the "PUSH" in the comment is highlighted. The Cursor Text Highlight makes no distinction as to what type of information is being highlighted. To determine matches that should be highlighted, a case-sensitive whole-word string compare is used.



This feature has additional functionality when used on registers. In this case, the highlight can be restricted to show the "Scope" of a register. The "Scope" of a register starts where the register is assigned a value and ends at the last use of the register before it is written with another value (when a register is both read and written, then it will be shown as a read).

When the middle mouse is clicked on a register, the scope of the register is computed. The register is highlighted from the point it is initially assigned a value (written) to the point it is last used that value. With "Scoping", three different colors are used. The read highlight color is used at locations where the register value is read (light green in the image below). A slightly darker color is used at the location where the register is written (dark yellow in the image below). All other matching registers outside of the current scope will be highlighted with the default highlight color (yellow in the image below).

This feature can be turned off by adjusting highlight settings (see below).



#### Default Settings

By default, the cursor text highlight is:

- Enabled
- Painted in yellow
- Only activated by using the middle mouse button
- Scoping turned on

#### Configuring

Use the [Tool Options](#) dialog to turn off the [Cursor Text Highlighting](#), as well as to configure the color and activating mouse button.

#### Configuring

There are many options for changing the appearance of the Listing. To change the arrangement of fields, use the [Code Browser Formatter](#). All other options are tool options. See [Tool Options](#) for a complete description of the options and how to change them.

## Errors in Fields

If an error ever occurs while the code browser is attempting to display information in a field, it will show an "Error" field instead. Double-click on the error field to display the **Log Details dialog**.

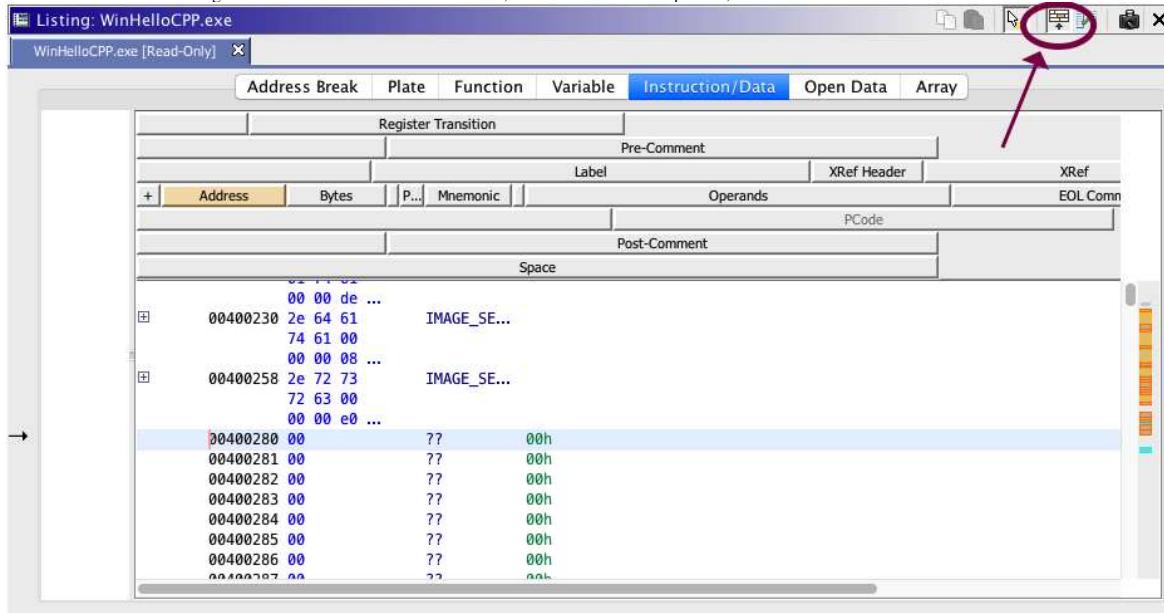
Provided by: *Code Browser Plugin*

Related Topics:

- [Code Browser Formatter](#)
- [Tool Options](#)
- [Program Tree](#)
- [Selection & Highlighting](#)

## Browser Field Formatter

The *Browser Field Formatter* is used to control how the fields in a listing are displayed by the Listing. The format specifies which fields are displayed, the order of fields, and the width of each field. The formatter is not normally displayed by default, but can be accessed at anytime by clicking on the  icon located in the Listing's toolbar. When the formatter is visible, the button becomes depressed, and can be clicked to hide the formatter.



The listing displays information at each address of a program. This information is broken down into the following categories:

Address Break	Separates non-consecutive addresses
Plate	Displays plate comments. Other comment types are included in the Instruction/Data element.
Function	Displays function signatures and function related attributes.
Variable	Displays information for the return, parameters and local variables associated with a function.
Instruction/Data	Displays information about instructions or data.
Open Data	Displays internals of a structure or array.

Each address has one or more of the above categories of information. Each category has a display format which can be configured. The format consists of multiple lines, each with some number of *field controllers*. A field controller manages the display of the corresponding field of information in the program. The listing displays fields in the same relative order, size and positioning as determined by the field controller. It is important to understand that the formats, regardless of how they appear, represent the layout of information for a single address.

Whenever the cursor is moved, the Brower Field Formatter automatically switches to the tab associated with the current cursor location. In addition, the field controller for the current cursor location is highlighted.

### Configuring the Format

#### Adding Fields

New fields can be added to the format by right-clicking on the Brower Field Formatter component and selecting **Add Field** → <name of field> from the popup menu. The new field will be inserted at the right-click point. Only the fields that have already been added to the format show up in the popup menu. If more than one field appears in the popup, an additional menu item, **Add Field** → All, is available.



The Spacer Field

A *spacer* field is used to take up space in the listing. Optionally, *spacer* fields can display some text (See [Set Text](#)). To add a *spacer* field, right click where a space is desired and select the **Add Field** → **Spacer** option from the popup menu. After adding the *spacer* field, adjust its size to take up more or less space.

#### Set Text

*Spacer* fields can optionally have an associated text value. Each spacer's text will be displayed at corresponding locations in the listing. To set text on a *spacer* field, right click on the field and choose the **Set Text** option from the popup menu.

Entering empty text into the Set Text dialog box will remove any text from the field thus returning the field to a blank spacer.

#### [Removing Fields](#)

Fields can be removed by right clicking on them and selecting **Remove Field** from the popup menu. All Fields can be removed by right clicking on the header and selecting the **Remove All Fields** option from the popup menu.

#### [Moving Fields](#)

Fields can be moved by dragging and dropping the corresponding field controller in the Browser Field Formatter. Field controllers can be dropped onto any line in the formatter. Since fields cannot overlap, dropping a field controller onto another field controller will cause the dropped field to appear either entirely before or directly after the other field, depending on which is closer to the drop point.

Once a field controller has been moved, the remaining field controllers always move as far to the left as possible to fill up any empty space. So if a field is moved off of a row, all the fields to its right, move left to fill in the empty space. If a field is dropped before another field, that field and all the fields to its right are moved to the right to make room for the new field. *Spacer* fields can be inserted before a field to move it further to the right.

#### [Disabling Fields](#)

Fields can be disabled by right clicking on them and selecting **Disable Field** from the popup menu. Disabled fields still take up space in the overall layout, but they don't display any information in the listing.

#### [Enable Field](#)

Disabled fields can be enabled by right clicking on them and selecting **Enable Field** from the popup menu. The field will then display information in the listing.

#### [Insert Row](#)

A new row for placing fields can be inserted by right clicking and selecting **Insert Row** from the popup menu. A new empty row will then be inserted at point where the mouse was clicked. Empty rows do not affect the listing because any rows (even rows that have fields) that don't have displayable information are suppressed from the listing.

#### [Remove Row](#)

Empty rows can be deleted by right clicking on the row and selecting the **Remove Row** option from the popup menu. The **Remove Row** option is not available if the row is not empty.

#### [Reset Format](#)

Reset the format of the currently displayed category to the default settings by right clicking on the Browser Field Formatter and select **Reset Format** on the popup.

#### [Reset All Formats](#)

Reset the formats for all categories to their default settings by right clicking on the Browser Field Formatter and select **Reset All Formats** on the popup.

### [Custom Formats](#)

A structure is normally displayed using the Instruction/Data category format. However, it is possible to create a custom format to display the structure. This custom format allow you to include members of the structure in its display.

#### [Create Custom Format](#)

To create a custom format, right click on the formatter when the cursor is over a structure and choose the **Create Custom Format** option. A new format will be created with the same format as the Instruction/Data format. In addition to all the standard fields, a displayable field is created for each member and is available in the **Add Field** submenu.

#### [Delete Custom Format](#)

To delete a custom format, select its tab, right click in the Browser Field Formatter and select the **Delete Custom Format** option. The format will revert to the general instruction format.

#### [Save Custom Format](#)

Newly created Custom Formats are temporary and must be saved to the program in order to be reused the next time the program is opened. To save a custom format, select its tab, right click in the Browser Field Formatter and select the **Save Custom Format** option.



The program must be saved after saving a custom format.

## Available Fields

The table below shows a listing of all fields along with a brief description of the type of information they display. The category from which the field can be accessed is also provided. In some instances a field can be accessed from more than one category.

Field Name	Category	Description
Address	Instruction/Data; Open Data	Displays addresses in the program
Bytes	Instruction/Data; Open Data	Displays the bytes that make up an instruction or data
EOL Comment	Instruction/Data; Open Data	Displays the end of line comment
Field Name	Open Data	Displays the name of the data fields in a structure
Function Call-Fixup	Function	Displays the name of the Call-Fixup associated with the function.
Function Purge	Function	Displays the number of bytes purged from the stack for a function
Function Repeatable Comment	Function	Displays the function's comment that will appear at all calls to that function
Function Signature	Function	Displays the full function prototype, including calling convention, function name, return data type and parameters. In addition the presence of the following function attributes will be indicated: inline, no-return, and thunk. If the function has varargs, the last parameter position will include "...".
Function Tags	Function	Displays all tags associated with this function
Label	Instruction/Data; Open Data	Displays all labels for an address
Memory Block Start	Plate	Displays information about the memory block
Mnemonic	Instruction/Data; Open Data	Displays the name of the instruction or data
Operands	Instruction/Data; Open Data	Displays the input to the instruction or the data value
Parallel	Instruction/Data	Displays a parallel indicator (e.g.,    ) to indicate that the current instruction executes in parallel with the previous instruction.
Plate Comment	Plate	Displays the block header comment
PCode	Instruction/Data; Open Data	Displays the micro-code for an instruction
Post-Comment	Instruction/Data; Open Data	Displays the comment following an instruction or data
Pre-Comment	Instruction/Data; Open Data	Displays the comment preceding an instruction or data
Register	Function	Displays the values of registers at the entry point of a function
Register Transition	Instruction/Data	Displays the values of registers at the points where the value transitions to a new value.
Separator	Address Break	Displays a "....." when there is a gap between addresses
Signature Source	Function	Indicates the source-type associated with the function signature (i.e., DEFAULT, ANALYSIS, IMPORTED, USER_DEFINED).
Space	Instruction/Data	Displays one or more blank lines as established by a plugin
Spacer	All	Used to separate other fields. Can optionally display static text
Stack Depth	Instruction/Data	Indicates the current stack-pointer offset relative to its state at the start of the associated function. The field is only displayed for instructions contained within a function. A bogus value indicates that the calculation failed to determine the stack depth (i.e., -7fffffff or 7fffffff).
Thunked-Function	Function	Shows the name of the thunked-function to for those functions designated as a "thunk" function (see Function Signature field).
Variable Comment	Variable	Displays the comment for a variable
Variable Name	Variable	Display the name of a variable
Variable Location	Variable	Displays the storage location associated with a the variable (e.g., register, stack, memory, unique-hash, etc.)
Variable Type	Variable	Displays the data type for variable
Variable XRef	Variable	Displays a list of addresses whose instructions reference a variable

Variable XRef Header	Variable	Displays the number of references and offcut references to a variable
XREF	Instruction/Data; Open Data	Displays a list of addresses whose instruction refers to this address.
XREF Header	Instruction/Data; Open Data	Displays the number of references and offcut references to this address.
+	Instruction/Data; Open Data	Opens and Closes structures and arrays.

Provided by: *Code Browser* plugin

**Related Topics:**

- [Code Browser](#)

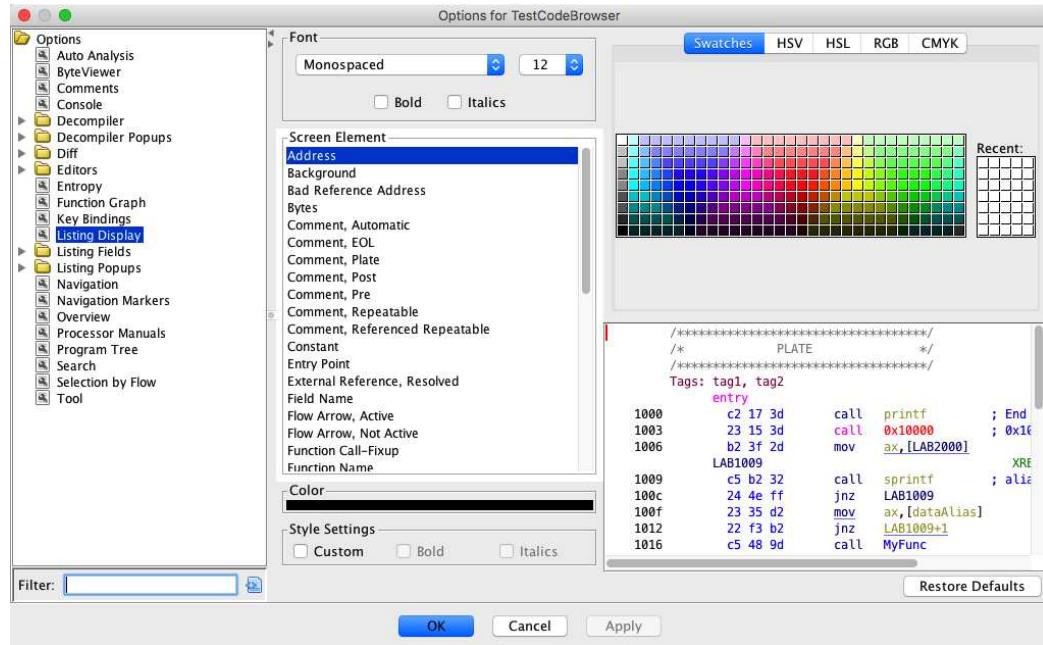
## Listing Options

Attributes of the Listing can be customized using the [Tool Options Dialog](#). These are the types of attributes that can be customized:

- [Colors and Fonts](#)
- [Fields, Highlights, and Selections](#)
- [Popups](#)

### Color and Fonts

To view or change colors and fonts for fields of the Listing, open the [Tool Options Dialog](#) and select the *Listing Display* node in the Options tree. The Listing Display panel will be displayed as shown below.



This panel is divided into the following sections:

#### Font

Sets the default font, size and style for all text fields in the listing. The font and size apply to all screen elements. The styles (bold/italic) will be used as the default for each screen element unless individually customized using the [Style Settings](#).

#### Screen Element

The *Screen Element* lists all the customizable items that can be displayed by the Listing. Some of the elements are fields such as the *Address* field. Others represent the *type* of information in a field. For example, the operand field may contain a label, scalar, address, or constant. Still others are *attributes* on the data in the fields. An example of this is label field. A label can be primary or local. Screen elements can be selected in order to change their associated color and font style.

The table below describes all the Screen Elements and indicates the fields in which they appear.

Screen Element	Description	Field
Address	Addresses	Address field
Background*	N/A	N/A
Bad Reference Address	Addresses or labels that are not in memory	Operand field
Bytes	Bytes	Bytes field
Comment, Automatic	System-generated comments	EOL comment field
Comment, EOL	End-of-line comments	EOL comment field
Comment, Plate	Block comments	Plate comment field
Comment, Post	Below-line comments	Post comment field
Comment, Pre	Above-line comments	Pre comment field
Comment, Repeatable	Repeatable comment at this code-unit that is displayed where there are references to this code unit	EOL comment field
Comment, Referenced Repeatable	Displays repeatable comment defined at code-unit referred to by this code unit	EOL comment field
Constant	Numbers	Operand field
Entry Point	Labels that are at Entry Points in a program	Label field
External Reference, Resolved	External addresses or labels	Operand field
Field Name	Name of elements in a structure or union	"Field Name" field
Flow Arrow, Active	Flow arrow when cursor is at the source	N/A
Flow Arrow, Not Active	Flow arrow when cursor is not at source	N/A

Function Name	Function name	Function Signature, Operands
Function Parameters	Parameter data types and names	Function Signature field
Function Return type	Data Type returned by the function	Function Signature field
Labels, Local	Labels that are local to a function	Label field
Labels, Non-primary	Additional labels at an address	Label field
Labels, Primary	Primary label at an address	Label field
Labels, Unreferenced	Labels at addresses that are not referenced	Label field
Mnemonic	Instruction or Data (Mnemonic)	Mnemonic field
Operands	defaults for items in operand field	Operand field
Registers	Registers	Operand field
Separator	Separator line (...) between address gaps	Separator field
Variable	Function variables	Variable Type, Variable Location, Variable Name, Variable Comment
Version Track	Version tracking information	Version tracker field
Xref	Cross reference addresses	Xref field
Xref, Offcut	Offcut cross reference addresses	Offcut references Xref field

\*The Background is a special case. Select this element to set the background color for the Listing.

#### Color

The color for the selected screen element is displayed in this section.

#### Style Settings

By default, all screen elements use the style settings set in the [Font](#) section. To customize individual screen elements, select the screen element from the Screen Elements section and select the *custom* checkbox in the Style Settings section. This will enable the **bold** and *italics* checkboxes which can be used to customize the style of the selected screen element.

#### Swatches/HSB/RGB

These tabbed components provide a variety of ways to choose a color for the selected screen element.

##### Swatches

Click on the desired color-square to set the color of the currently selected screen element.

##### HSB

This tab allows colors to be chosen using the hue, saturation, and brightness scale. The radio button selects which scale will be controlled by the slider bar. Use the slider bar to choose a value for the selected scale. This causes the large color box to display all possible colors with that selected value. Click in the box to choose a color.

##### RGB

This tab allows colors to be chosen by entering a red, green, and blue value.

#### Preview

The preview tab displays some sample code using the current set of screen element colors and styles.

### Fields, Highlights, & Selections

There are other customizable options in addition to color and font. To view or change these options, open the [Tool Options Dialog](#) and then open the *Listing Fields* node in the Options tree. Next, select the appropriate field within the *Listing Fields* node to view or change the options for that field. The available *Listing Field* options are:

#### Address Field

**Show Block Name** –This option prepends the memory block name to the address in the address field. For example, 0x10008000 might become .text:0x10008000.

**Fully Pad With Leading Zeros** –This option causes all addresses to be displayed with leading zeros to make the address string length the same as the length of the largest address in that address space. For example, in a 32 bit address space program the address 0x100 will be displayed as 00000100.

**Minimum Number of Address Digits** –This option is used to specify the minimum number of digits that should be used to display the address. The address string will be displayed with leading zeros to make the length contain at least this number of digits. If the largest address in the address space has fewer significant digits than this minimum, then the address will only be padded to the size of the largest address. If the address has more significant (non-zero) digits than the minimum, it will be displayed with more digits than the minimum, but will contain no leading zeros. This field is disabled and has no effect if the previous "Fully Pad With Leading Zeros" option is selected.

**Justification** –This option is used to align the address text to either the left or right side of the field. If the text is longer than the space allocated for it in the field, the text is clipped to the opposite side of the justification.

#### Array Options

**Group Array Elements** –Determines whether or not multiple array elements should be shown on the same line in the browser. If checked, this option will use the **Array** format, which typically only shows the address, the array index, and the array values. Note: this only affects arrays with primitive elements. Complex arrays will still only show one element per line.

**Elements Per Line** –The number of array elements to show on a line. This option is only enabled if the **Group Array Elements** option is selected.

#### Bytes Field

**Byte Group Size** –Bytes in the byte field can be displayed in groups. By default each bytes are displayed in groups of 1. Groups are separated by delimiters. Use this setting to change the number of bytes displayed in a group.

**Delimiter** –The string to use to separate groups. Normally, this is set to " " (a single space).

**Display in Upper Case** –Select this check box if the hex digits of the bytes should be displayed as ABCDEF instead of abcdef.

**Maximum Lines to Display** –Specifies the maximum number of lines to use to display the bytes. Any bytes that do not fit into this number of lines will not be displayed.

**Reverse Instruction Byte Ordering** –The bytes will be displayed in reverse order for Instruction Code Units. Note: this applies to instructions only. All other code units are unaffected.

**Display Structure Alignment Bytes** –Internal structure alignment bytes will be displayed with the bytes of the preceding component. These bytes will be colored differently than the bytes directly associated with the component. Note: this applies to structure data only. All other code units are unaffected.

#### Cursor Text Highlight

Use the following options to customize [Cursor Text Highlighting](#):

**Alternate Highlight Color** –Sets the alternate color used for cursor text highlighting. Double click on the color bar to bring up a color chooser dialog.

**Enabled** –Select this checkbox to enable cursor text highlighting.

**Highlight Color** –Sets the color used for cursor text highlighting. Double click on the color bar to bring up a color chooser dialog.

**Mouse Button To Activate** –Use the combo box to select which mouse button will be used to highlight text (Left, Right, Middle).

**Scope Register Operand** –Check this box to enable [Scoped highlighting](#) of registers within the operand field. If turned off, cursor highlighted text within an operand field is treated the same as text in other fields.

#### EOL Comments Field

The EOL Comments field displays the end-of-line comment at this address. If there is no EOL comment, it displays the repeatable comment. If there is no repeatable comment, it displays the repeatable comments from all referenced addresses. If there aren't any referenced repeatable comments, it displays an automatic comment if it can.

**Always Show the Automatic Comment** –Normally automatic comments are not shown if there is an EOL comment, repeatable comment, or referenced repeatable comment. By selecting this option, the automatic comment will be shown even if there is an EOL comment, repeatable comment, or referenced repeatable comment.

**Always Show the Referenced Repeatable Comment** –Normally referenced repeatable comments are not shown if there is an EOL comment or repeatable comment. By selecting this option, the referenced repeatable comment will be shown even if there is an EOL comment or repeatable comment.

**Always Show the Repeatable Comment** –Normally repeatable comments are not shown if there is an EOL comment. By selecting this option, the repeatable comment will be shown even if there is an EOL comment.

**Enable Word Wrapping** –If this option is not set, each comment line is displayed on a line by itself. By turning on word-wrapping, comment lines are displayed in paragraph form.

**Maximum Lines To Display** –The maximum number of lines to display for a single comment. Comments that cannot be displayed within this number of lines will be truncated.

**Prepend the Address to Each Referenced Comment** –Each referenced repeatable comment will display an annotated address before the comment. This address can be double clicked to go to the address where that referenced repeatable comment is defined.

**Show Semicolon at Start of Each Line** –Begins each line of an end-of-line comment with a semi-colon.

**Use Abbreviated Automatic Comments** –When possible, the auto comment will be made as small as possible. For example, when showing an offset string reference, only the portion of the string that is used will be displayed.

#### Format Code

Specific formatting of the listing is controlled via the *Format Code* options panel. The following table describes the effect of each format option. The options for displaying comments are listed in the order in which the options are processed when determining what formatting comments are to be displayed. (The order below is based on the assumption that the order of the comment fields in the Code Browser are *Plate*, *Pre*, and *Post*.)

Option	Field Type	Description
Show External Entry Plates	Plate Comment	Display an "EXTERNAL" plate comment at each label which is marked as an Entry Point.
Show Function Plates	Plate Comment	Display a "FUNCTION" plate comment at the entry point of each Function.
Show Subroutine Plates	Plate Comment	Display a "SUBROUTINE" plate comment at each label which has a call reference to it.
Show Transition Plates	Plate Comment	Display an empty plate comment when transitioning from between Instructions and Data.
Flag Function Entry	Pre Comment	Display the pre comment:         FUNCTION         at the entry point of each Function.
Flag Function Exits	Post Comment	Display the post comment: ***** MyFunction Exit ***** at the end of each Function, where <i>MyFunction</i> is the name of the Function.
Flag Jumps and Returns	Post Comment	Display a post comment "--" line separator following all unconditional jumps and returns.
Flag Subroutine Entry	Pre Comment	Display the pre comment:         SUBROUTINE         at each label which has a call reference to it.
Lines After Basic Blocks	Post Comment	Display the specified number of blank lines following all Basic Blocks of instructions.
Lines Before Functions	Plate Comment	Display the specified number of blank lines before each Function entry point. (Has precedence over <b>Lines Before Plates</b> .)
Lines Before Labels	Plate Comment	Display the specified number of blank lines before each labeled code unit (instruction or data).
Lines Before Plates	Plate Comment	Display the specified number of blank lines before each plate comment inserted during formatting. (Has precedence over <b>Lines Before Labels</b> .)

 If comments already exist in the listing, then the options to show the comments fields for formatting are ignored. The options that specify a number of blank lines are used regardless of whether comments exist.

 Applying the format options does not alter the program.

**Function Pointers**

**Display Function Header for External Function Pointers** –Select this option to show the function prototype header within the listing for any pointer which has an External Function reference (enabled by default).

**Display Function Header for Non-External Function Pointers** –Select this option to show the function prototype header within the listing for any pointer which has a memory reference to a Function (disabled by default).

**Function Signature Field**

**Display Namespace** –Select this option to include the function namespace as a prefix to the name of the function within the displayed function prototype (disabled by default).

**Labels Field**

**Display Function Label** –Select this option to show function labels. If you turn this off, the function name will only appear in the function signature. If it's on, the function name will also appear as a label below the function header.

**Display Non-local Namespace** –Select this option to prepend the namespace to all labels that are not in the current Function's namespace. Currently, this would only affect a label that is not global, but is in a namespace other than the function that contains the label's address.

**Display Local Namespace** –Select this option to prepend the namespace to all labels that are in the Function's namespace.

**Use Local Namespace Override** –Select this option to show a fixed prefix for local labels instead of the function's name. This option is only available if the "Display Local Namespace" option is on. The text box contains the prefix to use for local labels.

**Mnemonic Field**

**Show Data Mutability** –Option to display the mutability data setting associated with each data code unit (e.g., constant, volatile).

**Underline Fields With Non-primary References** –Option to underline mnemonic fields that have hidden (non-primary) references. This provides a quick visual indication that the field has references and that you can double-click the field to go to the references.

**Operands Field**

**Add Space After Separator** –Option to add an additional space after the operand separator ","

**Always Show Primary Reference** –Option to force the display of the primary reference on all operands. If a suitable sub-operand replacement can not be identified the primary reference will be appended to the operand preceded by a ">=" prefix.

**Display Abbreviated Default Label Names** –Uses a shortened form of the label name for dynamic String data types in the display of operand references (e.g., STR\_01234567).

**Display Non-local Namespace** –Select this option to prepend the namespace to all references that are not in the current Function's body. Currently, this would only affect a label that is not global, but is in a function other than the function that contains the current instruction.

**Display Local Namespace** –Select this option to prepend the namespace to all labels that are in the Function's body.

**Use Local Namespace Override** –Select this option to show a fixed prefix for local labels instead of the function's name. This option is only available if the "Display Local Namespace" option is on. The text box contains the prefix to use for local labels.

**Enable Word Wrapping** –Option to wrap strings in operand lines that are too long to fit in the operand field. Note that word wrapping can only occur where spaces exist in the string.

**Follow Read or Indirect Pointer References** –Option to follow referenced pointers, for read or indirect reference types, to show pointer's referenced symbol instead of symbol at pointer. When applied the resulting operand label will be preceded by a "→".

**Include Scalar Reference Adjustment** –Option to include a scalar expression which will indicate the relationship between a replaced scalar and the associated reference offset when the offset does not match the scalar value.

**Markup Inferred Variable References** –Option to markup instruction operands where references to stack and register variables can be inferred. This corresponding stack/register markup option must also be enabled for this option to have an effect.

**Markup Register Variable References** –Option to markup instruction operands where explicit register references exist. When this option is enabled, elements may be replaced within instruction operands to reflect an association with a register parameter/variable if one can be determined. Inferred register references will be included if the *Markup Inferred Variable References* is also enabled.

**Markup Stack Variable References** –Option to markup instruction operands where explicit stack references exist. When this option is enabled, elements may be replaced within instruction operands to reflect an association with a stack parameter/variable if one can be determined. Inferred stack references will be included if the *Markup Inferred Variable References* is also enabled.

**Maximum Length of String in Default Labels** –Sets the maximum number of characters from a String to include in dynamic String labels in operand references

**Maximum Lines To Display** –The maximum number of lines used to display a string in an operand. Strings that cannot be displayed within this number of lines will be truncated.

**Show Block Names** –This option prepends the memory block name to labels in the operand field. For example, the instruction "call printf" becomes "call .text:printf".

**Show Offcut Information** –Shows additional information for [offcut references](#), such as the original address, followed by the offset. For example, the string "foo\_bar", with an offcut reference to "bar" would look like:

```
s_bar_12345678+4
```

with this option on, and with it off would look like:

```
s_bar
```

**Underline References** –Option to underline operand references so that you can quickly identify those operands that have references and double-click to go to the reference. Select one of the following choices:

- *Hidden* –Underline the operand if it has a non-primary reference (i.e. there is no visible evidence that the operand has a reference.)
- *All* –Underline the operand if it has any references.
- *None* –Do not display any underlines.



You can change the color of the underline from the [Listing Display](#) options panel; select *Underline* from the *Screen Element* list.

**PCode Field**

**Display Raw PCode** –Option to display the raw PCode directly in the Code Browser (i.e., detailed varnode specifications are provided).

**Maximum Lines to Display** –The maximum number of lines used to display PCode. Any additional lines of PCode will not be shown.

#### Selection Colors

**Difference Color** –Sets the color used to highlight differences between two programs.

**Highlight Color** –Sets the Browser [Highlight](#) color.

**Selection Color** –Set the Browser [Selection](#) color.

#### XREFs Field

**Delimiter** –Delimiter string to use for separating multiple xrefs.

**Display Local Block** –Prepends the name of the memory block containing the XREF source address to each XREF.

**Maximum Number of XREFs To Display** –The maximum number of lines used to display XREFs. Additional XREFs will not be displayed.

**Display Non-local Namespace** –Select this option to prepend the namespace to all XREFs that are not from an instruction within the current Function's body. Currently, this would only affect XREFs that originate in some other function.

**Display Local Namespace** –Select this option to prepend the namespace to all XREFs that are from the current Function.

**Use Local Namespace Override** –Select this option to show a fixed prefix for local XREFs instead of the function's name. This option is only available if the "Display Local Namespace" option is on. The text box contains the prefix to use for local XREFs.

### Popups

Popups are transient windows used to display additional information about the item over which the mouse is hovering. There are options for enabling various types of popups and for resizing the popup window.

#### Listing Popups

To view or change the available popups, open the [Tool Options Dialog](#) and then select the *Listing Popups* node in the Options tree.

**Reference Code Viewer** –Shows the listing at the referenced location as a popup.

**Truncated Text Display** –Displays truncated text as a popup.

#### Reference Code Viewer

To view or change the size of popup windows, open the [Tool Options Dialog](#) and then open the *Listing Popups* node in the Options tree. Next select the *Reference Code Viewer* node.

**Dialog Height** –The height of the popups in pixels.

**Dialog Width** –The width of the popups in pixels.

## Program Differences

The Code Browser can be used to view a second program alongside the tool's program to determine the differences between them. This is called a Program Diff. The Program Diff highlights code units in the second program to indicate there is a difference. Highlighted code units in the second program can then have their differences applied to the tool's program causing it to change. The ultimate goal is to apply some or all of the differences from the second program to the tool's program in order to get another user's changes into your program.

### Opening and Closing the Second Program

The **Open/Close Diff View** icon  in the Code Browser tool bar is used for both opening and closing the second program in the Code Browser. This is one of two ways to open the Diff tool. The other is the menu action, **Tools → Program Differences...** which is described later under **Viewing Program Differences**.

#### Opening

When there is only a single program displayed in the Code Browser, select the Open Diff View icon  in the tool bar above the program to open a program for Diff. You will be prompted to select the second program and then prompted for Diff settings just as when [viewing program differences](#).

#### Closing

When you are done working with program differences and the second program, select the  icon in the tool bar above the second program. The second program and the associated Diff are closed and the Code Browser returns to displaying only the tool's program.

#### Bring Diff View to Front

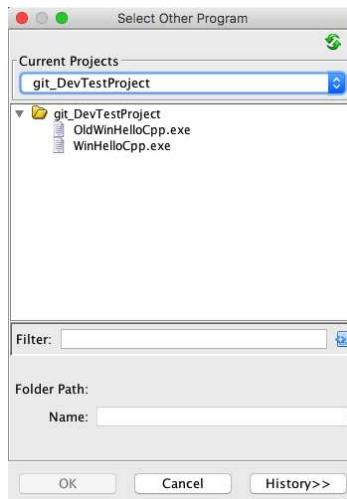
If you navigate away from the current diff session to another program tab, then clicking the  action will make the diff session be the active program tab.

### Viewing Program Differences

A Diff can be opened from the menu action, **Tools → Program Differences...**. This is one of two ways to open the Diff tool. The other is the **Open/Close Program Diff** icon  in the Code Browser tool bar which is described above in **Open/Close Diff View**. The following describes using the menu action to open the Diff tool.

With a program open in the Code Browser tool, do the following to view a second program's differences.

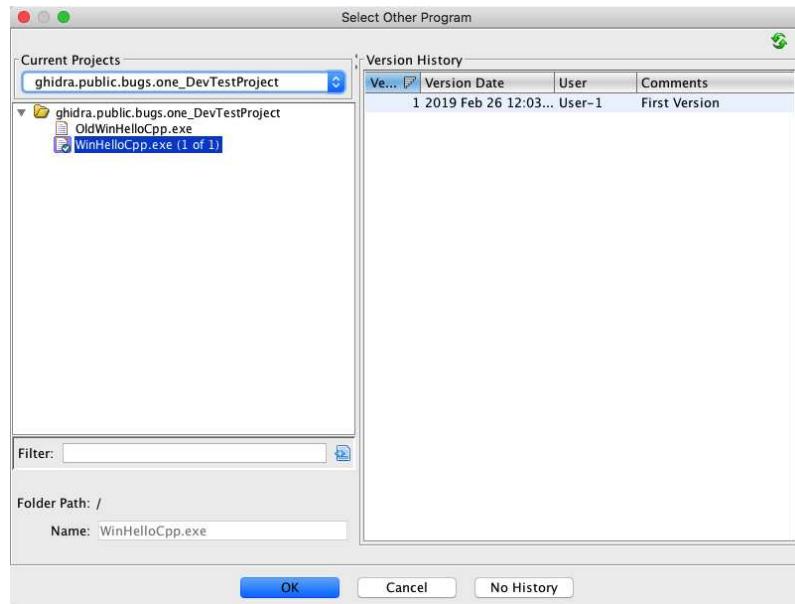
1. From the Code Browser, select **Tools → Program Differences...**.
2. The *Select Other Program* dialog is displayed.



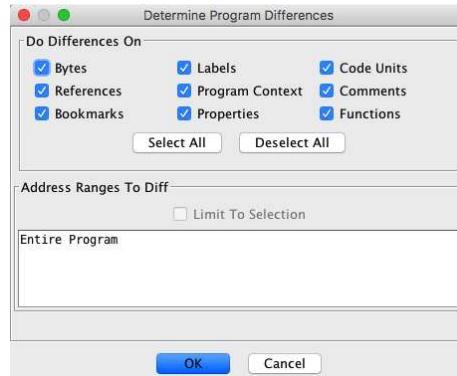
3. From the *Select Other Program* dialog select the second program to be viewed and compared with the tool's program.

 The second program must have the same address spaces defined. In other words, they should be the same type of program (i.e. based on the same program language).

 If you wish to Diff against a previous version of a **versioned program**, select the **History>>** button of a program. Select the versioned program from the tree. Next select the desired version of the program from the history list on the right side of the dialog.



4. Click the **OK** button.  
 5. The *DetermineProgramDifferences* dialog is displayed. This dialog is displayed whenever you are going to determine the differences between two programs. To get all the differences between the two programs make sure all the check boxes in the **Do Differences On** area have checks and the check box in the **Address Ranges To Diff** should not have a check as shown here.

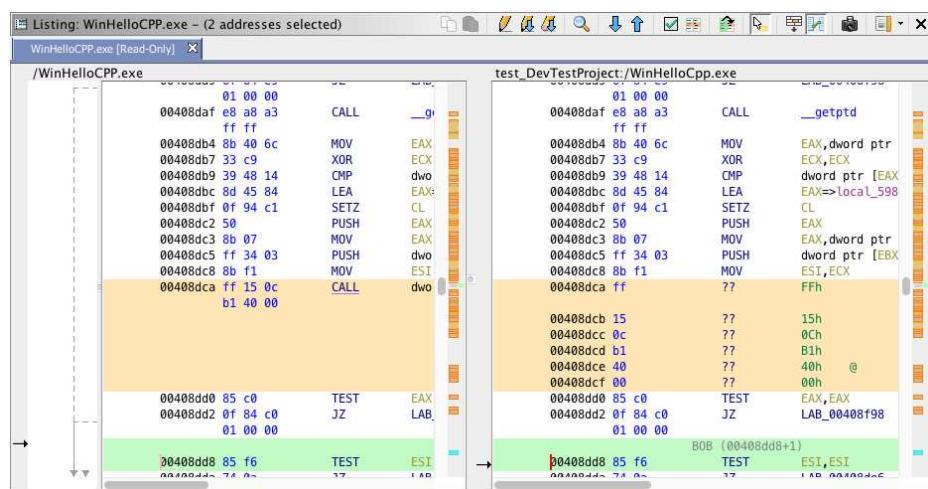


The check boxes allow you to limit the types of differences that are determined and where differences are determined in the program. To learn more, see [Determine Program Differences Dialog](#).

6. Click the **OK** button to determine the differences.  
 7. An *InProgress* dialog appears while the differences are being determined. Upon completion the dialog disappears and the code unit's in the second program where differences exist will be highlighted.

The selected program is displayed as the second program in the browser. Only the portion of the second program that matches the address set of the current view is displayed. Highlighted code units in the second program indicate where there are differences.

The following illustrates a Program Diff where differences between two programs are highlighted.



The tool's program, that will be modified by the Diff, is displayed on the left-hand-side of the Code Browser. The second program, where the differences to apply are obtained, is displayed on the right-hand-side. Markers in the right margin of the second program give an indication of all the positions in the current view where there are differences between the two programs. The view of both programs is still controlled by the program tree for the tool's program. In other words, the second program's view will only show addresses that match those viewed for the tool's

program.

 Notice that the difference highlights are light orange above. The selected difference is green. These are only the default colors. They can be changed from the tool's Options dialog (Edit → Options..., open the Listing Fields folder, and select Selection Colors, double click on the Difference color).

Blank space is added to each program as necessary to align the addresses between the two programs. The vertical or horizontal scrollbar of either program causes both programs to scroll in a synchronized manner. Also, changing the cursor location in one program, changes it in the other program.

The Program Diff colors the two programs where they differ. These differences can then be applied individually from the second program to the tool's program. Applying a difference changes the tool's program so it matches the second program where it was different previously. The ultimate goal is to apply some or all of the differences from the second program to the tool's program.

Program Diff is most commonly used to compare different users' copies of the same program. With versioned programs, it can also be used to determine differences against an older version of the same program. When working on a project as a team, each person can make changes to his own copy of a program file. The Program Diff can then be used to determine where the programs differ. Changes that another team member has in his program can be merged into yours by applying the difference.

The toolbar above the displayed programs provides buttons for use with a Program Diff. Click on the icon below to go to the help section that gives details of that feature.

-  Apply the selected differences from the second program to the tool's program.
-  Apply the selected difference and go to the next highlighted difference block.
-  Ignore the selected differences and go to the next highlighted difference block.
-  Show the details of the differences between the programs at the current cursor location.
-  Go to the next highlighted difference block.
-  Go to the previous highlighted difference block.
-  Display the Program Difference Settings dialog.
-  Determine the differences between the programs.
-  Open/CLOSE the second program and the Program Diff.

### Determine Program Differences Dialog

Whenever you initiate a Program Diff the *DetermineProgramDifferences* dialog allows you to control the following settings:

- **Do Differences On:** the types of differences to be determined and highlighted by the Program Diff. The **Select All** button is a convenience for selecting all of the check boxes for difference types. Likewise, the **Deselect All** button removes the check marks from all the difference type check boxes.
  - **Address Ranges To Diff:** indicates the addresses to be checked for differences.
- Limit To Selection: whether or not the Program Diff should be limited to the current selection in the tool's program.



Pressing the **OK** button will check the indicated addresses for differences. It then highlights the code units in the second program where differences were found. While the Program Diff is determining the differences, an *InProgress* dialog is displayed. Pressing the **Cancel** button will remove the *DetermineProgramDifferences* dialog without performing the Program Diff.

 If the two programs being compared are large, determining program differences can be time consuming. If you are not interested in all of the differences in the program, you can reduce the amount of time required to determine differences by removing some types of differences being determined or by determining differences on just a portion of a program.

#### Do Differences On

The **Do Differences On** area specifies the types of differences that will be determined. Placing a check in a box indicates that this type of difference should be detected.

**Bytes** –detect any code units that have memory bytes that differ between the programs being compared.

**Labels** –detect any code units where the labels differ.

Possible label differences are:

- More labels at a code unit in one program than the other
- Labels named differently
- Different labels could be marked as the primary label
- Same named label, but with different scope (parent namespace). For example, one program's label is local to a function and the other label is a global label.
- Same named label, but with different source. For example, one program's label was named by default and the other label was named by the user.
- One program's code unit with a label could be an entry point and the other program code unit is not an entry point

**Code Units** –detect any code units where the code unit or equates differ.

Possible code unit differences are:

- Different instructions.
- Instruction in one program and Data in the other program at the same address.
- Different equates on the instruction in each program.
- Different data types when data is defined in both programs at an address.

**References** –detect any code units where the references differ.

Possible reference differences are:

- One program has a reference and the other doesn't.
- References are not to the same address.

- References are not on the same operand or mnemonic.
- References types (Memory, Stack, External) are different.

**Program Context** –detect any code units where the values of the program context registers differ.



If the two programs contexts don't have the same registers defined, the program context is disabled and can't be compared between the programs.

**Comments** –detect any code units where the comments differ.

Possible comment types are:

- End of Line Comment
- Pre-Comment
- Post-Comment
- Plate Comment
- Repeatable Comment

The difference could be that a comment exists in one program and not the other. It could also be that the text of a particular comment type is not exactly the same for the two programs.

**Bookmarks** –detect any code units where bookmarks differ.

**Properties** –detect any code units where User Defined Properties differ.

**Functions** –detect any code units where the functions differ.

Possible differences are:

- Function in one program and not in the other.
- Function comment differs.
- Addresses that make up the function body differs.
- Function signature differs. (Function name, return type, or parameters differ.)
- Parameters differ by name, data type, size or offset.
- Local variables differ by name, data type, size, or offset.
- Stack offset differs.
- Stack Frame size differs.
- Whether or not the stack grows in a negative direction.
- Function tags differ.

When the *DetermineProgramDifferences* dialog is initially displayed, all the Differences check boxes are checked. This indicates that all the types of differences will be detected and displayed.

#### Address Ranges To Diff

The **Address Ranges To Diff** area indicates the address ranges where program differences will be determined. If the two programs have different address sets, then only the addresses that the two programs have in common can be compared. The addresses where differences are determined can also be limited to a set of addresses that are selected in the tool's program.

#### Limit To Selection

If there is a selection in the tool's program when you determine differences, the address set where differences are determined can be limited to the addresses in the selection. Selecting the **Limit To Selection** box, limits the differences to the selection's addresses.

#### Diff and Multiple Program Tabs

Only one Program Diff can be performed at a time. However, you can view another program using the tabs in the [Listing View](#) while a Program Diff is active. This will not terminate the current Diff. The Diff will reappear when you change back to the tab for the program that the Program Diff is being performed on.

While a Program Diff is active, the **Diff View** icon is visually pressed down. If the Diff is being performed on a program that is not being actively displayed (in a tab other than the current tab), then pressing the **Diff View** icon will bring the tab containing the Diff to the front. If you attempt to start a Diff using **Tools** → **Program Differences...** in the tool menu and a Diff is already being performed, the status bar will indicate the name of the program that currently has a Diff.

A Program Diff is terminated by any of the following:

- Selecting the **Diff View** icon from the Listing toolbar when the Program Diff is actively being displayed.
- Selecting the **Close Window** icon from the Listing toolbar when the Program Diff is actively being displayed. (Normally this will close the current program, but when Diff is active this closes the Diff.)
- Closing the Diff's program via the Tool's menu, **File** → **Close**, when the Diff is actively displayed. (In this case the Diff is closed along with the current program.)
- Exiting the Tool or Ghidra will also terminate an active Diff.



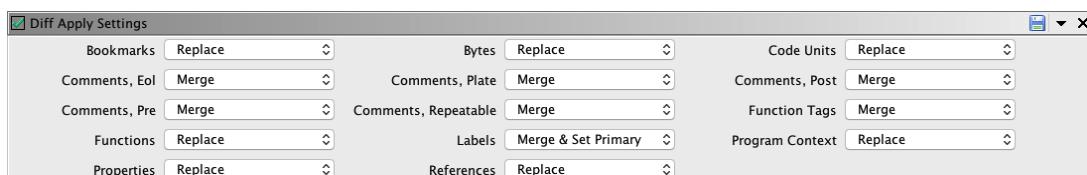
Hovering the mouse over the **Diff View** icon provides a tooltip indicating whether the current action is *Open Diff View*, *Close Diff View*, or *Bring Diff View to Front*.

#### Diff Apply Settings

The **Diff Apply Settings** dockable component allows you to control which types of differences get applied from the second program to the tool's program, and how each type gets applied.

For each type of difference, you can select one of the following apply settings:

- **Ignore** –do not change the selected code units in the tool's program for this type of difference.
- **Replace** –changes the tool's program to match the second program for this type of difference.
- **Merge** –(Only available for Labels and Comments) changes the tool's program by adding this type of difference from the second program to what is already in the tool's program. For Labels, this will not change which label is set to primary.
- **Merge & Set Primary** –(Only available for Labels) merges labels from program 2 into the current program and sets the primary label as it is in program 2, if possible.



The following types of differences are controlled by the *Diff Apply Settings*.

**Bookmarks** –all categories of bookmarks.

**Bytes**

**Code Units** –includes code units (**instructions & data**) and **equates**.

**Comments** –**Plate**, **Pre**, **End-of-line**, **Repeatable**, & **Post** comments.

**Functions** –functions (includes their associated stack frames).  
& **Function Tags**

**Labels** –labels or symbols.

**Program Context** –program context **register** values.

**Properties** –user defined properties that have been added to the program by plugins.

**References** –**Memory**, **External**, & **Stack** references.

The *Diff Apply Settings* toolbar has a **Save as Default** icon to save your current Diff Apply Settings to the tool as your new defaults. When you select it, the current *Diff Apply Settings* values are set to the default Diff Apply Settings. Whenever you start a new Diff, the current *Diff Apply Settings* will get set to the default settings for that Program Diff. If you determine the program differences again for an existing Diff using the **Determine Program Differences** icon , the current *Diff Apply Settings* will not be affected by the defaults.

The Default Diff Apply Settings can also be set by changing the [Diff Default Apply Settings tool options](#).

The *Diff Apply Settings* toolbar menu also has actions for changing all the apply settings at once. They are:

Set Ignore for All Apply Settings
Set Replace for All Apply Settings
Set Merge for All Apply Settings

- **Set Ignore For All Apply Settings** –Changes all the settings to **Ignore**. Before you can apply anything you would need to individually set at least one difference type's setting to **Replace** or **Merge**.
- **Set Replace For All Apply Settings** –Changes all the settings to **Replace**.
- **Set Merge For All Apply Settings** –Changes all the settings to **Merge** that allow merging and all others are changed to **Replace**. Labels will be set to **Merge & Set Primary**.

## Getting New Differences

There are several reasons to get new program differences.

- Changes were made to the tool's program directly (not by applying a difference) since you last determined program differences.
- The second program has been changed since you last determined program differences.
- You want to determine differences for different addresses of the program than the current Diff used. For example, you initially determined program differences limited to a selection and now you want to know differences for the whole program or for a different set of selected addresses.

When a second program is already displayed in the Code Browser tool, do the following to determine new program differences between that program and the tool's program.

1. From the toolbar above the second program, select the Get Differences button.  
or  
Press the right mouse in the second program area. From the popup menu, select **GetDifferences...**.
2. The [DetermineProgramDifferences](#) dialog is displayed.
3. From the dialog, select the types of differences to be determined and the addresses to check for differences.
4. Click the **OK** button.

An *In Progress* dialog appears while the program differences are being determined. When it is done determining the differences, the dialog is removed and the differences are highlighted in the second program.

## Limiting the Diff to the Browser Selection

The **Limit to Selection** box in the *ProgramDifferenceSettings* dialog can be selected whenever a Program Diff is started while there is a selection in the tool's program in the Code Browser. When the box is checked the differences will only be determined for addresses in that selection. This limits the differences being highlighted and manipulated by the Program Diff to only those that are in a specific set of code units in the program. To limit the Program Diff select the target code units in the tool's program in the browser and check the **Limit to Selection** box in the *ProgramDifferenceSettings* dialog. Notice that the **Address Ranges To Diff** area changes as you check or uncheck the **Limit to Selection** box. It switches between the addresses in common between the two programs and those in the selection.

Once you press the **OK** button on the *ProgramDifferenceSettings* dialog, the Program Diff is performed against the indicated address ranges (address set). You can make a selection in the tool's program before initiating the Get Differences to get differences against a different address set.

## Navigating Differences

When viewing program differences in the CodeBrowser, you can navigate on difference blocks. A difference block is a contiguous group of one or more highlighted code units in the second program. A difference block can be a single code unit where a difference was found if the code unit before and after it did not have a difference. There can be multiple code units in a difference block when each of the code units next to each other has a difference. To move to the next/previous code unit in a highlighted block use the up and down arrow keys on the keyboard or click the mouse on the desired code unit.

### NextDifference

Selecting the **NextDifference** button moves the current location in the Program Diff to the next difference block.

If only part of the program is in the current view, the next difference may be outside the current view of the program. If this is so, then navigating will add the fragment, with the next difference, to the view.

### PreviousDifference

Selecting the **PreviousDifference** button moves the current location in the Program Diff to the previous difference block.

If only part of the program is in the current view, the previous difference may be outside the current view of the program. If this is so, then navigating will add the fragment, with the previous difference, to the view.

### Marker Margin

Clicking on a difference marker in the right margin will navigate to that difference.



Some of the difference markers may overlap in the right margin. Therefore, it is best to use the markers for navigation to a region where there are differences and then use the **NextDifference** or **PreviousDifference** buttons to navigate to the next/previous difference block.

## Selecting Differences

### Automatically Selecting a Difference Block

Using the **NextDifference** or **PreviousDifference** to navigate to a highlighted difference block selects the code units contained in that difference block. Left mouse clicking on a difference block also causes the entire difference block to become selected. You can then right mouse click and the popup menu appears so you can apply the selection.

### Manually Selecting Differences

Differences can be selected within the right-hand Diff listing as follows:

- To select a single difference block, simply left mouse click on the highlighted difference block. It will become selected, so it can then be applied.
- Left mouse clicking on a code unit that is not highlighted as part of a difference block will clear the current Diff selection.
- To select individual code units within a highlighted difference block, drag the cursor in the second program as you normally would in the Code Browser. If you select code units that are not highlighted as a difference, they will automatically be removed from the selection when the mouse button is released. When the cursor is released, the selection will become restricted to only the code units with highlighted differences in the selection.
- The Ctrl key along with a left mouse click can be used to add/remove a code unit from the selection.
- The Shift key along with a left mouse click can be used to extend the selection.



When differences are applied from the second program to the tool's program, only selected code units in the current view will be applied.

### Select All

Invoking the **Select All Differences** from the second program's popup menu, selects all the currently highlighted differences in the second program.

### Setting the Program Diff Selection From the Tool's Program Selection

The selection from the tool's program can be used to select some of the differences in the second program. While viewing program differences in the Code Browser, make a selection in the tool's program. Select the **Set Diff Selection** icon

For example, you can use this to select all the differences in a subroutine. First select all the code units in the tool's program that make up the subroutine. The **Set Diff Selection** icon will become enabled. Select the **Set Diff Selection** icon . All the corresponding code units with highlighted differences in the second program become selected. The selection of differences can then be [applied](#) or [ignored](#).

## Applying Differences

Applying a difference, changes the tool's program to match the second program in the Program Diff for each type of difference being applied. The Program Diff can control which types of differences get applied from the second program to the tool's program (see [Apply](#)). It can also control whether some types of differences replace what is in the tool's program or whether they are merged into the tool's program. Only the highlighted code units in the second program, which are currently selected, can have their differences applied.

### To apply a selection

1. Select Differences in the second program.
2. Make sure the difference types you want to apply have their boxes checked in the *ProgramDiffSettings* dialog.
3. Click the **Apply Selection** button in the toolbar.  
or  
Press the Apply Selection hot key.  
or  
Click the right mouse on the highlighted difference and select the **Apply Selection** button.

Remember all the types of differences that are being [Replaced](#) or [Merged](#) in the *Diff Apply Settings* dockable component will be applied for the selected code units.

When applying differences, comments and labels can be replaced with the second program's comments or merged with them. Merging comments or labels in the Program Diff results in the union of the two programs comments or labels for each code unit being applied.

### Example:

A code unit in the tool's program does not have any comments or labels. The code unit in the second program has a pre-comment and a bookmark. All Apply boxes are checked. Apply a selection containing the code unit. The pre-comment and bookmark appear in the program in the Code Browser.

Say the Bookmarks box in the Apply area of the settings dialog was not checked and the Comments box was checked when the difference was applied. The pre-comment would appear in the Code Browser's program, but the bookmark would not.

## Applying Differences and Going To Next

This applies the selected differences and navigates to the next difference in a single step (see [Apply](#) and [Next](#)). It can also control whether some types of differences replace what is in the tool's program or whether they are merged into the tool's program. Only the highlighted code units in the second program, which are currently selected, can have their differences applied.

### To apply a selection and go to the next difference

1. Select differences in the second program.
2. Make sure the difference types you want to apply have their boxes checked in the *Diff Apply Settings* dockable component.
3. Click the **Apply & Go To Next** button in the toolbar.

All the types of differences that were last selected under Apply in the *ProgramDifferenceSettings* dialog will be applied for the selected code units. The Diff will then navigate to the next difference and select it.

## Ignoring Differences

Ignoring a difference indicates you no longer want the current Program Diff to highlight the code unit. The Diff then navigates to the next difference.

Only selected code units in the second program of the Program Diff can have their differences ignored. The selection can be made on an entire highlight block or individual code units in one or more difference blocks. (See [Selecting Differences](#)).

Note: All versions of Ghidra before 7.4.X did not navigate to the next difference after ignoring.

#### To ignore a selection

1. Select Differences in the second program.
2. Click the **Ignore Selection** button  in the toolbar.  
or  
Press the Ignore Selection hot key.  
or  
Click the right mouse on the highlighted difference and select the **Ignore Selection** button.

The selected code units will be ignored. This means they will no longer be highlighted as a difference. The Diff will then navigate to the next difference and select it.

 Once a code unit is ignored, it will no longer be marked in the browser with a change bar and will no longer be highlighted. If you Determine Program Differences, any previously ignored items where there are still differences will once again be highlighted.

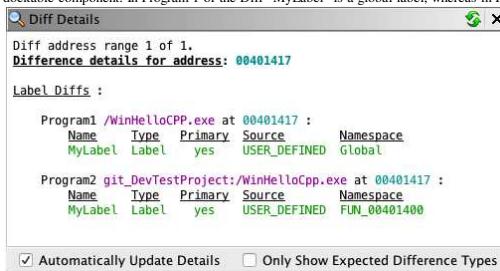
#### Viewing Difference Details at a Location

When viewing two programs in the Code Browser, it is possible to view all the differences at the current program location. The *DiffDetails* dockable component displays details indicating differences, if any, between the two programs code units at the current program location.

To view the difference details at a location:

1. Click on the code unit of interest to set the location and select the **Location Details** button  in the tool bar.  
or  
Press the right mouse button on the code unit of interest and select the **Location Details** button.
2. The *DiffDetails* dockable component is displayed indicating the detailed differences, if any, at the indicated location.

The following image shows the *DiffDetails* dockable component. In Program 1 of the Diff "MyLabel" is a global label, whereas in Program 2 "MyLabel" is a local function label.



 When the **Automatically Update Details** check box is selected, the Diff Details will update automatically to show the differences at the current location whenever the current address changes. This can be useful when you are navigating through Differences and need to see details that are not displayed by the CodeBrowser.

 When the **Only Show Expected Difference Types** check box is selected, the Diff Details will only show Diff Details for the types of Differences that you chose to look for when you determined your differences. If this box is not checked, you will see all types of differences that exist at the location regardless of whether it was one of the types of differences you were seeking.

 If you modify the program at the current location after the Diff Details are displayed, press the **Refresh** button .

#### Why view the difference details at a particular program location?

- The code units appear different between the two programs, but are not highlighted.
- If a particular type of difference was not selected when differences were determined, then it will not get highlighted.
- If the thing that appears different is due to a difference elsewhere in the program, then it will not get highlighted. For example, XREFs are not a difference where they are shown. The code unit where the reference is from is different, not where the reference is to. The XREF is displayed on the code unit the reference is to.
- The code units appear the same between the two programs, but are highlighted. This can happen when there is something different about the code unit, but that thing is not displayable. For example, references are not visually displayed, but they are a valid difference at the code unit the reference is from.
- The location is highlighted, but you want to know exactly what is different here.

In other words, are there other types of differences here besides the boxes I checked in the *ProgramDifferenceSettings* dialog? Remember, only the difference types with their **Apply** boxes checked in the dialog will be applied if an **Apply Selection** is done.

#### Diff Apply Settings Tool Options

The Program Diff adds **Default Apply Settings** options to the tool. To view or edit the option settings:

- From the tool's menu select **Edit → Tool Options...** which displays the [Tool Options Dialog](#).
- Open the *Diff* tree node.
- Click on the *Default Apply Settings* tree node to view its options.

Each time a new Program Diff begins, the *Diff Apply Settings* will have their values set to the default ones specified by the *Default Apply Settings Options*.

Each option will have one or more of the following settings available:

Diff Apply Setting	Functionality
Ignore	Do not change the selected code units in the tool's program for this type of difference.
Replace	Change the selected code units in the tool's program to match the second program for this type of difference.
Merge	(Only available for Labels, Comments and Function Tags.) Change the selected code units in the tool's program by adding this type of difference from the second program to what is already in the tool's program. For Labels, this will not change which label is set to primary.
Merge & Set Primary	(Only available for Labels.) Merges labels from the second program into the tool's program for the selected code units in the Diff and set the primary labels as in the second program, if possible.

The *Default Apply Settings Options* contains the following options:

Option	Functionality
--------	---------------

Bookmarks	Controls whether bookmark differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
Bytes	Controls whether byte differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
Code Units	Controls whether instruction, data, and equate differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
End Of Line Comments	Controls whether end of line comment differences will be applied. Can be: <i>Ignore</i> , <i>Replace</i> , or <i>Merge</i> .
Functions	Controls whether function differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
Function Tags	Controls whether function tag differences will be applied. Can be: <i>Ignore</i> , <i>Merge</i> or <i>Replace</i> .
Labels	Controls whether label differences will be applied and which to set as the primary label. Can be: <i>Ignore</i> , <i>Replace</i> , <i>Merge</i> , or <i>Merge &amp; Set Primary</i> .
Plate Comments	Controls whether plate comment differences will be applied. Can be: <i>Ignore</i> , <i>Replace</i> , or <i>Merge</i> .
Post Comments	Controls whether post comment differences will be applied. Can be: <i>Ignore</i> , <i>Replace</i> , or <i>Merge</i> .
Pre Comments	Controls whether pre comment differences will be applied. Can be: <i>Ignore</i> , <i>Replace</i> , or <i>Merge</i> .
Program Context	Controls whether program context register value differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
Properties	Controls whether user defined property differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
References	Controls whether reference differences will be applied. Can be: <i>Ignore</i> or <i>Replace</i> .
Repeatable Comments	Controls whether repeatable comment differences will be applied. Can be: <i>Ignore</i> , <i>Replace</i> , or <i>Merge</i> .

To change an option, click on the combo box to the right of the option name and select the desired setting from the list.

### Problems/Limitations

You cannot undo an **Ignore Selection** action. Undo of an **Apply Selection** or **Ignore Selection** in a Program Diff currently can't re-highlight the code units with differences that were ignored. If you want to get back differences after undo of an **Apply Selection** or if you want all ignored differences to no longer be ignored, you must re-Diff the programs by selecting the **GetDifferences** button .

Provided by: *ProgramDiff* Plugin

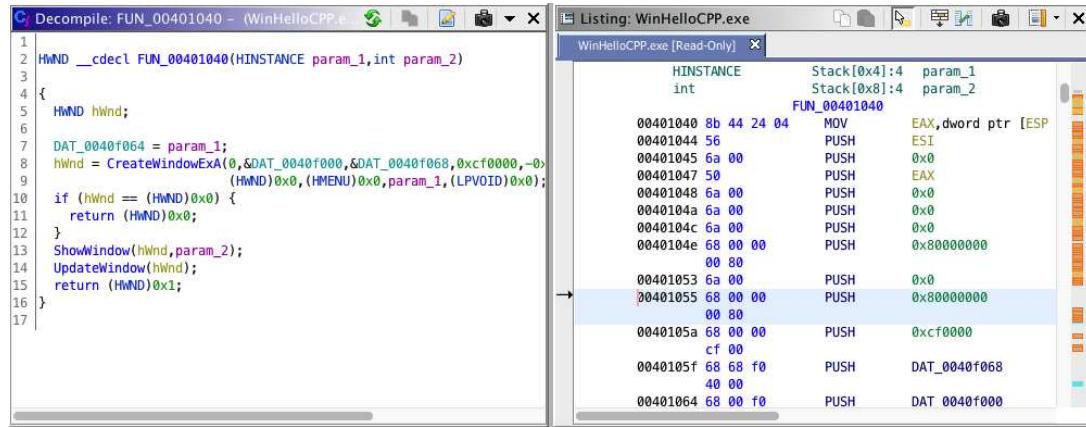
Related Topics:

- [Code Browser](#)

## Decompiler

The Decompiler plugin is a sophisticated transformation engine which automatically converts the binary representation of individual functions into a high-level C representation. The Decompiler presents a view of a program which is interactive and dynamically updated as the user adds or makes changes to the annotations associated with the program. A Decompiler window maintains the correspondence between the C representation and the assembly representation displayed in the Code Browser window, to the extent possible. The window allows instant visual association and navigation between C language expressions and their corresponding assembly instructions.

To display the decompiler window, position the cursor on a function in the Code Browser, then select the  icon from the tool bar, or the **Decompile** option from the **Window** menu in the tool.



Some of the primary capabilities of the decompiler include:

- **Recovers Expressions:** The decompiler does full dataflow analysis which allows it to perform slicing on functions. The most tangible benefit to the user is that complicated expressions, which have been split into distinct operations/instructions and then mixed together with other instructions by the compiling/optimizing process, are reconstituted into a single expression again by the decompiler.
- **Recovers High-Level Scoped Variables:** The decompiler understands how compilers use processor stacks and registers to implement variables with different scopes within a function. Data-flow allows it to follow what was originally a single variable as it moves from the stack, into a register, into a different register, etc. Thus it can effectively recover the original programs concept of a variable, minimizing the need to introduce artificial variables in the output.
- **Recovers Function Parameters:** The decompiler understands the parameter passing conventions of the compiler and can reconstruct the form of the original function call.
- **Uses Data type, Name, and Signature Annotations:** The decompiler automatically pulls in all the different data types and variable names that the user has applied to functions, and the C output is altered to reflect this. High-level variables are appropriately named, structure fields and array indices are calculated and displayed with correct syntax, constant char pointers are replaced with appropriate quoted strings, etc.
- **Performs Local Type Propagation:** In the absence of information, the decompiler does its best to fill in information from what it does know. Variables whose data type has not been explicitly labeled by the user can often be recovered by seeing how the variable is used or by allowing the known data types to propagate.
- **Can be used to Automatically Recover Structure Fields:** The decompiler can be leveraged to recover references to a structure.

### Variables

The decompiler will attempt to combine different locations (stack, memory, register) for variables within a function. Data type information for variables is gathered automatically from several sources. Any annotated function signatures, both of the function and of any sub-functions it calls, provide type information. If the function contains references to global memory locations that have a data type applied to them, these will also be used, and any local variables of the function can be annotated directly with data types. The user can provide data-type information to the decompiler by annotating all these sources. The more information that can be provided the better the produced C-code will be.

Variables not labeled directly are assigned types by analyzing local type propagation. Typically, assigning data types to a few key variables dramatically improves the readability of the C-code, as propagation will accurately fill in all the other data types. Assigning types in function signatures and to global variables is particularly effective because of their effect across multiple functions simultaneously.

 If you have C-header files for an API a program is using, there is a prototype C-Code parser than can extract the Data Type information from C-Code and create a [Ghidra Data Type Archive \(.gdt\)](#). The interface is currently fairly crude, but it handles most C syntax including macro expansion. The function signatures and data types extracted can be applied to the program. Just [open the archive](#) in the *Data Type Manager* window, select the archive, right mouse click, and select [Apply Function DataTypes](#). Ghidra currently provides definitions for the majority of windows API functions and data types automatically.

### Parameter Variables

Specifying data types for function parameters is especially useful. A function that has data types defined for its parameters will propagate these types into the variables of any calling functions.

C variable argument conventions, or varargs, are also supported. For instance, if the user has identified the standard C library routine *printf*, the signature can be defined to be void *printf*(char \* ...). Now whenever *printf()* is called, the decompiler will display the correct number of variable arguments.

Function signatures can be applied from a Ghidra data type database. Windows data types and standard C library function signatures are included with the standard distribution. More definitions will be added in the future.

### Internal Decompiler Functions

Occasionally, the decompiler may use one of several internal decompiler functions that don't get transformed into more C'-like expressions. Use of these can indicate that the pcode is incorrect or needs to be "Tuned" to make the decompiler output better. It can also mean that the decompiler needs an additional simplification rule to take care of that particular situation.

- **SUB41(x,c)** –truncation operation

○ The 4 is the size of the input operand (x) in bytes.

- The 1 is the size of the output value in bytes.
- The x is the thing being truncated
- The c is the number of least significant bytes being truncated

**SUB42(0xaabbccdd,1) = 0xbbcc**

When "c" is 0, the operation is almost always a cast between integer sizes, where the decompiler didn't quite figure it out. Usually the decompiler didn't figure out that "x" was an integer type or was forced to assume otherwise.

SUB41(x,0) is usually a cast from "int" to "char".  
 SUB42(x,0) is a cast from "int" to "short" and so on.  
 SUB84(x,4) is probably part of an extended precision multiplication but also turns up in other things like division strength reduction.

- **CONCAT31(x,y)** –concatenates two operands together into a larger size object

- The "3" is the size of x in bytes.
- The "1" is the size of y in bytes.
- The result is the 4-byte concatenation of the bits in "x" with the bits in "y". The "x" forms the most significant part of the result, "y" the least.

**CONCAT31(0xaabbcc,0xdd)=0xaabbccdd**

This usually crops up when a 1-byte sized (char) variable is being stored in a 4-byte register. All the basic arithmetic/logical ops on the 4-byte register give the correct result for doing the operation on a 1-byte variable; the compiler just has to make sure to ignore the 3 most significant bytes of the register. The CONCAT31 is the decompiler keeping track of these most significant bytes that the compiler was ignoring because it is mistakenly interpreting the register variable as being a 4-byte variable. In many cases the decompiler can figure this out, but especially in looping constructs, it cannot. This is really a dead code issue. The decompiler currently makes judgements about dead code for entire varnodes. A full fix of this problem would require a dead code elimination algorithm that could decide if part of a varnode were dead.

- **ZEXT14(x)** –zero extension

- The 1 is the size of the operand x
- The 4 is the size of the output in bytes

This is almost always a cast from small integer types to big unsigned types.

- **SEXT14(x)** –signed extension

- The 1 is the size of the operand x
- The 4 is the size of the output in bytes

This is probably a cast from a small signed integer into a big signed integer.

- **SBORROW4(x,y)** –true if subtracting the signed numbers would cause a borrow

- The 4 is the size of both x and y in bytes

Returns "true" if there is an arithmetic overflow when subtracting "y" from "x" as signed integers. These are generated particularly by signed integer comparisons. There are rules in place for recovering the original comparison, but this is a missed one special case. These could also conceivably be generated in extended precision subtraction.

- **CARRY4(x,y)** –true if there would be a carry adding x to y

- **SCARRY4(x,y)** –true if there would be a signed overflow adding x to y

- The 4 is the size of both x and y in bytes

Returns "true" if there would be a carry adding x to y.

If these are turning up everywhere in a particular binary, it could be a missed simplification that could be easily fixed.

## Register Settings

Occasionally a program will use a register to store a global constant. By using the <Set Register> function on the right mouse pop-up menu, the user can specify this value to the decompiler. The constant will be propagated automatically throughout the function, and the resulting code may be greatly simplified.

## Decompiler Options

The following Decompiler Analysis Options are available ( Edit->Options Decompiler/Analysis ):

- **Eliminate unreachable code** –causes the decompiler to eliminate branch paths which it considers unreachable as a result of constant propagation.
- **Ignore unimplemented instructions** –causes the decompiler to ignore instructions whose semantics have been marked as unimplemented. Otherwise a *halt\_unimplementedcall* will appear in the decompilation for such cases.
- **Infer constant pointers** –allows the decompiler to infer a data-type for constants it determines are likely pointers. In the basic heuristic, each constant is treated as an address, and if that address starts a known data or function element in the program, the constant is assumed to be a pointer.
- **Respect read-only flags** –causes the decompiler to treat any values in memory or blocks of memory marked read-only as constant values. Normally global memory is considered public writable, meaning you cannot depend on the initial value at a location. Any global value could be changed by another function. For areas of memory that are really read-only and never change their statically initialized value, mark the memory area as read only in the *MemoryManager* or specific Data locations as *Constant* (see Data Mutability below).

Typically as part of the import process, memory blocks are marked as read-only if the memory block is tagged as such in the imported binary.

- **Simplify predication** –causes the decompiler to simplify code that employs conditional (predicated) instructions, merging *if/else* blocks of code that share the same condition.

- **Simplify extended integer operations** –causes the decompiler to simplify integer operations, where a single logical value is split into high and low pieces that are acted on in multiple stages. The decompiler tries to identify these constructions and replaces the multiple stages with a single operation.
- **Use in-place assignment operators** –causes the decompiler to employ in-place C assignment operators such as `+=` in the decompiled syntax.
- **Decompiler Timeout (seconds)** –the number of seconds to allow the decompiler to run before terminating the decompiler. Currently this does not affect the UI, which will run indefinitely. This setting currently only affects background analysis that uses the decompiler syntax.

### Data Mutability

Decompiler output can be influenced by the mutability of data locations within memory. Supported mutability settings include:

- **Read-only/Constant** –indicates that a memory location's value never changes and the currently stored value can be treated as a constant.
- **Volatile** –indicates that a memory location's value may change asynchronously between reads. Reads and writes to such locations are never simplified by the decompiler and are wrapped with specially named function calls (e.g., `volatile_read`, `volatile_write`). The language definition and compiler specification may predefine specific volatile regions of memory and may also override the default volatile read/write function names.

Data mutability may be controlled by the user in one of two ways:

1. [Memory Block Settings](#)
2. [Data Settings](#)

### Tips:



It is important to note that the decompiler is only as good as the definition of the underlying assembly language code. Each assembly instruction has an associated PCODE definition that describes what the instruction does, essentially an RTL (Register Transfer Language). For example, the following MOV instruction which moves a value into an offset onto the stack also has a PCODE definition.

```
MOV local_1c[ESP], 0x804aac8
```

```
temp1 = INT_ADD 0x4, ESP
temp2 = COPY 0x804aac8
STORE ram(temp1), temp2
```

Irregularities in the produced C-code can often be attributed to errors in this underlying definition. Such errors can usually be fixed quickly. [Please feedback any problems or issues you find.](#)

- A good way to start using the decompiler is by defining the parameters to functions that are obviously "char \*" string references. This allows the decompiler to discover and display any static strings referenced anywhere the function is called.
- The decompiler can work out references to fields of a data structure and figure out array indexing given enough information about data types. Building these data type definitions greatly enhances readability of the C-code and is a natural way to encapsulate reverse engineering knowledge. If you notice many offset references from a base value other than the frame or stack pointer, that value is probably pointing to a structure or an array. Notice `psParm1` in the code below. There are several different references off of it. The parameter can be annotated to point to a structure. The user can create a new structure or use one from a Ghidra data type library.

Without knowing the data type, the decompiler produces the following C-code.

<pre>8b 40 0c  MOV  EAX,0xc[EAX] 3b 45 fc  CMP  EAX,local_8[EBP] 79 29     JLE  LAB_080483c6 8b 55 08  MOV  EDX,psParm1[EBP] 8b 45 fc  MOV  EAX,local_8[EBP] 89 42 0c  MOV  0xc,[EDX]EAX 8b 45 08  MOV  EAX,psParm1[EBP] 8b 50 08  MOV  EAX,0x4[EAX] 8b 45 08  MOV  EAX,psParm1[EBP] 8b 40 04  MOV  EAX,0x4[EAX] 89 42 04  MOV  0x4,[EDX]EAX 8b 45 08  MOV  EAX,psParm1[EBP] 8b 50 04  MOV  EDX,0x4[EAX] 8b 45 08  MOV  EAX,psParm1[EBP] 8b 40 08  MOV  EAX,0x8[EAX]</pre>	<pre>dword dVar1; dVar1 = sParm2[*psParm1]; if ((sdword)dVar1 &lt; psParm1[3]) {     psParm1[3] = dVar1;     *(sdword *)psParm1[2] + 4 == psParm1[1];     *(sdword *)psParm1[1] + 8 == psParm1[2]; } else {     if (dVar1 - psParm1[3] == 0) {         psParm1[4] = 100;     }     else {         psParm1[4] = dVar1 - psParm1[3];     } }</pre>
--	--

After applying the appropriate structure, the code becomes:

<pre>8b 40 0c  MOV  EAX,0xc[EAX] 3b 45 fc  CMP  EAX,local_8[EBP] 79 29     JLE  LAB_080483c6 8b 55 08  MOV  EDX,psParm1[EBP] 8b 45 fc  MOV  EAX,local_8[EBP] 89 42 0c  MOV  0xc,[EDX]EAX 8b 45 08  MOV  EAX,psParm1[EBP] 8b 50 08  MOV  EAX,0x4[EAX] 8b 45 08  MOV  EAX,psParm1[EBP] 8b 40 04  MOV  EAX,0x4[EAX] 89 42 04  MOV  0x4,[EDX]EAX 8b 45 08  MOV  EAX,psParm1[EBP] 8b 50 04  MOV  EDX,0x4[EAX] 8b 45 08  MOV  EAX,psParm1[EBP] 8b 40 08  MOV  EAX,0x8[EAX]</pre>	<pre>dword dVar1; dVar1 = sParm2(psParm1-&gt;id); if ((sdword)dVar1 &lt; (sdword)psParm1-&gt;max) {     psParm1-&gt;max = dVar1;     psParm1-&gt;prev-&gt;next == psParm1-&gt;next;     psParm1-&gt;next-&gt;prev == psParm1-&gt;prev; } else {     if (dVar1 - psParm1-&gt;max == 0) {         psParm1-&gt;count = 100;     }     else {         psParm1-&gt;count = dVar1 - psParm1-&gt;max;     } }</pre>
--	--

○ The parameters shown where a function is called may not agree with the parameters where the function is defined. This can be caused by several things:

- The function takes variable arguments.
- The parameters are not actually referenced (used) by the function.
- The decompiler does not see the parameter location being filled.

Parameters determined from the function definition are more likely to be correct.

## Decompiler Window

To display the decompiler window, position the cursor on a function in the Code Browser, then select the icon from the tool bar, or the **Decompile** option from the **Window** menu in the tool.

Errors from the decompiler process are reported in the status area of the tool and sometimes at the end of the C code in the decompiler window.

### Mouse Actions

- **Double-click** – Navigates to the symbol that was clicked.
- **Control-double-click** – Navigates to the symbol that was clicked, opening the results in a new window.
- **Control-shift-click** – Triggers the Listing in a **Snapshots** view to navigate to the address denoted by the symbol that was clicked.
- **Middle-mouse** – If you press the middle mouse button the decompiler will highlight every occurrence of a variable or constant under the current cursor location (the button changed in the tool options under **Browser Field->Cursor Text Highlight**).



You can navigate to the target of a `goto` statement by double-clicking its label (you can also double-click a brace to navigate to the matching brace).

Other actions available in the decompiler are described in the following paragraphs.

### [Copy/Copy Special...](#)

C Code from the decompiler window can be copied and pasted into any other system text window. Select the text to copy, and then choose **Copy** from the popup menu.

### [Comments->Set...](#)

Set a comment on a line of C-Code. The comment will be stored in the program database at the closest assembly line associated with the generated C-Code. Any type of comment (EOL, Post, Pre, Plate) can be attached to the representative C-Code. When this function is re-displayed at some later point, the comment will persist.

### [Commit Params/Return](#)

By default, the decompiler will analyze the code to try to discover function parameters, return type, and local variables. Each time the decompiler displays C-code for a function it does this analysis again. **Commit Params/Return** causes any parameter names and types and return type to be saved in the program database so that next time the function is decompiled the current definitions will be used. This is useful for "syncing" the function signature with the disassembly display. This causes the names and types of parameters and returns in the disassembly to agree with the decompiler names and types.

Ghidra will do stack analysis that will recover parameters and return types, but for many programs, the analysis the decompiler does is better.



There is a prototype plug-in that automatically pulls in the decompiler derived information and applies it to each function as the function is created.



If a variable displayed in the assembly window has an undefined type, the decompiler will still respect the name of the variable.

### [Commit Locals](#)

By default, the decompiler will analyze the code to try to discover function parameters, return type, and local variables. Each time the decompiler displays C-code for a function it does this analysis again. **Commit Locals** causes any local variable names and types to be saved in the program database so that next time the function is decompiled the current local variable definitions will be used. This is useful for "syncing" the local variable definitions with the disassembly display. This causes the names and types of locals in the disassembly to agree with the decompiler names and types.

Ghidra will do stack analysis that will recover local variables on the stack, but for many programs, the analysis the decompiler does is better.



There is a prototype plug-in that automatically pulls in the decompiler derived information and applies it to each function as the function is created. The plugin by default will not commit local variable definitions, either stack or register locals. Committing locals automatically can be turned on by changing the analysis options for the Decompiler Parameter ID plugin. In most cases it is better to commit locals only for certain functions that you really care about, or after the data type definitions (structures, etc...) have settled down for the program you are Reverse Engineering.



If a variable displayed in the assembly window has an undefined type, the decompiler will still respect the name of the variable.

### [Auto Create Structure / Auto Fill in Structure](#)

Automatically creates a structure definition for the pointed to structure, and fills it out based on the references found by the decompiler.

To use this, place the cursor on a function parameter variable, or any variable within a function that is a pointer to a structure. It could currently have a data type of undefined, int, void \*, char \*, etc... For example: `func(int *this)`, (for a C++ this call function).

If the variable is already a structure pointer, any new references found will be added to the structure, even if the structure must grow in size. This is very useful as you find more places the structure is used. If you have already started recovering a portion of a structure and find it used in another function. Retype the variable to be the structure, and then use **Auto Fill in Structure** to add any new fields recovered for the structure.

This feature is also available in the assembly listing when the cursor is placed on a defined parameter or return variable.



Currently this only recovers the structure by following the structure pointer through the current function and any function the structure is passed into within the current function. Eventually this will be put into a global type analyzer, but for now it is most useful interactively.



For best results, the function should be well formed with good flow, and all the switch statements should be recoverable.



There is also a script called **CreateStructure** that you can use for automated structure recovery. For instance if you have a set of ThisCall routines where the first parameter to all the routines is a pointer to a shared class structure, the script could be modified to recover the structure for each this parameter.

### [Highlight Def-Use](#)

Highlights all places a value is used, starting at the place it is first written, and including all the places where that one value is read. This is usually a proper subset of all the places a variable appears in the function. Place the cursor over a variable you would like to highlight and select **Highlight Def-Use** from the pop-up menu.

As an example the `a` at the top of the function is under the cursor when **Highlight Def-Use** is chosen.

```

void max_retry(node *ptr)
{
    dword a;
    sdword b;

    a = ptr->max + 7
    b = ptr->prev->max + a;
    if ((sdword)(ptr->next->max + a) < 7) {
        ptr->max = a;
    }
    else {
        a = ptr->max + 6
    }
    ptr->next->max = a + b
    return;
}

```

Notice that the first three references to *a* are highlighted but the final use of *a* is not because the value might have changed in the *else* clause.

#### Highlight Forward Slice

Highlight Forward Slice highlights each variable whose value may be affected by the value in the variable under the cursor.

As an example, *b*, the output of *max\_alpha*, is under the cursor when Highlight Forward Slice is chosen.

```

a = psParm2->id;
b = max_alpha(psParm1->next,psParm1->id);
c = max_beta(psParm1->prev, a);
c = c + b;
dStack8 = 0;
while (psParm1->count != dStack8 && (sdword)dStack8) {
    if (c < (sdword)(dStack8 + b)) {
        c = c + a;
    }
    else {
        a = a + 10;
    }
    dStack8 = dStack8 + 1;
}
psParm1->count = a + c;
return;

```

We can see that *c* is tainted by the value of *b* all the way through to the bottom of the function.

#### Highlight Backward Slice

Highlight Backward Slice highlights all points in the function that contain a value involved in the creation of the value in the variable under the cursor.

As an example the final *a* of the function is under the cursor when Highlight Backward Slice is chosen.

```

a = psParm2->id;
b = max_alpha(psParm1->next,psParm1->id);
c = max_beta(psParm1->prev, a);
c = c + b;
dStack8 = 0;
while (psParm1->count != dStack8 && (sdword)dStack8) {
    if (c < (sdword)(dStack8 + b)) {
        c = c + a;
    }
    else {
        a = a + 10;
    }
    dStack8 = dStack8 + 1;
}
psParm1->count = b + c;
return;

```

We can see that the final value of *a* is affected by the loop and by the input parameter but never by *b* and *c*.

#### Highlight Forward Instruction Slice

Highlight Forward Inst Slice highlights each instruction whose value may be affected by the value in the variable under the cursor, rather than the values themselves.

#### Highlight Backward Instruction Slice

Highlight Backward Inst Slice highlights all instructions in the function that contribute to the creation of the value in the variable under the cursor.

#### Rename Variable

Any parameter or local variable can be renamed. Just place the cursor over a variable definition, or any use of the variable and choose **Rename Variable** from the popup menu. The name will now be saved for this function, so the next time the decompiler displays the code for the function, the same name is used.

#### Rename Function

A shortcut for renaming the function from within the decompiler window.

#### Retype Variable

The decompiler does its best to recover the type of a variable automatically but often only has limited information for analysis. Explicitly changing the type of a variable can dramatically improve the C-code produced. This is especially true for structures. Changing the type of a parameter variable will affect the display for every place the function is called.

To change a variables data type; place the cursor over the variable definition or use of the variable, select **Retype Variable** from the popup menu, and then enter the name of the type. The name of any data type known to the program can be used.

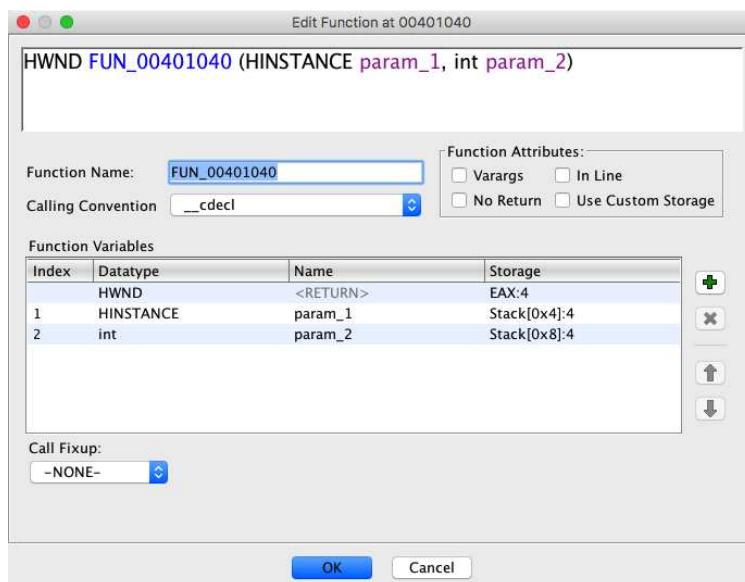
A simple code improvement is to locate any functions with obvious string parameters and re-type the parameter to be a "char \*". Any references to defined memory will now display the passed parameter as a character "string".

#### Edit Data Type of Variable

Only structure, union, and enum data types can be edited. If a variable's data type is one of these it can be edited. Also, if the data type is a type definition, array, or pointer based on an editable data type, then the base data type can be edited. For example, if you have a structure pointer for a variable then you can edit the structure. To edit a variable's data type; place the cursor over the variable definition or use of the variable and select **Edit Data Type...** from the popup menu. For structures and unions, the [structure editor](#) will appear, and for enums the [enum editor](#) will appear. If the data type for a variable can't be edited, the action will be disabled in the popup.

#### Edit Function Signature

The **Edit Function Signature** dialog allows you to change the function's signature, the calling convention, whether the function is inline and whether the function has no return.



The function signature includes

- function name
- return type
- number of parameters
- parameter names
- parameter type
- varargs (variable arguments)

This features allows you to edit a function signature text string to change any of these.

For example if a function is actually printf(), instead of changing the name, return type, and parameters individually, the entire function signature can be changed all at once. To do this you could enter

`void printf( char *fmt, ... )`

within the **Signature** field and then select the OK button.

In addition, you can select the **Calling Convention** for this function from a list of available calling conventions as determined by the program's language. Selecting the **Inline** checkbox indicates that the function is in-lined. Selecting the **No Return** checkbox indicates that the function does not return.



The signature of the current function, or any called function can be changed.

To edit a function's signature from the **Decompile** window. Just place the cursor over any function name, select **Edit Function Signature** from the popup menu, and the dialog will appear with the function's current information.

#### Override Signature

Overrides the signature of a called function at the point it is called. This allows you to set the parameter values for a particular call.

#### Remove Signature Override

This action allows you to remove a previously added function signature override.

#### Find...

Find any string of text within the currently decompiled function.

#### Debug Function Decomilation

For certain functions, the decompiler may produce an error message, produce incorrect code, or simply exit without producing results. Selecting ▾**Debug Function Decomposition**, from the decompiler provider window toolbar, will run the decompiler again, and save all relevant information to an XML file. Instead of submitting the entire program to be analyzed to discover the problem, only a small XML file is needed.

#### **Graph AST Control Flow**

Selecting ▾**Graph AST Control Flow**, from the decompiler provider window toolbar, will generate an abstract syntax tree (AST) control flow graph based upon the decompiler results and render the graph within the current Graph Service.



If no Graph Service is available then this action will not be present.

#### **Export to C**

You can export the current decompiled function to a file by selecting the icon in the local tool bar of the decompiler window. A file chooser dialog is displayed for you to select the name of the output file. If you do not specify a file extension, ".c" is appended to the filename.

#### **Snapshot**

Creates a **Snapshot** of the current decompiler window, which allows you to leave the current decompiled function in place while navigating to other functions.

#### **Properties**

The colors used in the decompiler window can be changed by editing the C Display Options through the *Edit Options* dialog. To edit the options, choose **Edit ➔ Tool Options...** from the tool menu. Click on the *C Display node* in the Options tree. A panel shows the colors that can be customized. Click on the color bar to bring up the color chooser to change the color.

The other options allow you to change the maximum characters in a line displayed in the decompiler window, and the number of characters for indenting in the code.

#### **Mouse Hovers**

These function similarly to [Code Browser Mouse Hovers](#)

Provided by: *Decompiler Plugin*

Related Topics:

- [Code Browser](#)
- [Snapshots](#)

# Eclipse Integration

Ghidra is capable of integrating with an existing Eclipse installation that has the GhidraDev Eclipse plugin installed. The GhidraDev Eclipse plugin listens for socket connections from Ghidra on configurable ports. Ghidra does not have to know the installation location of Eclipse for this communication to succeed. However, in order for Ghidra to launch Eclipse if it has not been opened already, Ghidra must know the installation location of Eclipse.

The Eclipse installation location and communication ports can be configured in the Front End tool's ***Eclipse Integration*** options. If an attempt is made to launch Eclipse from Ghidra and these things are not configured correctly, the user will be taken to Eclipse Integration options automatically./P>

## Eclipse Integration Tool Options

Tool Options	
Option	Description
Automatically install GhidraDev	Automatically install the GhidraDev plugin into the "dropins" directory of the specified Eclipse if it has not yet been installed. If this option is not set, you will be prompted to perform the installation when launching an Eclipse that does not have GhidraDev installed.
Eclipse Installation Directory	Path to an Eclipse installation directory. This is only necessary if you want to launch Eclipse from Ghidra. Ghidra will still be capable of communicating with an

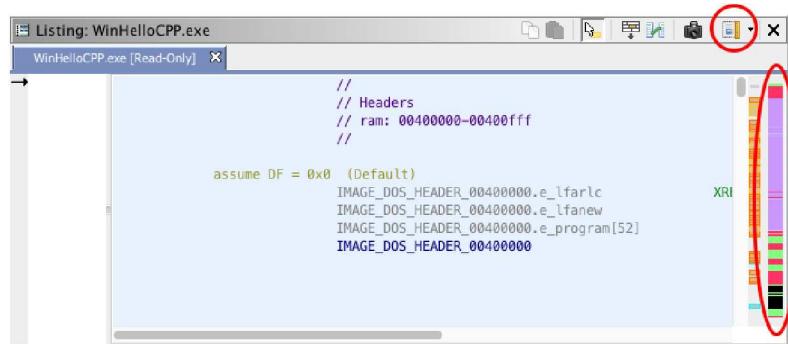
	open Eclipse as long as the GhidraDev plugin is installed and the communication ports are configured correctly on both ends.
Eclipse Workspace Directory (optional)	Optional path to an Eclipse workspace directory. If defined and the directory does not exist, Eclipse will create it. If undefined, Eclipse will be responsible for selecting the workspace directory.
Script Editor Port	The port number used to communicate with Eclipse for script editing. It must match the port number set in the Eclipse GhidraDev plugin preference page in order for them to communicate.
Symbol Lookup Port	The port number used to communicate with Eclipse for symbol lookup. It must match the port number set in the Eclipse GhidraDev plugin preference page in order for them to communicate.

## GhidraDev Eclipse Plugin

For more information on installing and using the GhidraDev Eclipse plugin, see [\*Extensions/Eclipse/GhidraDev/GhidraDev\\_README.html\*](#)

## Overview Bars

Ghidra supports multiple margin bars on the left side of the Listing that present various overviews for a program.



Each horizontal slice in the margin bar represents a relative address location in the program and is colored to indicate the property associated with that address or region in the program. The range of addresses represented in the margin bar is determined by the current view which by default is the entire address space of the program.

- Hovering the mouse on the margin bar will cause a tooltip to appear that gives you detailed information about the property and the address for that location.
- Left-clicking on the margin bar will navigate the listing to the associated address for the pixel that was clicked.
- Right-clicking on the margin bar will bring up a popup-menu which will at least include an option for displaying a legend for that particular overview.

Overview margin bars can be turned on or off using the control button on the Listing's toolbar.

## Overview Bar Types

### General Overview Bar (Address Type)

The Address Type Overview Display shows a high-level view of the currently open program. Different colors are used to represent different types present in the program. For each address as determined by the vertical pixel location, the program is consulted for what is at that address.

The order of precedence for the coloring is as follows:

1. **Function** –the address is within a function.
2. **External** –the address has references to external locations.
3. **Instruction** –there is an instruction at the address that is not currently defined to be in a function.
4. **Defined Data** –a datatype has been applied to that address.
5. **Undefined Data** –the address has an associated byte value, but no datatype has been applied.
6. **Uninitialized** –None of the above. The address falls in an uninitialized memory block (no byte values)

Note that the overview panel only provides an approximation of the contents of a program. Although the level of detail can be increased by selected a more restricted view, there may still be imprecise summaries. For instance, if a region of memory contains mostly defined data, but the particular address that is rendered in the overview panel falls on an undefined block, the color for that pixel is set to undefined. In practice, however, this gives a good general sense of the various regions.

### Legend



The legend indicates the colors that correspond to each type of program element shown in the overview display. The colors are specified as options and can be changed from the default values. To change the colors, click on the color, or edit the options through the **Edit → Options...** dialog. You can choose the color from a color chooser dialog.

### Options

The Overview display has options that you can change through the *Options* dialog:

- Data Color –color for defined data
- External Reference Color –color for external references
- Function Color –color for functions
- Instruction Color –color instructions
- Undefined Color –color for undefined bytes
- Uninitialized Color –color for memory that is not initialized

To view the options, select **Edit → Options...** on the tool, then choose the *Overview* node in the options tree. To change a color, double click on the color bar in the *Overview Options* panel. Choose the color from the color chooser dialog.

## Entropy Overview Bar

The entropy overview bar provides a byte based entropy statistic across the address set represented by the overview bar. The statistic can frequently distinguish between the encoding complexity of different types of data typically present in binary executables, such as machine code, ASCII, and compressed data. An overview of this entropy score can often provide an at-a-glance classification of the program into its major sections and sub-sections, without requiring the presence of an image format header.

### Calculation of Entropy

Entropy provides an estimate of the amount of *variation* in a set of data. For this plugin the data consists of the original bytes in the binary. Viewing the program as one long sequence of bytes, this sequence is split up into **chunks** with a default size of 1024 bytes per chunk. By calculating a histogram of all possible byte values, 0–255, we can easily calculate the probability,  $p(x)$ , of any particular value,  $x$ , occurring in that chunk. The **entropy** of this probability distribution is defined as:

$$\sum_{i=0}^{255} -p(i) \cdot \log_2(p(i))$$

This gives a single value, between 0.0 and 8.0, describing the amount of variation in that single chunk. A score of 0.0 indicates that only a single byte value occurred throughout the entire chunk, so the chunk can be described as having no variation or no entropy. The score can vary continuously through 8.0, or full entropy, which indicates that every possible byte value occurs equally often within the chunk.

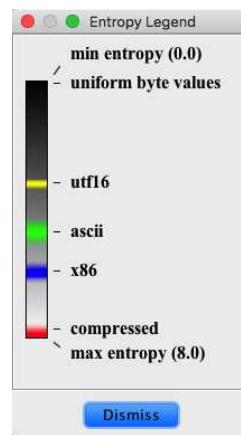
### Data Encoding

Most data encoding schemes show a bias in favor of certain byte values at the expense of others. ASCII, for instance, encodes only byte values between 0 and 127, and if the ASCII is being used to encode (English) error messages in a binary, there will be a further bias for the ASCII ranges encoding alphanumeric characters. Entropy picks up on this bias, and for many schemes, a chunk of data encoded with it will exhibit an entropy value in a very restricted range. ASCII error messages usually fall in the range 4.2 – 5.2. The Entropy Plugin can color-code these ranges so that certain encodings stand out immediately in the overview window. Because entropy is statistical in nature, a specific chunk of encoded data may not have an entropy value that falls inside the typical range. But across an entire program, the bias for particular ranges will be readily apparent, and major sections will stand out clearly.

Entropy can easily distinguish between these common data encodings.

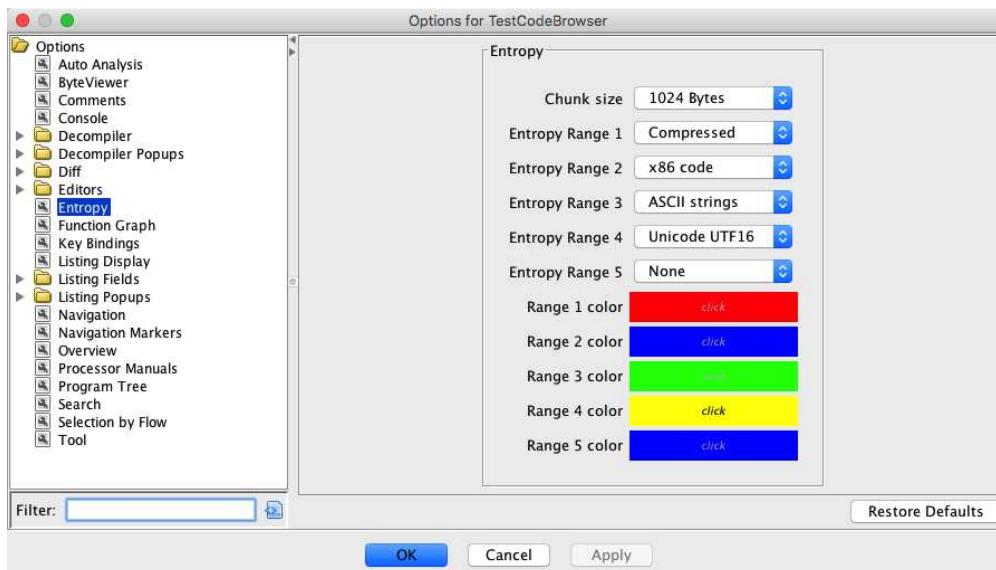
- **x86 Machine Code:** A specific instruction set like the Intel x86 has a very characteristic entropy range, which is well short of compression schemes, but packs more information per byte typically than ASCII. Different coding styles, compilers, etc. may have a consistent impact on the exact range of entropy values, but in general any block of machine code is easy to pick out.
- **ARM/THUMB Machine Code:** There are two machine code specifications for ARM chips: ARM instructions and THUMB instructions. These both have entropy ranges similar to x86 machine code, but the ranges for ARM vs THUMB are distinguishable. ARM instructions, which must use 4 bytes per instruction, are slightly more wasteful in their encoding than THUMB, and this stands out in their entropy range.
- **ASCII:** Entropy scores for ASCII encoded strings show its characteristic waste of the high bit in each byte and other biases for English letter frequency, null terminators, etc.
- **Unicode:** The *wide character* format typically used to encode Unicode characters is particularly wasteful, with every other byte encoded as 0 for typical English strings. This shows up as a characteristic range of low entropy values.
- **Compression/Encryption:** Data that has been compressed and/or encrypted typically shows very little bias at all in the byte values, and this corresponds to entropy scores very close to the maximum value of 8.0. Although entropy generally has little chance of distinguishing between different *kinds* of compression or encryption, this general category of encoding stands out quite clearly from other data typically found in a program.

### Color Palette Legend



Each color in the main bar encodes a specific entropy value, which can be determined by referring to the color palette which can be displayed by right-clicking on the bar and selecting the "show legend" action. The basic palette encodes entropy scores as a gradient, from black to white, for entropy scores from 0.0 to 8.0. In addition to this basic palette, the user can configure specific ranges to stand out with a specific color, which gets added into the base palette as a smaller color gradient. Multiple entropy ranges can be incorporated as distinct color gradients into the single palette. Each defined color range also has a label describing that range.

### Configuring the Entropy Window



Select the **Tool Options...** entry of the Code Browser **Edit** menu, and then choose **Entropy** from the tree navigator at the left of the Options dialog. This allows the user to configure different ranges incorporated into the palette and the size of chunk used in calculating a single entropy score. The Entropy Plugin has the following options:

#### **Chunk size**

The chunk size can be set to a value of 1024, 512, or 256 bytes. This controls over how many bytes a single entropy score is calculated. To a small extent, the user can trade off the granularity of the Entropy window with how much variation to expect across an entire region of similarly encoded data.

#### **Entropy Range #**

The Entropy window color palette supports up to 5 different highlighted ranges. For each of the 5 slots, this option presents a drop menu of common entropy ranges that can be selected. These include: **x86 code**, **ARM code**, **THUMB code**, **PowerPC code**, **ASCII strings**, **Compressed**, and **Unicode UTF16**. Slots that are unused can be set to **None**.

#### **Range # color**

The color that is used to highlight a specific range can be set with this option. Entropy values that hit the exact middle of the range will get assigned to the chosen color, and a steep gradient, connecting this color with the bounding colors within the base palette, will be used to fill out the color range.

Provided By: *OverviewPlugin*

Related Topics:

- [Code Browser](#)
- [Current View](#)
- [Edit Options Dialog](#)

# FileSystem Browser

## Introduction

The file system browser is a generic tool for browsing and accessing the contents of filesystems or container files (such as zips, tars, firmware images, etc).

## GHIDRA Tool File Menu Actions

### Open File System

Opens a file system container file (ie. a zip, tar, iso, etc) in a new browser tree.  
Subdirectories of your local computer's file system can also be opened in this manner.

## Right-click Context Menu Actions

### Get Info

Returns information about the selected file.  
Sometimes there will not be any available information. Generally, this information is not that useful. It will mostly consist of meta-data from the internal file system.

### Expand All

Expands all folders below, and including, the selected

node.

### [Collapse All](#)

Collapses all folders below, and including, the selected node.

### [Open File System](#)

Attempts to open the selected file as a sub-file-system. If this operation succeeds, the node will turn into a folder with one or more children. If this operation fails, the node will remain a leaf node. This operation could fail for many reasons, but generally it fails because the node does not represent a valid sub-file-system.

### [Open File System in new window](#)

Attempts to open the selected file as a sub-file-system. If this operation succeeds, a new file system browsing tree will be shown with the contents of the selected file.

### [Open Program\(s\)](#)

Opens the GHIDRA program(s) that correspond to the selected file(s). If no program in the current GHIDRA project is linked to the selected file, you will be able to import the selected file.

### [Import](#)

Imports the selected file into GHIDRA as new program in your current project.

## **Batch Import**

Imports the selected file(s) into GHIDRA as new programs in your current project using the Batch Import dialog.

## **Export**

Writes a copy of the selected files to a directory you select on your local computer.

## **Export All**

Recursively copies the contents of a selected folder to a directory you select on your local computer.

## **View As Image**

Attempts to render the selected file as an image.

## **View As Text**

Attempts to render the selected file as an text.

## **List Mounted File Systems**

Displays a list of the file systems that are currently open and mounted. Selecting one of the file systems will display that file system's browser tree.

## **Close**

Closes the currently highlighted file system root node. The file system itself will not be unmounted until all

open browser windows to it are closed and a caching timeout period has passed.

## Browser Dialog Actions

### Display Supported File Systems and Loaders

Lists the currently supported file systems.

### Open File System Chooser

Opens a new file system container file in a new browser tree. This is the same as "File | Open File System" in the main GHIDRA window.

## How To Handle Unsupported File Systems

If you receive this message: **No file system provider for the selected file.**

It means one of the following two things:

1. The file you attempted to open as a file-system is actually NOT a file-system
2. GHIDRA does not have an implementation for that file-system

If the file does not really represent a file system, then you may want to try importing it.

Otherwise new file-systems can easily be written by implementing the `ghidra.formats.gfilesystem.GFileSystemInterface`.

## Known issues

## Strong Crypto Support

Your Java JVM install may not have support for strong crypto currently installed.

In order to fix this issue, you must install Oracle's "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files"

## Function Bit Patterns Explorer Plugin

The Function Bit Patterns Explorer Plugin is used to discover patterns in the bytes around function starts and returns. When analyzing a single program, such patterns can be used to discover new functions based on the functions that have already been found.

The explorer can also be used to analyze a collection of XML files containing the function start/return information for a [collection of binaries](#). Such patterns can be used to guide the **Function Start Analyzer** during auto-analysis.

To bring up the explorer, select **Window > Function BitPatterns Explorer** from the Code Browser.

### Data Sources

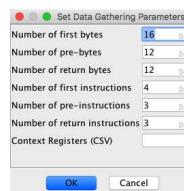
#### Current Program

Use the "Gather Data from Current Program" button to gather data to analyze from the current program. You can also select **Tools > Explore Function Bit Patterns** in the Code Browser.

#### Directory of XML Files

Use the "Read XML Files" button to select a directory of XML files containing data from windows around function start/returns (one XML file per program). To generate these XML files, run the script `DumpFunctionPatternInfoScript.java`

### Data Gathering Parameters



Several parameters control how much data is gathered around each function. When analyzing a single program, a dialog will pop up which allows you to enter values for these parameters. When running the script `DumpFunctionPatternInfoScript.java` you can set these parameters by editing the file `DumpFunctionPatternInfoScript.properties`. The parameters are:

#### Number of First Bytes

The number of bytes to gather starting at the entry point of the function.

#### Number of Pre-Bytes

The number of bytes to gather immediately before (but not including) the entry point of a function.

#### Number of Return Bytes

The number of bytes to gather immediately before (and including) a *return* statement in a function.

#### Number of First Instructions

The number of instructions to gather starting at the entry point of a function.

#### Number of Pre-Instructions

The number of instruction to gather immediately before (and not including) a function start.

#### Number of Return Instructions

The number of instructions to gather immediately before (and including) a function start.

#### Context Registers

The context registers whose values you wish to record. Enter a comma-separated list of registers into this field. For example: `TMode,LMode`

Recommended Parameters:  
Reasonable starting values for the parameters controlling the number of instructions to be gathered are 3, 4, and 5. When setting the number of bytes to gather, it's reasonable to choose a value that can hold most of the corresponding instruction sequences. For example, suppose you're examining x86 programs and set the number of first bytes to gather to 16. A reasonable number of first bytes to gather is 20, which should be enough to hold most 4-instruction sequences (even though the maximum length of an instruction on x86 is 15 bytes).

### Data Views

The main interface of this plugin is a panel with multiple tabs. All tabs, except for the Pattern Clipboard tab, are auto-populated, either after reading a directory of XML files or clicking "OK" on the Data Gathering Parameters dialog.

Each tab displays a different view of the gathered data:

#### Byte Sequence Tabs

#### Instruction Sequence Tabs

#### Function Start Alignment Tab

#### Context Register Information Tab

#### Pattern Clipboard

#### Byte Sequence Tabs

There are three byte sequence tabs: one for first bytes, one for pre-bytes, and one for return bytes. Two types of filters can be applied to byte sequences: *length filters* and *context register filters*. Length filters are required, but context register filters are optional.

#### Length Filters

A length filter requires two pieces of data: a minimum sequence length and a prefix/suffix length. The filter filters out all sequences which do not meet the minimum length constraint. For each sequence that does meet the constraint, it takes either a prefix or a suffix of appropriate length (suffixes are specified as a negative number in the input dialog for the filter data).

#### [ContextRegisterFilters](#)

These allow you to filter sequences by specifying values for some or all of the tracked context registers.

**Note:** Filters for byte sequences are not shared between tabs.

After applying such a filter, there will be a table containing bytes sequences, all of which have the same size. Select some rows in the table, right click, and select [Analyze Sequences](#) to look for patterns.

#### [InstructionSequence Tabs](#)

Similar to byte sequences, there are three instructions sequence tabs, containing first instructions, pre instructions, and return instructions, respective. These sequences are sorted into a tree. Note that the length of an instruction is taken into account. For example, sequences which begin with a one-byte *PUSH* instruction will go through a different path in the tree than sequences with begin with a two-byte *PUSH* instruction. There are two optional filters which you can apply to instruction sequences:

#### [PercentageFilters](#)

Filtering by X% will remove a node from the tree if the percentage of paths going through the node is less than X%.

#### [ContextRegisterFilters](#)

These allow you to filter sequences by specifying values for some or all of the tracked context registers.

**Note:** Filters for instruction sequences are not shared between tabs.

If you hover over a node, you can see the percentage of all paths in the tree which go through that node. To search for patterns in the byte sequences corresponding to a given node, right click on the node and select [Analyze Sequences](#).

#### [Function Start Alignment Tab](#)

This tab displays with  $n$  rows, where  $n$  is the specified alignment modulus. The number in row  $i$  is the number of functions whose addresses modulo the alignment modulus is equal to  $i$ . This allows you to determine whether function starts are aligned within the program (for example, on x64, compilers will frequently 16-byte align function starts at higher optimization levels). If you know that functions are aligned along a certain boundary, you don't have to search for function starts in the non-aligned bytes.

#### [Context Register Information Tab](#)

This tab displays all values recorded for the context registers you specified.

#### [Pattern Clipboard](#)

You can send patterns you find to the pattern clipboard for evaluation. In the clipboard, there are two types of patterns: PRE and POST. PRE patterns correspond to patterns that occur before the start of a function. Patterns from pre-byte and pre-instructions sequences are considered PRE patterns, as are patterns from return byte and return instruction sequences (the idea being that the return is followed by the start of another function). Patterns from first byte and first instruction sequences are considered POST patterns.

You can edit the "Alignment" column in the pattern clipboard. The context register column is populated from context register filters applied while exploring the data.

#### [Evaluating Patterns](#)

You can evaluate a selection of patterns in the clipboard by selecting them, right-clicking, and performing the "Evaluate Selected Patterns" action. This will search for the patterns in the current program (if there are both PRE and POST patterns, they will be combined). A table will pop up which displays all of the matches, information about each match, and aggregated information about all of the matches.

#### [Clipboard Buttons](#)

- Create Functions from the selected patterns.
- Export selected patterns to a pattern file. Such files can be used by the [Function Start Analyzer](#) during Auto Analysis. A dialog will appear asking for two values: TotalBits and PostBits. When the [Function Start Analyzer](#) reads in a pattern file, it makes a set patterns. This set consists of each PRE pattern concatenated with each POST pattern for which the concatenation has at least TotalBits fixed bits, at least PostBits of which must be after the PRE bits.
- Import patterns from a pattern file. **Note:** You should only do this with files generated by this plugin. Arbitrary XML files from the *Processors* directory may contain attributes not supported by this plugin.

#### [Analyzing Byte Sequences](#)

Byte sequences to analyze are displayed in a table along with information about each sequence, such as the number of occurrences the sequence or (possibly) the disassembly of the sequence. You can make a selection of rows in this table, right-click, and perform the following actions:

#### [Send Selected to Clipboard](#)

This will send the selected sequences to the Pattern Clipboard.

#### [MergeSelectedRows](#)

This will merge the selected sequences into one sequence. For a given bit position in the merged sequence, if all selected sequences agree on that position the the merge will contain that value, otherwise it will contain a dit in that position.

#### [Send Merged to Clipboard](#)

If you've merged a select of sequences, there will be an action to send the merged sequence to the pattern clipboard.

#### [Mine Sequential Patterns](#)

If the sequences you're analyzing came from a byte sequence tab, there will be an action to [Mine Sequential Patterns](#).

#### [Mining Closed Sequential Patterns](#)

A **Closed Sequential Pattern** is a pattern such that no proper super-pattern occurs more frequently in the sequences that you're analyzing. For example, suppose the sequence "111" occurs ten times. Then the sequences "11.", ".11", and ".11" also occur (at least) ten times. We'd like to avoid a combinatorial explosion of patterns; restricting to closed patterns ensures that any sub-patterns which are listed are strictly more common than the main pattern.

Before actually running the mining algorithm, a dialog will appear which asks you to set some parameters:

#### [Minimum Support Percentage](#)

The algorithm will only return patterns which occur in at least this percentage of the data being analyzed.

#### [Minimum Number of Fixed Bits](#)

Any pattern returned by the algorithm will contain at least this many non-dit bits.

#### [BinarySequences vs. CharacterSequences](#)

This allows you to treat sequences as either sequences of characters (nibble) or sequences of bits. If bit sequences take too long to mine, you can try the character sequences option, which will find fewer patterns but will run much faster.

**Note:** The longer the algorithm runs, the faster the progress bar will advance, so don't be too dismayed if it initially seems to be taking a lot of time.

## Function ID

[Next](#)

# Function ID

```

100129a0 5d          POP      EBP
100129a1 c2 04 00    RET      0x4

*****  

* Library Function - Single Match  

* Name: _onexit  

* Library: Visual Studio 2010 Release  

*****  

<ANALYSIS>   _onexit_t __cdecl _onexit(_onexit_t param_1)  

_onexit_t      EAX:4           <RETURN>  

_onexit_t      Stack[0x4]:4  param_1           XREF[2]: 10012  

                           10012  

                           _atexit:10012a  

                           XREF[1]:  

100129a4 6a 14        PUSH     0x14  

100129a6 68 78 68 01 10  PUSH     DAT_10016878  

100129ab e8 b0 00 00 00  CALL     _SEH_prolog4  

100129h0 ff 35 00 cc 01 10  PUSH     dword ptr [DAT 1001cc000]

```

## Overview

Function ID is an analyzer that performs function identification analysis on a program. For each function, the analyzer computes a hash of its body and uses this as a key to look up matching functions in an indexed database of known software. The recovered function name, and other meta-data about the *library* it is contained in, is applied to the program.

Function ID is suitable for identifying statically linked libraries or other software where the compiled form of the functions does not change. Because of the hashing strategy, functions remain identifiable even if the library is relocated during linking. Larger changes to the compilation process of the library however will likely prevent successful searches. Function ID databases are necessarily targeted to a specific processor.

Function ID generally runs as part of **Auto Analysis** but can also be initiated at any time by the user from the **One Shot** menu. Ghidra also comes with a [\*Function ID Plug-in\*](#), which provides more control over which databases to apply, and allows users to create and populate their own databases.

By default, Ghidra ships with databases that search for statically linked libraries from Microsoft Visual Studio for the x86 processor. These have been broken apart into separate Function ID databases, based on 32-bit or 64-bit and the version of Visual Studio. Within each database, there are two library variants —one for debug versions and one for production.

## Hashing

Function ID works by calculating a cumulative hash over all the machine *instructions* that make up the body of a function. For each function, two different 64-bit hashes are computed: a **full hash** and a **specific hash**. Both schemes hash the individual instructions of the function body in address order, but they differ in the amount of information they include from each instruction.

**full hash**

This hash includes the mnemonic and some of the addressing mode information from an instruction. Specific register operands are also included as part of the hash, but the specific value of constant operands are not.

**specific hash**

This hash includes everything used in the full hash but may also include the specific values of any constant operands. A heuristic is employed that attempts to determine if the constant is not part of an address, in which case the value is accumulated into the hash.

Both hashes are used to identify matches in a database. The **full hash** is robust against changes due to linking; the **specific hash** helps distinguish between closely related variants of a function.

## Parents and Children

When Function ID examines a function, its parent and child functions are also considered as a way of disambiguating multiple matches. For example, suppose two functions have identical sequences of instructions, except they each call to a different subfunction. In this situation, the full hashes of the functions will be identical, but the system will try to match the hash of one of the two subfunctions, allowing it to distinguish between the two.

## Libraries

Within a Function ID database, functions are grouped into *libraries*, which are intended to be recognizable named software components that get linked into larger programs. A **Library** has the following meta-data.

Name	This is a general descriptive name for the library.
Version	If there is a formal version number for the library, this field will hold this value as a string.
Variant	Version information that cannot be encoded in the formal <i>Version</i> field can be encoded in this field. This is used typically for <i>compiler settings</i> , which affect Function ID hashes but don't make sense in a version string describing the source code.

Generally, the analyzer is able to report all three fields describing the library for any function match it finds. In the case where a database contains multiple versions of the same library, it's common for a function to match into two or more libraries that differ in their *Version* or *Variant* field. In this case, the analyzer will still report a single match but will leave off the fields it couldn't distinguish.

## Single Matches

A **Single Match** for a function occurs under the following conditions:

- The analyzer can narrow down potential matches to a single function name.
- The function does not already have an imported or user-defined name.
- The number of instructions in the function exceeds the *instruction count threshold*.

Even if there are multiple potential matches in the database, the first condition may still hold because they all share the same base function name. The second condition does not need to apply if the "Always apply FID labels" option is toggled on (See "[Analysis Options](#)"). The number of instructions is computed as the *matchscore* and can include counts of instructions in parent or child functions. For details about the match score and thresholds, see "[Scoring and Disambiguation](#)".

If there is a Single Match, the analyzer will:

1. Apply the function name as a symbol.
2. Insert a comment describing the matching library.
3. Add a *Function ID Analyzer* bookmark.

Both the inserted comment and bookmark will include the phrase "Single Match".

## Multiple Matches

If the analyzer is not able to narrow down to a single function name, even after applying all of its disambiguation logic, then the reporting behavior depends on the remaining match scores. If they are too small the matches are deemed to be random, and nothing is reported at all. Otherwise, a **Multiple Match** is reported. In this case, multiple symbols and comments will be inserted, one for each remaining match, up to an arbitrary limit. All the comments will contain the phrase "Multiple Matches".

## Analysis Options

This analyzer appears under the heading **Function ID** in the dialog listing the standard analyzers whenever the user elects to auto-analyze a new program upon import, or by selecting **Auto Analyze** under the Code Browser's **Analysis** menu. From this dialog, users can toggle whether the analyzer is active or not, and if **Function ID** is selected and toggled on, the dialog presents some configurable options for the analyzer.

### Instruction count threshold

This is the **primary threshold** a potential match must exceed in order to be reported by the analyzer. This defends against *false positives* caused by randomly matching small functions. Roughly, the score counts the number of instructions in the function plus instructions in any matching parent or child.

### Multiple match threshold

In general for a single function, if there are multiple potential matches all with different names, this is a good sign that the matches are random, and the analyzer will not report a match. But if the match score exceeds this threshold, the analyzer will report a *Multiple Match*. (See "[Multiple Matches](#)")

#### **Always apply FID labels**

If this toggle is on, the analyzer will report matches even if there is already an imported or user defined name for the function.

#### **Create Analysis Bookmarks**

The analyzer will only create bookmarks for matches if this toggle is on. This does not affect insertion of comments and symbols.

## Scoring and Disambiguation

The Function ID analyzer assigns a **match score** to each of the potential matches it discovers in its database. The score is used both to filter matches which are too small to be significant and to disambiguate between potential matches.

The basic unit of the score is a single matching instruction with no constant operands, which receives a score of 1.0. Certain instructions, such as calls and *no operation* instructions are assigned a score of zero. Constant operands, in the rare case that they match via the *specific hash*, contribute an additional 0.67 units per operand.

Once a potential match is discovered, it is assigned a score based on:

- Instructions in the body of the function.
- Constant operands that match in the body of the function.
- Instructions in the body of any child function that also has a match.
- Instructions in the body of any parent function that also has a match.

Once scores are assigned, potential matches are filtered based on the *instruction count threshold* (See "[Analysis Options](#)"). This helps prevent small functions that randomly match database entries from being reported. Note however that a small function can still be correctly reported if its parent or child functions also have matches, increasing its overall score. If there are still more than one potential match, the highest assigned score is used to filter out matches with lower scores.

## Matching Function Names

If there are still multiple potential matches once thresholds have been applied to the match scores, the remaining matches will be grouped based on function names. If two potential matches share the same function name, they are grouped together. If the remaining matches can all be placed into a group sharing a single name, the result will still be reported as a "Single Match".

Function names are considered to match if their *base* names match. The base name is obtained by stripping off any namespace from the symbol plus any initial underscores. If the name is mangled, an attempt is made to demangle it first, then namespace and parameter information is stripped.

[Next](#)

Function ID

Function ID Plug-in

## Function ID Plug-in

[Prev](#)[Next](#)

# Function ID Plug-in

The Function ID Plug-in allows users to create new (.fidb) databases. A Function ID database is a collection of meta-data describing software libraries and the functions they contain. It is searchable via a hash computed over the body of unknown functions (See [Function ID](#)). The databases are self-contained and can be shared among different users. Using this plug-in, databases can be created, attached, and detached from an active Code Browser, and a database can be populated with function hashes from programs in the current Ghidra project.

A Function ID database may hold many distinct libraries. Within a single library, all functions must come from the same *processormodel* as determined by Ghidra's Language ID. Ideally, functions should come from a single software component, all compiled using the same compiler and settings.

Functions for a single library can be ingested from multiple files, usually from a series of *object* files analyzed within a Ghidra repository. If the library is spread across multiple files, accurate symbol information is necessary to properly compute parent/child relationships.

## Enabling the Plug-in

All plug-in functionality is accessible from the **Function ID** menu, under the main **Tools** menu. In order to access this menu, the tool must be configured to include the Function ID plugin. To do this, from the Code Browser select  
**File -> Configure**

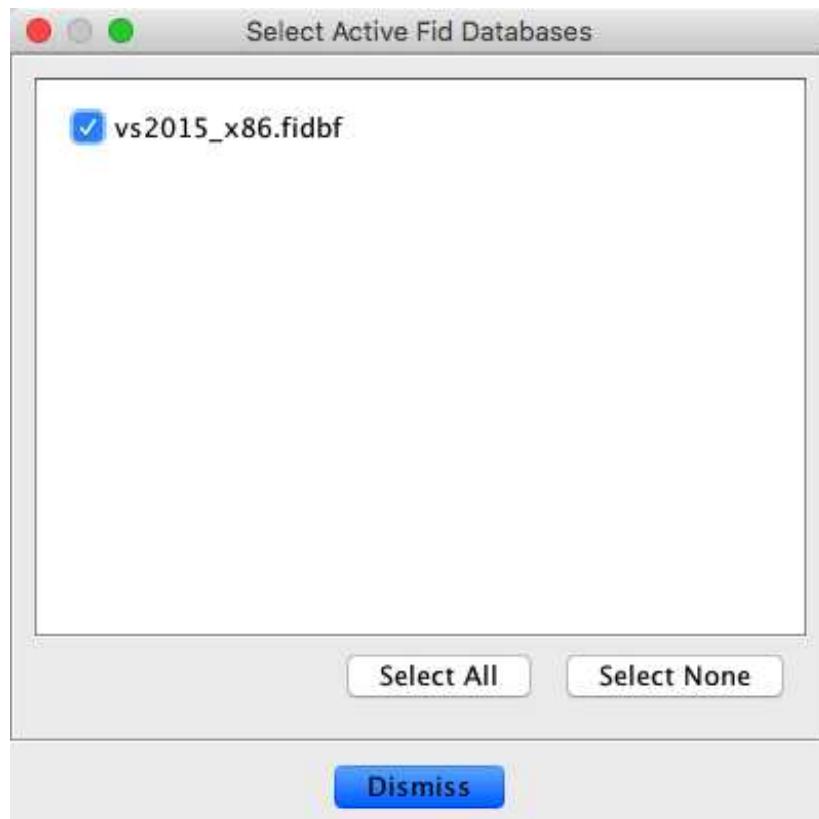
Then click on the *Configure* link under the **Function ID** section and check the box next to "FidPlugin".

## Plug-in Functionality

The Function ID plug-in provides the following actions under **Tools → Function ID**.

### [Choose active FidDbs...](#)

This brings up a dialog that allows the user to select which Function ID databases are active. By default, Ghidra ships with a set of databases and they are all initially active.



Once a database has been deactivated, it will no longer be used for matches in subsequent analysis. The selections for which databases are active are saved as a preference on a per-user basis.

#### **[Create new empty FidDb...](#)**

This brings up a file chooser dialog that allows the user to create a new Function ID database. This cannot be located under the Ghidra install directory root, because Ghidra considers files under the root read-only. We recommend ending this database with the extension .fidb for consistency, although not strictly necessary. Newly created databases are attached (which means "known" for the purposes of tracking) and initially active, even though they contain no match entries.

#### **[Attach existing FidDb...](#)**

This brings up a file chooser dialog that allows the user to attach an existing Function ID database. This cannot be located under the Ghidra install directory root, because Ghidra considers files under the root read-only. Attached databases are saved in the user preference system, and retain their active status across sessions of Ghidra.

#### **[Detach attached FidDb...](#)**

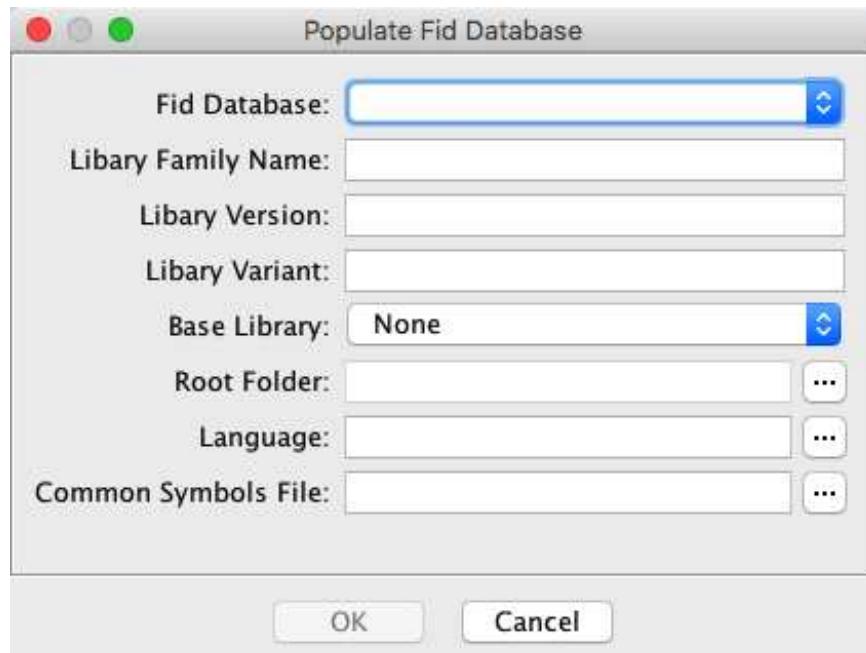
This brings up a dialog that allows the user to detach an already-attached Function ID database. None of the databases delivered with Ghidra can be detached; they can only be deactivated (see "[Choose active FidDbs...](#)"). Detaching a database removes it from use in

searching, and also causes the user preference system to forget about the existence of this database.



### [Populate FidDb from programs...](#)

Once a database has been created (or attached), the user may populate it with hash values from a set of programs in the current project. Choosing this option brings up a dialog where users enter the information needed to populate the database.



### *Dialog Fields*

#### **Fid Database**

Pick the database to populate. Users must select from attached databases that are writable.

#### **Library Family Name**

The name of the library being ingested. This is the identifier that will be inserted as part of the comment when a match is found for functions in this library.

#### **Library Version**

The formal version string for the library. This is frequently the <major>. <minor>. <patch> syntax but can be anything provided to

distinguish between different versions of the same library.

#### **Library Variant**

Any other string for distinguishing libraries that doesn't fit in the formal *Version* string. This frequently holds compiler settings.

#### **Base Library**

This is an optional setting for cross-linking with an existing library already ingested in the database. In the event that the user wants to incorporate parent/child relationships into this library from another related library, they can set this option to point at the other library. The library must already be ingested (from a previous use of this command) into the same database. The Function ID ingest process will match parents and children across the libraries via function symbols.

#### **Root Folder**

This specifies the set of programs from which the new library will be populated. The user can select any subfolder in the current Ghidra project. The ingest process will recursively search through all programs beneath this folder and, for each program, will collect any functions it contains.

#### **Language**

This is the required 4-field Ghidra Language ID (i.e. *x86:LE:64:default*) specifying exactly what processor the new language will contain. While scanning during ingest, any program that does not match this program model will be automatically skipped.

#### **Common Symbol File**

This is an optional parameter that provides a list of common function symbols to the ingest process. The parameter, if present, is a path to a text file that contains the list of function symbols, one per line. The ingest process excludes these functions from consideration as disambiguating child functions. (See [“False Positives”](#))

Once all fields have been filled, clicking **OK** causes all selected functions to be ingested for the new library. Depending on the number of programs and functions selected, this process may take some time. Upon completion, the process will present a summary window, containing ingest statistics and an ordered list of functions that were most commonly called within the library. This list can be used to create a *Common Symbol File* tailored for the library.

## Preparing Libraries for a Function ID Database

### Location of Programs

All functions going into a single Function ID *Library* must already be imported and analyzed somewhere within a single Ghidra repository (shared or non-shared). Multiple libraries contained within the same database can be ingested in different phases, but a single library must be written to the database in a single pass. The ingest dialog (see [“Populate FidDb from programs...”](#)) specifies a single subfolder as the root for the library. The process acts recursively, so there can be additional directory hierarchy under the root, but all programs to be included in the library must be under the one root.

## Analysis

All programs must be analyzed enough to have recovered the bodies of all the functions that are to be included in the library. Generally, the easiest way to accomplish this is to run Ghidra's default auto-analysis. If functions are spread across multiple programs, as is typically the case, users can run the Ghidra's `analyzeHeadless` command to analyze across the whole set. However, take note below of some of the modifications to the default analysis that may be necessary to get good Function ID results.

Every function to be included must have a *non-default* function name assigned. Function ID uses a function's *primary symbol* for the name. Symbols are typically imported from debug information, but any method for assigning names, script based or manual, will work. Any function that still has its default name, `FUN_00...`, currently will not be ingested.

When performing auto-analysis in preparation for ingest, it's best to disable the **Function ID** analyzer (and the **Library Identification** analyzer as well) in order to avoid cross contamination from different databases. If the function symbols are *mangled*, be sure to turn off the **Demangler** analyzer. This lets the future database apply the raw mangled symbol to new programs during analysis, which lets their **Demangler** analyzer pass run with complete information.

For an example of these sorts of modifications to the analysis process, see the file:  
`Features/FunctionID/ghidra_scripts/FunctionIDHeadlessPrescrip.java`

This is designed to be passed to the `analyzeHeadless` command as a pre-script option.

## False Positives

A **false positive** in the context of Function ID is a function that is declared as a match by the analyzer but has the incorrect symbol applied. As with any classification algorithm, it is generally not possible to eliminate this kind of error completely, but with Function ID there are some mitigation strategies.

### Causes

False positives for the most part only happen with small functions. There are two related causes with Function ID:

#### Tiny Functions

If a function consists of only a few instructions, it can be matched *randomly* if there are enough entries in the database. The fewer operations a function performs, the more likely an unrelated function is to do those exact same things.

#### Code Idioms

A set of functions that are effectively identical can have different names and be used in unrelated contexts. Functions such as *destructors* are typical: one might check that a particular structure field is non-zero, and then pass that field to `free`. Another destructor may perform the exact same sequence, but was designed for a completely

unrelated structure.

In either case, Function ID can apply a symbol that is misleading for the analyst.

### Mitigation via Threshold

All mitigation strategies, to some extent, trade-off false positives for **false negatives**, which are functions that should have been reported by the Function ID analyzer, but aren't (because of some threshold or strategy).

Most false positives by far are due to tiny functions. Function ID minimizes these via the *instruction count threshold*. Potentially matching functions with too few instructions that don't exceed this threshold will simply not be reported by the analyzer.

For users experiencing too many false positives, the instruction count threshold is the easiest thing to adjust. It is fully controllable by the user as an Analysis option (See [“Analysis Options”](#)), and increasing it will directly reduce the false positive rate, at the expense of missing some *true* matches whose scores now fall below the threshold.

### Specialized Mitigation

The default instruction count threshold is a good starting point for any new database, generally striking a reasonable balance limiting false positives without eliminating too many true matches. But even for an optimal threshold, there may be a small handful of functions in the new database (usually *Code Idioms*) that exceed the threshold and repeatedly cause the wrong label to be placed. Instead of increasing the threshold to filter out *all* functions with these higher scores, it is possible to turn on one of several mitigation strategies that target the offending database entries directly. These strategies include:

#### **Force Specific**

If this is set on an entry, the specific hash must match before the system will consider the entry as a potential match. This is useful when a code idiom contains a known constant that the full hash would usually miss.

#### **Force Relation**

This is probably the most useful specialized strategy. It forces at least one parent or child match to be present before the system considers the base function as a potential match. So even if an idiom is big, this forces a search for an additional confirmation.

#### **Auto Fail**

This is a strategy of last resort. If an obnoxious code idiom can't be eliminated any other way, this forces the particular database entry to never be considered as a match.

#### **Auto Pass**

This strategy is different from the others, in that it applies to function entries whose scores are slightly too *low*. If a low scoring function has an instruction sequence that is deemed unique enough, this strategy causes any potential match to automatically pass the threshold. This provides an alternative to lowering the instruction count threshold to

include a particular function.

These strategies can all be toggled for *individual function records* in the database. To do this manually from the Code Browser, the user needs to search for the specific records they want to change using the *Debug Search Window* and then make changes from its *Result Window*. For details see [“Debug Search Window”](#).

Strategies can also be toggled by running a Ghidra script. Within a script the basic instruction sequence looks like:

```
FidFileManager fidFileManager = FidFileManager.getInstance();
List<FidFile> allKnownFidFiles = fidFileManager.getFidFiles();
// Choose a modifiable database from the list
...
// Open a specific database
FidDB modifiableFidDB = fidFile.getFidDB(true);

// Toggle strategies based on the full hash of the function(s)
modifiableFidDB.setAutoFailByFullHash(0x84d01243dfb8b9cbL,true);
modifiableFidDB.setForceRelationByFullHash(0x4e0920960b48ae7eL,true);
modifiableFidDB.setForceSpecificByFullHash(0x5ef2f47ee7151243L,true);
modifiableFidDB.setAutoPassByFullHash(0x96a4a6fd5694523bL,true);

...
// Save and close the database
modifiableFidDB.saveDatabase("comment",monitor);
modifiableFidDB.close();
```

---

FunctionID hashes for specific functions can be obtained with the `FIDHashCurrentFunction` script.

[Prev](#)

Function ID

[Next](#)

Function ID Debug Plug-in

## Function ID Debug Plug-in

[Prev](#)

---

# Function ID Debug Plug-in

The Function ID Debug Plug-in allows users to inspect the individual records in a Function ID database. This functionality is generally only useful to users building their own databases. Users can look up individual records based on name or hash, but currently the plug-in is not capable of modifying records. The only exceptions are the “[Specialized Mitigation](#)” strategies, which can be modified using the “[Debug Search Window](#)”.

## Enabling the Plug-in

The Debug Plug-in adds options to the **Function ID** menu, under the Code Browser’s main **Tools** menu. These options are in addition to those introduced by the [Function ID Plug-in](#), which uses the same menu. In order to access the Debug options, the plug-in must be enabled. To do this, from the Code Browser select

**File -> Configure**

Then click on *Configure* link under the **Experimental** section and check the box next to "FidDebugPlugin".

## Plug-in Functionality

The Function ID Debug Plug-in introduces the following actions to the **Tools → Function ID** menu.

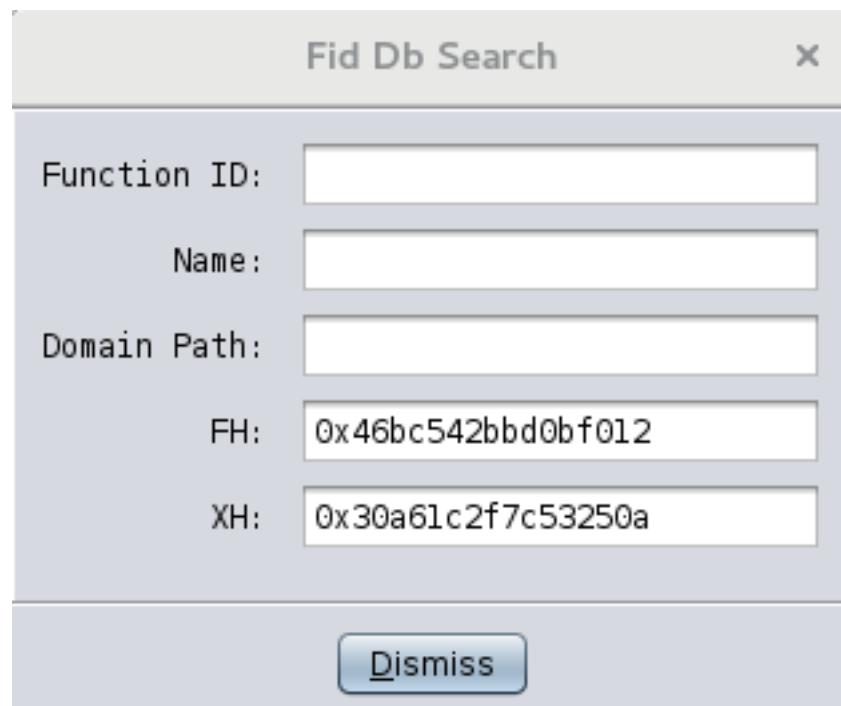
[Create Read-only Database](#)

Users can convert the read/write (.fidb) database into the a read-only (.fidbf) form. This is the more efficient final form used directly by the Function ID analyzer. The .fidbf form is uncompressed on disk and the analyzer can use it directly, where the .fidb form must be converted before use.

## Debug Search Window

This action brings up a Search dialog for the currently active Function ID databases. The text entry fields correspond to the individual fields in a *function record* that can be searched. If the variant **Debug Search Window (Current Function)** is invoked, the same dialog is brought up, but the *Full Hash* and the *Specific Hash* fields are pre-populated with hashes corresponding to the function at the current address.

A search is initiated for a specific search field by entering a value and then hitting the **RETURN** key, with the cursor and focus still in the desired field.



## Search Fields

**Function ID**

This is the internal row ID for function records in the database.

**Name**

This searches through function names. Searches here will match any record whose name contains the search string.

**Domain Path**

This searches through domain paths. A **domain path** is the file path, relative to the project root, of a program containing the function described by a particular record. Searches here will match any record whose domain path contains the search string.

**FH**

This searches for records matching the *full hash*. The text field expects a 64-bit value. The hash can be entered as a hexadecimal string by prepending with "0x".

**XH**

This searches for records matching the *specific hash*. The text field expects a 64-bit value. The hash can be entered as a hexadecimal string by prepending with "0x".

***Result Window***

Invoking a search will bring up the *Result Window*, presenting a row for each matching function record. Columns list properties of the function and correspond to the search fields described above. In addition to these, each record/row lists a few other columns:

**Library**

This is the library containing the function

**Code Unit Size**

This is the number of (scoring) instructions in

the function's body. See “[Scoring and Disambiguation](#)”.

### **Spec. + Size**

This is the number of distinct constant operands fed into the specific hash.

### **Warn**

This lists any special properties that have been toggled for the particular record. The column is presented as a string of single character codes, corresponding to each possible property. Properties include the mitigation strategies described in “[Specialized Mitigation](#)”.

- F – Auto Pass: The record automatically fails.
- P – Auto Fail: The record automatically passes.
- S – Force Specific: Specific hash must match for record to be considered.
- R – Force Relation: A parent or child must match for record to be considered.
- U – The function body was unterminated (analysis error).

### **Edit Menu**

The *Result Window* supports a small number of actions under the **Edit** menu that allow the user to change the mitigation strategies for a read/write database. Strategies are described in “[Specialized Mitigation](#)”. When a menu action is invoked, all records displayed in the current Result Window are affected. Currently there is no way to select a subset of records to effect. The actions all set or clear a specific strategy.

#### **Set/Clear auto-fail**

Toggle the *Auto Fail* strategy.

#### **Set/Clear auto-pass**

Toggle the *Auto Pass* strategy.

#### **Set/Clear force-specific**

Toggle the *Force Specific* strategy.

**Set/Clear force-relation**

Toggle the *Force Relation* strategy.

**Save changes**

Changes made using the **Edit** menu are not immediately saved back to the underlying database until this action is invoked.

## Table Viewer

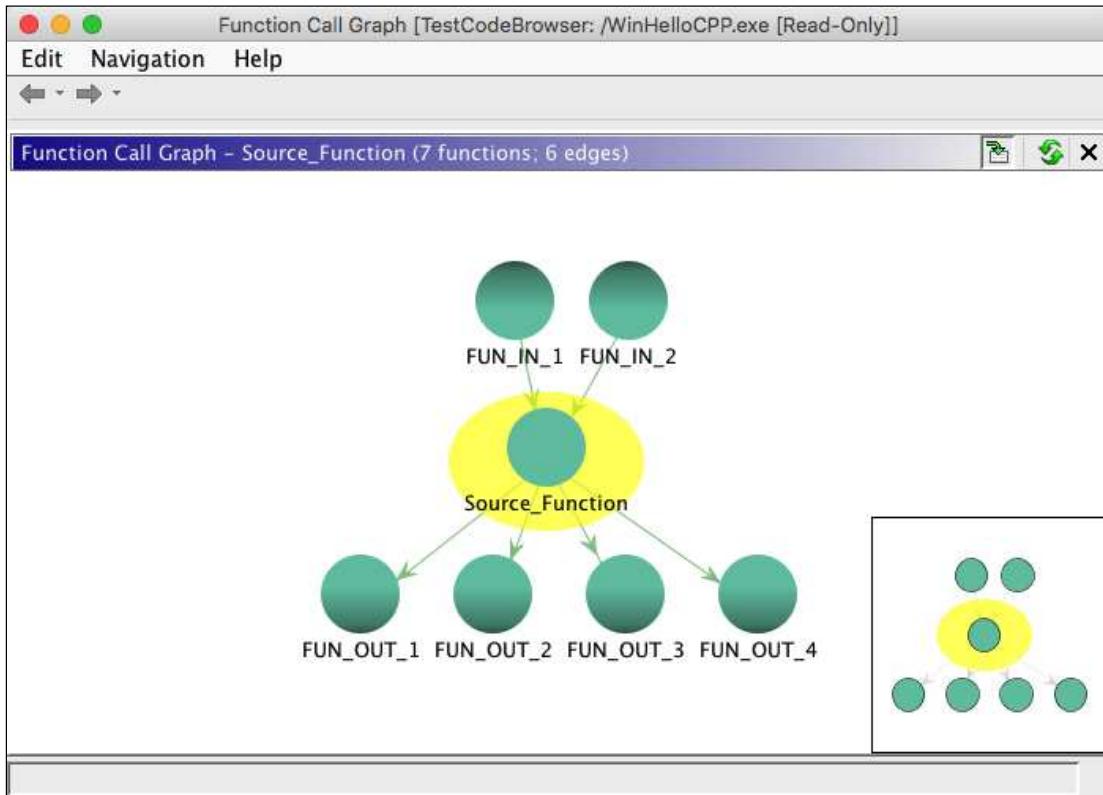
This invokes an extremely low-level view into the underlying tables that back a Function ID database. It can be invoked on any attached database. A window is brought up that lists individual records in one table. A drop-down menu allows the user to switch between the different underlying tables. Most of the columns contain row keys and don't present readable values. The only meaningful table is likely to be the *Libraries Table* which will list each library making up the database.

---

[Prev](#)

Function ID Plug-in

# Function Call Graph Plugin



The Function Call Graph Plugin is a simple graph display that shows incoming and outgoing calls (as edges) for the function containing the current address, also known as the *Source Function*, in the Listing. This display provides some context for how a function is used within the program. The functions are organized by [Level](#).

To show the Function Call Graph provider window, select the **Window ➔ Function Call Graph** option on the tool menu.

The graph of function calls related to the source function being displayed can be explored by [adding existing function calls](#) to the initial graph display.



The graph updates itself as you navigate within the tool. To prevent losing graph state (e.g., expanded functions, node locations, etc), a small number of graphs will be cached. For example, if you navigate away from a function and then immediately return, the graph will be restored to its previous state.

## Terms

- **Source Function:** the function that contains the current address in the Listing. This function is considered the center of the graph, with all other callers/callees added to the graph at a new level.

- **Level:** Each function node in the graph belongs to a level. The source function is at level 1; the source function's incoming calls are at level 2; the source function's outgoing calls are also at level 2. Organizing functions by level allows the user to quickly see how many hops, or calls, a given function is from the source function.

New levels of calls can be added to the graph by the user.

- **Direction:** Each function node, other than the source function, is considered to be in one of two directions: In or Out. All function nodes in a given level share the same direction. So, all nodes

that directly call the source function node are considered to be the *In* direction; all nodes directly called by the source function are considered to be the *Out* direction.

When a given node's level is expanded in the graph, the nodes added are based upon the selected node's direction: for *In* nodes, the newly added nodes will be those nodes that **call** the selected node; for *Out* nodes, the newly added nodes will be those nodes **called by** the selected node.

- **Direct Edges:** An edge (a call) between two adjacent levels.
- **Indirect Edges:** An edge (a call) between two non-adjacent levels or an edge within the same level. These edges are rendered with less emphasis than *direct edges*.

## Actions

### Show/Hide Edges Action

Within the *Function Call Graph* you can show and hide function calls as desired. Showing additional function calls can be accomplished multiple ways. From any function node, you can select the *Expand* icon () , which appears on a node when hovered. When clicked, this button will toggle related function calls: showing them if not already in the graph; hiding them if they are in the graph.

Additionally, these same functionality is provided from the popup menu actions (i.e., **Show/Hide Incoming Edges** and **Show/Hide Outgoing Edges**).



As new vertices are added to the graph, any indirect edges will be added to the graph.

Note here how new vertices may appear in odd places when expanding (such as when they are already in the graph at a previous level).



It is important to understand that the graph is only a subset of the entire program graph. This graph does not represent all functions and function calls in the program.



Sometimes a function may have too many references to display in the graph. When this happens, the function node will be a gray color, with the expand icon replaced with a warning icon, as so:



### Show/Hide Level Edges Action

All functions that relate to the Level of the selected function will be shown, **not just calls to the selected function**.

### Navigate on Incoming Location Changes

This action () , when toggled on, upon receiving Program Location changes from the tool, will graph the function containing that location. When toggled off, location changes will not affect the graph.

Having this action on is useful if you wish to quickly see the graph of different functions as you navigate the program. Alternatively, having this action off is useful when you wish to explore the

program by navigating from within the graph, say by double-clicking function nodes in the graph.

#### Layout Action

This action ( ) will relayout the current graph and reset the graph to show only the initial nodes.

#### Graph '*Function Name*'

This action is available from the popup menu of any node that is not the currently graphed node. When pressed, this action will graph the clicked function.

### Satellite View

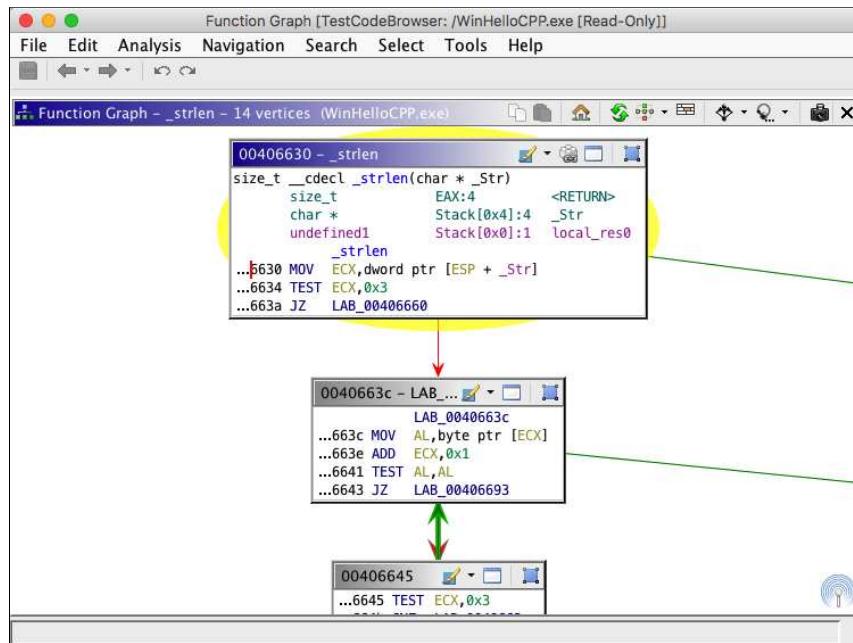
The Satellite View works exactly as the [Function Graph's Satellite View](#).

Provided by: *Function Call Graph Plugin*

Related Topics:

- [Graphs](#)

## Function Graph Plugin



The Function Graph Plugin is a simple graph display that shows the code blocks of the function containing the cursor in the [Listing](#).

The display consists of the [Primary View](#) and the [Satellite View](#). There is also a group of [actions](#) that apply to the entire graph.



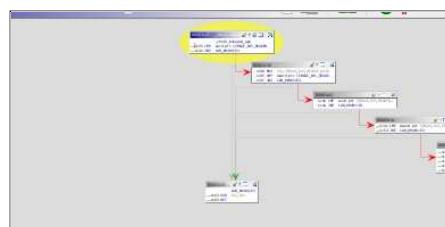
### What's New

- Added the Nested Code Layout. This layout shows the code blocks in a structure that more closely resembles the flow of C code.
- Small tweaks (6.0)
  - Detachable Satellite View
  - Function signature now included by default
  - Added a [regroup action](#) to allow for regrouping vertices after ungrouping.
  - Articulated edges will now disappear if the connected vertices are dragged outside of the original angles.
  - Moved [someactions](#) to the context popup menu, specifically:
    - Edit Label
    - Fullscreen Mode
    - Jump to XRef
- [Layout Compressing \(5.6\)](#)
- [Keyboard zooming and panning \(5.6\)](#)
- [Vertex Grouping \(5.5\)](#)

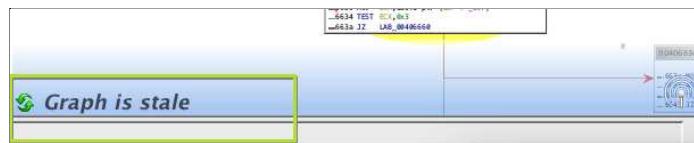
### Primary View

The Primary View displays the [Vertices \(or Blocks\)](#) and [Edges \(or Control Flow\)](#) of the graph. From this view you can interact with blocks, both for editing and arrangement.

The graph rendered in the Primary View may represent an undefined function, such as a subroutine. If this is the case, then the background of the Primary View will be a gray color, such in the following image:



By default, as changes to the program are detected, the graph will **not** relayout to account for these changes. The image below shows the bottom of the Primary View when the graph has detected changes and is considered **stale**.



Once a graph is stale, you can press the refresh button at any time to have the graph re-create itself **without performing a relayout**. The green box in the image above contains the refresh button. Alternatively, you can press the [relayout action](#) to refresh the stale graph **and** perform a relayout, which we reposition the vertices of the graph to their preferred locations.

If you would like to change the default behavior on program changes to perform a full graph update, then you can change this value via the tool options. You can access these options by right-clicking in graph and selecting the **Properties** action. Alternatively, you can click on the tool's menu bar **Edit->Tool Options...** to launch the options. From there you can find the option at **Function Graph-> Automatic Graph Relayout**

## Satellite View

The Satellite View provides an overview of the graph. From this view you may also perform basic adjustment of the overall graph location. In addition to the complete graph, the satellite view contains a **lens** (the white rectangle) that indicates how much of the current graph fits into the primary view.

When you single left mouse click in the satellite view the graph is centered around the corresponding point in the primary view. Alternatively, you may drag the lens of the satellite view to the desired location by performing a mouse drag operation on the lens.

You may hide the satellite view by right-clicking anywhere in the Primary View and deselecting the **Display Satellite View** toggle button from the popup menu.



If the Primary View is painting sluggishly, then hiding the Satellite View cause the Primary View to be more responsive.

## Detached Satellite

The Satellite View is attached, or **docked**, to the Primary View by default. However, you can detach, or undock, the Satellite View, which will put the view into a Component Provider, which itself can be moved, resized and docked anywhere in the Tool you wish.

To undock the Satellite View, right-click in the graph and deselect the **Dock Satellite View** menu item.

To re-dock the Satellite View, right-click in the graph and select the **Dock Satellite View** menu item.



To reshown the Satellite View if it is hidden, whether docked or undocked, you can press the button. This button is in the lower-right hand corner of the graph and is only visible if the Satellite View is hidden or undocked.

## Vertices (Blocks)

Each vertex in the displayed graph represents a code block within the graphed function. The term **block** is used synonymously with the term **vertex**. The block display consists of a header and a code listing. The fields contained in the listing are a subset of the available fields. You may change the fields displayed from the [Edit Code Block Fields](#) action.

The header contains the name of the block, as defined by the label at that location, or that address if no label exists. The header also contains buttons that allow you to perform some common operations on the block.

As long as you are within the [interaction threshold](#), you may interact with the block's listing just as you would with Ghidra's primary [Listing](#).

The following actions are available from the primary view.

### Selecting Blocks

Left-clicking a block will select that block. To select multiple blocks, hold down the **Ctrl** key (or the equivalent for your OS) while clicking. To deselect a block, hold the **Ctrl** key while clicking the block. To clear all selected blocks, click in an empty area of the primary view. When selected, a block is adorned with a halo.

You may also select multiple blocks in one action by holding the **Ctrl** key while performing a drag operation. Press the **Ctrl** key and start the drag in an empty area of the primary view (not over a code block). This will create a bounding rectangle on the screen that will select any blocks contained therein when the action is finished.

### Navigating Blocks

If you double-click a block header, then the [Zoom Level](#) of the Primary View will change. If the block is not at full zoom (1:1), then the zoom level will be changed to full zoom. Otherwise, the zoom level will be changed to fully zoomed out. If you are zoomed past the [interaction threshold](#), then you can double-click anywhere in the block to trigger a full zoom.

Assuming you are **not** zoomed past the interaction threshold, then double-clicking a field inside the block's listing will perform a navigation [as determined by that listing](#). If you **are** zoomed past the interaction threshold, then double-clicking anywhere in the block will trigger navigation in the same way as double-clicking the block header.

### Block Information

You can hover over a block to get descriptive information. Depending upon the [Zoom Level](#) of the primary view, you will get different hovers. When zoomed past the interaction threshold, the hover action will trigger a popup window showing a preview of the block. At full zoom, you will only receive popup windows [as determined by the listing](#) inside of the block. You may disable [popups](#) as desired.

### Vertex Actions



The button allows you to set the background color for the vertex. You may press the button to choose the color currently displayed in the icon, or you may use the drop-down menu to pick a previously used color. Additionally, from the drop-down menu you can clear the color or choose a new color to set.



By default, colors applied to a vertex are also applied to the primary disassembly Listing.

The button will **group** all selected vertices.

#### Popup Menu Vertex Actions

The button allows you to set the label for the code block (this will also change the block header).

The toggle button allows you to quickly view the contents of the block in a full window view, which uses the same format as Ghidra's primary [Listing](#). To restore the graph view from the full window view, click this action again, which will then have this icon: .

The button will show a table of xrefs to the entry point of the currently graphed function.



This action will also appear in the vertex containing the function entry point, for convenience.

#### Grouped Vertex Actions



This section describes vertex grouping, which is covered [later in this document](#).

The button allows you to set the background color for the vertex. You may press the button to choose the color currently displayed in the icon, or you may use the drop-down menu to pick a previously used color. Additionally, from the drop-down menu you can clear the color or choose a new color to set.

#### Group Vertex Coloring Algorithm

This group color feature allows you to easily color large numbers of vertices after you have grouped them and to keep already set user-defined colors as you are grouping vertices.

The Function Graph will automatically color your group vertex, depending on the color state of the vertices being grouped:

- If none, or some but not all, of the vertices being grouped have a user-defined color, then the group vertex will be made the default color (which you can change from the options).
- If all of the vertices being grouped have a user-defined color, but that color is not the same, then the group vertex will be made the default color.
- If all of the vertices being grouped have **the same** user-defined color, then the new group vertex **will be made the color of the vertices**.

When a group vertex has a user-defined color, then all vertices grouped therein will take on that color.



*Via the options you can disable this feature.*

The button allows you to set the text displayed in the group vertex. Unlike the action when used in a non-grouped vertex, this action will not edit the label at the start address of the vertex.

The button will **ungroup** the given vertex.

The button will **group** all selected vertices.

The button will add to the given group vertex all other selected vertices.

The difference between **group** and **add to group** is somewhat subtle. The **group** action creates a *new* group vertex with each selected vertex as a child, contained inside of the new grouped vertex. Alternatively, the **add to group** action adds to the *existing* group node chosen all other selected vertices.

The regroup button is included in the header of any **uncollapsed** vertex, which is a vertex that is the member of a group, where that group has been **ungrouped**. This action will **regroup** (or collapse) all vertices in the same group as the vertex containing the action. To regroup is to convert all members of a given group back into a single grouped vertex.



To remove an uncollapsed vertex from group membership, right-click on that vertex and select **Ungroup Selected Vertices**.

## Edges

The Edges of the vertices represent a flow from one code block to another. One end of each edge has an arrowhead that represents the direction of the flow. Furthermore, the color of the edge provides a visual indication as to the type of the flow. The default flow colors are:

	Fallthrough –the negative case of a conditional check
	Conditional –the positive case of a conditional check
	Unconditional –An unconditional flow

The following actions are available from the primary view.

### Selecting Edges

You may select an edge by left-clicking it. To select multiple edges, hold down the **Ctrl** (or the equivalent for your OS) while clicking. To deselect an edge, hold the **Ctrl** key while clicking the edge. To clear all selected edges, click in an empty area of the primary view.

### Navigating Edges

Double-clicking an edge will navigate to the one of the incident blocks. The navigation will first take you to the destination block if it is not already selected. Otherwise, the navigation will take you to the source block.

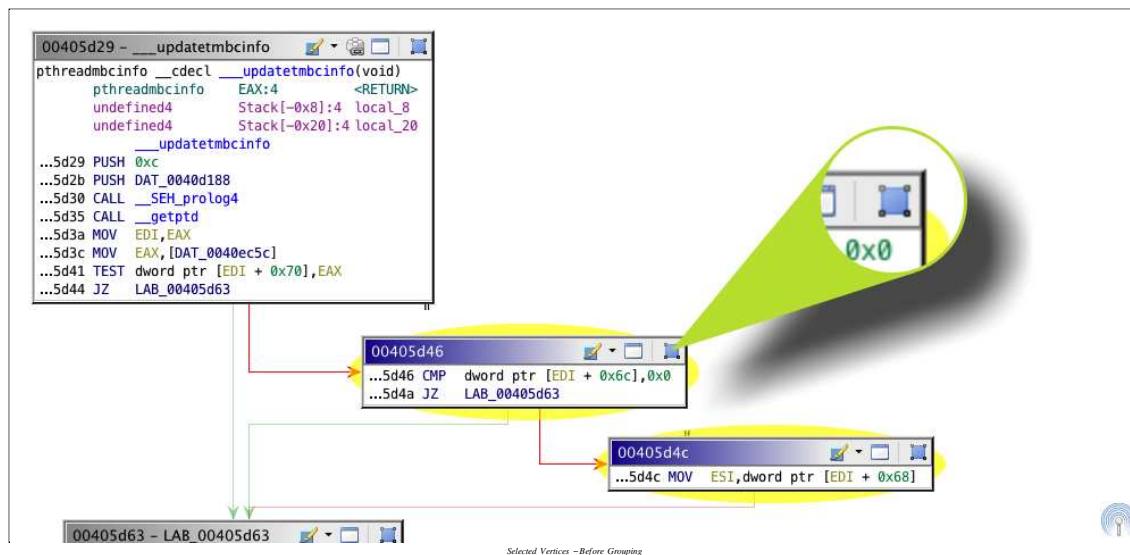
### Edge Information

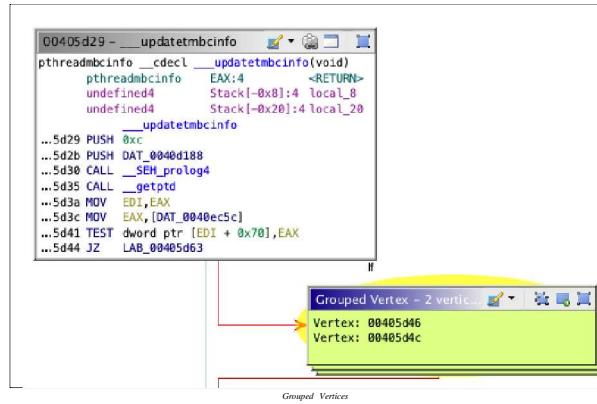
You can hover over an edge to get descriptive information. When you hover over an edge you will be presented with a popup window showing a preview for the source and destination blocks for the hovered edge. You may disable [popups](#) as desired.

### Articulated Edges

Some graph layouts create articulated edges, which are edges that contain bends in them to route around vertices. As you drag vertices around the graph the bends in the articulations in the dragged edge may disappear if the articulation causes the edge to contain awkward angles.

## Vertex Grouping





You may select 2 or more vertices to be turned into a single grouped vertex. This allows you to organize vertices to reduce the amount of information displayed in the graph. As an example, you may wish to place all branches of a switch statement into a single grouped vertex.

You can select a single vertex to group. This allows you to annotate a given vertex with text, without editing the label at the vertex address, which is the default behavior of the [edit\\_label\\_action](#). In addition to setting the text for the grouped vertex, it will remove the disassembly. In this regard, grouping a single vertex is a form of information hiding.

Before a group vertex is created you are prompted to enter text that will be displayed in the body of the group vertex. By default, the titles of each grouped vertex will be listed as the text of the grouped vertex.



The default contents of the group vertex text entry dialog are generated from the titles of each vertex being grouped. When grouping a vertex which is itself a group, the text of that group vertex will be used in the text entry dialog in addition to its title.

Grouped vertices may contain other grouped vertices.

As you group vertices, the graph may perform a relayout of the vertices, depending upon the Function Graph Options, as described [below](#).

#### The Ungrouping Process

Ungrouping a grouped vertex will restore to the graph all vertices contained in the grouped vertex. The layout behavior of the graph after performing an ungroup operation is dependent upon the graph options; specifically, the **Automatic Graph Relayout** option. The default setting for automatic relayout is **Vertex Grouping Changes Only**. This means that as you group and ungroup vertices, the graph will relayout its vertices, which may be a drastic layout change. To prevent the graph from performing a relayout during grouping or ungrouping, set the option listed above to be **Block Mode Changes Only** or **Never**.

You can access the Function Graph Options by right-clicking in an empty area of the graph and clicking the **Properties** menu item.

You can ungroup all group vertices in the graph via the right-click popup menu by selecting [Ungroup All Vertices](#). Warning: this will ungroup all groups, which is an operation that cannot be undone.

Ungrouped vertices can be regrouped by executing the [regroup action](#). This action is executed from an individual vertex, but will apply to all vertices in its group.

#### Graph Actions

The following actions are buttons in the Function Graph Plugin header.



The button will perform a copy action in one or more vertex listings. See the [Clipboard](#) help for more information on using copy in the Listing and Listing-based views.

The button will perform a paste action in one or more vertex listings. See the [Clipboard](#) help for more information on using paste in the Listing and Listing-based views.

The button will navigate to and select the entry point block.

The button clear all **position and grouping** changes made to the graph and then perform a reload and relayout of the graph.

The button allows you to both change the layout used to arrange the graph and to perform a relayout of the graph using the current layout. Simply pressing the button will trigger a relayout, whereas clicking on the drop-down arrow will allow you to choose a new layout.



*This action allows you to perform a graph relayout without losing grouping information*

The button allows you to change the fields of the blocks' listing.

By default, the format configuration of the vertices is greatly condensed. This is done to fit as many vertices on the screen as is possible. You can make the vertices larger or smaller as you see fit. For more information about adding and removing fields, as well as adjusting the size of the fields in the vertex listing display, see the [Listing Panel's format help](#).

The button will create a [Snapshot](#) of the current graph.

### Path Highlight Actions

The focus and hover path highlighting modes are designed to help show the flow of execution through the code blocks in a function, as well as illustrate some of the structure. Hover highlights are triggered when you move the mouse over a block. Focus highlights are triggered by selecting a block and only work from one selected block, not with multiple selected blocks.

The **focus highlight** paints the edges between certain code blocks with a bold stroke, thicker than the regular edges. The **hover highlight** paints a dashed, thicker stroke that also moves in the direction of flow for a limited period of time.

The path highlighting modes (described in the table below) are available for both focus and hover, except in special cases, as noted.

Icon	Name	Description
	Show Scoped Flow From Block	Highlights control flow to code blocks that are only reachable if the current code block is executed. This is useful to see a local neighborhood of blocks that follow the current block.
	Show Scoped Flow To Block	Highlights control flow from code blocks that must eventually reach the current code block. This is useful to see a local neighborhood of blocks that precede the current block.
	Show Paths To/From Block	Highlights control flow from code blocks that can reach the current code block, as well as control flow to code blocks that can be reached from the current block. This is useful to show all possible flows before and after the current block.
	Show Paths From Block	Highlights control flow to code blocks that can be reached from the current code block. This is useful to show all possible flows after the current block.
	Show Paths To Block	Highlights control flow from code blocks that can reach the current code block. This is useful to show all possible flows before the current block.
	Show Loops Containing Block	Highlights the control flow between all possible looped blocks (cycles) that pass through the current block. If a function has multiple non-intersecting loops, this helps resolve the loops from each other in the case that the graph layout has placed them too close to differentiate.
	Show Paths From Focus to Hover (hover mode only)	Highlights the control flow from the currently focused code block to the currently hovered code block. If there are no paths possible, no edges will be highlighted. This is useful to see reachability between two sections of the function.
	Show All Loops In Function (focus mode only)	Highlights the control flow between all possible looped blocks (cycles) in the current function. This mode doesn't actually depend on a focused code block; instead, selecting it highlights all loops immediately.

### Function Graph Options

The **Automatic Graph Relayout** option describes when the graph will perform an automatic relayout of the vertices as the graph changes. The available values are:

- **Always** –always perform a graph relayout anytime the code blocks change or when graph groups change
- **Block Mode Changes Only** –only performs a relayout when the code blocks of the graph change (e.g., from an external edit)
- **Vertex Grouping Changes Only** –only performs a relayout when the state of the graph groups changes (during a group or ungroup operation)
- **Never** –never perform a relayout of the graph automatically

The **Scroll Wheel Pans** option signals to move the graph vertical when scrolling the mouse scroll wheel. Disabling this option restores the original function graph scroll wheel behavior of zooming when scrolled.

The **Start Fully Zoomed Out** option causes the initial graph to zoom out far enough that the entire graph is displayed. When this option is off a new graph rendering

will zoom all the way in (no scaling) to the active vertex.

The **Update Vertex Colors When Grouping** option signals to the graph to make the color of the grouped vertex be that of the vertices being grouped.

The **Use Animation** option signals to the graph whether to animate mutative graph operations and navigations.

The **Use Condensed Layout** option signals to the graph to bring vertices as close together as possible when laying out the graph. Using this option to fit as many vertices on the screen as possible. Disable this option to make the overall layout of the graph more aesthetic.

The **Use Full-size Tooltip** When toggled off, the tooltip for a vertex will be the same size and layout of the vertices in the graph. When toggled on, the tooltip for a vertex will be larger, using the layout of the Listing. The larger size is more informative, but also takes up more space.

The **Use Mouse-relative Zoom** option signals zoom the graph to and from the mouse location when zooming from the middle-mouse. The default for this option is off, which triggers zoom to work from the center of the graph, regardless of the mouse location.

There are various edge color and highlight color options available to change. The highlight colors are those to be used when the flow animations take place.

## Creating Program Selections

### From Paths

You may create Program Selections from the **current path highlights** by clicking **Program Selection ➔ From Hovered Edges** and **From Focused Edges** from the popup menu of a block. If no paths are highlighted, then these actions will be disabled.

### From Code Blocks

You may select all Code Units in a Code Block by clicking **Program Selection ➔ Select All Code Units** from the popup menu (or by using the default keybinding, **Ctrl-A**). This action will select all Code Units in all selected Code Blocks in the graph. **If no Code Blocks are selected, then a Program Selection will be created for all Code Units in all Code Blocks in the graph.**

### Clearing Selections

You may clear the current Program Selection by clicking **Program Selection ➔ Clear Selection** from the popup menu.

## Popups

The primary view provides various popup windows to provide information as you hover over the blocks and edges in the graph. To enable and disable popups in the primary view, right-click anywhere in the primary view and select the **Display Popup Windows** toggle button from the popup menu.

## Grouping

The following popup menu items provide additional [grouping](#) functionality.

- **Group Selected Vertices** –Groups all selected vertices
- **Group Selected Vertices –Add to Group** –Adds the selected vertices to group vertex in the selection. This action will not be enabled if there is not one, and only one, group vertex in the selection.
- **Remove From Group** –Removes the [uncollapsed vertex](#) from its group.
- **Ungroup All Vertices** –Ungroups all vertices in the graph, not just those selected or visible. This operation cannot be undone!
- **Ungroup Selected Vertices** –Ungroups the selected group vertices

## Panning

There are various ways to move the graph. To move the graph in any direction you can drag from the whitespace of the graph.

By default, to move the graph vertically you can use the mouse wheel. In previous releases the scroll wheel was used to zoom. Now there is an option to restore that behavior, the **Scroll Wheel Pans** option. When this option is on, you can zoom by holding the **Control**key (**Command**key on the Mac) while using the scroll wheel. Alternatively, you can move the graph left to right using the mouse while holding **Control-Alt**.

The satellite viewer may also be used to move the primary graphs view by dragging and clicking inside of the satellite viewer.

## Zooming

At **full zoom**, or **block level zoom**, each block is rendered at its natural size, which is the same scale as Ghidra's primary [Listing](#). From that point, which is a 1:1 zoom level, you can zoom out in order to fit more of the graph into the display.

To change the zoom you may use the mouse scroll wheel while holding the **Control**key (**Command**key on the Mac). This works whether the mouse is over the primary viewer or the satellite viewer. Also, you may use the context popup menu from the primary viewer in order to quickly zoom to the block level (1:1) or to the window level (zoomed out far enough to fit the entire graph in the window). These actions are **Zoom to Vertex** and **Zoom to Window**, respectively.



To have the scroll wheel zoom without holding the **Control**key, you can disable the **Scroll Wheel Pans** option.



To zoom the graph incrementally using the keyboard you can use the **Zoom In** and **Zoom Out** actions. These actions have default keybindings of **Control-Minus** and **Control-Equals**.

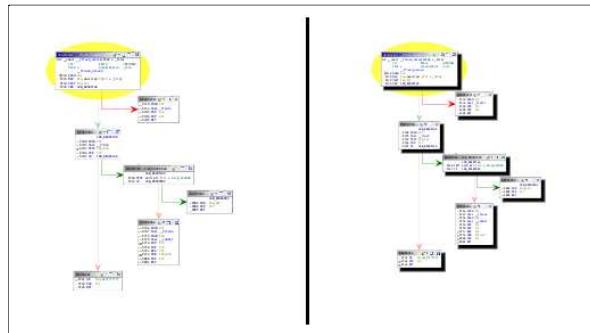
The [satellite viewer](#) is always zoomed out far enough to fit the entire graph into its window.

### Vertex Quick Zoom

If you double-click a block header, then the [Zoom Level](#) of the Primary View will change. If the block is not at full zoom (1:1), then the zoom level will be changed to full zoom. Otherwise, the zoom level will be changed to fully zoomed out. If you are zoomed past the [interaction threshold](#), then you can double-click anywhere in the block to trigger a full zoom.

### Interaction Threshold

While zooming out (away from the blocks) you will eventually reach a point where you can no longer interact with the listing inside of the block. The blocks provide a subtle visual indication when they are zoomed past this level, in the form of a drop-shadow. The image below shows this drop-shadow. The block on the left is not past the interaction threshold, but the block on the right is, and thus has a drop-shadow. This example is for illustrative purposes only and during normal usage all blocks will share the same zoom level. So, if one block is zoomed past the interaction threshold, all other blocks will be as well.



Interaction with blocks that are past the interaction threshold is simplified; for example, when scaled past the interaction threshold, dragging in the listing area of a block will drag the block, instead of making a selection in the listing, as would happen when not scaled past the interaction threshold.

### Painting Threshold

While zooming out (away from the blocks) you will eventually reach a point where contents each block will not be painted. Instead, each block will be painted by a rectangle that is painted with the current background color of the block.



Zooming past the painting threshold will improve the rendering speed of the Primary View.

### Saving View Information

The *Function Graph Plugin* will automatically save your changes to the graph, specifically, coloring nodes, grouping nodes, zooming and panning. This happens as you change the function displayed in the graph and when you close the graph window.



Changes made to [Snapshots](#) will not be saved. This is done to avoid conflict between changes made to the connected view and any of the snapshots

Provided by: *Function Graph Plugin*

#### Related Topics:

- [Code Browser](#)
- [Snapshots](#)

# Function Graph Layouts

## Nested Code Layout

The nested code layout uses the Decomplier to arrange the code blocks of a function in a way that mimics the nesting of the source code as seen in the decompiled function. As an example, any code block that must pass through an `if` statement will be nested below and to the right of the code block that contains the conditional check. The nested code block is **dominated** by the block containing the conditional check—code flow can only reach the nested block by passing through the block above it. Also, code blocks that represent a default code flow will be aligned to the left and below other code blocks in the function. This layout allows the user to quickly see the dominating relationships between code blocks.

The edges leaving a code block are labeled with the type of high-level conditional statement (e.g., `if`, `if/else`, etc) used to determine code flow.

By default, edges are routed such that they are grouped together such that any edges returning to a shared code block will overlap. This reduces visual clutter at the expense of being able to visually follow individual edges to their vertices. Another consequence of this routing is that sometimes edges will travel behind unrelated vertices, again, making it difficult to visually follow these edges. The

edge routing can be changed via the options below.

### Nested Code Layout Options

The **Route Edges Around Vertices** option triggers this layout to route edges around any vertex that would otherwise touch that edge. (See above for notes on how edges are routed for this layout.)

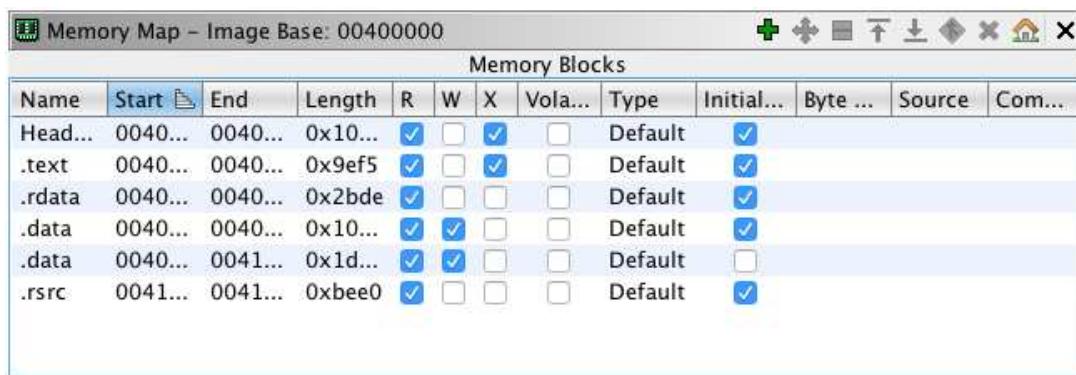
# Memory Map

The *Memory Map* window displays a list of memory blocks that make up the memory structure of the current program. The component provides actions for adding, renaming, moving, splitting, extending, joining, and deleting memory blocks.

Ghidra supports four different block types through the Memory Map window:

1. **Default** – The normal block type that can be *initialized* or *uninitialized*.
  - *Initialized* – The block has an initial value specified for the bytes
  - *Uninitialized* – The block has no initial value specified for the bytes
2. **Bit Mapped** – The block provides a bit-addressable map onto other blocks. This is useful when a processor can access some or all of the bits in memory directly using an alternative addressing space.
3. **Byte Mapped** – The block provides a byte-addressable map onto other blocks. This can be useful when the same bytes can be accessed via two or more addresses.
4. **Overlay** – The block is created in a new *overlay* address space. Overlay blocks can be *initialized* or *uninitialized*. Using Overlays is a way to get around the problem where the program is too large to fit completely in the target system's memory. Overlay blocks contain code that would get swapped in when the program needs to execute it. Note that Overlay blocks are fixed and may not be moved, split or expanded. In addition, Overlays do not relocate with image base changes.

To view the *Memory Map*, select **Window** → **Memory Map** from the main tool menu, or click on the  icon in the tool bar. Note that the current Image Base Address is specified within the title bar.



The screenshot shows the 'Memory Map' window with the title 'Memory Map - Image Base: 00400000'. The window contains a table with the following data:

Memory Blocks												
Name	Start	End	Length	R	W	X	Vola...	Type	Initial...	Byte ...	Source	Com...
Head...	0040...	0040...	0x10...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>			
.text	0040...	0040...	0x9ef5	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>			
.rdata	0040...	0040...	0x2bde	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>			
.data	0040...	0040...	0x10...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>			
.data	0040...	0041...	0x1d...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input type="checkbox"/>			
.rsrc	0041...	0041...	0xbbe0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>			

Each row displays information about one of the memory blocks. The following summarizes the information about each block.

**Name** – Name of the memory block.

**Start** – The starting address (in hex) of the memory block.

**End** – The ending address (in hex) of the memory block.

**Length** – The length (in hex) of the memory block.

**R** – Indicates read permission.

**W** – Indicates write permission.

**X** – Indicates execute permission.

**Volatile** – Indicates a region of volatile I/O Memory.

**Type** –Indicates whether the block is a [Default](#), [Bit Mapped](#), [Byte Mapped](#) or [Overlay](#) type of block.

**Initialized** –Indicates whether the block has been initialized with values; this property applies to Default and Overlay blocks.

**Byte Source** –Provides information about the source of the bytes in this block. If the bytes were originally imported from a file, then this will indicate which file and the offset into that file. If the bytes are mapped to another region of memory, it will provide the address for the mapping. Blocks may consist of regions that have different sources. In that case, source information about the first several regions will be displayed.

**Source** –The name of the file that produced the bytes that make up this block as set by the file importer; for [Bit Mapped](#) or [Byte Mapped](#) blocks, the *Source* shows the mapped source address.

**Comment** –User added comment about this memory block.

## Memory Block Edits

### Rename

Memory Blocks can be renamed by double-clicking on the name field and entering a new name.

### Change Read Permission

The read permission of a memory block can be changed by left-clicking on the checkbox.

### Change Write Permission

The write permission of a memory block can be changed by left-clicking on the checkbox.

### Change Execute Permission

The execute permission of a memory block can be changed by left-clicking on the checkbox.

### Change Volatile Setting

The volatile setting of a memory block can be changed by left-clicking on the checkbox.

### Initialize Memory Block

A memory block that is currently Uninitialized can be Initialized by clicking on the *Initialized* checkbox. A dialog will appear allowing you to enter a byte value to be used to fill the block. Deselecting the checkbox will cause the block be revert to uninitialized memory. This will have the additional side effect of removing all functions, instructions, data, and outgoing references in that block.

### Edit Comment

Memory Block comments can be changed by double-clicking on the comment field and entering the new comment. The comment can be a maximum of 256 characters.

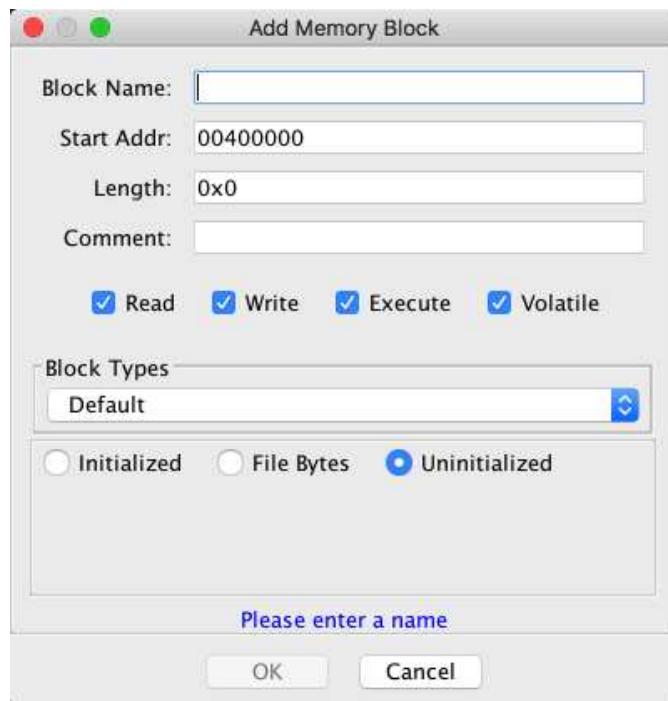
## Memory Block Operations

The memory block operations are available through the icons on the header of the Memory Map window, or select a block in the table, right mouse click, and choose an option.



**Add**

Select **Add** to bring up the *Add Memory Block* dialog. Fill in the requested information and select the **OK** button.



**Block Name** – Enter the name of the new memory block.

**Start Addr** – Enter the start address of the new memory block. If the program language defines multiple address spaces, the address space must also be specified. The address space selection will not appear if only one is defined. Overlay spaces are not included in the list of spaces. Within the default address space, a block may not span across the current Image Base Address.

**Length** – Enter the length of the new memory block.

**Comment** – Enter a comment for the block if desired.

**Read** – Sets the read permission.

**Write** – Sets the write permission.

**Execute** – Sets the execute permission.

**Volatile** – Marks this block as volatile I/O memory.

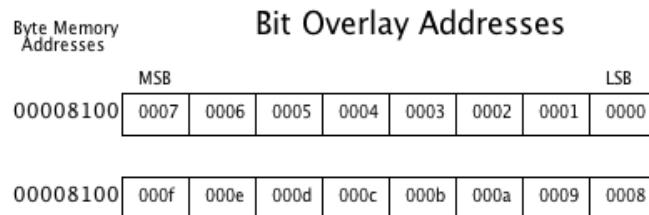
**Block Types** –Select the block type from the combo box: **Default**, **Bit Mapped**, **Byte Mapped**, or **Overlay**.

- **Default** –A normal memory block within the processor's address space. These blocks cannot overlap any other default block. Default blocks can be one of the following types:
  - **Initialized** –Specify a value and a new block will be created using that value for every byte in the block.
  - **Uninitialized** –An uninitialized block will be created.
  - **File Bytes** –Select from a list of imported files and enter a starting offset for that file. Those bytes will be the initial value for the block.

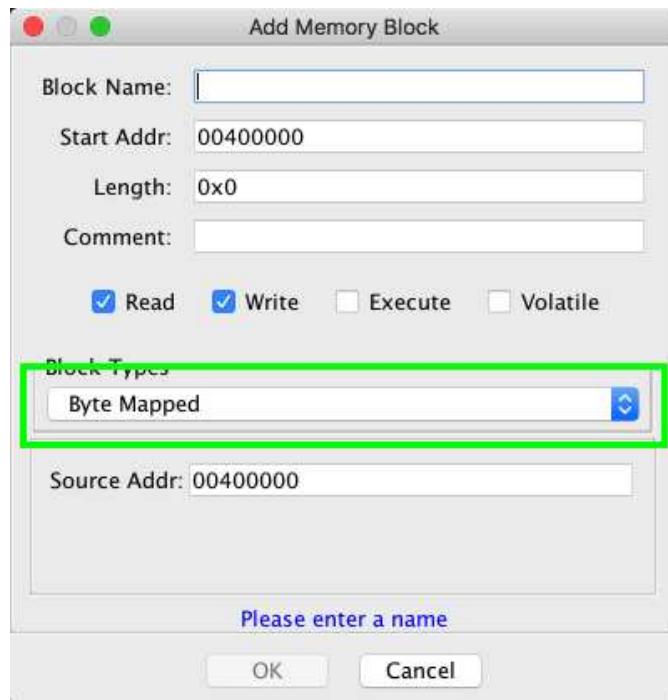


You can use the "Add To Program" using "Binary Import" to create new FileBytes that you can use here.

- **Overlay** –An overlay block is used to give an alternative set of bytes (and related information) for a range in memory. This is achieved by creating a new address space related to the actual processor address space and placing the block in the new space at the same offsets as the start address in the processor space. Overlay blocks can be either initialized or uninitialized. If you select *Initialized* you can enter a byte value that will be used to fill all the bytes in the new memory block.
- **Bit Mapped** –This is a block that allow bit addressing of a section of bytes in memory. For example, the first bit of the byte at memory location 0x1000 might also be addressed as BIT:0. The second bit at the same byte would then be addressed as BIT:1 and so on.
- The illustration below depicts a Bit Mapped block of *Length* 16 with a *Start Addr* of (BIT:) 0000, and a *Source Address* of 00008100. Note that Bit Overlay addresses are assigned from least significant bit to most significant bit.

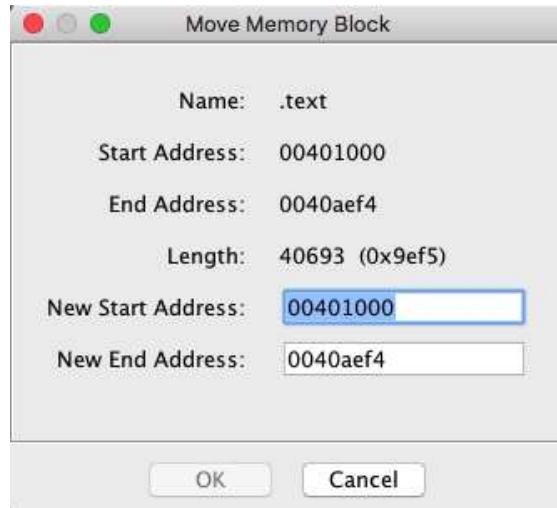


- This is used to model certain processors that allow this sort of addressing such as the INTEL 8051. When a Bit Mapped block is created you must specify the byte address on which the bit addressing will be based.
- **Byte Mapped** –This is a block that allows access to a range of bytes in memory using an alternative address. In other words, it allows the same set of bytes to be accessed by two different logical addresses. A source address must be specified that contains the actual bytes for this block.



### Move

Select **Move** to bring up the *Move Memory Block* dialog. The *Move* action is enabled when exactly one memory block is selected. Enter either a new start or end address to cause the block to be moved.



**Name** –Name of the memory block to be moved (not editable).

**Start Address** –Current starting address of the block to be moved (not editable).

**End Address** –Current ending address of the block to be moved (not editable).

**Length** – Length of the memory block to be moved (not editable).

**New Start Address** – Enter the NEW starting address for the block. The NEW ending address will be computed.

**New End Address** – Enter the NEW ending address for the block. The NEW starting address will be computed.



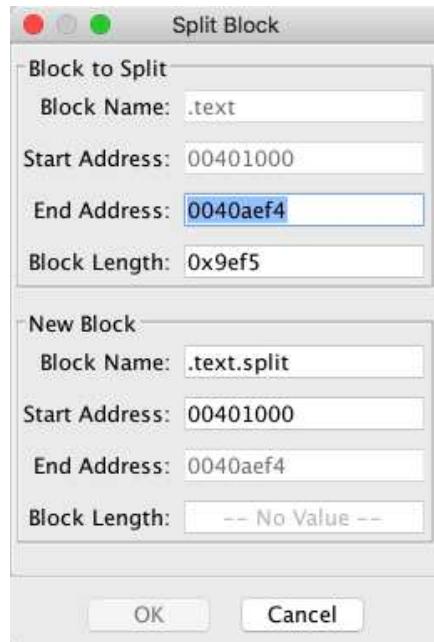
You cannot move a block under the following conditions:

- The changes specified would result in an overlap with the original block or any other existing block.
- The block is an [Overlay](#) block.

## ■ Split

Select **Split** to bring up the *Split Block* Dialog. The *Split* action is enabled when exactly one memory block is selected. Use the *Split Block* Dialog to split a memory block into two smaller memory blocks. There are four ways to enter the split point:

- Enter an end address for the first block (block to split), or
- Enter a length for the first block (block to split), or
- Enter a start address for the second block (new block), or
- Enter a length for the second block (new block).



### Block to Split

**Block Name** – Name of block being split (not editable)

**Start Address** – Start address of block being split (not editable)

**End Address** –New end address of the original block

**Block Length** –New length of original block

#### New Block

**Block Name** –Name of new block. Default name will be provided, but it can be changed by editing this field

**Start Address** –Start address for the new split block

**End Address** –End address of the original block (not editable)

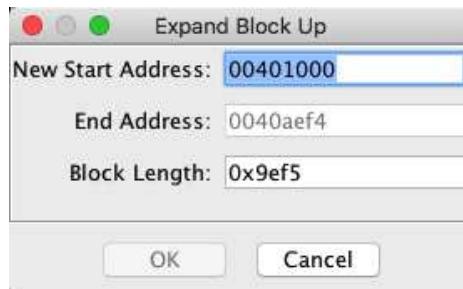
**Block Length** –Length of new split block



*Overlay type blocks cannot be split.*

#### ↑ Expand Up

Select **Expand Up** to bring up the *Expand Block Up* Dialog. The *Expand Up* action is enabled when exactly one memory block is selected. Use the *Expand Block Up* Dialog to cause a memory block to grow by adding additional bytes BEFORE the memory block. The block can be expanded by either entering a new start address or a new length.



**New Start Address** –A new start address can be entered here. It must be before the current start address.

**End Address** –Displays the end address of the block (not editable).

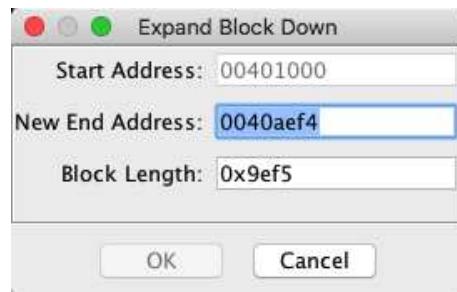
**Block Length** –Displays the length of the block. A new value can be entered here which will cause a corresponding change in the start address.



*Overlay type blocks cannot be expanded.*

#### ↓ Expand Down

Select **Expand Down** to bring up the *Expand Block Down* Dialog. The *Expand Down* action is enabled when exactly one memory block is selected. Use the *Expand Block Down* Dialog to cause a memory block to grow by adding additional bytes AFTER the memory block. The block can be expanded by either entering a new end address or a new length.



**Start Address** – Displays the start address of the block (not editable).

**New End Address** – A new end address can be entered here. It must be after the current end address.

**Block Length** – Displays the length of the block. A new value can be entered here which will cause a corresponding change in the end address.



*Overlay type blocks cannot be expanded.*

### Merge

The *Merge* action is enabled when two or more memory blocks are selected. It attempts to merge all selected blocks into one block. Any "gaps" will be "filled in" with 0s.



*Caution should be used because a VERY large memory block can be created if the memory blocks being merged are far apart in the address space. Ghidra will display a warning if a really large block is about to be created. Disregarding the warning may cause Ghidra to fail with an "out of memory" error.*



Only blocks of the same type can be merged. For example, [default](#) blocks can only be merged with another default block.

### Delete

The *Delete* action is enabled when one or more memory blocks are selected. All selected blocks will be deleted. If the blocks contained defined data or instructions, a confirmation dialog is displayed; select "yes" on the dialog to delete the block. A progress dialog is displayed while the block is being deleted.



For large blocks that may contain many symbols, references, instructions, etc., the delete operation may take a while to complete. You can cancel the delete operation at any time.

### Set Image Base

The *Set Image Base* action allows you to change the base address of a program. This action is useful when working with relocatable code such as DLLs or shared objects. All addresses, code units, references, etc. will immediately be re-based relative to the new base address.

For example, given a program base at 0x01000000 with a memory block starting at address 0x01001000. If the base address was changed to 0xeeee0000, then the new start address of the

memory block would become 0xeeeee1000.

To change the image base, enter a new image base in the text field and click on the *OK* button.



Provided by: *Memory Map* Plugin

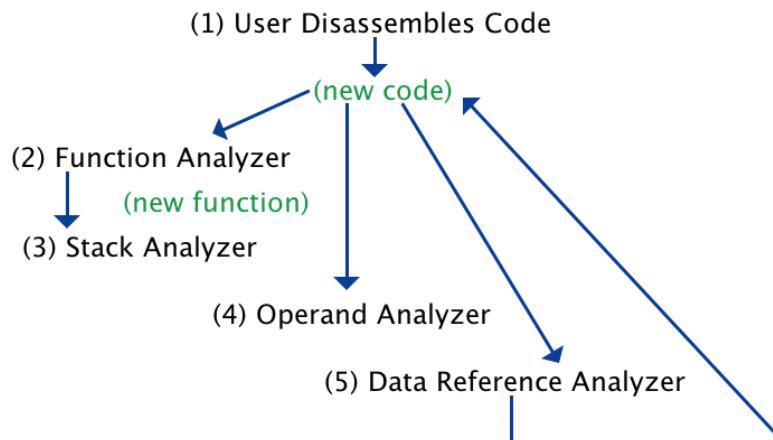
# Program Annotation

A variety of automated and manual actions are provided within the tool to annotate (add information to) the program file. Auto Analysis provides an automated way to disassemble bytes into code, create functions along with references to the stack variables, and create references. You can also manually define functions and references or disassemble code. In addition, you can manually define data in the file, set register values, add labels and comments to a code unit, define a name (equate) to substitute for a scalar value, or bookmark an address.

- [Auto Analysis](#)
- [Bookmarks](#)
- [Comments](#)
- [Data](#)
- [Disassembly](#)
- [Equates](#)
- [Functions](#)
- [Labels](#)
- [References](#)
- [Register Context](#)

## Auto Analysis

Auto Analysis watches for changes to the program, such as [disassembling](#) a new area of memory or definition of a function. When a change is noticed, it kicks off Auto Analyzer plugins interested in the change. Auto Analyzer plugins evaluate the changes and may decide to make further changes to the program. Any changes an Analyzer makes to the program may cause additional Analyzers to run. An example chain of auto analysis might be:



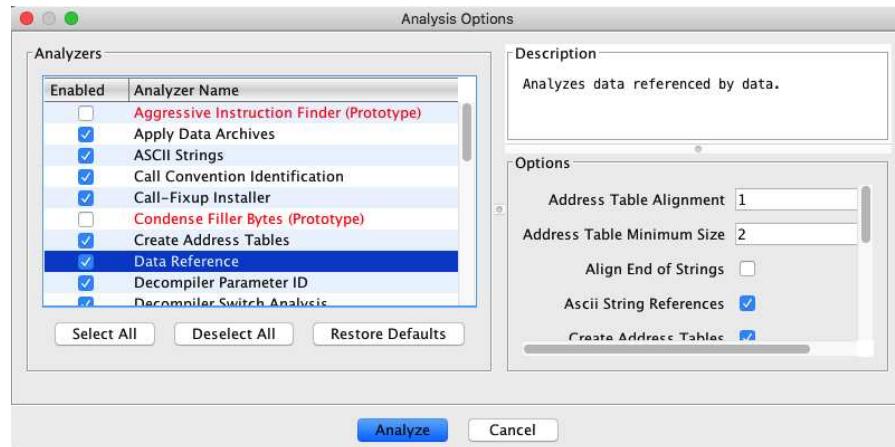
1. User Disassembles
2. Function Analyzer –looks at all calls and creates Functions
3. Stack Analyzer –looks at all new functions and builds a stack based on stack references
4. Operand Analyzer –looks at scalar operands for possible address references
5. Data Reference Analyzer –looks at references for possible strings or pointers to code.  
References to code are disassembled.  
..... Cycle repeats with 2) as additional code is disassembled.

One program change might cause several Analyzers to become active, however only one Analyzer is run at a time. Each Analyzer has a run priority. For instance, the Operand Analyzer will always run before the Data Reference Analyzer because the operand analyzer could create new references that the Data Reference Analyzer would need to analyze.

Auto Analyzers normally change the program only if the Analyzer can be certain the change is correct. For instance creating a function at the destination of a call is a fairly safe bet. Randomly looking for undefined data that could be disassembled where there are no actual code references would not be a good idea.

### Auto Analysis

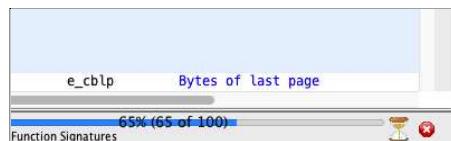
A program imported through Front End will have no initial analysis applied to it. To force analysis, use the **Analysis → Auto Analysis** menu item. The Auto Analysis Options dialog is displayed to allow changes to the analysis options before beginning analysis.



The display of this dialog may be controlled from the Auto Analysis Tool Options. Select the **Auto Analysis** node in the **Edit → Tool Options** menu and check or uncheck **Show Analysis Options**. When unchecked, the next time Auto Analysis is chosen, the options dialog is skipped, and analysis begins immediately.

Without a selection in the Code Browser, the entire program's memory space is analyzed. To restrict the analysis to certain areas of the program, select the area in the code browser before choosing **Auto Analysis**. If there is a current selection, **Auto Analysis** will be restricted to only to those areas of the program within the selection. Areas outside the selection may be changed by analysis. For example, if the OperandReferenceAnalyzer finds a reference within the selected area, a string or a function could be created at the referenced location.

When the **Analyze** button is pressed, the Options dialog will disappear. At the bottom of the Code Browser window, the background task bar will display. Any Analyzer activity status messages are within the task bar.



To Cancel the analysis, press the button.

#### Analyze All Open

This action will auto-analyze all open programs. The [options](#) will be displayed only one time and re-used for each program.

When you choose options to use for analyzing a program, they are based upon the architecture of the **current** program. When analyzing all open programs, any programs besides the current program will only be analyzed if their architecture (language ID and compiler spec ID) match that of the current program. Any other open programs with an architecture differing from the current program will **not** be analyzed using this action.

If the option to show the analysis dialog is disabled, then using this action **will** analyze all open programs **using their current analysis settings**. To disable the analysis dialog go to **Edit → Tool Options...** to show the options dialog upon which, select the **Auto Analysis** options and de-select **Show Analysis Options**.

#### One Shot Analysis

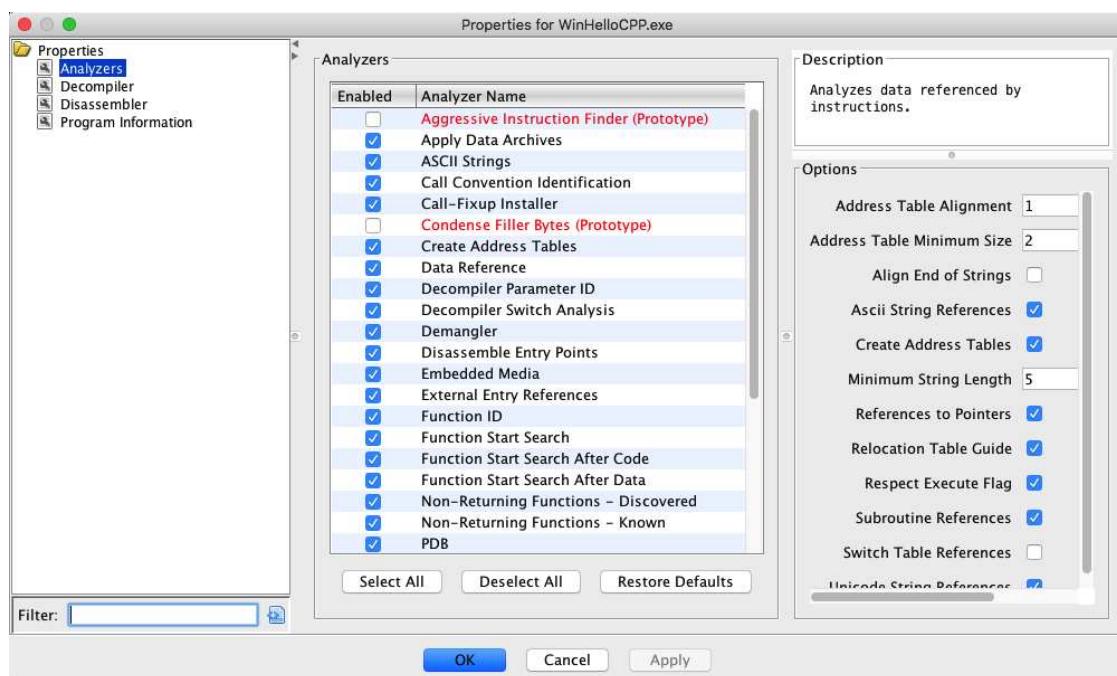
Those specific analyzers which support on-demand analysis are available as One Shot Analyzers in the Analysis menu. Only those analyzers which support one-shot use and are applicable to the current program will be available within the One Shot sub-menu.

#### Auto Analysis Options Panel

Each Analyzer can be turned off using the *Analysis* panel in the **Edit → ProgramOptions** dialog.

The *Analysis* panel configures Analyzer specific options and which Analyzers are enabled to run. To view these options, choose **Edit → Program Options...**; then open the **Analysis** node in the Options tree. Each analyzer is represented as a leaf under the **Analysis** folder node.

An analyzer may be marked as (Prototype). These analyzers can be very useful on certain programs, but have not been exercised on a large number of programs. Use them with caution.



A separate dialog showing the *Analysis* options is also displayed after a new program has been imported, or the **Analysis → Auto Analysis** menu item is chosen.

Some analyzers only work on certain processors. For example, the MIPS Instruction Analyzer only works on MIPS processors. Their options will only show up if the currently open program can be analyzed with the analyzer.

#### Auto Analyzers

The following is a description of the Auto Analyzers that have been implemented. Please note that this list may not be complete since additional Analyzers are continuously being added to Ghidra and can be supplied by add-on modules/contribs.

**Address Table Analyzer**

Looks at all undefined data locations to find possible [address tables](#).

You can also find and create address tables manually via the [Search for Address Tables](#)

Started By: Importing or adding to a program, Auto Analyze command

**ARM Analyzer**

This analyzer works only against the ARM processor. It looks at multiple instructions to discover references that are put together by multiple instructions. Since the ARM is a RISC processor and each instruction is only 32 bits wide, many references must be created by multiple instructions.

Started By: New disassembled code

**ASCII Strings Analyzer**

This analyzer searches for valid ASCII strings and automatically creates them in the binary. Candidate strings are found using the same method (and most of the same associated options) as [Search for Strings](#). Of the candidate strings found, valid strings are identified using a model trained on pre-identified valid strings. This analyzer runs at a very low priority.

Set options for this analyzer by selecting **ASCII Strings** on the options panel. Some options such as **Require null termination for string** and **String start alignment** work the same way as in [Search for Strings](#).

**Options**

- **Create strings containing existing strings** –if checked, strings will be created even if they contain existing substrings (existing strings will be cleared). The string will be created only if existing strings (a) are wholly contained within the potential string, (b) do not share the same starting address as the potential string, (c) share the same ending address as the potential string, and (d) are the same datatype as the potential string to be created).
- **Create strings containing references** –if checked, strings that contain, but do not start with, one or more references will be created.
- **Force model reload** –if checked, forces the model to be reloaded every time the analyzer is run (in cases where the user wishes to see the effect of changing a model without restarting Ghidra).
- **Minimum string length** –specifies the smallest number of characters in a string for it to be considered a valid string. For this analyzer, null termination characters are ignored for the purposes of counting characters. Note that smaller numbers will result in a larger number of false positives. String length must be at least 4.
- **Model file** –Specifies the model file built using the BuildStringModels class (default is 'StringModel.sng'). Note that the location of the model file does not need to be specified, as models should always be placed in the 'Ghidra/Features/Base/data/stringograms' directory.
- **Require null termination for string** –if checked, only null-terminated strings are created.
- **Search only in accessible memory blocks** –if checked, searches only in memory blocks that have at least one of the Read (R), Write (W), or Execute (X) permissions set to true. Enabling this option ensures strings are not created in areas such as overlays or debug sections.
- **String end alignment** –specifies the byte alignment requirement for the end of the string. An alignment of 1 means the string can end at any address. Alignments greater than 1 require that (a) the 'require null termination' option be enabled, and (b) if the null-terminated string does not end at an aligned boundary, that there exist enough trailing '0' bytes following the string to allow alignment. If neither (a) nor (b) apply, end alignment is not enforced.
- **String start alignment** –specifies the byte alignment requirement for the start of the string. An alignment of 1 means that strings can start at any address. An alignment of 2 means that strings must start on an even address. An alignment of 4 means that strings must start on an address that is a multiple of 4.

Started By: Auto Analyze command

**Data Archive Analyzer**

This analyzer looks at all the labels defined in the program and applies function signatures from standard data type libraries. For programs identified as "Microsoft Windows" executables, the data archive applied were parsed from the standard windows header files. All other executables will have function signatures applied from the standard "C" header files (stdio, fcntl, ...). For example, if a label exists for strcmp upon import, the analyzer will assume this is the standard C-library strcmp and apply a function signature of "int strcmp(char \*, char \*)".

Started By: Importing or adding to a program, Auto Analyze command

**Data Reference Analyzer**

Looks at all data references within newly disassembled code for Unicode/Ascii strings and functions. When a valid function is found, the code is disassembled starting at the referenced location. When a valid string is found, a string data type is created.

Enable this option by selecting **Ascii String References**, **Unicode String References**, or **Subroutine References** on the options panel for **Instructions**.

Started By: New disassembled code

**Decompiler Parameter ID Analyzer**

For each function created, run the decompiler and import the information recovered about the given function. The information includes:

- passed parameters
- local variables defined on the stack
- local variables defined in registers
- return value
- the prototype or calling convention (stdcall, cdecl, thiscall, fastcall, ...)
- switch tables recovered by data flow analysis

Applying type information recovered by the decompiler can be extremely useful if you have type information for library functions. You can apply function signatures to your library functions, and as code is disassembled, type information will be propagated from the library functions up into the parameters and local variables of the functions calling them.

Switch tables recovered by the decompiler can be applied to improve the disassembly of code. At times the basic switch table analysis cannot recover complex jump tables. The decompiler can also recover the value used to "switch" on for each case in the switch table. The label created at each switch case is created based on the recovered switch value.

This analyzer is being enhanced to pull more information gleaned from running the decompiler.

Enable this option by selecting **Decompiler Parameter ID** on the options panel for **Functions**.

Started By: Creating a function, or Re-Analyzing a program with functions already defined.

#### Demangler Analyzer

This analyzer examines the name of the newly created function. If the name appears to be a *GCC v3* or *Microsoft Visual Studio* mangled symbol, then it will demangle the name and create a new primary symbol for the demangled name. It will also assign the appropriate datatypes to the parameters and return type.

Started By: New defined functions

#### Entry Point Analyzer

Disassembles code starting at all Symbols in the symbol table marked as *External Entry Points*.

Enable this option by selecting **Disassemble Entry Points** on the options panel for **Byte**.

Started By: Importing or adding to a program, Auto Analyze command

#### Image Analyzer

This analyzer searches the program for images. If a valid image is found an appropriate image data type is applied at that location with the corresponding visual representation of the image. Also, a bookmark is added to indicate an image.

Started By: Importing or adding to a program, Auto Analyze command

#### MIPS address markup Analyzer

This analyzer works only against the MIPS R4000 processor. It looks at multiple instructions to discover references that are put together by multiple instructions. Since the R4000 is a RISC processor and each instruction is only 32 bits wide, many references must be created by multiple instructions.

It will also look for certain types of switch or jump tables and automatically create a jump table.

Started By: New disassembled code

#### PowerPC address markup Analyzer

This analyzer works only against the PowerPC processor. It looks at multiple instructions to discover references that are put together by multiple instructions. Since the PowerPC is a RISC processor and each instruction is only 32 bits wide, many references must be created by multiple instructions.

It will also look for "bccr" type switch statements and automatically create a switch table including references.

Started By: New disassembled code

#### Propagate External Parameters Analyzer

This analyzer uses external Windows function call parameter information to populate comments next to pushed parameters. In some cases, data is labeled and commented as well

Started By: Auto Analyze command

#### Scalar Operand Analyzer

Looks for scalar operands that are actually address references at each instruction operand within newly disassembled code.

Enable this option by selecting **Scalar Operand References** on the options panel for **Instructions**.

Started By: New disassembled code

#### Stack Analyzer

Creates a stack frame (parameters and local variables) based on references to the stack found in newly defined functions.

Enable this option by selecting **Stack References** on the options panel for **Function**.

Started By: New defined functions

#### Subroutine Reference Analyzer

Creates a function at each destination of a call instruction. If the destination would create a complex function, the function is not created. A complex function is one with multiple entry points, or shared code with function.

Enable this option by selecting **Create Functions** on the options panel for **Function**.

Started By: New disassembled code

#### Windows x86 PE Analyzers

The Windows x86 PE Analyzers analyze a Windows PE program for MS Visual Studio data structures and code. They currently attempt to identify RTTI structures, virtual function tables, and exception handling code. One also analyzes the external functions and attempts to propagate the data types associated with the parameters.

These analyzers only run on PE programs.

Started By: Analyzing the program or creating instructions or defined data with a Windows x86 PE Analyzer analysis option selected.

### (Prototype) Auto Analysis Plugins

The following is a description of the prototype Auto Analyzers that have been implemented.

#### [Aggressive Instruction Finder](#)

This analyzer runs at the lowest priority after there are no other analyzers needing to run. It looks at undefined bytes to see if code were disassembled at the start of an area it would form a valid subroutine. There can be no invalid instructions, and the subroutine must not "flow" into any existing code.

Needless to say this analyzer will not work well on every program, but on some it finds all code with no mistakes. Your mileage may vary. There's always undo...

Started By: Importing or adding to a program, Auto Analyze command

#### [Condense Filler Bytes Analyzer](#)

This analyzer searches the program for all specified filler bytes and collapses them. Some examples of filler bytes are: 0, 00, 90, cc. If you do not specify a certain byte pattern to search for then the default will be used. The default is the word *Auto* and it will allow the program to determine the best value to use based on the greatest count. You also have the option to change the minimum number of bytes to collapse. The default for this is 1.

Started By: Auto Analyze command

Provided by: *AutoAnalysisPlugin*

Related Topics:

- [Disassembly](#)
- [Functions](#)

# Disassembly

Disassembly is the process of translating bytes into assembly language instructions. Ghidra supports three different disassembly actions:

- [Disassembly](#)
- [DisassemblerOptions](#)
- [Static Disassembly](#)
- [RestrictedDisassembly](#)

## Disassembly

*Disassembly* starts disassembling at the current cursor location. After disassembling an instruction, it examines all possible execution flows from that instruction and disassembles at all those locations.

When disassembling in a selection, the disassembly starts at the first undefined byte within the selection. After exhausting all possible flows from the instruction, disassembly begins again at the next undefined byte within the selection.

To perform *Disassembly*

1. Place your cursor at the target instruction
2. Right-mouse-click
3. Select Disassembly

The disassembly progress is displayed in the progress status area (bottom right) of the tool. The progress status area displays the number of instructions disassembled and provides a "Cancel" button. Press the "Cancel" button to stop the disassembly process. Note: this will not undo any disassembly that has already occurred.



*Press the 'D' key to Disassemble starting at the current cursor location.*



*Selecting [Undo](#) will return the disassembled instructions to*

*undefinedbytes.*



*Be aware that some processors may allow for disassembly in different modes (eg: ARM vs. THUMB). Specific commands may be provided for these and are not discussed here.*

## Disassembler Options

The following *Disassembler* options exist which control certain behaviors during *Disassembly*

- **Mark Bad Disassembly** –places an error bookmark on any instruction or instruction flow which fails to disassemble due to parse error or instruction conflict (Enabled by default).
- **Mark Unimplemented Pcode** –places a warning bookmark on any instruction which has *unimplementedpcode*/semantics at the time disassembly was performed (Enabled by default). The *MarkUnimplementedPcode* script may be run to update these warnings since processor language updates may implement previously unimplemented instruction pcode.
- **Restrict Disassembly to Executable Memory** –If enabled will restrict disassembly to those memory blocks marked as executable (Not restricted by default).

The above *Options* may be changed via the program Options dialog which allows various program properties to be examined or modified. This dialog may be accessed via the menu action **Edit → Options for ...** for the currently active program. These options are program specific and stored within the program database. The above program properties can be accessed by clicking the *Disassembler* node within the *Properties* tree. Any changes to these options will be stored when the **OK** or **Apply** button is clicked within the dialog.

These option settings will be utilized for both forced disassembly and during auto-analysis.

## Static Disassembly

The difference between *Disassembly* and *Static Disassembly* is that

Static Disassembly does not follow flow and only disassembles one instruction. However, when disassembling in a selection, the disassembly starts at the first undefined byte within the selection and proceeds sequentially to the last byte.

To perform *StaticDisassembly*:

1. Place the cursor in the Code Browser on an undefined byte
2. Right-mouse-click
3. Select Disassemble (Static).

The disassembly progress is displayed in the progress status area (bottom right) of the tool. The progress status area displays the number of instructions disassembled and provides a "Cancel" button. Press the "Cancel" button to stop the disassembly process. Note: this will not undo any disassembly that has already occurred.



*Press the 'Ctrl-D' key to Static Disassemble one instruction starting at the current cursor location.*



*Selecting [Undo](#) will return the disassembled instruction to undefined bytes.*

## Disassembly (Restricted)

*RestrictedDisassembly* is similar to *Disassembly* except that only bytes in the current selection can be disassembled. Flows will be followed only if they are in the current selection. If there is no selection, only one instruction will be disassembled.

To perform *RestrictedDisassembly*:

1. Place the cursor in the Code Browser on an undefined byte.
2. Right-mouse-click, and select Disassemble (Restricted).

The disassembly progress is displayed in the progress status area (bottom right) of the tool. The progress status area displays the number of instructions disassembled and provides a "Cancel" button. Press the "Cancel" button to stop the disassembly process. Note: this will not undo any disassembly that has already occurred.



*Press the 'Alt-D' key to Disassemble only in a selected area.*



*Selecting [Undo](#) will return the disassembled instruction to undefined bytes.*

## Disassemble ARM / Disassemble Thumb

*Disassemble ARM and Disassemble Thumb* will only be available if the program you are working with is an ARM based processor. ARM processors have two states, ARM and Thumb mode. The instructions available in ARM mode are 4 bytes long. In Thumb mode, the instructions are "generally" 2 bytes long. ARM and Thumb mode are mutually exclusive, the processor is either executing Thumb encoded instructions or ARM encoded instructions. However, certain branching instructions can cause the processor to switch modes. The default mode for disassembly in an ARM program is to disassemble in ARM mode. If you come across a section of Thumb code that has not been disassembled, Disassemble Thumb will set the disassembler into Thumb mode and begin disassembly.

The actions work exactly like *Disassemble* in that they start disassembling at the address under the cursor, and will follow the execution flow.

To perform *Disassemble ARM or Disassemble Thumb*:

1. Place the cursor in the Code Browser on an undefined byte.
2. Right-mouse-click, and choose the appropriate option for the mode you believe the processor is using at the address.

The disassembly progress is displayed in the progress status area (bottom right) of the tool. The progress status area displays the

number of instructions disassembled and provides a "Cancel" button. Press the "Cancel" button to stop the disassembly process. Note: this will not undo any disassembly that has already occurred.



*Press the 'F11' key to Disassemble in ARM mode and press the "F12" key to disassemble in Thumb mode.*



*Selecting [Undo](#) will return the disassembled instruction to undefined bytes.*

## Modify Instruction Flow

With certain processors and situations it may be desirable to modify the default flow of an instruction to better reflect the nature of its flow. For example a jump may actually be performing a call type operation, a call may be performing a long-jump. This distinction primarily affects the subroutine models and flow analysis performed within Ghidra.

The following basic flow types may be imposed upon the default flow of an instruction:

- BRANCH
- CALL
- CALL\_RETURN
- RETURN

In all situations the conditional nature of the original flow is preserved.

To Modify Instruction Flow:

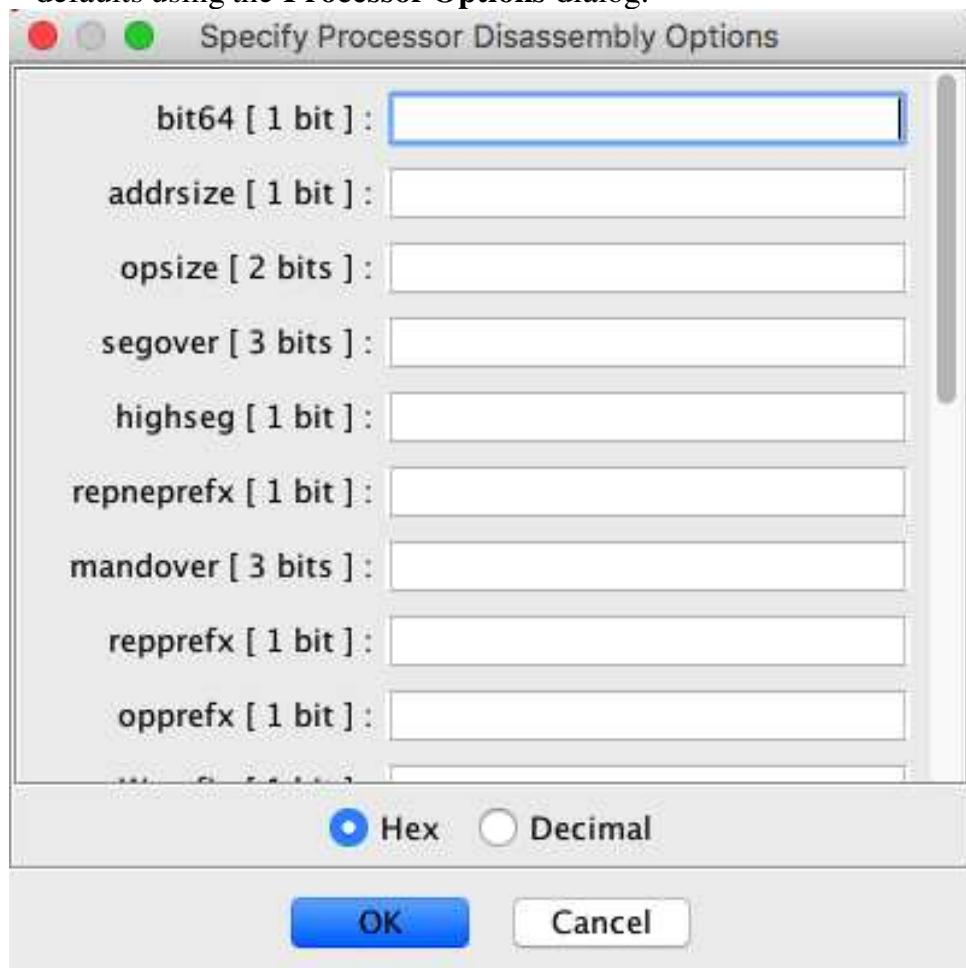
1. Place the cursor on an instruction within the Code Browser.  
Note that instructions which are purely fall-through can not be modified.
2. Right-mouse-click
3. Select *Modify Instruction Flow...* menu item.
4. Within the *Modify Instruction Flow* dialog select the desired basic flow behavior.
5. Click OK in the dialog



*An instruction whose flow has been modified will have its' mnemonic color modified.*

## Processor Options

Some processors have "modes" or state information that affects how they will interpret and execute the bytes that make up an instruction. To support this, Ghidra creates a pseudo register called the "contextRegister" that contains bits corresponding to the different modes or options of the processor. The language specifies a default for these modes, but the user can effectively change the defaults using the **Processor Options** dialog.



The options are specific to the processor and the user should refer to the processor manual for a description of a processor's modes. Whenever disassembly takes place these are the assumed values that will be used to initiate disassembly, unless the user has specifically

set the values at an address using the [\*\*Register Manager Window\*\*](#).

To changes the default processor options:

1. Place the cursor in the Code Browser.
2. Right-mouse-click, and select the "Processor Options" menu item.
3. Enter the values and press the "Ok" button.

Provided by: *Disassemblerplugin*

Related Topics:

- [Clear](#)
- [Importing Files](#)
- [Languages](#)

# Data

A newly imported program consists of bytes that have not yet been identified. These bytes are known as undefined data and are displayed using "?". Disassembly, analysis, and other user actions convert these bytes into *instructions* or *defined data*. In general, the term "data" refers to defined data.

## Data Types

Data is created by applying *Data Types* to bytes in memory. Data Types interpret bytes as values and provide a visual interpretation of those bytes based on the Data Type used, e.g., a four byte IEEE floating point number or a two byte little endian word. Ghidra comes packaged with a set of "Built-in" Data types (e.g., byte, word, float, etc). Ghidra also provides the capability of creating "User-defined" Data types (structure, array, typedef, etc) and supports dynamic data types whose structure depends on the underlying data.



It is important to note that the size of many of the primitive built-in types is determined by the language and compiler specification (e.g., the size of an integer can vary).

### Built-In Data Types

Undefined – Fixed Size Types	
Name	Size
undefined (default)	1
undefined1	1
undefined2	2
undefined3	3
undefined4	4
undefined5	5
undefined6	6
undefined7	7
undefined8	8

Settings	
Mutability (normal, volatile, constant)	

Numeric – Fixed Size Types			
Name	Mnemonic	Size	Signed/Unsigned
byte	db	1	Unsigned
sbyte	sdb	1	Signed
word	dw	2	Unsigned
sword	sdw	2	Signed
uint3	uint3	3	Unsigned
int3	int3	3	Signed
dword	ddw	4	Unsigned
sdword	sddw	4	Signed
uint5	uint5	5	Unsigned
int5	int5	5	Signed
uint6	uint6	6	Unsigned
int6	int6	6	Signed
uint7	uint7	7	Unsigned

int7	int7	7	Signed
qword	dqw	8	Unsigned
sqword	sdqw	8	Signed
<u>Settings</u>			
Endian (*default, big, little) Format (*hex, decimal, octal, binary, ascii) Mnemonic-style (default, *assembly, C) Mutability (normal, volatile, constant) Padding (*unpadded, padded)			

Miscellaneous -Fixed Size Types			
Name	Mnemonic	Size	Signed/Unsigned
void	void	0	n/a
wchar16	wchar16	2	Signed
wchar32	wchar32	4	Signed
<u>Settings</u>			
Mutability (normal, volatile, constant)			

Numeric -Dynamic Size Types	
<i>size determined by data organization within compiler specification –order is based upon relative sizes</i>	
Name	Signed/Unsigned
short	Signed
ushort	Unsigned
int	Signed
uint	Unsigned
long	Signed
ulong	Unsigned
longlong	Signed
ulonglong	Unsigned
<u>Settings (may vary)</u>	
Endian (*default, big, little) Format (*hex, decimal, octal, binary, ascii) Mnemonic-style (default, *assembly, C) Mutability (normal, volatile, constant) Padding (*unpadded, padded)	

Miscellaneous -Dynamic Size Types	
<i>size determined by data organization within compiler specification</i>	
Name	Description
char	Signed ASCII character
uchar	Unsigned ASCII character
wchar_t	Signed Wide Character
pointer	Pointer to memory address (may refer to another data type)
float	Floating point data
double	Double-precision floating point data
longdouble	Long double-precision floating point data
<u>Settings</u>	

String Types			
Name	Charset	Charsize	Layout
<a href="#">string</a>	settable	from charset	fixed length
<a href="#">string-utf8</a>	UTF-8	1-3	fixed length
<a href="#">TerminatedCString</a>	settable	from charset	null-terminated
<a href="#">unicode</a>	UTF-16	2 bytes	fixed length
<a href="#">TerminatedUnicode</a>	UTF-16	2 bytes	null-terminated
<a href="#">unicode32</a>	UTF-32	4 bytes	fixed length
<a href="#">TerminatedUnicode32</a>	UTF-32	4 bytes	null-terminated
<a href="#">PascalString255</a>	settable	from charset	pascal 255 max characters
<a href="#">PascalString</a>	settable	from charset	pascal 64k max characters
<a href="#">PascalUnicode</a>	UTF-16	2 bytes	pascal 64k max characters

Settings
<ul style="list-style-type: none"> <li>● Charset (defaults to US-ASCII if user settable)</li> <li>● Render Unicode (all, byte sequence, escape sequence)</li> <li>● Translation</li> <li>● Mutability (normal, volatile, constant)</li> </ul>

### User-Defined Data Types

User-Defined Data Types	
Name	Description
structure	Grouping of data types that are located consecutively in memory
union	Grouping of data types that share the same memory location(s)
typedef	An alias for an existing data type
enum	A list of named integer constants
pointer	A reference to an address (type information optional)
array	A specified number of consecutive data objects of the same type
function definition	A function signature tagged with a generic calling convention

### Applying Data Type

Data is created by *applying* a data type to undefined bytes in memory. There are numerous ways to apply data types. A status message indicating whether or not data was created is displayed in the tool's [status area](#).



Regardless of how a data type is applied, data is only created if the data type will fit within the available undefined bytes.

## Drag from Data Type Manager

Use the [Data Type Manager](#) to choose a data type from the set of available data types. To apply a data type from the data type Manager:

1. Open the Data Type Manager (select **Window ➔ Data Type Manager**)
2. Select a data type from the Data Type Manager
3. Drag the data type onto a single address or selection in the Listing window

## Cycle Groups

Cycle Groups are an easy way to apply *basic* data types (byte, word, float, etc). A Cycle Group is a collection of similar data types that are commonly associated together. Cycling a data type facilitates changing from one data type to the next data type in the same group. When the last data type is reached, the cycle restarts.

Each Cycle Group has an associated "hot key". Pressing the "hot key" cycles from the current data type to the next one within the group. Ghidra provides the following Cycle Groups:

<b>b</b>	Byte ➔ Word ➔ DoubleWord ➔ QuadWord ➔ Byte
' (single quote)	Ascii ➔ String ➔ Unicode ➔ Ascii
<b>f</b>	Float ➔ Double ➔ Float

To apply a data type from a cycle group:

1. Pick a basic data type to apply (for example double-word)
2. Place the cursor on the desired address or make a selection
3. Using the keyboard, press the associated "hot key" until the data type appears ("b" in this case, 3 times) . In this example, the first time "b" is pressed, a byte (db) will appear in the mnemonic field. The second time a word (dw) will appear, and the last time a double-word (ddw) will appear.
4. If you pressed the "hot key" too many times and passed the data type, continue pressing the "hot key" until you reach the data type again.

## Favorites

A *Favorite* data type is a data type that you use frequently and want to apply from the Data popup menu. By default Ghidra sets most commonly-used data types as Favorites. However, any data type can be [configured](#) to be a favorite.

To apply a favorite:

1. Right click on cursor location or selection
2. Select the **Data** popup submenu
3. In the Data popup, select the favorite data type to be applied



To quickly assign a key binding to a favorite, use the [Key Bindings Shortcut](#). Key Bindings allow you to assign "hot keys" to any menu item.

## Recently Used Data Type

The last applied data type is always available at the bottom of the data menu. By default, the "hot key" assigned to this option is 'Y'. This feature is a useful shortcut for applying the same data type multiple times.

## Clearing Data

Data and instructions can be [cleared](#), reverting them back to their undefined state.

To clear defined data back to an undefined state:

1. Place the cursor on the defined data to be cleared
2. Press mouse-right, and choose [Clear code bytes](#) from the popup menu

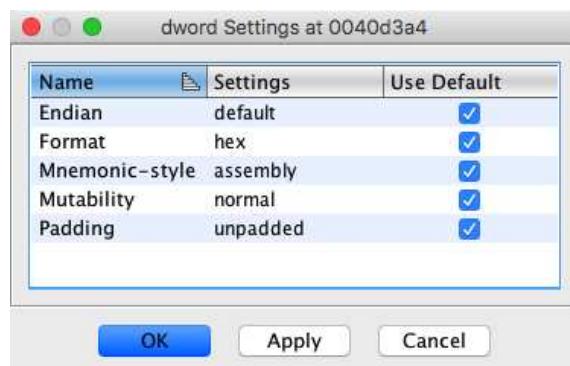
 [Deleting data types](#) from the *Data Type Manager* window is a quick way to clear every instance of a data type in a program.

## Changing Data Settings

Depending on the data type, data may have settings available which affect the way it is displayed. For built-in data-types, the available settings are indicated in the [table above](#). For example, the byte data type has a format option which allows the user to have the data displayed as hex, octal, etc. The available settings are defined in the [Data Type Table](#) above. Data settings can be changed for an individual data item or for all data of the same data type.

To change the settings of a single data item:

1. Place the cursor on the data item
2. Press mouse-right to bring up the popup menu
3. Select **Data ➔ Settings...** to bring up the settings dialog



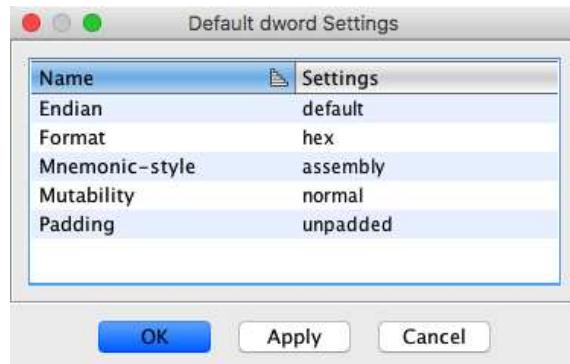
In the screenshot above, the settings dialog is shown for a byte data type applied at address 0x01007192. The dialog **shows all the available settings for that data type. For each setting, the dialog shows the setting name, value, and a checkbox indicating whether or not the setting matches the default setting.**

To change a setting, click on its value in the Setting column. This will display a list of valid choices for that setting. Choose a value from this list of choices and press **OK**. To reset a setting to its default value, set the **Use Default** checkbox.

The default settings for a given data type can be changed. When a default setting has been changed, every data item *currently* using the default setting for that data type will use the new default value. Data items that have a modified value for that setting will not be affected.

To change the default settings for a given data type:

1. Place the cursor on a data item of that type
2. Press mouse-right to bring up the popup menu
3. Select **Data ➔ Default Settings...** to bring up the default settings dialog



In the screenshot above, the default setting dialog is shown for the word data type. The dialog shows all the available settings for that data type. For each setting, the dialog shows the setting name and value.

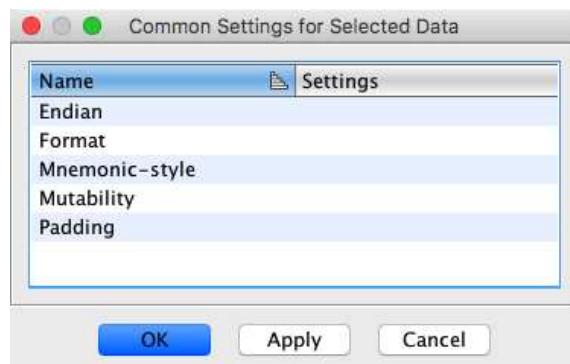
To change a setting, click on its value in the Settings column. This will display a list of valid choices for that setting. *Choose a value from this list of choices and press OK.*

 *Default* settings on components within a structure apply to that structure only. For example, if Struct\_1 has a byte component with its format set as Octal, then only other instances of Struct\_1 will be affected. Settings on other occurrences of byte are *not* affected.

 A typedef has the same set of settings as its underlying data type. For example, when you create a typedef on a byte, the default settings on the typedef will be the same as the original default settings of the byte. Changing the *default settings for either the typedef or its underlying data type doesn't affect the default settings of the underlying data type or typedef* respectively.

To change the data settings for multiple data items:

1. Select data items of interest (does not work for interior array elements)
2. *Press mouse-right to bring up the popup menu*
3. Select **Data ➔ Settings...** to bring up the settings dialog



In the screenshot above, the common data settings dialog is shown for the current selection. The dialog shows all the settings common to all defined data within the current selection. For each setting, the dialog shows the setting name and value, where the initial value is blank. Settings left blank will not be affected, allowing specific settings to be changed without affecting others.

## Creating Other Data Types

### Structure

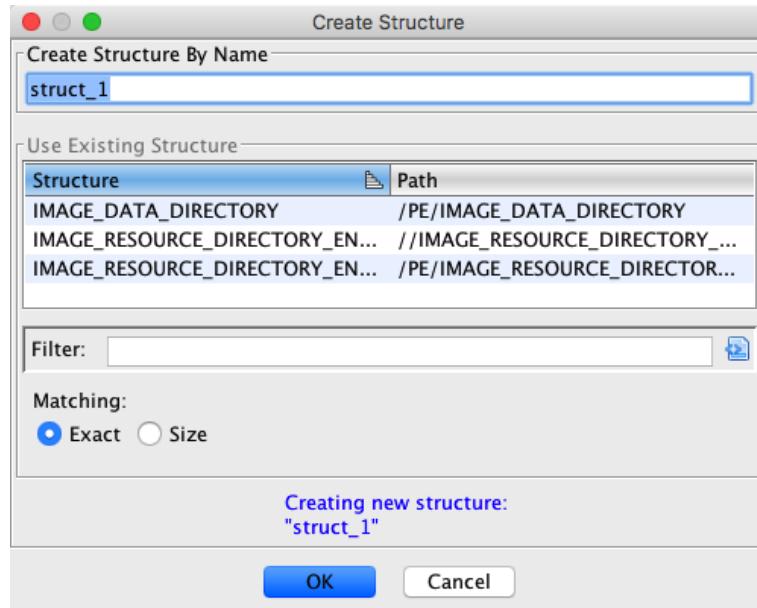
Ghidra provides two mechanisms to create and modify Structures. The [Data Type Manager](#) can be used to create structures without applying them immediately. It can also be used to edit them. Alternatively, structures can be created and modified *directly* in the browser.

This document describes how to directly create, edit and apply Structures in the browser.

### *Creating a new Structure*

Creating a new structure in the browser uses previously defined data to define the structure. The structure is created and applied at the same time. To create a structure *directly* in the browser:

1. Select a set of contiguous defined or undefined Data
2. Use the right-mouse popup **Data** → **Create Structure**
3. The Data items within the selection are used to define the structure
4. The **Create Structure** dialog is displayed. This dialog can be used to either create a new structure or apply an existing structure that has a matching format. If a new structure is created, it will be added to the Data Type Manager.

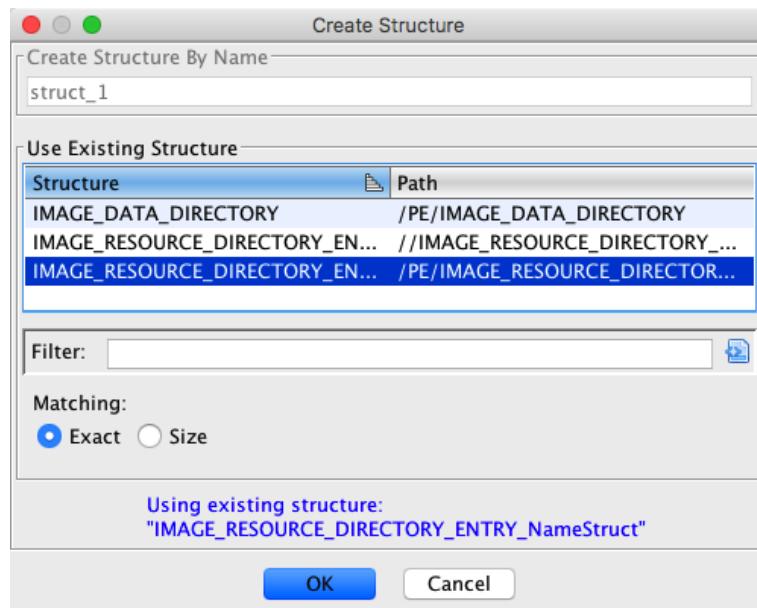


The **Create Structure** dialog is divided into two parts. In the top part is used to provide a name if you are creating a new structure. It is initialized with a default name. The bottom part shows a list of matching structures. You can select from this list to use that structure instead of creating a new one.



The method for finding matching structures is either by an exact match or by a match on structure size. You can change the type of matching used by clicking the *Exact* or *Size* radio buttons under the *Matching:* heading.

To create a new structure, enter a unique name in the **Create Structure By Name** text field and press OK.



To use an existing, matching structure, find and select the structure in the **Use Existing Structure** table and press OK.



You can create nested structures by following the instructions above. The only difference is in Step (1), select contiguous bytes *within* an existing structure.

#### *Changing a Structure Name*

1. Place the cursor on the first line of the structure
2. Press mouse-right over the structure and choose **Data ➔ Edit Data Type...**
3. Change the name in the [Structure Editor](#)

#### *Changing the name of a Structure member*

There are two ways to rename a structure member. The first way is useful for quickly changing the name of a single member:

1. Right mouse click on the field name of a structure member in the Code Browser
2. Choose the **Data ➔ Rename Field** option
3. Enter a new name in the **Rename Data Field** dialog



The "Field Name" field must be added to the "Open Data" tab in the Code Browser [header](#) in order for the data structure field names to show up in the Code Browser.

The second way is more useful for changing the names of multiple members:

1. Place the cursor on the first line of the structure
2. Press mouse-right over the structure and choose **Data ➔ Edit Data Type...**
3. Edit the field name for the structure member



You cannot set the field name of undefined member

#### **Union**

Unlike structures, unions can only be created using the [Union Editor](#) in the [Data Type Manager](#). Once a union has been created, it can be applied like any other data type.

## Enum

An Enum (*Enumeration*) data type is a C-style data type that allows the substitution of a value for a more meaningful name. Enums are created from the [Data Type Manager](#) using the [Enum Editor](#). When you apply the enum to a program, the name of the enum appears in the mnemonic field and the named value corresponding to the byte at that location appears in the operand field.

Example: Define an enum *color* with values *red*, *green*, and *blue* assigned to 0x00, 0x01, 0x02 respectively. When you apply *color* to:

```
010062ab    01    ??    01h
```

the result is:

```
010062ab    01    color    green
```

If the byte value is not one of the values defined in the enum, "Unknown value: <value>" is displayed as the operand.

## Pointer

A pointer data type points to another data type, including other pointers. Pointers can be *typed* or *untyped*. Typed pointers specify the data type of the referred-to location. By default, pointers have the same size as the size of an address for a particular processor. For example, the pointers on a 32-bit processor will be 4 bytes long.

To apply an untyped pointer:

1. Place the cursor over an undefined data
2. Press the 'p' Quick-Key  
– or –  
Drag a Pointer data type from the *Data Type Manager* window
3. A default-sized pointer is created
4. If a valid address can be formed at the location where you created the pointer, a reference is created to that address.

To create a typed pointer:

1. First create an untyped pointer (described above)
2. Apply a data type to the untyped pointer
3. The mnemonic will change from 'addr' to the referenced data type's mnemonic (ie: for a byte "db \*", for a pointer "addr \*")

Pressing the 'p' key invokes the pointer action. This will generally create a default pointer unless the existing data is already a pointer in which case that pointer will be wrapped with an additional pointer (e.g., int \* would become int \*\*). This action will always apply a default sized pointer. Otherwise you can drag one of the other pointer types from the *Data Type Manager* window. With existing pointer data, the base type of that pointer may be changed simply by applying another type onto the pointer (e.g., applying byte to default pointer becomes db \*). If you do not want this pointer stacking behavior to happen, it is best to clear the code unit(s) before applying a data type via drag-n-drop or key-binding actions.



To create a pointer of a specific size apply either pointer8, pointer16, pointer32, or pointer64 to create a pointer sizes of 1, 2, 4, 8, respectively.

## Array

An array is a collection of data items of a single data type. The number of elements in the array is specified when the array is created. Arrays can also be multi-dimensional. In this case, the innermost dimension is created first.

To create an array,

1. Place the cursor at the address where you want to create an array
2. Create one data item of the base data type for the array. Any data type is valid, including structures
3. Press the **T** Quick-Key,  
— or —  
Press mouse-right on the data item and choose **Data ➔ Create Array**
4. A dialog will prompt you for the number of elements in the array. It will be initialized with the maximum number of elements that will fit into the available undefined bytes. It will also let you know the largest array you can make if you allow it to clear existing data.
5. Enter the number of elements in the array, and press **OK**

To create a multi-dimensional array:

1. Create an array using the inner dimension for the number of elements
2. Place the cursor over the new array
3. Press the **T** Quick-Key,  
— or —  
Press mouse-right on the array and choose **Data ➔ Create Array**
4. Enter the number of elements for the next dimension, and press **OK**
5. Repeat steps (3) and (4) until all dimensions have been created

## Typedef

A **typedef** is an alias for another data type. It is useful for giving a more meaningful name to a data type. For example, you might **typedef dword** to be **int**. **Typedefs** are [created](#) using the Data Type Manager and applied like any other data type.

## Void

A **void** data type can only be used as the return type (i.e., <RETURN> variable) of a function and can be specified from the [Function Editor](#), [Set Data Type](#) popup action menu, as well as from the Data Type Manager using *drag-and-drop*.

## String Data Types

A String consists of a sequence of characters and is generally terminated by a null character (`\0`) or has a length value prefixed before the string.

### Characters in a string

The characters that make up a string can be encoded from bytes in a multitude of ways:

- Single bytes from the ASCII or other character set.
- 16 bit or 32 bit int values from the respective Unicode character sets.
- Variable length byte/int sequences that encode a single character, such as UTF-8 or UTF-16.

### Size of a string

The extent of the string is determined by:

- Null terminating character (`\0`).
- Containing field length (i.e. fixed length strings, or length of an array of characters).
- Prefixed length value for Pascal strings.

Each Ghidra string data type will document if its null-terminated or fixed length or pascal in its description or its type name.

The difference between null-terminated and fixed length strings is subtle. A null-terminated string will extend until a null character is found (with a sanity check max of 16k), regardless of the size of the field/structure that contains the string, whereas a fixed length string's length

is determined by its containing field, with trailing null characters trimmed, but interior null characters preserved.

In practice, the user will be unable to create a null-terminated string in Ghidra that exceeds its containing field/structure as the UI will size the containing field to match the detected length of the null-terminated string. However, if the bytes that make up the contents of the null-terminated string are changed, and the null-terminating characters are overwritten, the string instance could use bytes from outside of its footprint to construct itself. In this case the string will be displayed in red to indicate that there is an issue.

### Character sets

Character sets define how characters are represented as byte values, and how byte values are converted into characters.

Not all byte values are valid character encodings, and may result in undecodeable values –for example, byte values from 128..255 are not valid when using a US-ASCII character set, but are valid when using IBM437.

When an invalid mapping is encountered, it will be represented as the Unicode character ' REPLACEMENT CHARACTER', which will render on your screen with an OS and font specific shape (it is called the "REPLACEMENT CHARACTER" and is typically encoded as [uFFFD]).

The following character sets are always available:

- US-ASCII
  - limited to values between 0-127.
- ISO-8859-1 (Latin 1)
  - see also windows-1252.
- UTF-8
  - variable length 1-3 byte Unicode encoding.
  - only Unicode values greater than 007F cause multi-byte sequences, otherwise indistinguishable from US-ASCII.
- UTF-16, UTF-16BE, UTF-16LE
  - 2 byte Unicode encoding.
  - variable length, 2 or 4 bytes
- UTF-32, UTF-32BE, UTF-32LE
  - 4 byte Unicode encoding.

Other character sets that are typically implemented in the Java JVM:

- IBM437
  - old school extended ASCII.
- Windows-1252
- GB2312
  - Chinese
  - variable length 1-2 bytes
- Many many more...

### Unicode Byte Order Marks (BOM)

Unicode strings can start with a special character that signals the endian-ness of the string. The BOM character bytes will be FE FF (16 bit) or 00 00 FE FF (32 bit) if the string is big endian, otherwise it will be FF FE (16 bit) or FF FE 00 00 (32 bit) if the string is little endian.

If the BOM is present, it will override the endian-ness of the binary that contains the string.

### Arrays of character elements

Arrays of character elements (ie. char[16]) are treated as fixed-length string data types.

Arrays of wide char data types (wchar, wchar16, wchar32) are treated as Unicode strings.

### Creating string instances

When creating a String at a location, consecutive characters will be included in the String until a null character (\0) is encountered.

When applied to a selection, String data types absorb all bytes in the selection into a single string ignoring terminators.

TerminatedCStrings, on the other hand, create multiple strings for the selected bytes, beginning a new string at each terminator.

### Settings for string instances

Each string instance has settings that can be customized to change the way the string is decoded and how it is rendered when displayed.

- Charset
  - Any of the currently available java.nioCharsets.
  - Defaults to US-ASCII
  - See charset\_info.xml to customize display order or character size.
  - Not available on string types that have "UTF\*" or "Unicode" in the name.
- Render non-ASCII Unicode
  - all – attempt to render the character (display font may or may not provide it)
  - byte sequence – show the bytes that make up the problematic character
  - escape sequence – show as an escape sequence – "\u1234"
- Translation
  - Toggles display of translated string value on and off.
  - Same as popup menu action **Data | Translate | Toggle show translated value.**

## Dynamic Data Type

Dynamic data types adapt to the underlying bytes to which they are applied. These data types can only be created by [writing](#) a new Java class. For example, an IP header packet that has a header, body, and terminator might be a good candidate for writing a Dynamic data type. The header might specify the length of the body. The Dynamic data type can change its size and structure based on the information stored in the IP header.

The PE (Windows Portable Executable) data type is another example of a dynamic data type that manufactures new data types. When you apply the PE data type, it creates (1) a new [category](#) (using the address as the name) in the program data type manager, (2) a structure for the DOS header, and (3) a structure, PE, that contains the DOS header. In order for the PE data type to be successfully applied, you must import a DOS program as binary file.. The size of the structure varies according to the information in the program.

Provided By: *Data* Plugin

#### Related Topics:

- [Manage Data Types](#)
- [Structure Editor](#)
- [Enum Editor](#)

# Data Type Manager

The *Data Type Manager* allows users to locate, organize, and apply data types to a program. Allowing the user to build libraries of data types and to share them between programs, projects, and different users is a long term goal for Ghidra.

Prior to Ghidra 4.3, sharing data types even between programs in the same project was very difficult. (users would have to drag data types, one at a time, from one program or archive to another to propagate changes to data types). As of Ghidra 4.3, significant progress has been made to make sharing data types easier, but the inherent complexity of this problem requires users to understand a number of basic concepts to take advantage of the new features.

## Topics

- [Basic Concepts](#)
- [Data Type Manager Window](#)
- [Working With Data Type Archives](#)
- [Working With Categories](#)
- [Working With Data Types](#)
- [Managing Archives](#)

## Basic Concepts

### [Data Types](#)

Ghidra supports three kinds of data types: "Built-in", user defined, and derived.

<b>Built-in</b>	The built-in data types are implemented directly in Java and are used for the basic standard types such as byte, word, string, etc. They can't be changed, renamed, or moved within an archive.
<b>User Defined</b>	There are four user defined data types: Structures, Unions, Enums, and Typedefs. They can be created, modified, and renamed as needed.
<b>Derived</b>	There are two derivative data types: Pointers and Arrays. Pointers and arrays can be created and deleted as needed, but they take on a name derived from their base type.

### [Data Type Archives](#)

Data type archives are used to bundle and share data types between programs, projects, and users. There are two different types of user created archives: File data type archives and Project data type archives.

Data type archives can be accessed within the *Data Type Manager* window. When a data type archive is open, it is displayed as a node in the *Data Type Manager* tree. Archives can be opened by the user or automatically when a program is opened which references that archive. Data type archives can be [open for modification](#) or as read-only. Within the *Data Type Manager* window there are actions for opening, closing, renaming, and making archives modifiable.

### Built-In Data Type Archive

The "Built-in" archive is a special archive that is always available in the *Data Type Manager* window. It provides access to all of Ghidra's "Built-in" data types. "Built-in" types are discovered by searching through Ghidra's class path.



In the unlikely event that new data type class files or jar files are added while Ghidra is running, there is a refresh action which will find and add the new data types to the "built-in" archive.

### File Data Type Archive

File data type archives (or simply: file archives) store data types in files and have ".gdt" filename extensions. They can be located anywhere in the filesystem. File archives can be open by more than one user at the same time (assuming it is located on a shared filesystem), but only one of those users can have it open for modification.

### Project Data Type Archive

Project data type archives (or simply: project archives) store data types in Ghidra project database directories, along with the programs for that project. The [Ghidra Project Window](#) will display project archives in the same tree structure used to show the programs in the project. Project archives can be versioned and shared in a multi-user environment just like programs. They use the same "check-in/checkout" semantics for updating and sharing changes.

### Data Types in a Program

Besides being stored in archives, data types are also stored inside programs.



Any data type used in a program must be stored in that program even if it originally came from an archive.

Because of this, the "same" data types can live in more than one archive or program and can be modified in different ways in each place where it lives. Prior to Ghidra 4.3, keeping data types consistent between archives and programs was a problem because there was no way to maintain an association, except by location and name, which was

difficult due to potential conflicts. Now additional information such as source archive, unique ID and various change times are maintained to facilitate keeping data types consistent across archives and programs.

## Categories

Within a program or data type archive, data types can be organized using categories. Categories are like folders in a filesystem and allow data types to be organized into a hierarchical structure. Categories are ignored for purposes of [data type synchronization](#). In other words, the same data type might be in a category named "aaa" in an archive and be in a category named "bbb" in the program. As far as Ghidra is concerned, the different category does not constitute a data type difference.

## Data Type Manager Tree

The *Data Type Manager* window organizes data types using a tree structure. At the first level below the tree root, a node will exist for each open archive and the "Built-in" archive. If a program is open, a node will also exist at this level representing the program. In many ways, an open program behaves like a data type archive.

## Applying Data Types

The primary purpose of a data type is to apply it to various elements in a program. Data types can be applied to memory locations to provide interpretations of the bytes at those locations. They can also be used to describe function parameters and local variables.

One way to apply data types is to drag them from an archive and drop them on an element in the program [Listing](#). This action will cause a copy of the data type to be added to the program and then a reference to the copy will be used to annotate the program element. There are other ways of applying data types to a program and they are described in more detail [later](#).

## Resolving Data Types

When a data type from an archive is applied to a program, a copy of that data type is created in the program. The process of making that copy is called *resolving*. This is a complicated process because data types can contain or reference other data types, possibly in a circular fashion (via pointers). These referred data types may or may not already exist in the program and the resolving process must account for this. Sometimes, a data type might already exist with the same name as the *resolving* data type. Even if the conflicting data type is equivalent, Ghidra may not be able to determine if the existing data type is really meant to be the same as the *resolving* data type. Generally, such conflicts are resolved by renaming the new or moved type by appending ".conflict" to its name. For many of the data type actions initiated from the Data Type tree window, the specific conflict resolution is determined by the current [Data Type Conflict Mode](#). In the end, applying a single data type to a program can cause many new data types to be added to the program.

## Source Archive

Whenever data types are resolved from an archive to a program or another archive, the default behavior is to tag the copied data types with information about the archive from which the data types originated. So each archive or program maintains a list of source archives from which it has associated data types. Ghidra uses the term *Source Archive* as a descriptor that identifies a file or project data type archive where a data type originated. Whenever a program is opened, the *Data Type Manager* gets a list of source archives from the program and automatically opens the corresponding data type archives if it can find them. Using information stored in the source archive object along with information from the actual archive and the change time on a data type, Ghidra can determine if data types are *out-of-sync* with their corresponding data type in a program or other archive.



The data type tree will indicate data types that have a source archive association by displaying the name of the source archive in parenthesis after the data type name.

`DataTypeName (SourceArchiveName)`

## Client Archive

Ghidra uses the term *client archive* to represent the archive that contains a data type taken from a [sourcearchive](#). This includes normal data type archives **and the program**.

## Update Source Archive Names

When new releases of Ghidra are installed, old datatype archives may be dropped. In some cases these old archives may be replaced with a differently named archive. When this happens, this popup action can be used on a client archive to update the old source archive references. This popup action is only available when such an archive name change has occurred. Any programs or other client archives which reference the old archive will have this popup action available to update the source archive name reference.

## Committing Changes To Source Archive

If changes are made to a data type that has an association with a source archive, those changes must be applied to the original in the source archive to keep the data types in sync. Ghidra uses the term *commit* to mean propagating changes from a data type back to its corresponding data type in the source archive.

## Updating Data Types From Source Archive

If changes are made to a data type in an archive that has associated data

types in a client archive, those changes can be pushed out to those associations. Ghidra uses the term *update* to mean propagating changes from a data type in a source archive to its associated copies in a client archive.

### [Reverting](#)

If changes are made to a data type that has an association with a source archive, that data type is now different from its source data type. Ghidra uses the term *revert* to throw away the changes and put the data type back to the way it is in the source archive. If changes are made in both the referenced data type and the source data type, the *revert* action will not be available because the original state is not known. In this case, you must either [commit](#), which will lose the changes in the source archive, or [update](#) losing the changes in the client archive. Currently, there is no merge capability.

See [Reverting Changes](#) for more information.

### [Synchronizing an Archive](#)

When changes are made to data types in either the source archive or client archive, the archive is said to be *out-of-sync*. Ghidra provides a capability known as *synchronizing* to find all the data types that need to be [committed](#), [updated](#), or [reverted](#) and allow the user to make a decision on each data type.

### [File Archive Path](#)

Ghidra maintains a list of directories used to search for file archives. This list is known as the archive path. Ghidra provides a dialog for adding, removing, and re-ordering the list of these directories. When Ghidra maintains source archive information for a file data type archive, it only stores the name of the archive and not the entire file path. Storing absolute paths would make the file archive references very brittle when the program is shared or moved to another location. By using archive paths, different users can maintain copies of the source archives in different locations on their computer and still be able to find the archives in their shared programs.

See also: [editing archive paths](#).

## Working with Data Type Archives

There are two types of Data Type Archives: File and Project.

<b>File Data Type Archives</b>	Files containing data type definitions, which have ".gdt" as their file suffix.
<b>Project Data Type Archives</b>	Special files located

within the Ghidra project directory structure, which also contain data type definitions. These are available in the [Ghidra Project Window](#) and can be saved as versions in a shared project repository.

The data types contained in an archive are organized into categories similar to the way files are organized into directories in a filesystem. Archives are useful for sharing data types with other users, or making your data types available for use in other projects. Normally, file archives are opened in a read-only mode, but can optionally be [opened](#) for editing. Project archives are normally opened for editing, since they support sharing and version control and therefore allow more than one user to modify them at a time. Only one user at a time can have a file archive opened for editing.

### **Opening a File Data Type Archive**

From the local menu ▾, select ***Open File Archive....***. A file chooser will appear. Use the file chooser to find and select the data type archive to open. A new node will appear in the tree for the newly opened archive. Also, the directory containing the newly opened archive will be added to the archive path if it is not already there.

### **Opening a Project Data Type Archive**

From the local menu ▾, select ***Open Project Archive....***. A Ghidra project data type archive chooser will appear. This chooser will show all the project archives in the current project. Use the chooser to find and select the project data type archive to open. A new node will appear in the tree for the newly opened archive.

### **Creating a New File Data Type Archive**

From the local menu ▾, select ***New File Archive....***. A file chooser will appear. Use the file chooser to select a directory and enter a name for the new archive. If an archive already exists with that name, a dialog will appear asking if the existing archive should be over-written. A new node will appear in the tree for the newly created archive.

### **Creating a New Project Data Type Archive**

From the local menu ▾, select ***New Project Archive....***. A Ghidra project data type archive chooser will appear. Use the chooser to select a folder and enter a name for the new archive. A new node will appear in the tree for the newly created archive.



You can also create a new Project Data Type Archive by dragging a File Data Type Archive (.gdt) file onto the [Ghidra Project Window](#). This will create a new Project Data

Type Archive populated with the same data types as the dragged File Data Type Archive.

### Closing a Data Type Archive

Select the data type archive to close, right-click on it and select the ***CloseArchive*** action. The archive will be removed from the tree.

### Opening a File Data Type Archive for Editing

When an archive is first opened, it is not editable. In order to make any changes to the archive, it must be open for editing. Select the archive to edit, right-click on it and select the ***Open for Editing*** action. If someone else is currently editing that archive, this action will fail and a dialog will appear explaining that someone else is already editing that archive.

### Closing a File Data Type Archive for Editing

When an archive that is [open for editing](#) no longer needs to be edited, then it should be put back to a read-only mode so that other users can then modify it. Select the data type archive to close for editing, right-click on it and select the ***Close for Editing*** action. If the archive has unsaved changes, a dialog will appear providing an opportunity to save the changes.

### Saving Changes to a File Data Type Archive

Whenever a data type archive has been [opened for editing](#) and has unsaved changes, the node will display its name with '\*' attached. For example the archive "MyArchive" will display as "MyArchive \*". To save these changes, right-click on the unsaved archive and select the ***SaveArchive*** action. The changes will be saved and the name will be updated to not show a '\*'.

### Saving a File Data Type Archive to a New File

Right-click on the file archive to be saved to a new file, and select the ***Save As...*** action. A file chooser will appear which can be used to choose a location and filename for the new archive that will be created. The tree will be updated to show the new name for the archive (the filename). The original archive file is unaffected.

### Deleting a Data Type Archive

Deleting an archive will not only remove the archive from the tree, but will permanently remove it from the filesystem. To delete an archive, right-click on it and select the ***DeleteArchive*** action. An archive file must be open for editing before this action will appear (see [Opening a Data Type Archive for Editing](#)).

## Removing an Invalid Data Type Archive

When an archive file fails to open (when Ghidra can't find the file in the archive path or encounters a permission problem) it will be displayed with the  icon. If you wish to permanently remove the file path from the tool configuration and the current program options, you may right-click on it and select the **Remove Invalid Archive** action.

## Aligning All Data Types In a Program or Archive

Right-click on the program or data type archive where data types are to be aligned, and select the **Align All...** action. A confirmation dialog will appear to make sure you want to align all the structures and unions in the program or data type archive. If you continue, all structures and unions that are unaligned will be changed to aligned data types with no minimum alignment (the default) and no packing.

## Updating an Archive From a Source Archive

Datatypes within an archive that originally came from some other source archive may need updating if they have been changed in the originating archive. If a an archive has one or more datatypes that need updating, it will be marked with either the  or the  icon.

To update the datatypes, right-click on the node that needs updating and select **Update Datatypes From ➔<Source Archive Name>**. The Update Data Types dialog will be shown allowing you to select the datatypes to update.



Ghidra uses time stamps and flags to determine if an archive is out-of-sync. This can result in Ghidra indicating the archive needs updating when actually it does not. For example, if a data type is changed and then changed back, it will cause Ghidra to think the data type was changed. In this case, invoking the update action will cause Ghidra to search for updates, but when it finds none, a message dialog will appear indicating that no changes were detected and the archive will be considered updated.

## Committing Changes in an Archive To a Source Archive

Datatypes within an archive that originally came from some other source archive may have been changed and need to be pushed back (committed) to the originating archive. If a an archive has one or more datatypes that need committing, it will be marked with either the  or the  icon.

To commit the datatypes, right-click on the node that contains the changed datatypes and select **Commit Datatypes To ➔<Source Archive Name>**. The Commit Data Types dialog will be shown allowing you to select the datatypes to commit.



Ghidra uses time stamps and flags to determine if an archive is

out-of-sync. This can result in Ghidra indicating the archive needs committing when actually it does not. For example, if a data type is changed and then changed back, it will cause Ghidra to think the data type was changed. In this case, invoking the commit action will cause Ghidra to search for commits, but when it finds none, a message dialog will appear indicating that no changes were detected.



The source archive must be editable in order to commit File archives must be [open for editing](#) and project archives that are under version control must be checked-out.

### **Reverting Changes in an Archive Back To a Source Archive**

Datatypes within an archive that originally came from some other source archive may have been changed and need to be reverted back to same state as the source archive (i.e. discard the local changes) If a an archive has one or more datatypes that can be reverted, it will be marked with either the or the icon.

To revert the datatypes, right-click on the node that contains the changed datatypes and select **Revert Datatypes To ➔<Source Archive Name>**. The Revert Data Types dialog will be shown allowing you to select the datatypes to revert.

### **Disassociating Data Types in an Archive From a Source Archive**

Datatypes within an archive that originally came from some other source archive may be disassociated from their source archive. This will prevent them from being updated or committed back to the source archive. If a an archive has one or more datatypes that have source archive relationships, the **Disassociate** action will be available.

To disassociate datatypes, right-click on the node that contains the datatypes and select **Disassociate Datatypes From ➔<SourceArchive Name>**. The Revert Data Types dialog will be shown allowing you to select the datatypes to revert.

### **Refreshing Data Type Sync Indicators in an Archive For a Source Archive**

Datatypes that are associated with a source archive may have a commit, update, or conflict icon indicating they are out of sync with the data type in the source archive, when the data type actually matches the source datatype. This can happen if a data type is changed, but changed to match its source. Invoke the **Refresh** action to refresh all the sync indicators for that source archive.

To refresh sync indicators for datatypes associated with a particular source archive, right-click on the node that contains the datatypes and select **Refresh Sync Indicators For ➔<Source Archive Name>**.

### **Version Control / Multi-user Actions on Project Archives**

The full set of version control actions from the front-end project window are available when right-clicking on a project archive. See the [Version Control](#) section

of the Project Window for more information.

## Working with Categories

[Categories](#) are used to organize data types into a hierarchical structure. Categories can contain data types and other categories. Archive nodes represent the root or default category for their corresponding archive they represent in addition to representing the archive itself. In other words, most of the actions that apply to category nodes can also be applied to archive nodes. The two exceptions are **Delete** and **Rename**. The root category in an archive cannot be deleted and to rename it, you must use the **Save As...** action, since its name is the name of the archive.

### Creating a New Category

Right-click on the category where the new category is to be created. Select the **New ➔ Category** action and a new category named "New Category" will be created.

### Renaming a Category

Right-click on the category to be renamed. Select the **Rename** action and then type in the new name in the in-place text edit box.

### Deleting a Category

Right-click on the category to be deleted. Select the **Delete** action. A confirmation dialog will appear since **this action cannot be undone** (unless its in the program's archive)

### Moving a Category

Categories can only be moved **within the same archive**. Attempts to move categories across archives are converted to a copy action. When a category is moved, effectively all categories and data types contained in that category are moved as well. There are two ways to move a category:

- |  |  |
|--|--|
| 1. <b>Drag-N-Drop</b><br>2. <b>Cut/Paste</b> | Click on the category to be moved and drag it onto its new parent category.<br><br>Right-click on the category to be moved and select the <b>Cut</b> action. Then right-click on the destination parent category and select the <b>Paste</b> action. |
|--|--|

### Copying a Category

Categories can be copied within an archive or from one archive to another, but the behavior of the copy is very different for the two cases. When copying within an archive, the behavior is more natural. Copies

are made of the source category and its children and placed inside the destination category. However, when copying from one archive to another, the behavior is somewhat unusual. In this case, the selected categories and contained data types are copied into the destination category, but if there are additional data types that are referenced by the copied data types, those are copied into the destination archive as well. After the copy, the additional data types will appear in the same relative location as they exist in the source archive.

#### 1. Drag-N-Drop

Click on the category to be moved and copy drag (hold the <Ctrl> key while dragging) it onto its new parent category.

#### 2. Copy/Paste

Right-click on the category to be moved and select the ***Copy*** action. Next, right-click on the destination parent category and select the ***Paste*** action.

## Working with Data Types

Data types are the actual useful objects within archives. They can be applied to programs to bring meaning to the data, parameters, local variables and function return types contained in that program. User defined data types such as structures can be arbitrarily complex, consisting of other data types which can be built upon other data types and so on, until finally they are built on the primitive types (the built-in data types.)



Built-in types have several restrictions. They always live in the root category of an archive and they can't be renamed.

### Applying Data Types to a Program

Data types can be applied in several different ways:

#### 1. Drag-N-Drop

Data types can be dragged directly onto various locations in the [Listing](#) view. If dropped onto undefined bytes, a new Data object is created. If dropped onto a function, the return type can be set, etc.

#### 2. Favorites

Data types can be set to be a favorite. This causes a popup menu action to be generated for that data type whenever the mouse is right-clicked over the appropriate location in the [Listing](#) view.

#### 3. Last Used

Whenever a data type is applied to a program it is remembered as the "last used" data type and can be easily applied to other locations using a key binding or popup menu actions.



Applying a data type from an archive will automatically add that data type to the program's archive. Also, the archive will become associated with the program and automatically be opened whenever the program is opened.

### **Creating New User Defined Data Types**

There are seven types of data types that users can create: Structures, Unions, Enums, Function Definitions, Typedefs and Pointers.

Structures, unions, enums, and function definitions can be created by right-clicking on the category where the new type should be located, and then choosing either the **New ➔ Structure**, **New ➔ Union**, **New ➔ Enum** or **New ➔ Function Definition** action respectively. Each of these actions will bring up an appropriate editor ([structure editor](#) for structures and unions, the [enum editor](#) for enums and the edit function signature editor for function definitions) for creating the new desired data type.

Creating a new typedef is even easier. Right-click on the *data type to be typedef'ed* and select the **New ➔ Typedef on XYZ** action. A new typedef will be created on the XYZdata type in the same category as the original data type.

Alternatively, you can click **New ➔ Typedef...**, which will show a dialog that allows you to choose a typedef name and the data type from which the typedef will be created.\* This action can also be executed from any folder instead of directly on another data type.

To create a **pointer**, you can click **New ➔ Pointer to XYZ**. A new pointer will be created to the XYZ data type in the same category as the original data type.



\* If you create a typedef or pointer to a data type in the Built-in Data Type Manager , the newly created type will be placed in program's data type manager at the root category.

Structures can also be created directly in the [Listing](#) view. See [creating structures in the Browser](#) for details.

### **Renaming a Data Type**

Right-click on the data type to be renamed. Select the **Rename** action and then type in the new name in the in-place text edit box.

### **Editing a Data Type**

**Only structures, unions, enums and functionDefinitions can be edited.**

To edit one of these data types, either double-click on its node or right-click its node and select the **Edit** action. For structures and unions, the [structure editor](#) will appear, and for enums the [enum editor](#) will appear.

**[Creating a new Enum from a Selection of Enums](#)**

Select two or more existing enums. Select the **Create Enum from Selection** action. A dialog will appear asking you for the new enum's name. This name must be unique or you will be prompted to enter a unique name. The resulting enum will contain a combination of all names and values from the selected enums. NOTE: If more than one of the same value is contained in the enums, they will all be added to the new enum. However, only the first one entered will be applied when this enum is used.

**[Deleting a Data Type](#)**

Right-click on the category to be deleted. Select the **Delete** action. A confirmation dialog will appear since **this action cannot be undone** (unless it's in the program's archive)

**[Moving a Data Type](#)**

Data types can only be moved within the same archive. Attempts to move data types across archives are converted to a copy action. There are two ways to move a data type:

**1. Drag-N-Drop**

Click on the data type to be moved and drag it onto its new parent category.

**2. Cut/Paste**

Right-click on the data type to be moved and select the **Cut** action. Next, right-click on the destination parent category and select the **Paste** action.

**[Copying a Data Type](#)**

Data types can be copied within an archive or from one archive to another, but the behavior of the copy is very different for the two cases. When copying within an archive, the behavior is more natural. A copy of the source data type is placed inside the destination category. However, copying from one archive to another behaves somewhat unusually. In this case, the destination folder is only relevant as to which archive is the recipient of the copy. After the copy, the destination archive will contain the copied data type. However, any data types contained by the copied data type are also copied to exactly the same relative (to the root category node) category paths as the source archive. There are two ways to copy a data type:

**1.Drag-N-Drop**

Click on the data type to be moved

and copy drag (hold the <Ctrl> key while dragging) it onto its new parent category.

#### 2.Copy/Paste

Right-click on the data type to be moved and select the ***Copy*** action. Then right-click on the destination parent category and select the ***Paste*** action.

### Aligning a Data Type

Right-click on the structure or union to be aligned. Select the ***Align*** action. If the data type is unaligned it will be changed to an aligned data type with no minimum alignment (the default) and no packing.

### Committing Changes To Source Archive

If changes are made to a data type that has an association with a source archive, those changes must be applied (*committed*) to the original in the source archive as well to keep the data types in sync. Right-click on the data type to be committed. Select the **Commit To Archive** action and the changes will be applied back to the source archive.

### Updating Data Types From Source Archive

If changes are made to a data type in a source archive that has associated data types in a client archive (another archive or the program), the data type in the client archive can be *updated* from the source archive. Right-click on the data type to be updated. Select the **Update From Archive** action and the changes will be applied from the archive.

### Reverting Changes

If changes are made to a data type that has an association with a source archive, that data type is now different from its source data type. Those changes can be thrown away and the data type can be *reverted* back to its original state. Right-click on the data type to be reverted and select the **Revert** action and the changes will be removed.

### Disassociate a Data Type

To remove a data type association with a source archive, right-click on the data type in the client archive (another archive or the program) and select the **Disassociate From Archive** action. The data type will become a local data type within the client archive and any changes to it will not affect the original data type in the source archive.

### Associate a Data Type with a Source Archive

Whenever a data type is applied to a program, or dragged to a category

under the program's node in the data type manager tree from a file or project archive, a copy of that data type is created in the program. Also, an association back to the original data type is created. That is the normal case and it is designed to be fairly intuitive. Less intuitive is when a data type is originally created in a program and then is shared by dragging a copy to an archive. Since programs cannot be the source for a data type, a dialog is displayed asking the user if they want an association to be created. If the user answers yes, an association is created, but the archive will become the source and the program's data type is the one that gets the association. In other words, it appears as if the data type were created in the archive and copied to the program.

### Handling Data Type Conflicts

When you move or copy a data type to a category that has a data type with the same name, a conflict occurs. If the data types are not the same, then a dialog is displayed in order to resolve the conflict, as shown below:



In this example, you dragged (or pasted) the data type "SIZE\_T" from one category to the /basetsd.h category; the one being dragged is different from the one that already exists in the /basetsd.h category. The choices to resolve the conflict are:

1. *Rename* the data type that you are dragging to have ".conflict" appended to it to make a unique name.
2. *Replace* the existing data type with the one you are dragging (or, pasting); this means any use of the existing data types is *replaced* with the new data type; the existing data type is deleted.
3. Use the existing data type; if you did a cut/paste or drag/move operation, the "cut" or "dragged" data type is removed from its original category; the destination category is unaffected.

### Replacing a Data Type

A data type can be replaced by another data type. This means that every occurrence of the original data type in a program is replaced by the new data type and the original data type is deleted. There are two ways to replace a data type.

1. **Drag-N-Drop** Click on the replacement data type and

drag it onto the data type to be replaced.

## 2. Copy/Paste

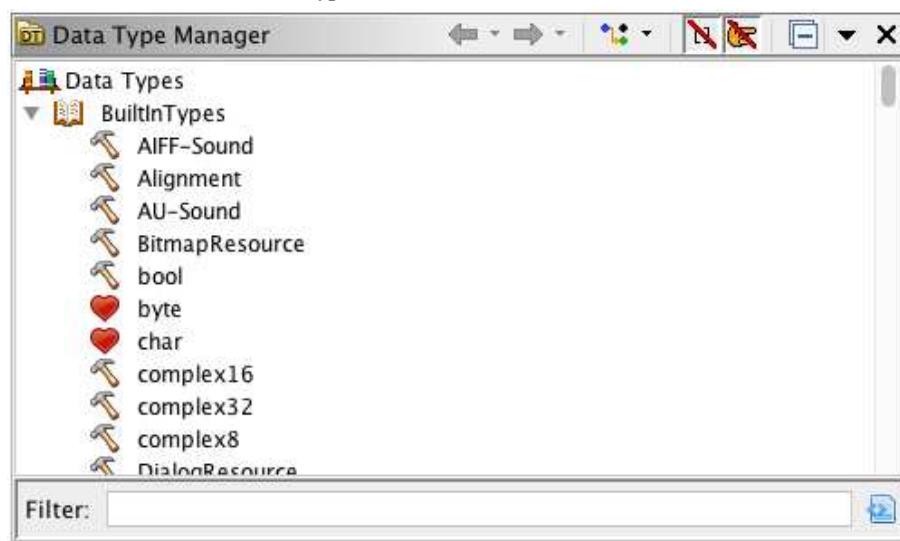
Right-click on the replacement data type to be moved and select the **Cut** action. Next, right-click on the data type to be replaced and select the **Paste** action.

Either way, a confirmation dialog will appear.

### Setting Favorite Data Types

Data types can be marked as *favorites* such that they show up in the **Data** option menu and the **Set Data Type** popup action menu in the Browser. This is a quick way to apply a data type to the Program. The default Code Browser has most of the well-known types in the **BuiltInTypes** category marked as a favorite.

To make a favorite, right-click on the data type and select the **Favorite** action. Favorite data types are marked with the ❤ icon.



To remove a favorite, right-click on the data type and deselect the **Favorite** action.

 The favorites are identified by name and must be unique, so you cannot have data type "fred" in one "categoryA" marked as a favorite and "fred" in "categoryB" also marked as a favorite.

 Any data type from any archive type (Program, BuildInTypes, or archive) can be marked as a favorite and used as such, however, only those marked in the **BuiltInTypes** category will be saved as part of your tool's state when you close the Project or exit Ghidra. Your list of favorites is restored when you re-open your project or restart Ghidra.

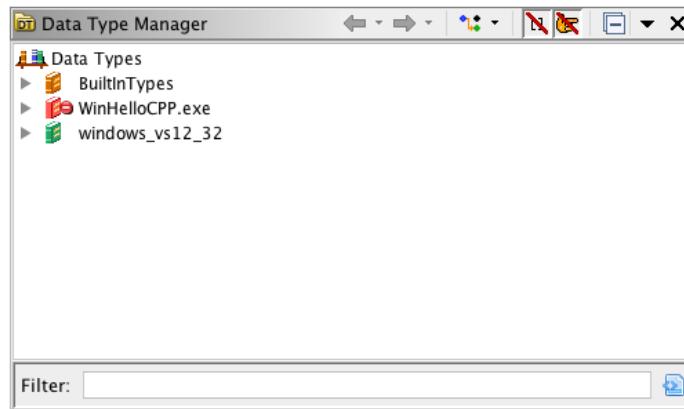
Provided by: *DataTypeManagerPlugin*

Related Topics:

- [Data Types](#)
- [Edit Plugin Path](#)
- [Apply Data Types](#)
- [Edit Structure](#)

# Data Type Manager Window

The *Data Type Manager* window shows the data types and their categories using a tree structure. To bring up the window, click on the  icon in the tool bar, or select the **Window ➔ Data Types...** menu option in the tool.



## The Data Type Tree

The first level nodes under the root  are *archive* nodes. There are two special archive nodes: one for standard ["Built-in"](#) data types and one for data types that are stored in the current program. Additional archive nodes are created for each open data type archive file. Archive nodes can contain both [category](#) nodes and "data type" nodes. Category nodes can contain data type nodes and other category nodes. Data type nodes are leaf nodes and they represent a specific data type.

The different colors of the archive nodes indicate the type of archive:

-  – Built-in archive data types
-  – Program data types
-  – Project Archive
-  – File Archive

Archive Icon modifiers (These icons are overlayed with the icons above)

-  – Indicates the archive is modifiable by you and possibly by others in your shared project
-  – Indicates the archive is currently modifiable only by you
-  – Indicates the archive is out of date (someone has checked changes for this archive into the repository)
-  – Indicates the archive is hijacked (modified without being checked out)
-  – Indicates the archive is a read only version and is not modifiable
-  – Indicates the archive has changes to datatypes that have not been committed back to its source archive. A commit is needed.
-  – Indicates the archive has datatypes from a source archive that has been modified. An update is needed.
-  – Indicates the archive has datatypes that need updating and datatypes that need to be committed.

Other Icons used in the tree:

-  – Category
-  – Built-in data type
-  – Structure
-  – Union
-  – Typedef
-  – Function Definition
-  – Enum
-  – Pointer
-  – Favorite

Data Type Icon modifiers (These icons are overlayed with the icons above)

-  – Indicates the data type has changes that need to be committed to the source archive
-  – Indicates the data type has been updated in the source archive and needs to be updated locally.
-  – Indicates the data type has changed both in this archive and in the source archive. (conflict)
-  – Indicates the data type is missing in the source archive.



Opening a program also causes file data type archives that have been associated with that program to be opened. As of Ghidra version 4.3, whenever a data type is applied from an archive to a program, that archive will become associated with the program.

By using the data type tree in various ways (drag-n-drop, cut/paste, menus, etc.), users can perform the following actions.

- Apply data types to appropriate places in a program (memory locations, parameters, local variables, etc.).
- Create new structures, unions, enums, and typedefs.
- Copy data types from one archive to another.
- Organize data types into categories and sub-categories.
- Rename data types and categories.
- Find data types by name.
- Create new file data type archives.
- Create new project data type archives.
- Delete data types and categories.
- Commit changes back to a source archive.
- Update changes from a source archive.
- Revert changes back from a source archive.
- Disassociate data types from a source archive.

## Filter

Located at the bottom of the tree component, there is a text field which can be used to quickly find data types by **name**. As text is entered into the filter, the tree is pruned such that only nodes **with names** (this means that members of composite types will not be matched) that match the entered text are displayed in the tree. This field interprets the '\*' character as a wild card and will match any sequence of characters. There is an assumed '\*' at the beginning of the text so that matches don't have to start with the entered filter text. For example, entering "foo" will not only match "foot" and "food", but will also match "snafoo."

The **Include Data Members in Filter** action, when toggled on, will trigger the filter field to search the components of Compositedata types and names, the fields of Enum names and values, and Function parameter data types and names. You may turn this feature off if you find that your search yields too many results for data types that have common names, such as `byte`.

## Data Type Conflict Resolution Mode

Anytime a data type is applied, copied or moved via the Data Type tree, name conflicts can occur with existing data types within the destination archive or program. While the default conflict handler simply appends ".conflict" to the new or moved data type name, actions initiated via the tree utilize a Data Type Conflict Mode identified and selected from the Data Type tree toolbar. Four conflict modes are supported:

– *Rename New or Moved Data Type* is the default behavior where a unique ".conflict" suffix is appended to the new or moved data type.

– *Use Existing Data Type* causes the existing data type within the destination category to be used in place of the new or moved data type. All references to the moved data type within the destination archive/program will be replaced with the existing data type within the destination category.

– *Replace Existing Data Type* causes the existing data type within the destination category to be replaced by the new or moved data type. All references to the existing data type within the destination archive/program will be replaced with the new or moved data type.

– *Replace Empty Structures else Rename* performs the default ".conflict" rename except in the case when adding a conflicting structure or union that appears similar to an existing structure. The intent is to keep the structure/union which appears to be more complete than the other in its field specification.

## Local Toolbar Actions

– Navigates forward and backward in the stack of visited data types.

– Collapses all open tree nodes, except for the root node.

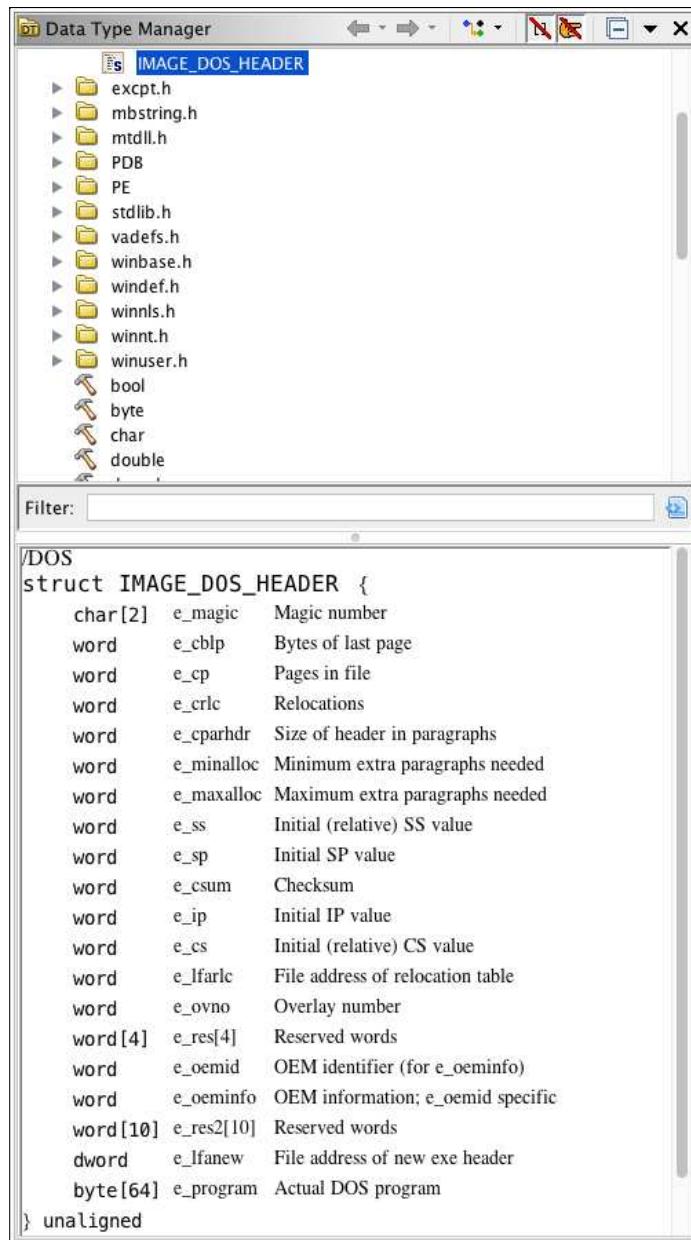
– Toggles the array filter on or off. If the filter is on, array data types are not shown in the data type tree.

– Toggles the pointer filter on or off. If the filter is on, pointer data types are not shown in the data type tree.

## Local Menu Actions

<b>Open File Archive...</b>	This action will launch a file explorer that can be used to find and open an existing <a href="#">file data type archive</a> . The location (file path) of the chosen file will be added to the <a href="#">archive path</a> if it isn't already there.
<b>Open Project Archive...</b>	This action will launch a project data type archive chooser that can be used to open an existing project archive. Project data type archives live in the project and can be seen in the <a href="#">Ghidra Project Window</a> . <i>Project data type archives can also be opened by dragging them from the Ghidra Project Window to the Data Type Manager tree.</i>
<b>New File Archive...</b>	This action is used to create a new file data type archive. A file chooser will be launched for choosing a location and name for the new file archive. The location (file path) of the new archive will be added to the <a href="#">archive path</a> if it isn't already there.
<b>New Project Archive...</b>	This action is used to create a new project data type archive. A project window browser will be launched for choosing a location in the project and a name for the new project archive.
<b>Standard Archive</b>	This menu option will have a sub-menu item for all file archives contained within the Ghidra installation. Only those archives contained within a module <code>data/typeinfo</code> subdirectory will be included. If contained within a contrib module, the contrib name will also be displayed.
<b>Recently Opened Archive</b>	This menu option will have a sub-menu item for all recently opened file archives. Choosing one of the recently open archives actions will open that archive.
<b>Refresh BuiltIn Types</b>	This action will search for new <a href="#">Built-in</a> data types in the classpath. This is only useful if <i>somewhat new data type java classes have been added since Ghidra was started.</i>
<b>Preview Window</b>	This action will show a preview window in the lower part of the data types window. This

preview window will be updated with a preview of the selected data type node. To close the preview window, simply click the **Preview Window** action again. Below is a picture of the data type manager with the preview window open.



**Find Data Types by Name...** This action will prompt for a search string and then launch a new *Find Data Type* window with its filter set to the search string.

**Edit Archive Paths...** This action will launch a dialog that can be used to add, remove, or reorder the directories used for searching for a file archive. Ghidra programs only store the name of the archive. Each directory in the [archive\\_path](#) is searched in order, looking for a file archive with that name. See the section on [editing archive paths](#).

## Popup Menu Actions

The set of actions on the right-mouse popup menu change depending on what nodes are selected. These actions will be described in the appropriate sections below.

## Miscellaneous Actions

[Cut](#)

The **Cut** action can be used to [move selected data types](#) and/or [move selected categories](#). The **Cut** action only primes the selected nodes to be moved. The **Paste** action must be used to complete the move. Any other **Cut** or **Copy** action will cancel the previous **Cut**. The **Cut** action can also be used to [replace one data type for another](#).

### **Copy**

The **Copy** action can be used to [copy selected data types](#) and/or [copy selected categories](#). The **Copy** action only primes the selected nodes to be copied. The **Paste** action must be used to complete the copy. Any other **Cut** or **Copy** action will cancel the previous **Copy**.

### **Paste**

The **Paste** action is used to complete a move or copy operation as initiated by either the **Copy** or **Cut** action. The node that is selected will be the destination for whatever nodes were selected when the **Copy** or **Paste** was invoked.

### **Rename**

The **Rename** action is used to [rename a selected category](#) or to [rename a selected data type](#).

### **Delete**

The **Delete** action is used to [delete data types](#) and/or [delete categories](#). A confirmation dialog will appear before actually deleting the selected data types and categories.

### **Collapse All**

To collapse all open nodes in a sub-tree, right-click on the root node of the sub-tree and select the **Collapse All** action. If this action is invoked via the local toolbar, then the entire tree is collapsed.

### **Expand All**

To expand all open nodes in a sub-tree, right-click on the root node of the sub-tree and select the **Expand All** action.

### **Apply Function Data Types**

You can apply all function signature data types from an archive to the currently open Program. Function signature definitions can also be applied from the currently open program's defined data types. Applying data types from the program is useful when source header files have been parsed into the program instead of an archive. **This action attempts to match the function definition with user defined symbol names in the Program.** When a match is found, the **Apply All** action does one of the following:

- If a function already exists at the symbol, apply the function signature to the existing function.
- If a pointer exists at the symbol, create a new pointer to the function definition.
- If no data exists at the symbol and if a valid function can be created at this address, then disassemble, create a function, and apply the function signature.

### **Capture Function Data Types**

You can capture the signatures of functions in the current program to function definition data types. These data types can be captured to the program or to a data type archive. To capture function data types, right-click on the data type manager tree node of the program or the archive where you want the function definition data types to be captured (saved) and select **Capture Function Data Types** action.

- If there isn't currently a selection in the listing, the signatures for all of the functions in the program will be captured.
- If there is a selection in the listing, only the functions whose entry points are within the selection will have their signatures captured.

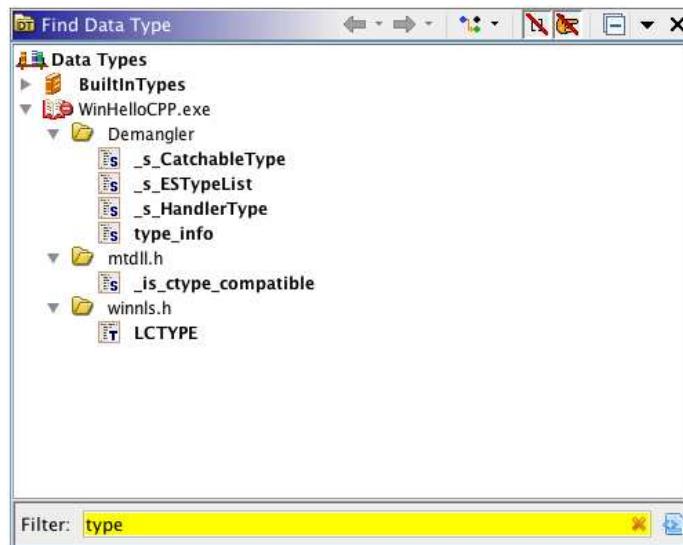
Once you have captured the function data types to an archive, you can use the [Apply Function Data Types](#) to apply the function definition data types to another program.

### Find Data Types by Name

You can find data types with names that match a string. A separate window is displayed to show the results. Multiple search results are shown as tabs in the results window. To find data types, select the Find Data Types by Name... action from the local menu. The following dialog is displayed.



Enter a value in the "Please enter the search string:" field, or choose a previously entered value from the *combo box*. Either press the <Enter> key or select the OK button. A new *Find Data Type* window will appear with the filter set to the entered search text.



This new window can be used in the same way as the original *Data Type Manager* window.

### Find Data Types by Size

This is similar to the **Find Data Types by Name** action (show above) except that it lets you specify a size that will be used to show only those data types with the same length.

### Export Data Type(s)

Data types may be exported as header files. To export a data type, select the data types to export, then right-click and select the **Export as C Header...** option. An output format dialog will appear. From this dialog select the language header output type (C header files are the default). A file chooser is displayed to choose the name of the output file. If the file already exists, a dialog is displayed to confirm that you want to overwrite the existing file.



Not only will the selected data types be exported, but so too will be any dependencies for the selected types. For example, if you select a structure to be exported, all of the types within that structure will be exported as well.

### Create Labels From Enums

Enum data types can be used to create labels with the names from the enumerated types wherever the value can be used as an address in the program. To create the labels, select the enum data types in the data type manager tree, then right click and select the **Create Labels From Enums** option. Labels will be created using the names from the enumerated types at program addresses based on their values as long as the name doesn't result in a conflict with another label.

### Show Base Data Type

For Typedefs, Pointers and Arrays, this action will navigate to the tree node of the type to which respective refers.

Provided by: *DataTypeManagerPlugin*

Related Topics:

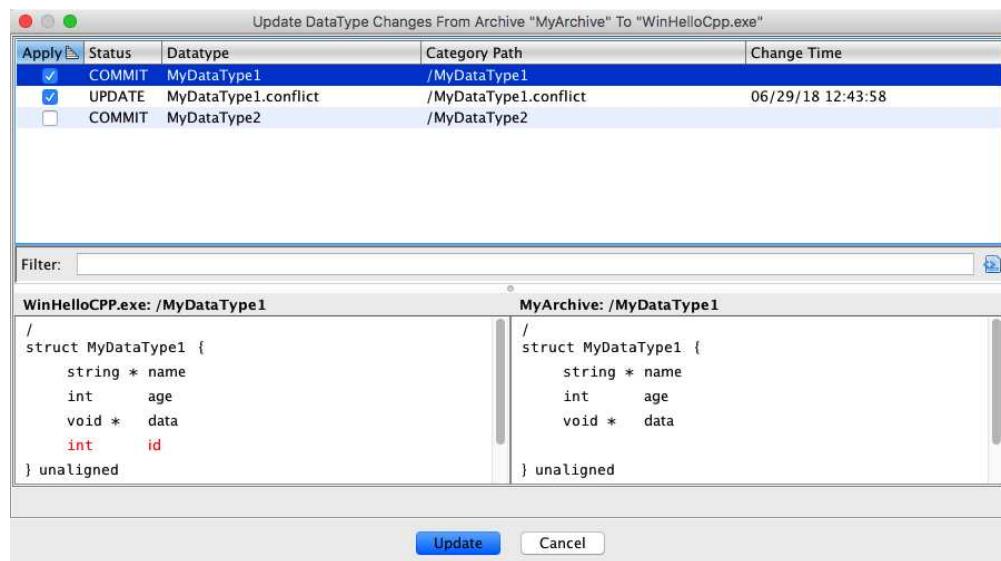
- [Data Type Manager](#)
- [Data Types](#)
- [Apply Data Types](#)
- [Edit Structure](#)

## Data Type Archives

Data type archives are described in the [Data Type Manager](#).

### Updating Archives

You can arrive at the *Update Data Types* dialog from the **Update Datatypes From** action on an archive. The following dialog illustrates updating datatypes from a source archive named **MyArchive** to the program **WinHello.CPP.exe**.



The *UpdateDataTypes* dialog displays a table that lists all the data types that have been changed in the source archive. The table consists of the following columns:

#### Apply

The apply checkbox. Selecting the checkbox will mark the data type to be updated to the version of the data type in the source archive.

#### Status

Indicates the status of the data type.

UPDATE	Data type was changed in the source archive, but not in the client program/archive.
CONFLICT	Data type was changed both in the client program/archive and the source archive. Currently there is no merge, so if you select this data type to update, then all local changes will be discarded.

#### Data Type

Displays the name of the data type as it is known in the local program/archive.

#### Category Path

Displays the full path and name of the data type in the local program/archive.

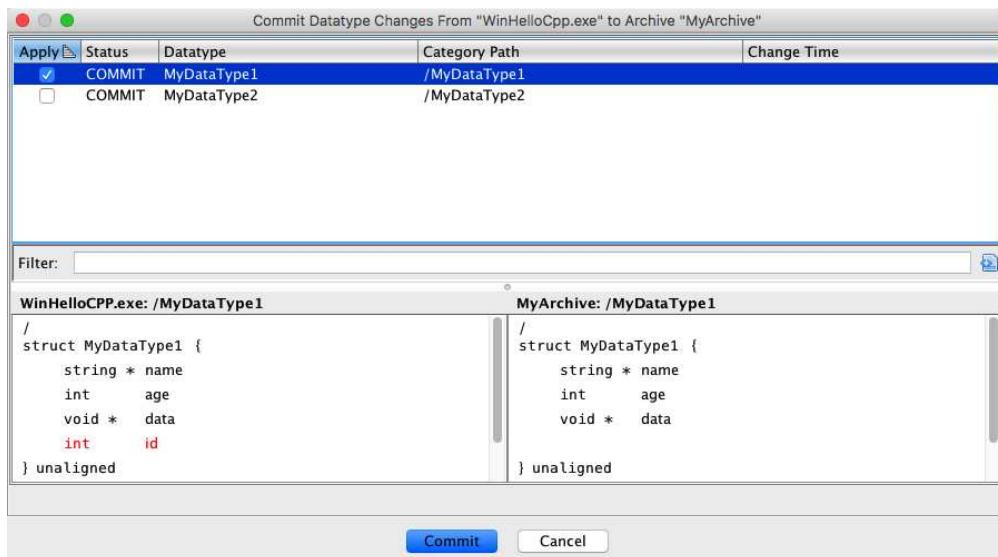
#### Change Time

Displays the time the data type was last changed in the source archive.

Below the table, a data type view panel shows the selected data type as it exists in the client program/archive and source archive. Differences are highlighted in red. Once all data types to be updated have been selected, press the *Update* button to do the update.

### Committing to Archives

You can arrive at the *Commit Data Types* dialog from the **Commit Datatypes To** action on an archive. The following dialog illustrates committing datatypes from a program named **WinHelloCPP.exe** to a source archive named **MyArchive**.



The *CommitDataTypes* dialog displays a table that lists all the data types that have been changed in the local archive. The table consists of the following columns:

#### *Apply*

The apply checkbox. Selecting the checkbox will mark the data type to have its changes committed to the source archive.

#### *Status*

Indicates the status of the data type.

<b>COMMIT</b>	Data type was changed in the local program/archive, but not in the source program.
<b>CONFLICT</b>	Data type was changed both in the client program/archive and the source archive. Currently there is no merge, so if you select this data type to commit, the changes in the source archive will be overwritten.
<b>ORPHAN</b>	The data type originally came from a source archive, but it has been deleted in the source archive. Committing it will "put it back" into the source archive.

#### *Data Type*

Displays the name of the data type as it is known in the local program/archive.

#### *Category Path*

Displays the full path and name of the data type in the local program/archive.

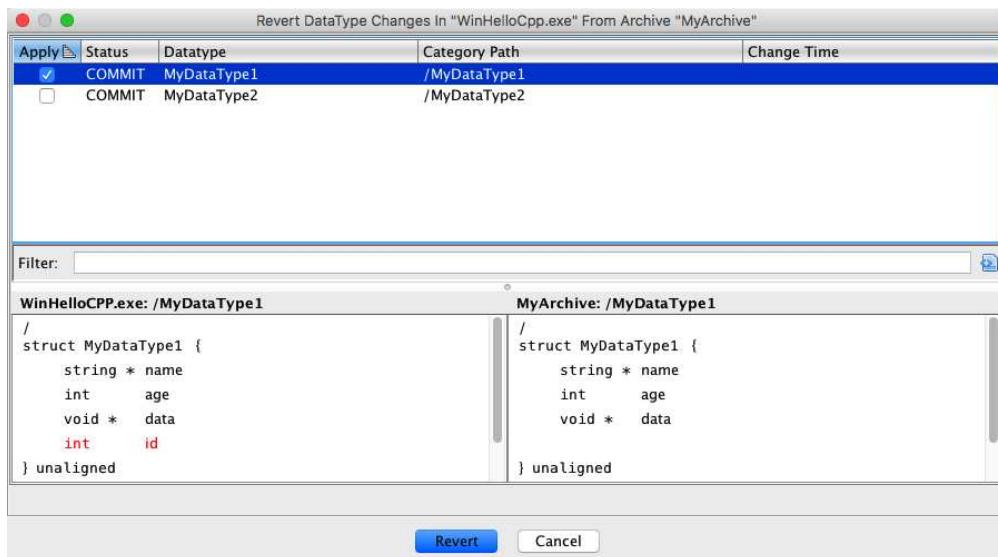
#### *Change Time*

Displays the time the data type was last changed in the local program/archive.

Below the table, a data type view panel shows the selected data type as it exists in the client program/archive and source archive. Differences are highlighted in red. Once all data types to be committed have been selected, press the *Commit* button to do the *commit*.

### Reverting Data Types

You can arrive at the *Revert Data Types* dialog from the **Revert Datatypes From** action on an archive. The following dialog illustrates reverting datatypes from a program named **WinHelloCPP.exe** from a source archive named **MyArchive**.



The *Revert DataTypes* dialog displays a table that lists all the data types that have been changed in the local archive. The table consists of the following columns:

#### Apply

The apply checkbox. Selecting the checkbox will mark the data type to have its changes reverted back to its state in the source archive.

#### Status

Indicates the status of the data type.

COMMIT	Data type was changed in the local program/archive, but not in the source program.
CONFLICT	Data type was changed both in the client program/archive and the source archive. Reverting a conflict is essentially the same as an update.

#### Data Type

Displays the name of the data type as it is known in the local program/archive.

#### Category Path

Displays the full path and name of the data type in the local program/archive.

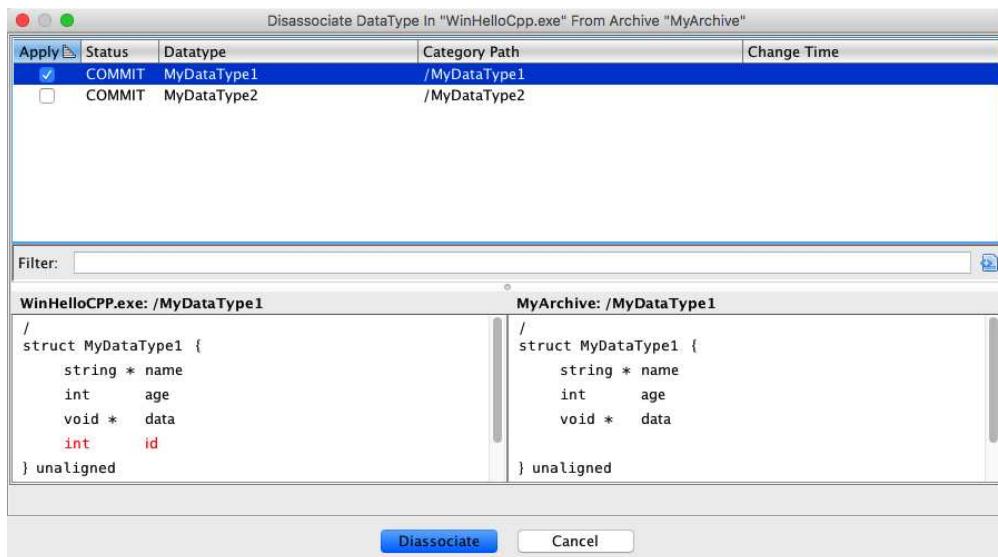
#### Change Time

Displays the time the data type was last changed in the local program/archive.

Below the table, a data type view panel shows the selected data type as it exists in the client program/archive and source archive. Differences are highlighted in red. Once all data types to be reverted have been selected, press the *Revert* button to do the revert.

### Disassociating Data Types

You can arrive at the *Disassociate Data Types* dialog from the **Disassociate Datatypes From** action on an archive. The following dialog illustrates disassociating datatypes in a program named **WinHelloCPP.exe** that originated from a source archive named **MyArchive**.



The *Disassociate DataTypes* dialog displays a table that lists all the data types that are associated with a particular source archive. The table consists of the following columns:

#### Apply

The apply checkbox. Selecting the checkbox will mark the data type to be disassociated from the source archive.

#### Status

Indicates the status of the data type.

UPDATE	Data type was changed in the source archive, but not in the client program/archive.
COMMIT	Data type was changed in the local program/archive, but not in the source program.
CONFLICT	Data type was changed both in the client program/archive and the source archive. Reverting a conflict is essentially the same as an update.
ORPHAN	The data type originally came from a source archive, but it has been deleted in the source archive. Committing it will "put it back" into the source archive.
IN_SYNC	The data type is identical to the version in the source archive.

#### Data Type

Displays the name of the data type as it is known in the local program/archive.

#### Category Path

Displays the full path and name of the data type in the local program/archive.

#### Change Time

Displays the time the data type was last changed in the local program/archive.

Below the table, a data type view panel shows the selected data type as it exists in the client program/archive and source archive. Differences are highlighted in red. Once all data types to be disassociated have been selected, press the Disassociate button to disassociate the selected data types.

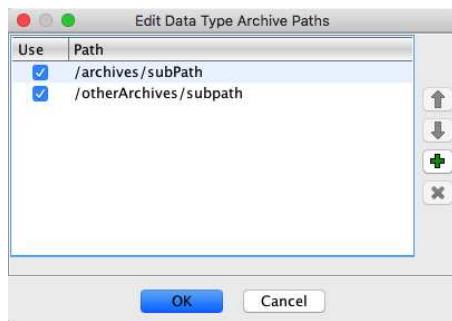
### Refreshing Sync Indicators For Data Types

Datatypes that are associated with a source archive may have a commit, update, or conflict icon indicating they are out of sync with the data type in the source archive, when the data type actually matches the source datatype. This can happen if a data type is changed, but changed to match its source. Invoke the Refresh action to refresh all the sync indicators for that source archive.

To refresh sync indicators for datatypes associated with a particular source archive, right-click on the node that contains the datatypes and select **Refresh Sync Indicators For ➔<Source Archive Name>**.

### Edit Data Type Archive Paths

The *Edit Data Type Archive Paths* dialog is displayed when the **Edit Archive Paths** action is invoked from the drop down menu in the *Manage Data Types* window.



From this dialog, users can add, remove, and reorder the path search order. The checkboxes can be used to temporarily remove a path from being searched. Only those paths that are checked are searched. To add a path use the *add* button. To delete a path, select it, then press the x button. To reorder the paths, select a path and use either the up or down arrows to move the path up or down in the list respectively.



Ghidra currently accepts two special strings in the *Edit Data Type Archive Paths* dialog:

- **\$GHIDRA\_HOME** – The installation directory of Ghidra.
- **\$USER\_HOME** – The user's home directory.

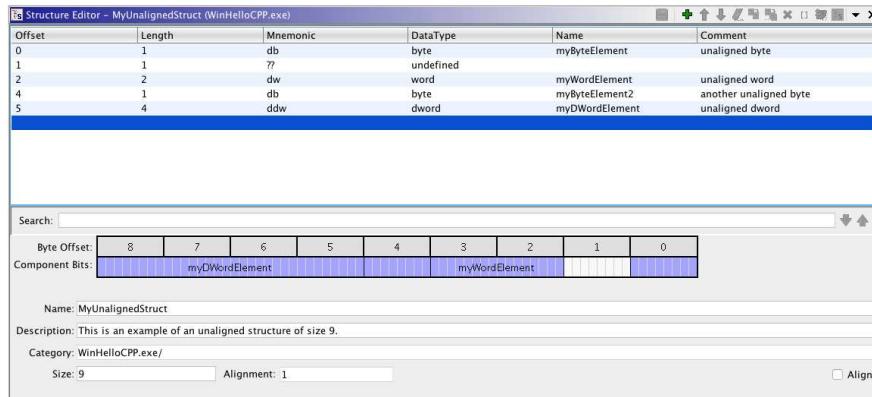
Provided by: *DataTypeManagerPlugin*

Related Topics:

- [Data Type Manager](#)
- [Data Types](#)
- [Apply Data Types](#)
- [Edit Structure](#)

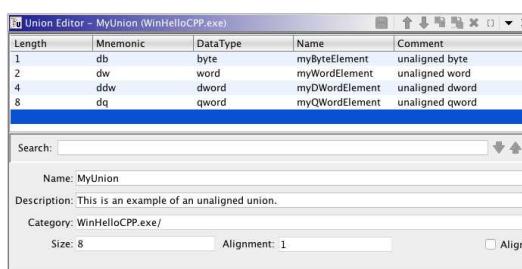
## Structure Data Type Editor

The Structure Editor is used to define the contents of a structure data type. It can be used to create a new structure or to modify existing ones. Structures are user defined data types that contain components. A component is a data type of a particular size and at a specified offset in the structure. The following illustrates the editor for a structure data type.



As shown above, the Structure Editor also includes a bit-level view of the structure layout to improve understanding of bitfield placement when they are present. Note that the byte ordering is reversed for little-endian data organizations to ensure that a bitfield is always rendered as a contiguous bit range. If inadequate space is available for a component label, within the bit view of the component, it will be omitted. However, holding down the Shift-key while using the mouse wheel on a component will zoom the view in and out allowing component labels to appear when space permits. The bit view may also be used to make and reflect single component selections with the table. Within the bit view any *padding* bits not visible within the table view as a component will include a small dot in the center of the displayed bit cell.

The Union Editor is very similar to the Structure Editor, but is used to create or modify a union data type. All the components of a Union are at an offset of 0. The following illustrates the editor for a union data type.



A Structure Editor or Union Editor can be launched from the [Data Type Manager](#), the [CodeBrowser](#), or the [Structure or Union Data Type Editors](#).

The Structure and Union Editors are composed of the following:

- EditActions:** The icon buttons at the top of the editor are used to modify this structure. Each button has a different [editaction](#) associated with it. These allow the user to apply changes, insert undefined bytes, reorganize the current components, duplicate components, clear components (changes them to Undefined bytes), delete components, create array components, and unpackage a structure or array component changing it into its component parts.
- PullDown Menu:** The located on the title bar of the editor provides additional editor actions. This menu shows the category path of a component's data type, edit a component's data type, edit the fields of a component, apply a cycle group or favorite.
- ComponentTable:** The upper portion of the editor contains a table with the structure's or union's components. Each component (or row) consists of its offset, length, mnemonic, data type, field name, and comment. See the [ComponentFields](#) section for more about these fields. The data type, field name and comment are editable fields. The data type's category can be determined by [showing the datatype category](#). Components can be [added](#), [inserted](#) or [replaced](#) in the table using [Drag and Drop](#) or by applying a [Equivalent](#) data type. The data type for a component can also be changed by [cycling](#) the data type.
- StructureInformationArea:** This is the area below the component table with the name, description, category, size and alignment of the structure or union. The structure or union being edited can be [renamed](#) from here. Its [description](#) can be specified here. Also, it can be changed between [unaligned](#) and [aligned](#).
- Immediately below the structure information area is a status line where status messages will appear.

### Applying Changes

Select the Apply Changes icon in the toolbar to apply the changes from the editor back to the program or archive.

If editor changes to a structure or union are applied and it is assigned to data in the program, all data items with the structure or union as the data type now have the new data type. In other words, the size or composition of those data items in the program will have changed due to the apply.

### Closing the Editor

Select the Close dockable component icon in the toolbar to exit from the editor. If you have unsaved changes to your data type, a dialog will ask if you want to save the changes.

### Searching in the Editor

To search for text in any column on any row in the editor, enter the text into the **Search** field and press return to search forward from the currently selected table row. Alternatively, the down and up arrows next to the search field can be used to search forwards and backwards respectively. Searches are not case sensitive.

### Changing the Name

To change the name of the structure or union being edited, edit the **Name** field in the bottom of the editor. If the name is valid, the structure's or union's name will change when the edits are applied by pressing the editor's Apply Changes icon in the toolbar.

### Entering a Description

To change the description of the structure or union being edited, edit the **Description** field in the bottom of the editor. The structure's or union's description will take effect when the edits are applied by pressing the editor's Apply Changes icon in the toolbar.

### Changing the Size

To change the size of the structure being edited, edit the **Size** field in the bottom of the editor and press the **Enter** key to apply the new size to the structure. If you are reducing the size of the structure, you will be prompted to determine that you really want to truncate the structure.

### Alignment

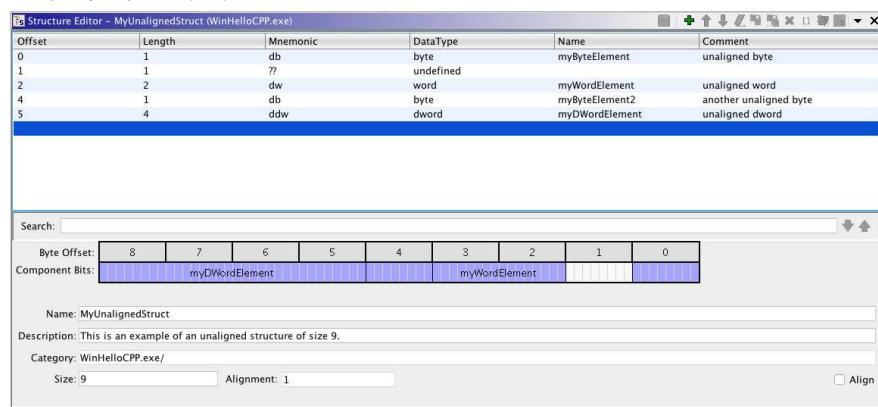
At the bottom of the editor is an **Alignment** field, which indicates how the structure or union being edited will be aligned within other data types. All unaligned structures and unions will have an alignment of 1 (byte aligned) and will therefore not be aligned within other data types. For aligned structures and unions the alignment value is determined based on the alignments of the individual components they contain, whether they are [packed internally](#) and whether they have a [minimum alignment](#) specified.

### Unaligned vs. Aligned

In the lower right corner of the editor is an **Align** check box. When the box is checked the structure or union is aligned. Click the checkbox to change between aligned and unaligned. When a structure or union is aligned, additional choices will appear that allow minimum alignment and packing to be specified.

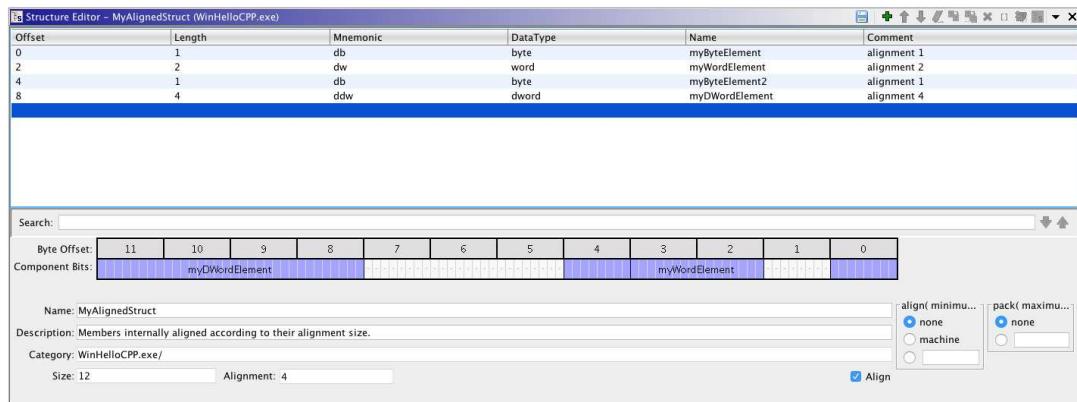
### UnalignedStructures

When a structure is unaligned, with the exception of bitfields, each component immediately follows the one before it. In other words no automatic alignment or padding occurs. Unaligned structures can contain Unaligned structures should be used to position components with known data types at specific offsets within the structure. Therefore the structure editor tries to prevent defined components (those other than Undefined bytes) from accidentally being moved to a different offset when performing operations like drag and drop, although undefined bytes may be consumed.



#### AlignedStructures

An aligned structure is defined similar to a structure in a C header file. The data types are specified for each of the components, but their offsets will be automatically adjusted to the correct alignment based on the data type and position in the structure. A default "undefined" byte cannot be added to an aligned structure, although an "undefined" component can be and is treated like any other fixed-length datatype. The overall size of the structure is determined by the components it contains and the specified pack value. The alignment and packing behavior is determined by the effective data organization as defined by each compiler specification (data type archives utilize a default data organization). When you select the Align checkbox, the GUI displays buttons to allow an align attribute, `align( minimum )`, to be specified. It also allows a pack value, `pack( maximum )`, to be specified. The following image shows these GUI components for aligning the data type being edited.



#### `align( minimum )`

This indicates the minimum alignment to be used when aligning this data type inside another data type. Setting this to a value other than none forces this structure to be aligned within other structures at an offset that is a multiple of the specified value. Specifying a minimum alignment also causes the end of the structure to be padded so its size is a multiple of the minimum alignment.

- **none** – Sets this data type to have no specified minimum alignment when aligning this data type inside another data type. Align this data type based only on its components and their minimum alignments and packing.
- **machine** – Sets this data type to have a minimum alignment that is a multiple of the machine alignment when aligning this data type inside another data type. In this case machine alignment means the largest alignment which is ever used for any data type on the program's intended machine.
- **by value** – The bottom button with a text field next to it allows you to specify a minimum alignment value. This sets this data type to have a minimum alignment that is a multiple of the specified value when aligning this data type inside another data type.

The equivalent of having `noCode align` attribute on the structure or union is to choose **none**. The equivalent for a C code attribute of `align()` without a value is to choose **machine** alignment. The equivalent of `align(4)` is to specify a value of 4.

#### `pack( maximum )`

This indicates a packing value to be used when aligning the components within this data type. Setting this to a value other than none forces each component of this structure to be packed so that its maximum alignment within the structure is no more than the pack value. However, a component will not be packed if the data type or one of its subcomponents data types has a minimum alignment value specified that exceeds the pack value. Packing also affects the padding at the end of this structure so that the overall size of the structure is the smallest possible multiple of the pack value. If the minimum alignment is greater than the pack value, the end of the structure or union is padded based on the minimum alignment instead.

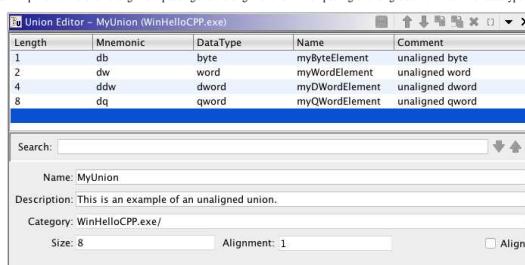
- **none** – The components within this structure should align themselves in the default way for the compiler. The components are not being packed with a reduced alignment.
- **by value** – The bottom button with a text field next to it allows you to specify a pack value. The specified value indicates the maximum alignment to use when packing any component.

The placement of bitfields may be influenced based upon the specified pack value.

The equivalent of having `#pragma pack` attribute on the structure or union is to choose **none**. The equivalent for a C code attribute of `#pragma pack(1)` without a value is to specify a **pack** value of 1. The equivalent of `#pragma pack(4)` is to specify a **pack** value of 4.

#### UnalignedUnions

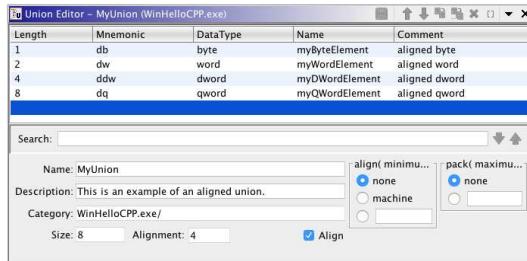
When a union is unaligned, the union is the size of its largest component. There is no alignment padding and the alignment is 1 when putting an unaligned union into other data types.



#### AlignedUnions

In an aligned union the overall size is at least the size of the largest component, but will be padded based on the alignment and packing. The **minimum alignment** is specified just as within a structure and affects the alignment in the same way. The

**pack** value is specified in the same manner as within a structure, but only affects the padding in the overall size of the union. All elements in a union have an offset of zero, so the pack value doesn't affect the component offsets. If both a minimum alignment and pack value are specified, the minimum alignment will override the pack value if it is larger.



#### Bitfield Component

A structure and union may define bitfield components which attempt to model bitfield definitions found within C/C++. Unlike other byte-oriented components, bitfield components have the following characteristics:

- A bitfield datatype may not exist anywhere other than within a structure or union.
- A bitfield datatype may not be selected via the datatype chooser or tree since it requires the specification of additional attributes (e.g., bit-size, bit-offset). The bit-size is generally appended to the base datatype for datatype specification and presentation purposes (e.g., `char:1`).
- A zero-length bitfield may be defined within a byte but its' precise bit position is controlled by endianness alone. A zero-length bitfield has no effect within an unaligned structure and is intended for use within aligned structures where it may impart alignment requirements based upon compiler conventions.
- Inserting a bitfield within an unaligned structure may cause component shifts based upon the specified offset and allocation unit byte size when a placement conflict occurs.
- The start/end byte offsets may be shared with adjacent bitfield components.
- Unoccupied bits within a partially occupied byte are not represented by any component (similar to padding bytes within aligned structures).
- As separate **BitfieldEditor**, for use with unaligned structures only, must be used to precisely place a bitfield component. Adding a bitfield component via the structure table view via datatype text entry (e.g., `char:1`) provides only rough placement for unaligned structures since additional bytes will be introduced. The BitField Editor may be displayed using the the Add Bitfield and Edit Bitfield popup menu actions on a selected structure component. The datatype text entry approach must be used for all unions and aligned structures.

While packing of bitfields within aligned structures is controlled by the compiler specification (e.g., data organization), bit-packing order is currently fixed based upon endianness. Little-endian packs starting with bit-0 (lsb) while big-endian packs starting with bit-7 (msb).

The use of bitfield components is not currently reflected in decompiler results or assembly markup.

#### Flexible Array Component

A structure may be defined with a trailing flexible array component which corresponds to an unsized array (e.g., `char[0]`). Such a component, if it exists, is always displayed as the last row within the structure editor with the open edit row for additional components located directly above it. The **DataType** column for a flexible array always shows the base type of the unsized array which can be modified similar to other components. A flexible array is added to the end of a structure by adding a last row specified by the base **DataType** (e.g. `char`) then invoking the array action and specifying an element count of 0. Attempting to specify an element count of 0 is only permitted on the last component row when a flexible array component does not already exist. The presence of a flexible array component does not affect the size of a structure and will not appear within the structure when applied to Data within memory as it corresponds to the memory location which immediately follows the end of the structure.

The use of flexible array components is not currently reflected in decompiler results or listing reference markup. Its primary purpose is to reflect the C/C++ source definition of a structure with correct alignment and structure sizing.

While C/C++ support flexible arrays anywhere within a structure, Ghidra only supports the case where it is the last structure component.

Offset	Length	Mnemonic	DataType	Name	Comment
0	1	db	byte	myByteElement	alignment 1
2	2	dw	word	myWordElement	alignment 2
4	1	byte:1	byte:1	myBitField1	alignment 1
4	1	byte:2	byte:2	myBitField2	alignment 1
4	1	byte:3	byte:3	myBitField3	alignment 1
5	1	db	byte	myByteElement2	alignment 1
8	4	ddw	dword	myDWordElement	alignment 4
12	0	char[0]	char	flex	unsized flexible array

#### Edit Actions

The edit actions are available both from the icon buttons in the editor and from the popup menu for the component table. To display the popup menu, right mouse click on the component table. There are also short-cut keys associated with each of the edit actions.

##### **Insert Undefined Byte**

Undefined Bytes can only be inserted into a structure that is not aligned. A single Undefined byte is inserted before the current selection by clicking the **Insert Undefined Byte** button.

##### **Move Up**

A contiguous group of components can be moved up or down in the structure. Select a block of one or more components. Each time the **Move Up** button is pressed, the group of components is moved up one row. In other words, the component that was immediately above the group is now below it. This can be done until the group of selected components is at the beginning of the structure (offset 0).

##### **Move Down**

Select a block of one or more components. Press the **Move Down** button to move the group of selected components down one row. The component that immediately followed them is now before them. The selected components can be moved down until they are at the end of the structure.

##### **Duplicate Component**

To duplicate a component within a structure or union:

1. Select the component.
2. Press the **Duplicate Component** button.
3. A single copy of the component is created immediately following the selected one.

If the structure is not aligned, then there must be enough Undefined bytes following the component to accommodate the duplicate. In an unaligned structure, Undefined bytes get replaced by the new copy of the component.

##### **Duplicate Multiple of Component...**

To create multiple copies of a component:

1. Select the component.
2. Press the **Duplicate Multiple of Component...** button.
3. The number of duplicates dialog appears. In an unaligned structure, you can only indicate as many duplicates as will fit in place of Undefined bytes.



4. Enter a valid number of duplicates and press the **Enter** key or the **OK** button.
5. The specified number of copies of the component are created immediately after the selected one.

#### **DeleteComponent(s)**

One or more components can be deleted from a structure or union. Deleting components in a structure will cause the components that come after those being deleted to shift to new offsets.

To delete components:

1. Select one or more components to delete.
2. Press the **DeleteComponent(s)** button.
3. The components are removed from the structure or union.

#### **ClearComponent(s)**

Clearing a component changes it into Undefined byte components that take up the same amount of space as the component being cleared. Components can be cleared in unaligned structures, but not in unions or aligned structures. This is because the resulting Undefined bytes are not valid components in a union or aligned structure.

To clear components in a structure:

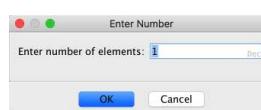
1. Select one or more components in the table.
2. Press the **ClearComponent(s)** button.
3. Each selected component is replaced by Undefined Bytes. The number of Undefined Bytes will be equal to the length of the component being cleared.



#### **Create Array**

To create an array from a single selected component:

1. Select a component in the table.
2. Press the **CreateArray** button.
3. A dialog pops up to request the number of elements in the array.



4. Specify the number of elements. For an unaligned structure, the maximum size of the array is limited by the number of Undefined bytes following the selected component.
5. Press the **OK** button.
6. The selected component becomes an array of that same data type with the specified dimensions.



To create an array from a selection of multiple components in a structure:

1. Select multiple contiguous components.
2. Press the **CreateArray** button.
3. An array is created. The data type of the first component in the selection is used as the data type of the array. The array's dimension is determined by how many of that data type will fit in the space that was occupied by the selection. The size of the array will be the array that can fit in the selected number of bytes. Any left over (unused) bytes at the end of the selection will become Undefined bytes.

#### **CreateFlexibleArray**

To create an unsized flexible array as the last structure component (assuming one is not already defined):

1. Add a component whose *DataType* corresponds to the base type of the unsized array.
2. Select this last structure component in the table.
3. Press the **CreateArray** button or the **T** action key binding.
4. A dialog pops up to request the number of elements in the array. Specify 0 as the number of elements in the popup dialog and click OK.

#### **Unpack Component**

Unpacking takes a single component and replaces it with its own component parts. When editing a structure, array and structure components can be unpackaged by selecting that component and clicking the **Unpack Component** button. For example, the array component Word[4] would become four Word components. If the structure struct\_1 was composed of a Float and a DWord, then unpackaging it would replace the struct\_1 component with a Float component and a DWord component.

#### **CreateStructureFromSelection**

Creating a structure from a selection takes a contiguous selection of components in the structure editor and creates a new structure from those components. It then replaces the selection with a single component whose data type is the new structure. The components that were in the selection will become the components that make up the new structure. The user does this by selecting the components in the editor and clicking the **CreateStructureFromSelection** button. The user is then prompted for the name of the new structure. Once a unique name is specified the new structure is created and a component containing it replaces the selected components.

#### **Bitfield Actions (unaligned structures)**

The following bitfield actions are available when modifying unaligned structures only and are available via the popup menu based upon a selected component or table row. When working with unions and aligned structures, bitfields may only be specified via datatype text entry specification within the table view (e.g., char:1).

##### **AddBitfield**

With a structure row selected in the vicinity of the desire bitfield placement, the popup menu action (right mouse-click) **AddBitfield** may be selected to launch the [BitfieldEditor](#) for a new bitfield.



##### **EditBitfield**

With a defined bitfield component row selected, the popup menu action (right mouse-click) **EditBitfield** may be selected to launch the [BitfieldEditor](#).

#### **Component Fields**

Each row displays information about a component in this structure or union. The *DataType*, *Name*, and *Comment* fields are editable.

The following summarizes the information about each field for a component.

**Offset** – The byte offset of this component from the beginning of the structure. For unions the byte offset of all components is zero, and therefore this field isn't shown.

**Length** – The length of this component in bytes.

**Mnemonic** – The mnemonic (brief identifier) for this component's data type.

**DataType** – The data type of this component. This field is editable.

**Name** – The field name associated with this component in the structure or union. When specified, the field names must be unique for the components in a structure or union. This field is editable, except on components that are Undefined bytes.

**Comment** – A comment associated with this component. This field is editable, except on components that are Undefined bytes.



To rearrange the order of the component fields position the cursor over the header for the table column. Left mouse click and drag the column left or right to its new position.

#### [Editing Component Fields](#)

The *DataType*, *Name* and *Comment* fields are editable. However, the *Name* and *Comment* are not editable when the component's *DataType* is an Undefined byte.

To place an editable field into edit mode:

- **Doubleclick** on an editable field.
- or
- Select a single component row and press **F2** to begin edit mode. This puts the first editable field in that component row into edit mode. **Tab** will then move to the next editable field.

#### [Applying an Edit](#)

In the editable field, pressing **Enter** applies the value to the field if it is valid and ends the edit session. If the field's value is invalid, a message is written to the editor's status line and the field remains in edit mode.

#### [Canceling an Edit](#)

In the editable field, pressing **Escape** cancels the edit session without applying the changes to that field.

#### [Editing More Than One Component Field](#)

You can move directly from editing the component's name or comment field to editing another by pressing **Tab**, **Shift-Tab**, **UpArrow** or **DownArrow**. The key press only moves the edit session if the current field edit can be applied. Otherwise, an error is displayed in the status line.

##### **Tab**

Pressing **Tab** applies the current edit and moves to the next editable field in the table. If the current field is the last editable one in this component then it moves to the first editable field in the next component. This key moves the edit session left to right and top to bottom in the table.

##### **Shift-Tab**

Pressing **Shift-Tab** applies the current edit and moves to the previous editable field in the table. If the current field is the first editable one in this component then it moves to the last editable field in the previous component. This key moves the edit session right to left and bottom to top in the table.

##### **Up-Arrow**

Pressing the **UpArrow** key applies the current edit and moves to the same field of the previous component in the table if that field is editable. This key moves the edit session bottom to top in the table.

##### **Down-Arrow**

Pressing the **DownArrow** key applies the current edit and moves to the same field of the next component in the table if that field is editable. This key moves the edit session top to bottom in the table.

#### [Editing the DataTypeField](#)

This can be any data type that is available in the data type manager. To edit the data type double-click the data type cell. This will show the [Data Type Selection Dialog](#), which allows you to easily enter a data type. It can also be Undefined, a pointer to a data type, or an array.

##### **Basic Data Type**

This can be a built-in data type (Byte, Word, etc.), a structure, a union, or a typedef.

For example, Word.

##### **Pointer Data Type**

This can be the basic Pointer data type or a pointer to a data type. A pointer to a data type is indicated by following the data type with an \*.

For example, Word \* is a pointer to a Word.

##### **Array**

This can be a multidimensional array of any data type.

For example, DWord[2][4] is an array with 2 elements, where each element is an array with 4 elements that are DWords.

##### **ArrayOfPointers**

Arrays of pointers can also be specified.

For example, Float\*[5] is an array with five elements where each element is a pointer to a Float.

##### **Effect of Changing a Component's Size**

An unaligned union's size will always be the size of its largest component. If you change a data type for a component and the component size changes, the union size will change if necessary. An aligned union is padded to make its size a multiple of the union's alignment.

How a structure is affected by changing a component's data type depends on whether the structure size is aligned or unaligned.

**Unaligned**—If the structure is *unaligned* then the new component must be less than or equal to the original component's size plus any Undefined byte components that immediately follow it in the structure. In an unaligned structure, decreasing the component size will create Undefined byte components following it to maintain the structure size. Increasing the component size replaces Undefined bytes immediately following the component. The last component of a structure can always be changed which can cause the structure to grow larger.

**Aligned**—If the structure is *aligned* the component can change size, which affects the structure's overall size and the alignment of individual components that follow it.

#### [Editing the NameField](#)

When specified, a field name must be unique for the components in a structure or union. It cannot contain blanks.

#### [Editing the CommentField](#)

The comment can be any Ascii text.

#### **Pointers**

Pressing the **p** key invokes the pointer action on a component. This will generally create a default pointer unless the existing data is already a pointer in which case that pointer will be wrapped with an additional pointer (e.g., int \* would become int \*\*). This action will always apply a default sized pointer. Otherwise you can drag one of the other pointer types from the *Data Type Manager* window. With existing pointer data, the base type of that pointer may be changed simply by applying another type onto the pointer (e.g., applying byte to default pointer becomes db \*). If you do not want this pointer stacking behavior to happen, it is best to clear the code unit(s) before applying a data type via drag-n-drop or key-binding actions.

#### **Cycling a Component Data Type**

Some data types are part of a cycle group. A cycle group is a collection of data types that are similar and are commonly associated with one another. Cycling a data type facilitates changing a component from one data type to the next data type in the same group. Each cycle group has a short-cut key associated with the group. Pressing the short-cut key cycles from the current data type to the next one in the group. For example, the **b** key is associated with the Byte cycle group. This group is Byte, Word, DWord, and QWord.



A single row of the component table must be selected to cycle a data type.

#### **Union or AlignedStructure**

The first data type in a cycle group can be added to the end of a structure as a new component.

To add a data type using a cycle group key:

1. Select the last row (the blank row) of the component table.
2. Press the short-cut key for the desired cycle group. For example, the **b** key will add a Byte; the **f** key will add a Float; the **apostrophe** key will add an Ascii.



Cycle groups can also be applied from the component table's popup menu under **Cycle**.

To cycle a component's data type to another one in the same group:

1. Select the component in the table.
2. Press the cycle group key until the data type has cycled to the desired data type. For example, if the current data type is a Byte, pressing the **b** key twice would change it to DWord.



When the current component data type is not in the cycle group of the desired data type, pressing the cycle group key of the desired data type will change the component to the first data type in the desired cycle group. The data type can then be cycled to any other data type in the group.

**UnalignedStructure**

Cycling is implemented similar to how it is implemented in an aligned structure. The only exception is that the user can only cycle to data types that will fit within the data boundary of the current component. If Undefined bytes follow the selected component, the Undefined bytes can be replaced by cycling the data type to a larger sized data type. Likewise, cycling to a smaller data type will add Undefined bytes after the component being cycled. However, the last component is not restricted to a particular size.

**Drag and Drop Basics**

Data types can be dragged from the [Data Type Manager](#) to the component table in the Structure Editor. A regular drag and drop results in the component in the editor being replaced by the one being dragged. Holding the Ctrl key during the drag and drop causes the dragged data type to be inserted instead of being replaced.

**KnownProblem:** Holding the Ctrl key to perform an insert of a data type does not currently work on a Mac. On a Mac try using the Alt key to insert rather than the Ctrl key. The "+" should then appear at the drop site.

 When a data type is dropped on a pointer component, the component becomes a pointer of that data type instead of simply being replaced by that data type. For example, dropping a Byte on a Pointer results in a Byte\*, which is a pointer to a Byte. Drag and Drop is discussed further in [Adding a Data Type](#), [Inserting a Data Type](#), and [Replacing a Data Type](#).

**Favorites Basics**

Favorite data types are defined from the Data Type Manager dialog. In the Data Editor, favorites are available through the popup menu in the component table. Applying a favorite data type is similar to dropping a data type. Favorites can be used to insert a component of that data type or to replace a component's data type. Favorites can only be applied to a contiguous selection. Therefore, individual favorites are only enabled when they can be applied (i.e. they fit at the selection and the selection is contiguous).

 When a favorite data type is applied to a pointer component, the component becomes a pointer of that data type instead of simply being replaced by that data type.

 If you right mouse click on a component where there is no selection, the selection becomes a single component selection containing that component and the popup menu will appear.

Favorites are discussed further in [Adding a Data Type](#), and [Replacing a Data Type](#).

**Adding a Data Type**

A data type can be added as a component by replacing Undefined bytes. When editing an unaligned structure, there must be enough Undefined bytes for the data type to replace. A data type can also be added to the end of a structure or union.

**Drag and Drop**

Drag a data type from the data type manager to the empty row at the end of the component table and drop it. The data type is added to the end of the structure in the editor.

If the data type can be various sizes, the user is prompted for the desired size. The following illustrates the dialog due to the drag and drop of a string.



Simply enter the number of bytes desired for the data type and press the Enter key or click the OK button.

 For information about drag and drop with pointers, see [Drag and Drop Basics](#).

**Favorites**

Right mouse click on the empty row at the end of the table and pull right to see the Favorites. Select the favorite from the popup and it is added as the last component.

 For information about Pointers as a Favorite, see [FavoritesBasics](#).

**Inserting a Data Type**

In an aligned structure or a union, a data type can be inserted as a new component.

**Drag and Drop**

1. While holding down the Ctrl key, drag the data type you want to insert from the data type manager. As you drag it over the components in the structure editor, the drag icon will have a "+" on it indicating insert mode.
2. Release the mouse button when you are on the desired component and the dragged data type is inserted before it.
3. **KnownProblem:** Holding the Ctrl key to perform an insert of a data type does not currently work on a Mac. On a Mac try using the Alt key to insert rather than the Ctrl key. The "+" should then appear at the drop site.

 For information about drag and drop with pointers, see [Drag and Drop Basics](#).

**Replacing a Data Type**

A component can have its data type replaced with a different data type. If an unaligned structure is being edited then Undefined bytes are created or consumed as necessary to maintain the position of other components within the structure. For unions and aligned structures, the data type simply changes for the component and the overall size is adjusted accordingly.

**Drag and Drop****Single Component Selected**

Drag a data type to a single selected component in the editor or to a non-selected component in the editor. If the mouse pointer is a 'circle with a slash' then the data type cannot be dropped to replace the component. This is probably because the data type being dropped won't fit in the structure in place of the original component. If editing a union or an aligned structure the data type should always fit and the drop is allowed. If editing an unaligned structure, the component is replaced only if the new component will fit. (see [Effect of Changing a Component's Size](#))

**Contiguous Selection of Multiple Components**

Drag a data type to a block of selected components. Whether the structure is aligned or unaligned doesn't matter when dropping a data type on a block of selected components. This is because the new component (s) will occupy the same space as the currently selected components.

In a **union**, all selected components will be replaced with a single component containing the data type dropped.

In a **structure**, as many components of the dropped data type will fit in the selection are created to replace the selection. In an unaligned structure any left over bytes in the selection will become Undefined bytes.

 For information about drag and drop with pointers, see [Drag and Drop Basics](#).

**Favorites****Single Component Selected**

 If you right mouse click on a component where there is no selection, the selection becomes a single component selection containing that component and the popup menu will appear.

To replace a component's data type with a favorite data type, select it in the table, right mouse click and pick the favorite. Only favorites that will fit in place of the component will be enabled. (see [Effect of Changing a Component's Size](#))

**Contiguous Selection of Multiple Components**

To replace a contiguous selection with a favorite data type, select it in the table, right mouse click and pick the favorite.

In a **union**, all selected components will be replaced with a single component containing the favorite data type.

In a **structure**, only favorites that will fit in place of the selection will be enabled. Just like with drag and drop, whether the structure is aligned or unaligned doesn't matter. The selection becomes as many of the data type as will fit and left over bytes become Undefined bytes for an unaligned structure.

 For information about Pointers as a Favorite, see [FavoritesBasics](#).

### Showing a Component's Data Type Category

Every component has a data type: Byte, Word, Float, etc.. The category is where that data type is located. Since you can have two data types with the same name in different locations (categories), the editor provides a way to see the category for any component's data type.

To see the category for a component:

1. Select a single component.
2. Right mouse click on the component in the table.
3. From the popup menu, select **Show Component Path**.
4. The category information appears in the editor's status line at the bottom of the dialog.

### Editing a Component's Data Type

Another Data Type Editor can be brought up from within the Structure Data Type Editor for any component that is an editable data type, such as Structure, Union, or Enum. In addition, another Data Type Editor can be brought up for any component whose base data type is an editable data type. For example, a `typedef` on a Structure would allow you to edit the Structure. Likewise, you can edit a Structure wherever the component is a Pointer to the Structure.

To display another editor for a component that has an editable data type:

1. Select the component with a data type that is a structure, union or enum or that has one of them as its base data type.
2. Right mouse click on the component in the table. Select **Edit Component...** from the popup.
3. The editable data type for the selected component is displayed in a new data type editor.

### Showing Numbers in Hexadecimal

The Component Table contains numeric fields such as the component's offset and length. The structure information area also shows the overall size and alignment of the structure. By default numbers are initially shown in decimal in the Structure Editor. There is a **tooloption**, **Show Numbers In Hex**, that lets you override the default and set whether these numbers should be displayed in hexadecimal or decimal. Changing this only affects the current editor where it is changed.

There is also a toolbar menu item in the editor that allows you to immediately change whether the numbers are displayed in hexadecimal or decimal. Changing this only affects the current editor where it is changed.

The Union Editor does not display offsets since they are always zero.

To switch between decimal and hexadecimal display of numeric values in the current editor:

1. Right mouse click on the table.
2. A check mark next to **Show Numbers in Hexadecimal** indicates that numbers are currently displayed in hex. If there is no checkmark, the numbers are currently shown in decimal.
3. From the popup menu, select **Show Numbers in Hexadecimal**.
4. The numbers being displayed will change from decimal to hexadecimal or vice versa. The check mark next to **Show Numbers in Hexadecimal** indicates whether the numbers are currently displayed in hexadecimal.

### Structure and Union Editor Tool Options

The Structure and Union Editors add options to the tool. To view or edit the option settings:

- From the tool's menu select **Edit ➔ Tool Options...** which displays the **Tool Options Dialog**.
- Open the **EditorTree** node.
- Click on either the **StructureEditor** or **UnionEditor** tree node to view its options.

The **StructureEditor** and **UnionEditor** tabs contains the following options:

Editor	Option	Functionality
Structure Editor	Show Numbers In Hex	If selected, the component offsets and lengths as well as the overall size and alignment of the structure will be displayed in hexadecimal when a new Structure Editor is initially displayed. Otherwise, the lengths are initially in decimal.
Union Editor	Show Numbers In Hex	If selected, the component lengths as well as the overall size and alignment of the union will be displayed in hexadecimal when a new Union Editor is initially displayed. Otherwise, the lengths are initially in decimal.

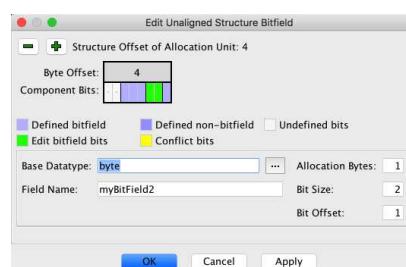
To select an option simply click on the check box.

### Bitfield Editor (unaligned structures only)

The Bitfield Editor is used by the Structure Editor when adding or modifying bitfield components within unaligned structures to facilitate precise placement at the bit level. The Bitfield Editor is not supported for unions and aligned structures since automated packing is performed (i.e., bitfields are specified via datatype text entry within the structure/union table view). While editing an unaligned structure, the Structure Editor popup menu actions **AddBitfield** and **EditBitfield** are used to launch the Bitfield Editor.

The Bitfield Editor includes a visual depiction of the storage allocation bytes and associated bits (8-bits per byte, with a left-to-right / msb-to-lsb sequence of 7,0). The displayed byte ordering as conveyed by the **Byte Offset** header will differ for big-endian vs. little-endian. This is done to ensure that bitfields which span byte boundaries will always visually appear as a consecutive range of bits. The **Component Bits** display provides both a popup menu and mouse assisted bitfield manipulations. Component labels are included within each component when space permits. The color legend indicates the bit color scheme reflecting Defined bitfields, Defined non-bitfields, Edit bitfield bits, Conflict bits and Undefined bits. A dot in the center of an Undefined bit indicates a padding bit not included within any defined component. Conflict bits correspond to the current edited bitfield where bits are specified in conflict with an existing component.

While the Bitfield Editor is displayed local popup menu actions are provided which can facilitate additional component manipulations (e.g., **AddBitfield**, **EditBitfield**, **Delete**). These actions relate to the component at the current mouse cursor location. Invoking either the **AddBitfield** or **EditBitfield** local popup menu actions will immediately cancel the current bitfield operation if one is active.



A component zoom feature is provided which can allow the user to increase the visual bit size allowing for component labels to be shown as size permits. While with the mouse cursor is over a component, use mouse wheel while the Shift key is depressed.

#### BitfieldParameters

**Structure Offset of Allocation Unit** –controls the minimum byte offset of the storage allocation unit represented by the bitfield edit view. This offset is controlled via the and buttons at the top of the bitfield editor.

**AllocationBytes** –controls the size of the storage allocation unit to be utilized in the event a bit conflict exists and the user chooses to *MoveConflicts*. This numeric entry can be directly entered within the range of 1 through 16 bytes. The mouse wheel may also be used while the cursor is over this entry.

**BitSize** –specifies the size of the bitfield in bits. This size may not exceed the size of the specified Base Datatype. This numeric entry can be directly entered or via the mouse wheel while the cursor is over this entry field or the rendered bitfield.

**BitOffset** –specifies the offset of the rightmost bit of the bitfield within the displayed allocation unit. This numeric entry can be directly entered or via the mouse wheel while the cursor is over this entry field.

**Base Datatype** –(required) specifies the numeric datatype associated with the bitfield. Valid datatypes include primitive integer types (e.g., char, bool, int, etc.), enum types, and typedef types of integer or enum types. This input allows direct text input with auto-complete assistance or via full access to a datatype tree chooser by clicking the .

**FieldName** –(optional) specifies the structure component name to be assigned to the bitfield. This entry utilizes a simple text entry field.

The bitfield offset and size may be fully specified by using the mouse. Clicking and dragging over the visual bit-range where the bitfield should reside will adjust these settings.

Within the Bitfield Editor the bit size may not exceed the size of the Base Datatype based upon the structure's associated compiler convention (i.e., data organization). Since archives use default integer type sizes which may differ from a target program's datatype sizing, the use of fixed-size base datatypes may be preferred. Otherwise, structure edits should be performed within the target program where datatype sizes may be larger. If an existing bitfield size exceeds the size of the base datatype within the associated data organization the "effective" bitfield size will be reduced. As with other components, structure component sizing may change when moving between a datatype archive and target program.

 It is important to note that the retained bitfield storage specification (byte offset, component byte size, bit offset) will utilize the smallest possible values while preserving bitfield positioning within the structure. The allocation unit offset and size conveyed by the editor are for editor use only.

Provided by: *DataTypeManager* Plugin

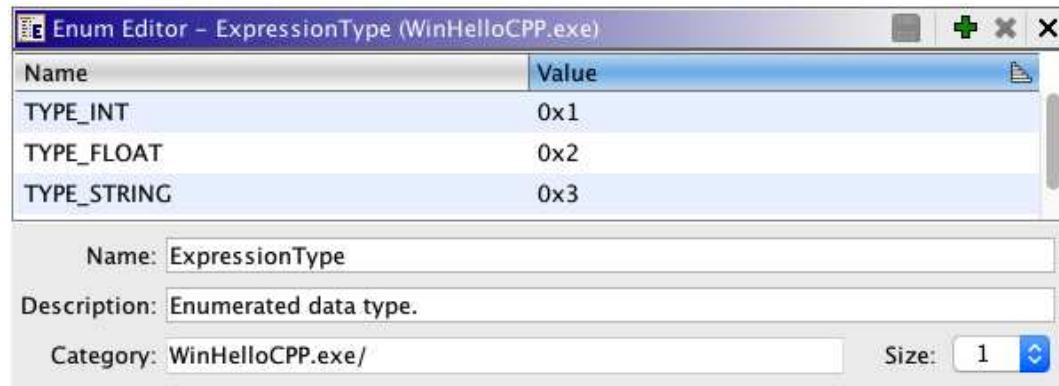
RelatedTopics:

- [Data Type Manager](#)
- [Enum Editor](#)
- [Data Type Selection Dialog](#)

# Enum Data Type Editor

An *Enumeration* data type ("Enum") is a C-style data type that allows the substitution of a value for a more meaningful name. Ghidra provides a special editor for Enum data types.

[Create an Enum data type](#) using the [Data Type Manager](#) display. The editor for an Enum is shown below.



## Enum Editor Fields

- The *Name* column in the table is the name of the enum entry.
- The *Value* column in the table is the value of the enum entry.
- The *Name* field below the table shows the name of the enum. Edit this field to change the name.
- The *Description* field shows a short description for the enum; edit this field to update the description.
- The *Category* field shows where the enum resides which corresponds to the folder you were selecting when creating the enum; the field is not editable, however, you can move it using the Data Type Manager after you have created it if you want to move it.
- The *Size* field shows the number of bytes required when you apply the enum. To edit this field, use the dropdown menu to choose a new size. Note: If you are applying an enum to a data definition and do not see expected results in the decompiler it is probably because the size is incorrect.

When you make any change to the enum, the (Apply) button is enabled.

## Edit an Enum Entry

To edit an entry in the enum table, select an entry; double click in the field that you want to change, OR press **F2**. A cell editor is displayed; enter the value (press the <Enter> key).



While editing, you can use the **Tab** key to navigate to the next cell, the **Shift-Tab** key to navigate backwards, the **Up** key to move editing to the cell above, and the **Down** to move editing to the cell below the currently edited cell.



Names and values must be unique.

## Add an Enum Entry

Create a new enum entry by selecting the (Add) button on the editor's tool bar. An entry with default values is added to the table; the default name is "New Name;" the default value is the next highest value. Double click on the fields to change the entries.

## Delete an Enum Entry

To delete enum entries, select the entries that you want to remove. Right mouse-click and choose the **Delete** option, OR select the button on the editor's tool bar.

## Apply Changes

When you have completed making changes, select the button on the editor's tool bar. If you have changed the name of the enum, the *Data Type Manager* display is updated to reflect the new name in the tree.

## Change the Sort Order

As with most tables in Ghidra, you can change the sort order of a column by clicking on the column header. The icons on the header indicate the sort order, ascending () or descending () . You can also rearrange the columns in the table by clicking on the column header and dragging it to the new position. Changing the sort order has no effect on the enum.

Provided By: *Data Type Manager* Plugin

Related Topics:

- [Create Enum using the Data Manager](#)

# Translate Strings

The Translate Strings Plugin provides a framework to allow strings found in a program to be decorated with an alternate value that is more meaningful to the Ghidra user.

This plugin doesn't perform any natural language translation by itself. The user must install **string translation services** that do the actual translation. Extensions to Ghidra are installed via the **File → Install Extensions...** menu.

When a string has been translated, the translated value will be shown in place of the original value, bracketed with »chevrons«

## Translate Menu

The **Data → Translate** menu will appear in the popup menu of the **Listing** window when a string or string-like datatype is selected, and in the **Defined Strings** table (found under **Window → Defined Strings**).

### Manual string translation

Allows the user to specify a translated string value manually, by typing a value in a pop-up dialog.

Select an existing string instance in the **Listing** window and right click and select **Data → Translate → Manual** to enter a manual translation.

In the **Defined Strings** table select a row or a range of rows and right click and select **Translate → Manual**.

### **Clear translated values**

Removes the stored translated value for the selected string data instances.

The selected string instances will default back to their true value.

Select an existing string instance in the **Listing** window and right click and select **Data → Translate → Clear translated values** to clear the translated value.

In the **Defined Strings** table select a row or a range of rows and right click and select **Translate → Clear translated values**.

### **Toggle show translated values**

Toggles the display of the translated string with the original value.

Select an existing string instance in the **Listing** window and right click and select **Data → Translate → Toggle show translated values** to toggle the display of the translated value of each of the strings.

In the **Defined Strings** table select a row or a range of rows and right click and select **Translate → Toggle show translated values**.

## **String translation services**

String translation services, which are separate from this Translate Strings Plugin, can be installed that will allow the user to translate strings.

Once installed, the translation service plugins, like all plugins, can be found in the **File ➔ Configure...** window and must be enabled before they will appear in the **Data ➔ Translate** menu.

Each string translation services will operate in a different way, please consult the documentation from the service for specifics.

The **Manual** string translation service is always available.

## Creating a String translation service

Please see the `SampleStringTranslationPlugin.java` source file for an example of how to create your own translation service.

Alternatively, you could customize the `TranslateStringsScript.java` file and operate directly on the string instances without using this plugin.

Provided by: *Translate Strings Plugin*

Related Topics:

- [Code Browser](#)
- [View Defined Strings](#)



# Labels

A label is a name associated with an address. Labels are used to make code easier to read. For example, instead of "call 0x103f2d", the instruction might read "call printf". The name "printf" has been associated with the address "0x103f2d". In other words, the address "0x103f2d" has been labeled printf.

## Label Name

All labels have an assigned symbol name which has either explicitly been established or a default generated name. When assigning a label name (or any other namespace or symbol name) the following naming restrictions apply:

- **Reserved Dynamic Name Pattern** –certain dynamic name patterns are reserved for Ghidra use and may not be explicitly set. A dynamic pattern generally consists of a default prefix (*SUB\_*, *LAB\_*, *DAT\_*, *UNK\_*, *EXT\_*, *FUN\_*, *OFF\_*) followed by an address (e.g., *LAB\_01234567*).
- **Space and Non-printable Characters Not Allowed** –a blank space character or other non-printable ASCII character (e.g., backspace, linefeed, etc.) are not permitted within a name
- **Length Limit of 2000** –the name length may not exceed 2000 characters/bytes



In addition to the above restrictions, the use of `::` within a label name may cause problems with certain name edit dialogs which may use this as a namespace separator.

## Label Source

Labels can be created or renamed by importing information, when auto-analysis is performed, or directly by the user. The label source indicates how the label was created or renamed.

- **User Defined** –Label name specified by the user.
- **Imported** –Label named during the import of the program or by some other imported information (for example, external libraries or C libraries.)
- **Analysis** –Label created by auto-analysis that do not have default names.
- **Default** –Label with a default name. (Ghidra generally creates default-named symbols at any address that is referenced by some other location.)

## Default Labels

Some labels are automatically generated during disassembly. They use a standard naming convention to derive the symbolic name from the address. The current convention is to use a prefix that gives some information about what is at the address, followed by the address. For example, the automatic label generated for address "0x01234" would be "LAB\_01234" if there is code at that address. If there was a data item at that address, the label would be "DAT\_01234". Labels with auto-generated names are known as *default* labels. The importer, auto-analysis, or the user can also create labels (or rename default labels) using more meaningful names.

Default label prefixes can be any of the following:

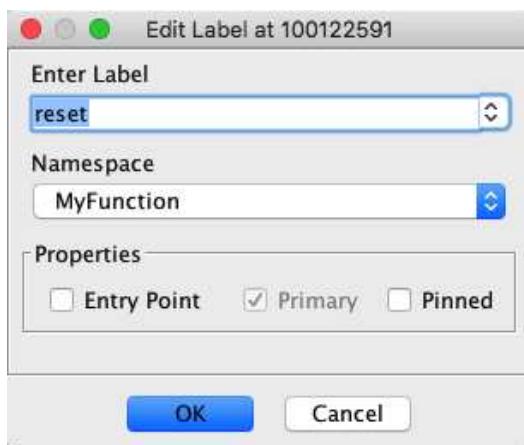
- **EXT** –indicates address is an external entry point.
- **FUN** –indicates there is a function at the address.
- **SUB** –indicates that code at the address has at least one "call" to it.
- **LAB** –indicates there is code at the address.
- **DAT** –indicates there is a data item at the address.
- **OFF** –indicates that the associated address is offcut, i.e. inside of an instruction or data item.
- **UNK** –default when one of the above cannot be recognized.

## Label Properties

- **Entry Point** –Indicates that the address associated with this label is an external entry point. External entry points are those addresses that can be used to initiate execution from outside the program. Most programs have a single "main" entry point. (Usually having the label "Entry".) Shared libraries (DLLs) usually have many entry points, one for each function in the library. Since the "entry point" property is really associated with an address and not a particular label, all labels at an address share this property.
- **Primary** –Indicates that this label will be the one used to represent the address everywhere the address is displayed, such as in the operand field of an instruction. Since multiple labels can be associated with an address, one and only one must be designated as primary.

## Add/Edit Label Dialog

The *Add Label* dialog and the *Edit Label* dialog are identical except for the title and the way the dialog's fields are initialized. The *Add Label* dialog will be filled in with suggested values for all fields. The *Edit Label* dialog, on the other hand will be filled in with the current values of the label being edited.



If you add a label where there is a function with a default label name, the label you add will become the function's new name.

### Dialog Fields

#### *Enter Label*

- Text field for entering the name of the label. A combo box is included which allows selecting recently used label names.

You may enter a namespace path in this text field that follows the format:

```
<namespace_name1>::<namespace_name2>::<...>::<label_text>
```

For example, the following string denotes a full namespace path that starts at the **Global** namespace and ends at the label name **myLabel**:

```
Global::foo::bar::myLabel
```

The namespace in the *Namespace* combo box will be used as the parent namespace for the label name and any included namespaces. However, if you provide a namespace path that starts with **Global**, then the value of the *Namespace* combo box will be ignored.

#### *Namespace*

- The defining scope of the label. The available namespaces are based upon the current address. When editing a label, the available namespaces are not necessarily those in the namespace hierarchy in which the label is located, but rather are those based upon the address of that label. The **Global** namespace is always included by default, as well as the parent namespace of the current label, if one is being edited.



This field is disabled, if there is a function with a default name at this address. The namespace will stay set to the parent namespace of the function and the label name you enter will become the new function name.

#### **Entry Point**

- Sets the entry point property for address associated with this label. Setting this property on one symbol, changes it for all symbols at the same address.

#### **Primary**

- Sets the primary property for this symbol. If there is only one symbol at this address, the checkbox will be selected and disabled, since it must be primary. Whenever the checkbox is selected, it will be disabled because the only way to make a symbol become non-primary is to select another symbol at the same address and make it primary. This ensures that there will always be one label that is primary. If there is a function at this address and you add a new label, the checkbox will be enabled such that if you select the checkbox, the function is renamed to the new label that you entered. The function symbol must always be the primary symbol.

#### **Pinned**

- Sets the label to pinned. A pinned label will not move if the image base is changed or a memory block is moved. A label that is not pinned, will move with the code or data if a memory block is moved or the image base is changed. Also, a pinned label will not be removed if the memory block that contains it is removed. Only code, data, or function labels may be pinned.

### **Set Label Dialog**

Normally, the primary label is used to replace an address reference in the operand field of an instruction or data item. Ghidra allows users to change which symbol is used to replace the address using the *Set Label* dialog.



#### **Label**

The list in the combo box will show all symbols associated with the address shown in the dialog title. Choosing a label from the list will cause that symbol to be associated with the operand reference being modified. Typing in a new name will cause a new symbol to be

created at the target address before associating it with the operand reference.

## Label Operations

### Adding a Label

Adding a label will place a label at a particular address in a listing. Labels may be of any length but may not contain spaces. To add a label:

1. Right-click and choose the **Add Label** menu option.
2. Enter the name of the label in the text field (or accept the suggested default).
3. Change any of the default options (see [Label Dialog](#))
4. Press the **OK** button.



Adding a label to an address where there is a function with a default name results in the function name becoming the new label name.

### Renaming a Label

To change the name of a label or referenced label appearing in an operand:

1. Right-click on the label, or referenced label appearing in an operand, then choose the **Edit Label** item from the popup menu.
2. Enter the new name of the label in the text field in the [Label](#) dialog.
3. Press the **OK** button.



If the label appearing in an operand corresponds to an external location the **Edit Label** action will be replaced by **Edit External Location**.

### Edit External Location

Similar to editing a label associated with the primary reference of an operand, an external location referenced by an operand may be renamed or modified. Right-click on the operand which references an external location and choose the **Edit External Location** menu option (see [Symbol Tree –Edit External Location](#) for more discussion on the use of the edit dialog).

### Removing a Label

Labels may or may not be eligible for removal depending on the following rules:

- Labels that have no references to them can always be removed.
- Labels at addresses that contain more than one label can always be removed.
- "Default" labels with at least one references and no other labels at that address can not be removed. (The menu option will be disabled.)
- "User-defined" labels with at least one reference and no other labels at that address can be removed, but will be replaced with a "default" label at that address.
- A "default" function label cannot be removed if there are no other labels at the address. To remove a "default" function label, you must remove the function itself.
- A "user-defined" function label can be removed. If there are no other labels at the address then the function label becomes a "default" label. If there are other labels at the address, one of these will become the new function label.

To remove a label:

1. Right-click on the label to be removed and choose the **Remove Label** menu option.



*Ghidra gives no confirmation on **Remove Label**. A status message is displayed if you try to remove a default function label.*

### Setting the Namespace

A *Namespace* defines a scope, such that symbol names are unique *within* a namespace. The types of namespaces that Ghidra supports are *Global*, *External*, *Function*, *Class*, and *Generic* namespaces that reside in the global namespace.

To set the namespace:

1. Right click on a symbol and choose the **Edit Label** menu option.
2. Select a namespace from the *namespace* combo-box in the **Label** dialog. Or, you may enter a new namespace with the label using ":" as a name separator. If a specified namespace does not exist a simple-namespace will be created.



*Any use of a class-namespace requires that it first be created prior to associating a label or other namespace with that class-namespace. This is most easily accomplished via the [Symbol Tree](#).*

### Setting an External Entry Point

External Entry points can only be created at addresses that have at least one symbol. To set an external entry point:

1. Right click on a symbol and choose the **Edit Label** menu option.
2. Check the *Entry Point* checkbox in the **Label** dialog.

### Making a Label Primary

Making a label primary gives it priority over other labels that are associated with the same address. The primary label is displayed by other Ghidra features instead of the address. For example, in the subroutine view, the subroutine names are primary labels. If another label were added and made the primary label, then the subroutine view will display that label instead of the label bearing the subroutine name.

If a function exists at an address, its name is the primary label; if you set another label at this address to be primary, then the symbol for this label is removed, and the function is renamed to that label that you were editing. The function symbol must always be primary.

To make a label primary:

1. Right click on a symbol and choose the **Edit Label** menu option.
2. Check the *primary* checkbox in the **Label** dialog.

### Selecting a referenced Label for an Operand

Referenced labels appear in instruction operands. By default, the primary label associated with the primary reference from an operand will be displayed within the operand. To have the operand display a different label corresponding to the primary memory reference:

1. Right click on the operand symbol and choose the **Set Associated Label...** menu option from the pop-up menu. This action only appears if the primary reference is a memory reference.
2. Choose a label from the drop-down list on the **Set Label** dialog or type in a name for a new label that will appear at the referred-to address.

Provided by: the *Edit Labels* Plugin

### Show Label History

You can show the history of changes on labels at a given address. You can also search the label history for all labels looking for old label names that no longer exist. Either way, a dialog is shown containing a table of label changes. The "Action" column indicates whether the label was added, removed, or renamed. If the label was renamed, the "Label" column shows the old name renamed to the new name. The "User" is the user who made the change. The "Modification Date" is when the change was made. Labels that were added as a result of disassembly are not recorded in the history; however, if you rename a default label, you will see an entry in the table, as shown below.



*A column for "Address" shows up in the table if you are viewing the history of changes on labels at all addresses.*

The screenshot shows a Mac OS X style dialog titled "Show Label History for 0040a671". The table has columns: Action, Label, User, and Modification Date. The data is as follows:

Action	Label	User	Modification Date
Add	MyLabel	User1	2018 Jun 27...
Remove	Phil	User1	2018 Jun 27...
Rename	Bob to John	User2	2018 Jun 27...

At the bottom right of the dialog is a blue "Dismiss" button.

To display the history of label changes at a specific address,

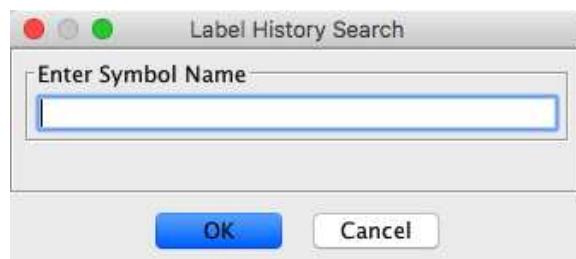
1. Right mouse click on a label (either in the label field or the operand field).
2. Choose the **Show Label History** option.



You can sort the label history by any of the columns and in ascending or descending order. By default, the history is sorted by ascending modification date (i.e., oldest date first). You can also reorder the columns by dragging the header to another column position.

To search for label history for a past or present label name:

1. Select **Search → Label History...**



2. A dialog is displayed so that you can enter a label name (or part of a label name)

- Enter a string you would like to see matches for.
- To display the label history for all the label changes, leave the field blank.

3. Select the OK button or press the <Enter> key in the text field.

- If label history was found, a dialog similar to the one shown above is displayed with the addition of an "Address" column. The input dialog remains displayed if no label history was found.
- From the label history dialog, you can navigate to each address by clicking on the row in the table.

Provided by: the *Edit Labels* Plugin

Related Topics:

- [Symbol Table](#)
- [Symbol Tree](#)
- [Edit Field Names](#)

# Comments

Comments can be added to any instruction or data item. There are five categories of comments that are supported:

<b>End-of-line (EOL)</b>	Displayed to the right of the instruction.
<b>Pre</b>	Displayed above the instruction.
<b>Post</b>	Displayed below the instruction.
<b>Plate</b>	Displayed as a block header above the instruction. Plate comments are automatically surrounded by *'s
<b>Repeatable</b>	Displayed to the right of the instruction if there is no EOL comment. These comments are also displayed at the "from" address of a reference to this code unit, but only if there is no EOL or repeatable comment defined at the from address.

You can create, edit, or delete comments. You can also view a [history of comment changes](#) for each comment type. You may also use [annotations](#) to change the display characteristics of data entered into the comment fields.

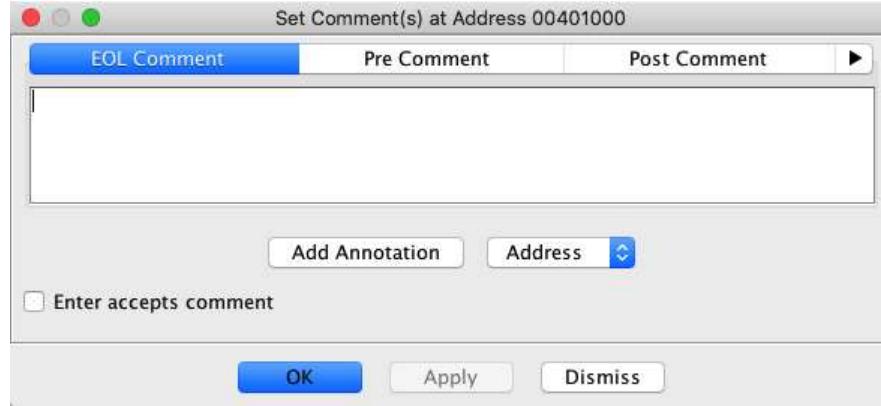
## Adding or Editing Comments (Set Comment)

Comments can be added or edited using the *Set Comment(s)* dialog as follows:

1. From the right-mouse pop-up menu over the Code Browser window, select the **Comments** ➔ **Set** menu option.
2. Choose the appropriate tab for the type of comment that is to be added or edited.
3. Enter the comment text in the text window.
4. Press the **OK** button to save the changes and close the dialog.



If the *Enter accepts comment* checkbox is selected, you can simply press the **Enter** key to set the comment and close the dialog. When the checkbox is not selected, the **Enter** key simply adds a new line to the current comment for a multiple line comment.



## Deleting Comments

Delete Comment will remove a specified comment from the listing. No confirmation is displayed before the delete.

To Delete a Comment:

1. Right-click on the comment to be deleted.
2. Choose **Comments** ➔ **Delete <comment type> Comment** from the popup-menu.

To undo a delete comment operation, use the [Undo Operation](#).

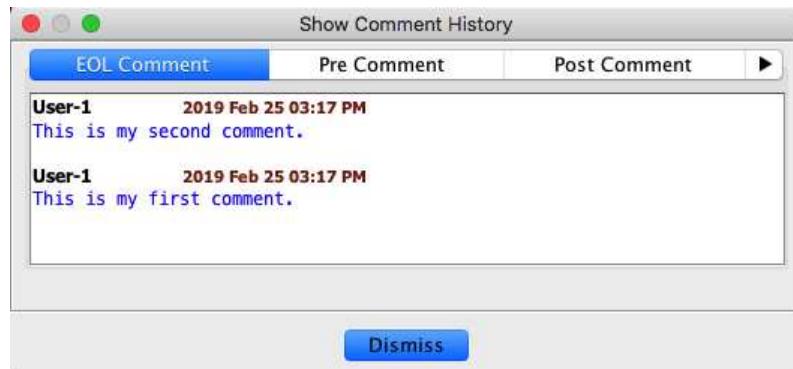
## Navigating in the Code Browser

If a comment contains a string that can be interpreted as an address or a label, you can navigate (in the [Code Browser](#)) to

that address or label by double clicking on the address or label in the comment. If the string matches more than one address or label, a [Query Results dialog](#) displays the matches.

## Displaying Comment History

Comment history is maintained for each change made to a comment. The history includes the name of the user who made the change, the modification date, and the comment. Comment changes are sorted in descending date order (most recent date first).



To view the history of changes,

1. Right-click on the comment that you would like to view the history.
2. Choose **Comments ➔ Show History for <comment type>....**
  - If you are not over a specific comment, right mouse click and choose **Comments ➔ Show History....**
3. A dialog containing a tab for each of the comment types is displayed.
  - If the comment does not contain history, then "No History Found" is shown in the tabbed panel for that comment.
  - Select a tab to view comment history for that comment type.



Only comments that are placed at the start address of data or instructions are displayed. Comments on addresses other than the start (interior) address will remain *hidden* until the data or instructions are cleared.

Provided by: *Edit Comments* Plugin

Related Topics:

- [Bookmarks](#)
- [Browser Field Formatter](#)
- [Labels](#)
- [Annotations](#)

# References

Within Ghidra, the term "Reference" covers two very broad areas:

1. *Forward References* from a data or instruction code unit to a memory, external, stack, or register location
2. *Back References* to a location of interest (e.g., code unit, variable, data-type, etc.).

## Forward References

*Forward References* can be explicitly defined within the program, on data or instruction code units. This is accomplished during disassembly, auto-analysis or manually by the user. In addition, *Forward References* to function parameters and variables may also be inferred. Both explicit and inferred *Forward References* can affect the code unit rendering and XRefs (i.e., Back References) displayed within the CodeBrowser listing. The effect of *Forward References* on the rendering of code unit operands can be somewhat controlled by means of various Code Browser Options (see [Operands Field options](#)).

Management capabilities for explicit *Forward References* is provided by the **ReferencesPlugin** and is discussed in detail within the [Forward References help section](#).

## Back References

*Back References* (also referred to as "*Location References*") include both the inverse of *Forward References*, and the identification of listing locations where specific data-types are utilized. The viewing of *Location/Back References* is provided by the **LocationReferencesPlugin** and discussed in more detail

on the [Location References Dialog help section.](#)

# Forward References

This page covers the follow topics relating to explicit forward references and the specific functionality provided by the ReferencesPlugin:

- [Introduction](#)
- [Types of References](#)
- [Reference Destination Symbols](#)
- [Ref-Types](#)
- [Actions for Creating and Deleting References from a Code Unit](#)
- [Viewing and Editing References](#)
- [Adding a Reference](#)
- [Editing a Reference](#)
- [Memory Reference Panel](#)
- [External Reference Panel](#)
- [Stack Reference Panel](#)
- [Register Reference Panel](#)
- [Adding Memory References from a Selection](#)

## Introduction

Explicit forward references exist within a program to identify execution flow or a data relationship between a '*source*' referent and a '*destination*'. Inferred forward references are sometimes rendered automatically within the Listing to aid the user (e.g., inferred function variable references). This page only covers the management of "explicit" forward references which are stored within the Program file.

A forward reference *source* is either an instruction or data code unit within the program identified by a memory address. In addition, the source is qualified by an *operand-index* which identifies the mnemonic or operand of an instruction which is making the reference. Associating a reference to the correct operand allows the Listing to render the data or instruction in a more friendly fashion and also facilitates navigation within the program when you double click on an operand field within the Listing.



*When a reference is placed on an operand, it will only change the rendering of that operand within the program listing if the reference is marked as 'primary'.*



*If a reference is placed on an instruction mnemonic within the Listing, the instruction mnemonic will be underlined.*



*If a non-primary reference exists for an operand (i.e., not reflected in the instruction markup), the corresponding instruction operand within the Listing will be underlined.*

## Types of References

The following types of explicit forward references may be defined from a mnemonic or operand of an instruction or data code unit:

- [Memory Reference](#) (*includes Offset and Offcut references*)
- [External Reference](#)
- [Stack Reference](#)
- [Register Reference](#)



*Ghidra does not permit mixing "types of references" for a given mnemonic or operand.*



*With the exception of Memory References, only a single reference may be placed on a given mnemonic/operand.*

## Memory References

All *MemoryReference* have a destination defined by a memory address within the current program. A variation of the typical memory reference is the *OffsetReference* which permits the destination address to be specified as some base

memory address plus a 64-bit signed offset.

*MemoryReferences* are used to specify either a data access or execution flow within the Program. This distinction is made by specifying an appropriate [Ref-Type](#) on the reference. Adding and removing certain flow references may change the set of instructions which make up subroutine and code blocks. Since function bodies are established based upon the [Block Models](#), it may be necessary to redefine the body of [Functions](#) affected by a new flow reference.

Any *MemoryReference* can be characterized as an *OffcutReference* if its destination address is contained within a data or instruction code unit and does not correspond to the minimum address of that code unit. Such *OffcutReferences* produce an offcut label which uses a special label color. In addition, since the corresponding label is hidden within a code unit, a special `OFF_<address>` label appears on the containing code unit within the Listing. Double-clicking the similarly colored XRefs within the Listing will allow you to quickly identify the *source* of the memory reference (see [Code Browser Navigation](#)). In general, the presence of an Offcut Reference may indicate an error within the disassembly.

### External References

External References are used to define either a data access or execution flow to a memory destination located within a different Program file. This type of reference is typically used when linking to a library module. An External Reference destination is defined by a *External Program Name* and memory location which may be identified by a label or address contained within the *External Program File*. The resulting destination symbol takes on the name of the external label or `EXT_<address>` if only an external address was specified. All external destination symbols have a namespace which corresponds to the associated *External Program Name* (e.g., `MSVCRT.DLL::_controlfp`).

An *External Program Name* is considered to be "resolved" when it has been linked to a Program file contained within the same project. All external symbol names, corresponding to unresolved *External Program Names*, will be displayed in red. The [External Program Names...](#) action/dialog can be used to set or modifying these external Program file linkages.

External References currently utilize the single RefType of EXTERNAL.

### Stack References

*StackReferences* define data access to a function parameter or local variable located on the stack. All *StackReferences* have a destination defined by a stack offset within the containing function's stack frame. If a real frame-pointer is not used, the Function analysis will track the stack pointer usage and establish a virtual stack frame for the purpose of defining stack parameters and local variables. When creating a *StackReference*, a corresponding stack parameter/variable may be explicitly bound to the reference (see [Function Variables](#)), although this is generally unnecessary since this relationship can generally be determined from the stack offset. The sign of the stack frame offset makes the distinction between a parameter or local variable assignment. Stack usage conventions are established by the Language module in use.



*Stack References should be placed on all stack parameter/variable data access operands.*



*Stack References may only be specified for source code units contained within a function.*

### Register References

*RegisterReferences* define data access to a function parameter or local variable located within a register. All *Register References* have a destination defined by a register within the context of a containing function. When creating a *Register Reference* a corresponding register variable may be explicitly bound to the reference (see [Function Variables](#)), although this is generally unnecessary since this relationship can generally be determined from the register used.



*Register References should be placed only on register variable data assignment operands.*



*Register References may only be specified for source code units contained within a function.*

### Reference Destination Symbols

An important characteristic of all references is that a symbol is created for all destinations. These symbols generally appear within the listing and can always be found within the [Symbol Table](#) or [Symbol Tree](#). The following default symbol names are produced by the creation of a reference.

Symbol Name Format	Type	Namespace	Description
--------------------	------	-----------	-------------

<b>LAB_&lt;address&gt;</b>	Memory	Global	A memory address "label" identifying a branch flow memory reference destination.
<b>SUB_&lt;address&gt;</b>	Memory	Global	A memory address "label" identifying a call flow memory reference destination.
<b>DAT_&lt;address&gt;</b>	Memory	Global	A memory address "label" identifying a data memory reference destination. If the <i>address</i> corresponds to defined data, the <b>DAT</b> prefix will be replaced by the corresponding data-type (e.g., <b>BYTE_&lt;address&gt;</b> , <b>DWORD_&lt;address&gt;</b> ).
<b>OFF_&lt;address&gt;</b>	Memory	Global	A memory address "label" identifying a memory reference destination which is located at an offcut address within a code unit. These labels only appear within the listing to flag the existence of a hidden offcut label. Double-clicking the corresponding offcut XRef will take you to the code unit which has the offcut reference,
<b>param_&lt;ordinal&gt;</b>	Stack/Register/ Memory	Function Name	A function parameter associated with stack, register or memory location. Parameter references identified by a parameter name which by default includes its ordinal position.
<b>local_&lt;reg-name&gt;[_&lt;firstUseOffset&gt;]</b>	Register	Function Name	A local function variable associated with register references. In addition, the first-use-offset within the function is included if non-zero.
<b>local_&lt;offset&gt;[_&lt;firstUseOffset&gt;]</b>	Stack	Function Name	A local function variable associated with stack references identified by an absolute offset within the stack frame. In addition, the first-use-offset within the function is included if non-zero.
<b>&lt;ext-label&gt;</b>	External	EXTERNAL_NAME	A external program location associated with external references and identified by a label name or address within the external program file. Each EXTERNAL_NAME can be associated with a specific program file within your Ghidra project (see <a href="#">External Program Names</a> ). If only an address is specified when creating an external reference, a label format of EXT_<addr> is used.

The above symbol name colors correspond to the default color scheme (see [Code Browser Options](#)). Memory Reference destinations outside the Program's memory map, and unresolved External Reference destinations utilize the "Bad Reference Address" color (e.g., **DAT\_00010000**) in place of the normal color shown.

The presence of

The actual default label utilized for a memory location can change dynamically and is produced by considering the following naming precedence (listed highest to lowest). Note that some of the following label types are not a result of a reference, but are considered when producing the default label name.

1. FUN\_<addr> : Function entry point
2. EXT\_<addr> : External entry point
3. SUB\_<addr> : Call flow reference destination
4. LAB\_<addr> : Branch flow reference destination

5. DAT\_<addr> : Data reference destination

## Ref-Types

The term **Ref-Type**, as used within Ghidra, is rather ambiguous –and will hopefully be changed in a future release of Ghidra. While the following "types of references" are supported in Ghidra: memory, stack, register and external. We define **Ref-Type** as the "type of data access" or "type of flow" associated with a reference. The following table attempts to clarify the various **Ref-Types** and when they can be used.

Ref-Type	*Reference	Flow/Data	Description
DATA	MSR	Data	General data/pointer reference
DATA_IND	M	Data	General indirect data reference
READ	MSR	Data	Direct data read reference
READ_IND	M	Data	Indirect data read reference
WRITE	MSR	Data	Direct data write reference
WRITE_IND	M	Data	Indirect data write reference
READ_WRITE	MSR	Data	Direct data read/write reference
READ_WRITE_IND	M	Data	Indirect data read/write reference
STACK_READ (Note 1)	S	Data	Direct stack read reference
STACK_WRITE (Note 1)	S	Data	Direct stack write reference
EXTERNAL_REF (Note 2)	E	—	External program reference (flow or data access unspecified)
INDIRECTION	M	Flow	Indirect flow via a pointer (reference should be from an instruction to a pointer). Alternatively, a COMPUTED CALL or JUMP reference could be placed from an instruction to one or more indirect destination instructions.
COMPUTED_CALL	M	Flow	Computed call flow from an instruction
COMPUTED_JUMP	M	Flow	Computed jump flow from an instruction
UNCONDITIONAL_CALL	M	Flow	Unconditional call flow from an instruction
UNCONDITIONAL_JUMP	M	Flow	Unconditional jump flow from an instruction
CONDITIONAL_CALL	M	Flow	Conditional call flow from an instruction
CONDITIONAL_JUMP	M	Flow	Conditional jump flow from an instruction
CALL_OVERRIDE_UNCONDITIONAL	M	Flow	Used to override the destination of a CALL or CALLIND pcode operation. CALLIND operations are changed to CALL operations. The new call target is the "to" address of the reference. The override only takes effect when the reference is primary, and only when there is exactly one primary CALL_OVERRIDE_UNCONDITIONAL reference at the "from" address of the reference.
JUMP_OVERRIDE_UNCONDITIONAL	M	Flow	Used to override the destination of a BRANCH or CBRANCH pcode operation. CBRANCH operations are changed to BRANCH operations. The new jump target is the "to" address of the reference. The override only takes effect when the reference is primary, and only when there is exactly one primary JUMP_OVERRIDE_UNCONDITIONAL reference at the "from" address of the reference.
CALLOTHER_OVERRIDE_CALL	M	Flow	Used to change CALLOTHER pcode

			operations to CALL operations. The new call target is the "to" address of the reference. The override only takes effect when the reference is primary, and only when there is exactly one primary CALLOTHER_OVERRIDE_CALL reference at the "from" address of the reference. Only the first CALLOTHER operation at the "from" address of the reference is changed. Note that this reference override takes precedence over those of CALLOTHER_OVERRIDE_JUMP references.
CALLOTHER_OVERRIDE_JUMP	M	Flow	Used to change CALLOTHER pcode operations to BRANCH operations. The new jump target is the "to" address of the reference. The override only takes effect when the reference is primary, and only when there is exactly one primary CALLOTHER_OVERRIDE_JUMP reference at the "from" address of the reference. Only the first CALLOTHER operation at the "from" address of the reference is changed.

\*In the above table, the **Reference** column indicates the "*type of reference*" for which the **Ref-Type** is applicable: Memory, Stack, Register and External.

#### NOTES:

1. The use of STACK\_READ and STACK\_WRITE Ref-Types will likely be replaced with READ and WRITE Ref-Types in a future release of Ghidra.
2. The EXTERNAL\_REF is a general purpose Ref-Type used for all External references. At this time, the type of flow or data access for an external reference is unspecified.
3. If you need to alter the FALL\_THROUGH flow behavior of an instruction, modify the [Fallthrough Address](#) instead of adding a memory reference.

## Actions for Creating and Deleting References From a Code Unit

There are three actions provided by the ReferencesPlugin which are accessible from the CodeBrowser Listing while the current cursor location is on the mnemonic or operand of an instruction or data code unit. The create and delete reference actions may be disabled under certain conditions.

- [References from](#)
- [Add Reference from...](#)
- [Create Default Reference](#) (menu item varies based upon current operand)
- [Delete References](#) (menu item varies based upon mnemonic/operand current references)



Default key-bindings for actions are indicated with {}'s.

### Add Reference From

While there is a separate action for creating a default reference on an operand (see [Creating a Default Reference](#) below), an arbitrary reference may be also be added directly to a mnemonic or operand by using the popup menu action **References** → **Add Reference from...**. This will cause the [Add Reference Dialog](#) to be displayed, allowing the user to specify any of the permitted reference types.

### Creating a Default Reference {Alt-R}

While the current cursor location is on the operand of an instruction or data code unit within the CodeBrowser Listing, the popup menu item **References** → **Create Default Reference**\* may be selected to create the default primary reference for an operand. This action will be disabled if the current location does not correspond to an operand field or a default reference can not be determined.

When creating a default *Memory* reference on a scalar operand, for programs with multiple memory spaces, repeatedly invoking this action will cycle the default reference through all suitable memory spaces. If the wrong memory space was used in creating the *Memory* reference, simply repeat the action.

When adding a *Stack* or *Register* reference, a corresponding parameter or variable may be created. If a local variable is created, the first-use-offset of the variable will correspond to the source instruction location. For this reason, it is recommended that the first reference to a variable be created on the the first "assignment" instruction. If a newly created variable is unwanted, it may be deleted by clicking on it within the Listing and hitting the "Delete" key. Keep in mind that when a variable is deleted, any explicit bindings to that variable will be cleared.



\*The popup menu item name *Create Default Reference* may differ based upon the type of reference which will get created: *Create Memory Reference*, *Create Stack Reference*, *Create Register Reference*.

#### **Deleting References from a Code Unit {Delete}**

While the current cursor location is on the mnemonic/operand of an instruction or data code unit within the CodeBrowser Listing, the popup menu item **References ➤ Delete References\*** may be selected to delete all references on the current mnemonic/operand. This action will be disabled if the current location does not correspond to a mnemonic/operand field or references do not exist on the current mnemonic/operand.



\*The popup menu item name *Delete References* may differ based upon the existing reference(s): *Delete Memory References*, *Delete Stack Reference*, *Delete Register Reference*, *Delete External Reference*.

#### **Viewing and Editing References (Add/Edit...) {'R'}**

All references "from" a data or instruction code unit can be edited and/or viewed by clicking on the code unit (or a specific operand) within the Listing and activating the *Add/Edit...* action via the popup menu item **References ➤ Add/Edit... {'R'}**.

Each time this action is invoked a new instance of the **ReferencesEditor** panel will be displayed. Once the panel is displayed, the toggle button may be pushed-in to have the *source* location follow the current location within the [Listing](#) display.

References Editor						
Source						
Operand	Destination	Label	Ref-Type	Primary?	Source	
MNEMONIC	0040235c	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	00402377	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	004023c6	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	00402400	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	00402408	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	0040243f	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	004024f5	switchD_00402355::cas...	COMPUTED_JUMP	<input checked="" type="checkbox"/>	ANALYSIS	
MNEMONIC	00402537	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
MNEMONIC	00402ad0	switchD_00402355::cas...	COMPUTED_JUMP	<input type="checkbox"/>	ANALYSIS	
OP-0	00402b2b	->switchD_00402355::::...	READ	<input checked="" type="checkbox"/>	ANALYSIS	

#### **Source**

The references displayed and managed within this panel are all "from" a single *source* instruction or data code unit. The current *source* code unit is displayed at the top of the **ReferencesEditor** panel as it would appear in the [Listing](#). Using this display, you can click on either the code unit Mnemonic or an individual operand to highlight the corresponding references within the table below and to set the operand target when adding additional references. The selected Source operand will be treated as the "active source operand" used for Add actions. These operand labels will also act as drag-n-drop target zones for code unit selections dragged from the [Listing](#) (see [Adding Memory References from a Selection](#)).



The table entries that match the selected source element will be gray in color.

The Home Button  can be used to set the current Listing location to the Source code unit address.

### References Table

All references "from" the current *source* code unit are listed within the table with the following columns:

- **Operand** –Indicates on which portion of the code unit the reference has been placed (MNEMONIC, OP-0, OP-1, OP-2, . . .).
- **Destination** –Indicates the destination location associated with the reference. The destination displayed for each type of reference utilizes a different format:
  - <address> : indicates a memory destination
  - <address><signed-offset> : indicates an offset memory reference relative to a base address.
  - Stack [<signed-offset>] : indicates a stack reference with a specified stack frame offset.
  - <register> : indicates a register reference for the specified register.
  - External : indicates an external reference
- **Label** –Indicates the namespace-qualified symbol name associated with the destination (See [Reference Destination Symbols](#)).
- **\*Ref-Type** –Identifies the type of data access or instruction flow associated with a reference.
- **\*Primary?** –Allows the user to choose a single memory reference which will be reflected in the rendering of a code unit operand.
- **User Ref?** –References which were manually added by the user or by means of auto-analysis will have a check displayed.

 \*With the exception of External references, both the Ref-Type and Primary? choices may be changed directly within this table.

 References and symbol names corresponding to memory references outside of the program's defined memory blocks will be displayed in red (e.g., **DAT\_00000000**). These red references frequently correspond to well-known memory locations, although they could point out a bad reference. [Creating memoryblocks](#) for valid fixed memory locations (e.g., memory mapped I/O regions) will help to resolve some of these apparent "BAD" references.

### Actions

The following actions are available from the **ReferencesEditor** panel. For those actions with a default key-binding or mouse-click-binding, this has been indicated with { }'s.

 **Add Reference** {Insert-key} –Invoking this action will launch the Add Reference Dialog for the current code unit (see [Adding a Reference](#)).

 **DeleteReferences** {Delete-key} –Invoking this action will delete all selected references.

 **Edit Reference** {Enter-key or double-click a row} –Invoking this action will popup the **Edit Reference Dialog** for the selected reference (see [Editing a Reference](#)). This action is only available when a single reference row is selected.

 **Select Memory Reference Destination** –With one or more memory references selected in the table, invoking this action will cause the corresponding locations within the [Listing](#) to become selected.

 **Follow Tool Location Changes** –Once enabled (i.e., button pushed-in), any location change within the tool (e.g., Listing panel) will cause the currently displayed *source* code unit and associated references to reflect the new location.

 **Send Location Change for Selected Reference Destination** –Once enabled (i.e., button pushed-in), selecting a single row within the references table will send a location change to the tool corresponding to the selected *destination*. This will have the effect of scrolling the [Listing](#) to selected *destination*. In the case of an external location, an attempt will be made to open the corresponding program and scrolling to the corresponding external label within that program.

 **GoTo Reference Source Location** –Invoking this action will send a location change to the tool corresponding to the *source* code unit. This will have the effect of scrolling the [Listing](#) to the current *source* code unit.

## Adding a Reference

Invoking the **Add...** action from the **ReferencesEditor** window will cause the **Add Reference Dialog** to be displayed for the current *Source* code unit. Once displayed, the *Source* code unit mnemonic or operand may be selected by clicking on it, as well as the *Type of Reference*. The available choices for Type of Reference may be constrained based upon the chosen operand.



*In general, only flow references should be set on an instruction mnemonic, unless of course the instruction has no operands. References from data code units (e.g., addr/pointer) should always specify the scalar operand as the source, not the mnemonic (i.e., data-type).*



*Stack and register references may only be specified for source code units contained within a function. Register references may only be set on operands containing a single register and in general should correspond to a WRITE Ref-Type.*



*With the exception of memory references, only a single reference may be set for a given operand or mnemonic.*



*An External reference may not be set on a mnemonic.*



Based upon the chosen *Type of Reference*, the lower portion of the dialog will change. The following sections discuss the input panels for each of the four possible choices:

1. [Memory Reference Panel](#) (*includes Offset and Offcut references*)
2. [External Reference Panel](#)
3. [Stack Reference Panel](#)
4. [Register Reference Panel](#)

Once the appropriate reference panel has been filled-in as required, the **Add** button may be clicked to complete the operation.

When adding a *Stack* or *Register* reference, a corresponding parameter or variable may be created. If a local variable is created, the first-use-offset of the variable will correspond to the source instruction location. For this reason, it is recommended that the first reference to a variable be created on the the first "assignment" instruction. If a newly created variable is unwanted, it may be deleted by clicking on it within the Listing and hitting the "*Delete*" key. Keep in mind that when a variable is deleted, any explicit bindings to that variable will be cleared.

## Editing a Reference

Invoking the **Edit...** action from the **ReferencesEditor** window will cause the **Edit Reference Dialog** to be displayed for

the current *Source* code unit. Once displayed, the *Source* code unit mnemonic or operand corresponding to the edited reference will be selected, as well as the *Type of Reference*. Neither the *Source* operand nor the *Type of Reference* may be changed when editing a reference. If you wish to change either of these settings you must delete the reference and [add a new reference](#).

The *Edit Reference Dialog* uses the same layout as the [Add References Dialog](#) with the only exception being the dialog title and the *Add* button which is named ***Update*** in the Edit mode. Similarly, the lower portion of the dialog will vary based upon the *Type of Reference*. The following sections discuss the input panels for each of the four possible choices:

1. [Memory Reference Panel \(includes Offset and Offcut references\)](#)
2. [External Reference Panel](#)
3. [Stack Reference Panel](#)
4. [Register Reference Panel](#)

Once the specific reference panel settings have been modified, the ***Update*** button may be clicked to complete the operation.

## Memory Reference Panel

A [Memory Reference](#) identifies a data access or instruction flow to another memory location within the same program space. A memory reference may optionally be specified as an [Offset Reference](#) relative to a specified *Base Address*. The term Offcut is used to characterize a memory reference or its resulting label whose destination address does not correspond to the start of a data or instruction code unit (see [Offcut References](#)).

Below is an image of the *Memory Reference Panel* as it might appear in the *Add Reference* or *Edit Reference Dialog*. The two views reflect a regular memory reference and an *Offset* reference (*Note the Address label change based upon the Offset selection state*).



### Offset

If the *Offset* checkbox is "checked", this memory reference will be treated as an *Offset Reference* relative to the specified *Base Address*. The actual "to" address will be computed by adding the specified signed *Offset* value to the *Base Address*. The number format is assumed to be decimal unless the "0x" prefix is used when entering a Hex value.

### To Address

[ *Offset*] The *To Address* entry is required for normal memory references and specifies the reference destination as a memory offset. This entry is always interpreted as a unsigned hex value (i.e., the "0x" entry prefix is assumed). For those processors with multiple address-spaces, a pull-down is also provided allowing the address-space to be selected.

### Base Address

[ *Offset*] The *Base Address* entry is required for offset memory references and specifies the offset base location as a memory offset. This entry is always interpreted as a unsigned hex value (i.e., the 0x entry prefix is assumed). For those processors with multiple address-spaces, a pull-down is also provided allowing the address-space to be selected.

#### **Address History Button**

The *AddressHistory* pulldown button may be used to recall a previously applied *To Address* or *Base Address* entry. Only the last ten (10) address entries are maintained for each open Program.

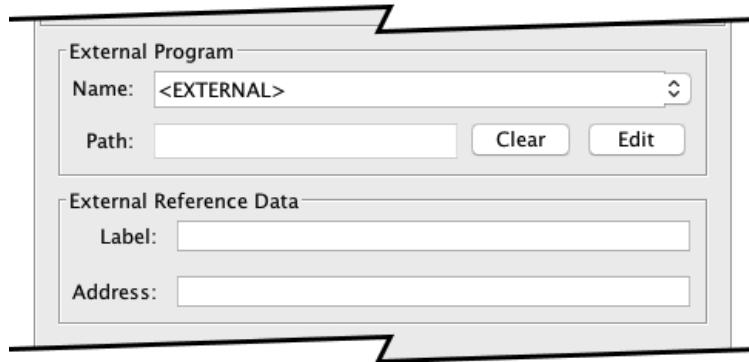
#### **Ref-Type**

Allows selection of the data access or instruction flow type associated with this reference (see [Ref-Types](#)).

### **External Reference Panel**

An [External Reference](#) identifies a memory destination within another Program file. Such references are generally used to indicate a library module linkage. The memory location within the *External Program* is identified by either a *Label* or an *Address*.

Below is an image of the *External Reference Panel* as it might appear in the *Add Reference* or *Edit Reference Dialogs*:



#### **Name**

This field identifies a namespace name corresponding to the *External Program* and may be typed-in or chosen from the pull-down list of those previously defined. This is a required input.

#### **Path (Clear/Edit)**

This field identifies the Program file within the Ghidra Project which corresponds to the selected Name. Associating the *External Program Name* with a Program file *Path* is optional, but can be useful to facilitate navigation to an associated library if it is contained within the same project. This *Name/Path* association can easily be "resolved" at a later time via the [External Program Names Dialog](#).

#### **Label / Address**

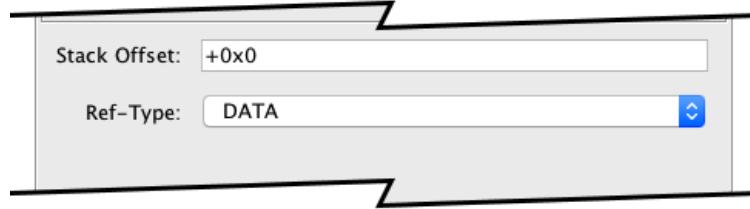
The specific memory location within the External Program is identified by either a Label defined within the corresponding Program file, or via a specific Address. If both a Label and Address are specified, the Label will take precedence during navigation. The *Address* field is always interpreted as a hex value (i.e., the 0x entry prefix is assumed) offset within the default address space.

### **Stack Reference Panel**

A [Stack Reference](#) identifies a data access to a function parameter or local variable within the containing function's stack

frame.

Below is an image of the *Stack Reference Panel* as it might appear in the *Add Reference* or *Edit Reference Dialogs*:



#### **Stack Offset**

Specifies a signed offset within the containing function's stack frame. The number format is assumed to be decimal unless the "0x" prefix is used when entering a Hex value.

#### **Ref-Type**

Allows selection of the data access or instruction flow type associated with this reference (see [Ref-Types](#)).

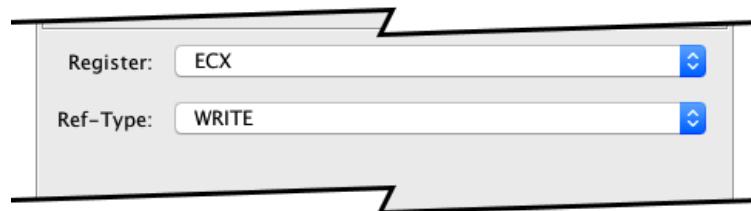
#### **Variable Name**

An optional entry which identifies the variable name to be associated with this reference. Selecting an existing variable will automatically change the Stack Offset to match the selected variable. Entering a new name which does not exist will cause a new stack parameter or variable to be created with the reference. Clearing this field will have the same effect as keeping the initial default variable choice.

## **Register Reference Panel**

A [Register Reference](#) identifies a data access (i.e., value assignment) to a function local variable within the containing function's stack frame.

Below is an image of the *Register Reference Panel* as it might appear in the *Add Reference* or *Edit Reference Dialogs*:



#### **Register**

Indicates the selected operand's register to which a local register variable reference will be established.

#### **Ref-Type**

Allows selection of the data access or instruction flow type associated with this reference (see [Ref-Types](#)).

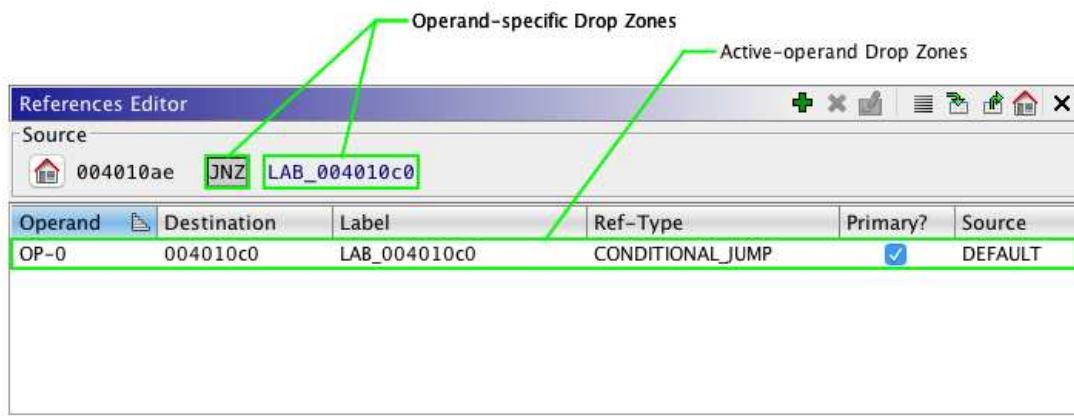
#### **Variable Name**

An optional entry which identifies the variable name to be associated with this reference. Entering a new name which does not exist will attempt to create a new local register variable with the reference. Clearing this field will have the same effect as keeping the initial default variable choice.

## Adding Memory References from a Selection

A code unit selection from the CodeBrowser Listing may be dragged and dropped onto the **ReferencesEditor** panel to create [Memory References](#) in bulk for the current Source. This capability must be used carefully since a separate reference will be created "to" every code unit contained within the selection.

The specific mnemonic/operand which will be used as the source for the new memory references depends on where the selection is "dropped" within the **ReferencesEditor** panel (see figure below). The preferred method is to "drop" the selection on the correct monc/operand within the **Source** code unit area (i.e., *Operand-specific Drop Zone*). Alternatively, the selection may be "dropped" on the reference table (i.e., *Active-operand Drop Zone*) to utilize the current mnemonic/operand choice from the **Source** code unit area. When "dropping" on the table, be careful "dragging" the selection across the **Source** code unit area since this could change the active Source mnemonic/operand for the panel.

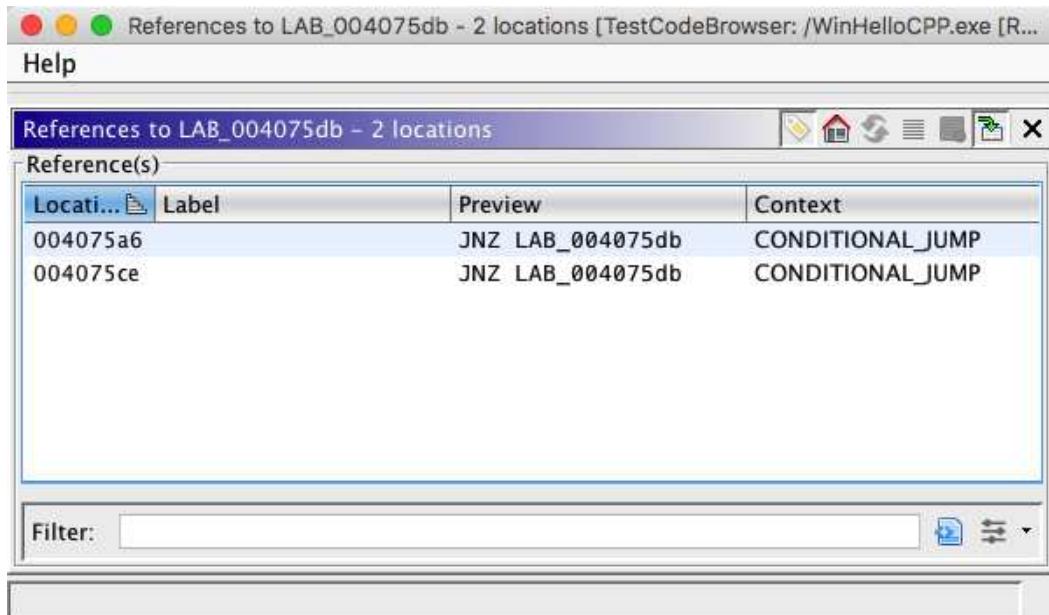


Provided By: *ReferencesPlugin*

Related Topics:

- [Resolving External Names](#)
- [Show References to a location](#)
- [Code Browser Navigation](#)

# Location References Dialog



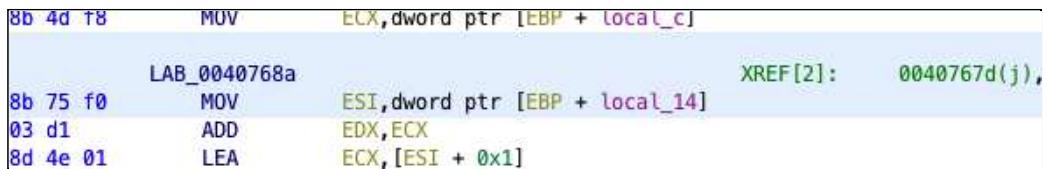
The *Location References Dialog* displays all the memory reference destinations to a specific code unit that is described by the item under the current location. For example, if the current location is a referenced label, then the dialog will show references to that label and not the current address. In the following snippet, the label "LAB\_40768a" is the field at the current location.



Showing the *Location References Dialog* dialog at this location will show all references to the label "LAB\_4078d2".

This dialog also works with data types. If the item at the current location is a data type, then all places that data type is applied will be shown and the data type will be highlighted in the *Data Type Manager* tree.

As another example, if the location is an XRef field location, then the dialog will show references to the address that the XRef field represents. In the following snippet, the Xref field "0040767d(j)" is the field at the current location.



Showing the *Location References Dialog* dialog at this location will show all references to the address 0040767d.



You can also [show references to data types](#) from the Data Type Manager. In this case, all locations where the selected data type is applied will be highlighted.

## To Find Location References

1. Right-mouse anywhere on the code unit\*
2. Select whichever of the following is available from the popup menu:
  - Select **References** → **Show References to** from the popup menu.
  - Select **References** → **Find References to** from the popup menu.
  - Select **References** → **Find Uses of** from the popup menu.

\*If the Location References Dialog cannot show references to the location that was clicked, then the menu item will be disabled.

The results of performing this action depend upon the location the cursor when the popup is shown. Here are some examples:

- When over a data type, a **search** is performed for all uses of that data type, such as in function signatures and locations that type is applied in memory,
- When over an address, all references to the code unit containing that address will be shown, (which is what the [Show References to Address](#) action does),

### Results Window Header

This title displays the item that is the destination of all the references in the *Reference(s)* table.

The  button will cause the code browser to return the cursor to the location from which the dialog was launched.

The  button will refresh the table of references. This button will appear disabled when the data is not stale. However, if Ghidra detects that the data *may* be stale, then the button will become color filled, as it is here. You may push the button for a refresh in either state.

The  button will create a selection in the code browser with the reference entries selected in the table. You may also access this feature by right-clicking an item in the table and selecting **Make Selection**.

The  button toggles the highlighting of the matching references. In this case, the term highlight refers to the background color of the item in the Listing and not a [Program Selection Highlight](#).



To instead make a Program Selection Highlight, use the select button mentioned above. Then, click from the menu bar **Select** → **Program Highlight** → **Entire Selection**

### Search Results

This table displays a list a reference sources, one for each reference associated with the selected destination instruction or data item. There are four fields by default that are displayed in the table:

**Location** –Shows the source address.

**Label** –Shows the primary label at the source address (if one exists).

**Preview** –Shows the instruction or data item at the source address.

**Context** –Shows the type of reference to the current location. If the result is a Ghidra

in-memory Reference, then this column will show the type of reference. If the result is found by way of the Decompiler, then the line from the decompiled function will be shown.



You can make a selection in the Code Browser from the entries in the table:

1. Click on an entry in the table, or Ctrl-click to select multiple entries.
  2. Right mouse click and choose **Make Selection**.
- The current selection in the Code Browser is set to include the addresses that you selected in the Address table.

## To Find Location References to Data Types

From within the [Data Type Manager](#) you can right-click on a data type and select **Find Uses of**. This will show a *Location References Dialog* that lists all locations where the selected data type is applied, such as in function signatures, return types, structures, etc. For **FunctionDefinition** the action will display all places that function is applied.



By default, finding uses of data types will search not only for applied data types, but also will perform dynamic discovery of data types using the **Data Type ReferenceFinder** service. This causes the search to be slower, but also reports many more type uses. To disable the dynamic searching, use the [Search → Dynamic Data Type Discovery tool option](#).

## To Show Location References to Code Unit

1. Right-mouse anywhere on the code unit\*
2. Select **References → Show References to Address** from the popup menu.

This will show all references to the current address, as well as all addresses consumed by the current instruction or data.

\*If the Location References Dialog cannot show references to the location that was clicked, then the menu item will be disabled.



This action will show only direct references to the current code unit. No other special reference finding will take place.

## Renaming Windows



see [Docking Windows –Renaming Windows](#)

Related Topics:

- [Forward References \(Reference Editor\)](#)
- [Data Type Manager](#)

# Equates

An Equate is a string substitution for [scalar](#) (a numeric value) in any code unit (instruction operand or data). For example, consider the instruction below:

```
MOV R2, $0xb
```

The scalar `$0xb` can be replaced with the string `BATTERY_FLAG_CRITICAL`. This will yield the following:

```
MOV R2, BATTERY_FLAG_CRITICAL
```

The substitution of "BATTERY\_FLAG\_CRITICAL" for `$0xb` is called an equate. That is, `$0xb` is equated to "BATTERY\_FLAG\_CRITICAL". Note that the default choice for the new equate application is your current location. "BATTERY\_FLAG\_CRITICAL" will replace the scalar value `$0xb` only at the current cursor location unless you choose a different option. However, when replacing another scalar of the same value, a list of previously declared Equates for that scalar value is presented.

Scalars can only be equated to strings. The string can be of any length and may contain spaces and special characters. Duplicate equate names are not allowed.

 It should be noted that for the purposes of this document, "scalars" refers to scalar values contained in *code units*. A code unit is an instruction operand or other data element.

There are several operations that are associated with Equates. They are:

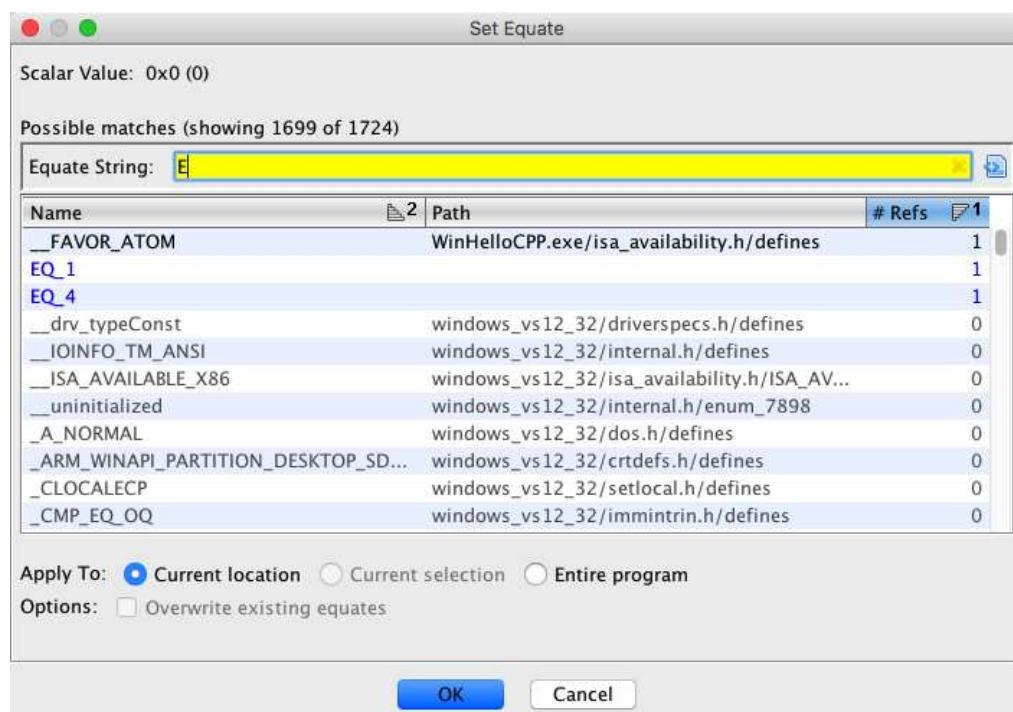
- [Set Equate](#)
- [Rename Equate](#)
- [Remove Equate](#)
- [Apply Enum](#)
- [View Equates](#)
- [Convert](#)

## Set Equate

The *Set Equate* action will create one or more Equates in the Listing.

To set an Equate:

1. Right-mouse-click on the target scalar value, and select **Set Equate** or press <E>.



2. When the dialog appears, either type in an equate string or choose one from the list of strings that are known to be associated with the given value. As you type in the "Equate String" field, the list will be narrowed down to show only the strings that contain the text that has been typed.
3. Select one of the choices from the "Apply To" list. **Current Location** is the default choice unless a selection is made, in which case the **Current Selection** option will be set. The other option is **Entire Program**.
  - a. **Current Location**: When selected the equate will be applied to the scalar value at the current location of your cursor only.
  - b. **Current Selection**: When selected the equate will be applied to all of the scalar values in your current selection that match the value of the scalar that you originally clicked. When you make a selection in your program this button will become enabled. If you do not make a selection then it will not be enabled and the option will be grayed out. Note that scalars in your selection that already have an equate set will not be affected by this unless you also select the overwrite option.
  - c. **Entire Program**: When selected the equate will be applied to all of the scalar values in the entire program that also match the value of the scalar that you originally right-mouse-clicked on.
  - d. **Overwrite existing equates**: This option is only enabled when setting equates in a selection or the whole program. If this option is selected, all scalars and all named equates in the selection or entire program, depending on which option is selected, will be set with the user-given equate name. If the overwrite option is not selected, only scalars not already with equates set will be assigned the user-given equate name.
4. Double-click on an entry in the list, or select an entry in the list and press **OK**, or type in the string and press **OK**. If any item in the list is selected it will be used, otherwise the text in the "Equate String" field will be used.



The list of strings shown in the **Set Equate** dialog are generated from two sources. The first source is all the currently assigned equates to the given value. The other source is all the Enum datatypes that exist in all the **open** datatype archives. If an Enum datatype exists that has a member value equal to the given equate value, then that string will be included.



The **open** data type archives contain valid enums and "fake" enums. The fake enums are created from #define values (parsed from .h files), specifically so that they will be available in the **Set Equate** dialog.



Each entry in the dialog is color-coded based upon how it is being used as an equate.

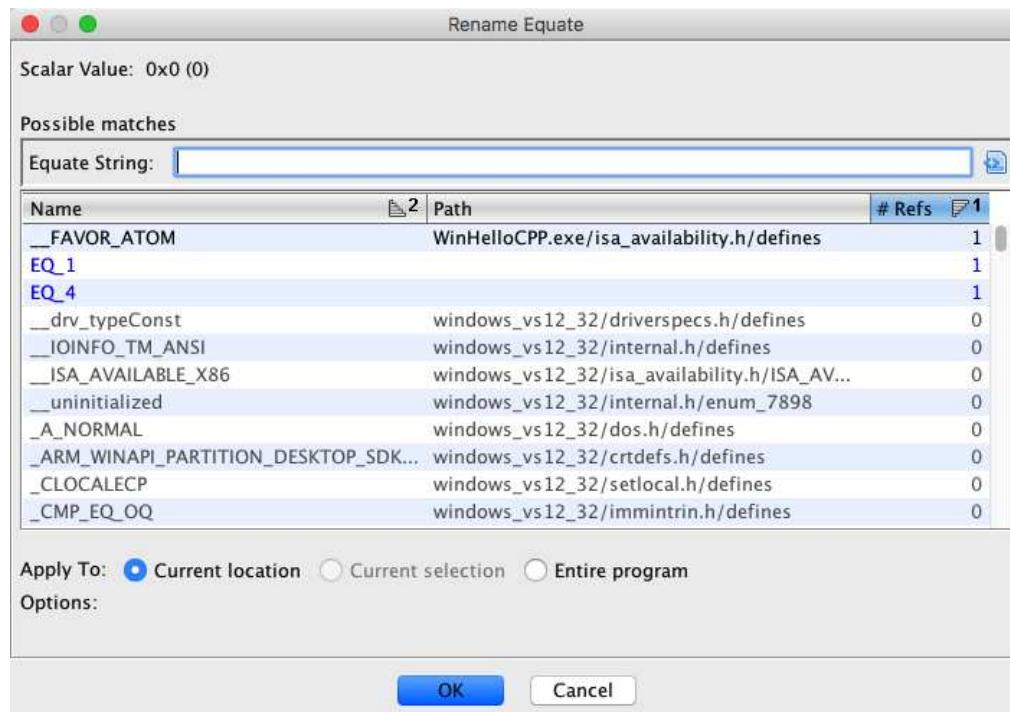
- a. **Blue**: Blue entries are existing user-defined string equates that are being used for that scalar.
- b. **Black**: Black entries are existing enum field equates being used for that scalar.
- c. **Gray**: Gray entries are suggested enum fields that have the same value as the scalar. These entries are only suggestions, and have not yet been made equates.

## Rename Equate

The *Rename Equate* action will rename one or more instances of a named Equate in the Listing.

To rename an Equate:

1. Right-mouse-click on the current Equate, and select **Rename Equate** or press <E>.



2. Select one of the choices from the "Apply To" list. **Current Location** is the default choice unless a selection is made, in which case the **Current Selection** option will be set. The other option is **Entire Program**.
  - a. **Current Location:** When selected the equate will be applied to the scalar value at the current location of your cursor only.
  - b. **Current Selection:** When selected the equate will be applied to all of the scalar values in your current selection that match the value of the scalar that you originally clicked. When you make a selection in your program this button will become enabled. If you do not make a selection then it will not be enabled and the option will be grayed out.
  - c. **Entire Program:** When selected the equate will be applied to all of the scalar values in the entire program that match the value of the scalar that you originally clicked. Scalars that already have an equate set that is different from the one you selected will not be affected.
3. Double-click an entry in the list, select an entry in the list and press **OK**, or type in the string and press **OK**. If any item in the list is selected it will be used, otherwise the text in the "Equate String" field will be used.

## Remove Equate

The *Remove Equate* action will remove an Equate(s) from a listing; effectively returning the operand to its original scalar value.

To remove references to an Equate via the context popup menu:

1. Right-mouse-click on an existing Equate, or select a group of equates and right-click on an equate within that selection, then choose **Remove Equate** or press <Delete>.
2. If you made a group selection, a confirmation dialog will appear to ensure you want to remove all equates in the selection; equates within the selection matching the one you clicked will be removed.



To remove all references of an Equate via the *Equate Table* window:

1. Select the Code Browser menu option **Window → Equates Table** to bring up the *Equate Table* window.
2. Right-mouse-click on the Equate to be deleted and select **Delete**.
3. A confirmation dialog will appear.



4. Select **Delete** to remove all references to the equate and the Equate's definition itself.

## Apply Enum

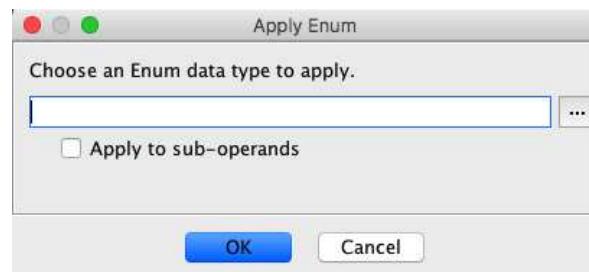
The *Apply Enum* action (only available when there is a selection), will apply enum member names to scalars in the current selection if any of the enum values match those scalars.

To apply an enum to the selection:

1. Make a selection and then Right-mouse-click, then choose **Apply Enum**.

004019aa 6a 00	PUSH	0x0
004019ac 6a 00	PUSH	0x0
004019ae 6a 00	PUSH	0x0
004019b0 6a 00	PUSH	0x0
004019b2 6a 00	PUSH	0x0
004019b4 8d 45 fc	LEA	EAX=>local_8,[EBP + -0x4]
004019b7 50	PUSH	EAX
004019b8 68 23 01 00 00	PUSH	0x123
004019bd e8 a5 fe ff ff	CALL	_CallSETTranslator
004019c2 83 c4 1c	ADD	ESP,0x1c
004019c5 8b 45 fc	MOV	EAX,dword ptr [EBP + local_8]
004019c8 8b 5d 0c	MOV	EBX,dword ptr [EBP + param_2]

2. A dialog similar to the one below should appear. Select the enum that you want to be applied to the selection. The data type must be an enum for the action to work.



- *Apply to sub-operands* – Applies the enum to scalars within operands.

Once the data type is selected, the scalars in the selection will have equates applied to them as shown below.

004019aa 6a 00	PUSH	0x0
004019ac 6a 00	PUSH	zeros
004019ae 6a 00	PUSH	zeros
004019b0 6a 00	PUSH	zeros
004019b2 6a 00	PUSH	zeros
004019b4 8d 45 fc	LEA	EAX=>local_8,[EBP + -0x4]
004019b7 50	PUSH	EAX
004019b8 68 23 01 00 00	PUSH	oneTwoThree
004019bd e8 a5 fe ff ff	CALL	_CallSETTranslator
004019c2 83 c4 1c	ADD	ESP,oneC
004019c5 8b 45 fc	MOV	EAX,dword ptr [EBP + local_8]
004019c8 8b 5d 0c	MOV	EBX,dword ptr [EBP + param_2]

## View Equates

The *Display Equates Table* action displays a window which lists all of the Equates and their references in a tabular format.

The screenshot shows a window titled "Equates Table". It has two main panels: "Equates" on the left and "References" on the right. The "Equates" panel contains a table with columns "Name", "Value", and "# Refs". The "References" panel contains a table with columns "Ref Addr" and "Op Index". A "Filter:" input field is at the bottom of the left panel.

Equates			References	
Name	Value	# Refs	Ref Addr	Op Index
EQ_1	0x0	1	0040e00d	0
EQ_2	0x4	1		
EQ_3	0x4	1		
EQ_4	0x0	1		
0x44 <BAD E...	0x44	1		
__FAVOR_ATOM	0x0	1		

The left panel, *Equates*, lists name, value, and number of references for all Equates. The right panel, *References*, lists the address and operand index of each location that references the Equate selected in the left panel. Selecting an address on the *References* panel will cause the Code Brower to go to that address in the listing. The *Equates* panel and the *References* panel can each be sorted by any column. The ascending and descending indicator displays the sort order of the information.

To view the *Equates Table* select the Code Brower menu option **Window → Equates Table** to bring up the *Equates Table* window.

You can re-order the columns in the Equates table by dragging the header to another position in the table. Sort the columns by double-clicking on the header. By default, equates are sort alphabetically. You can re-order the References table and sort by the operand index, Op Index. By default, the references are sorted by reference address in ascending order.

You can also rename the equate by double clicking in the name field and entering a new name. If the equate is based off of an enum, then an enum editor dialog will appear. Changing the matching field name will also change the equate name.



Each equate is color-coded based upon how it is being used.

- Blue:** Blue equates are existing user-defined string equates that are being used for that scalar.
- Black:** Black equates are existing enum field equates being used for that scalar.
- Red:** Red entries are bad equates. A bad equate could either mean that the enum that this equate is based off of was deleted, the field inside the enum was deleted, or the field's value was changed.

## Convert

The various convert actions are used to change the number format display of scalars displayed in the code browser. These actions are available whenever the cursor is on a number in the operand field, or the value field of a data item (byte, word, dword, qword). Note that these actions and equates are not currently supported for composite and array data. For instruction operands, the scalar number is converted visually by replacing the number with an appropriately named equate. Such a conversion can be cleared by removing the equate from the operand. For data value fields, a combination of data format settings and signed/unsigned data type alteration is used to reflect a conversion. The available formats are as follows.

### Signed Decimal

The existing scalar value will be displayed as a signed decimal number. This action is only available if the value can be interpreted as a negative value.

### Unsigned Decimal

The existing scalar value will be displayed as an unsigned decimal number. If the value would be positive even if the signed decimal format was selected, the action will simply be name **Decimal** instead of **Unsigned Decimal**.

### Unsigned Octal

The existing scalar value will be displayed as an unsigned octal number.

### Signed Hex

The existing scalar value will be displayed as a signed hexadecimal number. This action is only available if the value can be interpreted as a negative value, and is only supported on instruction operands since the data hex format currently supports unsigned rendering only.

**Unsigned Hex**

The existing scalar value will be displayed as an unsigned hexadecimal number.

**Char / Char Sequence**

The existing scalar value will be displayed as either a single ASCII character or a sequence of ASCII characters, whichever is more appropriate. Invalid and non-printable ASCII characters will be rendered in hex (e.g., \x20).

**Unsigned Binary**

The existing scalar value will be displayed as an unsigned binary number.

**Float**

The existing scalar value will be displayed as a IEEE 754 single precision floating point number. The floating point size is processor specific and will match the size of the Float data type. This action is only supported on instruction operands.

**Double**

The existing scalar value will be displayed as a IEEE 754 double precision floating point number. The floating point size is processor specific and will match the size of the Double data type. This action is only supported on instruction operands.



The convert actions also work on an instruction selection. Just make a selection then choose an operand scalar value to convert. All matching instruction scalar values in the selection will be converted.



Based upon how an instruction is implemented by its' associated language module, a hexadecimal operand which appears to be negative may in fact be a positive scalar with negative sign '-' character prepended. In such cases, the convert action may not produce the expected result.



The presence of a primary reference on an operand may prevent rendering of the converted scalar value since reference markup takes precedence over equates and data formatting.

Provided By: *EquatePlugin* and *EquateTablePlugin*

# Functions

Functions store information about locations within a program that may be referenced by a call instruction, although no direct reference to a function is required. External Functions may also be defined and associated with a Library Namespace. A function definition consists of:

- An entry point address or external location symbol
- A body of instructions (*does not apply to External Functions*)
- A function signature/prototype specification, consisting of:
  1. Function Name (*same as primary label at entry point*)
  2. Calling Convention (*available conventions defined by active compiler-spec*)
  3. Return data type (*with storage*)
  4. Parameter arguments (*with storage*)
- Optional function attributes, including:
  1. Varargs enablement
  2. No-Return enablement (*if enabled, calls to function will not return and can prevent fallthrough from call*)
  3. Inline enablement (*if enabled, callers will inline function code*)
  4. Custom Storage enablement (*if enabled, return and parameter storage to be explicitly defined*)
  5. Call Fixup
- Additional function listing markup (*does not apply to External Functions*):
  1. Local variables (*with storage*)
  2. Parameter and local variable references from instructions
  3. Code and data references from instructions
  4. Comments
  5. Optional function repeatable comment
  6. [Function Tags](#)

When displayed in the browser, a function includes:

- The entry point is usually called by another instruction, although there may be no direct reference to the function within the program. The entry point of a function must be an instruction.
- The body of the function is under user control, but can be automatically calculated when the function is defined. The body can be a contiguous range of addresses or may be multiple address ranges. Data may also be included within the body.
- The complete function signature and optional attributes are displayed within the listing at the function entry point. This information may also be displayed at pointers which reference a function provided the appropriate tool option for displaying function headers is enabled (see [Listing Fields options / Function Pointers](#)).

## Function Signature, Attributes and Variables

Please refer to [Function Signature and Variables](#) for details on this subject and how to modify a function signature/prototype specification, including function attributes and variable storage.

## Create Function

*Create Function* creates a function with an entry point and a body of instructions.

To Create a Function,

1. Place the cursor in the Code Browser at the address with a defined instruction.
2. Right-mouse-click, select the **Create Function** popup menu item



*As part of creating a function, function parameters and local variables may also be created. See [Variables](#) for the operations on variables.*



*Functions may be automatically created via [Auto Analysis](#).*



If a function starts with an unconditional jump instruction, the function will be created as a [Thunk Function](#) if possible.

The entry point for the function is the address at the current cursor location when there is no selection. With a selection, the entry point is the minimum address in the selection.

The current code browser selection is used as the function body. In the absence of a selection, *Create Function* will follow the control flow from the entry point to determine the function body. The resulting code may not be contiguous.



To see the body of the function that has been defined, place the cursor on the first instruction within the function and choose **Select ➔ Functions** from the Code Browser's main menu.

The symbol at the entry point is used as the name of the function. If no symbol exists at the entry point a default label starting with **FUN\_** is created. Prior to creating the function, the symbol may have started with **SUB\_** if it was a default symbol and there were call references to it. If a symbol does exist at the entry, a dialog is displayed so that you can change the suggested function name, **FUN\_ <address>**. After the function is created, a symbol is created with the name from the dialog.

If the symbol name is changed, the function name displayed will also change. [Rename Function](#) can be used to rename the function.

In stack-based processors, *Create Function* will try to identify parameters and local variables used by the function. By default, the variables data type will be *UndefinedN* where *N* is the size (in bytes) of the stack reference. See [Function Signature and Variables](#) on how to modify the stack variables. See [Stack References](#) on how to add stack variables.

**Select ➔ Subroutines** will display the scope of a subroutine from any address within the scope of the subroutine. It is helpful to use the **Subroutines** option to determine what the potential scope of a function would be if you create it.

## Re-Create Function

*Re Create Function* rebuilds a function's body of addresses without destroying any parameters or stack references that may have already been created. This action is useful when additional code has been found, for example from a computed jump (or switch), that was not known when the original function was created. Most likely auto-analysis will have fixed the function's body already and re-creating the function won't be necessary.

With no selection, the function's body is re-calculated based on the flow of the instructions from the function's entry point address. With a selection, the body of the function is set to the current selection.

To Re Create a Function,

1. Place the cursor in the Code Browser at the top of an already defined function.  
The cursor can be on any field at the entry point of the function.
2. Right-mouse-click, select the **Function ➔ Re-create Function** popup menu item

To Re Create a Function, with a forced new body

1. Create a selection in the Code Browser that should be the body of the function. The cursor should be at the top of the already defined function.  
The cursor can be on any field at the entry point of the function.
2. Right-mouse-click, select the **Function ➔ Re-create Function** popup menu item



Recreating a function will kick off auto-analysis on the function if there are any changes to the function's body. New parameters or locals may be created since more code may now be part of the function's body. See [Variables](#) for the operations on variables.

## Thunk Functions

*Thunk Functions* are a common artifact of compiled code and are frequently used to facilitate access to external functions, functions located far from the caller, and other relocation scenarios. Ghidra has the ability to specify a

function as being a *thunk* for another function. A *thunk* has the same function signature and parameter storage as the real function (also referred to as the *thunked-function*), although its name may differ. A *thunk* function within Ghidra acts as a proxy to the real/thunked-function where all parameter and attribute changes to one are reflected onto the other. One exception to this is the name. If a *thunk* is created without a name, its name will reflect the name of the *thunked* function. Renaming the *thunk* allows the thunk to have a name which differs from the thunked-function. Local variables are not supported for *thunk* functions.



Within the Code Browser, double-clicking on a *thunk* function name will navigate to the associated *thunked* function, while *thunked* functions will display back-references (i.e., XREFs) to the associated *thunk* functions with a Ref-Type of 'T'.

To Create a Thunk-Function:

1. Select the instructions which corresponds to the body of the new thunk function, or place your cursor on a single unconditional jump instruction which jumps to the thunked-function.
2. Right-mouse-click, select the **Create Thunk Function** popup menu item
3. If unable to determine the thunked-function, the user will be prompted to specify the thunked-function by label or address. The specified location must correspond to an existing function.

To Edit a Thunk Function (i.e., set the associated *thunked* function) or Convert a normal Function to a Thunk Function:

1. Place the cursor in the Code Browser at the top of an already defined *thunk* function.  
The cursor can be on any field at the entry point of the function.
2. Right-mouse-click, select the **Function ➔ Set Thunked Function...** popup menu item
3. The user will be prompted to specify the *thunked* function by label or address. The specified location must correspond to an existing function.

To Revert a Thunk Function (i.e., revert a Thunk Function to a normal Function):

1. Place the cursor in the Code Browser at the top of an already defined *thunk* function.  
The cursor can be on any field at the entry point of the function.
2. Right-mouse-click, select the **Function ➔ Revert Thunk Function...** popup menu item
3. The user will be prompted to confirm the action.

## External Functions

Defining an *External Function* allows a function to be defined which does not reside within the current program listing or whose actual memory address is unknown. Similar to a simple *External Location*, these external symbols are associated with named *Library* namespaces and are most easily managed via the [Symbol Table](#) or [Symbol Tree](#) under the *Imports* category. If the actual *Library* name is unknown, the "<EXTERNAL>" *Library* (or any other named *Library*) may be used as a parent namespace.

From either the [Symbol Table](#) or [Symbol Tree](#), an existing *External Location* may be converted to a function using the *Create External Function* popup action on the selected node. The resulting *External Function* may be converted back to a simple *External Location* by deleting the function node. To really remove the function and its location will require a second delete on the *External Location*.

From either the [Symbol Table](#) or [Symbol Tree](#), an existing *External Function* may be modified using the **Function ➔ Edit Function...** popup action on the selected function node.

Creating an [External Reference](#) is currently the only mechanism within the Ghidra GUI to establish an *External Location*. Once an *External Location* has been established, it can be converted to a function (see above). This limitation should hopefully be resolved in a future release of Ghidra.

## Create Multiple Functions

*Create Multiple Functions* creates functions from a selection in the listing. It works from the minimum address to the maximum address in the selection trying to create functions if possible. Any addresses that are already part of a function are discarded and not used to determine new functions. Also whenever a function is created by this action, all the addresses in the body of the created function are also discarded from being possible addresses for starting a new function.

A common use of this action is on a selection containing the entry point addresses of the functions you want to create.

## Edit Function

For information on editing functions, see [Function Signature Help](#).

## Rename Function

*Rename Function* renames an existing function. As discussed in [Create Function](#), the function name is the same as the primary label at the functions entry point.

To rename a function,

1. Right-mouse-click on the function header in the Code Browser
2. Select the **Function → Rename Function** popup menu item
3. Enter the new function name and/or namespace, click OK. The name may also be entered with a fully qualified namespace (e.g., mynamespace::myfunction). The `::` is used as a namespace delimiter.

## Delete Function

*Delete Function* removes a function. There is no confirmation for the Delete Function operation. However, the results can be undone using the [Undo operation](#).

When a function is deleted all stack variable definitions are removed, along with all references to those variables from instructions within the function's body. If a stack reference refers to a stack variable that is deleted, any references will be replaced with **Stack [offset]**, where offset is the relative offset to the stack.

To Delete a Function,

1. Right mouse-click on the function header
2. Select the **Function → Delete Function** popup menu item

When a function is deleted, all stack and register references from instructions within the function body are removed. The function comment (which is really the [plate comment](#) for that address) remains intact if you had made changes to it, or if the plate comment existed before the function was created.

If there are still *call* references to this address, the label changes from **FUN\_** to **SUB\_**.

## Function Purge

A **function purge** is the number of additional bytes (not including the return value) a function pops from the stack when it returns. The value is calculated as the difference between the stack pointer's value exiting the function and its value coming into the function but excludes the final pop of the return address.

For most calling conventions, the function purge is always zero. A major exception is the 32-bit x86 *stdcall* calling convention, where the function may pop off its own stack parameters in addition to the return value. The function purge in this situation can be positive indicating that more values are popped from the stack. For other unusual situations, a negative function purge can be set indicating that the function *pushes* additional values.

For architectures where the stack grows in the *positive* direction, the meaning of the function purge sign is reversed. A positive function purge indicates additional bytes are *pushed* to the stack, and a negative function purge indicates bytes are *popped* from the stack.

To change the function purge:

- Right mouse-click on the function header
- Select the **Function → Edit Function Purge...** popup menu item
- Enter the new function purge size in the dialog that appears

## Function Repeatable Comment

When a repeatable comment exists at the entry point of a function, the repeatable comment is displayed in the *Function Repeatable Comment* field rather than the *EOL Comment* field. See [Edit Comments](#) for more information

on comments.

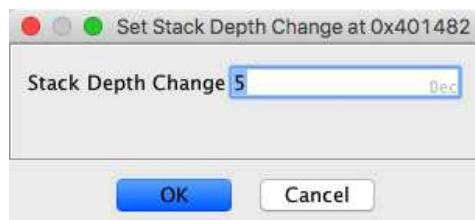
## Stack Depth Change

You can specify a relative change in the stack depth at the address of the current location in the program.

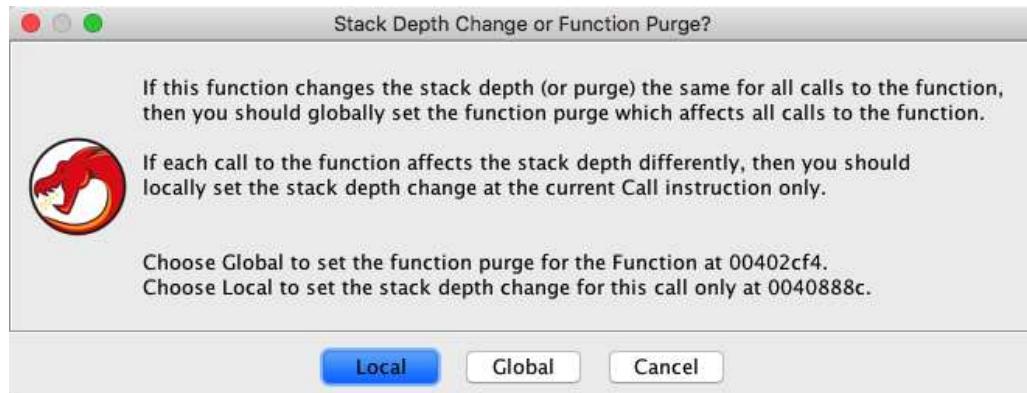
### Set Stack Depth Change

To set a change in stack depth:

- Right mouse-click on the Listing.
- Select **Set Stack Depth Change...** from the popup menu.
- The *Set Stack Depth Change* dialog is displayed. The Stack Depth Change textfield initially contains the current stack depth change value. If the stack depth change is not explicitly set at this address, the default value will be based on the instruction. For a call instruction, the default stack depth change will be based on the function purge value of the called function.



- Enter the desired change in stack depth. This can be either decimal or hexadecimal. Hexadecimal is indicated by a "0x". For example, -0x1a.
- Press the Return key or the **OK** button to set the stack depth change.
- If you are not on a Call instruction, the stack depth change will be set. Otherwise, you will see a dialog allowing you to choose whether the value should be applied as a stack depth change at the current address (**Local**) or as a function purge at the called function (**Global**). Choose **Local** to set the stack depth change or **Global** to set the function purge.



### Remove Stack Depth Change

To remove a change in stack depth where it is currently set:

- Put the cursor location on the *StackDepth = StackDepth + ...* line in the Listing.
- Press the **Delete** key.

or

- Right mouse-click on the *StackDepth = StackDepth + ...* line in the Listing.
- Select **Remove Stack Depth Change** from the popup menu.

Provided By: *Functions* plugin

Related Topics:

- [Function Signature and Variables](#)
- [Auto Analysis](#)
- [Stack References](#)
- [Comments](#)

# Function Signature, Attributes and Variables

## Function Signature

A function's signature conveys the following information about a function:

1. The compiler-specific named calling convention
2. The function's return data type (void indicates no return value)
3. The function's name
4. Ordered list of named parameters and associated data types.

## Function Attributes

The following function attributes affect disassembly and semantic analysis and may be set via the [Edit Function](#) dialog.

### Custom Storage

The *Custom Storage* option, if enabled, provides the ability to explicitly specify return/parameter storage. By default, storage will be dynamically computed based upon calling convention and return/parameter data types. When dynamic storage is computed, hidden "auto-parameters" may be injected as well as the use of "forced-indirect" storage.

### Inline

Any *Inline* function called by another function may be treated as inline code instead of a function call during analysis.

### No Return

While a typical function call is always assumed to return and continue flowing to the next instruction, marking a function as *No Return* forces an implied return immediately following a call to such a function. Depending upon the state of disassembly, marking a function as *No Return* may help to prevent a call to such a function from falling-thru to the next instruction during disassembly. If disassembly has already been done and the fall-thru has been improperly disassembled, the [Clear Flow and Repair](#) action may be used to cleanup the bad fall-through.

### Varargs

Designated by a trailing '...' in the function signature, indicates that a variable number of parameters is allowed. Common C functions which employ Varargs are *printf* and *scanf*.

### Call-Fixup

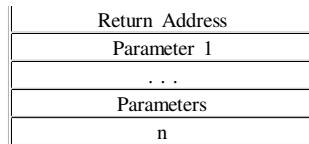
A function may be tagged with a predefined *Call-Fixup* which can be used to alter/simplify the semantic effect of calling such a function. The available set of predefined call-fixups are defined within the compiler specification (\*.cspc file) associated with a program. This feature is typically used when the effects of calling a well-known function need to be simplified so that the caller can be more easily analyzed and/or understood.

## Variables

There are two classes of function variables: Parameters and Local Variables. The term Parameter also includes the Function return value which has an associated data type and storage but has no ordinal.

Many processors use a stack as part of their calling conventions and/or for local variable storage. In some cases, the return address is pushed onto the stack during the call operation. Any references to the stack (e.g., references and variable storage) are relative to the stack pointer when the function is entered. For a negative growing stack (like the X86 processor), the stack would look something like this:

-n
...
Local Variables
...
Saved Registers



Structures, Unions, and Arrays can be used to define variables as well as the built in primitive [data types](#). To define structures, see [Data Structure Editor](#)

### Parameters (includes Function Return)

Storage for return/parameters can be computed dynamically based upon the associated calling convention and data types (i.e., complete function signature). The ability to properly compute the correct storage is limited by the current capabilities of the function prototype models and Decompiler within Ghidra. In some cases it will be necessary to enable Custom Storage for a function and specify the correct storage locations for the return/parameters including the ability to join multiple storage elements (i.e., varnodes) for a single parameter.

When parameter storage is determined dynamically, the calling convention may dictate the use of hidden parameters which we refer to as "*auto-parameters*" (**auto**). In addition, the "*forced-indirect*" (**ptr**) condition may be imposed on parameters when large data is forced to be passed by reference. When either of these is in use for the return or parameters, the (**auto**) or (**ptr**) designation will appear with the storage location displayed for each parameter. The parameter names '**this**' and '**\_return\_storage\_ptr\_**' are reserved for the two supported auto-parameter cases. The '**this**' *auto-parameter* is imposed by the *thiscall* calling convention, while *forced-indirect* is imposed by certain calling conventions which limit the maximum size of any parameter passed by value. If *forced-indirect* is imposed on the return storage, the '**\_return\_storage\_ptr\_**', *auto-parameter* will be imposed to allow the function caller to specify a pointer to the full return storage data location. When using custom storage, it is assumed any *auto-parameters* will be explicitly defined as normal parameters if applicable. Within the Program API, *auto-parameters* may not be directly manipulated and are immutable.

The [Function Editor](#) is the most affective means of modifying the function signature either via the Code Browser listing or within the Decompiler

### Local Variables



*Currently, there is no specific user interface action for creating Local Variables. Stack and register variables will be created automatically when a suitable stack or register reference is created via the user interface. Additionally, the Decompiler's commit actions will create Local Variables as needed. They can also be created programmatically.*

In addition to register and stack locations, Local Variables also have the ability to be defined by the Decompiler to reflect temporary storage identified by a hash value. Please note that these hash type variables can sometimes not work as expected within the Decompiler. In addition, the scope of a local register variable is determined by its *first-use-offset* which reflects the instruction offset relative to the function entry point at which the variable is first written. The variable will remain in scope from that point forward until another local variable comes into scope for the same storage location. Local stack variables assume a *first-use-offset* of zero (0).

## Define Variable Data Type or Function Return Type

Variables can be annotated with one of the [built in](#) or [user defined data types](#). The undefined or previously defined variable will be redefined to the new data type.

Variable data types, including parameters and return, can be defined one of four ways:

- Using right-mouse-click on the return type or a parameter within a function signature displayed within the listing.

1. Right mouse click on the return type or parameter within the function signature
2. If the cursor is over the return type, Select **Set Data Type** ; if the cursor is over a parameter, select **Set Data Type**
  - The pull right menu lists data types that you have marked as "[favorites](#)."
  - After you apply a data type, this becomes your [most recently used data type](#) and is shown on the menu with the 'Y' as the "hot key."

- Using right-mouse-click on a parameter, <RETURN> or local variable listed within the function variable listing

1. Right mouse click on a parameter, <RETURN> or local variable
2. Select **Set Data Type**
  - The pull right menu lists data types that you have marked as "[favorites](#)."
  - After you apply a data type, this becomes your [most recently used data type](#) and is shown on

the menu with the 'Y' as the "hot key."

- Using the [Data Type Manager](#) window (drag and drop)

1. From the Code Browser tool bar select Display Data Types 
2. In the *Data Type Manager* window select the appropriate data type
3. Drag and Drop the data type onto the target parameter, <RETURN> or local variable in the Code Browser

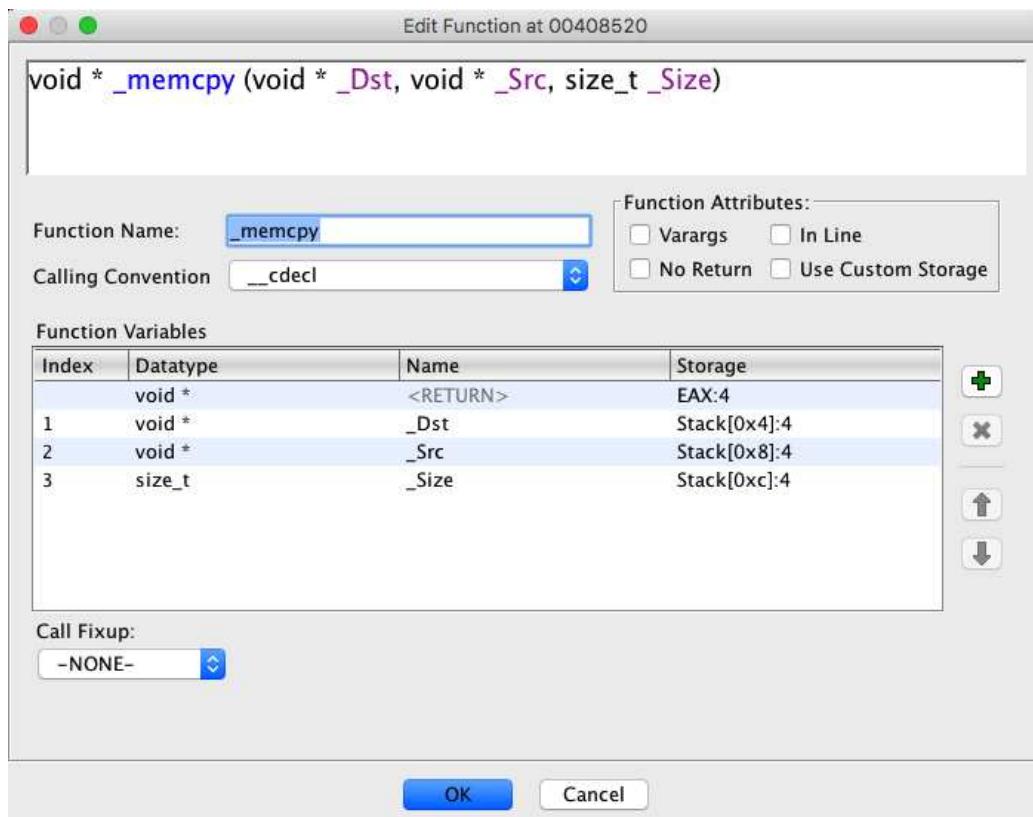
- [Assigned Quick key](#),

- Place the cursor on the target parameter, <RETURN> or local variable
- Press a quick key set to specify a data type (i.e., "b"-byte, "p"-pointer, ...)

## Edit Function

Once a function has been created, there are many attributes of the function and its parameters that can be changed. The **Function Editor Dialog** allows you to make those changes. To edit a function do the following:

1. Place the cursor on a function signature
2. Right-mouse-click, and select *Edit Function*.
3. Edit any attributes of the function using the dialog.
4. Press *OK* to save your changes.



### Function Signature Field

The area at the top of the dialog is used to show the complete function signature. It will update as you make changes to the various fields in the dialog.

You may also use this field to directly edit the signature, but **beware** that your changes will have to be parsed and the current parser is severely limited. Once you make a change in this field, all the other fields will be disabled temporarily until you complete your changes and press either <TAB> or <RETURN> to continue. You may also complete the edit by clicking outside of the signature field.

If the parser fails to successfully parse your changed signature field, a dialog will appear giving you the option to continue typing in the field or aborting your edits in that field.



*Due to limitations in the parser, there are many function signatures that Ghidra supports that you cannot directly enter by typing in the function signature field. For example, you cannot use the signature field to enter templated types. Also, the parser currently only supports common datatypes and datatypes that are currently used in your program. To enter more complicated values or find datatypes from open archives, use the more precise controls that the dialog provides.*

### Function Name

This text field can be used to change the name of the function.

### Calling Convention

This field is a combobox that allows you to choose a calling convention from the list of known calling conventions for this processor and compiler specification. This choice will have no affect on storage if the **Custom Storage** checkbox has been selected.

### Function Attributes

This sections contains a set of miscellaneous checkboxes that affect the function.

- **Varargs** –sets the function to have a variable number of arguments.
- **In Line** –indicates this functions code is placed in line with the calling function.
- **No Return** –used to indicate if this function does not return.
- **Use Custom Storage** –If selected, the user can edit and change the storage of the return value and the parameters. Otherwise, the storage is determined by the selected calling convention.

### Parameters/Return Type Table

The parameters/return type table allows the user to add or remove parameters as well as changing their names and datatypes. It also displays the return value datatype and storage. Also, if the **Use Custom Storage** checkbox is selected, the storage of the parameters and return type can be changed.

#### Table fields

- **Index** –indicates its ordinal position in the signature (starts with 1). This field can't be edited directly, but can be affected by the **Up** and **Down** buttons. Note: this field is blank for the return value.
- **Datatype** –indicates the datatype of the parameter or return type. Clicking on this field will bring up a [DataType Chooser Dialog](#).
- **Name** –the name of the parameter. This field can be edited directly in the table cell. Note: the name of the "return value" is <RETURN> and can't be changed.
- **Storage** –the storage for the parameter or return value. If the **Use Custom Storage** checkbox is selected, this field can be edited by clicking on it and bringing up the parameter editor dialog. If not using custom storage, *auto-parameters* or *forced-indirect* storage may be imposed as determined by the selected calling convention and is designated by **(auto)** or **(ptr)** with the displayed computed storage.

#### Table Buttons

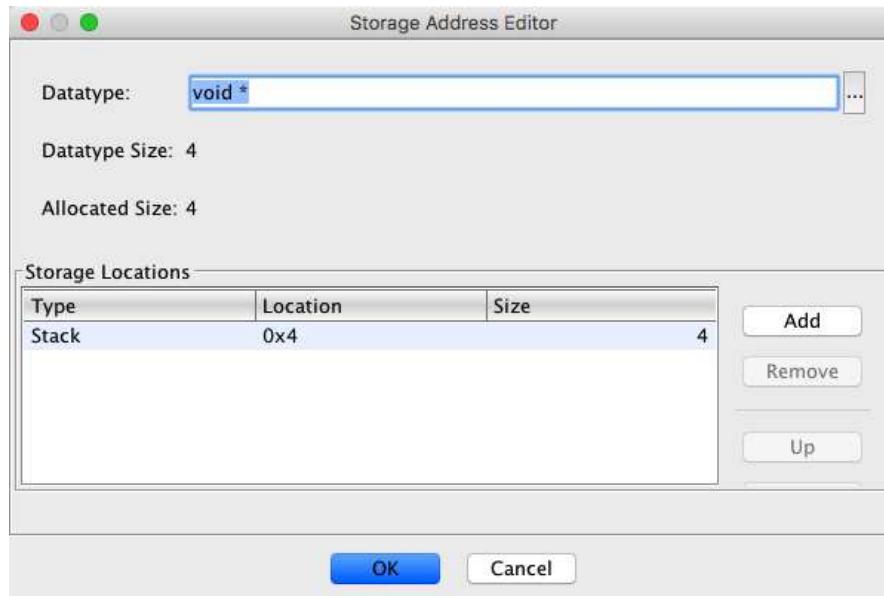
- **Add** –Adds a new parameter to the function
- **Remove** –Removes the selected row (parameter) from the table.
- **Up** –Moves the selected parameter earlier in the signature.
- **Down** –Moves the selected parameter later in the signature.

### Call Fixup

This field is a combobox that allows use to use a predefined Call-Fixup. A function may be tagged with a predefined Call-Fixup which can be used to alter/simplify the semantic effect of calling such a function. The available set of predefined Call-Fixups are defined within the compiler specification (\*.cspc file) associated with a program. This feature is typically used when the effects of calling a well-known function need to be simplified so that the caller can be more easily analyzed and/or understood.

## Edit Parameter Storage Dialog

This dialog is invoked by clicking on the storage column in the [Edit Function](#) dialog. This dialog allows you to precisely specify the storage of a parameter. The parameter can even be divided amongst multiple storage locations. Each row of the table specifies a storage location used by the parameter.



### [Size Information](#)

The top of the dialog shows two sizes. The **Datatype** size is the size required to store the parameter based on its current datatype. The **Allocated Size** shows how much storage as been allocated based on the rows in the table.

### [Storage Table](#)

A table of storage locations where each row represents a storage location. You must add enough storage location rows to get enough storage space for the size of the parameters datatype.

#### *Table Columns*

- **Type** –the type of storage. Can be either Stack, Register, or Memory. Clicking on the field will bring up a table version of a combo-box.
- **Location** –Indicates the specific location for the type. For stack, it will be an integer offset. For register, it will be the name of the register. For memory, it will be the address. Clicking on this field will bring up an editor appropriate for the storage type.
- **Size** –The size for this storage. For stack and memory, it will be the number of consecutive bytes to use for this storage. For a register, it will the number of bytes to use within the register, up to the size of the register.

#### *Table Buttons*

- **Add** –Adds a new storage location
- **Remove** –Removes the selected row (storage location) from the table.
- **Up** –Moves the selected storage location earlier in the allocation.
- **Down** –Moves the selected storage location later in the allocation.

## Create Function Definition

Once you have defined a function, you can make a function signature definition which is a new data type that can be applied to another function so that it has the same signature. The data type appears under the [program\\_node](#) in the [Manage Data Types](#) window.

To create a function definition, position the cursor on a function signature, right mouse click and select *Create Function*

#### *Definition.*

A new data type is created; the name of the data type is the same name as the function.

To create a new function signature definition using the one you created, drag the data type from the *Data Type Manager* window and drop it on the existing function where you want the new function signature to be created.



If you attempt to create a function definition on one that you have already defined, nothing happens.

## Rename Variable

**Rename Variable** will change the name of a variable from its default name to a user-defined name.

To Rename a Variable,

1. Place the cursor on the target variable within the function variable listing
2. Right-mouse-click, select **Function Variables**→**Rename Parameter...** or **Function Variables**→**Rename Local Variable...**
3. Type the new variable name in the dialog and press<Enter>, OR click on the **OK** button

## Delete Variable

**Delete Variable** will remove the target variable from the listing. There is no confirmation with *Delete Variable*. However, the operation can be undone using the [Undo operation](#).

To delete a variable,

1. Place the cursor on the target variable within the function variable listing
2. Right-mouse-click, select **Function Variables**→**Delete Parameter** or **Function Variables**→**Delete Local Variable**

## Edit Comment

Stack Parameters and Local variables can have comments associated with them. The comment is free form text. If a comment already exists, the comment is modified.

To add/edit a comment to a variable,

1. Place the cursor on the target variable
2. Right-mouse-click, select *Edit Comment*
3. Enter the comment.
4. Select *OK*

## Remove Comment

To remove a function variable comment,

1. **Place the cursor on the variable comment**
2. Hit the <Delete> key

## Recently Used Data Type

The data menu shows an option for the data type that was most recently used. By default, the "hot key" assigned to this option is 'y,' however, you can change the key assignment through the [key bindings panel](#) on the [Edit Options dialog](#).

Related Topics:

- [Data Structure Editor](#)
- [Data Type Manager](#)
- [Functions](#)

# Function Tag Window

The Function Tag window provides a list of function tags defined in the currently open program. It will also show which tags are assigned to the currently-selected function. Tags may be created by the user, or loaded from a predefined set (see section on [loading tags](#) below).

To display the Function Tag window, select the **Window → Function Tags** option on the tool menu. Optionally, the dialog may be activated by right-clicking on a function header in the listing and selecting the **Edit Function Tags** option.



## Window Components

This window has five distinct sections:

- **Available Tags List:** Displays all tags that are available to assign to the current function.
- **Assigned Tags List:** Displays all tags that have been assigned to the current function.
- **Tag Input Field:** Allows users to create new tags. Multiple tags may be created at one time.

**Create new tag(s):** tag 1, tag 2, ...

- **Filter Field:** Allows users to filter what is shown in the Available and Assigned tag lists.

**Filter:**

### ● Action Buttons

- Assigns the selected tag(s) to the current function
- Removes the selected tag(s) from the current function
- Deletes the selected tag(s) from the program and removes them from all functions

## Tag Operations

### Create

Tags can be created by using the *Tag Input Field* described above. Users may enter multiple tag names delimited by a comma. All newly-created tags will be displayed in the Available Tags List and are NOT assigned to any function.



Each tag may have an associated comment that is visible as a tooltip. This can be assigned after the tag has been created (see [edit](#) section below).

### Delete

Tags may be deleted by selecting a set of tags and pressing the  icon. Users will be prompted with the following:



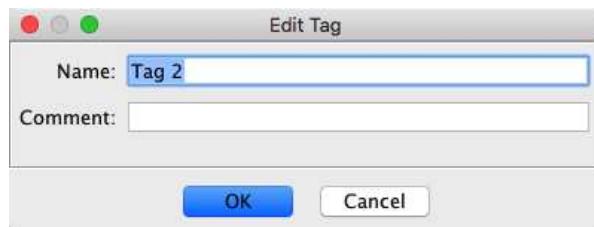
If confirmed, the tag will be removed from the system and from all functions to which it had been assigned.

### Edit

Tag names and comments may be edited by double-clicking the item in the list. If the tag is not editable the user will be presented with the following warning:



If editing is allowed, the following dialog will be shown:



### Add to Function

Tags may be added to a function by selecting a set of tags and pressing the  button. The tags will be removed from the Available Tags List and added to the Assigned Tags List.

### Remove from Function

Tags may be removed from a function by selecting a set of tags and pressing the  button. The tags will be removed from the Assigned Tags List and added to the Available Tags List.

## Loading External Tags

Tags may be loaded on startup from an external source if desired. These will be shown in blue and cannot be edited or deleted, with one caveat: once a tag has been assigned to a function it ceases to have any special protections and can be edited/deleted just as any other.

To make these available there must be a file named *functionTags.xml* available on the classpath. Edit (or create) this file and add tags as needed. The format is as follows:

```
<tags>
  <tag>
    <name>TAG1</name>
    <comment>tag comment</comment>
  </tag>
  <tag>
    <name>TAG2</name>
```

```
<comment>tag comment</comment>
</tag>
</tags>
```



Be aware that any external tags that have removed/edited will reappear with Ghidra is restarted, as these are always loaded from this file.

Provided By: *FunctionTagPlugin*

Related Topics:

- [Functions](#)

# Register Values

Each program has a set of registers that is determined by the program's *language*. When a program is actually running, the registers contain values that make up the *processor state* at any given instant. Since Ghidra programs represent a *static* view (i.e. not running), there is no "instant" that can be examined to see register values. The best approximation is to assign registers values at addresses. This is useful if the register value can be determined to be constant anytime the instruction at that address is executed.

For example, if an instruction at address 0x1000 is "mov ax,20", then it might be possible to assume that the value of register ax will be 20 for the next several instructions (assuming there are no "jumps" into that code).

Setting register values can sometimes be the critical link for successfully performing various types of analysis or even getting the correct disassembly. For example, some processors have a "mode" that is stored in some register. Depending on the mode, the processor may have completely different instruction sets. To disassemble properly, the mode register must be set at the address where the disassembly begins.

## Register Manager

The *Register Manager* displays the assigned values of registers at addresses within a program.

To display the *Register Manager*, select the **Register Manager** menu item from the **Window** menu of the toolbar or.

The screenshot shows the 'Register Manager' window. On the left is a tree view labeled 'Registers' containing categories like CONTROL, DEBUG, FLAGS, FPU, ST, XXM, and various BND registers (BND0 to BND3) which further expand into BNDCFGS, BNDCFGU, and BNDSTATUS. Below the tree is a 'Filter:' text input field. On the right is a table with columns 'Start Address', 'End Address', and 'Value'. Two rows are visible: one for BND0 (addresses 00401000 to 00401013, value 0x23) and one for BND3 (addresses 00401c37 to 00401c5a, value 0x56).

	Start Address	End Address	Value
	00401000	00401013	0x23
	00401c37	00401c5a	0x56

The left side of the Register Manager is a tree containing all the registers defined for the program's language. If the registers have been grouped into categories by the language, those registers will appear under a folder with that group name. Registers that break down into smaller registers are marked with the icon. These nodes can be further opened to reveal their component registers. For example, EAX can be opened to show AX, which can then be opened to show AL and AH. Registers that don't have sub pieces use the icon. The filter text field, located under the register tree can be used to quickly find any register. As you begin to type the name of a register, the tree will shrink eliminating any registers that don't contain the filter text.

The right side of the Register Manager is a table that displays ranges of addresses that have assigned values for whichever register is selected in the register tree. Initially, the table only contains values that have been explicitly associated with the selected register and addresses. There is an option in the drop down menu that will cause the table to also show *default* values for a register. Default values are assigned by the language and usually only apply to context registers.

## Tool Buttons

Toggles whether or not to select the row in the currently selected register value table whose address range contains the current address of the cursor in the listing view. For example, in the Register Manager image shown above, if the user clicks on any address between 804c12 and 804c24, then the first row of the table will

be selected if this action toggle is on.

Deletes the register value associations for all the selected ranges in the table.

Creates a selection in the browser for all the address ranges selected in the register values table.

Filters out all registers in the register tree that don't have any associated values (default or otherwise).

## Menu Actions

**Show Default Values** –if selected, the register manager will show default register value ranges mixed in with user set register value ranges.

## Editing an Address Value Range

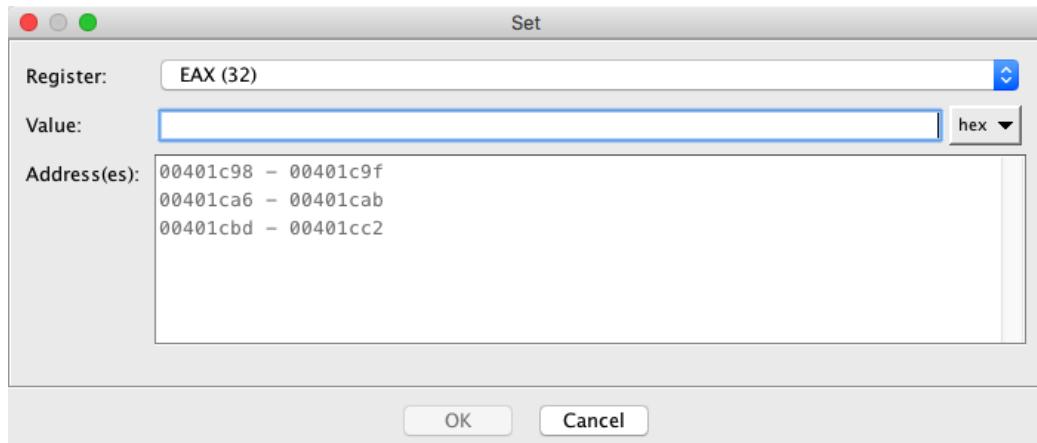
Double click on any row in the register value table to bring up the *Edit Register Value Range* dialog.



Using the dialog, you can adjust the start or end address and/or change the value associated with the range. If you change the start or end address such that the range is smaller, then value associations for address range that was truncated is effectively cleared. For example, in the dialog shown above, if you change the end address to 01001b47 and change the value to 111, then addresses 01001b33 to 01001b47 will have the value 111 and address 01001b48 will have no value.

## Setting Register Values Over Address Ranges

To set a value for a register across an address range, first create a selection in the browser and then invoke the *Set Register Values* action by either using the right-mouse popup or using the <Ctrl>R quick key. The following dialog appears.



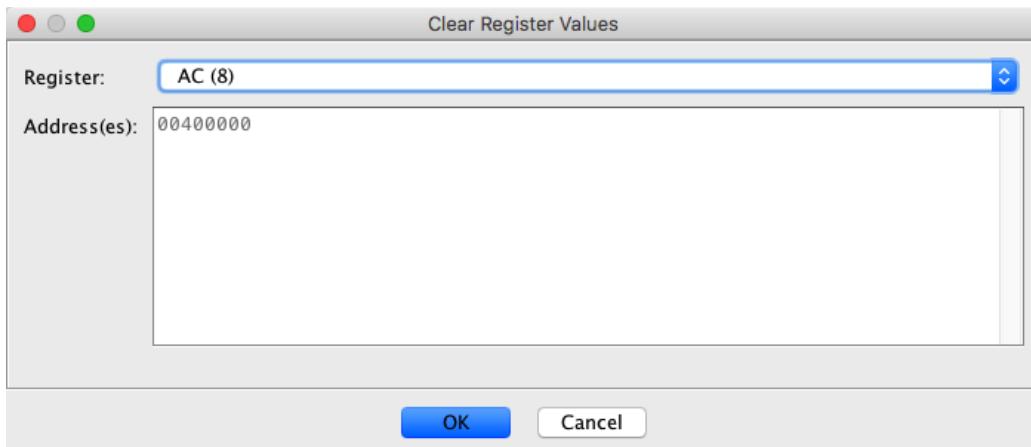
Choose the register for which you want to set a value, enter the value and press the *Ok* button. The *Address(es)* field shows

the set of addresses that will be affected. By default, the *Value* field is entered as an unsigned hex value, but the format can be changed using the adjacent drop-down menu.

 If the browser cursor is on a register when this dialog is invoked, then the register combo will be set to that register.

## Clearing Register Values Over Address Ranges

To clear a register value over an address range, first create a selection in the browser and then invoke the *Clear Register Values* action by using the right-mouse popup. The following dialog appears.



Choose the register for the association and press the *Ok* button. The *Address(es)* field shows the set of addresses that will be affected.

## Delete Existing Register Value Ranges Associations

You can delete associated register values using *Delete Register Value Range* action when over a **Register Transition** field in the browser ("assume ESI = 0x20".) Activating this action will remove the register association over the entire range associated with that "Assume" statement.

Provided by: *RegisterPlugin*

## Bookmarks

Bookmarks are used to flag addresses of interest in a Program. Each Bookmark consists of an address, a type, a name, a category (optional), and a description (optional). Bookmarks may be organized using the category field. Ghidra places various Bookmark icons in the [Marker Margin](#) of the [Code Browser](#) to indicate locations of defined bookmarks. The tooltip (shown when the mouse hovers over the Bookmark icon in the Marker Margin) shows the Bookmark's type and comment.

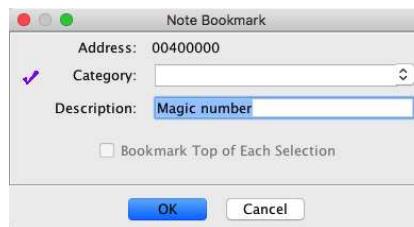
The type refers to how the bookmark was added. Ghidra supplies five types of bookmarks:

Bookmark Types		
Type (icon)	How Bookmark is Added	Navigator Color
Note ✓	Added via the <a href="#">Note Bookmark dialog</a> ; Notes are intended to be user-defined only.	Purple
Info ⓘ	May be added by a plugin to mark an address of interest.	Cyan
Analysis ⚙	Added during the <a href="#">Auto Analysis</a> process. Indicates automatic changes which have been made to the program (e.g., code found, address tables, etc.).	Orange
Error ✖	Added by the <a href="#">Disassembler</a> or <a href="#">Auto Analysis</a> process when an unexpected condition is identified at a specific address (e.g., bad instruction).	Red
Unknown ?	Represents a custom Bookmark type which was added by a plugin not currently configured into the tool. A properly designed plugin will assign a custom icon and color to its custom type.	Magenta

Ghidra also places a marker for the bookmark in the [Navigation Margin](#) of the Code Browser. Clicking on the Navigation Margin causes the Code Browser to go to that address, and centers it in the browser.

The following paragraphs describes the [Bookmarks window](#), and how to [add](#) and [remove](#) bookmarks.

### Add a Bookmark (in the CodeBrowser)

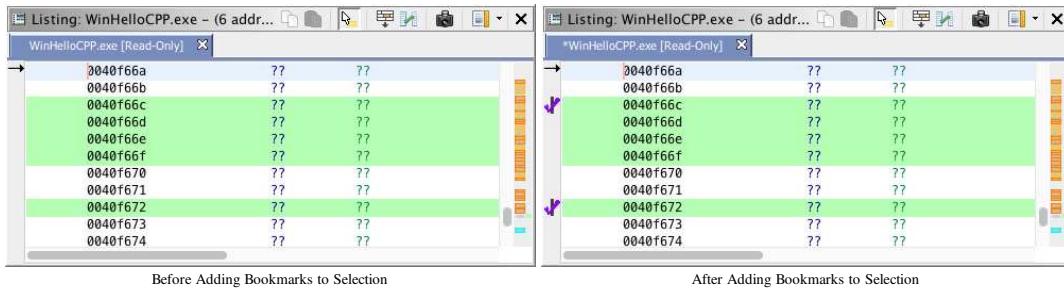


To add a Note Bookmark,

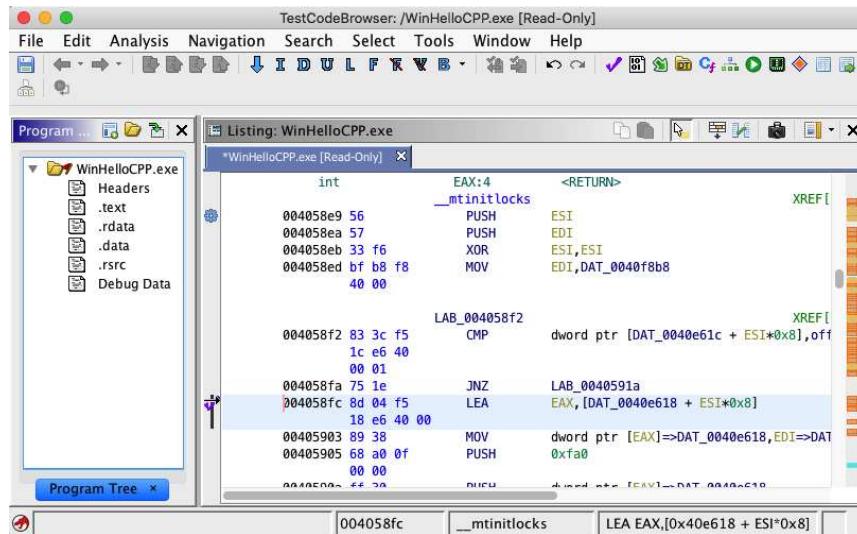
1. Position the cursor at an address
2. **Right-mouse-click** in the Code Browser, select **Bookmark**
3. The **Address** field on the **Note Bookmark** dialog shows the location where the bookmark will be added
4. Enter a **Category**(optional) or choose an existing one from the combo box
5. Enter a **Description** (optional); if an end of line comment exists at this address, then this text becomes the default description of the bookmark, as shown in the image above.
6. Click on the **OK** button.



When adding a bookmark with multiple selections in the Code Browser, the **Bookmark Top of Each Selection** checkbox is both enabled and selected by default. Leave the checkbox selected to create a bookmark at the start of each address range in the selection. Deselecting the checkbox will cause the bookmark to be created at the cursor location.



The following image depicts the Code Browser with Bookmarks. Notice the checkmarks in the [Marker Margin](#) on the left and the markers in the [Navigation Margin](#) on the right.



### Remove a Bookmark (in the CodeBrowser)

To remove a bookmark,

1. Position the cursor on the address of the bookmark to be deleted
2. From the **Marker Margin**, right-mouse-click, select **Delete Bookmark** → <type>: <description>

### Bookmarks Window

The **Bookmarks** window lists all of the bookmarks within a Program, including the bookmark type, category, description, address, label, and code unit where the bookmark was placed. Click on a **Bookmark** to navigate to the selected address in the **Code Browser**.

Type	Category	Description	Location	Label	Code Unit
Analysis	Found ...	Found code from o...	00401...	LAB_004010a0	MOV EAX,d...
Analysis	Found ...	Found code from o...	00401...	LAB_00401837	PUSH ESI
Analysis	Found ...	Found code from o...	00402f...	LAB_00402fb5	CALL dwor...
Analysis	Found ...	Found code from o...	00403...	LAB_00403174	PUSH 0x8
Analysis	Found ...	Found code from o...	00405...	LAB_00405483	PUSH ESI
Analysis	Found ...	Found code from o...	00406...	LAB_00406a2c	MOV ECX,d...

To display the **Bookmarks** window, click the bookmark icon in the Code Browser toolbar, or select the **Window** → **Bookmarks** option from the menu.

Each of the columns may be sorted by clicking on the header. The sort graphic illustrates which column is being sorted on, and whether it is ascending ( ) or descending ( ). In the image above, the *Preview* column is sorted in ascending alphabetical order. By default, the bookmarks are sorted in ascending order by the *Type* column.

In the Bookmark table, only the *Category* and *Description* columns are editable. To edit entries in these columns, double-click on the appropriate cell and begin typing. Click outside of the cell to apply the changes. When the *Category* column is being edited, it shows a combo box, listing all of the categories. Choose an existing category or enter a new one. If a new category is entered, the combo box is updated.



The list of Bookmarks displayed can be filtered by clicking the button in the toolbar of the Bookmarks window. The displayed bookmarks will correspond to the selected checkboxes in the *Bookmark Filter* dialog.

You may also filter the contents of the bookmark table by using the [filter text field](#).

The following describes the features available from the Bookmarks window (Note: some of these features are also available from inside the CodeBrowser):

#### Edit Category

1. Double click in a *Category* cell to display the cell editor.
2. Click on the down-arrow button in the cell editor to display the list of categories.
3. Select a category from the list OR enter a new category in the cell editor.
4. Press the <Enter> key or click outside of the editor.

#### Edit Description

1. Double click on a *Description* cell to display the cell editor.
2. Enter a new description.
3. Press the <Enter> key or click outside of the editor.

### Change the Sort Order

Click on the desired column header to change the sort order.

### Navigate to a Bookmark

Click anywhere in the row to navigate to the bookmark.

### Filter Bookmarks

1. Click the Filter  button in the local toolbar of the *Bookmarks* window to display the *Bookmark Filter* dialog.
2. Configure filter information.
3. Click on the **OK** button.



If you have turned off some of the filter types, then the filter icon will show a checkmark (  ).



You may save the settings of the bookmark filter dialog by saving the tool.

In addition to filtering on the type of bookmarks you may also filter the contents of the bookmark table by entering text into the filter text field found at the bottom of the bookmark table. This filter will include only those Bookmarks whose Category or Description contain the specified text. For example, to show only the entry point bookmarks, you would enter "entry" in the filter field. The results would show only those bookmarks with a Category or Description containing the word "entry". The text filter is not case sensitive, nor does it support *regular expressions*.

### Reorder Columns

Reorder columns in the Bookmarks window by dragging the column header to another position in the table.

### Make Selection in the Code Browser

1. Select one or more rows in the Bookmarks table.
2. Click the Select Bookmark Locations  button in the local toolbar.
3. The corresponding addresses are selected in the browser.
4. Navigate to the selected addressed by using the *navigation buttons* (, ) on the *main* tool bar.

### Remove Bookmarks

1. Select one or more rows in the Bookmarks table.
2. Hit the <Delete> key, or press the  icon on the *Bookmarks* toolbar, or right mouse click and choose the **Delete** option.

### Dismiss the *Bookmarks* Window

Click the **Dismiss** button to exit the *Bookmarks* window.

Provided by: *Bookmarks* Plugin

Related Topics:

- [Navigate on Selection](#)
- [Marker Margin](#)
- [Code Browser](#)

# Clear

## Clear Code Bytes

Clear Code Bytes reverts [disassembled](#) instructions and [defined data](#) to their undefined state. It can also be used to clear the components of an applied structure directly from the browser.

To Clear a single instruction,

1. Position the cursor on a defined instruction or data code unit
2. From the right-mouse pop-up menu over the Code Browser window, select *Clear Code Bytes*  
Alternately: press the 'C' key

To Clear an area,

1. Select the defined code units (data and instructions)
2. From the right-mouse pop-up menu over the Code Browser window, select *Clear Code Bytes*  
Alternately: press the 'C' key

To Clear components of a structure directly in the browser,

1. Select the structure elements in the open structure
2. From the right-mouse pop-up menu over the Code Browser window, select *Clear Code Bytes*

Alternately: press the 'C' key



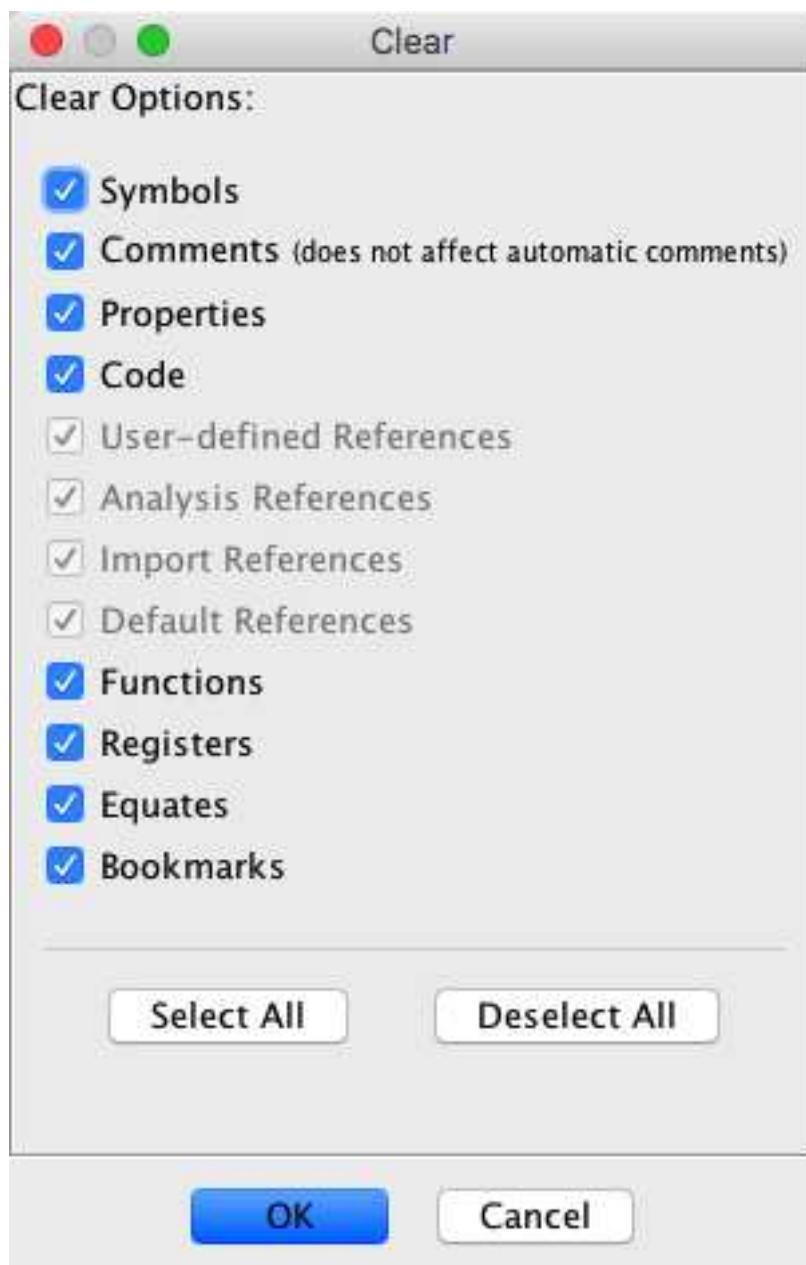
Clearing bytes inside a structure from the CodeBrowser, changes the definition of the structure and will affect all occurrences of the structure.

## Clear With Options

*Clear With Options* more selectively clears information defined in the program. While *Clear Code Bytes* only clears code units and their associated data, Clear With Options can selectively remove additional information, e.g., symbols, references, functions, etc.

The *Clear* dialog has check boxes for each type of information that can be cleared. If the checkbox is selected, that item will be removed everywhere within the selection when the OK button is pressed.

---



To Clear With Options,

1. Create a selection in the Code Browser containing instruction and/or defined data to be cleared.
2. From the right-mouse pop-up menu over the Code Browser window, select *Clear With Options...*
3. De-selected the check boxes from the items that should not be cleared.
4. Click *OK*.



You can undo clearing of code units if it has an undesired effect.

The following paragraphs describe each option.

## Symbols

Any User defined [symbol](#) in the program. Automatically generated symbols won't be cleared if references still exist to the symbol's defined address.



This option will **not** clear function names. To delete these you must select the [functions](#) option.

## Comments

Clears Pre, Post, End of Line (EOL), Plate [comments](#). It will not clear comments on functions or function variables.



This option will **not** clear repeatable or automatic comments. To delete those comments you must delete the associated reference.

## Properties

Properties are placed at addresses in a program by plugins to store information. This information is usually only understood by the plugin that placed it there or other cooperating plugins.



Select the [properties](#) icon (  ) on the tool bar to display all the properties that are currently in the program.

## Code

Data and Instructions are cleared.

## User-defined References

Any references added by the user are cleared.

## Analysis References

Any references created by analysis tasks are cleared.

## Import References

Any references created during the import process are cleared, such as calls to external library functions.

## Default References

References automatically created by Ghidra during disassembly are cleared. This includes references to stack variables within the body of a function.

## Functions

Defined functions are cleared, including comments and any defined variables. Any references to stack variables within the body of the function will be cleared as well.

## Registers

Registers within the selection with a defined value will be cleared and set to undefined. Registers can be set to a value using Set Register Values.

## Equates

Instructions with Scalar Operands set to display an alternate string with an [Equate](#) are cleared.

## Bookmarks

All types of [Bookmarks](#) are cleared.

# Clear Flow and Repair

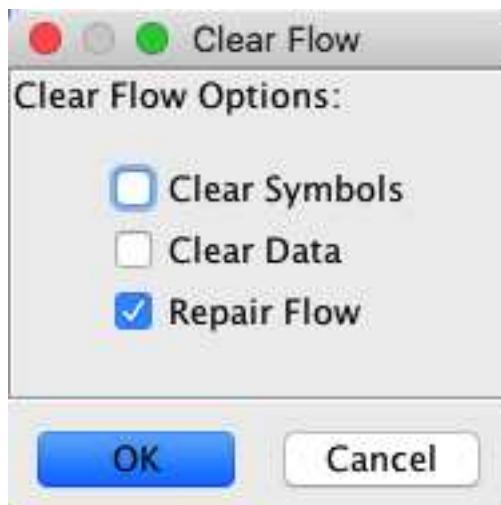
*Clear Flow and Repair* is intended to be used to clear and

optionally repair code which was produced in error. The duration of this action will vary depending on the extent of the instruction flow. If good code with extensive flow is encountered, the action may take a long time to complete. If the flow analysis is lengthy, a task dialog will be displayed with a *Cancel* button. The *Cancel* button may be pressed to terminate the action.



You can undo clearing of code units if it has an undesired effect

The *Clear Flow* options dialog has check boxes to control its behavior. Pressing the *OK* button will begin the clear process using the selected options.



To Clear instructions produced by an invalid fall-through or bad code produced by a data reference:

1. Click on the first bad instruction
2. From the right-mouse pop-up menu over the Code Browser window, select *Clear Flow and Repair...*
3. Choose the desired *Clear Flow Options*.
4. Click *OK*.

To Clear instructions referenced by one or more pointers :

1. Select all pointer data units
2. From the right-mouse pop-up menu over the Code Browser window, select *Clear Flow and Repair...*
3. Choose the desired *Clear Flow Options*.
4. Click *OK*.

Note that clearing pointer referenced code will also clear all computed flow references to the address(es) referenced by the selected pointer(s).

The following paragraphs describe each option.

### **Clear Symbols**

All non-default [symbols](#) at cleared code unit locations will be removed.

### **Clear Data**

All data whose only references were from cleared code unit locations will also be cleared.

### **Repair**

Following the clearing of the flow, attempt to repair the disassembly of references into the cleared region.

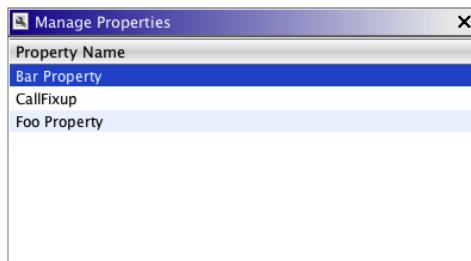
Provided By: *ClearPlugin*

Related Topics:

- [Disassembly](#)
- [Importing Files](#)
- [Property Viewer](#)

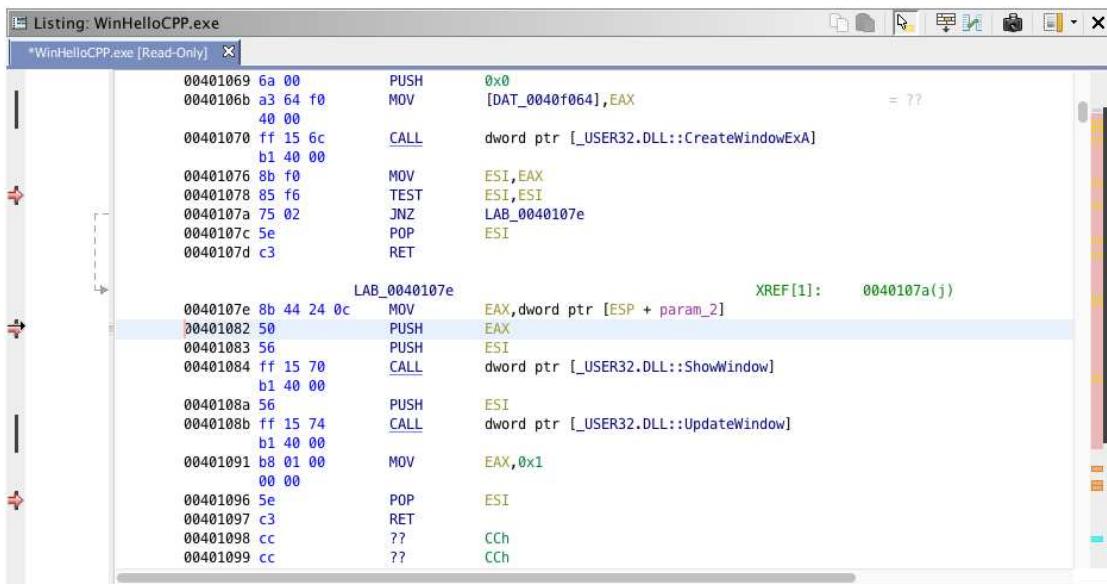
## Property Viewer

The Property Viewer window shows all of the properties in a Program. Properties are assigned to a code unit and can store values at addresses. [Plugins](#) can define their own properties for storing plugin specific information at an address. The display provides a convenient way to see all the properties that exist in the Program. When you select a property, the [navigation\\_margin](#) in [Code Browser](#) shows a pink marker for each location of that property. The window provides a quick way to remove all properties at once.



To display the Property Viewer window, select **Window**→**Manage Properties...** from the tool menu.

In this example, a plugin has placed several source related properties on code units. Select the row for "Source File" to see all the locations in the Code Browser where a "Source File" property is defined, as shown in the image below. The [left\\_margin](#) on the shows marker for the "Source File" property; the [right\\_margin](#) shows the other locations where the Space properties exist; click on the right margin to navigate to that location.



To delete all properties in the Program,

1. Select a property to delete.
2. Right mouse-click and select **Delete**.
  - The property will be removed from the list of properties in the dialog.
  - To undo the delete, select the button on the tool.

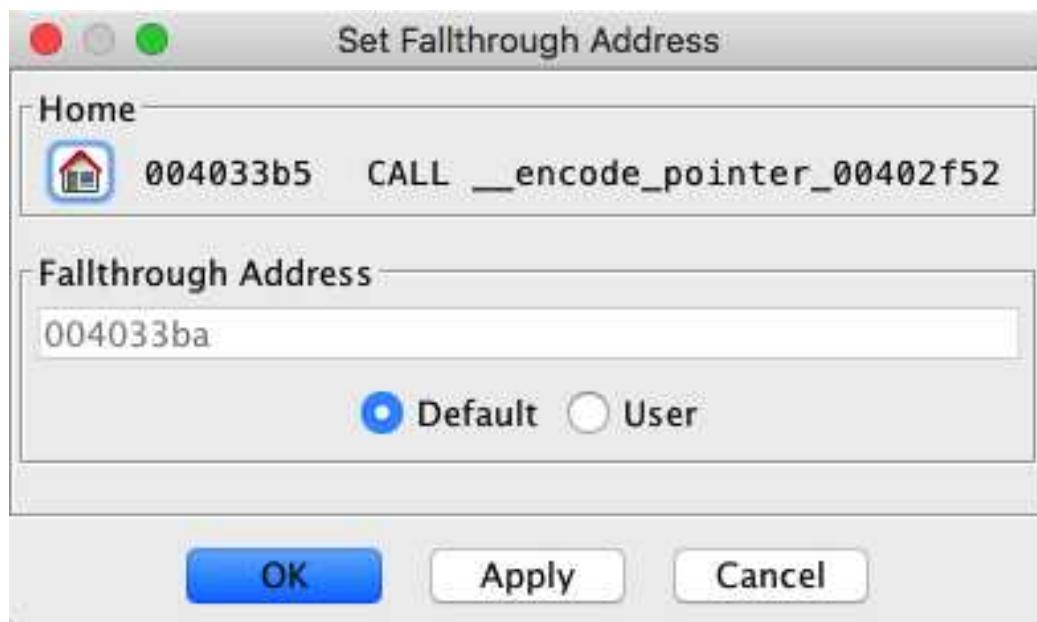
Provided by: *PropertyManagerPlugin*

Related Topics:

- [Code Browser](#)
- [Navigation Margin](#)
- [Marker Margin](#)

# Fallthrough Address

The fallthrough address on an instruction is the address of the *next* instruction that will be executed. You can change the fallthrough address by using the *Set Fallthrough Address* dialog, as shown below. By default, an instruction's fallthrough address (or lack thereof) is determined by the language processor. For example, a "return" or a "jump" instruction does not have a fallthrough address.



## Set a Fallthrough Address

The dialog shows the default fallthrough address of the instruction. The radio buttons below the *Fallthrough*

*Address* field indicate whether the address is the default fallthrough or user defined. When the **Default** button is selected, the *Fallthrough Address* field is disabled. If an instruction has no default fallthrough (e.g., "jump"), the Fallthrough Address field is empty. Choose the **User** button to enter a new fallthrough address. When the **User** button is selected, the *Fallthrough Address* field is updated as you move the cursor in the the Code Browser.

Select the Home button to navigate the Code Browser back to this address. The home panel shows the address and the instruction when you selected the **Set** option.

To change the fallthrough address,

1. Position the cursor on an instruction.
2. Right mouse click and select **Fallthrough → Set...** to display the dialog.
3. Select the **User** radio button.
4. Enter an address, or click in the Code Browser at the address of the new fallthrough.
5. Select the **Apply** button to change the fallthrough and leave the dialog intact; select the **OK** button to change the fallthrough and dismiss the dialog.

You can see the effects of setting the fallthrough address by selecting the [limited flows from option](#); the instructions that are skipped over via setting the fallthrough address are not included in the selection.



Just below the overridden address will be a comment indicating the override, containing the text **Fallthrough Override**, along with the updated fallthrough address.

To clear a fallthrough address using this dialog, select the **None** button, then **Apply** or **OK**.

## Auto Override

The "auto override" feature skips over data following an instruction, finds the next instruction following the data and sets this address as the fallthrough address. You can automatically override the fallthrough address for a single instruction or override the fallthrough addresses over a [selection](#).

To auto override,

1. [Make a selection](#) in the Code Browser or position the cursor at an instruction.
2. Right mouse click and select **Fallthrough → Auto override**



The **Auto Override** option is disabled for a single instruction if the instruction's fallthrough was already overridden.

## Clear Overrides

To clear overridden fallthroughs,

1. [Make a selection](#) in the Code Browser or position the cursor at an instruction whose fallthrough address was overridden.
2. Right mouse click and select **Fallthrough → Clear Overrides**



The **Clear Overrides** option is disabled for a single instruction if the instruction's fallthrough address was not overridden.

Provided by: *FallthroughPlugin*

Related Topics:

- [Selections](#)
- [Code Browser](#)
- [Languages](#)

# Navigation

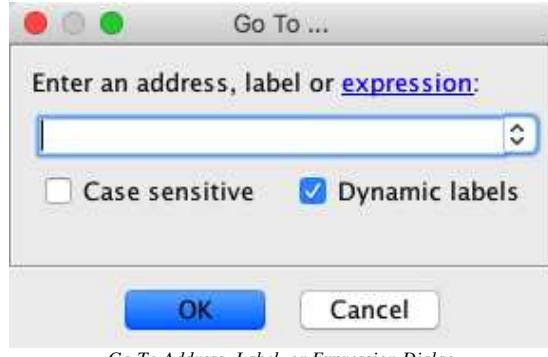
Often, users need to navigate to specific locations in a program. Ghidra provides several different ways to do this:

- Enter a particular address or label ([Go To](#))
- Double-click on any displayed label or address in the [Code Browser](#) tool
- Jump directly to the [Next/Previous Code Unit](#)
- Use the [Navigation History](#) to return to a previously visited location

## Go To Address, Label, or Expression

### To perform a Go To:

1. In the menu-bar of a tool, select **Navigation ➔ Go To...**
2. The *Go To* dialog will be displayed, as shown below:
3. Enter either an [address](#), [label](#), or [expression](#) as specified below and press "OK"
4. If the address, label, or expression is valid, the Code Browser will be repositioned to that location and the dialog will be dismissed
5. If the address, label, or expression is not valid, the dialog will display an error message



*Go To Address, Label, or Expression Dialog*

### Go To Address

Enter an address into the text area of the dialog. The value entered will be assumed to be in hexadecimal. That is, "0x1000" and "1000" are the same value.



*When the program has multiple address spaces and the destination address is ambiguous (based on the current location), a query results dialog will be displayed.*

**Consider the following examples:**

Given:

A program with the following memory blocks **which reside in different address spaces**:

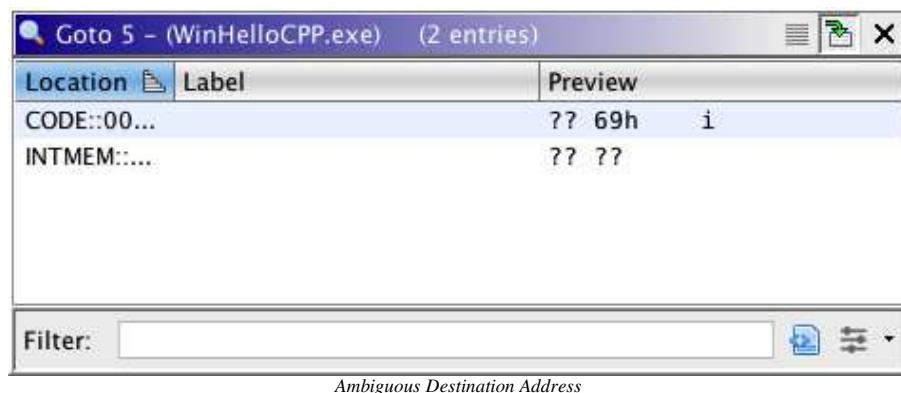
Name	Start Address	End Address
BLOCK1	BLOCK1:00000000	BLOCK1:0000ffff
BLOCK2	BLOCK2:00000000	BLOCK2:0000ffff
BLOCK3	BLOCK3:00000080	BLOCK3:0000ffff

Example #1 – Unambiguous address

1. Set the current location to BLOCK1:00001000
2. Goto address "5"
3. The destination is not ambiguous:
  - The BLOCK1 block has an address "5" so the listing will move to address 5.

Example #2 – Ambiguous Address

1. Set the current location to BLOCK3:00000080
2. Goto address "5"
3. The destination is ambiguous because:
  - The BLOCK3 block does not have an address "5"
  - Both the BLOCK1 and BLOCK2 blocks have an address "5"
4. A [Query Results](#) dialog will be displayed as shown in the image below.



### Go To Label

Enter the name of an existing label into the text area of the dialog.

**Case Sensitive**

By default, the values entered are case sensitive. That is, "LAB1000" is not the same as "lab1000." If you want to find both of these labels, turn off the case sensitive option. If more than one match is found, they are displayed in a Query Results dialog.



Even if the case sensitive option is off, if a label has an exact match, no other labels will be found.

### ***Dynamic Labels***

This option only affects queries that could potentially result in multiple results, i.e when a search must be performed versus a lookup. This occurs when either the case sensitive is turned off or a wildcard is used. Specifically, this option tells Ghidra, when doing a search versus a direct lookup, to consider all the *Dynamic* symbols (symbols that are not stored, but are generated on the fly because of a reference to that location.) If this option is off, only defined labels are searched.



Turning off this option can result in significantly faster results in larger programs.

### **Go To Expression**

Enter an arithmetic expression that can include addresses, symbols, or can be relative to the current location. All numbers are assumed to be hexadecimal. Supported operator are "+ - \* / << >>". Also, parentheses are supported to control order of expression evaluation.

For example:

- ENTRY+10 Positions the cursor at the address 0x10 addresses past the symbol ENTRY.
- 0x100000+30 Positions the cursor at address 0x100030.
- 0x100000+(2\*10) Positions the cursor at address 0x100020.
- +20 Positions the cursor at an address that is 0x20 past the current location.

### **Executing a Query**

A Query performs a case-insensitive search for all labels that match the criteria. A Query is specified using wildcards.

### ***Using Wildcards***

Wildcard characters ("?" or "\*") can be used when searching for labels. Wildcards are useful if you don't know the full label name or don't want to type the entire name.

### ***Asterisk (\*)***

You can use the asterisk as a substitute for zero or more characters.

### ***Example***

If you're looking for a label that you know starts with "gloss", type the following:

**gloss\***

The *Go To Address or Label* dialog will locate all labels that begin with "gloss" including **Glossary.txt**, **Glossary.doc**, and **Glossy.doc**. To narrow the search to a specific extension, type:

**gloss\*.doc**

In this case, the *Go To Address or Label* dialog will find all labels that begin with **gloss** but have the extension **.doc**, such as **Glossary.doc** and **Glossy.doc**.

### ***Question Mark (?)***

Use the question mark as a substitute for a single character in a name.

### ***Example 1***

If you typed **gloss?.doc**, the *Go To Address or Label* dialog would locate the label **Glossy.doc** or **Gloss1.doc**, but not **Glossary.doc**.

### ***Example 2***

Suppose that two of the labels in a program were **FUN\_0040816d** and **FUN\_004081bd**. A possible query string to match these two labels would be **FUN\_004081?d**. The results of the query are displayed in a *Query Results* dialog, as shown below.



The screenshot shows the 'Goto' dialog with the title 'Goto FUN\_004081?d - (WinHelloCPP.exe) (2 entries)'. The table has four columns: Loc..., Label, Namespace, and Preview. The first entry is '00408... FUN\_0040812d Global void FUN\_0040...' and the second entry is '00408... FUN\_0040817d Global void FUN\_0040...'. A 'Filter:' input field is at the bottom.

Loc...	Label	Namespace	Preview
00408...	FUN_0040812d	Global	void FUN_0040...
00408...	FUN_0040817d	Global	void FUN_0040...

*Query Results Dialog*

## Repeating a Previous Go To

Each time a Go To Label or a Query is performed, it is stored in the drop-down box as shown in the image below.



To repeat a previous Go To or Query:

1. Select the item from the *Enter an address or label:* drop-down box
2. Click the **OK** button

## Error Messages

When a Go To or Query fails, an error message will be displayed in the status area of the dialog.

1. Entering an invalid address or non-existing label
  - The dialog displays "*This is not a query, label, or address.*"
2. Specifying a query that has no results
  - The dialog displays "*No results for ...*", where "..." is the query string.

Provided by: *Go To Address or Label* plugin

## Next/Previous Code Unit

The Next/Previous Code Unit feature allows the user to jump directly to the next or previous Instruction, Data, Undefined, Function or Non Function. The search starts at the current cursor location and proceeds either forward (next) or backwards (previous).



When searching for Instructions, Data or Undefined items, Ghidra will skip all contiguous items of the same type. For example, if the cursor is on an address with an Instruction, and you go to the next Instruction, then all Instructions immediately following the current one will be skipped until a non-Instruction is found. Once that non-instruction is found, then Ghidra will take you to the next Instruction after the address of that non-Instruction.

### Search Direction

The icon indicates the search will be performed in the forward (next) direction, and the icon indicates the search will be performed in the backward (previous) direction. To change the direction of the code unit search, toggle the arrow icon on the toolbar.

### Navigate to Instruction

To move the cursor to the next instruction click on the Navigate by Instruction icon, . This icon is disabled when no more instructions exist in the current search direction.

### Navigate to Data

To move the cursor to the next data code unit, click on the Navigate by Data icon, . This icon is disabled when no more data code units exist in the current search direction.

### Navigate to Undefined

To move the cursor to the next undefined code unit, click on the Navigate by Data icon, . This icon is disabled when no more undefined code units exist in the current search direction.

### Navigate to Label

To move the cursor to the next Label, click on the Navigate by Label icon, . This icon is disabled when no more labels exist in the current search direction.

### Navigate to Function

This () action will move the cursor to the next function in the current direction. If inside a function and the direction is towards lower addresses, then this action will go to the current function's entry point.

### Navigate to Non-Function

This task () will attempt to the navigate to the next instruction block not contained in a function. This can be useful when manually creating functions and stepping over them to identify potential function candidates.

### [Navigate to Different Byte Value](#)

This task () will attempt to the navigate to the first code-unit where the byte value is different from the byte value of the first byte of the current code unit. This can be useful when trying to navigate past a series of 0s or FFs;

### [Navigate to Bookmark](#)

To move the cursor to the next bookmark, click on the Navigate by Bookmark icon, . This icon is disabled when no more bookmarks exist in the current search direction. You may use the pull-down menu to choose a specific type of bookmark (, , , , , ) to navigate to as opposed to all types.

Provided by: *Go To Next-Previous Code Unit* plugin

## Next/Previous Function

Navigating to the next or previous function is a commonly used feature. As such, separate actions have been created so that keybindings can be assigned for each direction.

### [Next Function](#)

This action navigates the cursor to the closest function entry point that is at an address greater than the current address. The default keybinding is

**Control-Down Arrow**

### [Previous Function](#)

This action navigates the cursor to the closest function entry point that is at an address less than the current address. The default keybinding is

**Control-Up Arrow**

Provided by: *CodeBrowser* plugin

## Navigation History

As the user performs various types of navigations, the current location is pushed onto the navigation history stack. The navigation history feature allows the user to revisit previous locations.

### [Go To Next/Previous Location](#)

To traverse the history stack:

1. In the tool-bar, click either the **Go to previous location** ( ) button or the **Go to next location** ( ) button
2. The Code Browser will be repositioned to the saved location

*Some Operations that add to the navigation history:*

- **Go To Address or Label**
- **Double-clicking on operands containing addresses or labels**
- Double-clicking on XREFs ([field](#) in the [Code Browser](#))
- Clicking on the start or end address of a memory block using the memory map dialog
- Clicking on the address of an equate reference using the equates table
- Performing a search ([Memory](#), [Program Text](#), etc)



The button is only enabled after performing a

### [Clear History](#)

To clear the navigation history stack, select **Navigation ➔ Clear History**

After clearing the history, the and buttons are disabled

Provided by: *Next/Previous* plugin

## Component Provider Navigation

This section lists actions that allow the user to navigate between component providers.

### **Go To Last Active Component**

Allows the user to switch focus back to the previously focused component provider.

Provided by: *ProviderNavigation* plugin

Related Topics:

- [Query Results](#)
- [Code Browser](#)
- [Search Memory](#)
- [Search Program Text](#)

# Selecting

Selecting is the process of [highlighting](#) all or portions of a program in order to perform a task (for example, modular analysis) on the selection. Selecting can be done manually in providers that support selection, like the [Code Browser](#). Selecting can be done based on subroutines, functions, or by following certain types of control flows in the program.

Selections can be created manually within the Code Browser. There are also some predefined types of selections that are available from the tool **Select** menu. Each predefined type of selection exposes different characteristics of the program.

To create a selection using one of the predefined methods:

- From the Code Browser, select the menu option **Select** → *SelectionType*.

The *SelectionTypes* and their descriptions are as follows:

## Predefined Selection Methods

Selection Type	Description
Program Changes	Select all program changes.
All Flows From	This follows all program flows from the cursor location if no selection or from the selected code units if there is a selection. Select All Flows will traverse all branches conditional and unconditional. All the code units that are traversed are selected.
All Flows To	This selects all program flows to the cursor location if no selection or to the selected code units if there is a selection. Select All Flows To will traverse all branches conditional and unconditional. All the code units that are traversed are selected. In other words, this follows all program flows backwards from the location or selection.
Limited Flows From	The types of program flows to be followed in this case are based on options configured on the <a href="#">Selection by Flow Options tab</a> . Limited Flows only follows the indicated types of program flows from the cursor location if no selection or from the selected code units if there is a selection. All the code units that are traversed are selected.
Limited Flows To	The types of program flows to be followed in this case are based on options configured on the <a href="#">Selection by Flow Options tab</a> . Limited Flows To only follows the indicated types of program flows to the cursor location if no selection or to the selected code units if there is a selection. All the code units that are traversed are selected. In other words, this follows the types of program flows, as indicated by the options, backwards from the location or selection.
Subroutine	If there is no selection, this selects the code units of the subroutine that contains the current cursor location. If there is a selection, this selects all the subroutines that contain the selected code units.
Dead Subroutines	Selects the code units of all subroutines that not directly referenced, also known as "dead" code.
Function	If there is no selection, this selects the code units of the function that contains the current cursor location. If there is a selection, this selects all the functions that contain the selected code units.
All in View	Selects all code units being displayed in the browser view.
Clear Selection	Clears the current selection in the browser view.
Complement	Changes the selection to everything in the current view that is not in the current selection.
Data	Selects all the defined data in the current program if there isn't a selection. Otherwise, it selects the defined data within the current selection.
Instructions	Selects all the instructions in the current program if there isn't a selection. Otherwise, it selects the instructions within the current selection.
Undefined Data	Selects all the undefined data in the current program if there isn't a selection. Otherwise, it selects the undefined data within the current selection.
Forward Refs	Selects all addresses that the current address is referring to.
Back Refs	Selects all addresses that refer to the current address.

Selections are meant to be temporary. For example, a left mouse click outside a selection in the Code Browser will make the selection go away. Also, creating a new selection replaces any previous selection rather than adding to it. To retain a selection in a manner that isn't so transient, change it to a [highlight](#).



At any time you can restore the previous selection for the current program by pressing the **Select** → **Restore Selection**.

### Selection by Flow Tool Options

The *Select By Flow* plugin adds options to the tool. To view or edit the option settings:

- From the tool's menu select **Edit** → **Tool Options...**
- Click on the *Selection by Flow* tree node

The *Selection by Flow* tab contains options for indicating the types of flows that will be followed when selecting *Limited Flows*. The table below lists the *Selection by Flow* options and their default settings. To follow a particular flow type simply click on the box to check it.

Flow Type to Follow	Default
Follow computed call	false
Follow computed jump	false
Follow conditional call	false
Follow conditional jump	true
Follow pointers	false
Follow unconditional call	false
Follow unconditional jump	true

Example: Given *Follow conditional jump* and *Follow unconditional jump* are the only types checked. As the program flow is followed the instructions that are encountered are added to the selection, wherever the program jumps will also get added to the selection and the program flow is also followed from there. If a conditional call is encountered, then this instruction gets added to the selection. But the conditional call is not followed and the code at the call's destination is not added since this type isn't being followed. *The code for the subroutine at the call's destination could still end up in the selection, if it is flowed to by a different flow path using the flow types being followed.*

Provided by: *Select By Flow* Plugin

## Navigating Over a Selection

A selection will often consist of one or more address ranges. The address ranges do not have to be contiguous. The navigation margin can be used to directly move the location to a particular selected range. To move to the next or previous selection range relative to the current cursor location use the **Next Selected Range** and **Previous Selected Range** buttons.

### Using the Navigation Margin

One or more green markers will appear in the **Navigation Margin**, where each green marker corresponds to an address range included in the selection. Clicking on a green

marker will cause the Code Browser to navigate to the beginning of the corresponding address range.

### **Next Selected Range**

To move the program's cursor location to the beginning of the next selected range of addresses:

From the menu-bar of the Code Browser, select **Navigation ➔ Next Selected Range**

OR

From the tool-bar of the Code Browser, click the **Go to next selected range** () button



*When the Code Browser is on or after the last address range in the selection, the "Next Selected Range" menu-bar and tool-bar options will be disabled.*

### **Previous Selected Range**

To move the program's cursor location to the beginning of the next selected range of addresses:

From the menu-bar of the Code Browser, select **Navigation ➔ Previous Selected Range**

OR

From the tool-bar of the Code Browser, click the **Go to previous selected range** () button



*When the Code Browser is on or before the first address range in the selection, the "Previous Selected Range" menu-bar and tool-bar options will be disabled.*

Provided by: *Go To Next-Previous Selected Range* Plugin

Related Topics:

- [Margin & Navigation Markers](#)
- [Navigation](#)
- [Highlighting](#)

# Selecting Scoped Flow

Scoped [Flow](#) is the potential flow through a function, removing any flow that is not in the selected scope. This is useful for determining how to reach a set of instructions.

## Forward and Reverse Scoped Flow

*Forward scoped flow* is the set of addresses that can only be reached by passing through the [Basic Block](#) containing the current address.

*Reverse scoped flow* is the set of addresses that must pass through the [Basic Block](#) containing the current address.

To create forward or reverse scoped [selections](#), use one of the following two actions, respectively:

- From the menu bar choose **Select → Scoped Flow → Forward Scoped Flow**
- From the menu bar choose **Select → Scoped Flow → Reverse Scoped Flow**

Provided by: *Select By Scoped Flow* Plugin

Related Topics:

- [Selecting in Ghidra](#)
- [Highlighting](#)

# Highlighting

A highlight is a more permanent variation of a [selection](#). As you may recall, a selection can be cleared simply by clicking in the Code Browser. In order to clear a highlight, you must explicitly select the **Select → Highlight →** option.

You would commonly use a highlight when you do not want to lose a selection. For example, you just used [Search Memory](#) to search for all the occurrences of a particular Ascii string. From the [Query Results](#) window, you selected all of the results. Now you wish to visit each code unit in the selection and perform some operations. A selection would not be entirely useful, because, if you clicked anywhere in the Code Browser, the selection would be lost. Converting this selection to a highlight enables you to visit each code unit and click around in the browser, without losing the highlight for the search results.

## To set a highlight from a selection

1. Create a selection in the Code Browser
2. From the menu-bar of the Code Browser, select  
**Select → Highlight → Selection**

OR

From the right mouse popup menu of the Code Browser, select **Highlight → Entire Selection**

## To clear a highlight

1. From the menu-bar of the Code Browser, select  
**Select → Highlight → Clear**
2. OR

From the right mouse popup menu of the Code Browser, select **Highlight → Clear**

### To add a selection to a highlight

1. From the menu-bar of the Code Browser, select  
**Select → Highlight → Add Selection**

OR

From the right mouse popup menu of the Code Browser, select **Highlight → Add Selection**

### To subtract a selection from a highlight

1. From the menu-bar of the Code Browser, select  
**Select → Highlight → Subtract Selection**

OR

From the right mouse popup menu of the Code Browser, select **Highlight → Subtract Selection**

### To create a selection from a highlight

1. Create a highlight in the Code Browser
2. From the menu-bar of the Code Browser, select  
**Select → From Highlight**

OR

From the right mouse popup menu of the Code Browser, select **Select → Entire Highlight**

## Navigating over a highlight

A highlight will usually consist of one or more address ranges. The address ranges do not have to be contiguous. Two methods exist to allow navigation over these ranges:

### Using the Navigation Margin

One or more yellow markers will appear in the [Navigation Margin](#), where each yellow marker corresponds to an address range included in the highlight. Clicking on a yellow marker will cause the Code Browser to navigate to the beginning of the corresponding address range.

### Using the Navigation Menu

From the menu-bar of the Code Browser, select either **Navigation → Previous Highlighted Range** or **Navigation → Next Highlighted Range**.

OR

From the tool-bar of the Code Browser, select either the **Go to previous highlighted range** (☞) and **Go to next highlighted range** (☜) buttons.

Clicking on the menu-bar or tool-bar options will cause the Code Browser to navigate to the beginning of the corresponding address range.



*When the Code Browser is on or before*

*the first address range in the highlight, the "Previous Highlight Range" menu-bar and tool-bar options will be disabled. Similarly, when the Code Browser is on or after the last address range in the highlight, the "Next Highlight Range" menu and toolbar options will be disabled.*

Provided By: *Next Prev Highlight Range Plugin*

#### Related Topics:

- [Selecting](#)
- [Code Browser](#)
- [Navigation Margin](#)
- [Query Results](#)

# Searching

Ghidra offers a variety of searching capabilities. The *Search Program Memory* feature performs fast searching for byte patterns in program memory. The [\*Search Program Text\*](#) feature searches for text strings in various parts of the listing such as comments, labels, mnemonics, and operands. The [\*Search For Strings\*](#) feature automatically finds potential ascii strings within the program memory.

## Related Topics:

- [Search Memory](#)
- [Search Program Text](#)
- [Search For Strings](#)
- [Search For Address Tables](#)
- [Search For Direct References](#)
- [Search For Instruction Patterns](#)

# Search Memory

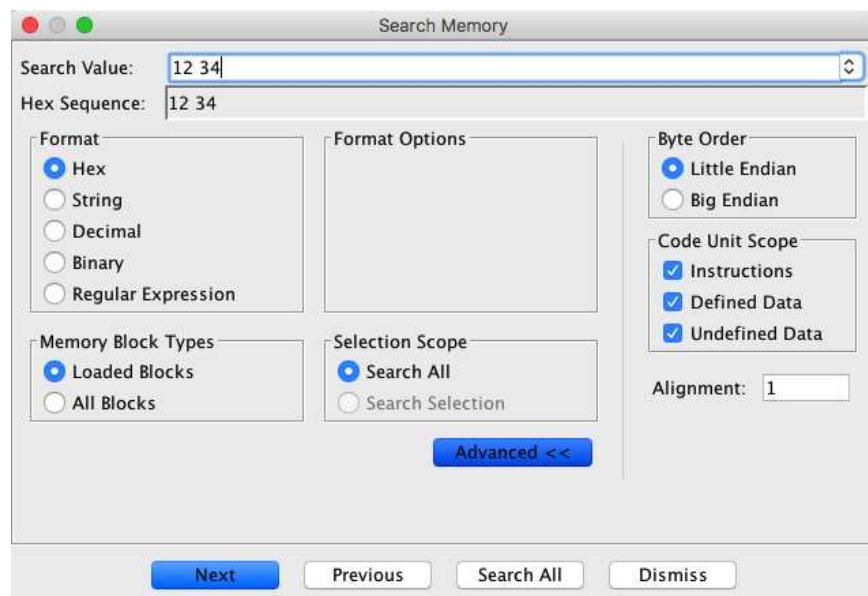
Search Memory locates sequences of bytes in program memory. The search is based on a value entered as hex numbers, decimal numbers or strings. The byte sequence may contain "wildcards" that will match any byte (or possibly nibble). String searching also allows for the use of [regular expression](#) searches.

To Search Memory:

1. From the Tool, select **Search ➔ Memory**
2. Enter a Hex String in the Value field  
This will create a Hex Sequence for searching.
3. Choose "Next" to find the next occurrence  
— or —  
Choose "Previous" to find the previous occurrence  
— or —  
Choose "Search All" to find all occurrences.

## Search Formats

- [Hex](#)
- [String](#)
- [Decimal](#)
- [Binary](#)
- [Regular Expression](#)



## Search Options

### Search

#### Search Value

- The value to search. The values entered will be interpreted based on the *Format* options.

#### Hex Sequence

- As the search value is entered, this field will display the exact hex byte sequence that will be searched for in memory.

### Format

**Hex:**

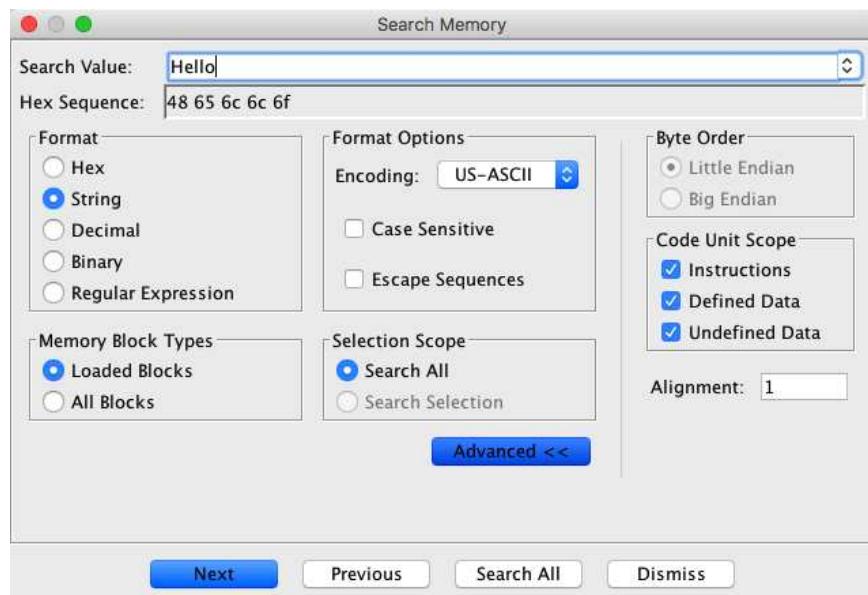
- Value is interpreted as a sequence of hex numbers, separated by spaces. Wildcard characters can be used to match any single hex digit (i.e. any 4 bit value). Either the `.' or `?' character can be used for the wildcard character.
- Each hex number (separated by spaces) will produce a sequence of bytes that may be reversed depending on the Byte Order.
- The byte search pattern is formed by concatenating the bytes from each hex number.

**Example:**

<b>Value:</b>	"1234 567 89ab"
<b>LittleEndian Hex Sequence</b>	34 12 67 05 ab 89
<b>BigEndian Hex Sequence</b>	12 34 05 67 89 ab

**String:**

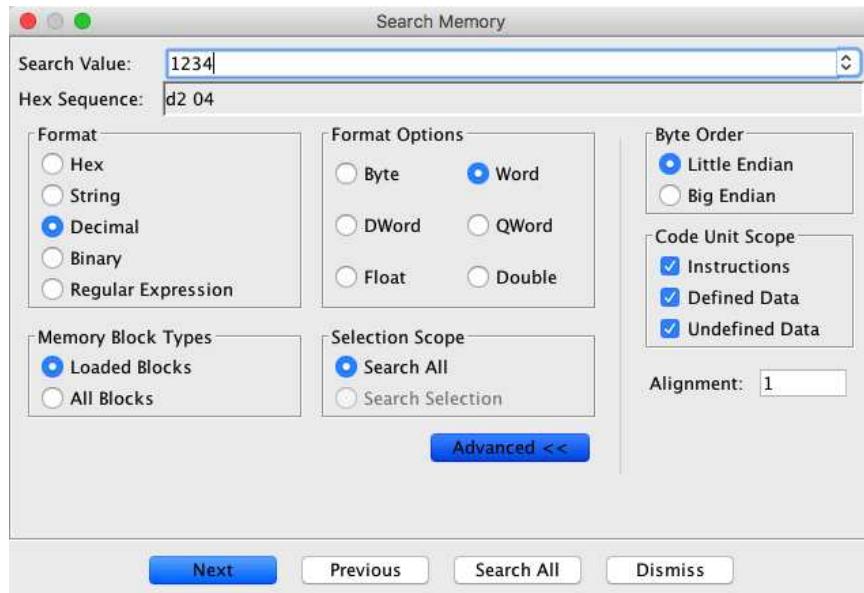
Value is interpreted as the specified character encoding. The center panel of the Search Memory dialog shows the *Format Options*, described below.



- Encoding** –Interprets strings by the specified encoding. Note that byte ordering determines if the high order byte comes first or last.
- Case Sensitive** –Turning off this option will search for the string regardless of case using the specified character encoding. Only applicable for English characters.
- Escape Sequences** –Enabling this option allows escape sequences in the search value (i.e., allows \n to be searched for).

**Decimal:**

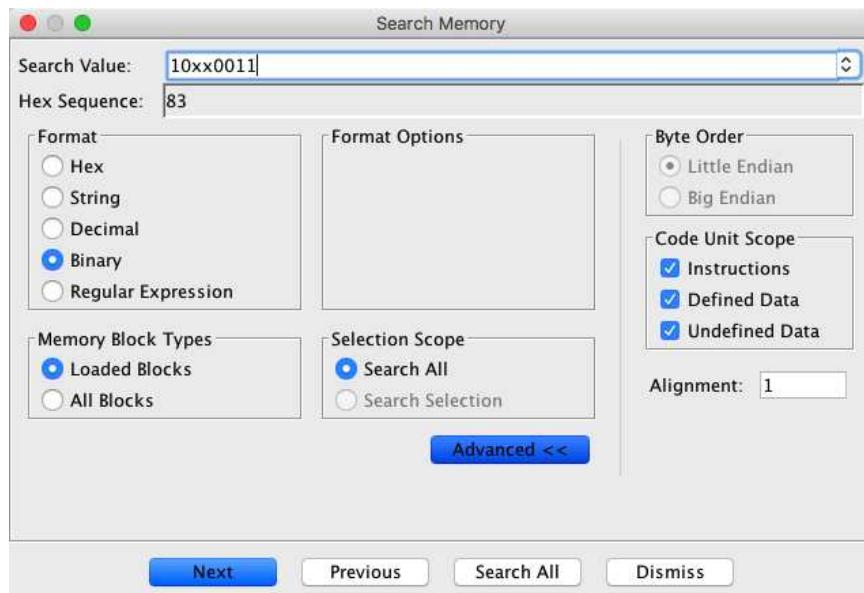
Value is interpreted as a sequence of decimal numbers, separated by spaces. The center panel of the Search Memory dialog shows the *Decimal Options*, described below.



- Only numbers that fit the specified Decimal Options are allowed to be entered.
- The byte search pattern is formed by concatenating the bytes from each number.
- Valid decimal numbers are:
  - Byte –any fixed point 8 bit number (-128 to 255)
  - Word –any fixed point 16 bit number (-32768 to 65535)
  - DWord –any fixed point 32 bit number (you get the idea.....)
  - QWord –any fixed point 64 bit number
  - Float –any 32 bit floating point number
  - Double any 64 bit floating point number

#### Binary:

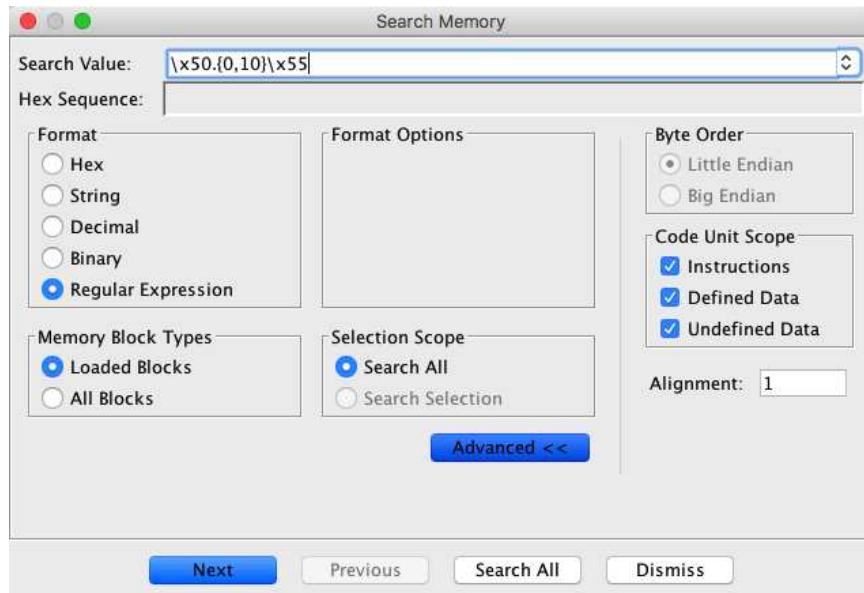
Value is interpreted as a sequence of binary numbers, separated by spaces. Wildcard characters ('x' or '?' or '.') can be used to match any bit.



- Only binary digits (0 or 1) or wildcard characters (\*?). are allowed to be entered.
- The byte search pattern is formed by concatenating the bytes from each number.
- An additional Mask byte which is not shown, is generated for each search byte to handle the wildcards.

#### RegularExpression:

Value is interpreted as a Java *Regular Expression* that is matched against memory as if all memory was a string. Help on how to form regular expressions is available on the [Regular Expression Help](#) page.



- Regular Expressions can only be used to search forward in memory.
- No Hex Sequence is displayed for regular expressions.

#### Memory Block Types

- Selects which initialized memory blocks are searched. Ghidra now stores external information from the program's file header in special memory blocks. These blocks do not live in the program's address space, but instead are stored in the "OTHER" address space. Memory blocks which would be found in an actual running version of the program are referred to as "Loaded Memory Blocks."
- Loaded Blocks –will search only "loaded" memory blocks (memory blocks that would appear in an actual running instance of the program) and not "Other" information memory blocks.
- All Blocks –will search all memory blocks including "Other" blocks.

#### Selection Scope

- **Search All** –If this option is selected, the search will search all memory in the tool.
- **Search Selection** –If this option is selected, the search will be restricted to the current selection in the tool. This option is only enabled if there is a current selection in the tool.

#### Code Unit Scope

Filters the matches based upon the code unit containing a given address.

- **Instructions** –includes instruction code units in the search.
- **Defined Data** –includes defined data in the search.
- **Undefined Data** –includes undefined data in the search.

#### Byte Order

Sets the byte ordering for multi-byte values. Has no effect on non-Unicode Ascii values, Binary, or regular expressions.

**LittleEndian** –places low-order bytes first.

For example, the hex number "1234" will generate the bytes "34", "12".

**BigEndian** –places high-order bytes first.

For example, the hex number "1234" will generate the bytes "12", "34".

#### Alignment

- Generally the alignment defaults to 1, but can be set to any number greater than 0. The search results will be limited to those that begin on the specified byte alignment. In other words, an alignment of 1 will get all matching results regardless of the address

where each begins. An alignment of 2 will only return matching results that begin on a word aligned address.

### Searching

- Next / Previous –Finds the next/previous occurrence of the byte pattern from the current cursor location; if you mouse click in the Code Browser to move focus there, you can choose **Search** ➔ **Repeat Memory Search** to go to the next/previous match found.
- Search All –Finds all occurrences of the byte pattern in a [Query Results display](#).



For very large Programs that may take a while to search, you can cancel the search at any time. For these situations, a progress bar is displayed, along with a **Cancel** button. Click on the **Cancel** button to stop the search.



Dismissing the search dialog automatically cancels the search operation.

### Highlight Search Option

You can specify that the bytes found in the search be highlighted in the Code Browser by selecting the *Highlight Search Results* checkbox on the Search Options panel. To view the Search Options, select **Edit** ➔ **Tool Options...** from the tool menu, then select the *Search* node in the Options tree in the Options dialog. You can also change the highlight color. Click on the color bar next to *Highlight Color* to bring up a color chooser. Choose the new color, click on the **OK** button. Apply your changes by clicking on the **OK** or **Apply** button on the Options dialog.



Highlights are displayed for the last search that you did. For example, if you bring up the Search Program Text dialog and search for text, that string now becomes the new highlight string. Similarly, if you invoke [cursor\\_text\\_highlighting](#), that becomes the new highlight string.

Highlights are dropped when you close the search dialog, or close the query results window for your most recent search.

### Search for Matching Instructions

This action works only on a selection of code. It uses the selected instructions to build a combined mask/value bit pattern that is then used to populate the search field in the Memory Search Dialog. This enables searching through memory for a particular ordering of instructions. There are three options available:

- **Include Operands** –All bits that make up the instruction and all bits that make up the operands will be included in the search pattern.
- **Exclude Operands** –All bits that make up the instruction are included in the search pattern but the bits that make up the operands will be masked off to enable wild carding for those bits.
- **Include Operands (except constants)** –All bits that make up the instruction are included in the search pattern and all bits that make up the operands, except constant operands, which will be masked off to enable wild carding for those bits.

Example:

A user first selects the following lines of code. Then, from the Search menu they choose **Search for Matching Instructions** and one of the following options:

LAB_00401e8c		
00401e8c a1 20 0d	MOV	EAX, DAT_00410d20]
41 00		
00401e91 85 c0	TEST	EAX, EAX
00401e93 56	PUSH	ESI
00401e94 6a 14	PUSH	0x14
00401e96 5e	POP	ESI
00401e97 75 07	JNZ	LAB_00401ea0

Option 1:

If the **Include Operands** action is chosen then the search will find all instances of the following instructions and operands.

85	c0	TEST	EAX,EAX
56		PUSH	ESI
6a	14	PUSH	0x14
5e		POP	ESI

All of the bytes that make up the selected code will be searched for exactly, with no wild carding. The bit pattern **10000101 11000000 01010110 01101010 00010100 01011110** which equates to the byte pattern **85 c0 56 6a 14 5e** is searched for.

#### Option 2:

If the **Exclude Operands** option is chosen then the search will find all instances of the following instructions only.

TEST
PUSH
PUSH
POP

Only the parts of the byte pattern that make up the instructions will be searched for with the remaining bits used as wildcards. The bit pattern **10000101 11..... 01010... 01101010 ..... 01011...** is searched for where the . indicate the wild carded values.

#### Option 3:

If the **Include Operands (except constants)** option is chosen then the search will find all instances of the instruction and all operands except the 0x14 which is a constant.

TEST	EAX,EAX
PUSH	ESI
PUSH	N
POP	ESI

The bit pattern **10000101 11000000 01010110 01101010 ..... 01011110** which equates to the byte pattern **85 c0 56 6a xx 5e** is searched for where xx can be any number N between 0x0 and 0xff.



The previous operations can only work on a **single** selected region. If multiple regions are selected, the following error dialog will be shown and the operation will be cancelled.



Provided by: the *MemSearchPlugin*

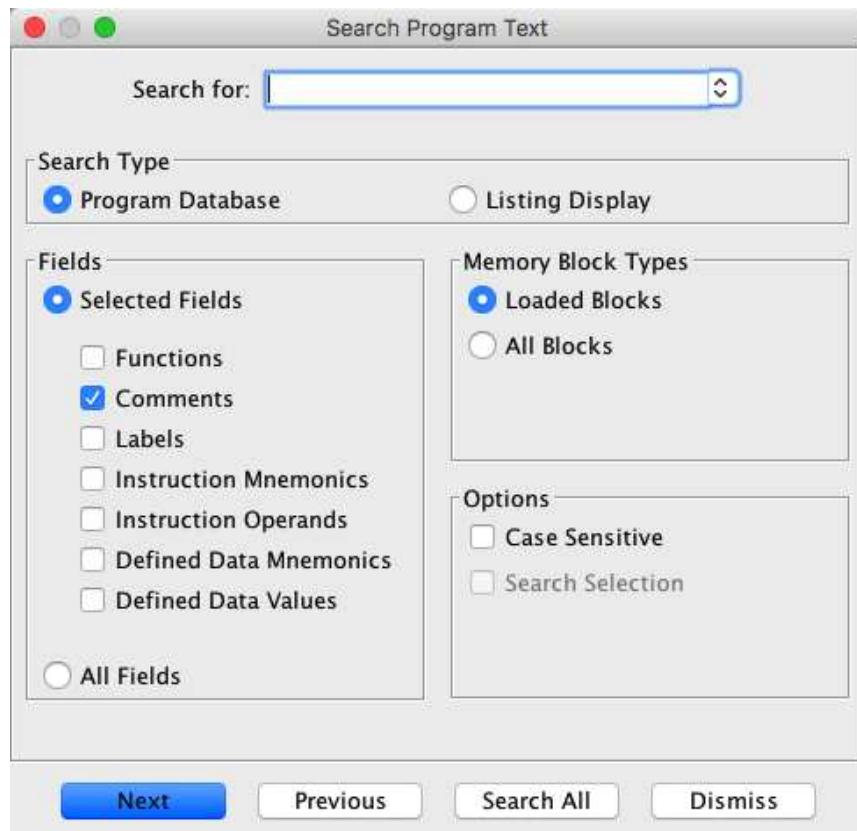
#### Related Topics:

- [Searching Program Text](#)
- [Query Results](#)
- [Regular Expressions](#)

# Search Program Text

The Search for Program Text feature allows you to search for textual strings within [functions](#), [comments](#), [labels](#), [instructions](#), and [defined data](#). You can search incrementally or get a list of all of the search results. You can search the entire program, or limit the search to your current selection in the Code Browser.

To bring up the *Search Program Text* dialog, as shown below, select **Search**→**Program Text...** from the tool menu.



## Search Type

There are two ways that you can search for text: the *Program Database Search* and the *Listing Display Search*.

The *Program Database Search* option searches the program database, not what you actually see in the Code Browser. Conversely, the *Listing Display Search* searches exactly what you see in the Code Browser. These searches yield different results because the listing includes derived and auto-generated information that is not stored in the database and the database can contain information that is not currently displayed, such as offcut comments. The following table summarizes the pros and cons of using each search type:

	<i>Program Database Search</i>	<i>Listing Display Search</i>
	<ul style="list-style-type: none"> <li>-Faster than the <i>Listing Display Search</i></li> <li>-Can search information that</li> </ul>	<ul style="list-style-type: none"> <li>-Search Results reflect what you see in the Code Browser Listing window.</li> <li><small>Allows a search of all fields.</small></li> </ul>

<b>Advantages</b>	<p>is not currently displayed. For example, comments that are offcut are not displayed, but the database search can still find them.</p>	<p>–Allows a search of all fields that are displayed in the Code Browser, which includes auto generated and derived information that is not stored in the database.</p>
<b>Disadvantages</b>	<p>–Search Results may not match what the Listing displays, since this search looks at information that is directly stored in the database and not the derived and enhanced information that is shown in the Listing Display. Searches are also restricted to specific fields and may not cover all the fields shown in the Code Browser listing window.</p> <p>–Assumes a specific <a href="#">search order</a> of the fields; you may have rearranged the fields in your Code Browser such that they appear in a different order from the search order; this may cause cursor movement for incremental searches to appear "random" within the content for a single address.</p>	<p>–Can be MUCH slower than a database search. For example, if you have a large program with one comment, the database search can find a match in that comment instantly, while the listing display search will have to render every address in the program one at a time until it finds a hit.</p> <p>–Does not find information that is currently not displayable by the Listing window. For example, comments at offcut locations are not displayed in the browser, so this search would not find them.</p>

By default, the **Program Database Search Type** type is selected.



If you select the **All Fields** button, the **Listing Display Search Type** button automatically becomes selected, as the **Search All Fields** does not apply to the **Program Database**. The **Selected Fields** option applies to either the **Program Database** or the **Listing Display**.

### Incremental Search

To search for text strings incrementally,

1. Select the **Search** → **Program Text** from the Code Browser tool menu.
2. In the **Search for** field, enter the string for which you want to search, using wildcards (\*) or (?) as needed. The \* matches any character. The ? matches a single character.



● This field does not support [regular expressions](#).



● If you need to search for one of the wildcard characters, then escape the character with a backslash. For example, to search for any occurrence of an asterisk, you would enter \\* as the search string.

● Use the combo box next to the **Search for** field to view the list of search strings that you previously entered.



● If you have selected text within a single field, then if you invoke the dialog, it will automatically load that text into the **Search for** text box for your convenience.

3. Select the options for where in the Program to search.

- Functions –search function headers, comments, signature, variable types, variable names, and comments on variables
- Comments –search Plate, Pre-, Post-, End of Line Comments, and Repeatable; by default this check box is selected
- Labels –search Labels
- Instruction Mnemonics –search the Mnemonics of instructions
- Instruction Operands –search the Operands of instructions
- Defined Data Mnemonics –search Mnemonics of defined data
- Defined Data Values –search Values of defined data
-  The **Program Database Search Type** does not include components of [Structures](#) or [Unions](#). Use the **Listing Display Search Type** for this case. If you do want to search structures or unions, they must be open in the Code Browser.
-  If you have made a selection and it has been loaded into the *Search for* text box then the dialog will automatically select the field that the text was found in as your choice of *Field* to search. You have the option to add more or remove this selection if you wish.

4. Select whether or not to search "Other" memory blocks (blocks that not actually loaded in a running program); *Loaded Blocks* is selected by default which means "Don't search the "Other" blocks.
5. You can select or deselect the *Case Sensitive* check box depending on whether you want your search to consider case.
6. If you make a selection in the Code Browser, the *Search Selection* check box will be selected by default. If you do not want the search to be restricted to the selection, then deselect the check box.
7. Click on the **Next** or **Previous** button to search forwards or backwards in the program, (or from the *Search for* field, press the <Enter> key to search forward).
  - The start of the search operation begins at your current location in the Code Browser.
8. If a match is found, the current location in the Code Browser is moved to the location of the match. If no match is found, then a "Not found" message is displayed in the dialog.
9. If you mouse click in the Code Browser to move focus there, you can choose **Search→Repeat Text Search** to go to the next match found.

 Search operations do not "wrap" once you have reached the maximum address in memory or within a selection. Select the *Backward* direction check box to search backwards from your current location.

 For very large Programs that may take a while to search, you can cancel your search at any time. For these situations, an indicator for "search in progress" is displayed with a **Cancel** button. Click on the **Cancel** button to stop the search

 Dismissing the search dialog automatically cancels the search operation. For [search all](#), partial results are ignored if the search dialog was dismissed while the search was still in progress, therefore, the "View Results" question dialog will not be displayed.

## Search All

- To find all matches in the Program (or a selection in the program),
  1. Follow the Steps 1 through 6 for [searching incrementally](#). (Skip Step 4 as *Direction* is irrelevant in this case.)
  2. Click on the **Search All** button.
  3. The [Query Results](#) display shows all the matches.

Location	Label	Preview
004098a3	LAB_004098a3	CMP dword ptr [EBP +...]
004098a9		JNZ LAB_004098b3
004098b3	LAB_004098b3	MOV ESI,dword ptr [_...]
004098da		JZ LAB_0040998b
004098e0		JLE LAB_0040991e
004098e8		JA LAB_0040991e
004098f3		JA LAB_00409908
004098fe		JZ LAB_0040991c
00409906		JMP LAB_00409919
00409908	LAB_00409908	PUSH EAX
00409911		JZ LAB_0040991c
00409919	LAB_00409919	ADD EAX,0x8
0040991c	LAB_0040991c	MOV EBX,EAX
0040991e	LAB_0040991e	TEST EBX,EBX
00409920		JZ LAB_0040998b

Filter:



When performing a "Search All" on large Programs, the results table will appear before the search is completed. At the bottom of this window, there will be a cancel button that you can use to stop the search.

There may be multiple entries for the same address, depending on what you search for. For example, a string may appear multiple times in the same pre-comment, so you will see as many entries in the Query Results display. When you click on a row in the [Query Results](#) display, your cursor in the Code Browser is moved to that location where the match was found. So, if the match was found in an operand, then the location is moved to the matching string within the operand.

The tool has an [option](#) for the number of search results. The search will stop after this number has been exceeded. A dialog as shown below warns you of the partial results. To see more search results, [increase your search limit](#).



### Highlight Search Option

You can specify that the string found in the search be highlighted by selecting the *Highlight Search Results* checkbox on the Search Options panel. To view the Search Options, select **Edit**→**Tool Options...** from the tool menu, then select the *Search* node in the Options tree in the Options dialog. You can also change the highlight color. Click on the color bar next to *Highlight Color* to bring up a color chooser. Choose the new color, click on the **OK** button. The option for *Highlight Color for Current Match* indicates the color used to highlight the match when it occurs at the current location in the Code Browser. Apply your changes by clicking on the **OK** or **Apply** button on the Options dialog.

The highlight options also apply to [searching memory](#).



Other notes of interest on highlighting:

- Highlights are displayed for the last search that you did. For example, if you bring up the Search Memory dialog and search for bytes, that string now becomes the new highlight string. Similarly, if you invoke [cursor text highlighting](#), that becomes the new highlight string.
- Highlights are displayed only for those items that you selected to search. For example, you did not select *Labels* to search but a label matched the string you searched for. Thus, the field for that label will not be highlighted.
- Highlights are dropped when you close the search dialog, or close the query results window for your most recent search.

### **Default Search Order for *Program Database Search***

For the *Program Database Search* option, as you incrementally step, the order in which the cursor is positioned at the match in the Listing fields is as follows:

- Functions
- Plate Comments
- Pre-Comments
- Labels
- Instruction Mnemonic
- Instruction Operands
- Defined Data Mnemonics
- Defined Data Values
- End of Line Comments
- Repeatable Comments
- Post Comments

Within a Function, the order is as follows:

- Function Comments
- Function Signature
- Stack Variable Type
- Stack Variable Name
- Stack Variable Offset
- Stack Variable Comment

The [Query Results](#) display will show the search results in this default search order.



If your [Listing fields](#) are organized in a different order from the search order (e.g., Plate Comment is after the End of Line Comment), then as you search incrementally, your cursor potentially would move back and forth at the same address where there are multiple matches all at the same address. In this case, the cursor movement may look "random." This is the case only for *Program Database Search*; the *Listing Display Search* searches in the order of the displayed fields in the Listing.

Provided By: *TextSearchPlugin*

Related Topics:

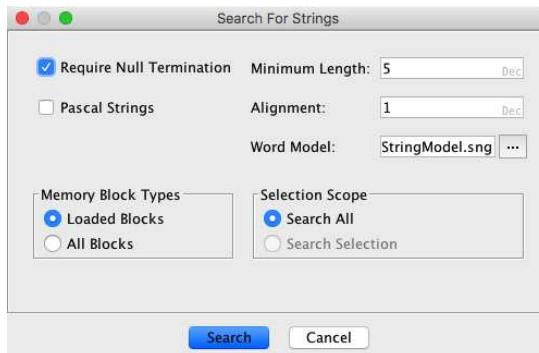
- [Query Results](#)
- [Search Limit tool option](#)
- [Data](#)
- [Listing Fields](#)
- [Search Memory](#)

## Search for Strings

*Search for Strings* searches the entire program or a specific selection for possible *Ascii* or *Unicode* strings from the main menu. The results will be displayed in a table that can be filtered and sorted and provides actions for creating strings.

### String Search Dialog

To search for strings, select **Search ➔ For Strings...**. This will bring up the String Search Dialog where you can configure the search criteria before initiating the search.



### Search Options

- **Minimum Length** –determines the shortest length of possible strings to display.
- **Alignment** –indicates the search should only return results that start at an address with the indicated byte alignment.
- **Require Null Termination** –if checked, the string resulting from the search must be null terminated. If not checked, the strings resulting from the search may or may not be null terminated.
- **Pascal Strings** –if checked, only strings that are valid pascal, pascal 255 or pascal unicode strings will be found.
- **Word Model** –specifies the Strings Analyzer model file used to detect high-confidence words. If this field is populated with a valid model file (default is *StringModel.sng*), the resulting table will contain an "Is Word" column and an option to filter by whether the string is a word or not. This field can be left blank and the word-related options will be omitted from the results table.

User-defined models should be placed in the **Ghidra/Features/Base/data/stringograms** directory.

- **Memory Block Types** –Allows the user to specify if only loaded memory blocks should be searched for the strings, or all (unloaded + loaded).
- **Selection Scope** –Allows the user to specify if the entire address space should be searched or only the user selection.
- **Search** –press this button to find all strings using the current search options.

### String Search Results

The results are displayed in tabular format. Strings can be created by selecting one or more rows from the resulting table and pressing the "Make String" button or ascii arrays with the "Make Ascii" button. An offset for the string(s) start can be specified to change the starting location(s) past the beginning of the string. String(s) can be automatically labeled.

...	Location	Label	Preview	Ascii	Stri...	Le...	Is Word
⚠	0040004d	db 21h (byte[64]e_pr...	!This program cannot be r...	ASCII	45	true	
⚠	004001e0	IMAGE_SECTION_HEADER	".text",00	ASCII	6	false	
⚠	00400207	ddw 60000020h (IMAGE...	".rdata",00	ASCII	8	true	
⚠	0040022f	ddw 40000040h (IMAGE...	"@data",00	ASCII	7	false	
⚠	00400258	IMAGE_SECTION_HEADER	".rsrc",00	ASCII	6	false	
⚠	00401dcc	LAB_00401dcc	PUSH 0xa	"j\nYQP",00	ASCII	7	false
⚠	004033a8	JZ LAB_0040340f	"teht1@",00	ASCII	7	false	
⚠	004069ab	PUSH EBP	"URPOQH,@",00	ASCII	10	false	
⚠	00407e2b	PUSH EBX	"SVWUJ",00	ASCII	6	false	
⚠	0040b1dc	s_bad_allocation_0040b1dc	ds "bad allocation",00	"bad allocation",00	ASCII	15	true
⚠	0040b1ec	s_%s_%s_0040b1ec	ds "%s %s\n",00	"%s %s\n",00	ASCII	7	true
⚠	0040b1f4	s_hates_Cheese_0040b1f4	ds " hates Cheese",00	" hates Cheese",00	ASCII	14	true
⚠	0040b204	s_likes_Cheese_0040b204	ds " likes Cheese",00	" likes Cheese",00	ASCII	14	true
⚠	0040b214	s_Infrared_Garden_Gnome_...	ds "Infrared Garden ...	"Infrared Garden Gnome",00	ASCII	22	true
⚠	0040b22c	s_Wallace_0040b22c	ds "Wallace",00	"Wallace",00	ASCII	8	true
⚠	0040b234	s_Gromit_0040b234	ds "Gromit",00	"Gromit",00	ASCII	7	true
⚠	0040b23c	s_Rabbit_0040b23c	ds "Rabbit",00	"Rabbit",00	ASCII	7	true
⚠	0040b244	s_Were_Rabbit_0040b244	ds "Were Rabbit",00	"Were Rabbit",00	ASCII	12	true
⚠	0040b250	s_Lady_Tottington_0040b2...	ds "Lady Tottington",00	"Lady Tottington",00	ASCII	16	true
⚠	0040b260	s_Lord_Victor_Quartermain...	ds "Lord Victor Quar...	"Lord Victor Quartermaine",...	ASCII	25	true
⚠	0040b286	?? 40h @	"@null",0000	"@null",0000	Uni...	8	false
⚠	0040b298	s_(null)_0040b298	ds "(null)",00	"(null)",00	ASCII	7	false
⚠	0040b2d9	?? 60h `	"h`",00	"h`",00	ASCII	7	false
⚠	0040b2fc	s_EncodePointer_0040b2fc	ds "EncodePointer",00	"EncodePointer",00	ASCII	14	true

**Table Fields**

- **Defined** –shows an icon that indicates the status of the string.
  - –indicates a string that has already been defined.
  - –indicates a string that is not defined.
  - –indicates a string that has been partially defined at some offset.
  - –indicates a string that conflicts with an instruction or some other data already defined at that address.
- **Location** –The address of the found string.
- **Label** –If one exists, the label at the location of the found string.
- **Preview** –If data or code already exists, the representation of that code or data. If no code or data exists, the undefined byte at the location of the found string.
- **Ascii** –What the string at the found location would look like if it were created.
- **String Type** –What type of string has been found. Currently, we support ascii strings, unicode strings, pascal strings, pascal 255 strings, and pascal unicode strings.
- **Length** –The number of characters in the string.
- **Is Word** –Whether the word model has determined, with high confidence, that the string is a valid word or sequence of words. NOTE: this table field is only available if the String Search Dialog 'Word Model' field contains a valid model file.

**String Filters**

There are four toggle buttons in the table window's title bar that are used to control which strings are included in the table base on the strings "defined" state.

- –toggles inclusion of **defined** strings.
- –toggles inclusion of completely **undefined** strings.
- –toggles inclusion of **partially defined** strings.
- –toggles inclusion of strings that **conflict** with an instruction or some other data at the start address.
- –toggles inclusion of **high-confidence word** strings. NOTE: this icon is only available if a String Search Dialog 'Word Model' field contains a valid model file.

**Make String Options**

- **Make String** –press this button to create either a string of the appropriate type (ascii or unicode) at the address(es) selected in the results table –if the option to automatically label is checked, a label will be placed at the beginning of the string(s).
- **Make Char Array** –press this button to create an array of chars at the address(es) selected in the results table –if the option to automatically label is checked, a label will be placed at the beginning of the char array.
- **Offset** –allows the user to specify a different starting point for the string or ascii array. The automatic label will reflect the changes in address and name. NOTE: This option is ignored for pascal strings because changing the offset would result in making the string an invalid pascal string.
- **Auto Label** –if checked, a label will be created when the string is created, that matches the string
- **Include Alignment Nulls** –if checked, strings will be created including any alignment nulls after the string, up to the alignment value.
- **Truncate if Needed** –if checked, a truncated string will be created if the string conflicts with an existing instruction or data that exists internal to the string. Otherwise, no string will be created if a conflict exists.



The "Make Strings" panel can be hidden/shown using the / toggle button at the end of the text filter.

**Refresh**

This action will cause the table to reload. The table attempts to keep the table up to date, but for efficiency reasons, not all external program changes will be accurately reflected in the strings table if those changes result in a conflict or partially defined string. A refresh will force the table to completely reload, resulting in accurate results.

**Make Selection**

*See [MakeSelection](#).*

**Selection Navigation**

*See [SelectionNavigation](#).*

Provided By: *StringTablePlugin*

**Related Topics:**

- [Search Memory](#)
- [Search Program Memory](#)
- [Search Program Text](#)
- [Searching](#)

## Search for Scalars

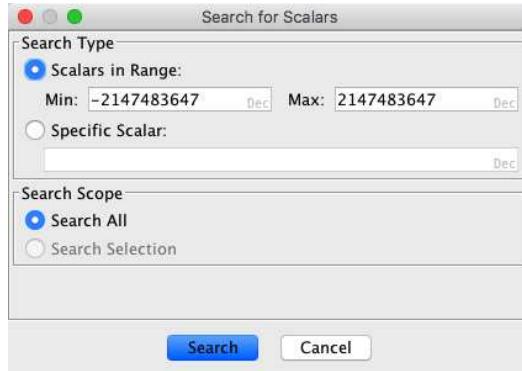
Search for Scalars locates scalar operands and values in the current program. The search is based on a value entered as hex or decimal numbers. The scalar can be in instructions, data, or structures.

### To Search for Scalars:

1. From the Tool, select **Search ➔ For Scalars...**
2. Select "Scalars in Range;" or "Specific Scalar;"
3. Once the search type is selected, enter scalars into the value fields in either decimal or hexadecimal (0x...) notation.
4. Choose "Search" to begin the search.



Once a value is entered in a text field, **Ctrl+M** toggles the value display between the decimal and hex representation of the value in that field.



### Search Options

#### Search Type

##### *Scalars in Range*

Search the program for all scalars within the given range.

##### *Specific Scalar*

Search the program for all instances of one scalar value.

#### Selection Scope

##### *Search All*

The search will search all memory in the program.

##### *Search Selection*

The search will be restricted to the current selection in the tool. This option is only enabled if there is a selection in the tool.

### Scalar Table

After the user begins a scalar search, the plugin will display a results table to the user. The table shows the address of the scalar, a preview of the item at that address, the scalar in Hex, and the scalar in signed decimal as shown in the image below:

Location	Preview	Hex	Decimal...	Function Name
00400002	dw 90h (IMAGE_DOS_HEADER.e_cblp)	90	144	
00400004	dw 3h (IMAGE_DOS_HEADER.e_cp)	3	3	
00400006	dw 0h (IMAGE_DOS_HEADER.e_crlc)	0	0	
00400008	dw 4h (IMAGE_DOS_HEADER.e_cparhdr)	4	4	
0040000a	dw 0h (IMAGE_DOS_HEADER.e_minalloc)	0	0	
0040000c	dw FFFFh (IMAGE_DOS_HEADER.e_maxalloc)	ffff	-1	
0040000e	dw 0h (IMAGE_DOS_HEADER.e_ss)	0	0	
00400010	dw B8h (IMAGE_DOS_HEADER.e_sp)	b8	184	
00400012	dw 0h (IMAGE_DOS_HEADER.e_csum)	0	0	
00400014	dw 0h (IMAGE_DOS_HEADER.e_ip)	0	0	
00400016	dw 0h (IMAGE_DOS_HEADER.e_cs)	0	0	
00400018	dw 40h (IMAGE_DOS_HEADER.e_lfarlc)	40	64	
0040001a	dw 0h (IMAGE_DOS_HEADER.e_ovno)	0	0	
0040001c	dw 0h (word[4]e_res[4][0])	0	0	
0040001e	dw 0h (word[4]e_res[4][1])	0	0	
00400020	dw 0h (word[4]e_res[4][2])	0	0	

Each element of the table is a scalar found in either data or an instruction in the program. Any new code units containing scalars added to the program will automatically appear in the table.

To bring the **Scalar Table**, choose **Window**→**Scalar Table** from the tool's menu. This table can be docked in the tool if desired.

The Scalar Table contains the following default columns:

- **Location** –displays the address of the code unit containing the scalar.
- **Preview** –displays the code unit containing the scalar.
- **Hex** –displays the scalar as a hex number.
- **Decimal (Signed)** –displays the scalar as a decimal number.
- **Function Name** –displays the name of the function containing the scalar.
- **Decimal (Unsigned)** –displays the scalar as a decimal number (this column is hidden by default).

### Scalar Table Filters

The scalar table has the following [filters](#) at the bottom of the table:

1. **Text Filter** –allows you to filter based on any text in the table.
2. **Range Filter** –allows you to filter on a range of scalars **based upon their signed value**.
3. **Column Filter** –allows you to filter on specific column values.

### Actions

#### Make Selection

*See [MakeSelection](#)*

#### Selection Navigation

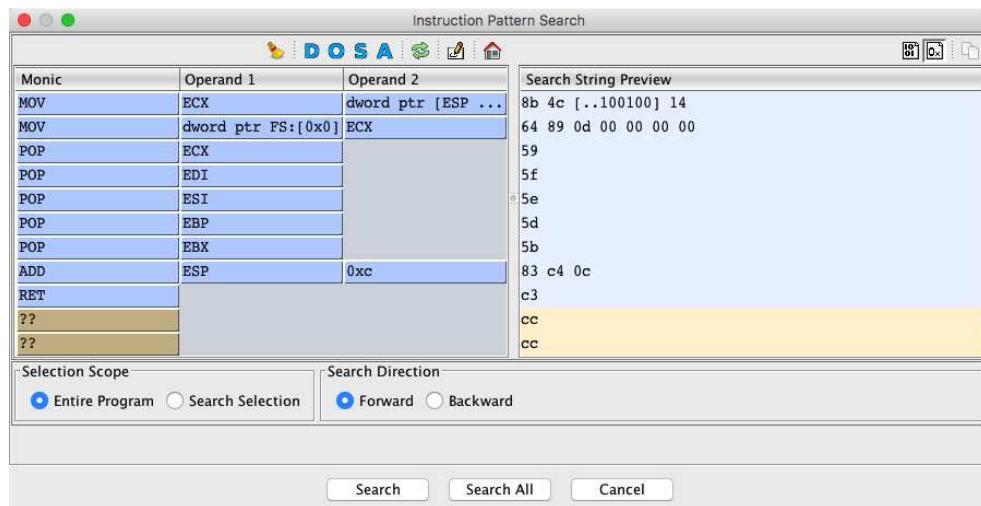
*See [SelectionNavigation](#)*

#### Remove Items

*See [RemoveItems](#)*

## Instruction Pattern Search

This dialog allows users to search the current program for specific instruction sequences. Operands and mnemonics may be masked to allow maximum flexibility in modifying the search pattern.



### Dialog Layout

The search dialog consists of two main components: the *Instruction Table* and the *PreviewTable*. The former contains all instructions selected by the user; the latter displays the string (in binary or hex) that will be used for searching.

#### Instruction Table

Monic	Operand 1	Operand 2
MOV	ECX	[ESP + 0x14]
MOV	FS:[0x0]	ECX
POP	ECX	
POP	EDI	
POP	ESI	
POP	EBP	
POP	EBX	
ADD	ESP	0xc
RET		
??		
??		

This table is populated when a selection is made in the code listing and the icon is selected. All items in the selection range will have an entry in the table, even non-instructions. Users may click on an item in the table to mask it from the final search string.

Color-coding is used to indicate the code unit type. Instructions are displayed in blue, data items are tan.

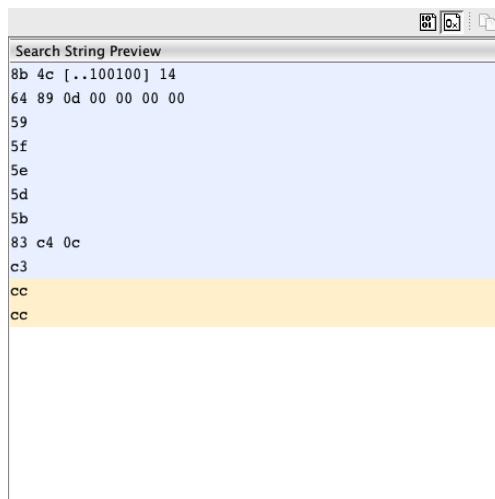
#### Instruction Table Toolbar



These tools provide ways to manipulate the Instruction Table and are discussed in detail below:

- Clears all masks.
- Masks all data (non-instructions).
- Masks all operands.
- Masks all scalar operands.
- Masks all address operands.
- Reloads the table from what is currently selected in the listing.
- Allows users to manually enter bytes to be loaded.
- Navigates in the listing to the location defined by this set of instructions.

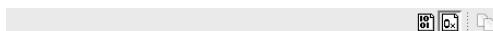
#### Preview Table



The Preview Table shows what the search string will look like, in either binary or hex format. It will change dynamically as masks are applied/removed in the Instruction Table.

- When viewing the table in binary mode, any masked bits will appear as periods [.]
- In Hex mode, if any part of a byte has masked bits, the hex value will not be shown; instead, the binary value with the masked bits will be displayed inside brackets. ie: [001..011]

#### Preview Table Toolbar

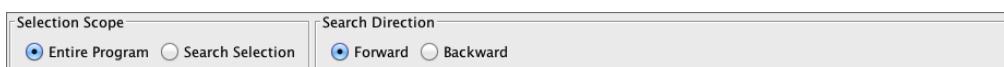


These tools provide ways to manipulate the Preview Table and are discussed in detail below:

- [ ] Switches to binary display mode.
- [ ] Switches to hex display mode.
- [ ] Copies the preview table contents to the clipboard.

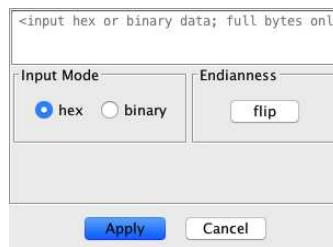
#### Search Bounds

By default any search will be run against the entire program. If you only want to search a particular range, that can be done using the options below:



- Selection Scope
- Allows the user to specify what region of the program will be searched. The default is to search for the entire program. If *SearchSelection* is chosen, whatever region is currently selected in the listing will be used as the search bounds.
- Search Direction
- Indicates whether subsequent invocations of the *Search* button will look forward or backward in the listing for the search pattern.

#### Manual Entry

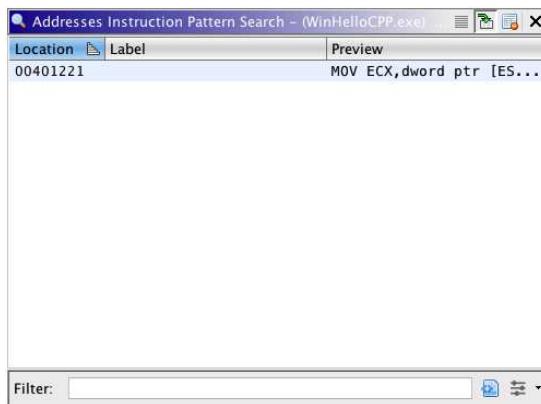


If the user clicks the button, the manual entry dialog above will be displayed. Users may enter either a binary or hex string here (full bytes, no nibbles!) and if the string represents a valid set of instructions for the loaded program, then activating the *Apply* button will cause them to be displayed in the *Instruction Table*.

#### Search Results

If the user clicks the *Search All* button, all search results will be shown in a single window, where each entry represents the starting address of a match. Clicking on an entry will take you to that spot in the listing.

If the user clicks the *Search* button, no results table will appear but the cursor will immediately move to the next match in the listing. Whether the cursor moves to the next or previous match depends on the *SearchDirection* setting.



 It should be noted that the search will look for exact byte pattern matches, not simply the mnemonic and/or operand text. eg: If you load a program and select a *RET* instruction, you can't expect to use that same search pattern to find a *RET* instruction in a different program. Unless they represent the same architecture, their byte representations will likely be different.

### Usage

The most basic usage of the search dialog is as follows:

1. Select a range of instructions in the code listing.
2. From the Tool, select **Search ➔ For Instruction Patterns**.  
The dialog will launch and be populated with the instruction set.
3. Select/deselect items in the table to mask the desired instructions.
4. Click the *Search All* button. A dialog will pop up showing all occurrences of the pattern in the program.

### Saving a Search Pattern

Users may copy and save the generated search pattern by right-clicking on the preview table and selecting one of the options; the text will be copied to the clipboard.

### Constraints

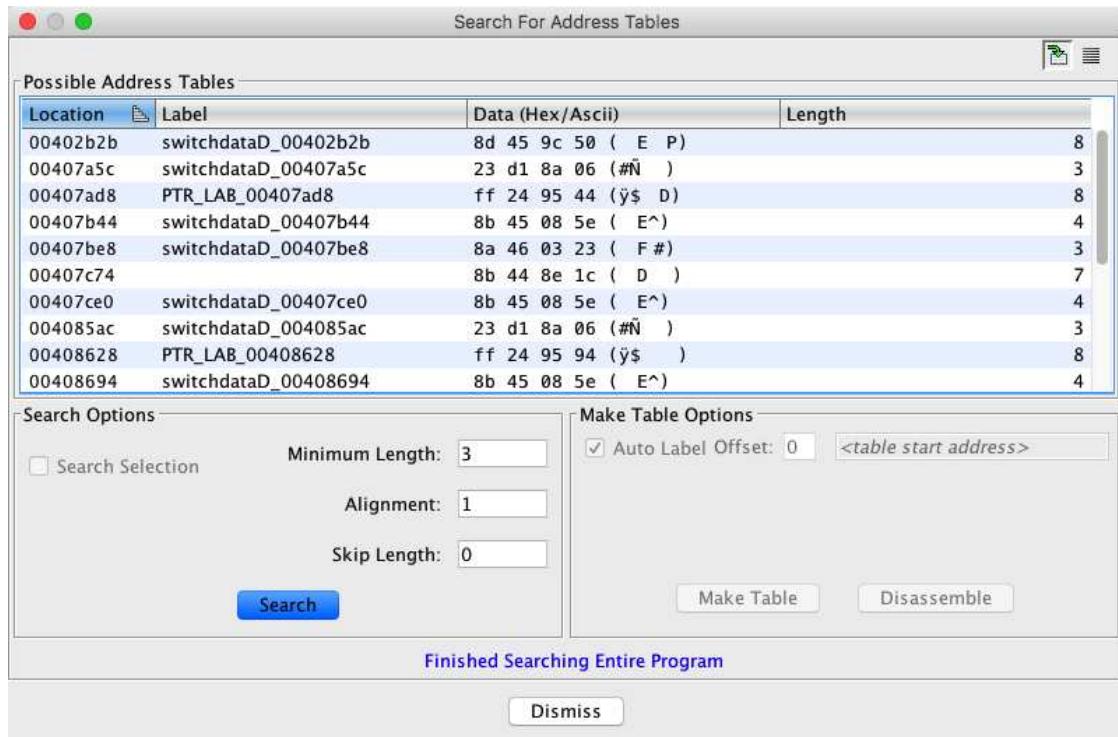
- The number of instructions that can be used in a pattern is capped at **500**.
- Only one range of addresses may be selected in the code browser. MULTIPLE SELECTIONS ARE NOT ALLOWED.

### Multiple Programs

The instruction search dialog will always operate on the program currently selected in the listing. If you select a set of instructions in Program A, then switch over to Program B and select the *Search* button, Program B will be searched for the selected instructions.

# Search for Address Tables

The **Search for Address Tables** feature searches the program for possible [address tables](#). The search can span the entire program or be limited to a selection. The results are presented in list form, allowing the user to make tables and disassemble the addresses in those tables.



## Searching for Address Tables

To search for address tables,

1. Select the **Search ➔ For Address Tables...** option
2. Select the **Search** button; the results of the search are displayed in the table.

### Search Options

- **Search Selection** –limit the search to the current selection in the Listing. (The checkbox is disabled if there is no selection).
- **Minimum Length** –determines the shortest length of the possible address tables to display.
- **Alignment** –the address tables must be aligned on the indicated byte boundary. For example, 2 indicates alignment on a word boundary.
- **Skip Length** –the number of bytes to skip between address matches. This is useful when finding addresses in structures that are known to have non-address byte sequences between the addresses.



For very large Programs that may take a while to search, you can cancel the search at any time by hitting the cancel (X) button. A progress bar is displayed as needed.

## Search Results Table

Default columns:

- **Location** –location of the possible address table
- **Label** –primary label at the location of the possible table.
- **Data (Hex/Ascii)** –Ascii and hex views of the bytes pointed to by the first element in the table.
- **Length** –the length of the possible address table

## Making Address Tables

To make an address table,

1. Select one or more rows in the **Possible Address Tables** table.
2. Select the **Auto Label** checkbox to automatically create labels at the beginning of each address table that was created, at all of the addresses contained in those tables, and at the top of the indexes to the tables if there are any.
3. If necessary, enter an offset (number of addresses to be skipped) to be added to the starting address(es) (offset multiplied by 4, the address size).
  - For one address table, the entered offset cannot be greater than one less than the length of the possible table.
  - For multiple address tables, the offset value cannot be greater than one less than the length of the smallest selected table.
  - The field next to *Offset* shows the adjusted start address for the table; this field cannot be edited. It is empty for multiple table selections.
4. Select the **Make Table** button.
5. An address table gets created at the location you chose, containing defined addresses which now point to the address created. These addresses now contain XREFs to the table entries. If an index to the table exists immediately after the table, it will get created as an array of bytes, as well.



A warning dialog is displayed if address tables could not be created due to a collision with existing data at either the start or end of possible address tables. If the auto label option is selected, you can determine from the label column those address tables that were not created.

To disassemble the address tables,

1. Select one or more rows in the **Possible Address Tables** table.
2. Select the **Disassemble** button; disassembly will begin at each address in the selected address tables.

## Actions

### Make Selection

*See [MakeSelection](#).*

### Selection Navigation

*See [SelectionNavigation](#).*

Provided by: *AutoTableDisassemblerPlugin*

Related Topics:

- [Search Memory](#)
- [Search Program Memory](#)
- [Search Program Text](#)
- [Searching](#)

## Search for Direct References

Search for Direct References will search the entire program for possible direct references to the current location or to locations within the current selection in the program. This search attempts to find the actual bytes that make up the address of the current location/selection. The search takes into account the [endianness](#) of the processor. The results are displayed in a [Query Results](#) table. The following table shows the results of searching for direct references to a location (i.e. the program doesn't have a selection). You can navigate to any resulting reference by selecting it in the table.

The screenshot shows a software interface titled "Find References to: Direct Refs to 00407b44 - (WinHelloCPP.exe)". The main area is a table with the following columns:

From Location	Label	From Preview	To Location	To Preview
00407a2f		JMP dword ptr [...]	00407b44	addr switchD_00...
00407a8d		JMP dword ptr [...]	00407b44	addr switchD_00...
00407ab3		JMP dword ptr [...]	00407b44	addr switchD_00...
00407ad1		JMP dword ptr [...]	00407b44	addr switchD_00...
00407b3e		JMP dword ptr [...]	00407b44	addr switchD_00...

Below the table is a "Filter:" input field and a set of toolbar icons.

The Search Results Table shows the following for a search on a location:

- **From Location** – address of the possible direct reference
- **Label** – primary label at the location of the possible direct reference
- **From Preview** – current definition of the code unit at the location of the possible direct reference
- **To Location** – address that is being referred to directly
- **To Preview** – current definition of the code unit at the "referred to" location

To search for possible direct references to an address,

1. Click on the address in the [Code Browser](#).
2. Select **Search ➔ for Direct References...**.
3. If the search will take a while, an "in progress" dialog pops up so that you can see the search progress, as well as cancel the search at any time.
4. A [Query Results](#) window is displayed to show the results of the search.

If you have a selection in your program when you perform the search, **Search For Direct References** will search for possible references to any of the addresses in the selection. This can be very useful for finding references into an area of memory that currently has no references to it.



To search for all possible references within the current program's memory space, press **Ctrl+A** to select the entire program before performing the search



If you use this search multiple times on different addresses without closing the window, one window will show all the results. Each result for an address is displayed when you click on the tab at the bottom of the window.



This plugin works with 16-bit, 16-bit segmented, and 32-bit programs.

### Restoring the Search Selection

If your search results came from searching on a selection, you can restore the program's selection that was used for the search. To do this, click the ▾ menu button in the Search Results button bar and select **Restore Search Results**. This will set the program selection back to what it was when you initially performed the search.

### Filtering Results Based on Alignment

Once you have search results you can filter them based on the address alignment of the **From Location**. To do this, click the ▾ menu button in the Search Results button bar, pull right on **Alignment**, and select the desired alignment (1, 2, 4, or 8). This will

limit the displayed results to those where the **From Location** is an address that matches the selected byte alignment.

## Actions

### [Make Selection](#)

See [Make Selection](#).

### [Selection Navigation](#)

See [Selection Navigation](#).

Provided by: *FindPossibleReferencesPlugin*

Related Topics:

- [Code Browser](#)
- [Query Results](#)

# Query Results Window

The *Query Results* window displays search results from various sources (plugins). The window is useful for navigating to a resultant location or selecting some number of result locations for another operation that requires a selection. The information displayed in each column may differ depending on the search operation that populated the window. However, the window's capabilities are the same. The image below shows the results of a search. You will also see the *Query Results* window when you do a [Go To](#) for matching a wildcard entry.

The screenshot shows a Windows application window titled "Search Text - 'LAB' [Listing Display Match] - (WinHelloCP...)". The window contains a table with three columns: "Location", "Label", and "Preview". The "Location" column lists memory addresses, the "Label" column lists labels found at those addresses, and the "Preview" column shows the assembly instructions at those locations. The results are as follows:

Location	Label	Preview
004098a3	LAB_004098a3	CMP dword ptr [EBP +...]
004098a9		JNZ LAB_004098b3
004098b3	LAB_004098b3	MOV ESI,dword ptr [I_...]
004098da		JZ LAB_0040998b
004098e0		JLE LAB_0040991e
004098e8		JA LAB_0040991e
004098f3		JA LAB_00409908
004098fe		JZ LAB_0040991c
00409906		JMP LAB_00409919
00409908	LAB_00409908	PUSH EAX
00409911		JZ LAB_0040991c
00409919	LAB_00409919	ADD EAX,0x8
0040991c	LAB_0040991c	MOV EBX,EAX
0040991e	LAB_0040991e	TEST EBX,EBX
00409920		JZ LAB_0040998b

Below the table is a "Filter:" input field with a dropdown menu.

Each row of the table is associated with an address or location within the program. There may be multiple entries for the same address, depending on the type of search. For example, when searching text, a string may appear multiple times in the same pre-comment. So, you will see an entry for each in the Query Results window. The title bar shows the number of entries being displayed.

The tool has an [option](#) for the maximum number of search results. The search will stop after this number has been exceeded. To see more search results, [increase your search limit](#).

## Navigation

As with most tables in Ghidra, you may change the location of the cursor within the tool by clicking on the various table cells. The location to which the cursor is moved depends upon which table cell is clicked. For example, clicking on the location column will move the cursor to the location where the match was found. Alternatively, clicking on a label in the label column will move the cursor to the corresponding label in the listing panel.

## **Marker Margins**

When searching strings or memory, a yellow arrow marker 

## **Make Selection**

A selection in the Listing can be created from the entries in the results table. The selection can also be used as input to another operation that uses the current selection, e.g., a follow-on search that is restricted to the results of the first search using the current selection.

To create a selection from all the result items,

1. Click in the results table and press **Ctrl+A**
2. Click on the  in the tool bar, or right mouse click and choose **Make Selection**.
3. The current selection will be set to the address of each result item.

To create a selection from a subset of the items,

1. Select the result items with the **Ctrl+left-mouse** or **Shift+left-mouse**
2. Press and hold the right mouse button over the results table.
3. Click on the  in the tool bar, or right mouse click and choose **Make Selection**.
4. The current selection will be set to the address of all the highlighted items.

## **Remove Items**

You can remove entries from the table via this action. Items are **only** removed from the table—no program data is changed. This can be useful when you wish to exclude results that are of no interest to you.

## **Selection Navigation**

This action causes the Listing to navigate to the address represented by a row when that row is selected. Toggling this action on allows you to use the up and down arrow keys to change the selection in the table while also navigating to that address in the Listing. You may also use this action to trigger navigation when single-clicking a table row.

## **Renaming Windows**



*see [Docking Windows –Renaming Windows](#)*

Related Topics:

- [Tool Options](#)
- [Text Search](#)
- [Go To](#)

# PDB

Ghidra offers the ability to download and apply PDB debug information for Microsoft programs. The [Download PDB File](#) feature allows users to download a PDB file that matches the user's current program, given an accessible Symbol Server. The [Load PDB File](#) feature allows users to apply a local PDB file to the current program. The *PDB Analyzer* also automatically applies PDB symbols (attempting a search for matching PDB files locally) during [Auto-Analysis](#).

## Related Topics:

- [Download PDB File](#)
- [Load PDB File](#)
- [Auto Analysis](#)

# PDB Parser Application

GHIDRA includes a bundled application *pdb.exe* for use within Microsoft Windows environments. This application is used to parse program debug information provided in the form of PDB files which are associated with specific executable programs and libraries. PDB files are produced during the compilation and linking process and may be made available by the software vendor for debugging purposes.

## Prerequisites

The native PDB parser application has been built with Microsoft Visual Studio 2017 using the 8.1 SDK to allow for possible use under Windows 7, 8.x and 10. For this application to execute properly the following prerequisites must be properly installed:

- [Microsoft Visual C++ Redistributable for Visual Studio 2017](#) with its' prerequisite updates, and
- [DIA SDK runtime support](#).

## PDB File Processing

Execution of the native PDB parser for a specified PDB file produces an XML output which is subsequently parsed by GHIDRA during PDB Analysis. If running under windows the native PDB parser may be invoked directly if the appropriate PDB file can be located locally, while on other platforms only the XML file form produced by the PDB parser is supported. Batch conversion of PDB files to XML is facilitated by the *support/createPdbXmlFiles.bat* script. In the near future GHIDRA will adopt a pure Java implementation which will eliminate the Microsoft Windows native execution issue and the use of an intermediate XML format.

## Microsoft Symbol Server

Although GHIDRA has been primarily designed to utilize locally stored PDB files during analysis, the ability to interactively download individual PDB files from a web-based Microsoft Symbol Server is also provided. This capability is accessed via the GUI while a program is open via the *File->Download PDB File...* action.

## DIA SDK Dependency

In order for the native PDB parser to work on your Microsoft Windows machine, you must:

1. Ensure you have *msdia140.dll* on your computer, and
2. Register *msdia140.dll* in the Windows registry.

### NOTES:

- The following instructions assume you have a 64-bit operating system. If you have rebuilt *pdb.exe* with a newer version of the DIA SDK you will need to register the corresponding version of the 64-bit DLL. The DIA SDK 14.0 corresponds to Visual Studio 2017.
- The PDB format is known to change over time and may be incompatible with the current *pdb.exe* parser contained within Ghidra. A Microsoft Visual Studio project is provided within the *Ghidra/Features/PDB/src/pdb* directory which will allow you to rebuild it with a newer version of Visual Studio and DIA SDK.

### Ensure you have *msdia140.dll* on your computer

First, check to see if you already have the *msdia140.dll* library installed on your system. It is generally installed with Microsoft Visual Studio 2017 when C/C++ development support is included ( may be Community, Professional, or other VS 2017 distribution package name).

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\\\DIA SDK\bin\amd64\msdia140.dll
```

This file is commonly located here, although it may be installed in other locations as well. Any 64-bit copy may be registered provided it is the correct version. There is no need to register more than one.

### Register '*msdia140.dll*' in the Windows registry

Please register 64-bit *msdia140.dll* even if you already had a copy of it on your computer since it is not registered by the Visual Studio installation process. You will need administrative rights/privileges in order to register the DLL in the Windows registry.

1. Start a command prompt as an administrator:
  - Click Windows Start menu, enter CMD in the search box to locate CMD program.
  - Right-click on CMD program and then click Run as administrator.
  - If the User Account Control dialog box appears, confirm that the action it displays is what you want, and then click Yes to continue. You may be prompted for an Admin password to elevate permissions.
2. At the prompt within the displayed CMD window, navigate to the parent folder that contains the 64-bit version of *msdia140.dll* or specify the full path of the DLL to regsvr32 command below.
3. Enter the following command to register the DLL:

```
regsvr32 msdia140.dll
```

When the registration has completed you should see a popup that indicates that "DllRegisterServer in *msdia140.dll* succeeded".

## Download PDB File

Ghidra offers the ability to download and apply a PDB file that corresponds to the program currently open in the CodeBrowser. Successful downloading requires, at a minimum, that:

1. A Symbol Server URL is available and accessible from the client or computer where you are running Ghidra.
2. The program open in the CodeBrowser is a PE file that was compiled by a Microsoft compiler.

### A Note for Windows Users

If set, Ghidra parses the `_NT_SYMBOL_PATH` environment variable that is used to specify a PDB download location and Symbol Server URL(s). The syntax for `_NT_SYMBOL_PATH` is shown below:

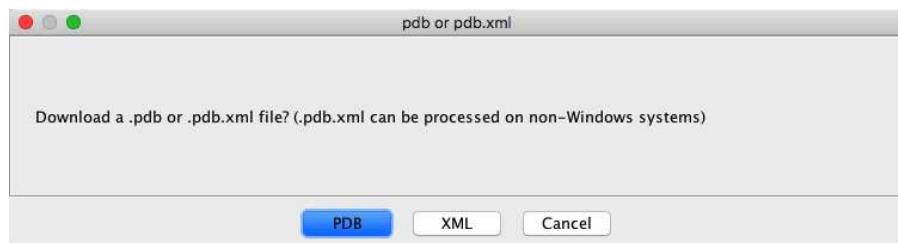
```
srv*[local symbols location]*[Symbol Server URL]
```

The `_NT_SYMBOL_PATH` location is used to pre-populate the dialog that asks for the local storage location (as long as that location is valid). The `_NT_SYMBOL_PATH` symbol server URL is used to pre-populate the dialog that asks for the symbol server location.

 Although multiple symbol server URLs can be specified in the `_NT_SYMBOL_PATH` variable, Ghidra only uses the first listed URL.

### To Download a PDB

1. From the menu-bar of a tool, select **File ➔ Download PDB File**
2. A dialog appears asking whether you want to download a PDB or XML (PDB.XML) file. Select the type of file you want to download and click OK.



 A symbol server should always have PDB files available for download. In contrast, .PDB.XML files are Ghidra-created files, and are only available to download from the symbol server if Ghidra tools have been used to create them and the server's admin has made them available. If you choose to download a .PDB.XML file and it is not found on the server, you will see a dialog message telling you so. For more information on creating and using .PDB.XML files, see the [Load PDB File](#) section.

3. Before attempting to download the file, an attempt will first be made to locate it using file and path names associated with the program. A dialog appears asking whether you want to include the PE-Header-Specified Path, which could include a Universal Naming Convention (UNC) path of a location that might not be trusted. Select OK if you want to perform this potentially unsafe retrieval.



4. A dialog appears asking where to save the downloaded file. Pick a location to store your PDB files. A common location on Windows is `C:\Symbols`.

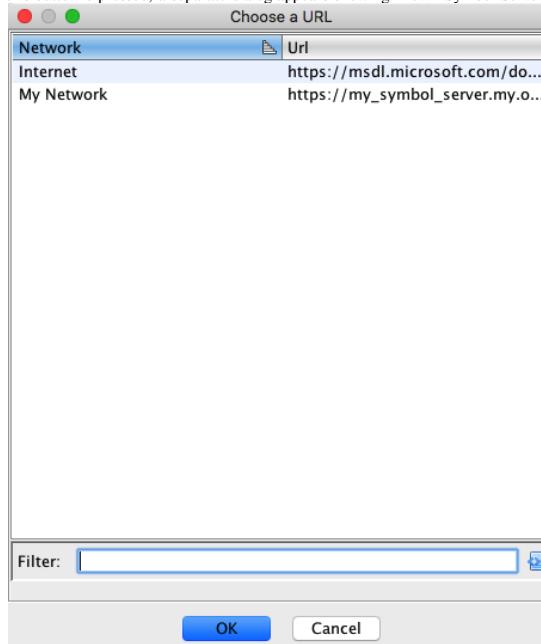
5. At this point, if a PDB file of the type you have chosen (either .PDB or .PDB.XML) already exists in the selected location, you will see a message indicating that a potential matching PDB has been found. You will then be asked if you would like to continue with the download.

- If you select "No", jump to Step 7.
- If you select "Yes", please keep the following things in mind relating to a found .PDB or .PDB.XML file:
  - If the found file is not in a directory that contains the current binary's GUID (i.e., `C:\Symbols\<pdffilename>\<GUID>`), then the file is not guaranteed to be an exact match for the current binary (when there is no GUID subfolder, a matching file is found based on expected PDB filename).
  - If there is any doubt about whether the found PDB file matches, it is a good idea to try to download the matching file, anyway (the matching file will be saved in a directory of the form `<download location>\<pdffilename>\<GUID>`).
  - If you do choose to continue to apply the found PDB file, and its GUID does not match the GUID of the current binary, you will be warned and given the option of canceling the application of the PDB file.

6. Next, you will see a dialog asking for the symbol server URL.



If a list of known URLs exists in your distribution (the file will have the extension .pdbsurl), the dialog will also include a button with the text "Choose from known URLs". When this button is pressed, a separate dialog appears showing known Symbol Server URLs.

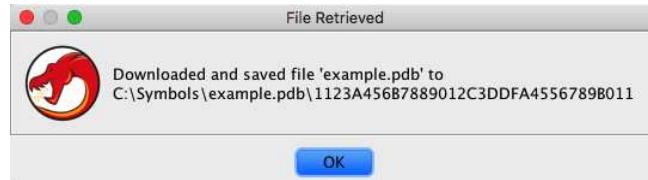


You may choose any of these URLs or manually type one in. If manually typing in a URL, be sure to include the protocol (*http* or *https*).



Always be sure to check your organization's security policy before downloading any file from the internet.

7. Next, if the Symbol Server contains a matching PDB that is the same file type that you chose earlier, it will return with a message indicating that the download was successful. The message also contains the path where you can find the downloaded file.



8. If the download was successful or an existing PDB file was found, you may be asked whether you want to apply the PDB to the program.



You will not be asked if you want to apply the found file if the file is of type .PDB and you are not on a Windows system. This is because .PDB files can only be parsed when running on Windows, while .PDB.XML files can be parsed on any Operating System.

## Troubleshooting

- If you are connecting to a Symbol Server that requires user authentication using PKI, you must first set your PKI Certificate before attempting to download from the server. See [PKI Certificate](#) for more details.

### Related Topics:

- [Load PDB](#)

# Program Tree

The Program Tree shows a program organized into folders and fragments. Fragments contain code units (instructions, defined data, or undefined data). Code Units can reside in *one and only one fragment*. A folder is a container for other folders and fragments, however, it is not analogous to a folder in a "file system." Any folder, except for the root folder, and any fragment can have *multiple* parents. When you copy a folder or fragment to another parent folder, you are not creating a new instance of the folder or fragment, but simply adding the destination folder as another parent of the copied folder or fragment. This means that if you make a change to Folder "A" (e.g., create a new fragment or folder), the change is reflected where ever Folder "A" appears in the tree, regardless of the parent. Moving code units from one fragment to another does not affect the underlying memory.

You can manipulate the tree organization by cutting, copying, and pasting folders and fragments to other folders. You can also use drag and drop to reorder the contents of a folder. You can apply algorithms to produce [organizations](#) based on a [Block Model](#). The following paragraphs describe features of the Program Tree.

## Folders and Fragments

### Create Folders and Fragments

- To create a new folder,
  1. Select a folder in the Program Tree.
  2. Right mouse-click, choose the **Create Folder** option.

3. A new folder is created with the default name of "New Folder." (A one-up number is appended to "New Folder" if that name already exists.) The cell editor for the tree is displayed immediately so that you can change the name of the folder you just created. Names are unique across all folders.

- To create a new fragment,

1. Select a folder in the Program Tree,
2. Right mouse-click, choose the **Create Fragment** option.
3. A new empty fragment is created with the default name of **New Fragment**. (A one-up number is appended to the "New Fragment" name if that name already exists.) The cell editor for the tree is displayed immediately so that you can change the name of the fragment you just created. Names are unique across all fragments.



After you are done editing the name, the icon for the fragment indicates that it is empty ( ). You can drag code units from the Code Browser and drop them onto the empty node. The icon changes to indicate that the fragment is not empty.

To create a new fragment via drag and drop,

1. Drag code units from the Code Browser.
2. Drop them onto a folder.



The default name of the fragment is the name of the first address in the set of code units that you dragged. This operation actually *moves* the code units to this fragment. If the first code unit in the set that you are dragging has a label, then the name of the fragment defaults to this label name.

## Delete Folders and Fragments

- You can delete a folder or fragment if (1) it is empty, or (2) if it exists elsewhere in the Program at some other folder. (The delete option will be disabled if this criteria is not met.)

To delete a folder or fragment,

1. Select a folder or fragment to delete.
2. Right mouse-click, and choose the **Delete** option.

- You can delete multiple folders and fragments; the option will be enabled if at least one folder or fragment in the selection can be deleted; you will get an error message for the other ones in the selection that could not be deleted.

## Rename Folders and Fragments

To rename a folder or fragment,

1. Select the folder or fragment,
2. right mouse-click, and choose the **Rename** option.
3. The cell editor for the tree is displayed.  
Enter a new name.



Duplicate folder or fragment names are not allowed, regardless of where they are in the hierarchy. If you enter a name that already exists, an error message is displayed; the name reverts back to its original name. Hit the <Esc> key to cancel editing at any time.

## **Expand/Collapse Folders**

- You can recursively expand a folder. Select a folder that has subfolders; right mouse-click and choose the **Expand All** option. All of the descendant folders are expanded.
- Similarly, you can recursively collapse a folder. Select a folder that has subfolders; right mouse-click and choose the **Collapse All** option. When you open the folder, you will see that all of its descendant folders are collapsed.

## **Merge a Folder with its Parent Folder**

You can "flatten" a folder such that all of its immediate children are moved to the folder's parent. For example, consider folder A that contains folder B; folder B contains five fragments and another folder, C. You can select folder B, right mouse-click, and choose the **Merge with Parent** option. This operation results in the five fragments and folder C in folder B get moved to folder A. Folder B is

removed.

You can make a multiple selection, however, if the selection does not contain at least one folder, the **Merge with Parent** option is disabled.

## Move Code Units to a Fragment

To move code units to an existing fragment,

1. Make a selection in the Code Browser.
2. Drag the selection over to a fragment in the Program Tree and drop it.

The code units are *moved* from the source fragment to the destination Fragment.



Drag and drop the selection on a folder to create a new fragment.

## Tool Tips on a Fragment

The tool tip on a fragment shows the address ranges that comprise this fragment. The tool tip is displayed when you let the mouse pointer hover over a fragment node in the Program Tree.

## Sort by Address or Name

You can sort the descendants of a folder by address order or by name.

To sort by address,

1. Select a folder in the Program Tree.
2. Right mouse click and choose the **Sort ➔ by Address** option.

All descendants of the folder are rearranged such that they appear in address order in the tree. This is a recursive operation.

To sort by name,

1. Select a folder in the Program Tree.
2. Right mouse click and choose the **Sort ➔ by Name** option.

All descendants of the folder are rearranged such that they appear in alphabetical order. This is a recursive operation.

Provided By: *ModuleSortPlugin*

### **Auto Rename on a Fragment**

From the Program Tree, you can automatically rename a fragment to the label at the minimum address of the fragment. Also, you can rename a fragment to any label in that fragment using the pop-up menu in the code browser.

To automatically rename a fragment,

1. Select a fragment (or fragments) in the Program Tree.
2. Right mouse click and choose **Auto Rename**.

To automatically rename a fragment to any label in the fragment from the Code Browser,

1. Position the cursor over a label field.
2. Right mouse click and choose **Rename Fragment to Label**.

Provided By: *AutoRenamePlugin*

## Select Addresses in a Folder or Fragment

To select all the addresses in a folder or fragment,

1. Select a fragment or module in the Program Tree.
2. Right mouse click and choose the **Select Addresses** option.

All addresses contained within the selected folders, fragments are shown in the Code Browser's selection.



This option is available for a multiple selection of fragments and/or folders.

Provided By: *ProgramTreeSelectionPlugin*

## Control the View in the Code Browser

The view in the code browser is controlled by the following:

### Show Folders/ Fragments in the Code Browser

You control what you see in the right side of the Code Browser tool by adding folders and fragments to your view.

- Select a fragment that is not in the view (indicated by ),
  1. Right mouse-click and choose the **Go To in View** option.
  2. The code units in this fragment now appear in the Code Browser. The fragment's icon in the Program Tree changes to  to indicate that it is part of the view. The cursor in the Code Browser is moved to the minimum address of the fragment or folder.
- Select an open folder that is not in the view (indicated by )
  1. Right mouse-click and choose the **Go To in View** option.
  2. All of the descendant folders

and fragments are added to the view.

- The folder's icon in the Program Tree changes to .
- If the folder is closed and is in the view, then the icon is .
- If a closed folder not in the view has descendants that *are* in the view, the icon is .
- When you add a folder to the view, the cursor in the browser moves to the first code unit in the first fragment of the folder. When you add a fragment to the view, the cursor in the browser moves to the first code unit in this fragment.
- You can add multiple folders and fragments to the view by selecting those folders and fragments that you want, and choosing the **Show in View** option.



The **Go To in View** option is always enabled regardless of whether the folder or fragment is in the view or not.

## **Remove Folders/Fragments from the view in the Code Browser**

- To remove folders and fragments from the view in the code browser,
  1. Select a folder or fragment that is in the view.
  2. Right mouse-click, and choose the **Remove from View** option

**from view option.**

- The icon for the folder or fragment updates to indicate that it is no longer in the view. The code browser updates its view accordingly.
- You can remove multiple folders and fragments by selecting those folders and fragments that are marked as being in the view, and choosing the **Remove from View** option.

### Replace the View in the Code Browser with other Folders/Fragments

- To replace the view in the code browser with other folders and fragments,
  1. Select a folder or fragment (or select multiple folders and fragments),
  2. Right mouse-click and choose the **Replace View** option. The code browser now shows the code units for these folders and fragments.



The program tree can be configured, via tool options, such that a double-click performs a simple navigation, or the **Replace View** action. The default behavior for a double-click is to perform the **Replace View** action.

(Double-clicking on a folder always causes it to expand if it is collapsed and to collapse if it is expanded.)

### Navigation

You can navigate to the first address of the code unit in a fragment by choosing the **Go To in View** option.

- If the fragment or folder is already in the view, the code browser navigates to the first code unit in the fragment.
- If the fragment or folder is not in the view, it is added to the view; then the code browser navigates to the first code unit in the fragment that was added.

## Cut/Copy/Paste and Drag and Drop

### Cut and Paste

- You can move fragments and folders to other folders by cutting and pasting.

1. Select a folder.
2. Right mouse-click, and choose the **Cut** option.

The icon for the folder changes to indicate the cut operation. Choose another folder that does not already contain this folder, right mouse-click, and choose the **Paste** option. The "cut" folder (and all of its descendants) should now show up in the destination folder. You can select multiple folders and fragments for cutting and

pasting.

- You can merge fragments by cutting and pasting.

1. Select a fragment (or multiple fragments).
2. Right mouse-click, and choose the **Cut** option.
3. The icon for the fragment changes to indicate the cut operation; choose another fragment.
4. Right mouse-click and choose the **Paste** option.

The code units from the "cut" fragments are moved to the destination fragment. The resulting empty fragments are removed from the program.

If you paste a folder or fragment not in the view to a folder that is in the view, then the view in the code browser will be updated to show the code units for the folder or fragment that was pasted at the destination folder.

## Copy and Paste

You can copy fragments and folders to other folders by copying and pasting.

1. Select a folder
2. Right mouse-click, and choose the **Copy** option.
3. Choose another folder that does not already contain this folder,
4. Right mouse-click, and choose the **Paste** option.

The copied fragment or folder should now show up in the destination folder.

### **Drag and Drop (Move)**

You can get the same effect of "Cut" and "Paste" by using Drag and Drop.

Drag a folder or fragment or to another folder and drop it. The folder or fragment is *moved* to this folder. If the fragment or folder already exists at a folder, then you will not get a valid drop target.

### **Drag and Drop (Copy)**

You can get the same effect of "Copy" and "Paste" by holding down the Ctrl key while dragging. (If you release the Ctrl key, the drag operation becomes a Move.)

Drag/Copy a folder or fragment to another folder; the cursor changes to indicate the copy operation. Drop the folder or fragment; a copy is made and placed in the destination folder.

As stated earlier, you can drag code units from the Code Browser view and drop them onto a folder (creates a new fragment), or onto an existing fragment (moves the code units to this fragment).

If you try to drag/copy a folder or fragment to a folder that already contains the folder or fragment, you will not get a valid drop target.

## Reorder Folder Contents using Drag and Drop

- Using Drag and Drop, you can reorder the elements within a folder. As you drag between nodes in the Program Tree, the cursor will change to indicate that a reordering operation is possible; you will see a

 solid bar between the nodes (  ). When you release the mouse, the dragged folder or fragment will be repositioned at this location, i.e., *between* the two nodes where you released the mouse.

- If you are dragging to a different parent, in addition to the reorder, the drop operation will also cause the dragged fragment or folder to be **moved** to the parent of the destination drop site. To make a copy of the folder or fragment, hold down the Ctrl key while you are dragging. The cursor will change

 to  depending on where the cursor is. If you release the mouse when the cursor indicates a potential reorder operation, the dragged folder or fragment is **copied** to the **parent** of the destination drop site, and is placed between the two nodes where you released the mouse. (Note that if the dragged folder or fragment already exists in the parent, then you will not get a valid drop target for reordering purposes.)

## Menu Enablement

If you select multiple nodes in the Program Tree, some menu items in the popup menu (right mouse click) may not be enabled if the multi-selection is not valid. A valid multi-selection meets the following criteria:

- If a folder is selected, then either all of its immediate descendants must be

selected, or none of its immediate descendants is selected.

- The root folder is not part of the selection.

A valid multi-selection will cause the **Copy**, **Cut**, **Delete**, and view options to be enabled.

The **Delete** option may be enabled for a multi-selection, however, if the delete operation is not allowed on a particular folder or fragment, you will get a notification of why the delete failed.

Provided By: *Program Tree Plugin*

Related Topics:

- [View Manager](#)
- [Program Organizations](#)
- [Code Browser](#)

# Program Organizations

You can automatically organize the structure of a Program by applying subroutine algorithms to folders and fragments. The new organization is reflected in the [Program Tree](#).

## Subroutine Modularization

This partitions the Program into subroutines that are all based on a specific [Block Model](#); the algorithm is summarized as follows:

1. Use the selected block model to gather the code blocks that overlap the address set of the selected folder or fragment.
2. For each code block create a new folder; use the code block name as the name of the folder.
  - If you selected a folder to modularize, new folders are created under the selected folder.
  - If you selected a fragment to modularize, a new folder is created using the fragment name. Folders for the code blocks are created under this folder.
3. For each code block from Step 2, use the selected block model to gather code blocks that overlap the address set of the code block.
4. For each of these code blocks from Step 3 create a new fragment; use the block name as the name of this fragment.
5. Move the code units in the code block to the new fragment.
6. Update the names of folders to include the number of elements (folders and fragments) the folder contains.

To organize a folder or fragment by modularization and block model, right mouse click on a folder or fragment in the [Program Tree](#) and choose **Modularize**  
By → **Subroutine** → <*block model name*>.

Provided By:*ModularizeAlgorithmPlugin*

## Dominance Tree Modularization

This action modularizes the program tree by creating a call tree of the code blocks and then arranges that tree by dominance such that all blocks only reachable from a parent  $p$  are children of  $p$  in the program tree.

## Complexity Depth Modularization

This action modularizes the program tree by placing code blocks in a folder that marks the longest possible call tree path that calls that block. For example, blocks at Level 0 are not called by any other code; blocks at Level 30 are blocks that have a call tree path to them that contains 30 nodes. Note that the nodes at Level 30 may have shorter paths that reach them, but the longest of all the paths is 30 nodes.

You can cancel the organization process at any time using the progress bar. Whatever fragments or folders that were generated up to the point when you canceled the operation will remain in the tree. Click on the  button to undo the organization changes.

## Default Tree Organizations

When you [import a Program](#) or [create a new tree](#), a *default tree organization* is created; a fragment is created for each

of the memory blocks. The fragment name is the same as the memory block name.

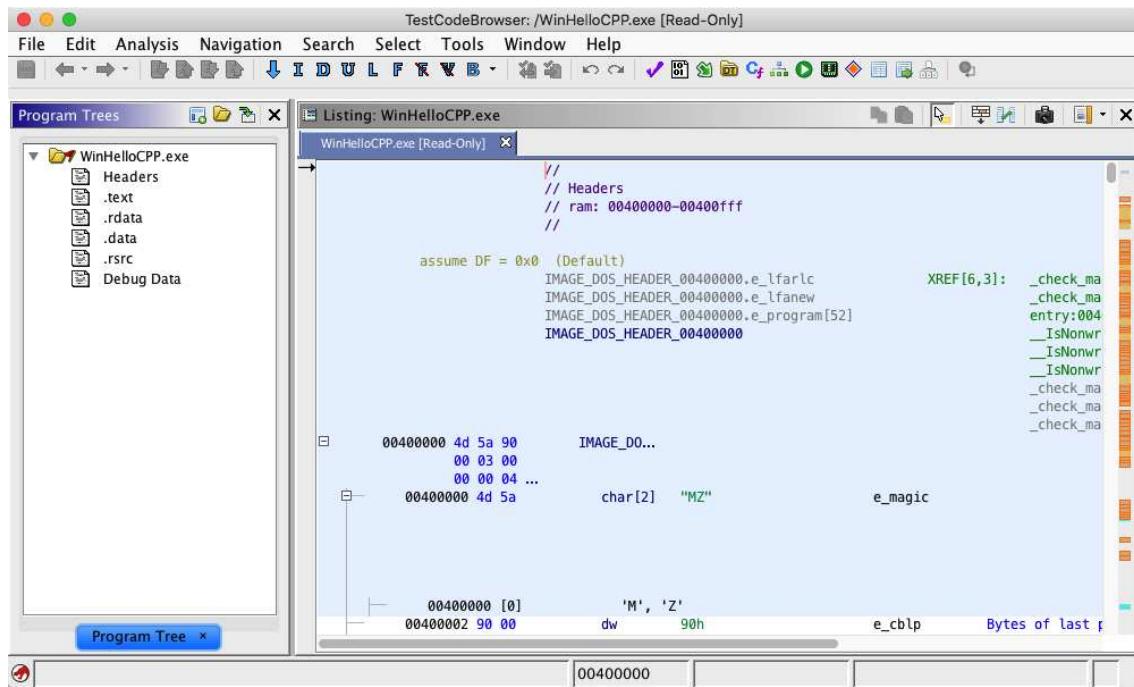
Provided by: *ProgramTreePlugin*

Related Topics:

- [Create a Default View](#)
- [Block Models](#)
- [Import a Program](#)
- [Program Tree](#)

## Program Tree Plugin

Program Trees are used to organize programs into a tree structure. Nodes within the a program tree can be used to navigate to the corresponding address in the [Code Browser](#). Also, the program tree can be used to restrict the view (set of addresses) that are displayed in the [Code Browser](#). The **Program Tree Manager** allows you to [create](#), [delete](#), [rename](#), and [close](#) program tree views.



The following paragraphs describe features of the Program Tree Manager.

### Create a Default Program Tree

A default tree has a fragment for each memory block in the program; the fragments are named the same as the memory blocks.

When you bring up a Code Browser, the Program Tree Manager (the tabbed pane on the left side of the Code Browser) shows a default view with no program open. When you open a program, the Program Tree Manager will create a tab for each tree view that is in the program. When you re-open the project, the Program Tree Manager will show the view from when you last closed the project.

You can create a new default program tree by selecting the icon. A new tab is displayed with the default name of the view, "Program Tree." If a view named "Program Tree" exists, then the name has a one-up number appended to it to ensure the name is unique, e.g., "Program Tree(1)."

Provided by: *ProgramTreePlugin*

### Open Program Tree

You can see a list of existing Program Trees in the Program by selecting the icon. Select the program tree name from the popup menu; a tab is created in the panel for this tree, if one does not already exist. The selected tree becomes the current tree in the tabbed pane.

Provided by: *ProgramTreePlugin*

### Select Fragments Corresponding to a Program Location

The icon is a toggle button that controls whether the fragment(s) that correspond to the location in the code browser should be selected in the Program Tree. **On** means to select the fragment(s) that contain the address of the location. While the button is **On**, the Program Tree will track the location in the browser by selecting the appropriate fragments. The toggle is **Off** by default.

Provided by: *ProgramTreePlugin*

#### **[Close a Program Tree](#)**

To close a program tree, right-mouse click on the tab of that program tree and choose the "Close" option. Closing a program tree does not affect the Program.

Re-open the program tree by selecting it from the list of views described above.



You cannot close the last program tree..

#### **[Rename a Program Tree](#)**

To rename a program tree,

1. Right-mouse click on the tab of the program tree.
2. Choose the "Rename" option.
3. A text field is created over the tab; the value defaults to the current view name and is selected. Enter a new name.

If another view exists with this name, a message is displayed in the status area of the tool. The list of existing views will show the new name.



If you move focus out of the edit window, the edit window is removed, and no change is made to the name.

Click the button to undo the rename.

#### **[Delete a Program Tree](#)**

To delete a program tree,

1. Right-mouse click on the tab of the program tree.
2. Choose the "Delete" option.



You cannot delete the last program tree. You must first create a new default tree, then delete your other tree.

Click the button to undo the Delete.

#### **[Change to Other Program Tree](#)**

To switch to another tree view, either click on another tab, OR select a program tree name from the list of program trees.

Provided By: *View Manager Plugin*

Related Topics:

- [Program Tree](#)
- [Program Organizations](#)
- [Code Browser](#)

# Validate Program

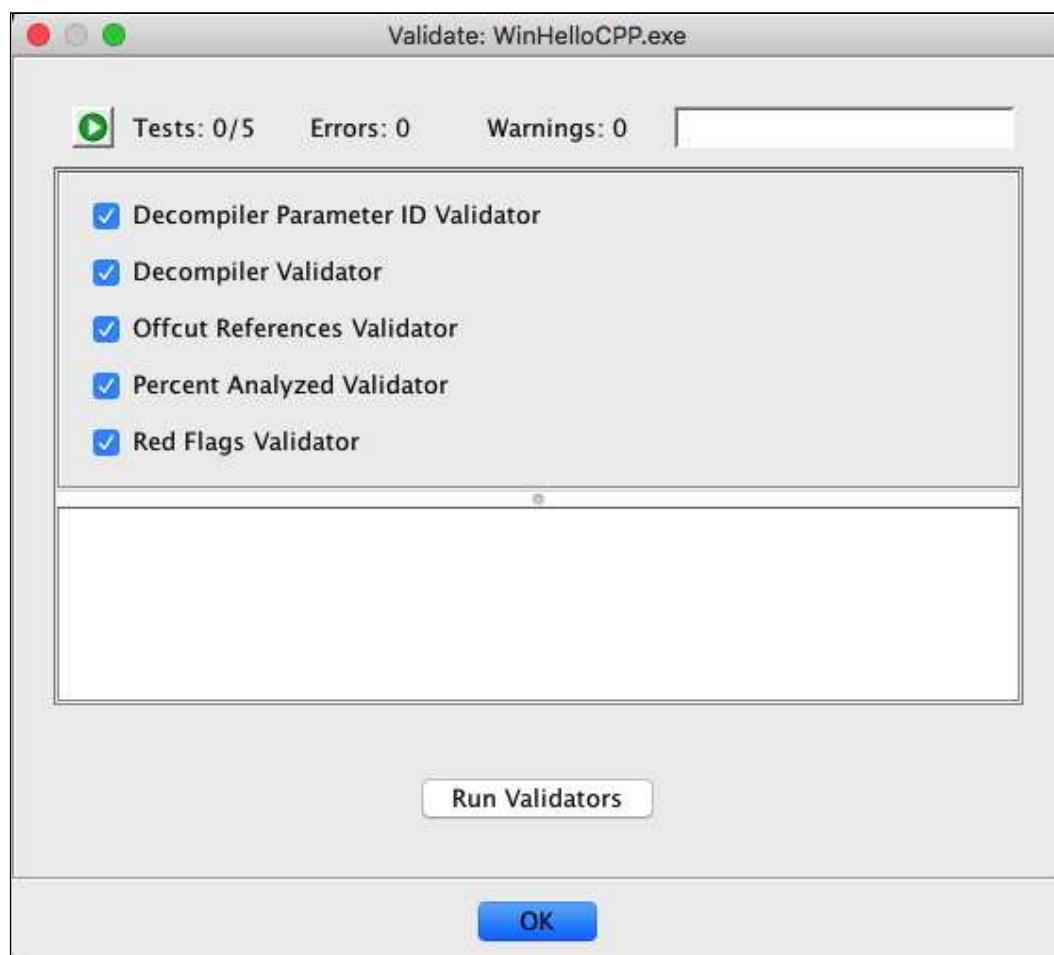
## Post-Analysis Problem Detection

After running [Auto Analysis](#), there may be problems with your program due to aggressive analyzers or strange executable patterns (functions that do not return, calling stack canary checks, etc.). Running program validators helps to show potential problem areas after analysis has run.

## Program Validator

You can launch the Validate Program window by using the **Analysis ➔ Validate {Current Program}** menu item.

This command option will open the **Validate Program** window providing a list of program validators to run.

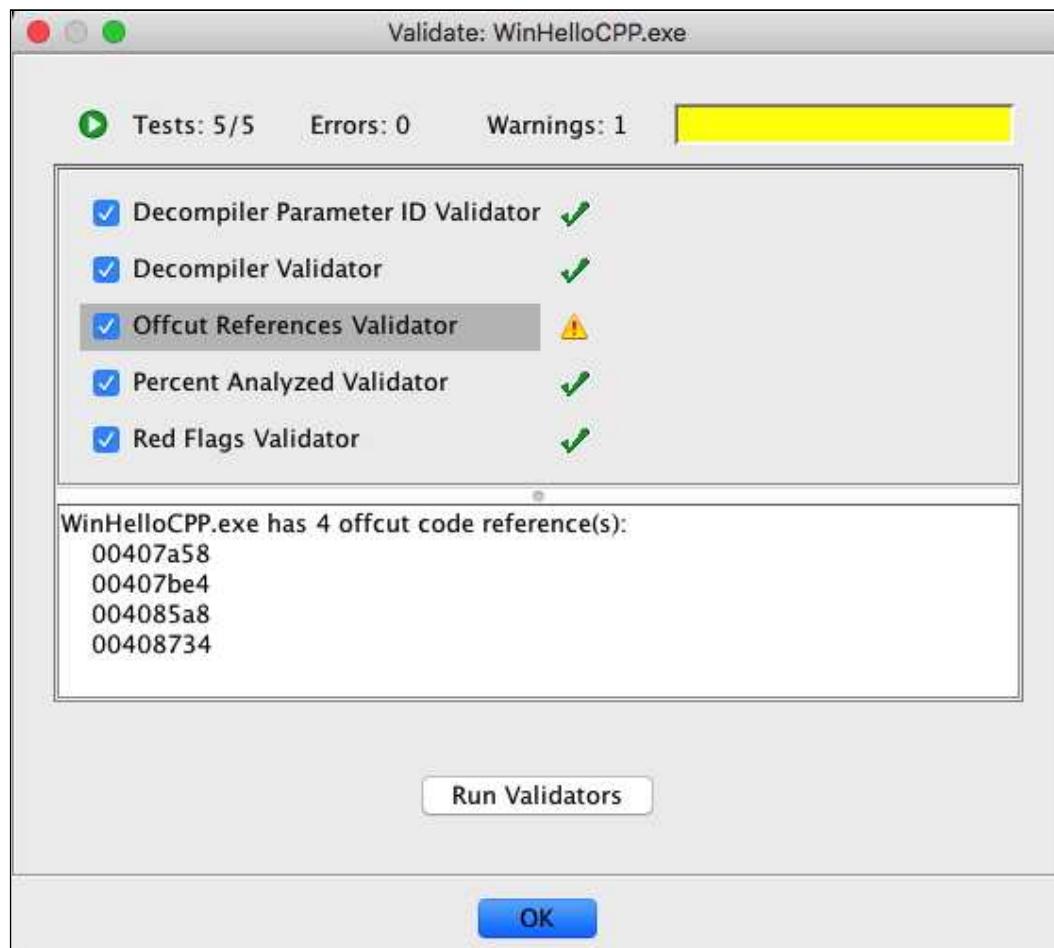


"Validate Program" Dialog

Note that each validator can be turned on or off individually (they are all on by default).

Once you press the **Run Validators** button, each selected validator will run in the order presented in the window. As they run, each will display its progress and then an icon which represents the results of the validator. This progress is displayed using a normal Ghidra task monitor and each validator can be aborted individually by pressing its stop  button.

At completion, each validator will be followed by an icon showing its results: a green check  (OK), a yellow caution  (warnings), or a red stop  (errors). By selecting each completed validator, you can see the warnings or errors encountered in the text area below the validator list. For example, this test program had one warning:



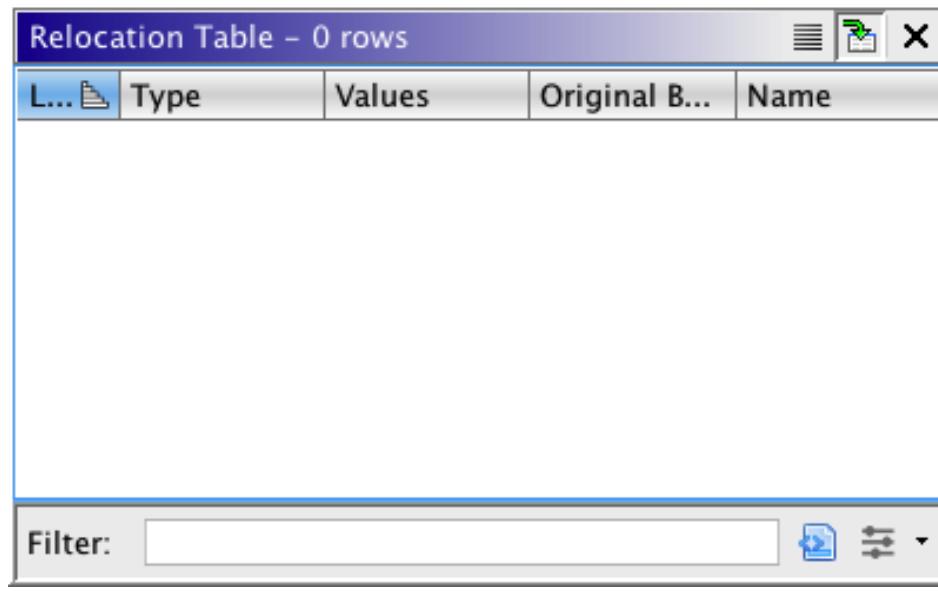
*Results after running validators*

#### Related Topics

- [Auto Analysis](#)

# Relocation Table

The Relocation Table displays a tabular view of each relocation defined in the program. Relocations are address locations that need to be updated to reflect where the program is loaded into memory. The relocation information is specific to the original format of the program and it is generated by the [Automatic Importer](#).



The screenshot shows a software interface titled "Relocation Table - 0 rows". The window has a standard title bar with icons for minimize, maximize, and close. Below the title bar is a toolbar with a refresh icon and other buttons. The main area is a table grid with five columns: "Address" (with a dropdown arrow icon), "Type", "Values", "Original B...", and "Name". A "Filter:" input field is located at the bottom left of the grid, along with a search icon and a sort icon.

Relocation Table

## Displaying the Relocation Table

From the window menu of a tool, select **Relocation Table**

## Relocation

The dialog displays a table of the following relocation information.  
The columns in the table are:

**Address** address where the  
relocation is  
defined

**Type** type of relocation  
to perform

<b>Values</b>	the values to use when performing relocation
<b>Bytes</b>	the original bytes to use when performing relocation

**Provided by:** *Relocation Table* plugin

## Ghidra Script Manager

The Ghidra Script Manager allows for rapid development of extended Ghidra functionality. Unlike conventional Ghidra plugins that require a full IDE for development, Ghidra scripts can be developed right inside of Ghidra while it is running. You can interactively change your script and immediately re-run it.

See [Ghidra Script Development](#) for details on how to write a script.

The screenshot shows the Ghidra Script Manager window. On the left is a tree view of script categories: Scripts, Analysis, Binary, Data Types, Examples (which is expanded), Functions, and Import. The main area is a table titled "Script Manager - 49 scripts (of 310)". The columns are: In T..., Status, Name, Description, Key, Category, and Modified. A filter bar is at the bottom of the table. Below the table is a panel for the selected script, "HelloWorldScript.java". It contains the code: "Writes \"Hello World\" to console.", author information (Author: Examples, Key Binding: Ctrl-Shift-COMMA, Menu Path: Help.Examples.Hello World), and a "Script Category Tree" section.

### Script Category Tree

The script category tree will organize scripts by category. If you click on a node of the tree it will only display scripts in the table that are in that category or any sub-categories. Each script defines its own category, which is arbitrary and optional. Scripts without explicitly defined categories will appear in the root "Scripts" category.

### Script Table

The script table displays information about the scripts.

The *first column* indicates if an action should be created for the script. If a script has a menu path or default key binding, then selecting this column will cause an action to be created with that menu path and/or key binding. Selecting this column for a script that has no menu path or key binding set will have no effect until the script is modified to have a menu path or key binding. Deselecting this column will also remove any key bindings that were defined via the script manager gui.

The *second column* indicates the status of the script. A blank field is a happy field. If the column contains !, then that script contains an error.

The *Filename* column indicates the filename of the script.

The *Description* column indicates the description as defined in the meta-data comment of the script.

The *KeyBinding* column indicates the key binding associated to that script. If the field is blank, then a key binding has not been assigned to the script. Setting a key binding will cause an action to get created and therefore the first column will become checked.

### Filter

The *Search Filter* allows you to narrow the list of scripts displayed in the table. Only those scripts, whose name or description contains the string that you enter as the filter, will be displayed. As you type, the table is updated to reflect the filter.

### Description Panel

The *Description Panel* allows you to view meta data about the selected script in the *Script Table*, including such things as author, description, key binding, etc.

### Script Manager Actions

#### Run Script

Runs the selected script. If the script source file is out of date, then it will first be compiled. If the compilation is successful, then the script will be run. If the script does not compile, the compilation errors will be displayed in the [Console](#) and an error icon ! will be displayed in the first column of the table.



#### Run Last Script

Runs the last run script. This action is available as a keybinding from within anywhere in the tool, whether or not the Script Manager is showing. To see the current keybinding for this action, hover over its icon in the toolbar of the Script Manager.

#### Edit Script

Edits the selected script. For more information on script meta data, see [Ghidra Script Development](#)

```

MyHelloWorldScript.java
/*
 * Writes "Hello World" to console.
 */
//@category Examples.Test
//@menupath Help.Examples.Hello World
//@keybinding ctrl shift COMMA
//@toolbar world.png

import ghidra.app.script.GhidraScript;

public class MyHelloWorldScript extends GhidraScript {
    @Override
    public void run() throws Exception {
        println("Hello World");
    }
}

```

**Refresh**

Will load the contents of the current script from the file on the filesystem. This action is useful if you have edited the script outside of Ghidra and would like to have the editor update to show those changes.

**Save**

Saves the changed script back to the original file. The *Save* option is only enabled when changes have been made.

**Save As...**

Saves the script (with any changes) to a new script file. The default directory is your home directory, however if additional script directories exist, then you will be prompted to select a directory. This new script file becomes the active script in the editor. When selecting *Save As...*, Ghidra will prompt for a filename.

**Undo**

Undo reverts the editor to the state prior to the last edit. You can undo up to 50 edits.

**Redo**

Redo returns the last edit back into the editor.

**Select Font**

Changes the font for all open editors. It will also set the default font that will be used for all future editors. The dialog allows you to specify the font type, size, and style.

**Edit Script with Eclipse**

Edits the selected script in Eclipse using the GhidraDev plugin.

Before a script can be edited in Eclipse, an Eclipse installation and workspace directory must be defined in the Tool's [Eclipse Integration](#) options.

For more information on developing Ghidra scripts in Eclipse, see [Extensions/Eclipse/GhidraDev/GhidraDev\\_README.html](#).

**Assign Key Binding**

Allows you to assign a key binding the selected script.

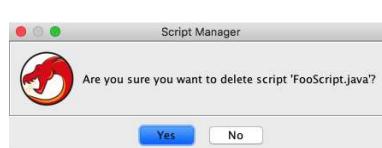
1. Click in the text field and type the key or keystroke combination that you wish to assign to the script.



The script key bindings are stored in the Tool's [KeyBinding](#) options.

**Delete Script**

Deletes the selected script. You will receive a confirmation dialog.



This is a **permanent** operation.

You cannot delete scripts in the [system directory](#), as this may affect other users. If you attempt to delete a system script, you will receive a warning dialog.

**Rename Script**

Renames the selected script. When selecting *Rename*, Ghidra will prompt for a new filename.



#### [Create New Script](#)

Creates a new empty script and displays it in a Script Editor.

If more than one [GhidraScriptProvider](#) exists, then you will have to choose what type of script to create.



See [Ghidra Script Development](#) for details on how to write a script.

#### [Refresh Script List](#)

Refreshes the script list by re-scanning the script directories.

#### [Script Directories](#)

Allows you to add and remove directories to search for scripts. The default directories are your home directory and the various system directories (e.g., \$GHIDRA\_HOME/Features/Base/ghidra\_scripts). You can save directories, but ignore them in the Script Manager dialog by selecting/deselecting the "Use" column checkbox.



#### [Help](#)

Opens the Ghidra help viewer on the GhidraScript API.

Provided by: [Ghidra Script Manager Plugin](#)

#### Related Topics

- [Key Bindings](#)

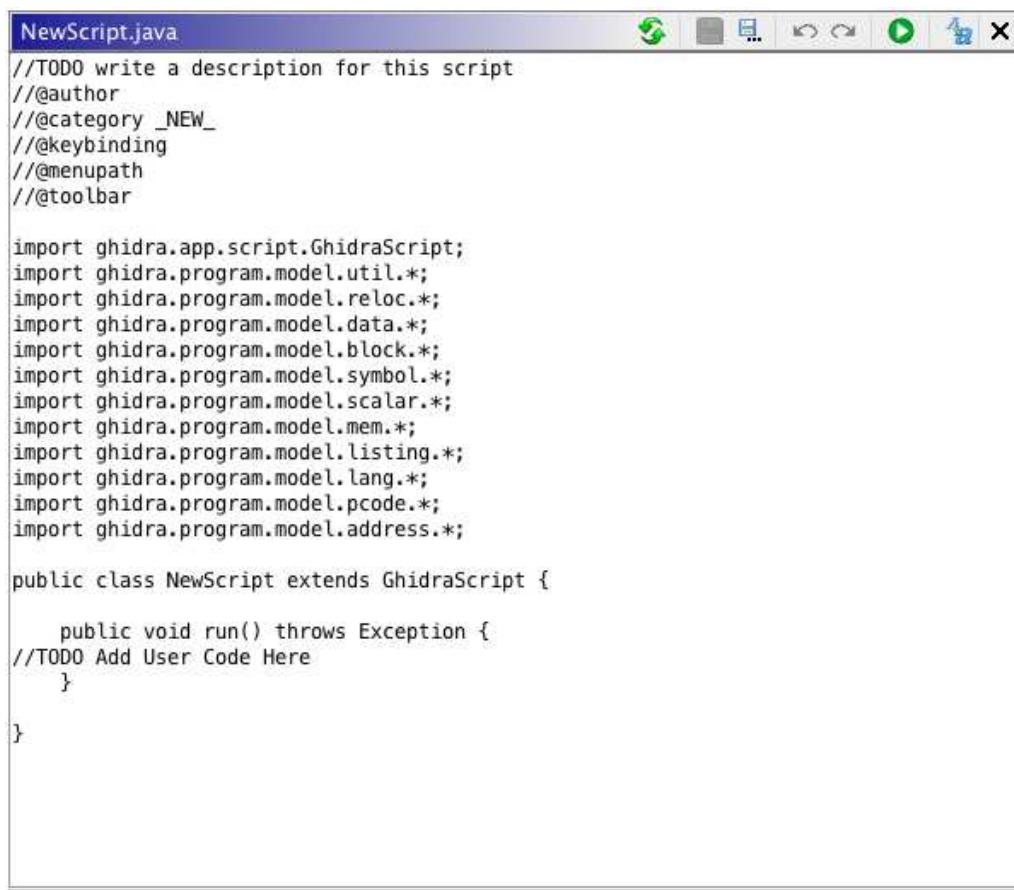
# Ghidra Script Development

Ghidra provides built-in support for creating scripts using the Java language and add-on packages to support other scripting languages.

## Creating Scripts in Java

In order to write a script in Java:

1. Your script class must extend **`ghidra.app.script.GhidraScript`**
2. You must implement the **`run()`** method. This is where you insert your script-specific code.
3. Of course if you choose Java, the Ghidra script must be written in Java. An implementation for Python (based on Jython) is also provided.



```
NewScript.java
//TODO write a description for this script
//@author
//@category _NEW_
//@keybinding
//@menupath
//@toolbar

import ghidra.app.script.GhidraScript;
import ghidra.program.model.util.*;
import ghidra.program.model.reloc.*;
import ghidra.program.model.data.*;
import ghidra.program.model.block.*;
import ghidra.program.model.symbol.*;
import ghidra.program.model.scalar.*;
import ghidra.program.model.mem.*;
import ghidra.program.model.listing.*;
import ghidra.program.model.lang.*;
import ghidra.program.model.pcode.*;
import ghidra.program.model.address.*;

public class NewScript extends GhidraScript {

    public void run() throws Exception {
//TODO Add User Code Here
    }

}
```

## Tips

Classes for performing file I/O are located in the `java.io` package.

To develop using the Eclipse IDE, instead of a simple text editor, install the developer add-on package. Next, install the GhidraDev plugin for Eclipse. If you don't already have Eclipse, you will need to install this separately. The GhidraDev extension is distributed in `<GHIDRA_INSTALLATION>/Extensions/Eclipse/GhidraDev`. Please see the README. This is highly recommended if you plan to do more than simple edits to scripts.

## Meta-data

The scripting framework supports special meta-data comments. This comment is treated specially by the script manager.

1. Each line of meta-data should start with the scripting language comment character. For example, "://".
2. The first portion is the description.
3. The first line that starts with "@" terminates the description.

The available tags are listed below:

### **@author**

This tag indicates the author of this script. It may include contact information.

### **@category**

This tag indicates the category path for this script. Category levels are delimited using the "." character.

For example, "@categorycategoryA.categoryB".

### **@keybinding**

This tag indicates the default keybinding that will activate this script. If the Script Manager is unable to interpret the keybinding, it will be ignored. The format for the key binding is ["ctrl"] ["alt"] ["shift"] [A-Z,0-9,F1-F12]. The format string is not case-sensitive.

For example:

```
@keybinding ctrl shift H
@keybinding ctrl alt shift F1
@keybinding L
@keybinding ctrl shift COMMA
```

### **@menupath**

The tag indicates the top-level menu path. Path levels are delimited using the "." character.

For example, "@menupath File.Run.My Script".

### **@toolbar**

This tag indicates a top-level toolbar button should be created to launch this script and the image to use for the button. The Script Manager will attempt to locate the image in the [Script Directories](#) and then in the Ghidra installation. If the image does not exists, a toolbar button will be created using the default Ghidra  image.

For example, "@toolbarmyScriptImage.gif".

## Supporting Other Languages

The scripting framework can be extended to support scripts written in other languages.

In order to write scripts in other languages, you must extend  
**ghidra.app.script.GhidraScriptProvider**

The methods that must be overridden are:

**File `createNewScript()`**

Creates a file containing a new template in the script language

**String `getCommentCharacter()`**

Returns the comment character used in the script language. For example, "//" or "#".

**String `getDescription()`**

Returns of description of the script language. For example, "JAVA" or "Python".

**String `getExtension()`**

Returns the file extension for script language. For example ".java" or ".py".

**GhidraScript `getScriptInstance(File sourceFile, PrintWriter writer)`**

Returns a new instance of the GhidraScript

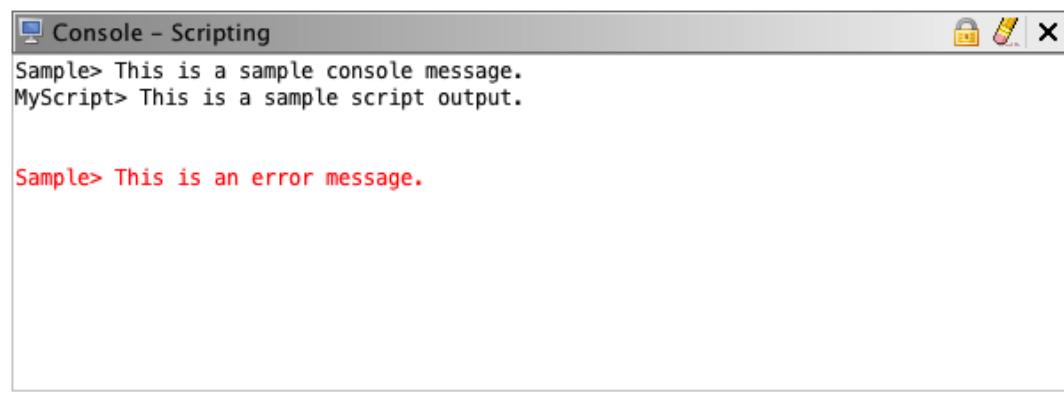
# Console

The *Console* is a generic message output component.

Any plugin can use the console service to send messages to the console. It is mostly used to display output from scripts.

The *Console* shows two different kinds of output, each in a different color.

- Standard output (black)
- Standard error (red)



## Scroll Lock

This command stops the console from automatically scrolling as new output is appended.

## Clear

This command clears all content in the *Console* component.

## Navigation

If the console contains text that represents a valid symbol or address, then you can navigate to that symbol or address by double-clicking on it. You will know if the text is valid, because the mouse cursor will change from the pointer to a hand.

Provided by: *ConsolePlugin*

Related Topics:

# Python Interpreter

The Ghidra *Python Interpreter* provides a full general-purpose Python interactive shell and allows you to interact with your current Ghidra session by exposing Ghidra's powerful Java API through the magic of Jython.

## Environment

The Ghidra *Python Interpreter* is configured to run in a similar context as a Ghidra script. Therefore, you immediately have access to variables such as `currentProgram`, `currentSelection`, `currentAddress`, etc without needing to import them. These variables exist as Java objects behind the scenes, but Jython allows you to interact with them through a Python interface, which is similar to Java in some ways.

As in Java, classes outside of your current package/module need to be explicitly imported. For example, consider the following code snippet:

```
# Get a data type from the user
tool = state.getTool()
dtm = currentProgram.getDataTypeManager()
from ghidra.app.util.datatype import DataTypeSelectionDialog
from ghidra.util.data.DataTypeParser import AllowedDataTypes
selectionDialog = DataTypeSelectionDialog(tool, dtm, -1, AllowedDataTypes.FIXED_LENGTH)
tool.showDialog(selectionDialog)
dataType = selectionDialog.getUserChosenDataType()
if dataType != None: print "Chosen data type: " + str(dataType)
```

`currentProgram` and `state` are defined within the Ghidra scripting class hierarchy, so nothing has to be explicitly imported before they can be used. However, because the `DataTypeSelectionDialog` class and `AllowedDataTypes` enum reside in different packages, they must be explicitly imported. Failure to do so will result in a Python `NameError`.

## Clear

This command clears the interpreter's display. Its effect is purely visual. It does not affect the state of the interpreter in any way.

## Interrupt

This command issues a keyboard interrupt to the interpreter, which can be used to interrupt long running commands or loops.

## Reset

This command resets the interpreter, which clears the display and resets all state.

## Keybindings

The Ghidra *Python Interpreter* supports the following hard-coded keybindings:

- **(up):** Move backward in command stack
- **(down):** Move forward in command stack
- **TAB:** Show code completion window

With the code completion window open:

- **TAB:** Insert currently-selected code completion (if no completion selected, select the first available)
- **ENTER:** Insert selected completion (if any) and close the completion window
- **(up):** Select previous code completion
- **(down):** Select next code completion
- **ESC:** Hide code completion window

## Copy/Paste

Copy and paste from within the Ghidra *Python Interpreter* should work as expected for your given environment:

- **Windows:** CTRL+C / CTRL+V
- **Linux:** CTRL+C / CTRL+V
- **OS X:** COMMAND+C / COMMAND+V

## API Documentation

The built-in `help()` Python function has been altered by the Ghidra *Python Interpreter* to add support for displaying Ghidra's Javadoc (where available) for a given Ghidra class, method, or variable. For example, to see Ghidra's Javadoc on the `state` variable, simply do:

```
>>> help(state)
#####
class ghidra.app.script.GhidraState
    extends java.lang.Object

    Represents the current state of a Ghidra tool
#####

PluginTool getTool()
    Returns the current tool.

@return ghidra.framework.plugintool.PluginTool: the current tool
-----
Project getProject()
    Returns the current project.

@return ghidra.framework.model.Project: the current project
-----
...
...
...
```

Calling `help()` with no arguments will show the Javadoc for the `GhidraScript` class.

**Note:** It may be necessary to import a Ghidra class before calling the built-in `help()` Python function on it. Failure to do so will result in a Python `NameError`

## Additional Help

For more information on the Jython environment, such as how to interact with Java objects through a Python interface, please refer to Jython's free e-book which can be found on the Internet at  
[www.jython.org/jythonbook/en/1.0/](http://www.jython.org/jythonbook/en/1.0/)

Provided by: *PythonPlugin*

Related Topics:

# Symbol Table

The *Symbol Table* displays a tabular view of each symbol currently defined in the program.

A symbol, also known as a label, is an association between a name and an address.

## Displaying the Symbol Table component

- From the menu-bar of a tool, select **Window** → **Symbol Table**
- From the tool-bar of a tool, click on the  button

Symbol Table - (Filter settings matched 613 Symbols)					
Name	Location	Symbol Type	Data Type	Namespace	Source
_rec_	00401e40	Function	_s_TryBlockMapEntry *	Global	Imported
_GetR...	004019d9	Function	undefined4	Global	Analysis
_has_o...	0040ab2d	Function	void	Global	Imported
_incon...	00404571	Function	int	Global	Analysis
_islea...	0040686b	Function	void	Global	Imported
_Jump...	00401778	Function	_LocaleUpdate *	_LocaleUpdate	Imported
_Local...	00402090	Function	void *	Global	Analysis
_malloc	00403762	Function	int	Global	Analysis
_mbto...	0040a723	Function	void *	Global	Analysis
_mem...	00408520	Function	void *	Global	Analysis
_mem...	004079d0	Function	void *	Global	Analysis
_mem...	00406920	Function	void *	Global	Analysis
_parse...	00404e99	Function	void	Global	Analysis
_raise	00407fb9	Function	int	Global	Analysis
_rand	00401689	Function	int	Global	Analysis
_realloc	004089a3	Function	void *	Global	Analysis
_siglo...	00407f78	Function	uint	Global	Analysis
_strcat...	004083fa	Function	errno_t	Global	Analysis
_strcmp	004077b0	Function	int	Global	Analysis
_strcp...	004076cd	Function	errno_t	Global	Analysis
_strcs...	00409c90	Function	size_t	Global	Analysis
_strlen	00406630	Function	size_t	Global	Analysis
_strnc...	00401b4e	Function	errno_t	Global	Analysis
_strpb...	00409ce0	Function	char *	Global	Analysis
_strtol	0040a22e	Function	long	Global	Analysis
_Unwi...	004017af	Function	void	Global	Imported
_V6_H...	00403713	Function	int *	Global	Analysis
_Valid...	004079c2	Function	int	Global	Imported
_wcto...	0040681a	Function	errno_t	Global	Analysis
_WinM...	00401e46	Function	Global	Analysis	...

The columns in the table are:

<b>Label</b>	Name of symbol.
<b>Location</b>	Address where the symbol is defined.
<b>Type</b>	Symbol type (Function, External, Class, etc).
<b>Datatype</b>	Datatype (i.e., byte, float, etc.) applied at symbol address.
<b>Namespace</b>	Namespace of the symbol; i.e., the scope.
<b>Source</b>	Indicates where the symbol name came from.
<b>Reference Count</b>	Total number of references made to this symbol.



You can sort the table on any column by clicking on the column header. The column can be sorted in ascending or descending order.



The colors for **bad references**, **entry points**, **dead code**, **offcut code**, **function names**, **local symbols**, **primary** and **non-primary** symbols correspond to the colors used in the Code Browser. Any changes you make to these colors through the [Code Browser Display](#) options will be reflected in the Symbol Table.

## Filter Text Field

The filter text field allows you to filter the list of symbols. By default it will do a "Contains" filter, but you can change that behavior to be "Starts With", "Matches Exactly", or "Regular Expression". See [Filter Options](#) for more details on the various filter text strategies.

The **Name Only** checkbox allows you to toggle whether to filter on only the name column or all the columns in the table.

The filter text field will accept basic globbing characters such as '\*' and '?' within the filter text unless the "Regular Expression" filter strategy is selected, in which case you should use standard regular expression syntax.

## Viewing Symbol References

See [Symbol References](#)

## Deleting Symbols

You can use the *Symbol Table* to delete symbols from the program.

To delete symbols:

1. Select the symbols in the Symbol Table (hold the <Ctrl> key down to add to the selection) to be deleted.
2. Right-mouse-click and select "Delete" from the popup menu, or click the  button in the *Symbol Table* toolbar.



*Notes on deleting a symbol:*

1. You can only delete a default symbol when it has zero (0) references.
2. If you delete a user-defined symbol with references, then a default symbol will automatically be created and assigned those references.
3. You can delete a non-primary symbol with references, but those references will be reassigned to the primary symbol.

## Renaming a Symbol

You can use the *Symbol Table* to rename a symbol.

To rename a symbol:

1. In the Symbol Table, double-click in the "Label" field on the row of the symbol to be renamed
2. The field will become editable
3. Enter a new name and press return
4. The new name for the symbol should display in the table and Code Browser
5. If the table is being sorted on the "Label" field, then the new name should be sorted into the table and the selection should move accordingly

## Edit External Location

You can edit the external location and associated library details for any *External Data* or *External Function* symbol within the symbol table. Right mouse click on the symbol table row and choose the **Edit External Location** action from the popup menu (see [Symbol Tree -Edit External Location](#) for more discussion on the use of the edit dialog).

## Making a Selection

You can make a selection that corresponds to the symbol addresses that are selected in the *Symbol Table*.

To make a selection:

1. Select the symbols in the Symbol Table (hold the <Ctrl> key down to add to the selection) to be added to the selection.
  2. Right-mouse-click and select "Make Selection" from the popup menu.
- Or, click the  button in the *Symbol Table* toolbar.

## Pinning a Symbol

Code, data, or function labels may be pinned which keeps them from moving to a new address in the event of a memory block move or an image base change.

To pin a label:

1. Select the symbols to be pinned in the Symbol Table (hold the <Ctrl> key down to add to the selection) to be added to the selection.
2. Right-mouse-click and select "Set Pinned" from the popup menu.

To unpin a label:

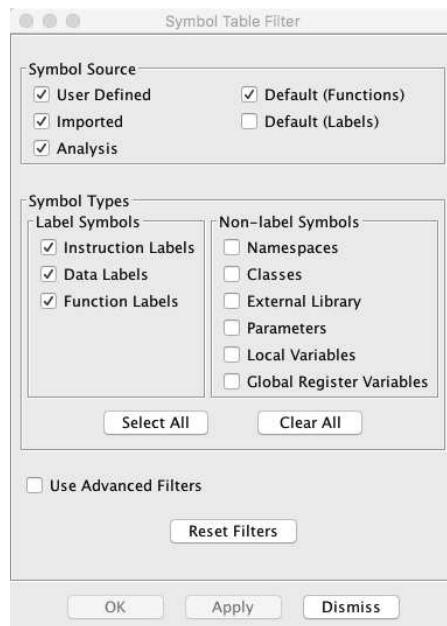
1. Select the symbols to be unpinned in the Symbol Table (hold the <Ctrl> key down to add to the selection) to be added to the selection.
2. Right-mouse-click and select "Clear Pinned Symbol(s)" from the popup menu.

## Filtering

The list of displayed symbols is determined by the current symbol table settings. These settings can be adjusted by clicking the *Filter*  button in the toolbar of the *Symbol Table* window or from the right-mouse popup menu. The displayed symbols will correspond to the selected checkboxes in the *Symbol Table Filter* dialog.



*Symbol Table Filter Dialog*



The Symbol Table Filter dialog consists of three sets of filters –Symbol Source, Symbol Types, and miscellaneous [Advanced filters](#) which are not initially shown. The Symbol Types are further divided into label symbols and non-label symbols. This grouping is for informational purposes only. For most situations, only the Source and Type filters need to be set. This will generate a query that will include all symbols that have one of the selected sources AND have one of the selected types.

**Symbol Source Filters**–this group determines which symbols (based on how they originated) should be included in the query. At least one of the source filters must be selected.

- **User Defined** – This filter includes all symbols named by the user in the query.
- **Imported** – This filter includes all symbols named by some imported information.
- **Analysis** – This filter includes all symbols created by auto-analysis that do not have default names.
- **Default (Function)** – This filter includes all function symbols that have default names.
- **Default (Labels)** – This filter includes all non-function symbols that have default names (Ghidra generally creates default-named symbols at any address that is referenced by some other location.)

**Symbol Type Filters** –This group of filters determines which types of symbols to include in the query. All symbols in Ghidra are one of the following types. At least one of these type filters must be selected.

- **Instruction Labels** –labels at addresses with instructions . Note these do NOT include labels where functions exist.
- **Data Labels** –labels at addresses with data or external labels. Note these do NOT include labels where functions exists.
- **Functions** –labels at addresses where functions have been defined (includes external functions).
- **Namespaces** –Namespace name symbols.
- **Classes** –C++ class names symbols.
- **External Library** –External library name symbols.
- **Parameters** –Function parameter name symbols.
- **Local Variables** –Function local variable name symbols.
- **Global Register Variable** –global register variable name symbols.

Use the **Select All** button to select all symbol types and the **Clear All** to de-select all types.

**Advanced Symbol Filters** –Advanced filters are used to further refine a query to only include symbols that meet various specific criteria. Each of the advanced filters only applies to a subset of the symbol types, so to use one of these filters, the appropriate symbol type filter must also be selected. Advanced filters that do not have any of their associated type filters set, are disabled. Advanced filters can be tricky to use because each filter only applies to a subset of the types and has no effect on the other selected types during the query. See the [examples](#) below for more information.

- **Externals** –Accepts only those symbols which are external.
- **Non-Externals** –Accepts only those symbols which are not external.
- **Primary Labels** –Accepts only labels that are the primary label at an address. Applies to *Labels and Functions*.
- **Non-Primary Labels** –Accepts only labels that are not the primary label at an address. Applies to *Labels and Functions*.
- **Globals** –Accepts the symbol if it is in the global namespace. Applies to *Labels, Functions, Namespaces, and classes*.
- **Locals** –Accepts the symbol if it is NOT in the global namespace. Applies to *Labels, Functions, Namespaces, and classes*.
- **Register Variables** –Accepts function parameters or local variables that are register based. Applies to *Parameters and local variables*.
- **Stack Variables** –Accepts function parameters or local variables that are stack based. Applies to *Parameters and local variables*.
- **Entry Points** –Accepts labels or functions at external entry points. Applies to *Labels and Functions*.
- **Subroutines** –Accepts labels that are "called" by some instruction. (Does not include labels where functions are defined.) Applies to *Labels*.
- **Not In Memory** –Accepts labels that are at an address not contained in memory. Applies to *Labels*.
- **Unreferenced** –Accepts labels or functions that have no references to them (also known as "dead code"). Applies to *Labels and functions*.
- **Offcut Labels** –Accepts labels that are at an address that is not the start of an instruction or data. Applies to *Labels*.

**Advanced Filters** affect the query using the following algorithm. For each symbol that matches the selected source(s) and symbol type(s):

1. Find all selected advanced filters that are appropriate for the symbol's type.
2. If no selected advanced filters are appropriate, include the symbol.
3. If at least one advanced filter is appropriate, then the symbol is included if at least one of those filters accepts the symbol.

Select the *Use Advanced Filters* checkbox to see the advanced filters.



The **Reset Filters** button sets all checkboxes back to their default states.

#### Sample Queries

##### *Example 1:*

Setup – the following checkboxes are selected:

- Symbol Source: User Defined
- Symbol Types: Instruction Labels, Data Labels, and Function Labels
- Advanced Filter: none

Result:

- All labels and functions that are "user defined" will be shown in the symbol table.

##### *Example 2:*

Setup:

- Symbol Source: User Defined, Imported, Analysis, and Default
- Symbol Types: Instruction Labels and Data Labels
- Advanced Filter: Subroutines

Result:

- All labels that are the start of a subroutine (not including functions) are displayed.

If you want to see all subroutines including those that have been defined as functions, also select the Functions type filter.

##### *Example 3:*

Setup:

- Symbol Source: User Defined, Imported, Analysis, and Default
- Symbol Types: Functions Labels and Parameters
- Advanced Filter: Stack Variables

Result:

- All functions are displayed
- All parameters that are stack based are displayed. (Register parameters have been filtered out.)

Note that the advanced filter *Stack Variables* is applicable only to Parameters, and therefore did not affect the display of functions.

##### *Example 4:*

Setup:

- Symbol Source: User Defined, Imported, Analysis, and Default
- Symbol Types: Instruction Labels, Data Labels, and Function Labels
- Advanced Filter: Primary Labels and Non-primary Labels

Result:

- All labels are displayed

Note that since all labels are either Primary labels or Non-primary Labels, selecting both of these advanced filters accomplished nothing. The results would have been the same if neither was selected.

Provided by: *Symbol Table Plugin*

Related Topics

- [Labels](#)
- [Listing Display Options](#)

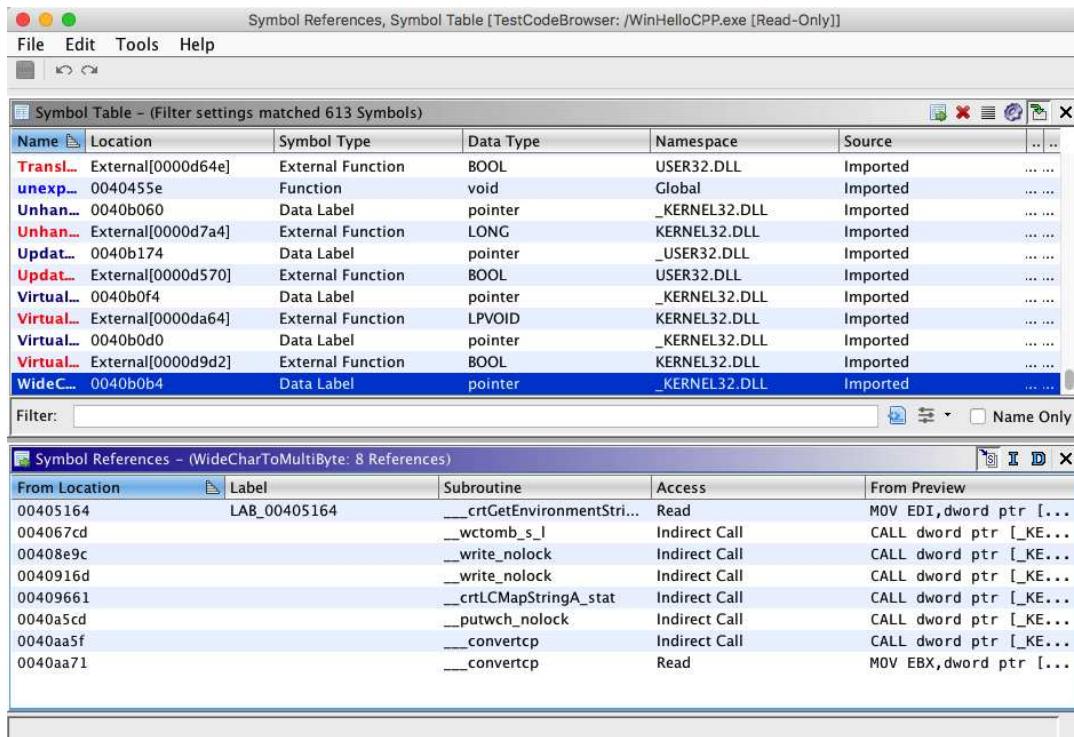
## Symbol References

Displays a table of reference-related information. When a symbol is selected in the *Symbol Table*, the *Symbol References* table updates to display the reference information for that symbol. The type of references displayed is controlled by the three toggle buttons in the toolbar: *References To Instructions From*, and *Data From*.

### Displaying the Symbol References component

- From the menu-bar of a tool, select **Window** → **Symbol References...**
- From the tool-bar of a tool, click on the 

#### References To



The screenshot shows the 'Symbol References' interface with two main tables:

- Symbol Table - (Filter settings matched 613 Symbols)**: A table listing symbols with columns: Name, Location, Symbol Type, Data Type, Namespace, and Source.
- Symbol References - (WideCharToMultiByte: 8 References)**: A table listing references with columns: From Location, Label, Subroutine, Access, and From Preview.

The 'Symbol References' table is currently active, displaying references for the symbol 'WideCharToMultiByte'. The 'Label' column lists the primary symbols being referenced, and the 'Subroutine' column lists the subroutines/functions containing these references.

Displays all references to the selected symbol. If the number of references to the symbol is less than the total number of references to the address, then the "References To" label above the References table would show "<symbol reference count> of <total reference count>". The following data columns are displayed in the *Symbol References* table:

**Address** – the address which corresponds to the code unit which references the selected symbol. Clicking on this address will generate a location event and cause the tool (e.g., Code Browser) to re-position to this address.

**Label** – displays the name of the primary symbol at the *from* address of the reference to the selected symbol. Clicking on this name will generate a location event and cause the tool (e.g., Code Browser) to re-position to the corresponding label.

**Subroutine** – displays the name of the subroutine/function containing the *from* address of the reference to the selected symbol. Clicking on this name will generate a location event and cause the tool (e.g., Code Browser) to re-position to the corresponding subroutine/function.

**Access** – indicates the type of reference. This column will display one of the following:

*RW* – read/write data access  
*Read* – read-only data access  
*Write* – write-only data access  
*Data* – general data access  
*Branch* – conditional jump  
*Jump* – unconditional jump  
*Call* – subroutine/function call  
*Unknown* – all other reference types

**Preview** – preview of the instruction or data located at *Address* which is the source of the reference.

#### Instructions From

The screenshot shows the 'Symbol References, Symbol Table [TestCodeBrowser: /WinHelloCPP.exe [Read-Only]]' window. It contains two main tables:

- Symbol Table – (Filter settings matched 613 Symbols)**: A table with columns: Name, Location, Symbol Type, Data Type, Namespace, and Source. Some rows are highlighted in blue.
- Symbol References – (\_malloc: 23 References)**: A table with columns: From Location, Label, Subroutine, Access, and From Preview. The 'Label' column shows '\_malloc' repeated 23 times, and the 'Access' column shows various types like Branch, Call, and Jump.

If the selected symbol corresponds to an entry point of a subroutine or function, all instruction references from the corresponding subroutine/function will be displayed. If the selected symbol is not a subroutine/function entry point, the list will be empty. The following data columns are displayed in the References table:

**Address** –the address which corresponds to the instruction within the subroutine/function which is the source of the reference. Clicking on this address will generate a location event and cause the tool (e.g., Code Browser) to re-position to this address.

**Label** –displays the name of the primary symbol at the *from* address of the reference to the selected symbol. Clicking on this name will generate a location event and cause the tool (e.g., Code Browser) to re-position to the corresponding label.

**Subroutine** –displays the name of the subroutine/function containing the *from* address of the reference to the selected symbol. Clicking on this name will generate a location event and cause the tool (e.g., Code Browser) to re-position to the corresponding subroutine/function.

**Access** –indicates the type of code access associated with the reference. Code access will generally be limited to a flow\* type reference unless it is the target of self-modifying code. This column will display one of the following:

*RW* –read/write data access  
*Read* –read-only data access  
*Write* –write-only data access  
*Data* –general data access  
*Branch* –conditional jump\*  
*Jump* –unconditional jump\*  
*Call* –subroutine/function call\*  
*Unknown* –all other reference types

**Preview** –preview of the instruction or data located at *Address* which is the source of the reference.

#### Data From

---

The screenshot shows the 'Symbol References, Symbol Table [TestCodeBrowser: /WinHelloCPP.exe [Read-Only]]' window. It contains two main tables:

- Symbol Table – (Filter settings matched 613 Symbols)**: A table with columns: Name, Location, Symbol Type, Data Type, Namespace, and Source. Some entries include additional details like pointer types and DLL names.
- Symbol References – (FUN\_004010e0: 6 References)**: A table with columns: From Location, Label, Subroutine, Access, and From Preview. It lists specific memory addresses and their corresponding subroutine names and access types.

If the selected symbol corresponds to an entry point of a subroutine or function, all data references from the corresponding subroutine/function will be displayed. If the selected symbol is not a subroutine/function entry point, the list will be empty. The following data columns are displayed in the References table:

**Address** –the address which corresponds to the instruction within the subroutine/function which is the source of the reference. Clicking on this address will generate a location event and cause the tool (e.g., Code Browser) to re-position to this address.

**Label** –displays the name of the primary symbol at the *from* address of the reference to the selected symbol. Clicking on this name will generate a location event and cause the tool (e.g., Code Browser) to re-position to the corresponding label.

**Subroutine** –displays the name of the subroutine/function containing the *from* address of the reference to the selected symbol. Clicking on this name will generate a location event and cause the tool (e.g., Code Browser) to re-position to the corresponding subroutine/function.

**Access** –indicates the data type.

**Preview** –preview of the instruction or data located at *Address* which is the source of the reference.

Provided by: Symbol Table plugin

# Symbol Tree

The Symbol Tree shows symbols from a program in a hierarchical view. The Symbol tree is organized by the following categories: *Externals*, *Function*, *Labels*, *Classes*, and *Namespaces*

To display the Symbol Tree, select the icon  on the tool bar, OR select the option **Window** →  **Symbol Tree**.



## Display

The root of the Symbol Tree is the *Global namespace*. A *Namespace* defines a scope, such that symbol names are unique *within* a namespace. The types of namespaces that Ghidra supports are **Global**, **External**, **Function**, **Class**, and

"generic" **Namespaces** that reside in the global namespace. Every namespace has an associated symbol.

- The **Imports** category contains symbols that represent external library namespaces which in turn contain anything considered to be "external" to the program such as external locations/functions. The library name "<EXTERNAL>" is reserved to hold those external symbols which have not yet been associated with a specific library.
- The *exports* category contains symbols that represent exported entry points into the program.
- The **Functions** category contains symbols that represent functions within the program (excluding external functions). Expand a function symbol to show its parameter and variable symbols.
- All symbols in the **Labels** category are in the Global namespace.
- The **Class** category contains class namespaces that may contain function namespaces or label symbols. Class namespaces reside in the global namespace.
- The **Namespace** category contains generic namespaces that reside in the Global namespace, and may contain class, function, label, or other namespaces.
  - The [import](#) process creates a namespace (under the *Namespace* category) for each external library it encounters. (The name is created as \_<library name>, e.g., \_USER32.DLL, to avoid naming collisions with the external library symbols, under *Externals*, e.g., USER32.DLL.)
  - Based upon the importer capability and/or analysis, external locations or functions may be associated with a *Library* namespace. You will find these external location/function symbols under the *Imports* category node.
  - Any function node (internal or external) may be expanded to show parameters. In addition, various popup function actions are available to modify the function. An external location may be converted to a function using the *Create External Function* popup action.

When a category or symbol has many elements, the Symbol Tree will show *group nodes* that represent groups of symbols in order to reduce the "clutter" in the tree. In the sample image above, the *Labels* category contains a group node (indicated by the  icon). The *group node* shows the names of the first and last symbols in the group. Long names are truncated and are displayed with "..." to indicate this. The tool tip for the node shows the complete names for the first and last symbols in the group.



The Symbol Tree does not show *dynamic* symbols, i.e., those symbols that are created due to references. Use the [Symbol Table](#) display to view dynamic symbols; turn off the [filter](#) for user-defined symbols.

Each group below lists operations that can be performed via the Symbol Tree.

## Create a Library

You can create a *Library* name used to refer to a external library. The name can then be associated with a program (or library) to allow navigation or viewing of code. Right mouse click on the Imports folder and choose the **Create Library** option. A cell editor is displayed in the tree; the name defaults to "NewLibrary." You can change the name, or press the <escape> key to exit the cell editor. If you choose the **Create Library** option from the root node, the new library name appears in the *Imports* folder in the tree.

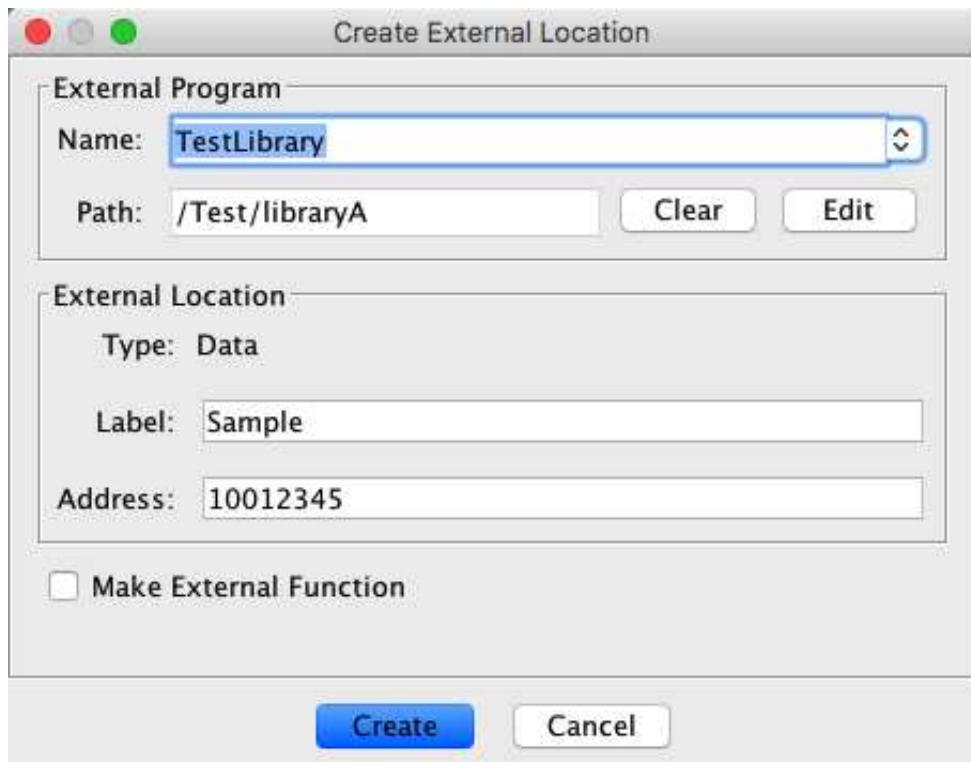
## Set External Program

You can *Set the External Program* associated with a external library. Once the name is associated with a program (or library), external references to a function or address in that library will allow navigation or viewing of code. Right mouse click on the Library node in the *Imports* folder that you want to associate with a program and choose the **Set External Program** option. A program chooser dialog is displayed which shows the programs in your project. You can then click on a program to select it and then click on the OK button to associate that program with your named library.

## Create External Location

You can create a *External Location* used to refer to a location or function in a external library. Right mouse click on the Imports node or on a particular library and choose the **Create External Location** option. If invoked from a specific library node, the *External Program Name* will default to that library for the location being created. The default library <EXTERNAL> should be used when the library is unknown. Otherwise you can type the name in the field or choose it from the combo box list. The *External Program Path*, which is the program in the project which corresponds to the named library, will be filled in if there is an associated program in the project. Otherwise, it can be selected via the *Edit* button, which displays a chooser dialog. Either a *Label*, an *Address*, or both must be specified for the External Location. If you check the *Make External Function* box, then an external function will be created. Otherwise, if you don't check the box, a non-function external location gets created. The ability to associate a data type with the location is not yet supported. Press the *Create* button to create the external location.

The following dialog shows an external function being created. It will be named "Sample" and is in "TestLibrary" at address 01001234. The library name, TestLibrary, is associated with "libraryA" in the project.



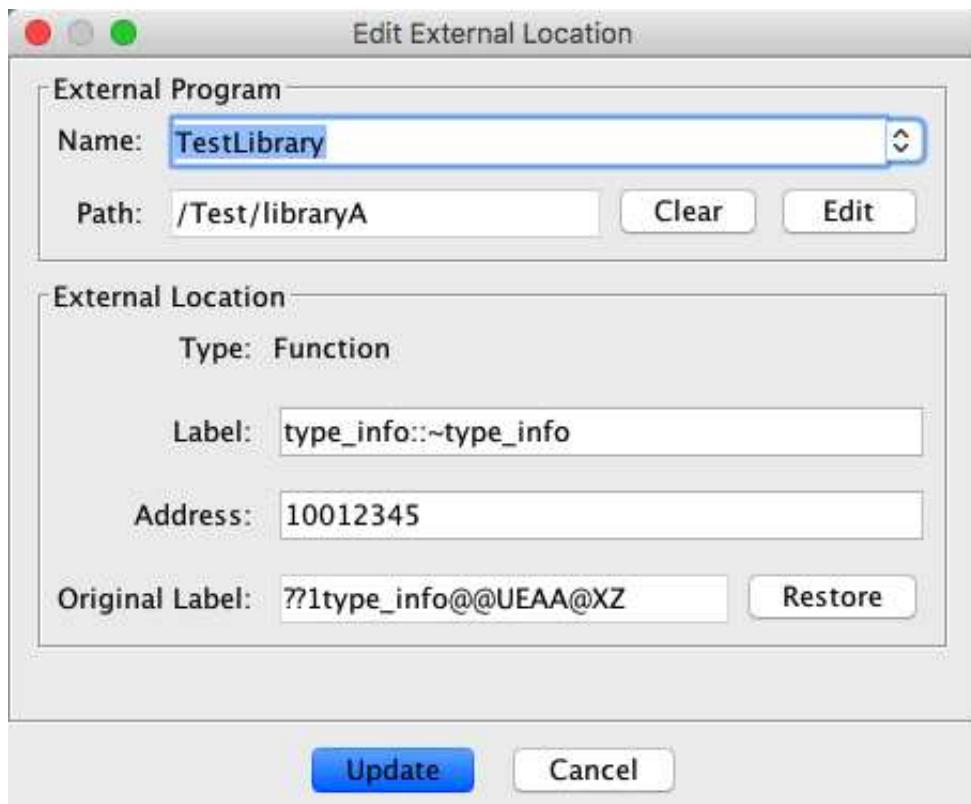
## Edit External Location

You can edit the external location and associated library details for any *External Location* symbol within the symbol tree (including locations which have been converted to an external function). Right mouse click on the external location/function node in the symbol tree under Imports and choose the **Edit External Location** action.

The Edit External Location dialog will default to the current values for the location being edited. You can change the *External Program Name* in the field or by selecting a different one from the combo box list. The *External Program Path*, which is the program in the project corresponding to the named library, will be filled in if there is an associated program. Otherwise, it can be selected via the *Edit* button, which displays a chooser dialog. You can modify the *Label* and/or *Address* for the location being edited. Either a *Label*, an *Address*, or both must be specified for the external location. Press the *Update* button to apply your edits to the external location or function.

If the location was initially created by the importer, and was subsequently demangled or changed, the *Original Label* will be specified. While you can not change this original name, you can revert the location to this label by clicking the *Restore* button. If the symbol was not an imported symbol or has never been demangled or changed the *Original Label* and *Restore* button will not be

displayed.



## Show References to

Shows all locations that reference the given symbol.

## Create a Class

You can create a *Class* namespace within the Global namespace, or within another namespace. Right mouse click on the parent namespace and choose the **Create Class** option. A cell editor is displayed in the tree; the name defaults to "NewClass." You can change the name, or press the <escape> key to exit the cell editor. If you choose the **Create Class** option from the root node, the new class appears in the *Class* category in the tree. This produces the same result as choosing the option from the *Class* node. All of the classes in the *Class* category are in the global namespace.

## Create a Namespace

You can create a namespace within the Global namespace, a Class, or another namespace. Right mouse click on the parent namespace and choose the **Create Namespace** option. A cell editor is displayed in the tree; the name defaults to

"NewNamespace." You can change the name, or press the <escape> key to exit the cell editor. If you choose the **Create Namespace** option from the root node, the new namespaces appears in the *Namespace* category in the tree. This produces the same result as choosing the option from the *Namespaces* node. All of the namespaces in the *Namespace* category are in the global namespace.

## Rename a Symbol

You can rename any symbol by right mouse clicking on the symbol and choose the **Rename** option. Enter the new name in the cell editor in the tree.



Category names and group nodes are not editable; they exist only for organizational purposes.

## Move Symbols

You can reorganize the symbol tree by using drag and drop or cut and paste. The following lists the valid move operations for each symbol type:

- Global namespace Labels can be moved to a class or other namespace; labels can be moved to a function *if the address of the label is contained in the function body.*
- Labels within a Function namespace can be moved to a class or to another namespace that is not a function namespace.
- Function namespaces can be moved to a class or other namespace.
- Class namespaces can be moved to another namespace.
- Namespaces can be moved to a class or other namespace.
- External library and functions may only be moved to another external Library or associated namespace.

### Drag and Drop

To move a symbol, drag symbols and drop on destination symbol. To drag multiple symbols, hold the Ctrl key down to add to the selection, then start dragging the mouse pointer. Release the mouse when the cursor is over the destination node.

### Cut and Paste

To use cut and paste menu options, make a selection of symbols to move, right mouse click and select the **Cut** option; select a destination symbol, right mouse click and select the **Paste** option.



Other points of interest on moving symbols:

1. Parameters and stack variables within a function cannot be moved.
2. The nodes for *Functions*, *Labels*, *Classes*, or *Namespaces* as a drop site or paste destination is equivalent to the root node (Global), as the Functions, Labels, Classes, and Namespaces are just groupings of function, label, class, and namespace symbols in the global namespace.
3. A label in the global namespace can be moved to a function *if the label's address is contained in the function body*.

## Delete Symbols

To delete a symbol, make a selection of symbols, right mouse click and choose the **Delete** option.



If you select a [group node](#) to delete, *all* the symbols in that group are deleted.

## Make a Selection

To make a [selection](#) (in the [Code Browser](#)) corresponding to selected symbols in the Symbol Tree, right mouse click and choose the **Make Selection** option. If you select a [group node](#) for the **Make Selection** option, all symbols in that group are selected in the Code Browser.

## Navigation

### Navigating to Locations within the Program



The  button controls whether a symbol is selected in the Symbol Tree when you click in the Code Browser. While the  button is selected, if the location from the Code Browser can be interpreted as a symbol, the symbol is automatically selected in the Symbol Tree. This toggle state is off by default; to turn on this feature, click on the icon. The state of the  button is saved when you close the project or exit Ghidra and restored when you reopen the project.

The Symbol Tree allows you to navigate within the Code Browser when you mouse click on a symbol in the tree. The  button applies only to locations coming from the Code Browser and has no effect when you click in the tree.

### Navigating to External Locations

If you select an external symbol (under the *Externals* folder in the symbol tree), you will navigate to an external *reference source* which has a *destination* corresponding to the external symbol (i.e., where it is being called from). To actually **go to** the external location, right mouse click on the external symbol and choose **Go To External Location**.

If the *External Program Name* associated with the external symbol has not been resolved (i.e., no Program file has been associated with the *External Program Name*) then a program chooser is displayed. Select a program to be associated with the *External Program Name* for the selected external symbol. Then this program is opened in the tool and becomes the *active* program. The current location in this opened program will be set to the location corresponding to the external symbol you had selected.

## View Qualified Names in Code Browser

To include namespace names in the display of labels and names within the Code Browser, select **Edit ➔ Tool Options...** from the tool menu. This will display the *Options* dialog. Within the *Options* tree, navigate to the folder node ***Listing Fields***. The ***DisplayNamespace*** option in the right-hand panel should be checked to include the namespace names within the Listing (see [CodeBrowser Options](#)).

Provided By: *SymbolTreePlugin*

Related Topics:

- [Symbol Table](#)
- [Selection in the Code Browser](#)
- [Listing Fields](#)
- [Labels](#)

- [External References](#)
- [Resolving External References](#)

# *Version Tracking Introduction*

Version Tracking refers to the process used by reverse engineers to identify matching code or data between different software binaries. One common use case is to version track two different versions of the same binary. Alternatively, version tracking techniques can be used to check for the presence of a particular piece of code within a given binary of interest.

Perhaps the most common version tracking scenario is when you have a binary file that you have previously analyzed, identifying important areas of interest and annotating the code with comments and labels. Suppose the software developer releases a newer version of the software including bug fixes and feature modifications. Since customers may be using the more up-to-date version, the analyst will probably want to continue evaluation with the newer version as well. However, it can be very time-consuming to have to initiate the analysis from scratch. In order to leverage the existing work, version tracking enables the reverse engineer to port comments and labels into the new context.

Perhaps the second most common version tracking scenario is where you wish to check for the presence of existing code within a given binary. As an example, given a small collection of functions, say from some library of routines or code representing known malware, you can use version tracking to search for that code in a given binary.

The remainder of this document describes high-level version tracking concepts used by Ghidra, followed by links to documents that describe how to get started with version tracking in Ghidra.

Version Tracking Concepts:

- [Session](#)
- [Associations](#)
- [Matches](#)
- [Markup Items](#)
- [Correlators](#)
- [How to Start](#)

## Version Tracking Session

A *session* is created as a result of running one of Ghidra's matching algorithms (a.k.a., a [correlator](#)) against two binaries. The newly created session is stored in the [Ghidra Project Window](#). The session records the history of any work done within that session (e.g., applying markup). Furthermore, since changes are saved, you may close and reopen a session to continue work at a later time. Sessions can be updated with new data by running additional matching algorithms at any time.

## Version Tracking Associations

An *association* is any pairing of information between the two versions of the same program, which suggests that the items correspond with one another in some way. An association is characterized by a collection of metadata including the correlating algorithm that determined the association, a memory address reference locating the items in each version, and the type of the items being associated (data or function).

Sometimes a variable or function in the source program will be associated with several variables or functions in the destination program. This happens because the version

tracking algorithm has found enough evidence to support each candidate as a possible correspondence between the two versions. When this happens, we say that they are conflicting associations. It may be that only one of the associations is exactly right or that the modularity of the program has changed sufficiently and none of the associations is quite right. Ultimately, the analyst has to inspect the actual code to make a determination about which associations represents a valid match.

Once an association is accepted by the user, any other associations which may be conflicting because they include either the same source or the same destination address will become blocked because the tool only allows one-to-one mappings. Blocked and conflicting associations which lead to other inconsistencies can be a useful way of identifying valid matches between two different versions.

## Version Tracking Matches

A *match* is an association that has been assigned a score. As a correlator finds an association it will assign that association a score, thus creating a match. The [matches table](#) contains all matches discovered by any correlators run within a given session. Users can use the score to help determine the accuracy of a given match, as not all matches created by the correlators are correct.

When you determine that a match is valid, then you can [accept](#) the match, which will block conflicting, related matches. When you apply markup for a given match, then that match is automatically accepted. Finally, you cannot apply markup for a match that has been blocked by another already accepted match.

Ghidra also has the concept of an [Implied Match](#). If you accept a function match, then Ghidra will generate implied matches for any functions called by the two functions that make up the function match.

## Version Tracking Markup Items

During analysis of a program, the analyst will develop a greater understanding of the code and will annotate the disassembly with comments, labels, data type information, and parameter and variable names to document the code and to make it more readable. Ghidra refers to all of these annotated details as markup items.

For any given match we can apply its markup items and port these annotations in an appropriate manner so that the labels and comments appear in the corresponding locations in the new binary. This is the ultimate purpose of version tracking, to retain any progress that has already been made in understanding the code and be able to proceed despite any changes to the original binary.

## Version Tracking Correlators

There are many strategies for identifying how different versions of the a program are related to each other. Any scheme or algorithm that determines these relationships is referred to as a correlator. Correlators may be based on the underlying bytes in a program, the program flow, or any other aspect of the code upon which similarities may be observed. Additional documentation is available for the specific [correlators used in Ghidra](#).

## How to Start

This list presents a few different useful links for getting started with version tracking.

- [Workflow](#)
- [Version Tracking Tool](#)
- [Version Tracking Wizard](#)

Provided by: *Version Tracking Plugin*

Related Topics:

- [Workflow](#)
- [Version Tracking Tool](#)
- [Version Tracking Wizard](#)
- [Version Tracking Matches Table](#)
- [Version Markup Items Table](#)

## Version Tracking Workflow

The goal of this document is to help users understand not only the basic process of Version Tracking, but also how and when to use the myriad of methods for tweaking that process to produce results tailored to individual user needs. The list below outlines the sections available in this document:

1. [Preconditions](#)
2. [Correlators](#)
3. [Example Work Flow](#)
4. [Workflow FAQ](#)
5. [Common Problems](#)

### Preconditions

One of the first items you run across in the Version Tracking Wizard is the [Precondition Panel](#). This panel is described in more detail as part of the [Version Tracking Wizard Help](#). However, it is also mentioned here as an important initial step in the Version Tracking process. In the past, users trying to match functions and pull relevant [markup](#) such as labels and comments into a new version of a binary, would encounter problems if one or both of the binaries were not sufficiently analyzed or had major analysis problems. Users were given no indication that these issues were a direct result of having a poorly analyzed binary. The [Precondition Panel](#) is an indication of how well your binaries have been analyzed and how well their analyses "match" each other. If this panel indicates problems, it is important to fix them before moving on or there will probably be problems with the Version Tracking process. Problems that might be encountered are missing matches and incorrect matches. In general, Version Tracking will work best if the same methods of cleaning up a binary are used and if similar numbers of functions are created.

### Correlators

Once a binary has passed the precondition checks, a user must decide which correlator (Ghidra terminology for a matching algorithm) to use, what part of the binary to run each correlator on, and what to do with the matches once they are generated. Depending on the ultimate goal of the Version Tracking user, the work flow might be different. Some users might want to identify all matching functions and pull over all related markup items as quickly as possible. Others might want to quickly identify what has changed in the new binary. Others might want to only run Version Tracking on a select number of items they care about. The goal of this section is to help users learn how to determine which correlators best suit their individual needs.

#### What Correlator Should I Use First?

One new feature of version tracking is the ability to sequentially run more than one correlator to find matching code and data and view their results simultaneously in the same Version Tracking Session. This ability to choose brings up the question of which to choose first. There are benefits to choosing certain correlators before others. In general, users should first run correlators that will find obvious matches and allow for automated matching and markup of a good portion of the destination program.

#### Exact Correlators

Some of these "obvious" correlators are ones that find matches that are unique and the same in both binaries:

- [Exact Data Match Correlator](#)
- [Exact Function Bytes Correlator](#)
- [Exact Function Instructions Correlator](#)

These correlators all report back unique and exact matches so it is almost assured that the matches are correct. Also, since they are identical in some way (bytes and/or instructions), the markup items, such as labels and comments, are going to line up correctly, allowing an automated pull of all markup at once. To automatically accept all of these items as matches and apply all related markup, the user can do a **CTRL-A** in the match table after making sure only the exact matches are in the table. Then, click on the [Apply Markup](#) icon to accept all the matches and apply all related markup. If the two binaries are very similar, this can do the majority of the matching very quickly. **It is recommended to run these three exact correlators in this order before running any other correlators.**

#### Symbol Match Correlator

Another one of the "obvious" correlators is the [Symbol Match Correlator](#). If you have unique matching symbols there is a high likelihood that the corresponding functions or data will be a match. However, it is not immediately apparent, without visiting them individually, whether these matches are exactly the same in both versions of the binary. The markup items such as labels, comments, data types, and parameters might not match up exactly so the user should use the [Accept](#) icon to accept the matches, then if necessary, individually [visually inspect and apply the markup items](#). In some cases, the user might not care about the markup. For example, if there are no user-generated markup items associated with the match, there is no reason to do anything other than "Accept" the match. Users might wonder why it is important to even accept a match for these items. One reason to accept matches, even if no markup needs to take place, is to help other correlators get better results. Some correlators use known accepted match information to make their decisions more accurate. Another reason is to save time. If there is an easy, quick way to identify matches, other correlators can ignore them and not waste time or memory trying to identify them.



**NOTE:** It is also a good idea to check for wildly differing lengths between matches in case there is the case where a wrapper function in one program has the same name as the actual function in the other program. You can either take care of length issues before running the correlator by making the length large enough to not include wrapper functions, or use the resulting match table to inspect for different lengths. An easy way to do this is to add the **Length Delta** column in the match table and sort it. A delta of zero means there is no difference in the lengths. A high delta means there is a big difference in lengths.

#### Duplicate Exact Correlators

Sometimes binaries will have more than one identical match. This often happens if there are multiple copies of strings or data in a binary. It also happens in functions that have the exact same instructions but different parameters. There are three correlators that find duplicate matches. These are:

- [Duplicate Data Match Correlator](#)
- [Duplicate Function Instructions Correlator](#)
- [Duplicate Exact Symbol Name Correlator](#)

The user won't know right away which ones match. The [Related Matches Table](#) in the Source and Destination CodeBrowsers are useful for determining which matches are correct. Once the user determines the correct matches, the markup should line up correctly and be pulled over all at once with the [Apply Markup](#) icon.

#### Other Correlators

Once you finish determining the obvious matches you can run other correlators to find the rest of the matches. The availability of these other correlators depend on which version of Ghidra is available. Please refer to the individual help for these correlators for instructions on how to use them and how to incorporate them into a work flow process. In general these other correlators do not use "exact" methods of matching so it is important to be careful when accepting matches or applying markup.

### Example Work Flow –Match All Possible Functions Between Two Binaries

1. Open an empty version tracking tool ([more info](#))
2. Enable the [Version Tracking Accept Match Option](#) to [Auto Create Implied Matches](#)
3. Temporarily set the [Match Table Filter](#) to remove [Implied Matches](#), [Accepted Matches](#), and [Blocked](#) matches from the Match Table so they are not accidentally selected in the next few steps.
4. **(Optional)** Bring up the [Version Tracking Functions Table](#) and either tab it with the [Match Table](#) or dock it in its own location. Configure it to [Show Only Unaccepted Functions](#). This will allow the user to continually see the list of functions they need to still match.

5. [Create a new session](#) by specifying your source and destination programs and then running the precondition checks. This will give you a session that initially has no version tracking matches. You will then Add to the Existing Session to begin getting matches.

6. [Add to the existing session](#) choosing the [Exact Function Bytes Correlator](#).

After the correlator is finished, in the matched table:

- Press **CTRL-A** to select all matches currently listed in the table
- Click the [Apply Markup](#) icon to accept all matches and apply all their markup items.



**NOTE:** For any of the following runs, there is an option to [Exclude Accepted Matches](#) so that the correlator being run will not report matches that are already made. It is up to personal preference whether to use this option. In large binaries it will speed up processing time. However, it is sometimes useful to have duplicate information for verification of results.

7. [Add to the existing session](#), choosing the [Exact Function Instructions Correlator](#).

After the correlator is finished, in the matched table:

- Press **CTRL-A** to select all matches currently listed in the table
- Click the [Apply Markup](#) icon to accept all matches and apply all their markup items.

8. [Add to the existing session](#), choosing the [Exact Data Match Correlator](#).

After the correlator is finished, in the matched table:

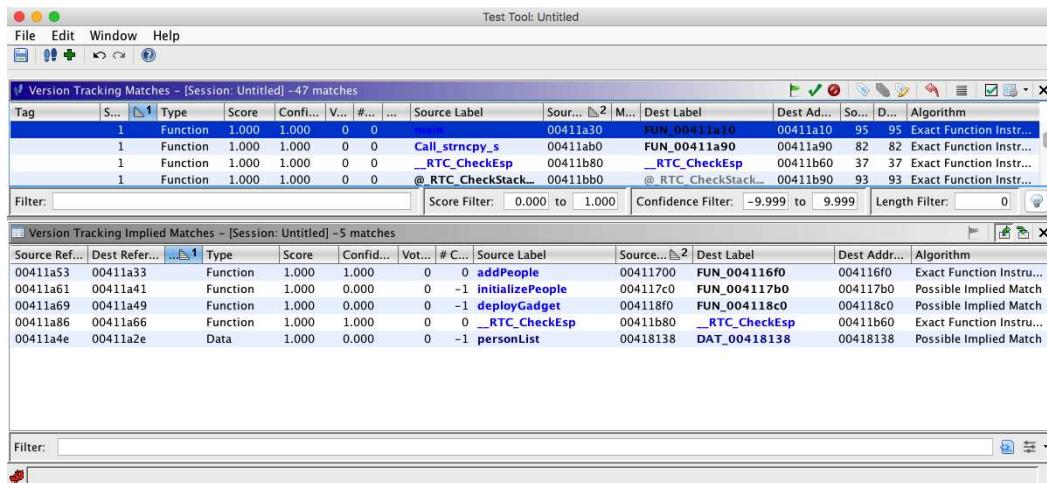
- Press CTRL-A to select all matches currently listed in the table
  - Click the **Apply Markup** icon to accept all matches and apply all their markup items.
9. Add to the existing session, choosing the **Symbol Match Correlator**.  
After the correlator is finished, in the matched table:
- First use the **Match Table Filter** to only show Imported and Analysis symbol types (this is assuming these types do not have user generated parameters or comments)
  - Press CTRL-A to select all matches currently listed in the table
  - Choose Accept to accept the matches but not pull over any markup – there probably isn't any user markup and actually it is better to take the newest analysis markup
  - Edit the filter to show all symbol types again
  - If desired, inspect to see if there are other symbol matches to individually or automatically match depending on score/confidence/etc...
10. Add to the existing session, choosing the **Duplicate Function Instruction Match Correlator**
- Use the **Related Matches Tables** to figure out which ones match
  - Use the **Apply Markup** icon to accept and apply markup for each match individually.
11. Add to the existing session, choosing the **Duplicate Data Match Correlator**
- Use the **Related Matches Tables** to figure out which ones match
  - Use the **Apply Markup** icon to accept and apply markup for each match individually.
12. Run any other correlators available but don't do anything with their matches yet.
- Reset the **Match Table Filter** to show **Implied Matches** and **Blocked** matches in the Match Table.
  - Sort the Match Table in various ways to help determine any likely matches. Here a list of useful ways to sort the table
    - Primary sort by **Score** and secondary sort by **Confidence** – if both are high it is a good indication of a good match
    - By Votes – high number of votes means the match has many of the same Implied Match suggestions
    - By Length Delta – 0 implies matching length Source and Destination match
    - By Source Address to see if more than one algorithm reports the same match
  - To help determine valid matches, use the **Related Matches Tables** to see all correlated matches for a particular match item.
  - Use the **Accept** icon to individually accept matches once they are determined.
13. Use the **Version Tracking Functions Table** to see what is left and manually match them using this table.
14. Use the **Markup Table** to inspect and manually accept or apply any leftover interesting markup items in the matched functions. Use the **Status** column in the Match Table to determine which markup items are not finished. Users should set the filters on the Match Table so that all matches are visible again to make sure none are missed.

 NOTE: This is only one sample workflow for matching all possible functions. In general, the exact, one-to-one correlators should be run first and the rest can be used in any order. Users will come up with their own preferences for workflow as they get used to the tool.

## Workflow FAQ

### What are Implied Matches?

An Implied Match is a match that is found when a function match is Applied or Accepted and operand references in the matched functions apparently point to a matching function or matching data. For example, in the matched function pictured below, it is apparent that FUN\_004011a0 in the Destination program is probably a match to the function addPerson. Also notice that the string s\_Lady\_Tottington matches the string s\_Lady\_Tottington. Notice that the **Implied Match Table** below the matching functions, lists this function and string and several other references to strings as Possible Implied Matches. Users can use this table to make matches from Implied Matches. This will cause a match to be placed in the Match table with the type Implied Match. However, the user still has to accept or apply the match. There is also an **Version Tracking Accept Match Option** to automatically generate Implied Matches whenever a function match is Applied or Accepted. The user still has to Accept or Apply the matches. The determination to use the automatic generation of Implied Matches or the Implied Match Table depends on how likely the match is to be correct and how closely aligned the match listings are. For matches found with the exact one-to-one correlators, it is relatively safe to turn on the auto option. For other correlators, it is safer to use the table.



Tag	Source Ref...	Dest Refer...	Type	Score	Confid...	Vots...	# C...	Source Label	Source...	Dest Label	Dest Addr...	So...	D...	Algorithm
1			Function	1.000	1.000	0	0	addPerson	00411a30	FUN_00411a10	00411a10	95	95	Exact Function Instr...
1			Function	1.000	1.000	0	0	Call_stncpy_s	00411ab0	FUN_00411a90	00411a90	82	82	Exact Function Instr...
1			Function	1.000	1.000	0	0	_RTC_CheckEsp	00411b80	_RTC_CheckEsp	00411b60	37	37	Exact Function Instr...
1			Function	1.000	1.000	0	0	@ RTC_CheckStack...	00411bb0	@ RTC CheckStack...	00411bb0	93	93	Exact Function Instr...

Source Ref...	Dest Refer...	Type	Score	Confid...	Vots...	# C...	Source Label	Source...	Dest Label	Dest Addr...	Algorithm
00411a53	00411a33	Function	1.000	1.000	0	0	addPeople	00411700	FUN_00411f60	00411f60	Exact Function Instr...
00411a61	00411a41	Function	1.000	0.000	0	0	initializePeople	004117c0	FUN_004117b0	004117b0	Possible Implied Match
00411a69	00411a49	Function	1.000	0.000	0	-1	deployGadget	004118f0	FUN_004118c0	004118c0	Possible Implied Match
00411a86	00411a66	Function	1.000	1.000	0	0	_RTC_CheckEsp	00411b80	_RTC_CheckEsp	00411b60	Exact Function Instr...
00411a4e	00411a2e	Data	1.000	0.000	0	-1	personList	00418138	DAT_00418138	00418138	Possible Implied Match

### What do the Scores and Confidence Columns Mean?

The **Score** is an indication of how similar two potential matches are. For example, if two functions are matched with the Exact Function Bytes Correlator, the score will be 1.0. The **Confidence** is an indication of how likely the two potential matches are actually a match. For example, if two functions are matched with the Exact Function Bytes Correlator, the Confidence will be 1.0 because this correlator only reports one-to-one matches. However, if two functions are matched with the Duplicate Exact Function Instruction Correlator, the Confidence value will be lower because this correlator reports two or more matches. Users should look at both of these indicators to help decide whether to Accept, Apply, Reject, or Ignore a match.

**It is important to note that score and confidence values cannot be reliably compared between different correlators.** A confidence value of 0.8 for one correlator might be theoretically higher than a confidence value of 0.9 for another, because each correlator computes similarity and confidence differently.

### How Do I Know Which Functions Have Not Been Matched Yet?

Bring up the **Windows->Version Tracking Functions Table**. By default, two separate lists of functions from both programs are shown. To see which functions in both programs have no matches generated by any correlator so far, click on the black triangle and choose **Show Only Unmatched Functions**. To see which functions in both programs have no matches accepted by the user (includes functions with and without a correlator generated match) choose **Show Only Unaccepted Functions**.

### Why Do I See The Same Match More Than Once in the Match Table?

If a match is found by more than one correlator, it will show up in the table for each correlator. This allows the user to gain more confidence in a match. It can clutter up the table, though, so users can filter the table based on algorithm and many other ways. See the **Matches Table** help for more information on how to filter out information from the table.

### What Part of the Binary Should I Run The Correlators On?

This is dependent on what the user is trying to do and which correlator is being run. Most correlators may be run on a subset of the binary. Others correlators must run on the entire binary. Those that allow subsets will give the user a choice when that correlator is chosen. If a user has specific items they are interested in finding matches for, they can run most correlators on those items only without seeing results for the entire binary. Other users want to match all possible items. Also, once items have been accepted by a user, users have the choice to ignore them on subsequent correlator runs. There are pros and cons to doing this. Pros include saving memory, compute time, and less clutter in the match table, although it can be filtered to remove the clutter. Cons include losing information about multiple correlator agreements

or disagreements.

### Common Problems

This section lists common problems or things that should be avoided when version tracking programs.

#### Making Program Changes

While version tracking two programs you should not make changes to either of the programs (aside from applying markup through the version tracking tool). Changing programs, especially the destination program, can cause the state of version tracking data (e.g., matches and their markup) to be incorrect. If you need to make changes to either of the programs, then we recommend restarting the version tracking process from the beginning after your changes have been made.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Introduction](#)
- [Version Tracking Tool](#)
- [Version Tracking Wizard](#)
- [Version Tracking Matches Table](#)
- [Version Tracking Markup Items Table](#)

# Version Tracking Preconditions

One of the first items you run across in the Version Tracking Wizard is the Precondition Panel. \*\*\*Put picture here\*\*\*\* It is an important initial step in the Version Tracking process. In the past, users trying to match functions and pull relevant "mark-up" such as labels and comments into a new version of a binary, would encounter problems if one or both of the binaries were not sufficiently analyzed or had major analysis problems. Users were given no indication that these issues were a direct result of having a poorly analyzed binary. The success or failure of the various preconditions are indicators of how well your binaries have been analyzed and how well their analyses "match" each other. If the preconditions indicates problems \*\*\* show and describe red x, warning, green check\*\*\*\*, it is important to fix them before moving on or there will probably be problems with the Version Tracking process, such as identifying incorrect matches or failing to find valid function matches. In general, Version Tracking will work best if the same methods of cleaning up a binary are used and if similar numbers of functions are created.

## Current Preconditions

Precondition Name	Precondition Description	Potential Problems	How to Fix
<b>Memory Blocks</b>	This validator checks to see if both program memory maps have been split into the same memory blocks with the same permissions.	Potential problems include incorrect analysis and decompilation due to incorrect execution or data permissions being set.	Use the Window→Memory Map to adjust the memory blocks and permissions so that they are correct.
<b>Number of Functions</b>	This validator checks to see if both programs have a similar number of defined functions	Potential problems include missing function matches.	Change analysis options and rerun it, run analysis scripts, run aggressive analyzer, and/or manually disassemble and create functions.
<b>Number of "No-Return" Functions</b>	This validator checks to see if both programs have a similar number of "No-Return" functions	Potential problems include bad analysis due to disassembly falling through past the end of a function.	Run the FixupNoReturnsScript.
	This validator checks to see	Potential problems include bad analysis due	Using the Symbol Table find offcut references. If they are incorrect, fix them and the problems that

<b>Offcut References</b>	if either program has offcut references.	to disassembly or data creation in incorrect locations.	caused them to be created such as incorrect operand reference assumptions or incorrect memory map flags, etc...
<b>Percent Analyzed</b>	This validator checks to see if both programs have a similar percentage of analyzed code in the code segments of each binary.	Potential problems include missing potential matches due to incomplete analysis.	Change analysis options and rerun it, run analysis scripts, run aggressive analyzer, and/or manually disassemble and create functions.
<b>Red Flags</b>	This validator checks to see if either program has red flags indicating errors in analysis.	Potential problems include incorrect instruction definitions at the language level and incorrect analysis of code or data.	Use the Bookmark Manager or Margin Markers to find red flags. Fix them and the problems that caused them to be created such as bad flow (most likely) or bad instruction definitions.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Matches Table](#)
- [Version Tracking Markup Table](#)
- [Version Tracking Introduction](#)
- [Code Browser](#)

## Version Tracking Program Correlators

A program correlator compares two versions of a program and generates matches between them. These matches are either functions (this function in the old version became this function in the new version), or data (this piece of defined data in the old is now here in the new version).

Each correlator has its own strengths and weaknesses, as well as characteristics for generating scores. For instance, some correlators may run better after already having run certain others (and processing their results). Some correlators return hard-coded scores due to the nature of their correlation. It is important to read the specifics of each correlator's description to understand how to best use it.

Below is a list of built-in (i.e., not discovered) program correlators.

- [Data Match Correlators](#)
  - [Exact Data Match](#)
  - [Duplicate Data Match](#)
  - [Similar Data Match](#)
- [Function Match Correlators](#)
  - [Exact Function Bytes Match](#)
  - [Exact Function Instructions Match](#)
  - [Duplicate Function Instructions Match](#)
  - [Exact Function Mnemonics Match](#)
- [Legacy Import Correlator](#)
- [Implied Correlator](#)
- [Manual Match Correlator](#)
- [Symbol Name Match Correlator](#)
  - [Exact Symbol Name Match](#)
  - [Duplicate Exact Symbol Name Match](#)
  - [Similar Symbol Name Match](#)
- [Reference Correlators that Use Match Information to Find Other Matches](#)
  - [Data Reference Correlator](#)
  - [Function Reference Correlator](#)
  - [Combined Function and Data Reference Correlator](#)

### Data Match Correlators

#### [Exact Data Match](#)

Exact Data Match will iterate through your entire source program's list of defined data, and look for a 1:1 correspondence for this data in the destination program. For instance, if you have a defined string "This is a string" that appears only once in the source program, and the Exact Data Matcher finds a single area in the destination program with "This is a string", it will report it as a match.

Exact Data Match reports 1.0 for similarity score (because by definition, the data are exactly the same) and 1.0 for confidence score (because it was found only once in each program).

#### [Duplicate Data Match](#)

Duplicate Data Match will iterate through your source program's defined data, look for matches in the destination program, and only create matches if the correspondence is NOT 1:1, i.e. 1:N, M:1, or M:N. Switch tables are often found by this correlator.

Duplicate Data Match reports 1.0 for similarity score (because by definition, the data is exactly the same), but the confidence scores will all be less than 1.0. Duplicate Data Match uses 10/(total source and destination matches) for raw confidence, but due to the  $\log_{10}$  scaling of the confidence reporting column these values will always be less than 0.7<sup>\*</sup>.

Note that the data does not need to be defined in the destination program for these correlators to find matches.

#### [Similar Data Match](#)

Similar Data Match will iterate through your source and destination programs' defined data, and look for similar data based on locality sensitive hashing of 4-grams.

Similar Data Match reports a similarity score based on the cosine difference between the vector representations of the two data objects. A similarity of 1.0 means the data matches exactly. The confidence scores are based on the similarity score and the vector lengths of the data objects.

### Function Match Correlators

 Note that functions MUST be defined in the destination program for these correlators to find matches.

#### [Exact Function Bytes Match](#)

Exact Function Bytes Match is very similar to the Exact Data Match correlator, except it looks for functions. If there is an exact byte-for-byte, 1:1 correspondence between a function in the source program and a function in the destination program, it will create a match.

Exact Function Bytes Match reports 1.0 for similarity score (because by definition, the functions are exactly the same) and 1.0 for confidence score (because it was found only once in each program).

#### [Exact Function Instructions Match](#)

Exact Function Instructions Match is almost exactly like Exact Function Bytes Match, except that it removes the operands from the instructions by masking them out. This has an advantage on certain CPU architectures (RISC)/compilers that can compile the same source using different registers but otherwise identical opcodes. If a 1:1 correspondence is found between a masked source function and a masked destination function, a match is created.

Exact Function Instructions Match reports 1.0 for similarity score, even though the functions aren't necessarily identical byte-for-byte. It reports 1.0 for confidence score (because it was found only once in each program).

#### [Duplicate Function Instructions Match](#)

Duplicate Function Match will iterate through your source program's functions, look for matches in the destination program, and only create matches if the correspondence is NOT 1:1, i.e. 1:N, M:1, or M:N. It uses the operand masking to eliminate operands from the opcodes when searching for matches. Boilerplate or template functions are often found by this correlator.

Duplicate Function Match reports 1.0 for similarity score, even though the functions aren't necessarily identical byte-for-byte. The confidence scores will all be less than 1.0. Duplicate Function Match uses 10/(total source and destination matches) for raw confidence, but due to the  $\log_{10}$  scaling of the confidence reporting column these values will always be less than 0.7<sup>\*</sup>.

#### [Exact Function Mnemonics Match](#)

Exact Function Mnemonic Match is almost exactly like Exact Function Instructions Match, except that it uses the mnemonic names representing an instruction instead of the bytes representing the instruction. This will find cases where the same instruction names are used but the underlying instruction bytes have changed. Differences between the two correlators will be rare. If a 1:1 correspondence is found between a masked source function and a masked destination function, a match is created.

Exact Function Instructions Match reports 1.0 for similarity score, even though the functions aren't necessarily identical byte-for-byte. It reports 1.0 for confidence score (because it was found only once in each program).

### Legacy Import Correlator

The Legacy Import correlator is a legacy results file importer (and not really a correlator). It reads text files with a very specific format, and only creates functions matches based upon the contained data.

The format of the file is a series of matches, each match on its own line. The match is comprised of 9 fields, each separated by whitespace. The 9 fields are as follows:

```
score (similarity, in the range [0.0, 1.0])
source length (in bytes)
destination length (in bytes)
source program name
source function name
source address
destination program name
destination function name
destination address
```

 NOTE: this format is provided for users who have existing results that absolutely positively can't see any way to get them into Version Tracking another way. It is legacy, deprecated, and will likely be replaced in the future with a new import format (most likely a legacy format -> new data import format conversion tool will be released simultaneously). We strongly recommend you don't use this importer, and instead start your Version Tracking analysis from scratch using our provided correlators.

## Implied Correlator

The "Implied Match" correlator is a placeholder for matches that were created based on references from other accepted matches. There is no "Implied Match" algorithm that can be run, but since all matches must have a correlator, the Implied Match correlator was created. Its only purpose is to display Implied Matches in the matches table. See the [Implied Matches Table](#) for more information.

## Manual Match Correlator

The manual match correlator is a placeholder for matches that were created manually by the user. There is no "Manual Match" algorithm that can be run, but since all matches must have a correlator, the Manual Match correlator was created. Its only purpose is to display Manual Matches in the matches table.

## Symbol Name Match Correlators

### Exact Symbol Name Match

Exact Symbol Name Match is very similar to the Exact Data Match correlator and the Exact Function Match correlators, except it looks for symbol names. If there is an exact 1:1 correspondence between a symbol name (after removing the \_address suffix that is sometimes added to symbols) in the source program and a symbol in the destination program, it will create a match. Note that this correlator only works on symbols where there is a defined function or defined data in the source program. For the function case, there also must be a defined function in the destination program. For the data case, there does not have to be defined data in the destination program. Also note that this correlator ignores the default symbols that Ghidra automatically creates such as, but not limited to, those that start with FUN\_, DAT\_, except strings that start with s\_, u\_, etc... Note: There is an option to that allows users to find external symbol matches as well.

Exact Symbol Name Match reports 1.0 for similarity score (because by definition, the symbols are exactly the same) and 1.0 for confidence score (because it was found only once in each program).

### Duplicate Exact Symbol Name Match

Duplicate Exact Symbol Name Match will iterate through your source program's symbols, look for matches in the destination program, and only create matches if the correspondence is NOT 1:1, i.e. 1:N, M:1, or M:N. When there is more than one symbol with the same name in Ghidra, Ghidra appends the address onto the end of it. This correlator strips off the ending address before correlating symbol names and ignores the default symbols that Ghidra automatically creates such as, but not limited to, those that start with FUN\_, DAT\_, except strings that start with s\_, u\_, etc... Note: There is an option to that allows users to find external symbol matches as well.

Duplicate Exact Symbol Name Match reports 1.0 for similarity score, since the symbols (minus the tacked on address) are identical. The confidence scores will all be less than 1.0. Duplicate Exact Symbol Name Match uses  $10/\log_{10}$  (total source and destination matches) for raw confidence, but due to the  $\log_{10}$  scaling of the confidence reporting column these values will always be less than 0.7\*.

\*  $\log_{10}(10/N) < 0.7$  for  $N > 1$ ; e.g.  $\log_{10}(5) \approx 0.69897$

### Similar Symbol Name Match

Similar Symbol Name Match will iterate through your source and destination programs' symbols, and look for similar names based on locality sensitive hashing of trigrams.

Similar Symbol Name Match reports a similarity score based on the cosine difference between the vector representations of the two symbol names. A similarity of 1.0 means the symbols match exactly. The confidence scores are based on the similarity score and the vector lengths of the names.

## Reference Correlators that Use Match Information to Find Other Matches

Each of the following program correlators determines correlation based on feature vectors constructed from matched and unmatched references of their respective types. The algorithm used is

1. Identify functions that reference each ACCEPTED (green) match of the correct type.
2. Construct a sourceMap and a destinationMap of the [referenceFunction:featureVector] where the featureVector identifies an ACCEPTED match with the log weight for the probability that it appears in any one function in the system. NOTE: The same feature and log weight are added to sourceMap and destinationMap for each match.
3. For each referenceFunction add a unique feature\* to its featureVector with probability 0.5 for each of its UNMATCHED references of the correct type.
4. Score each pair of SOURCE and DESTINATION functions by the angle between their feature vectors, taking the highest scoring pairs as the result.
5. Refine the results by removing matches that have no clear winner.

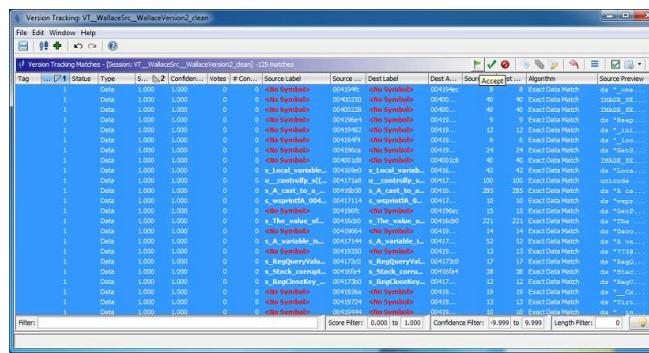
\* This is to account for the probabilistic cost of a reference being switched, dropped or picked up between SOURCE and DESTINATION versions. Theoretically this should be dependent on the probability of the referenced element occurring, but for simplicity we consider the model for a generalized switch and drop/pickup by assigning a probability of 0.5 to each of the unmatched references made in any of our considered functions.

## Data Reference Correlator

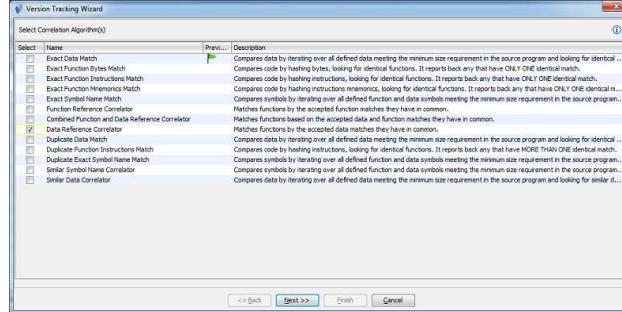
The Data Reference Correlator uses ACCEPTED matched data to find other function matches based on common data they reference. That is, a reference is considered if it is a reference to matched data location.

### Example

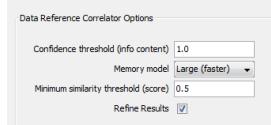
1. Start a Version Tracking session
2. Run the "Exact Data Match" correlator
3. Accept all the matches



4. Run the "Data Reference Correlator"



5. Try using the default options (click 'Next' and 'Finish').



6. Now try sorting on Score for the Data Reference Correlator and note that the lowest score is above the minimum threshold set in the options. If a function is missing that you expect to see in the results, try lowering the thresholds and/or unchecking the "Refine Results" checkbox.

Version Tracking Matches - [Session: VT_WallaceSrc_WallaceVersion2_clean] -138 matches														
Tag	Sess...	Status	Type	S...	# Confid...	Votes	# Conflic...	Source Label	Source ...	Dest Label	Dest A...	Source ...	Dest ...	Algorithm
2		Function	0.553	1.711	0	0	0	_RTC_GetSrcLine	00413520	_RTC_GetSrcL...	00412...	696	696	Data Reference Cor...
2		Function	0.712	2.188	0	0	0	_RTC_SetStackalu...	00412810	_RTC_SetStacka...	00412770	252	252	Data Reference Cor...
2		Function	0.712	2.188	0	0	0	_RTC_Ummituse	00412800	_RTC_Ummitu...	00412...	233	233	Data Reference Cor...
2		Function	0.774	2.489	0	0	0	GetPubDoll	00413800	GetPubDoll	00412...	491	491	Data Reference Cor...
2		Function	0.774	2.489	0	0	0	print	00415600	print	00411560	181	181	Data Reference Cor...
2		Function	0.822	1.982	0	0	0	deployGadget	00411800	00411800	00411800	251	251	Data Reference Cor...
2		Function	0.881	3.012	0	0	0	_RTC_Failure	00412380	_RTC_Failure	00412...	81	81	Data Reference Cor...
2		Function	0.881	2.410	0	0	0	failwithmessage	00412390	failwithmessag...	00412...	529	529	Data Reference Cor...
2		Function	0.937	2.711	0	0	0	_RTC_AllocaFailu...	00412950	_RTC_AllocaFa...	00412...	307	307	Data Reference Cor...
2		Function	1.000	2.479	0	0	0	addPeople	00411700	addPeople	00411690	152	152	Data Reference Cor...
2		Function	1.000	1.711	0	0	0	_getMemBlockd...	00412260	_getMemBlock...	00412...	111	111	Data Reference Cor...
2		Function	1.000	1.711	0	0	0	_RTC_GetErrRe...	00412260	_RTC_GetErrR...	00412...	26	26	Data Reference Cor...
2		Function	1.000	2.188	0	0	0	_setdefaultfaul...	00412690	_setDefaultfaul...	00412...	55	55	Data Reference Cor...
1		Data	1.000	1.000	0	0	0	<No Symbol>	004194fc	<No Symbol>	004194ec	8	8	Exact Data Match
1		Data	1.000	1.000	0	0	0	<No Symbol>	00400250	<No Symbol>	00400...	40	40	Exact Data Match
1		Data	1.000	1.000	0	0	0	<No Symbol>	00400228	<No Symbol>	00400...	40	40	Exact Data Match
1		Data	1.000	1.000	0	0	0	<No Symbol>	004196e4	<No Symbol>	00419...	9	9	Exact Data Match
1		Data	1.000	1.000	0	0	0	<No Symbol>	00419482	<No Symbol>	00419...	12	12	Exact Data Match
1		Data	1.000	1.000	0	0	0	<No Symbol>	004194f4	<No Symbol>	00419...	6	6	Exact Data Match
1		Data	1.000	1.000	0	0	0	<No Symbol>	004196ca	<No Symbol>	00419...	24	24	Exact Data Match

#### Function Reference Correlator

The Function Reference Correlator uses ACCEPTED matched functions to find other function matches based on common functions they reference. That is, a reference is considered if it is a reference to a matched function.

##### Example

1. By first running the "Exact Function Instructions Match" correlator, and then running the "Function Reference Correlator" with the default options as above, we see potential matches with Scores  $\geq 0.5$ .

Version Tracking Matches - [Session: VT_WallaceSrc_WallaceVersion2_clean] -245 matches														
Tag	Sess...	Status	Type	Score	# Confid...	Votes	# Conflic...	Source Label	Source Add...	Dest Label	Dest Address	Source Le...	Dest Le...	Algorithm
2		Function	0.514	1.711	0	0	0	_RTC_GetSrcLine	00413520	_RTC_GetSrcL...	00412...	696	696	Function Reference C...
2		Function	0.536	1.525	0	2	2	FUN_004134e0	004134e0	GetPubDoll	00413870	41	491	Function Reference C...
2		Function	0.669	2.188	1	0	0	__mainCRTStartup	00411f00	__mainCRTStart...	00411ee0	523	523	Function Reference C...
2		Function	0.708	1.925	0	0	0	addPeople	00411700	addPeople	004116f0	152	152	Function Reference C...
2		Function	0.779	2.088	0	0	0	_RTC_AllocaFailure	00412950	_RTC_AllocaFailure	00412930	307	307	Function Reference C...
2		Function	0.788	1.711	0	0	0	FUN_00411e70	00411e70	FUN_00411e50	00411e50	90	90	Function Reference C...
2		Function	0.795	2.274	4	0	0	failwithmessage	00412390	failwithmessag...	00412380	529	529	Function Reference C...
2		Function	0.802	1.925	0	0	0	main	00411a30	main	00411a10	95	95	Function Reference C...
2		Function	0.817	2.188	0	0	0	FUN_00411dc0	00411dc0	FUN_00411da0	00411da0	138	138	Function Reference C...
2		Function	0.844	1.575	0	0	0	_RTC_Failure	00412380	_RTC_Failure	00412360	81	81	Function Reference C...
2		Function	0.881	1.711	0	0	0	_setDefaultprecis...	00412e90	_setDefaultprecis...	00412e70	55	55	Function Reference C...
2		Function	0.907	2.132	0	0	0	deployGadget	004118f0	FUN_004118e0	004118e0	250	261	Function Reference C...
2		Function	0.924	1.925	0	0	0	addPerson	00411860	addPerson	00411830	115	115	Function Reference C...
2		Function	0.933	1.982	0	0	0	__RTC_CheckStack...	00411c50	__RTC_CheckStack...	00411c50	212	212	Function Reference C...
2		Function	1.000	2.012	2	0	0	__IsNonwritable...	00413350	__IsNonwritable...	00413350	210	210	Function Reference C...
2		Function	1.000	2.012	0	0	0	__mainCRTStartup	00411e60	__mainCRTStartup	00411e60	17	17	Function Reference C...
2		Function	1.000	1.548	0	0	0	__RTC_CheckStack...	00411b80	__RTC_CheckStack...	00411b80	95	95	Function Reference C...
2		Function	1.000	1.711	0	0	0	RTC_CheckStack	00411b80	RTC_CheckStack	00411b80	37	37	Function Reference C...
2		Function	1.000	1.635	0	1	1	FUN_004134e0	004134e0	print	00411c40	41	41	Function Reference C...
2		Function	1.000	1.711	0	0	0	__exit	004130b0	__exit	004130b0	78	28	Function Reference C...
1		Function	1.000	1.000	0	0	0	FUN_00412200	00412200	FUN_004121e0	004121e0	13	13	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	FID_conflict__RTC...	00412f00	FID_conflict__RTC...	00412f00	38	38	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	__RTC_GetErrDesc	00412280	__RTC_GetErrDesc	00412280	26	26	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	addPerson	00411860	addPerson	00411830	115	115	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	addPeople	00411700	addPeople	004116f0	152	152	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	print	00411560	print	004115c0	181	181	Exact Function Instruc...
1		Function	1.000	1.000	4	0	0	failwithmessage	00412390	failwithmessage	00412380	529	529	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	ValidateImageB...	00413250	ValidateImageB...	00413230	95	95	Exact Function Instruc...
1		Function	1.000	1.000	0	0	0	security_init_c...	00413110	security_init_c...	004130f0	224	224	Exact Function Instruc...

2. By lowering the "Minimum similarity threshold (score)", we see one additional possible match. The lower score indicates to us that this function, "`__onexit`", makes reference to functions that have not been matched yet.

Function Reference Correlator Options														
Confidence threshold (Info content)	1.0	Memory model	Large (faster)	Minimum similarity threshold (score)	0.2	Refine Results	<input checked="" type="checkbox"/>							

Version Tracking Matches - Session: VT_WallaceSrc_WallaceVersion2_clean - 266 matches																
Tag	Sess...	#1	Status	Type	Score	L2	Confidence (...	Votes	# Conflicting	Source Label	Source Addr...	Dest Label	Dest Address	Source Le...	Dest Le...	Algorithm
3	1	Function	0.452	1.711	0	0	0	0	0	__onexit	00412f60	__onexit	00412b80	232	232	Function Reference C...
3	1	Function	0.514	1.711	0	0	0	0	0	__RTC_GetSrcLine	00413520	__RTC_GetSrcLine	00413500	696	696	Function Reference C...
3	1	Function	0.536	1.625	0	2	0	2	0	FUN_004134e0	004134e0	GetPubDfl	00413870	41	491	Function Reference C...
3	1	Function	0.669	2.188	1	0	0	0	0	__mainCRTStartup	00411900	__mainCRTStart...	00411680	523	523	Function Reference C...
3	1	Function	0.670	1.925	0	0	0	0	0	addPeople	00411680	addPeople	00411680	152	152	Function Reference C...
3	1	Function	0.779	2.088	0	0	0	0	0	__RTC_AllocFailure	00411270	__RTC_AllocFailure	00411230	307	307	Function Reference C...
3	1	Function	0.788	1.711	0	0	0	0	0	FUN_00411e70	00411a70	FUN_00411e50	00411a50	90	90	Function Reference C...
3	1	Function	0.795	2.274	4	0	0	0	0	failwithmessage	00412280	failwithmessage	00412240	529	529	Function Reference C...
3	1	Function	0.802	1.925	0	0	0	0	0	main	00411a30	main	00411a30	529	529	Function Reference C...
3	1	Function	0.817	2.188	0	0	0	0	0	FUN_00411dc0	00411dc0	FUN_00411da0	00411da0	138	138	Function Reference C...
3	1	Function	0.844	1.575	0	0	0	0	0	__RTC_Failure	00412380	__RTC_Failure	00412380	81	81	Function Reference C...
3	1	Function	0.881	1.711	0	0	0	0	0	__setDefaultPrintRes...	00412e90	__setDefaultPrintRes...	00412e70	55	55	Function Reference C...
3	1	Function	0.907	2.132	0	0	0	0	0	deployGadget	004118f0	deployGadget	004118c0	250	261	Function Reference C...
3	1	Function	0.924	1.925	0	0	0	0	0	addPerson	004118e0	addPerson	00411830	115	115	Function Reference C...
3	1	Function	0.933	1.982	0	0	0	0	0	@_RTC_CheckStack...	00411c50	@_RTC_CheckStack...	00411c50	212	212	Function Reference C...
3	1	Function	1.000	2.012	0	0	0	0	0	__mainCRTStartup	00411ee0	__mainCRTStartup	00411ec0	17	17	Function Reference C...
3	1	Function	1.000	2.012	0	0	0	0	0	_IsNonwritableInC...	00413370	_IsNonwritableInC...	00413350	210	210	Function Reference C...
3	1	Function	1.000	1.711	0	0	0	0	0	_atexit	00413060	_atexit	00413080	28	28	Function Reference C...
3	1	Function	1.000	1.648	0	0	0	0	0	@_RTC_CheckStack...	0041bb50	@_RTC_CheckStack...	0041bb90	93	93	Function Reference C...
3	1	Function	1.000	1.525	0	1	0	0	1	FUN_004134e0	004134e0	FUN_004134c0	004134c0	41	41	Function Reference C...
3	1	Function	1.000	1.711	0	0	0	0	0	__RTC_CheckStack...	00411b60	__RTC_CheckStack...	00411b60	37	37	Function Reference C...

3. By using the same thresholds as above, but also unchecking the "Refine Results" box, we get many more results returned from the Reference Correlator. In this case, most of the additional results are BLOCKED by our previously ACCEPTED matches, which, for example, results in only the correct pair for "print" being made available for matching.

Function Reference Correlator Options

Confidence threshold (info content)	1.0
Memory model	Large (faster) ▾
Minimum similarity threshold (score)	0.2
Refine Results	<input type="checkbox"/>

Version Tracking Matches - Session: VT_WallaceSrc_WallaceVersion2_clean - 363 matches																
Tag	Sess...	#1	Status	Type	Score	L2	Confidence (...	Votes	# Conflicting	Source Label	Source Addr...	Dest Label	Dest Address	Source Le...	Dest Le...	Algorithm
4	1	Function	0.440	1.515	0	11	0	0	0	initializePeople	00411700	addPeople	00411680	120	152	Function Reference C...
4	1	Function	0.514	1.525	0	2	0	0	2	FUN_004134e0	004134e0	Call_strcpy_s	00411a70	120	120	Function Reference C...
4	1	Function	0.702	1.515	0	0	0	0	0	initializePeople	00411700	print	00411520	150	181	Function Reference C...
4	1	Function	0.702	1.515	0	0	0	0	0	initializePeople	00411700	main	00411a10	120	95	Function Reference C...
4	1	Function	0.468	1.515	0	11	0	0	11	initializePeople	00411700	FUN_004118c0	00411a80	120	261	Function Reference C...
4	1	Function	0.392	1.515	0	0	0	0	0	initializePeople	00411700	addPerson	00411a30	120	115	Function Reference C...
4	1	Function	0.502	1.515	0	11	0	0	11	initializePeople	00411700	addPerson	00411a30	95	95	Function Reference C...
4	1	Function	0.261	1.515	0	11	0	0	11	main	00411a30	FUN_004118c0	00411a80	95	261	Function Reference C...
4	1	Function	0.468	1.515	0	0	0	0	0	main	00411a30	print	00411a50	95	181	Function Reference C...
4	1	Function	0.293	1.515	0	0	0	0	0	main	00411a30	addPeople	00411a60	95	152	Function Reference C...
4	1	Function	0.468	1.515	0	0	0	0	0	main	00411a30	Call_strcpy_s	00411a90	95	82	Function Reference C...
4	1	Function	0.335	1.515	0	0	0	0	0	main	00411a30	addPerson	00411a80	95	115	Function Reference C...
4	1	Function	0.702	1.515	0	11	0	0	11	print	00411520	print	004115c0	181	181	Function Reference C...
4	1	Function	0.502	1.515	0	0	0	0	0	print	00411520	Call_strcpy_s	00411a90	181	82	Function Reference C...
4	1	Function	0.440	1.515	0	0	0	0	0	print	00411520	addPeople	00411a60	181	152	Function Reference C...
4	1	Function	0.502	1.515	0	0	0	0	0	print	00411520	addPerson	00411a80	181	115	Function Reference C...
4	1	Function	0.468	1.515	0	11	0	0	11	print	00411520	main	00411a10	181	95	Function Reference C...
4	1	Function	0.392	1.515	0	0	0	0	0	print	00411520	FUN_004118c0	00411a80	181	261	Function Reference C...
3	1	Function	0.933	1.982	0	2	0	0	2	@_RTC_CheckStack...	00411700	@_RTC_CheckStack...	00411c50	212	212	Function Reference C...
2	1	Function	1.000	1.648	0	0	0	0	0	@_RTC_CheckStack...	00411700	@_RTC_CheckStack...	00411a80	93	93	Function Reference C...

NOTE: By default the Version Tracking table does not display "Blocked" matches. To see them, go to the Match Table Filters (☒ in the upper right corner) and check the box next to "Blocked" under "Association Status".

Association Status

<input checked="" type="checkbox"/> Accepted
<input checked="" type="checkbox"/> Available
<input checked="" type="checkbox"/> Blocked
<input checked="" type="checkbox"/> Rejected

#### Combined Function and Data Reference Correlator

The Combined Function and Data Reference Correlator matches functions based on the accepted data and function matches they have in common. This means that the set of references considered for each tested function includes all its data and function references. That is, a reference is considered if it is a reference to a matched function or matched data location.

NOTE: If no matches are returned, make sure there are existing ACCEPTED matches (☒). This means you will need to run other correlators first, such as

- Exact Data Match
- Exact Function Bytes Match

Provided by: Version Tracking Plugin

Related Topics:

- Version Tracking Matches Table
- Version Tracking Tool
- Version Tracking Introduction

# Version Tracking Wizard

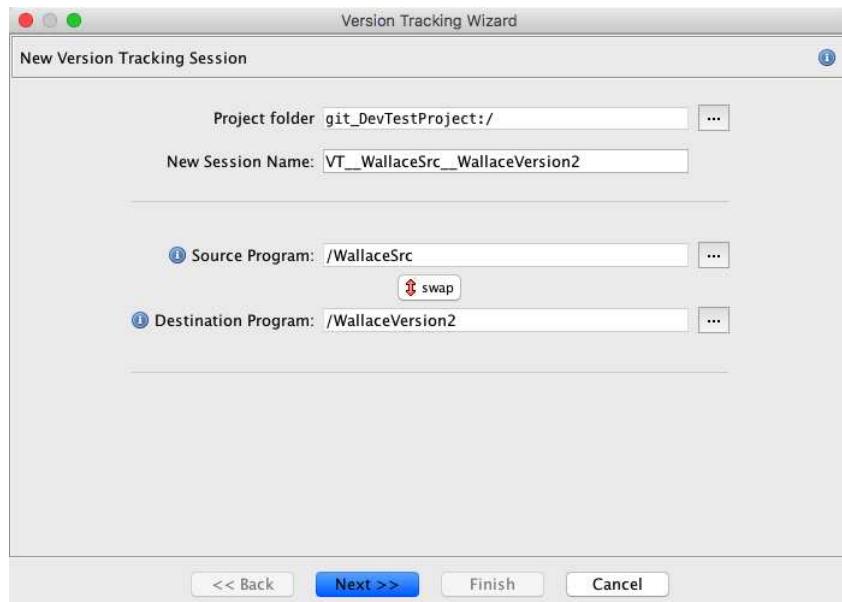
The version tracking wizard guides you through the process of creating a new version tracking session or adding results to an existing session.

## Creating a New Session

To create a new version tracking session you can:

- Drag the two programs to be tracked onto a running tool or the Version Tracking tool button ([more info](#))
- Press the **Create Session**  action from within the [Version Tracking Tool](#)

### New Version Tracking Session Panel

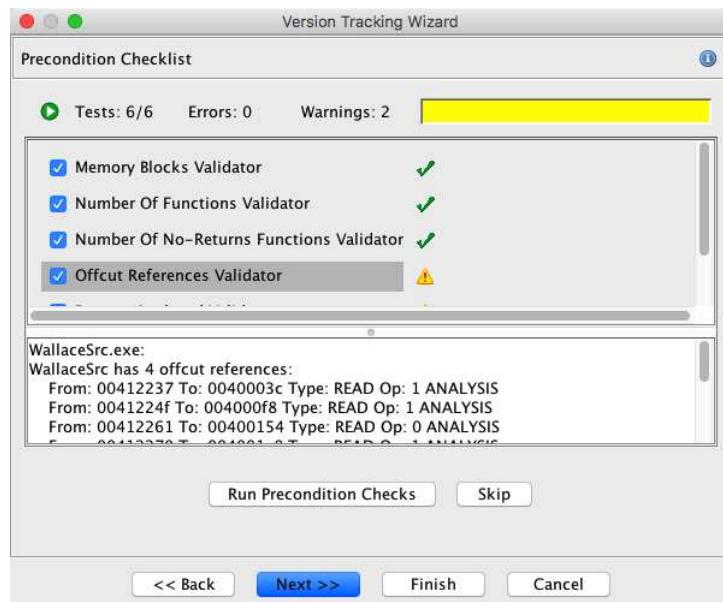


The new version tracking session panel appears when creating a new session only. On this page you specify the domain folder in your project in which to store the session information, the name of the session domain object, and the source and destination programs. The source program is the existing program that's been analyzed and contains markup to transfer. The destination program is the new program that will receive the markup items.

When dragging two programs from the front-end onto the version tracking tool, the source and destination might not be properly identified. If they are backwards (source is destination, destination is source) simply press the "swap" button  to correct the ordering.

### Preconditions Panel

---



The preconditions panel has a list of mini-analysis routines called "validators". These validators analyze parts of your source and destination programs, looking for potential problems which will adversely affect version tracking success. For instance, a large difference in the number of defined functions between the source and the destination programs could be an indication that they are not ready to be correlated.

Press the button "Run Precondition checks", and then review the results in the panel by clicking on the individual tests in the list.

[Click here](#) for a list of known preconditions.

#### Summary Panel



The summary panel shows a summary of the selections provided to the Version Tracking wizard before it creates the new version tracking session. Selecting the "Finish" button creates the new version tracking session.

#### Add to an Existing Session

To add to an existing session you can:

- Press the **Add to Session**  action from within the [Version Tracking Tool](#)

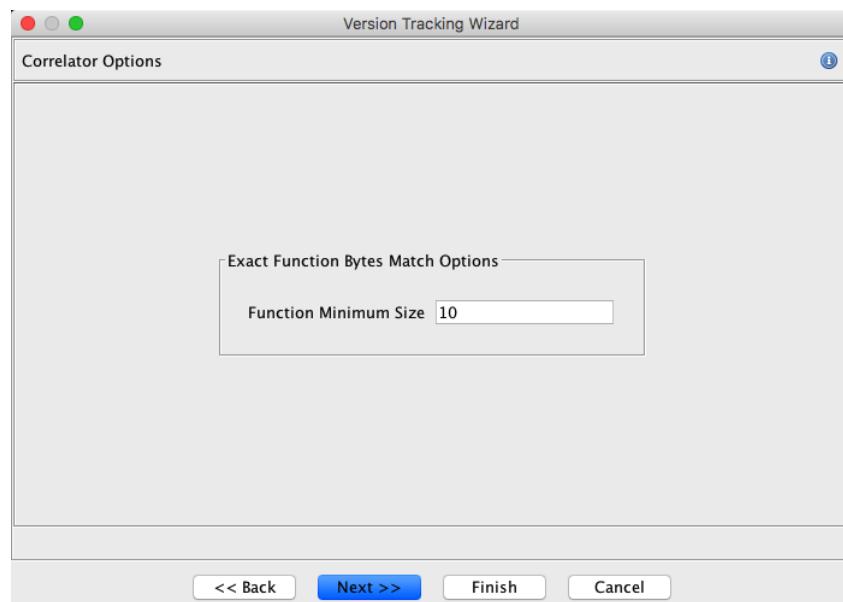
The wizard panels that appear can vary depending on which options are selected on prior panels. For example, the correlator options will depend upon which correlation algorithm was chosen. Also the Select Address Ranges panel is only displayed if you choose to Limit Addresses on the Address Set Options panel. The following wizard panels are for adding correlation results to an existing session.

#### Correlation Algorithm Panel



The correlation algorithm panel lets you choose which program correlator to use. The list is dynamically populated based upon which features you have installed, so the actual list may look different than that above. See the [Correlators](#) help page for more information about individual correlators or the [Workflow](#) help page for information about which correlators to run first.

#### Options Panel



The options panel displays correlation algorithm specific options to select. Please see notes on the individual correlator algorithms for more information about their options.

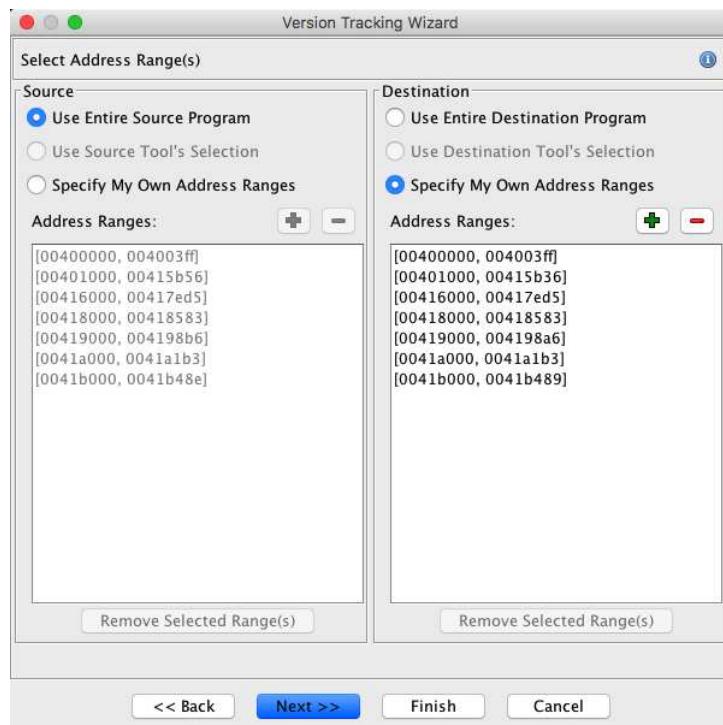
#### Address Set Options Panel



The address set options panel lets the user add or remove specific address ranges from consideration by the chosen program correlator.

- This **Exclude accepted matches** option will cause the correlator algorithm to not consider any functions or data that have already been accepted. Using this option can greatly speed up the processing time of the correlator algorithm; however, this options should only be used when you trust that your accepted matches are correct. As an example, if you accepted matches from the **Exact Function Bytes** correlator algorithm, you can be very confident that your matches are correct because they are unique matches with all bytes in each function are identical to each other. These matches would be ok to exclude in the next correlator run.
- NOTE:** This option will be disabled when the correlator does not support limiting search scope.
- The **Limit source and destination address sets** option allows the users to choose ranges of their program to limit the current correlator algorithm to when matching functions and data. If you need to restrict the address ranges of your programs, you should select the checkbox here. For instance, this might be useful if you have two copies of your functions in memory and only want to consider one copy.  
If you select this option you will be presented with another panel for limiting the source program's address ranges and the destination program's address ranges.

#### Select Address Range(s) Panel



The select address ranges panel lets the user limit the source and destination address sets that will be used by the chosen program correlator when determining version tracking matches. The left side lets you specify what addresses to use from the Source program and the right side allows you to specify the address ranges to use from the Destination program.

#### Source

The left side of this panel allows you to specify the address ranges to use from the Source program when adding version tracking matches to a session. Select the appropriate radio button to indicate the addresses that you want the correlator to use.

- The **Use Entire Source Program** option will use all memory addresses currently defined in the source program when determining match results.



**NOTE:** This is the default option which you normally get if you don't check the **Limit source and destination address sets** box on the previous panel for Address Set Options.

- The **Use Source Tool Selection** option will use all memory addresses that were selected in the Source Tool's listing when you invoked the Add To Session version tracking wizard. If necessary, you can Cancel the current wizard, select addresses in the version tracking Source Tool, and restart the wizard if you want to limit the addresses using a selection.



**NOTE:** This option will be disabled if there wasn't a selection in the version tracking Source Tool when the Add To Session version tracking wizard was started.

- The **Specify My Own Address Ranges** option allows you to manually specify address ranges of the Source program to include when determining matches with the current correlator algorithm. All other addresses will be ignored. If you choose this option, you can specify the ranges of addresses, which then appear in the **Address Ranges** list. The list and its associated buttons are disabled whenever the option is not the currently selected radio button. When you have a Source Tool selection, the list will initially contain the same address ranges as the selection. Otherwise, it will contain the address ranges for the current memory in the source program.



Press the **Add Range** button to add another address range to those already in the list.



Press the **Remove Range** button to remove all addresses from the list that fall within a specific range.



Pressing the **Remove Selected Range(s)** button will remove any address ranges from the Address Ranges list that are currently selected.



**NOTE:** This button will be disabled if the **Specify My Own Address Ranges** option isn't selected or if there isn't an address range selected in the list.

### Destination

The right side of this panel allows you to specify the address ranges to use from the Destination program when adding version tracking matches to a session. Select the appropriate radio button to indicate the addresses that you want the correlator to use from the destination program.

The options within the Destination function in the same manner as they did for the Source, but instead apply to addresses within the destination program.

- The **Use Entire Destination Program** option will use all memory addresses currently defined in the destination program when determining match results.



**NOTE:** This is the default option which you normally get if you don't check the **Limit source and destination address sets** box on the previous panel for Address Set Options.

- The **Use Destination Tool Selection** option will use all memory addresses that were selected in the Destination Tool's listing when you invoked the Add To Session version tracking wizard. If necessary, you can Cancel the current wizard, select addresses in the version tracking Destination Tool, and restart the wizard if you want to limit the addresses using a selection.



**NOTE:** This option will be disabled if there wasn't a selection in the version tracking Destination Tool when the Add To Session version tracking wizard was started.

- The **Specify My Own Address Ranges** option allows you to manually specify address ranges of the Destination program to include when determining matches with the current correlator algorithm. All other addresses will be ignored.



If you choose this option, you can specify the ranges of addresses, which then appear in the **Address Ranges** list. The list and its associated buttons are disabled whenever the option is not the currently selected radio button. When you have a Destination Tool selection, the list will initially contain the same address ranges as the selection. Otherwise, it will contain the address ranges for the current memory in the destination program.



Press the **Add Range** button to add another address range to those already in the list.



Press the **Remove Range** button to remove all addresses from the list that fall within a specific range.



Pressing the **Remove Selected Range(s)** button will remove any address ranges from the Address Ranges list that are currently selected.



**NOTE:** This button will be disabled if the **Specify My Own Address Ranges** option isn't selected or if there isn't an address range selected in the list.

### Summary Panel

---



The summary panel shows a summary of the selections provided to the Version Tracking wizard before it runs the correlation. Selecting the **Finish** button will run the correlation and add its results to the current version tracking session.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Introduction](#)
- [Version Tracking Tool](#)

## Version Tracking Tool

The primary Version Tracking Tool, by default, consists of a few actions and a couple provider views. The primary view of the tool is the [MatchesTable](#). The other default component view is the [Markup Items Table](#).

The Version Tracking Tool also has sub-tools that are present when a Version Tracking Session is open.

### Version Tracking Session

A version tracking session is created when you run the [Version Tracking Wizard](#). Once created, the session will be saved to the [Ghidra Project Window](#). As you make changes to version tracking data (i.e., matches and markup items), those changes are applied to the current session.

You can open an existing session by dragging it from the Ghidra Project Window's data tree onto:

- A running Version Tracking Tool,
- The Version Tracking Tool icon in the Ghidra Project Window's [Tool Chest](#), or
- The icon of a running Version Tracking Tool in the [Running Tools](#) panel of the Ghidra Project Window.

You can also double-click the session file in the Ghidra Project Window. This will launch the session in a [new Version Tracking Tool instance](#).

When you open a session, the Version Tracking Sub-tools (mentioned below) are also opened. When you close a session, the sub-tools are closed.

### Version Tracking Sub-tools

When a session is open in the primary Version Tracking Tool, then two other tool windows will be opened: the source tool and the destination tool. Both tools look the same. They differ in which program they show, the source program or the destination program.

Each of these tools is similar to default Ghidra [Code Browser](#) in that they each provide full Ghidra functionality. However, these tools differ from the default Ghidra Tool in that they offer a few extra plugins, which add version tracking functionality.

### Version Tracking Tool Actions

The [CreateSession](#) action will launch the [Version Tracking Wizard](#) to guide you through the process of creating a new [version tracking session](#).

The [Add to Session](#) action will launch the [Version Tracking Wizard](#) to guide you through the process of adding new [Program Correlator](#) results to an existing [version tracking session](#).

The [Automatic Version Tracking](#) action uses various correlators in a predetermined order to automatically create matches, accept the most likely matches, and apply markup all with one button press. The following correlators are run in this order:

- [Exact Symbol Name Correlator](#)
- [Exact Data Correlator](#)
- [Exact Function Bytes Correlator](#)
- [Exact Function Instructions Correlator](#)
- [Exact Function Mnemonics Correlator](#)
- [Duplicate Function Instructions Correlator](#)
- [Combined Function and Data Reference Correlator](#)

**NOTE:** It is unlikely that all matches in the entire program will be made and there is no guarantee that no mistakes will be made. This action was designed to try to save as much time as possible while also taking a conservative approach to the automation.

The **OpenSession...** action will launch a session chooser dialog that allows you to pick a previously created session. This action is available only from the **File** menu.

### Version Tracking Menu

This section describes the various menu actions available from the Version Tracking Tool.

The **File** menu

- **Add to Session** –Shows the [Version Tracking Wizard](#) so that you can perform [program correlation](#) and have the results **added to the currently open version tracking session**.
- **NewSession** –Shows the [Version Tracking Wizard](#) so that you can create a new version tracking session.
- **Auto Version Track** –Runs various correlators in a predetermined order and automatically creates matches, accepts the most likely matches, and applies markup all with one button press.
- **OpenSession** –Shows a chooser dialog that allows you to open an **existing version tracking session**.
- **Close Session** –Closes the **currently open version tracking session**.
- **Save Session** –Saves **any changes to the current version tracking session**.
- **Save Tool** –Saves the state Version Tracking Tool (e.g., window locations, size and open state).
- **CloseTool** –Closes the Version Tracking Tool and the current version tracking session, if one is open.
- **Exit Ghidra** –Exits the Ghidra application.

The **Edit** menu

- **Tool Options...** –Shows the Options Dialog for the Version Tracking Tool.
- **Undo** –Performs an undo of the last edit (e.g., accepting a match, applying markup, etc.).
- **Redo** –Performs a redo of the previous undo action.
- **Reset Source and Destination Tools** –Will reset the sub-tools to be the default configurations. This is useful if you have made changes (layout, size, etc.) to the tools and would like to undo those changes.

The **Window** menu –Contains menu actions to show the various view components that are available in the Version Tracking tool. For help with a specific view component, press **F1** on the view component itself or the menu action for that component.

### Version Tracking Menu

This section describes the version tracking actions that are available from the [sub tools](#).

#### Create Manual Match

The **Create Manual Match** action () allows the user to create a match for the selected function in the source sub tool to the selected function in the destination sub tool. The action will only appear in the popup menu if your cursor is in a function in both tools.

#### Create And Accept Manual Match

The **Create And Accept Manual Match** action () allows the user to create a match for the selected function in the source sub tool to the selected function in the destination sub tool. It then accepts the match if possible. The action will only appear in the popup menu if your cursor is in a function in both tools.

#### Create And Apply Manual Match

The **Create And Apply Manual Match** action () allows the user to create a match for the selected function in the source sub tool to the selected function in the destination sub tool. It then applies the match if possible. The action will only appear in the popup menu if your cursor is in a function in both tools.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Matches Table](#)
- [Version Tracking Markup Table](#)
- [Version Tracking Introduction](#)
- [Code Browser](#)

## Version Tracking Matches Table

The Version Tracking Matches Table is the primary window for managing a version tracking session. It displays a list of all matches contained in the current session. All other version tracking windows are driven by selecting a match within this table. This window is also the primary means for accepting matches and bulk applying markup items. In addition, this table provides an extensive filtering system.

Version Tracking Matches - [Session: Untitled] - 47 matches															
Tag	S...	Type	Score	Confid...	V...	#...	...	Source Label	Sour...	M...	Dest Label	Dest Ad...	So...	D...	Algorithm
1	Function	1.000	1.000	0	0			main	00411a30	FUN_00411a10	00411a10	95	95	Exact Function Instr...	
1	Function	1.000	1.000	0	0				00411ab0	FUN_00411a90	00411a90	82	82	Exact Function Instr...	
1	Function	1.000	1.000	0	0			__RTC_CheckEsp	00411b80	__RTC_CheckEsp	00411b60	37	37	Exact Function Instr...	
1	Function	1.000	1.000	0	0			@_RTC_CheckStack...	00411bb0	@_RTC_CheckStack...	00411b90	93	93	Exact Function Instr...	
1	Function	1.000	1.000	0	0			@_RTC_AllocHelper...	00411c30	@_RTC_AllocHelper...	00411c10	50	50	Exact Function Instr...	
1	Function	1.000	1.000	0	0			@_RTC_CheckStack...	00411c70	@_RTC_CheckStack...	00411c50	212	212	Exact Function Instr...	
1	Function	1.000	1.000	0	0			FUN_00411da0	00411da0	FUN_00411d80	00411d80	290	290	Exact Function Instr...	
1	Function	1.000	1.000	0	0			FUN_00411dc0	00411dc0	FUN_00411da0	00411da0	138	138	Exact Function Instr...	
1	Function	1.000	1.000	0	0			FUN_00411e70	00411e70	FUN_00411e50	00411e50	90	90	Exact Function Instr...	
1	Function	1.000	1.000	0	0			_mainCRTStartup	00411ee0	_mainCRTStartup	00411ee0	17	17	Exact Function Instr...	
1	Function	1.000	1.000	0	0			__tmainCRTStartup	00411f00	__tmainCRTStartup	00411ee0	523	523	Exact Function Instr...	
1	Function	1.000	1.000	0	0			FUN_00412200	00412200	FUN_004121e0	004121e0	13	13	Exact Function Instr...	

### Version Tracking Match

A match represents an opinion that a function or data in one program is the equivalent function or data in another program. The pairing of a function or data from one program to another is called an association. There can be multiple matches (opinions) for the same association by one or more correlation algorithms. When a match is considered correct, it should be marked as accepted. When a match is marked as accepted, it is really the association that is accepted and therefore all matches that have the same association are considered accepted. Also, when a match is accepted, all competing matches (matches that have the same source or destination address, but not both) become blocked. For example, if one match has the opinion that A in one program is associated with X in another program and another match has the opinion that A is associated with Y, then accepting the first match, will block the second match since A can't be associated with both X and Y.

### Match Status

Each match has a primary status that is one of **Available**, **Accepted**, **Rejected**, or **Blocked**. In addition, matches that are **Accepted** have additional status for its markup items. The status column uses overlaid icons to provide information about both types of status. The table below lists the various combined status of a match.

Status	Icon	Description
AVAILABLE		The match is available to be accepted and applied.
REJECTED	🚫	The match has been rejected by the user.
BLOCKED	🔒	The match can't be accepted because a conflicting match has been accepted.  Note: To see which associations are causing a blocked association, sort the table by source or destination address to see all conflicting associations.
ACCEPTED	:green flag:	The match has been accepted.
ACCEPTED – Not Done	:green flag: 🚨	The match has been accepted. There is at least 1 markup item that has not been examined.
ACCEPTED – Fully considered	:green flag: 🟧	The match is accepted and all markup items have been applied or ignored.
ACCEPTED – Fully Applied	:green flag: 🟩	The match is accepted and all markup items have been applied.

### Match Table Columns

Column Name	Description
Session ID	A one-up number for the correlation algorithm run this match belongs to.
Tag	User-defined text that has been applied to a given match. This can be set from the <a href="#">Choose Tag</a> action.
Status	The match status. See the section above on <a href="#">Match Status</a>
Type	Type of match. Either function or data
Source Label	The label at the source address of this match.
Dest Label	The label at the destination address of this match.
Multiple Source Labels?	Icon indicating there is more than one label at the match's source address and a number indicating how many labels. The tooltip can be viewed to see the label names.
Multiple Dest Labels?	Icon indicating there is more than one label at the match's destination address and a number indicating how many labels. The tooltip can be viewed to see the label names.
Score	The primary similarity score for this match. The value will be between 0.0 and 1.0. This score indicates how similar two match items are, not necessarily that they are THE correct match. Scores should NOT be compared between different correlator algorithms.
Confidence Score	A score where higher numbers indicate more confidence that the two items are a match. These numbers have no intrinsic meaning other than higher numbers are better for the same correlator algorithm. Confidence scores should NOT be compared between different correlator algorithms. Typically, this number is a combination of the similarity score and some length indicator or the number of duplicate matches.
Source Length	The length of the source function or data item.
Dest Length	The length of the destination function or data item.
Votes	The number of references from previously accepted matches that would suggest that this is a correct match.
Source Address	The address of the function or data object in the source program.
Destination Address	The address of the function or data object in the destination program.
Algorithm	The algorithm that was used to generate this match.
Length Delta	The difference in lengths between the source and destination objects.
Source Label Type	The source of the label in the source program. (Imported, Analysis, User Defined, etc.)
Destination Label Type	The source of the label in the destination program. (Imported, Analysis, User Defined, etc.)
Markup Status	Displays an overview of the markup items status. There is a colored orb for each status type and that orb is either colored if at least one markup item has that status or else it is greyed out. An orange orb indicates that one or more markup items have been applied or marked. A green orb indicates at least one markup item has been applied. A purple orb indicates markup items that are rejected. A blue orb indicates there are markup items that have been ignored (either "Don't Know" or "Don't Care". And finally, a red orb indicates that at least one markup item could not be applied due to some error.
# Conflicting	The number of unique associations with either the same source or same destination address.  This is the number of associations that will become <a href="#">BLOCKED</a> if you accept the match in this row of the table.

### Match Table Actions

The **Accept Match** action marks a match (and all matches that have the same association) as being accepted. All competing matches will become blocked. [There are options](#) to auto-apply function names and create implied matches when accepting a match.

The **Apply Blocked Match** action will clear conflicting matches and then apply the match, which had been blocked by those conflicts, and its markup items [according to the apply settings](#).

The **Apply Markup** action will attempt to apply all the markup items for the match [according to the apply settings](#). If the match is not already accepted, it will first be marked as accepted.

The **Reject Match** action will mark the match as rejected.

The **Choose Match Tag** action allows the user to set a user-defined tag that has been created via the [Edit Tag](#) action.

The **Remove Match Tag** action removes any tag associated with the selected match(es).

The **Edit Tag** action allows the user to manage (create and delete) custom tags that can be applied to matches.

The **Clear Match** action will reset the match to unaccepted and undo any applied markup.

The **Remove Match** action will remove a manually created match from the matches table.

The **Make Selections** action will create selections in the source and destination tools for all matches selected in the table.

The **Table Selection Mode** allows you to change the behavior of the match table with regard to how it tracks table selections as you apply matches.

As you make changes to a match, the table will update. Sometimes as the table updates the changed match will disappear from the table (for example, if your filter settings are setup to hide applied matches and you have just applied a match). The default behavior is to keep the table selection on the same row, regardless of whether the match changes its position in the table or is removed from the table altogether.

Table Selection States:

Action Icon	Action Name	Description
	Track Selected Index	Causes the match table to maintain the selection for the selected <b>row</b> . So, for example, if you change a match, and that match is moved as a result of the table re-sorting, then the selection will remain on the row where the applied match used to be.
	Track Selected Match	Causes the match table to maintain the selection for the selected <b>match</b> . So, for example, if you change a match, and that match is moved as a result of the table re-sorting, then the selection will change to keep the applied match selected.
	No Selection Tracking	In this state the table will not restore selections. If changes are made to matches, the selection will be lost.

The **Settings** action will bring up the version tracking accept and apply options.

### Match Filters

The match table has an extensive assortment of filters. There are several commonly used filter controls at the bottom of the table:

1. **Text Filter** –allows you to filter based on any text in the table
2. **Score Filter** –allows you to filter on a range of scores. All scores are between 0 and 1
3. **Confidence Filter** –allows you to filter a range of confidence values. All confidence values will be greater than -9.999 and smaller than 9.999.
4. **Length Filter** –is used to filter out functions that are smaller than some number

Finally, the will show the ancillary filters available. The table below lists and describes the available filters. When an ancillary filter is applied, the icon will change to . Further, the icon may occasionally flash as a reminder that there is a filter applied.

Filter Name	Description
Match Type	This filter allows the user to show only function or data matches.
Association Status	This filter allows the user to show only matches whose association has one of the included status types. A useful setting for this filter is to turn off all but the <b>Available</b> status. This will cause the table to act like a "To Do" list.
Symbol Type	This filter allows the user to show only matches whose source or destination labels are of one of the included symbol types.
Algorithms	This filter allows the user to show only matches that were generated by one of the included types of correlating algorithms
Address Range	This filter allows the user to show only matches whose source or destination address is within the specified range.
Tags	This filter allows the user to show only matches whose tag is an included tag.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Markup Items Table](#)
- [Version Tracking Tool](#)
- [Version Tracking Introduction](#)

## Version Tracking Markup Items Table

*Marking up* a program means adding information to a program to help understand it. A markup item is some type of information that has been specified in the source program that you now wish to move to the equivalent function or data within the destination program. The Version Tracking Markup Items table shows a list of markup items for the currently selected match in the [Version Tracking Matches table](#).

Below the Markup Items table you can also view two listings, one for the source program and the other for the destination program. The dual listings provide an alternate way, besides the table, to view and manipulate the markup items. This also provides the capability to drag a markup item from the source program and apply it, where you want, by dropping it in the destination program. The [dual listing](#) will be described below in more detail.

The screenshot shows the 'Version Tracking Markup Items' window. At the top is a table titled 'Version Tracking Markup Items - [Session: Untitled] - 6 markup items'. The columns are: Status, Source Address, Dest Address, Markup Type, Source Value, Current Dest Value, and Original Dest Value. There are six rows, each with a status icon (green checkmark, green plus, yellow star, red circle) and corresponding values for the other columns. Below the table is a 'Filter:' input field and a 'Listing View' button. The main area contains two side-by-side assembly code listings. The left listing is for 'Source: Call\_strncpy\_s@0 in /VersionTracking/WallaceSrc' and the right listing is for 'Destination: FUN\_00411a900 in /VersionTracking/WallaceVersion2'. Both listings show the assembly code for the function, with various memory locations and registers highlighted in green. Arrows point from the source listing to the destination listing, indicating the flow of data or the application of markup items.

Once you have decided that you believe a function or data match is correct, the Markup Items table allows you to decide whether you want to apply or ignore individual markup items. Applying a markup item changes the information in the destination program using information from the source program. The Markup Items table lets you decide what to apply from the source program to the destination program and how to apply it.

### Markup Items Table Columns

You can add columns to and remove columns from the Markup Items table by right clicking on any column title in the table and choosing the **Add/Remove Columns...** button from the popup menu. The following table describes the default columns for the Markup Items table.

Column Name	Description
Status	The current status of this markup item. Is this item unapplied, applied, rejected, etc. Each possible markup item status is described below in more detail.
Source Address	This column shows the address where the markup item's value is coming from in the source program.
Dest Address	This column shows the address where the markup item will be applied or has been applied in the destination program. <b>No Address</b> is displayed if the address correlator couldn't determine a destination address to associate with the source address and the user hasn't manually specified the destination address.
Displacement	This column shows the relative displacement of the destination address as compared to the source address. Positive numbers indicate additions in the destination and negative numbers indicate subtractions.
Markup Type	This column identifies the type of markup item for this row. Each possible markup type is described below in more detail.
Source Value	This column shows the value for this markup type at the associated address in the source program.
Current Dest Value	This column shows the current value for this markup type at the associated address in the destination program.
Original Dest Value	This column shows what the value for this markup type was originally (before being applied) at the associated address in the destination program. If the item hasn't been applied yet, this will be the current destination value.

### Markup Type

There are many different types of markup that can be applied from the source to the destination. The table below lists the various types of markup items.

Some types of markup items can conflict with others. This simply means that once any markup is applied, then any of its conflicting markup types cannot be applied for that same Match result. The conflicting markup item that can no longer be applied is given a status of [Conflict](#).

Markup Type	Description
Function Name	The name labeling the function. Any non-default name in the source program can replace the function name in the destination program. A default function name has the "FUN_<name>" prefix. You can choose to only replace the destination function name when it is a default name or you can always replace it whether a default or not.
Function Signature	The signature of the function. The items included in the function signature are: the return type, the number of parameters and each parameter's associated data type, name, and comment and whether the function has variable arguments (varargs). There are several other markup apply options that are included with the function signature markup: the calling convention, the inline flag, the no return flag, and the call fixup for the function. These each have their own options for whether they get applied with the function signature.

	defined data type.
Parameter Names	A default parameter name begins with "param_". Default parameter names in the source program will not replace a defined parameter name in the destination program. If you choose to do a "Priority Replace" for the parameter names, whether or not the name is replaced depends on their source types. The options let you choose whether User Defined names or Imported names are higher priority. You can also specify whether the source name should replace the destination whenever their source types are the same.
Label	The labels on an instruction or defined data can replace or be added to those in the destination program at the associated address. If replacing the destination labels you can choose to only replace if the destination label is a default or replace all destination labels.
EOL Comment	The end of line comment from the source program can be added to an existing comment in the destination program or can simply replace the destination comment.
Plate Comment	The plate comment from the source program can be added to an existing comment in the destination program or can simply replace the destination comment.
Pre Comment	The pre-comment from the source program can be added to an existing comment in the destination program or can simply replace the destination comment.
Post Comment	The post comment from the source program can be added to an existing comment in the destination program or can simply replace the destination comment.
Repeatable Comment	The repeatable comment from the source program can be added to an existing comment in the destination program or can simply replace the destination comment.
Data Type	The data type for an address with defined data in the source program can replace the data type of an associated address of data in the destination program.

#### Markup Item Status

Each markup item has a status (indicated in the **Status** column). The table below lists each markup item status along with its associated icon in the table, background color in the code listing, and a description of that status.

Status	Icon	Color	Description
Unapplied		Orange	The markup item has not had anything done to it and if a destination address is specified the source and destination values are not the same.
No Address		Purple	The markup item has not had anything done to it because it doesn't have a destination address associated with it.
Same		Light Blue	The destination already has the same value as the source.
Added		Green	The source value was applied by adding it to the destination value.
Replaced		Green	The destination value was applied by replacing it with the source value.
Don't Care		Grey	The user set this markup item to indicate that we don't care about it and it should be ignored when applying the match for this item.
Don't Know		Grey	The user set this markup item to indicate that we don't know if it is correct or if it should be applied and therefore it should be ignored when applying the match for this item.
Conflict		Yellow	The user applied another markup item for this match that conflicts with this markup item and prevents it from being applied.
Reject		Red	The user rejected this markup item and it should be ignored when applying the match for this item.
Failed		Red	An attempt to apply this markup item failed. If you hover over the icon in the markup item table a tooltip will appear with more information about the failure.

#### Markup Item Actions

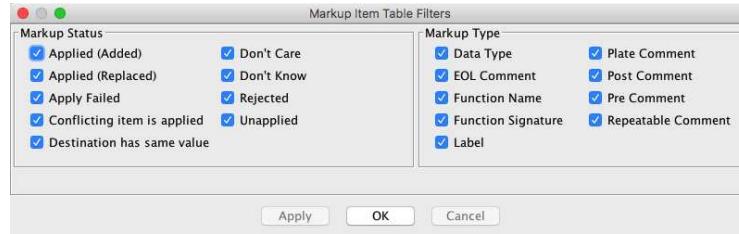
There are actions that can be performed on markup items in the table and in the dual listing. The table below lists these actions.

Action	Icon	Description
Apply (Use Options; Force If Necessary)		This action applies the selected markup items using the current apply match option for each markup type. For each selected markup item this action will force (or try to cause) the option to be applied. For example, if the option for a selected markup type is set to "Do Not Apply", this action would cause an add or replace to be performed depending on the item.
Apply (Add As Primary)		This action applies the source value for each selected markup item by adding it to the destination value. If possible the source markup item becomes the primary one.
Apply (Add)		This action applies the source value for each selected markup item by adding it to the destination value. The destination markup item remains the primary one.
Apply (Replace Default Only)		This action applies the source value for each selected markup item by replacing the destination value with the source value only if the destination value is a default value.
Apply (Replace First Only)		This action applies the source value for each selected markup item by replacing the destination value with the source value if the only defined data being replaced is the Data item at the destination address. If there is more than one defined data in the destination program that would be replaced by the source data type, then no replace will occur.
Apply (Replace)		This action applies the source value for each selected markup item by replacing the destination value with the source value. For data this will replace all defined data that is currently in the way of the source data type being created. <b>Important :</b> For Data markup that is replaced, the <i>Reset Mark-up</i> action will only be able to restore the single Data item that was originally at the Destination Address. Any other Data items that were replaced by this action will not be restored by <i>Reset Mark-up</i> .
Don't Care		This action sets the status of this markup item to Don't Care. This indicates that we don't care about this markup item and it shouldn't be applied when the match is applied.
Don't Know		This action sets the status of this markup item to Don't Know. This indicates that we don't know if this markup item is correct or don't know if it should be applied when the match is applied.
Reject		This action sets the status of this markup item to rejected. Once a markup item is rejected, it will be ignored when applying the match for this item.
Edit Destination Address		This action provides an edit address dialog that allows you to specify a destination address for any markup item that doesn't already have an address as indicated by "No Address". It also allows you to change the destination address if the one specified for the markup item is incorrect. In some circumstances you can't edit the destination address and an information dialog indicating why it's not editable will be displayed instead.
Reset		This action resets the markup item back to its original unapplied state and restores the markup in the destination program back to its original state if necessary.

#### Markup Items Filters

Below the Markup Items table is a text field labeled "**Filter**". This text field is the primary text filter. Typing text into the filter will remove all markup item rows that don't have that text in at least one of the displayed fields.

The button to the right of the primary text filter will show the ancillary filters that are available. You can apply an ancillary filter by removing the check mark from its box and pressing the **Apply** or **OK** button. Once an ancillary filter is applied the ancillary filters icon will change to . Further, the icon may occasionally flash as a reminder that there is a filter applied. The following image shows the available ancillary filters.



The table below lists and describes the groups of available filters.

Filter Group	Description
Markup Status	Markup items with any status that is checked will appear in the markup item table. Any item with a status that is not checked will be filtered out and not displayed in the table.
Markup Type	Markup items of any type that is checked will appear in the markup item table. Any item of a type that is not checked will be filtered out and not displayed in the table.

The window title above the Markup Items table displays an indicator that a filter is set on the markup items table and not all items are shown. For example, the title might contain "2 markup items (of 8)" indicating 6 items are hidden or filtered out.

### Comparison Views

There can be multiple types of comparison views which are provided by various plugins. Illustrated below is the Listing View. This view provides a side by side comparison of two functions (source and destination) that make up the currently selected function match. For a selected data match, the Listing View will provide side by side comparison of the data.

#### Dual Listing Window

The dual listing appears below the Markup Items table when the **Listing View** tab is selected. It shows one listing for the source program and another listing for the destination program. When a function match is selected, each listing displays only the addresses that are part of the function. When a data match is selected, the source listing will display the addresses that make up that data and the destination listing will display the addresses that would be affected if it were applied.

The two listings can be positioned with one above the other or they can be side by side. You can also hide the dual listings altogether. The colored background areas in each listing indicate anywhere that one of the markup items exists for the current match. If you right click on a markup item in either of the listings, a popup menu will appear containing actions, such as apply, reject, reset, etc., for that markup item. You can also drag and drop a markup item from the source listing to an address in the destination listing causing it to be applied at that address.

The following shows the dual listings below the Markup Items table with the source listing positioned above the destination listing.

Status	Source Address	Dest Address	Markup Type	Source Value	Current Dest Value	Original Dest Value
Applied (Added)	00411ab0	00411a90	Function Name	Call_strncpy_s	FUN_00411a90	FUN_00411a90
Applied (Replaced)	00411ab0	00411a90	Function Signature	void __cdecl Call_strnc...	void __cdecl FUN_0041...	void __cdecl FUN_0041...
Apply Failed	00411ad3	00411ab3	EOL Comment	rsize_t _MaxCount for s...	rsize_t _MaxCount for s...	rsize_t _MaxCount for s...
Conflicting item is applied	00411ad7	00411ab7	EOL Comment	char *_Src for strncpy_s	char *_Src for strncpy_s	char *_Src for strncpy_s
Destination has same value	00411ad8	00411ab8	EOL Comment	rsize_t _SizelnBytes for ...	rsize_t _SizelnBytes for ...	rsize_t _SizelnBytes for ...
Unapplied	00411add	00411abd	EOL Comment	char *_Dst for strncpy_s	char *_Dst for strncpy_s	char *_Dst for strncpy_s

#### Dual Listing Actions

The **Toggle Dual Listing Visibility** toolbar action allows you to control whether or not the dual listing is displayed. When it is toggled on (appearing pressed in) the dual listings are shown below the Markup Items table.

The following shows the Version Tracking Markup Items window with the Dual Listings visibility toggled off.

Version Tracking Markup Items - [Session: Untitled] - 6 markup items						
Status	Source Address	Dest Address	Markup Type	Source Value	Current Dest Value	Original Dest Value
✓	00411ab0	00411a90	Function Signature	void __cdecl Call_strc...	void __cdecl FUN_0041...	void __cdecl FUN_0041...
✓	00411ab0	00411a90	Function Name	Call_strcpy_s	FUN_00411a90	FUN_00411a90
✓	00411ad3	00411ab3	EOL Comment	rsize_t _MaxCount for ...	rsize_t _MaxCount for ...	
✗	00411ad7	00411ab7	EOL Comment	char * _Src for strncpy_s		
✗	00411ad8	00411ab8	EOL Comment	rsize_t _SizeInBytes for ...		
✗	00411add	00411abd	EOL Comment	char * _Dst for strncpy_s		

From the toolbar menu ▾ there are selectable options that apply to the dual listing.

- Show Listing Format Header

When this item has a checkmark next to it, the header is displayed above the source listing and allows the listing fields to be adjusted. More information about using the header to change the listing format can be found in the help for the [Browser Field Formatter](#).

- Show Listings Side By Side

When this item has a checkmark next to it, the destination listing will be positioned to the right of the source listing. Otherwise the destination listing is positioned below the source listing.

- Synchronize Scrolling

When this item has a checkmark next to it, the button will show in the toolbar and the dual listing will synchronize scrolling of the left and right function in their views. Scrolling or moving the cursor in one side will cause the location in the other side to also scroll or move if a matching location can be determined.

Otherwise when this item has no checkmark next to it, the button will show in the toolbar and the left and right sides of the dual listing will scroll independent of the other side.

The following shows the *format header* being displayed for the dual listings.

Version Tracking Markup Items - [Session: Untitled] - 6 markup items						
Status	Source Address	Dest Address	Markup Type	Source Value	Current Dest Value	Original Dest Value
✓	00411ab0	00411a90	Function Name	Call_strcpy_s	FUN_00411a90	FUN_00411a90
✓	00411ab0	00411a90	Function Signature	void __cdecl Call_strc...	void __cdecl FUN_0041...	void __cdecl FUN_0041...
✓	00411ad3	00411ab3	EOL Comment	rsize_t _MaxCount for ...	rsize_t _MaxCount for ...	
✗	00411ad7	00411ab7	EOL Comment	char * _Src for strncpy_s		
✗	00411ad8	00411ab8	EOL Comment	rsize_t _SizeInBytes for ...		
✗	00411add	00411abd	EOL Comment	char * _Dst for strncpy_s		

Filter:

Listing View

Source: **Call\_strcpy\_s()** in /VersionTracking/WallaceSrc

Address Break	Plate	Function	Variable	Instruction/Data	Open Data	Array
Function Signature						
Thunked Function						
Function Call-Fixup						
Function Tags						
Register						

```

*****
*          FUNCTION
*****
void __cdecl FUN_00411a90(char * _Dst, char * _Src, rsize...
<VOID>      <RETURN>
Stack[0x4]:4   _Dst
Stack[0x8]:4   _Src
Stack[0xc]:4   _MaxCount
undefined1     Stack[-0xc4]:1 local_c4
FUN_00411a90
XREF[1]:      00411aba(R)
XREF[1]:      00411abb(R)
XREF[1]:      00411abc(R)
XREF[1]:      00411ac(R)
XREF[1]:      thunk_FUN_00411a90:00411172(T),
thunk_FUN_00411a90:00411172(i)

```

Destination: **FUN\_00411a90()** in /VersionTracking/WallaceVersion2

```

*****
*          FUNCTION
*****
void __cdecl FUN_00411a90(char * _Dst, char * _Src, rsize...
<VOID>      <RETURN>
Stack[0x4]:4   _Dst
Stack[0x8]:4   _Src
Stack[0xc]:4   _MaxCount
undefined1     Stack[-0xc4]:1 local_c4
FUN_00411a90
XREF[1]:      00411aba(R)
XREF[1]:      00411abb(R)
XREF[1]:      00411abc(R)
XREF[1]:      00411ac(R)
XREF[1]:      thunk_FUN_00411a90:00411172(T),
thunk_FUN_00411a90:00411172(i)

```

The following shows the dual listings being displayed *side by side*.

The screenshot shows the 'Version Tracking Markup Items' window. At the top is a table with columns: Status, Source Address, Dest Address, Markup Type, Source Value, Current Dest Value, and Original Dest Value. The table contains six rows, each with a status icon (green checkmark, green plus, orange star, red circle), source address (e.g., 00411ab0, 00411ad3), destination address (e.g., 00411a90, 00411ab7), type (Function Signature, Function Name, EOL Comment), source value (void \_\_cdecl Call\_strnc..., Call\_strncpy\_s, rszie\_t \_MaxCount for s...), current dest value (void \_\_cdecl FUN\_00411a90, FUN\_00411a90, rszie\_t \_MaxCount for s...), and original dest value (void \_\_cdecl FUN\_00411a90, FUN\_00411a90, rszie\_t \_MaxCount for s...).

Below the table is a 'Filter:' input field and a 'Listing View' button. The main area displays two assembly code listings side-by-side under the heading 'Source: Call\_strncpy\_s@ in /VersionTracking/WallaceSrc' and 'Destination: FUN\_00411a90@ in /VersionTracking/WallaceVersion2'. The source listing shows the assembly code for the source function, and the destination listing shows the assembly code for the destination function. The assembly code includes instructions like void \_\_cdecl, char \*, Stack[0x8]:4, Stack[0xc]:4, Stack[-0xc4]:1, and local\_c4.

#### Background Colors in the Listings

The background colors in the listings are used to identify where you have markup items. The colors are also used to indicate the status of each markup item. For example a green background indicates markup items that have already been applied, orange indicates markup that is still unapplied, blue indicates a markup item where the source and destination already match, grey indicates a markup item you have chosen to ignore, pink indicates a markup item you have rejected, and red indicates a markup item where an attempt to apply the item to the destination failed. Each Markup Item Status and its associated color is defined above in a [Markup Item Status table](#).

#### Drag and Drop

You can use drag and drop in the dual listing window to apply markup items. This provides a quick visual way to apply a particular markup item to a specific spot in the destination listing. If the address correlator has associated the incorrect destination address with the markup item's source address or if the correlator couldn't determine a destination address, you can quickly apply an item by dragging it to the correct address in the destination listing. Manually editing the destination address and then applying the item is more time consuming, but can be done instead of using drag and drop. Drag and drop of markup items isn't available from the separate Source and Destination Tools because dragging begins selection of code there.

Drag and drop is available for all markup items in the dual listing where it makes sense to apply a markup item. For example, you can't drag an already applied markup item. To apply an item, click on the markup item in the source listing you want to apply and drag it to the code in the destination listing where you want it applied. When you start dragging, the cursor will change to indicate that dragging is allowed for the item. Once you drop the item, the destination listing will update to show the markup where it was dropped, the background color will change, and the markup item's status should change in the table.

Some markup item's can only be applied at a particular spot. For example, a function's name or signature can only be applied to the function. Therefore the destination function's entry point is the expected destination address. In this case, the destination address will become the destination function's entry point no matter where you drop the markup item in the destination listing.

Provided by: Version Tracking Plugin

#### Related Topics:

- [Version Tracking Matches Table](#)
- [Version Tracking Tool](#)
- [Version Tracking Introduction](#)

## Version Tracking Functions Table

The screenshot shows a software interface titled "Version Tracking Functions - [Session: Untitled] - Source Functions / Destination Functions". It displays two tables side-by-side. The left table is for the source program ("Source = /VersionTracking/WallaceSrc") and the right table is for the destination program ("Destination = /VersionTracking/WallaceVersion2"). Both tables have columns for "Label", "Location", and "Function Signature". The source table contains functions like Gadget, FUN\_004114b0, FUN\_004114f0, etc. The destination table contains functions like FUN\_00411430, FUN\_004114a0, FUN\_004114e0, etc. There are filters at the bottom of each table.

The functions table shows a list of all functions in the source program and the destination program. You can filter this table to show only those functions that are not part of a match. This is useful if you would like to [create a manual match](#) from two functions.

If you select function in each table and there exists already a match between the two functions, then the following status message will be displayed in the table: A match already exists between <sourcefunctionname> and <destinationfunctionname>.

The title of this window will show, for each program, the number of total functions, as well as the number of functions filtered-out of the table, if any filters are applied.

### Functions Table Columns

Column Name	Description
Label	This column shows the label for function in the given row.
Location	This column shows the address for the function in the given row.
Function Signature	This column shows the function signature for the function in the given row.

### Functions Table Actions

#### CreateManualMatch

The **CreateManualMatch** action (✚) allows the user to create a match for the selected function in source table to the selected function in the destination table. The action will be disabled if you do not have a single function selected in both tables.

#### Create And AcceptManualMatch

The **Create And Accept Manual Match** action (✚) allows the user to create a match for the selected function in source table to the selected function in the destination table and then automatically accept it. The action will be disabled if you do not have a single function selected in both tables.

#### Create And Apply Manual Match

The **Create And Apply Manual Match** action (✓) allows the user to create a match for the selected function in source table to the selected function in the destination table and then accept it and then automatically apply any appropriate markup items from the source to the destination program. The action will be disabled if you do not have a single function selected in both tables.

#### Select Existing Match

The **Select Existing Match** action (≡) will select the existing match in the matches table. To use this action you must have one function selected in source table and one function selected in the destination table. Further, the action will only be enabled if a match exists for the two selected functions.

#### Functions Table Filter

The **Functions Filter** action filters functions from the tables based upon the chosen state of the action. You can change the state of the filter from the actions toolbar using the drop-down menu. (✚ ≡ ⌂ X).

This list below shows the available filter states:

- **Show All Functions** (⊕) –Shows all functions found in the source and destination programs.
- **Show Only Unmatched Functions** (⊖) –Shows only functions in the source and destination programs that are not part of any match. This is useful for showing functions that were not matched by any of the [program correlators](#).
- **Show Only Unaccepted Match Functions** (⊖) –Shows only functions in the source and destination programs that are not part of **an accepted match**. This means that the functions visible in the tables will either be part of no match or part of a match that has not been accepted. This is useful for showing functions that you have not yet accepted as being part of a valid match.

#### Show/Hide the Function Comparison Panel

The **Toggle Visibility of Dual Comparison Views** action (□) will toggle whether or not a function comparison panel is displayed below the source and destination function tables. As you select a function in the source or destination table, it is displayed in the function comparison panel so you can visually compare the source and destination functions.

There are other toolbar and popup actions that are available for this function comparison panel. See the help for the [Function Comparison Window](#) to learn more about using these.

### **Text Filter**

Each table has a text field at that bottom labeled **Search Filter** that allows you to search for text in the respective table. If the text searched is contained in any column, then that column will remain in the table; otherwise the row will be filtered out.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Matches Table](#)
- [Version Tracking Tool](#)
- [Version Tracking Introduction](#)
- [Function Comparison Window](#)

## Version Tracking Related Matches Table

### Related Matches

The **Related Matches** table is a dockable window that is available in both the Version Tracking Source Tool and Destination Tool. By default it appears as a small window in the bottom right corner of the tool. It shows a list of potential matches for the function or data where the cursor is currently located within the tool's listing. This match information is generated by the correlators chosen by the user.

The **Related Matches** Table provides fields with the same information as those in the [Matches Table](#). For a description of the table characteristics, see that table. The main difference between this table and the Matches Table is that this table only displays the matches related to the current function or data.

#### Version Tracking Matches For Source

In the Source Tool the **Related Matches** table shows all correlator or manually generated matches for the single function or data that is currently active in the Source Tool. Likewise the **Related Matches** table in the Destination Tool shows all correlator or manually generated matches for the single function or data that is currently active in the Destination Tool.

The following image shows an example **Related Matches** table from a Version Tracking Source Tool.

Score	Confid...	Votes	# C...	Dest Label	Dest A...	Sour...	Des...	Algorithm
1.000	1.000	0	1	addPerson	00411830	115	115	Exact Symbol Name M...
1.000	1.000	0	1	addPerson	00411830	115	115	Exact Function Instruc...
1.000	0.000	0	1	FUN_004118c0	004118c0	115	261	Manual Match

The information above the table indicates the name and address for the current tool's function or data. In the Source Tool if the cursor is within a function then the information above the table shows the current source function's name and entry point address. *In this case, our cursor is in the function addPerson in the Source Tool.*

The table itself contains a row for each of the matches (if any) for the current function or data. Each table row shows the name and address of each possible match in the Destination Tool, along with other match information. *In this case, correlators have found more than one potential match to the current function in the Source Tool. These matches might be entirely different functions or the same function but found with different correlators. In this example, the user decided that "addPerson" in the source program matched function addPerson in the destination program. It has already been accepted by the user as the correct match. Therefore both rows that indicate a match to the same destination function are marked accepted. By definition, there can only be one accepted match per function.*



Selecting a row in the Related Match Table causes the Destination Tool to navigate to that selected destination address.

#### Version Tracking Matches For Destination

The Destination Tool has its own **Related Matches** table. This has information for the function or data where the cursor is currently located in the Destination Tools listing. Each row of this table indicates a match to a source function. The following shows the related matches in the Destination Tool that are related to what was selected above in the Source Tool's table.

Score	Confid...	Votes	# C...	Source Label	Source A...	Sour...	Des...	Algorithm
1.000	1.000	0	1	addPerson	00411860	115	115	Exact Symbol Name M...
1.000	1.000	0	1	addPerson	00411860	115	115	Exact Function Instruc...

*In this case the Destination Tool's cursor is in function addPerson. There are two matches displayed in the table which are to the same source address but were arrived at by different correlation algorithms.*



Selecting a row in this table causes the Source Tool to navigate to that selected source address.

### Actions

#### Select Match in VT Matches Table

As you select various matches in either **Related Matches** table, the other (source or destination) tool will navigate to the function or data associated with the match. However, the selected match in the **Version Tracking Matches** table will remain the same. To force the selection in the **Version Tracking Matches** table to match selection in the **Related Matches** table, use this action. This action can be initiated as follows:

Select a match in the **Related Matches** table and do either of the following.

- Click the toolbar button,
- Right-click on a row in the **Related Matches** table to get a popup menu. From the popup menu choose **Select Match in VT Matches Table**.

Either of these will cause the same match to be selected in the Version Tracking Matches table. When the match becomes selected in the Matches table, the Markup Items table will update to show the items for that match and the markup items will be marked with colors in the listings.

Provided by: *Version Tracking Plugin*

Related Topics:

- [Version Tracking Matches Table](#)
- [Version Tracking Tool](#)
- [Version Tracking Introduction](#)

## Version Tracking Implied Matches Table

The implied matches table displays a list of [Implied Matches](#) for the selected match in the [Matches Table](#). The implied matches are generated by pairing up the **outgoing** references from the two functions in the selected match. Both [function references \(calls\)](#) and [data references](#) can generate implied matches. If there is already a match in the session that has the same [association](#) as the implied match, then an implied match is not created. Instead, it is indicated by incrementing the vote count for the current match. If no other match exists, and implied match is created if it is listed as a "Possible Implied Match". For those implied matches that don't already have a corresponding existing match, the user can create new matches.

Tag	S...	Type	Score	Confid...	Vot...	#...	Source Label	Sour...	M...	Dest Label	Dest Ad...	So...	D...	Algorithm
1		Function	1.000	1.000	0	0	FUN_004116c0	004116c0		FUN_004116b0	004116b0	42	42	Exact Function Instr...
1		Function	1.000	1.000	0	0	addPeople	00411700		FUN_004116f0	004116f0	152	152	Exact Function Instr...
1		Function	1.000	1.000	0	0	addPerson	00411860		FUN_00411830	00411830	115	115	Exact Function Instr...
1		Function	1.000	1.000	0	0	main	00411a30		FUN_00411a10	00411a10	95	95	Exact Function Instr...

Source Ref...	Dest Refer...	Type	Score	Confid...	Vot...	# C...	Source Label	Source...	Dest Label	Dest Addr...	Algorithm
00411a53	00411a33	Function	1.000	1.000	0	0	addPeople	00411700	FUN_004116f0	004116f0	Exact Function Instr...
00411a61	00411a41	Function	1.000	0.000	0	-1	initializePeople	004117c0	FUN_004117b0	004117b0	Possible Implied Match
00411a69	00411a49	Function	1.000	0.000	0	-1	deployGadget	004118f0	FUN_004118c0	004118c0	Possible Implied Match
00411a86	00411a66	Function	1.000	1.000	0	0	_RTC_CheckEsp	00411b80	_RTC_CheckEsp	00411b60	Exact Function Instr...
00411a4e	00411a2e	Data	1.000	0.000	0	-1	personList	00418138	DAT_00418138	00418138	Possible Implied Match

In the example shown above, the matches table has a match selected for a function named "\_mainCRTStartup". The Implied Matches table shows a list of implied matches wherever references from "\_mainCRTStartup" in the source program match a similar reference in the "\_mainCRTStartup" function in the destination program. The idea is that if the selected match in the matches table is Accepted, then these other matches may also be "correct".

### Version Tracking Implied Match

An implied match is simply a function or data match that is implied because of the correlation of references made by some other match.

#### Implied Match Table Columns

Column Name	Description
Source Reference Address	Displays the address in the source program where the implied match is referenced.
Destination Reference Address	Displays the address in the destination program where the implied match is referenced.
Status	Displays the status of the association for this implied match.
Type	Displays the type of this implied match. Either Function or Data.
Score	The score of the best match with the same association as this implied match, or 0 if no matches exist with this association.
Confidence	The confidence score of the best match with the same association as this implied match, or 0 if no matches exist with this association.
Multiple Source Labels?	Icon indicating there is more than one label at the match's source address and a number indicating how many labels. The tooltip can be viewed to see the label names.
Source Label	Displays the label at the source address of the implied match.
Source Address	Displays the source address of the implied match.
Multiple Dest Labels?	Icon indicating there is more than one label at the match's destination address and a number indicating how many labels. The tooltip can be viewed to see the label names.
Dest Label	Displays the label at the destination address of the implied match.
Dest Address	Displays the destination address of the implied match.
Algorithm	Displays the algorithm of the best match with the same association as this implied match, otherwise, "Possible Implied Match" is displayed.

#### Implied Match Table Actions

**NavigateReferences** When this action toggled on, selecting a row in the implied matches table will cause the [sub-tools](#) to navigate to the [From Address](#) of the references used to create the selected implied match.

**Navigate Match** When this action toggled on, selecting a row in the implied matches table will cause the [sub-tools](#) to navigate to the source and destination addresses of the selected implied match.

The **Accept Implied Match** action creates a match in the [Matches Table](#) if one does not already exist, and then sets its status as 'Accepted'. Note, the Implied Matches shown in the table are not saved unless this action is applied.

#### Text Filter

The text filter allows filtering on any text displayed in the table.

Provided by: Version Tracking Plugin

Related Topics:

- [Version Tracking Matches Table](#)
- [Version Tracking Tool](#)
- [Version Tracking Introduction](#)

# *Additional Support Tools*

This help section provides help on support features that are useful for debugging Ghidra and using Ghidra in alternate ways.

Most of the available support tools and documentation can be found in:

**<Ghidra Install Dir>/support/**

# Headless Analyzer

The *HeadlessAnalyzer* is a command-line-based (non-GUI) version of Ghidra that allows users to:

- Create and populate projects
- Perform analysis on imported or existing binaries
- Run non-GUI scripts in a project (scripts may be program-dependent or program-independent)

The Headless Analyzer can be useful when performing repetitive tasks on a project (i.e., importing and analyzing a directory of files or running a script over all binaries).

## Headless Analyzer Options

The following options are available for the Headless Analyzer:

```
analyzeHeadless[<project_location><project_name>[/<folder_path>]] |  
[ghidra://<server>[:<port>]/<repository_name>[/<folder_path>]]  
  
[[-import [<directory>|<file>]+] | [-process [<project_file>]]]  
[-preScript<ScriptName> [<arg>]*]  
[-postScript<ScriptName> [<arg>]*]  
[-scriptPatM<path1>[;<path2>...]" ]  
[-propertiesPatM<path1>[;<path2>...]" ]  
[-scriptlog <path to script log file>]  
[-log <path to log file>]  
[-overwrite]  
[-recursive]  
[-readOnly]  
[-deleteProject]  
[-noanalysis]  
[-processor<languageID>]  
[-cspc<compilerSpecID>]  
[-analysisTimeoutPerFile<timeout in seconds>]  
[-keystore<KeystorePath>]  
[-connect [<userID>]]  
[-p]  
[-commit["<comment>"]]  
[-max-cpu <max cpu cores to use>]  
[-loader <desired loader name>]
```

## Accessing the Headless Analyzer

- The shell script that launches the Headless Analyzer can be found in your Ghidra installation's *support* folder.

```
ghidra_x.x/support/analyzeHeadless[.bat]
```

- Execute the *analyzeHeadless* shell script from the command line with the desired options.

## Headless Analyzer Documentation

- The *analyzeHeadlessREADME.html* file contains details on Headless Analyzer usage and options. It is located in your Ghidra installation's *support* folder.

```
ghidra_x.x/support/analyzeHeadlessREADME.html
```

# Undo/Redo

*Undo* is a generalized capability for returning a program back to a state prior to the last edit operation. *Redo* is the opposite of *Undo* and it effectively reapplies the last edit operation. *Undo* can be applied repeatedly to erase the effects of a sequence of edit operations up to a current limit of 20. After performing a sequence of *Undo* operations, the same number of *Redo* operations are available. However, the instant a new edit is performed, the *Redo* list is cleared.

- To *undo* an edit operation, select **Edit** → **Undo** from the main menu or press the  button on the tool bar.
- To *redo* an edit operation, select **Edit** → **Redo** from the main menu or press the  button on the tool bar.



Hovering the mouse over the  button will display the name of the edit operation that would be "undone". Similarly, hovering the mouse over the  button will display the name of the edit operation that would be "redone".

# Ghidra Glossary

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#)  
Y Z

## A

### Action

An operation the user can perform in Ghidra. All menu items, keybindings, and toolbar buttons are actions.

### Address

A number that identifies a specific location in memory.

### Address Range

A sequential set of addresses within a single Address Space identified by a minimum and maximum address.

### Address Set

A collection of Addresses.

### Address Space

The set of all legal addresses in memory for a given processor. The nature of each address space is defined by the processor and language implementation.

## Address Table

Two or more consecutive addresses in memory.

## Analyzer

A software module that examines and annotates the code in a program to help reveal the behavior of that program. Examples are disassembly, function generation, and stack analysis.

## Archive

See [Data Archive](#).

See [Project Archive](#).

## Assembly Language

Programming language closely associated with an individual processor.

## Auto Analysis

Automated way to run all the analyzers in the appropriate order (Example: function creation before stack analysis).

# B

## Back Reference

Another name for the [Source Address](#) in a reference.

## Background Task

Any action that runs in the background allowing the user to perform other tasks.

## Base Address

An address from which other addresses are derived using offsets.

## Basic Block

A sequence of instructions that has no flow into or out of the sequence except for the top and bottom (i.e., no branching).

## Basic Block Model

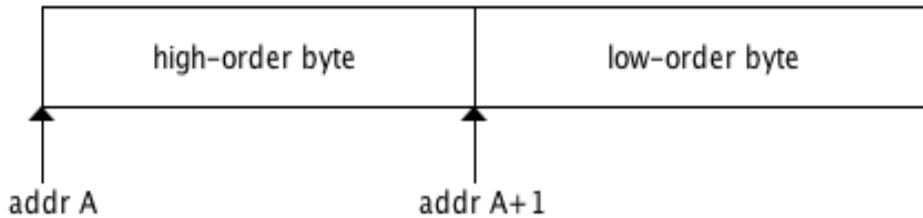
A [Block Model](#) which partitions code into small runs of instructions based on points where instruction flow changes. Jump, Call, and Branch instructions will cause the execution flow to change. Arithmetic and store/load instructions do not break a Basic Block because they do not change the execution. A label will also end one block and begin another.

## Binary Data

Bytes that make up a program.

## Big Endian

Byte order for storage such that the high-order byte is at the starting address, as shown in the figure below, where increasing memory addresses are going from left to right.



## Block Model

A model which partitions the code into address ranges based on some set of rules. The most obvious model partitions the code into subroutines.

## Bookmark

Marker used to designate frequently-visited or important locations.

## Bookmark Type

Attaches meaning to a bookmark to indicate its use.  
Example: Note, Info, Analysis, etc.

## Browser Field

Individual program elements displayed by the Code Browser's listing window.

## Byte Viewer

A Ghidra component used to display and edit the bytes in

a program.

## C

### Call Block Model

See [Subroutine Model](#).

### Call Graph

Graph displaying the relationships between "function calls" in a program.

### Calling Convention

Determines how a function receives arguments and returns results. The available calling conventions are currently determined by the program's language specification.

### Check-In

The process of contributing and merging changes from a "checked-out" program to the globally shared version of the program.

### Check-Out

The process of retrieving the latest version of a shared program for the purpose of making changes.

### Classpath

Path to search for Ghidra java code.

## Clear

Process of removing information from a program (Example: symbols, comments, everything, etc).

## Code Browser

A [Default Ghidra tool](#) for displaying and working with program listings.

## Code Block

See [Basic Block](#)

## Code Unit

An [Instruction](#) or [Data Item](#) in the listing.

## Computed Call

Call instruction whose destination is dynamically computed.

## Conditional Call

A call instruction that is executed conditionally.

## Conditional Jump

A jump instruction that is executed conditionally.

## Connecting Tools

Process of coordinating two or more [Tools](#) with respect to navigation and selection.

## Context Sensitive Menu

Menu that changes depending on cursor location. In other words, only actions which are appropriate for the type of information on which the cursor resides are available from the menu.

## Contrib Plugins

Term used to indicate user-contributed plugins.

## Core Plugins

Term applied to plugins that are supported by the Ghidra team.

## Cycle Group

A sequence of data types applied using the same action repeatedly (i.e. byte→word→dword→qword).

# D

## Data (item)

Bytes in the program's memory that are not interpreted as instructions.

## Data Type Archive

File used to store user-defined data types independent of a specific program.

## Data Component

A data item inside a structure or array.

## Data Type

A generalization of data which uniquely defines its specific attributes such as size, structure and format.  
Example: byte, float, double.

## Dead Code

Unreachable code.

## Decompiler

Ghidra module for translating assembly language to C.

## Default Ghidra Tool

- 1 A pre-configured CodeBrowser [Tool](#) that is ready to use when Ghidra is installed.
- 2 The tool that has been designated to run when you double click on a program in the Project Window.

## Destination Address

The "To" address in a reference "From–To" address pair. An address that is referred "To" by an instruction operand or pointer.

## Diff

See [Program Diff](#).

## Direct References

Locations in memory where the bytes make up the address of the current location in the browser. See [Search for Direct References](#).

## Disassemble

The process of interpreting program bytes as assembly instructions.

## Disassemble, Restricted

The Ghidra disassembly mode where disassembly is restricted to the current selection.

## Disassemble, Static

The Ghidra disassembly mode where code flows are not followed and only bytes at the current location or selection are disassembled.

## DLL

Abbreviation for Dynamic Link Library. A shared library on a Windows platform.

## Docking Window Component

Ghidra user interface component that can be positioned and sized by the user.

## DWord

A 4-byte integer data type.

## Dynamic Data Type

Data types whose structure varies depending on the data bytes on which they are applied.

## E

### ELF

Abbreviation for Executable Linking Format. File format used by Unix and Linux operating systems for storing executable programs.

## End of Line (EOL) Comment

Comments that are displayed to the right of the instruction.

## Endian

Byte ordering. See [Big Endian](#), [Little Endian](#).

## Entry Point

Location in a program where execution begins.

## Enum Data Type

Ghidra data type for modeling C-type enums.

## Equate

A string substitution for numeric values appearing in instruction operands.

## Exporter

Ghidra module for storing program information in various file formats (XML, HTML, ASCII, etc).

## External Reference

A reference from a location in one program to a location in another program.

# F

## Fall Through Address

The address of the next sequential instruction to be executed.

## Favorite Data Type

Ghidra data types that can be accessed via the popup-menu.

## Flow

A.K.A. Instruction Flow or Control Flow. This the sequence of instructions that are executed as a program runs, including branching and fall-through.

## Flow Graph

Graph that shows basic instruction flow.

## Forward Refs

Another name for the [Destination Address](#) in a reference.

## Fragment

A set of addresses used by the [Program Tree](#) to organize code.

## Front End

See [Ghidra Project Window](#).

## Function

A program element that is referenced via a call instruction. A [function](#) has an entry point, a body of instructions, a return data type, and optionally parameters, local variables, and local register variables.

## Function Signature

The name, return type, and parameters of a function.

# G

## Ghidra

Ghidra is a java-based framework for reverse engineering. It provides built-in capabilities for reverse engineering along with support for user-provided plugins.

## Ghidra File

Any file that is part of a [Ghidra Project](#).

## Ghidra Program File

Ghidra files containing information about a program.

## Ghidra Project

Ghidra organizes work into projects. All work is performed in the context of a project.

## Ghidra Project Window

The main Ghidra interface for managing [program files](#).

## Global Namespace

Symbols that are not in any specific namespace are said to be in the "global" namespace. The global namespace is the default namespace.

## **gzf**

File extension given to Ghidra program database files that have been "zipped up".

## **H**

### **Hex Integer**

A display format in the [Byte Viewer](#) used to display integer values in hex.

### **Hijacked File**

A local program file in a project that is "hiding" a shared program with the same name. Users cannot access the shared program until the local file is removed.

### **Highlight**

A more permanent type of selection.

### **History**

List of changes made to labels or comments.

## **I**

## IDA Pro

A commercially available reverse engineering tool.

## Importer

Reads a file (.xml, .dll, .so, etc), and converts its contents into a Ghidra program. Ghidra contains multiple importers (corresponding to a specific set of formats). Additional importers can easily be added.

## Initialized Block

Memory block whose byte values exist as opposed to uninitialized blocks whose byte values are unknown.

## Instance Settings, Data Type

Display options for *individual* data items in Ghidra. For example one byte can be displayed as decimal while another is displayed as hex. Also see [Data Type Default Settings](#).

## Instruction

An assembly level command such as MOV, JMP, etc.

## Intel Hex

Binary file format specified by Intel generally used for ROM images.

## Isolated Entry Model

A block model defining subroutines. A subroutine block must have only one entry point, but may share code with another subroutine. The subroutine body will stop if another is called or a source entry point is encountered.

## K

### Key Binding

Keyboard shortcut for invoking Ghidra functionality.

## L

### Label

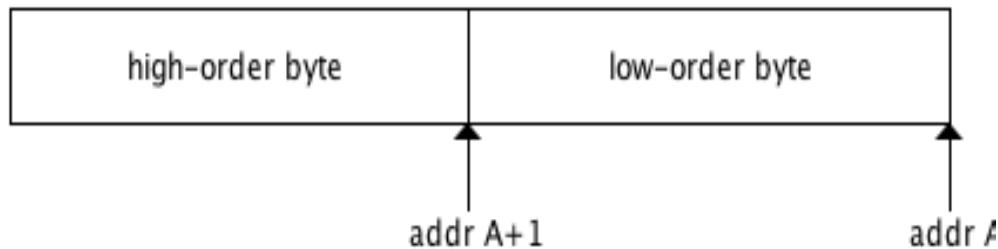
Symbol name associated with an address in memory.

### Language

The set of instructions associated with a computer processor.

### LittleEndian

Byte order for storage such that the low-order byte is at the starting address, as shown in the figure below, where the increasing memory addresses are going from right to left.



## **Listing**

The display of the assembly language along with comments and other markup information.

## **Local Menu**

A menu that is associated with a specific Ghidra docking window.

## **Local Symbol**

A symbol that is local to a particular function.

# **M**

## **Marker**

Used to indicate a significant location in a program (ex: bookmarks, search results, analysis problems, etc).

## **Memory**

The component of a program that contains the raw bytes and the addresses where those bytes are located in the program's address space.

## Memory Block

A contiguous set of bytes anchored at an address. Memory consists of one or more Memory Blocks.

## Memory Map

Ghidra GUI that allows the user to view and edit the Memory Blocks of a program.

## Memory Reference

A reference from a mnemonic or operand to another address in the same program.

## Merge

1. The process of resolving the differences between a checked out version with the globally shared version of a program.
2. The process of retrieving changes made by others to the globally shared version of the program and incorporating those changes into your program **without** introducing your changes into the shared version (i.e. doing a check-in).

## Microcode

Low-level instructions used to implement a machine instruction.

## Mnemonic

The name of an instruction as it appears in the assembly listing. (i.e. mov, add, jmp, etc.)

## Modal

Dialogs that prevent user interaction with any other Ghidra component until the dialog is dismissed.

## Motorola Hex

Binary file format specified by Motorola generally used for ROM images.

## Multiple Entry Model

A block model defining subroutines. A subroutine block may have multiple entry points and may not overlap code from other subroutines.

## Multi-User

Term used when multiple users are working together using shared projects and programs.

# N

## Name Space

Defines a scope such that all symbol names within that scope are unique.

## NE

Abbreviation for New Executable. File format used by Windows 3.1.x operating systems for storing executable programs.

# O

## Offcut

A reference into the middle of some instruction or data item.

## Offline

A situation when a shared Ghidra project cannot connect to the server repository.

## Operand

The arguments of an assembly instruction.

## Overlapped Code Model

A block model defining subroutines. A subroutine block is all code accessible from a single entry point and terminates at returns. Code may be shared with other subroutines. Each subroutine is defined to include the overlapping code as part of its body.

## Overlay

A memory block that occupies the same memory address range as some other block.

# P

## Partitioned Code Model

A block model defining subroutines. There is exactly one entry point which may be a call or any other flow instruction. Each instruction belongs to exactly one subroutine (code is not shared).

## PCode

A form of microcode used by Ghidra to model the semantics of machine—or assembly-level instructions.

## PE

Abbreviation for Portable Executable. File format used by Microsoft for storing executable programs.

## PKI Authentication

One of the ways a Ghidra client can use to identify itself to the Ghidra server.

## Plate Comment

Comments that are displayed as a block header above the instruction. Plate comments are automatically surrounded by \*'s.

## Plugin

Software bundles that can optionally be added to Ghidra

to add additional functionality.

## Plugin Dependency

The required presence of additional plugins before a particular plugin can be loaded.

## Plugin Table

A table view of available Plugins that can be added to a tool, as shown in the *Configure* tool dialog.

## Plugin Tree

A graphical view of available Plugins that can be added to a tool, as shown in the *Configure* tool dialog. The Plugins are grouped by functionality and displayed in a tree-like format.

## Popup menu

A [context-sensitive menu](#) that appears when you press the right mouse button.

## Post Comment

Comments that are displayed below the instruction.

## Pre Comment

Comments that are displayed above the instruction.

## Project Archive

Project compressed into a single file for archival or transfer purposes.

## Program

Ghidra's representation of an executable binary, its analysis and annotations.

## Program Diff

Abbreviation for **Program Difference**. It is the process of comparing and contrasting programs in order to determine their similarities and differences.

## Primary Label

The most important label at a location. It is the label that will appear by default in all references to that location.

## Processor Language

The assembly language for a processor.

## Program Context

The set of register values at any location in a program.

## Program Tree

The Ghidra GUI module that allows the user to organize the memory of a program hierarchically.

## Project

A collection of files (programs, etc) and user configuration information.

## Project Repository

A directory on a server that is used by Ghidra's Multi-User module to store shared programs.

## Property

A storage mechanism used by plugins to store information in a program at specific addresses.

# R

## Read-Only Project

A project that can be opened for viewing but cannot be changed (i.e. someone else owns this project).

## Redo

The process of repeating the last change that was "undone".

## Reference

A link from the mnemonic or operand of an instruction to a destination. The destination is an address, stack variable, or external address in another program.

## Register

A special-purpose storage location in a processor.

## Regular Expression

A character sequence used to match patterns in strings.

## Relocation Table

Relocations are address locations that need to be updated to reflect where the program is loaded into memory.

## Running Tools

An area on the [Ghidra Project Window](#) that displays a list of icons which represent [Tools](#) currently in use.

# S

## Scalar

A numeric value in a program.

## Scope

The set of addresses for which a variable is defined.

## Screen Element

The individual listing items that are displayed by the

code browser (i.e. address, mnemonic, operand, comment, etc.)

## Select Limited Flow

A Ghidra process that involves following a [program](#)'s logic but excluding all branches (conditional and unconditional). Select Limited Flow often reveals the high-level algorithm associated with a program. Select Limited Flow is an option in the [Code Browser](#).

## Selection

A set of addresses that have been chosen by the user in order to perform some operation.

## Shared Project

A project that is associated with a Ghidra Server. The files in a shared project are accessible by other users.

## Shared Program

A program that can be modified by multiple users. Shared programs reside in project repositories on a server rather than in local projects on the user's workstation.

## Simple Block Model

See [Basic Block Model](#)

## SLED

Table-based mechanism for specifying the syntax and

semantics for a processor language.

## SLEIGH

Improved version of SLED. Allows language-writers to more accurately represent all features of a language.

## Source Address

The "From" address in a reference "From-To" address pair. An address of an instruction or pointer that refers to another address.

## Stack Reference

A reference from a mnemonic or operand to a stack variable.

## Stack Variable

A parameter or local variable definition on the stack frame defined by a function.

## Static Disassembly

A version of disassembly where jump and call instructions are not followed.

## Status Window

Area at the bottom of a Ghidra tool used to display messages to the user.

## Subroutine Model

A [Block Model](#) which partitions code into address ranges based upon a set of rules defined by a specific model. Subroutine models generally define blocks whose entry point(s) correspond to called locations.

## Symbol

A label that associates a name with an address.

## Symbol Table

A component of [program](#) containing all the label information.

## Symbol Tree

Ghidra GUI module used to display symbols in a tree structure.

# T

## Tabbed Window

A window containing two or more sub-windows that can be selected using tabs.

## Terminator

Any assembly instruction that has no flow (ex: Halt).

## Text Highlighting

The mode in the Code Browser where all the uses of a given word are highlighted in yellow.

## Thunk

Thunks are functions, called by other functions, usually to perform an indirect or external function call.

## Tool Chest

An area on the [Ghidra Project](#) window that displays icons for the configured and saved [Tools](#) which are available to a user.

## Tool

A collection of [Plugins](#) that work together to produce a useful GUI for performing some user level task.

## Tool Tip

A popup description that appears when the mouse is hovered over a GUI item.

## Toolbar (local and global)

An icon bar used to invoke Ghidra functionality.

# U

## Unconditional Call

A call instruction that always executes.

## Unconditional Jump

A jump instruction that always executes.

## Undefined Data

Bytes in the program that have yet to be defined as instructions or data. By default, all bytes in a program begin as Undefined Data.

## Undo

The process of removing the last change made to a program.

## Undocked Window

A Ghidra window that has its own frame and can be positioned and sized independently from other Ghidra windows.

## Unresolved External Reference

An external reference that has not been linked to any program in the project.

## User Access List

The list of users that have access to a particular shared

repository on a Ghidra server. The list contains usernames and permissions.

## User Authentication

The process of verifying the identity of a client user to the server.

# V

## Version Control

The process of maintaining multiple versions of a program as changes are made.

## Version History

A dialog displaying the history of changes made to a program. Any previous version of the program can be selected for viewing from the **View History** dialog.

## Versioned Program

A program that has been placed under version control in a Ghidra project in order to maintain a history of all the changes made to that program.

## Viewed Project

A project that has been open as read-only. Projects that you do not own can only be opened as a Viewed Project.

# W

## Workspace

A virtual Ghidra desktop for a set of [running tools](#).

## X

### XREF

Abbreviation for cross reference. CodeBrowser's display of [Source Addresses](#).

# *Ghidra: NSA Reverse Engineering Software*

Ghidra is a software reverse engineering (SRE) framework developed by NSA's Research Directorate. This framework includes a suite of full-featured, high-end software analysis tools that enable users to analyze compiled code on a variety of platforms including Windows, MacOS, and Linux. Capabilities include disassembly, assembly, decompilation, graphing, and scripting, along with hundreds of other features. Ghidra supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes. Users may also develop their own Ghidra plug-in components and/or scripts using the exposed API.

In support of NSA's Cybersecurity mission, Ghidra was built to solve scaling and teaming problems on complex SRE efforts, and to provide a customizable and extensible SRE research platform. NSA has applied Ghidra SRE capabilities to a variety of problems that involve analyzing malicious code and generating deep insights for NSA analysts who seek a better understanding of potential vulnerabilities in networks and systems.

## *What's New in Ghidra 9.1*

### **The not-so-fine print: Please Read!**

Ghidra 9.1 is fully backward compatible with project data from previous releases. However, programs opened in 9.1 may no longer be accessible by an earlier Ghidra version if the processor model has been updated.

This release includes many new features and capabilities, performance improvements, quite a few bug fixes, and pull-request contributions. Many thanks to all those who have contributed their time, thoughts, and code. The Ghidra user community thanks you too!

NOTE: The Ghidra 9.0 server is compatible with Ghidra 9.0 and 9.1 clients, however the 9.1 server now requires clients to use a TLS secure connection for the initial RMI registry port access. If the Ghidra multi-user server is upgraded to 9.1, then all clients must upgrade to 9.1. A 9.1 Ghidra client will fall back to a non-TLS connection when accessing the RMI Registry on a 9.0 server. Note that all other server interaction including authentication were and continue to be performed over a secure TLS connection.

Minor Note: Ghidra-compiled .sla files are not backwards compatible due to the newly added OTHER space for syscalls support. In the pre-built ghidra, all .sla files are re-built from scratch. However if you have local processor modules, or are building ghidra from scratch, you may need to do a clean build. You will get an error if an old .sla file is loaded without recompilation of the .slaspec file. Any processor modules with changes are normally recompiled at Ghidra startup so this situation is rare.

## Data Improvements

Bitfields within structures are now supported as a Ghidra data type. Bitfield definitions can come from PDB, DWARF, parsed header files, and can also be created within the structure editor. All Data type archives delivered with Ghidra have been re-parsed to capture bitfield information. In addition, compiler bitfield allocation schemes have been carefully implemented. Full support for bitfield references within the decompiler is planned for a future release.

In support of creating bitfields within structures, a new bitfield editor within the structure editor has been added. The Bitfield Editor includes a visual depiction of the data type byte layout and the associated bits. The BitField Editor simplifies the creation of

## System Calls

Ghidra now supports overriding indirect calls, CALLOTHRE pcode ops, and conditional jumps via new overriding references. These references can be used to achieve correct decompilation of syscall-like instructions. A new script, `ResolveX86orX64LinuxSyscallsScript`, has been provided as part of this initial implementation. Future releases will automatically identify and apply system calls for other operating systems and versions.

To support system calls, the decompiler follows references into OTHER address space overlays. This allows users to create address spaces on the fly without worrying about conflicts with existing spaces. For example, instructions with a unique calling convention can be properly handled by adding a reference to a custom function signature.

## Processor Specification

A new set of tools designed to make processor specifications easier to create, modify, and validate have been added. The tools consist of a context sensitive Sleigh file editor, a pcode validation framework, an external disassembler field, and several scripts to make development easier. The Sleigh editor is a plugin for Eclipse and provides modern editor features such as syntax coloring, hover, navigation, code formatting, validation, reference finding, and error navigation. The test suite emulates the pcode to automatically validate the instructions most commonly used by the compiler for that processor.

## iOS DYLD and Macho Format

DYLD shared cache images, extracted from an iOS image, can now

be imported in their entirety. A DYLD's embedded DYLIB's are split into memory blocks, greatly enhancing follow-on analysis. Internal Macho headers are retained and marked up similarly to ELF and PE files, which includes tracking the origin of the program bytes from the initial import binary.

## Ghidra Server

The Ghidra server now requires the client to use a TLS secure connection for the initial RMI registry port access. Previously, TLS was used for all remote object interactions and data transfers on the two other ports. This change will now ensure that all connections to the Ghidra Server utilize TLS. As noted above a 9.1 clients can connect to a 9.0 or 9.1 server, while clients prior to 9.1 will be unable to connect to a 9.1 server.

The Ghidra server has two additional authentication methods, Active Directory using Kerberos and Pluggable Authentication Modules (PAM) using JAAS. To utilize these new methods you must configure the server.conf file and use either -a1 for windows authentication or -a4 along with -jaas. The JAAS mode will require setup of an additional configuration file (jaas.conf).

## Import

When importing files, the origin of all imported bytes can be tracked back to their offset within the original binary source. This change lays the ground work for exporting back to the original file after modifying the bytes. There are programmer API methods to get the bytes either from the memory block or the underlying original source bytes. To see the original bytes a memory block can be mapped onto the original filebytes. The source of each memory block within the memory map is shown in a new Byte Source column. When hovering on the bytes in the program listing, the origin of the bytes at that address are displayed.

## Decompiler

The decompiler now implements a more detailed analysis of local variables on the stack. This change resolves many problems with disappearing structure initialization and incorrect dead code removal.

The decompiler now generates fewer duplicate assignments. For example, repeated assignment of the same value to a variable in two branches will now appear before either branch is taken.

In addition the decompiler now recognizes more optimization patterns used by compilers for signed division, resulting in simplified decompilation.

AARCH64-based binary decompilation will be cleaner due to better handling of zero extensions into larger registers. This improves data flow analysis and primarily affects functions using floating point Neon instructions.

Renaming a parameter in the decompiler will no longer commit the data types of all parameters, allowing data types to continue to "float" without getting locked into a potentially incorrect initial data type. In addition, the cumbersome warning dialog for renaming and retying has been removed, improving your RE workflow.

## Languages

There are many new processor specifications including SuperH4, MCS-96, HCS12X/XGATE, HCS08, and user-contributed specifications for MCS-48, SuperH1/2a, and Tricore.

The 16-bit x86 processor specification has been re-worked to include protected mode addressing, which the NE loader now uses by default. Handling of segmented or paged memory has been updated to use a newer scheme, hiding its complications from decompilation

results. The implementation handles the HCS12X paging scheme as well.

Many improvements and bug fixes have been made to existing processor specifications: ARM, AARCH64, PIC, 68K, MIPS, PPC, JVM, Sparc, AVR8, 8051, 6502, and others.

## Bug Fixes and Enhancements

Numerous other bug fixes and improvements are fully listed in the [ChangeHistory](#) file.

# *What's New in Ghidra 9.0*

## Ghidra Released to the Public!

In case you missed it, in March 2019, a public version of Ghidra was released for the first time. Soon after, the full buildable source was made available as an open source project on the NSA github page. The response from the Ghidra Open Source community has been overwhelmingly positive. We welcome contributions from github including bug fixes, requests, scripts, processor modules, and plugins.

## Bug Fixes and Enhancements

Bug fixes and improvements for 9.0.x are listed in the [Change History](#) file.

# *Ghidra Tips of the Day*

- Ghidra provides context sensitive help on menu items, dialogs, buttons, and tool windows. To access the help, press F1 or Help on any menu item or dialog. If specific help is not available for an item, this page will be displayed.
- In sortable tables you can sort on multiple columns by clicking on a column and then Control-clicking additional columns.
- You can add and remove table columns as desired by right-clicking on a table header.
- You can change the look and feel of your tools by using the "Edit→Options" command.
- You can UNDO and REDO using the curved arrows on the menu bar.
- You can have more than one program open in the same tool. Click on the tabs to switch between them.
- You can bring a new program into Ghidra by selecting File→Import File from either the front end manager or any tool.
- You can import programs by dragging them onto the Ghidra Front End Manager.
- You can assign keybindings to actions, highlight an action and click the F4 key.
- The Program Tree is an organizational view of the program that is initialized with the same organization as the memory map. Changes to the Program Tree only change your view of the program, not the Memory Map.
- The Program Tree allows a program to be organized into a hierarchy of folders and fragments. You can have multiple Program Trees.

- You can add memory overlays to your program using the Memory Map dialog.
- Did you know that you can create and edit structures from the Decompiler?
- Did you know that you can apply the local variables, parameters, and return values that the Decompiler figures out? Just right click on them and choose Apply Locals or Apply Params/Return to get them all for a particular function. You can also run the Decompiler Parameter ID analysis option to apply the Decompiler parameters when the program is initially being analyzed or choose Analysis->One Shot->Decompiler Parameter ID to do it after analysis.
- Did you know that you can run various analyzers separately and after the initial analysis run? See Analysis->One Shot for a list of them.
- Did you know that you can right-click on the marker margin to learn the meaning of each color.
- The largest diamond that was ever found was 3106.75 carats.
- You can edit program bytes using the byte viewer provided those bytes are not disassembled.
- You can add URL's to your comments to link to other documents. Hit F1 while making a comment to learn how.
- You can embed links to other program locations in your comments.
- A cubic yard of air weighs about 2 pounds.
- You can create structures, unions, and enums using the Ghidra data type manager.
- Use enums (not equates) to get constant names to appear in the decompiler. Enums are also listed in the Equates menu if you want to make one an equate in the Listing.

- To edit a row in the datatype (eg structure) editor, double-click on the "DataType" column.
- Ghidra can extract datatype and function signatures from C header files using the CParser.
- You can drag datatypes from the datatype manager and drop it into the browser to apply at address.
- You can apply multiple copies of a data type by making a selection and then dragging that data type from the Data Type Manager onto the selection.
- You can find view information about your current program by selecting "Help → About {program name}...".
- You can edit program information and analysis options by selecting "Edit→Options for {program name}".
- You can compare any two Ghidra programs (or Ghidra versions of a program) using the "Open Diff View" action from the Listing's toolbar.
- Did you know that Ghidra does version tracking? It includes data version tracking as well as function version tracking. It also has numerous algorithms for finding matches. Use the "Footprint" tool to get started.
- Did you know that you don't have to remember a whole label to navigate to it? Simply type 'g' to bring up the Goto dialog and type in a partial label then a \*. If there are more than one matches it will bring up a navigable list of matches.
- New processor languages can be added to Ghidra using the Sleigh language syntax and compiler.
- You can bring up an online processor manual (for most processors) by right mousing on an instruction and choosing Processor Manual.
- You can have snapshot (disconnected) views of the Listing, Byte

Viewer, and Decompiler. Click the camera icon to create a snapshot.

- Windows within a Ghidra tool can be moved, stacked, resized, and undocked to suit your layout preferences.
- Did you know there is a Call Tree Window that shows calls to and from a given function? Click on the green arrow in the icon bar or choose References→Show Call Trees.
- A jiffy is an actual unit of time for 1/100th of a second. Thus the saying, I will be there in a jiffy.
- Did you know that all searches and selections work on the current selection?
- Did you know that you can restore your last Selection if you accidentally clear it by choosing Select→Restore Selection?
- Did you know you can make a table from a selection? See Select→Create Table from Selection.
- Did you know that you can make a selection from a table? Simply highlight one or more rows in the table and right mouse choose Make Selection.
- Ghidra provides many customizable tool options, see Edit→Tool Options.
- You can add symbol information to your comments that automatically update when your symbols change. Hit F1 while making a comment to learn how.
- Bamboo plants can grow up to 36 inches in a day.
- To change direction of the "Next/Previous Code Unit" buttons, use the UP or DOWN arrow in the toolbar.
- You can use the Byte Viewer to display bytes not just in hex, but also octal, decimal, ascii, etc.
- You can use the Byte Viewer to edit bytes. If you want to edit in

hex use the hex view. If you want to edit in ascii, use the ascii view, etc...

- You can have more than one label at the same location.
- In 1890, there was no sunshine for the whole month of December in Westminster, London.
- You can change the representation of scalars (hex, char, decimal, octal, etc) by using the right mouse Convert command.
- You can clean up those pesky runs of cc's, ff's, 90's, and/or 00's by placing the cursor on the byte value you wish to condense and running the CondenseAllRepeatingBytes script.
- In 1992, the Antarctic Ozone hole was larger than the continent of North America.
- Did you know you can see where a register is initialized in its current scope by clicking on it with the middle mouse button? All instances of the register in the current scope will highlight in bright yellow. The mustard yellow one is where it is initialized in the current scope.
- You can perform a program memory search using a regular expression (regex).
- If a Windows executable contains Icons or Bitmap Resources, they are displayed in the CodeBrowser. Do a Search → Program Text on Labels for "Rsrc\_Icon\*" and "Rsrc\_Bitmap\*" to find them.
- The average temperature on Earth is 15 degrees celsius.
- If you hover on a reference in the XREF or operand fields, a popup with the reference code or data will appear.
- If you hover on a data type in the CodeBrowser or Data Type Manager, a popup with the data type definition will appear.
- You can add your own references just about anywhere and you can have more than one of them on an item. See the right mouse

References options.

- You can write Ghidra scripts using Java or Python.
- You can open Ghidra scripts in Eclipse from the Script Manager.  
Install the GhidraDev plugin for Eclipse to get started!
- Double-clicking on addresses and labels in the console will navigate to them.
- Double-clicking on the "function" area of the tool status bar will navigate to the function signature.
- Selection by flow can be configured to follow computed and conditional calls and jumps.
- You can reconfigure the browser display by adding / removing / moving / resizing fields. (Be sure to save your tool!)
- Did you know that you can run Ghidra from the command line without invoking the user interface? (See analyzeHeadlessREADME.html in the {Install Dir}/support folder.)
- This is the last tip. You can turn them off now.

# Appendix

<a href="#"><u>Block Models</u></a>	Details of partitioning code into address ranges
<a href="#"><u>Language</u></a>	Processor language modules

## Block Model

A **Block Model** partitions the code into address ranges based on some set of rules. The most obvious partitioning is by subroutine. There are four Subroutine Models (i.e., Call Block Models). Each subroutine model defines a subroutine slightly differently. The primary differences are based on number of entry points and whether overlapping code between subroutines is allowed. It is important to note that all of these subroutine models will all produce the same result for an M-Model subroutine which contains a single entry point.

**Subroutine Block Models:**

Model Name	Model	Entry Point	Overlapping Code?	Entry Point Type
Isolated Entry*	S	1	Yes	only call
Multiple Entry	M	1 or more	—	only call
Overlapped Code*	O	1 or more	Yes	only call
Partitioned Code*	P	1	No	any

- **IsolatedEntryModel\*** –A subroutine must have only one entry point but may share code with another subroutine. The subroutine body will stop if another called or source entry point is encountered.
- **MultipleEntryModel** –A subroutine may have multiple entry points and may not overlap code from other subroutines.
- **OverlappedCodeModel\*** –A subroutine is all code accessible from a single entry point and terminates at returns. Code may be shared with other subroutines. Each subroutine is defined to include the overlapping code as part of its body.
- **PartitionedCodeModel\*** –There is exactly one entry point which may have any type of source flow. Each instruction belongs to exactly one subroutine (code is not shared).



The default subroutine model for the tool can be specified from **Edit**→**Tool Options** dialog on **Tool** panel. The default subroutine model is generally used by those plugins and actions which do not provide a subroutine model choice (e.g., subroutine selection, call graph, symbol table, etc.).

There is a more primitive **Block Model** called a **Basic** (or Simple) Block Model. Such blocks generally consist of small runs of instructions partitioned based on change in instruction flow. Jump and Branch instructions will cause the execution flow to change, creating a new block. Arithmetic and store/load instructions do not break a Basic Block because they do not change the execution flow. A label will also end one block and begin another.

In the example below each color change represents a different basic block.

```

004074c6 8b 4b      MOV    ECX,dword ptr [EBX + 0x10]
                      10
004074c9 89 01      MOV    dword ptr [ECX],EAX
004074cb 8b 43      MOV    EAX,dword ptr [EBX + 0x10]
                      10
004074ce 83 38      CMP    dword ptr [EAX],-0x1
                      ff
004074d1 74 e5      JZ     LAB_004074b8

LAB_004074d3
004074d3 89 1d      MOV    dword ptr [DAT_0040fbb4],EBX
                      b4 fb
                      40 00
004074d9 8b 43      MOV    EAX,dword ptr [EBX + 0x10]
                      10
004074dc 8b 10      MOV    EDX,dword ptr [EAX]
004074de 83 fa      CMP    EDX,-0x1
                      ff
004074e1 89 55      MOV    dword ptr [EBP + local_8],EDX
                      fc
004074e4 74 14      JZ     LAB_004074fa
004074e6 8b 8c      MOV    ECX,dword ptr [0xc4 + EAX + EDX*...
                      90 c4
                      00 00 ...
004074ed 8b 7c      MOV    EDI,dword ptr [EAX + EDX*0x4 + 0...
                      90 44
004074f1 23 4d      AND    ECX,dword ptr [EBP + local_c]
                      f8
004074f4 23 fe      AND    EDI,ESI
004074f6 0b cf      OR     ECX,EDI
004074f8 75 29      JNZ   LAB_00407523

```

Related Topics:

- [CreateFunction](#)
- [SelectSubroutines](#)

## Processor Languages

All processors have an associated language that defines the mapping between user readable assembly language instructions (e.g. MOV, ADD, etc.) and their corresponding byte values. In order to disassemble a binary image for a specific processor, Ghidra requires a *language module* for that processor. A language module is the software that implements the language translation. Ghidra has a set of language modules for the most commonly-used processor languages. New languages can be added to Ghidra by writing new language modules.

Ghidra uses a processor modeling language called *Sleigh* to define the binary parsing and instruction semantics associated with each language. The semantic information allows for more advanced analysis and enables features such as the decompiler.

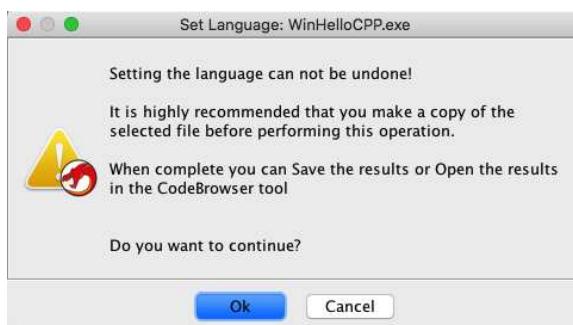
The list of available languages can be found in the [Ghidra Release Notes](#).

### Setting a Program's Language

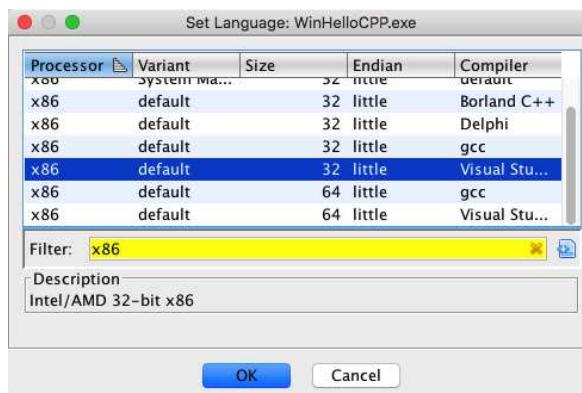
All programs are initially assigned a Processor Language when they are first imported into a project. The processor language can be changed, but only to a closely related language. Any stored register values will be transferred if there is a register with the same name in the new language. If a matching register can't be found those values will be removed.

 If your program has been added to version control in a [shared project](#), you must first have an [exclusive check out](#) on the file before you can set the language. In addition, the program file may not be open in a tool when changing the assigned processor language.

To change the processor language, right-click on the program file within the *Project Window* data tree, and select **Set Language...** from the popup menu. Since setting the language is such a major change, the following warning will appear.



Alternatively, if your file is versioned, you should check-in any recent changes prior to performing this operation. If you press the "OK" button, the Select Language dialog will be displayed:



Select the language from the list and press "OK" to change the processor language for the current program. If the selected language is sufficiently similar to the existing language, the change will be made. Otherwise, an error dialog will appear and the change will not be allowed. *By default, a filter will automatically be applied that displays the most compatible languages. It is recommended to only use one of these languages.*

 Once the operation completes successfully the only way to revert to the previous language (aside from attempting another Set Language) is to undo your checkout if it is versioned. Otherwise, you must rely on a backup copy which you hopefully made prior to the operation.

 Set Language will fail if any old address space can not be mapped to the same size or larger address spaces within the new language. This allows migration to larger processor implementations (e.g., 32-bit to 64-bit), but not the reverse.

### PCode

The semantic information provided by the SLED and SLEIGH-based languages is called PCode (a form of generic microcode). Each assembly language instruction can be broken down into one or more PCode instructions. The more advanced automatic analysis features in Ghidra require PCode in order to operate.

To see PCode, add the PCode field in the Instruction/Data tab of the [Browser Field](#). The figure below shows a CodeBrowser with the PCode field added.

Listing: WinHelloCPP.exe

WinHelloCPP.exe [Read-Only] X

Address	Break	Plate	Function	Variable	Instruction/Data	Open Data	Array
					Register Transition		
					Pre-Comment		
					Label	XRef Header	XRef
+ Address	Bytes	P...	Mnemonic		Operands		EOL Comm
					PCode		
						PCode	
					Post-Comment		
					Space		
undefined	Stack[0x24]:1	param_9					
undefined	Stack[0x28]:1	param_10					
undefined4	Stack[0x4]:4	param_11					
			FUN_00401000			XREF[1]: 00401002(R)	
00401000 8b c1	MOV	EAX,param_1				XREF[1]: FUN_00401130:00401202()	
00401002 8b 4c 24 04	MOV	param_1,dword ptr [ESP + param_11]					
		\$U4a0:4 = INT_ADD 4;4, ESP					
		\$U16f0:4 = LOAD ram(\$U4a0)					
		ECX = COPY \$U16f0					
00401006 89 08	MOV	dword ptr [EAX],param_1					
		\$U16f0:4 = COPY ECX					
		STORE ram(EAX), \$U16f0					

Related Topics:

- [Analysis](#)
- [Disassembly](#)