# VENUS Programming Practices

Shareable Hamilton PDF

## Contents

# 1 Revision History

| Document Version | Date | Revised By | Revision Notes |
|---|---|---|---|
| Rev 0.01 | 2017-10-19 | Eric Sindelar | Document Creation. Used ASW Programming Guideline and Good Programming Practice Hamilton EMEA as templates. |
| Rev 0.02 | 2017-12-20 | Eric Sindelar | Edited document based on feedback from applications managers and specialists |
| Rev 0.03 | 2018-12-19 | Logan Falk | Updated document style and content for external use |
| Rev 0.04 | 2019-02-27 | Eric Sindelar | Renamed document to specify VENUS Software. Updated variable naming format. |
| Rev 0.05 | 2022-07-22 | Eric Sindelar | Updated GUIs and Versioning sections |

The Microlab® STAR™, STAR$^{PLUS}$, STAR$^{LET}$, NIMBUS®, NIMBUS HD, and VANTAGE Liquid Handling System® will be referred to as STAR, NIMBUS, and Microlab VANTAGE for the remainder of this manual.

**For the latest revisions of Hamilton manuals, drivers, and software, contact Hamilton support.**

# 2 Introduction

## 2.1 Purpose

**This document provides best practices for programming methods for Hamilton products.** These guidelines include a general approach to programming methods and descriptions of the tools available to simplify programming. Using these processes and tools should make methods more usable, shareable, and supportable.

## 2.2 Scope

**These guidelines apply to all platforms that use Hamilton's VENUS software.** This includes the ML STAR line's VENUS software, NIMBUS software, and VENUS on VANTAGE.

**Exceptions to these guidelines are understandable, and even expected.** Different applications have different needs, and the methods that automate these applications must accommodate these needs. That said, the guidelines in this document should be followed whenever possible.

**These guidelines are intended for individual programmers working on a specific workflow.** Different guidelines may apply for larger projects where multiple programmers are involved, or in cases when a method must be distributed and shared across multiple sites.

## 2.3 Related Documents

**This document assumes familiarity with VENUS software.** Refer to the appropriate Programmer's Manual for details about programming for a specific instrument.

**For basic information on pipetting and automated liquid handling, refer to the Liquid Handling Reference Guide.**

**For best practices for programming Assay Ready Workstation (ARW) methods, refer to the ARW Programming Practices.**

# 3 The Process

**This chapter provides a general guideline for each stage of the method programming process.**

## 3.1 Before Programming

**Prior to any programming, it is important to understand what the project needs as much as possible.** Avoid making assumptions which could lead to wasted effort and time.

### 3.1.1 Review the System Configuration

**Review the system configuration and discuss with the pre-sales team or system owner.** Identify any key features, project commitments, and personal training gaps with respect to the product configuration or included devices. Communicate and address these gaps with management and colleagues.

### 3.1.2 Review the Requirements

**Consider each requirement for the workflow.** How will the deck be set up? What is the step-by-step process for completing the method? These questions will drive how the method is programmed. The following factors should also be considered, even if they are not explicitly covered in the project's requirements:

- The required throughput compared to the system's capacity
- Any access restrictions caused by the system configuration that could limit the process or increase the complexity of the method – for example, systems with more than 8 independent channels cannot access all locations on the deck, which is not always captured in the software
- The volumes used, and the minimum corresponding number of tips, reagents, and other consumables
- Assay parameters such as timing and temperature

**Identify any gaps in the training or knowledge required for programming the method** and consult subject matter experts for assistance when necessary.

### 3.1.3 Collect Required Labware

**Collect the input, output, and intermediate labware required for the method.** Use existing labware definitions whenever possible; create new definitions only if necessary. Minimize labware usage whenever possible, even if multiple labware of the same type are available.

#### 3.1.4 Identify Liquid Types

**Make a list of all liquid transfers in the method.** This includes not just the liquid type and volume, but how and under what conditions it will be pipetted. Determine which transfers can use existing liquid classes, and which transfers (or liquid types) will require liquid class development.

**If liquid class development is required, then allocate the appropriate amount of time and tools to complete the task.** Refer to the Liquid Handling Reference Guide for further instruction.

### 3.2 During Programming

**While programming the method, maintaining attention to user level of understanding is critical.** The method should be as user-friendly as possible to ensure its robustness and supportability.

#### 3.2.1 Keep it Simple

**Refrain from adding more than what the workflow requires.** Every additional feature adds complexity and test effort, which increases the scope of the project and its risk of possible error.

#### 3.2.2 Make it Readable

**Make sure the method can be easily read and interpreted by others.** This approach requires consistent use of functions like comments, grouping, variables, and sub-methods. Use clear and concise language so that it is easily understood and meaningful to other users.

#### 3.2.3 Avoid Copy/Paste Programming

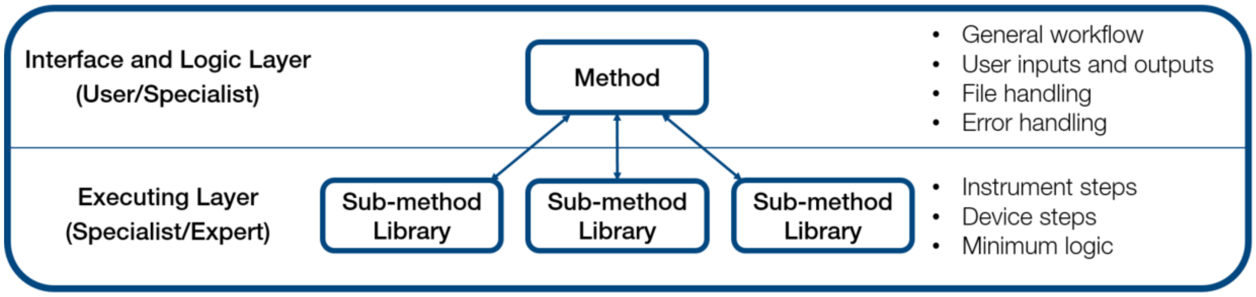
**Copying and pasting repeated steps can lead to lengthy methods with improper settings.** If steps must be repeated, consider using loops or sub-methods instead. Refer to section 4.6 for details on sub-methods.

#### 3.2.4 Keep it Organized

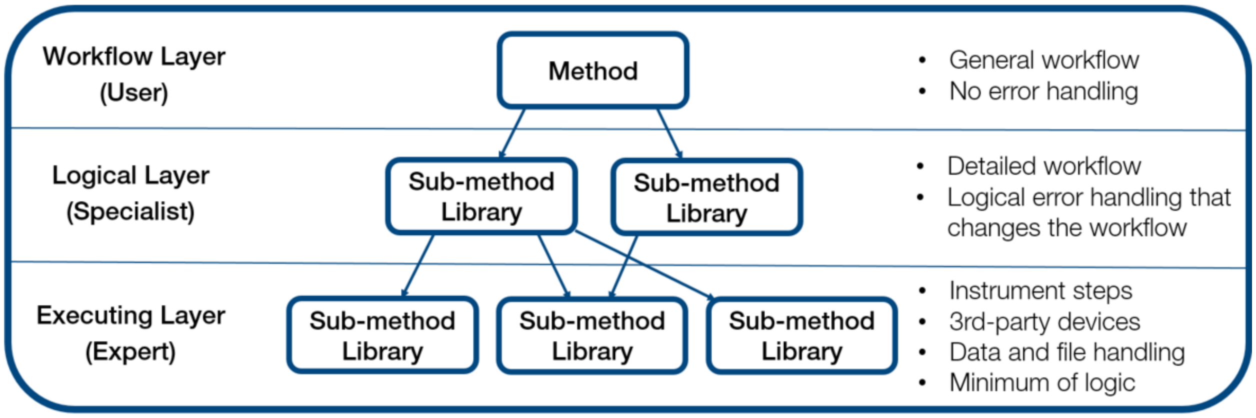
**For most small projects, the use of a "one-layer" structure (a method with sub-methods) is enough to organize a method.** Refer to section 4.6 for more information. Functions like grouping and commenting are still recommended for one-layer methods.

**For more complex projects, adding sub-method libraries or additional layers might be required to keep the method organized.** In rare cases, a three-layer structure may be required: a **workflow layer** with the method that is visible to the user, a **logical layer** with sub-method libraries visible to the specialist, and an **executing layer** of the low-level commands for control that are visible to the expert.

Two-layer structure:



Three-layer structure:



### 3.2.5 Use Standard Libraries

**When libraries are required, begin by using default libraries included with the software.** If additional functionality is needed, then utilize Application Software (ASW) and Hamilton Standard Language (HSL) Extension libraries. If necessary, user-created libraries that have been vetted and posted on knowledge bases may also be used.

**Refrain from using personal or custom libraries.** If any existing libraries are missing features, consult with management and colleagues to include these features in future revisions or new libraries. Test any new libraries thoroughly and create a help file for the library before Implementing into methods or distributing.

**Refrain from developing libraries that deviate from the Vector development environment.** Libraries that use a batch file, Python script, Microsoft Excel VBA, or other external formats to perform data or sequence manipulation should be used only when necessary. Methods that use these libraries must be documented using comments and help files.

**Limit the use of libraries that require a separate installer.** Using such libraries, while sometimes unavoidable, can complicate sharing methods via the default export function, since the additional installers must also be shared and run before using the method.

### 3.2.6 Be Mindful of the Method Settings

**Be mindful of the values used for settings like cLLD and fixed height.** Refer to the Liquid Handling Reference Guide for detailed descriptions of these settings and best practices when using them.

### 3.2.7 Avoid Out-of-Bound Inputs

**Be careful with user inputs for values that could cause errors like volume exceptions or number of samples exceptions.** Use confirmation checks and set ranges for variables whenever possible prevent overage and errors.

### 3.2.8 Maintain Version Control

**All methods should include a version identifier suffix in their file name.** Refer to section 5.1 for more information. Include a comment to document the version history. Refer to section 4.1 for more information on using comments.

### 3.2.9 Create Test Methods

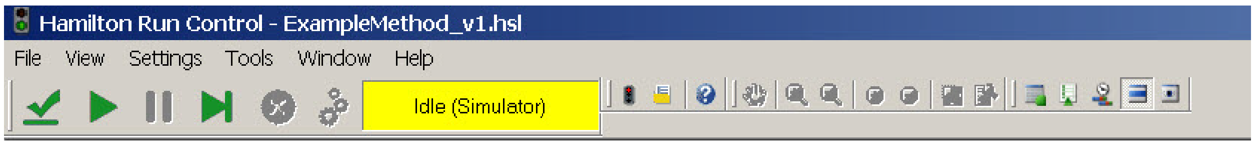
**If the method uses on- or off-deck devices, create simple test methods for each device.** The tests should include device communication, control of its basic functions, and transport of labware to and from the device. These methods are useful for device setup and troubleshooting.

## 3.3 After Programming

**Test the method in incremental steps to avoid costly repeat test runs** and to ensure that method parameters are thoroughly tested.

### 3.3.1 Simulate and Review

**Vector software includes a simulation mode that can catch a large amount of programming errors.** Run the method in simulation and review the generated log file to ensure that the logic of the program is behaving correctly under many conditions. If the method cannot be easily run in simulation mode, avoid extra programming that could add further complexity and detract from the actual runtime mode operation code.



**Exceptions may include devices that do not have a supported simulation mode.** For such devices, the status of the mode can be captured using the GetSimulationMode command in the HSLUtil.hsl library. The device commands can then be skipped in simulation.

**Test against the extreme use cases** such as minimum and maximum number of plates or samples to ensure proper functionality prior to conducting actual test runs. Not all conditions can be tested in a timely fashion, so an effort to mitigate higher-risk scenarios in simulation is helpful.

### 3.3.2 Conduct Water Runs

**Conduct and observe water runs for the method to troubleshoot any improper liquid transfer steps.** Correct these steps in the method and conduct another water run to confirm that the steps are executed properly. Refer to the Liquid Handling Reference Guide for more information on liquid handling methodology.

**Ensure the method can run to completion without generating any errors before proceeding.** If there is some likelihood of error, determine the proper automatic error recovery to allow for continual processing.

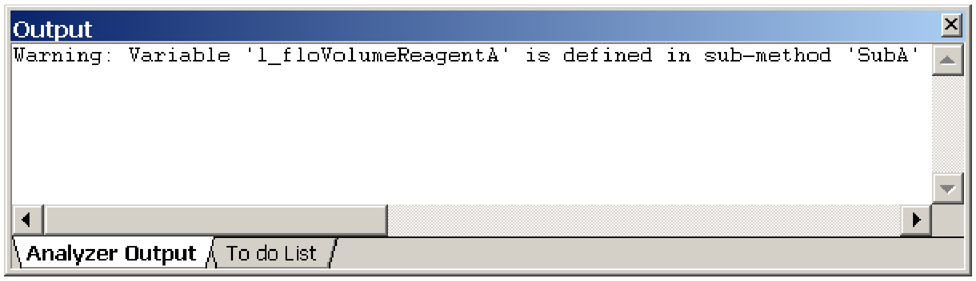
### 3.3.3 Test with Actual Liquids

**Observe test runs with the actual liquids and address any issues**, such as droplet formation or imprecise volumes. Refer to the Liquid Handling Reference Guide for more information on liquid handling methodology. Review the results with any stakeholders and incorporate any feedback.

**Be mindful of tip and consumable usage during testing** in order to conserve consumables.

### 3.3.4 Clean Up the Method

**Using the results from testing, identify any unnecessary parts of the method.** Remove unused or unnecessary labware, variables, and steps from the method to avoid clutter and confusion. If any disabled steps are left in, provide comments to explain why they are included. Remove any warnings in the Output window.



```
Warning: Variable 'l_floVolumeReagentA' is defined in sub-method 'SubA'
```

### 3.3.5 Save and Export

**Make frequent backups of the program and maintain version control while saving and exporting methods.** Refer to sections 5.1 and 5.2 for best practices for saving and exporting.

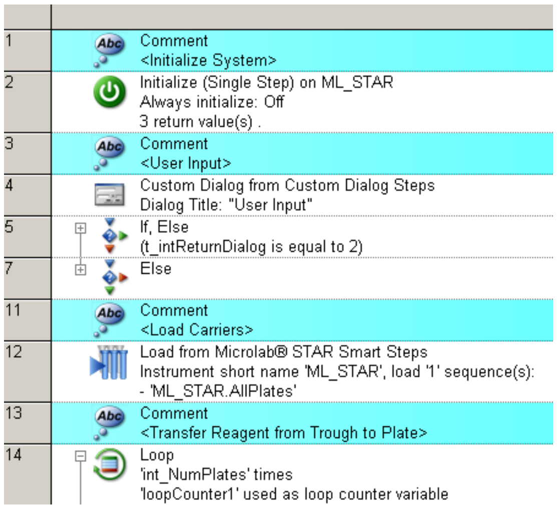
# 4 Use of Specific Functions

**This chapter details tools and techniques for keeping methods organized.**

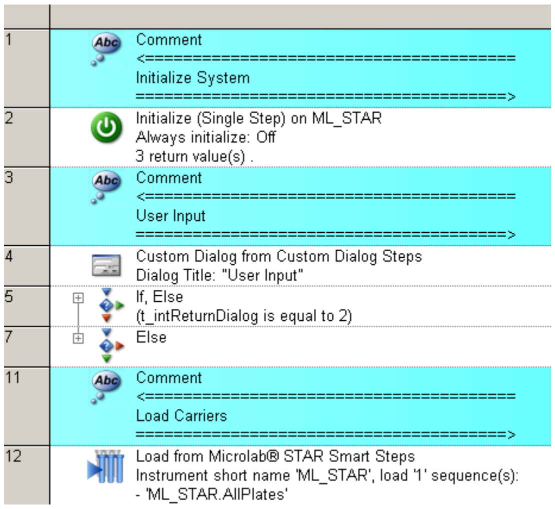
## 4.1 Comments

**Using comments is a basic requirement for providing organization and clarity to a method.** Use comments to separate and explain each section in the method. The function of each section should be clear to a non-programmer who reads the comments.

**Make the comments more visible by customizing the color and adding a “frame” of special characters.** The color helps to visually separate the sections in the method, and the frame is helpful for separating the sections in log files.

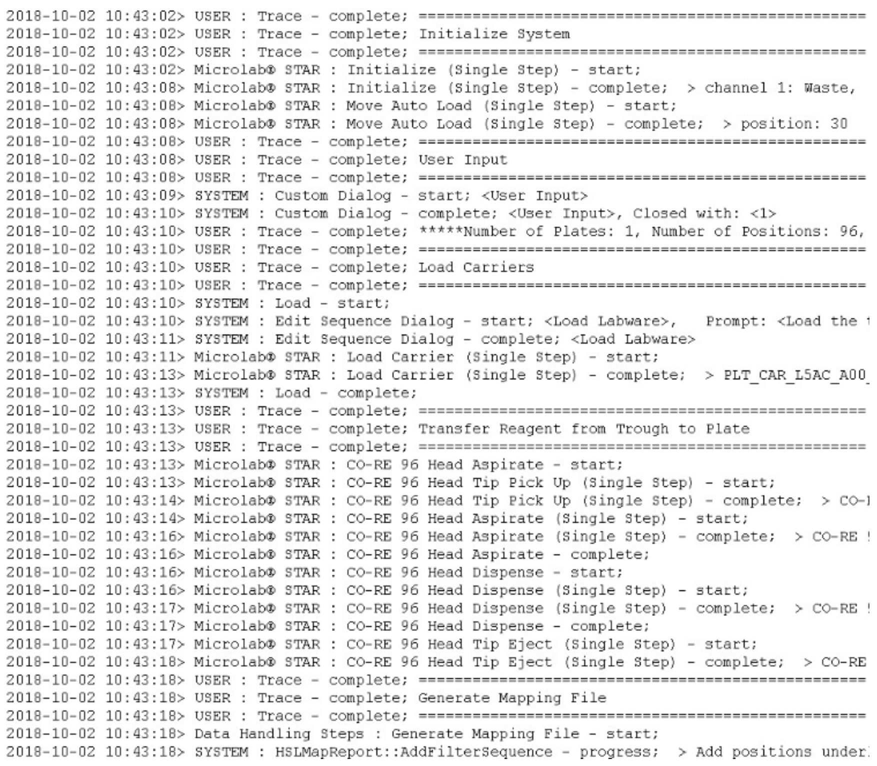


```
Comment
<Initialize System>
Initialize (Single Step) on ML_STAR
Always initialize: Off
3 return value(s) .
Comment
<User Input>
Custom Dialog from Custom Dialog Steps
Dialog Title: "User Input"
If, Else
(t_intReturnDialog is equal to 2)
Else
Comment
<Load Carriers>
Load from Microlab® STAR Smart Steps
Instrument short name 'ML_STAR', load '1' sequence(s):
- 'ML_STAR.AllPlates'
Comment
<Transfer Reagent from Trough to Plate>
Loop
'int_NumPlates' times
'loopCounter1' used as loop counter variable
```



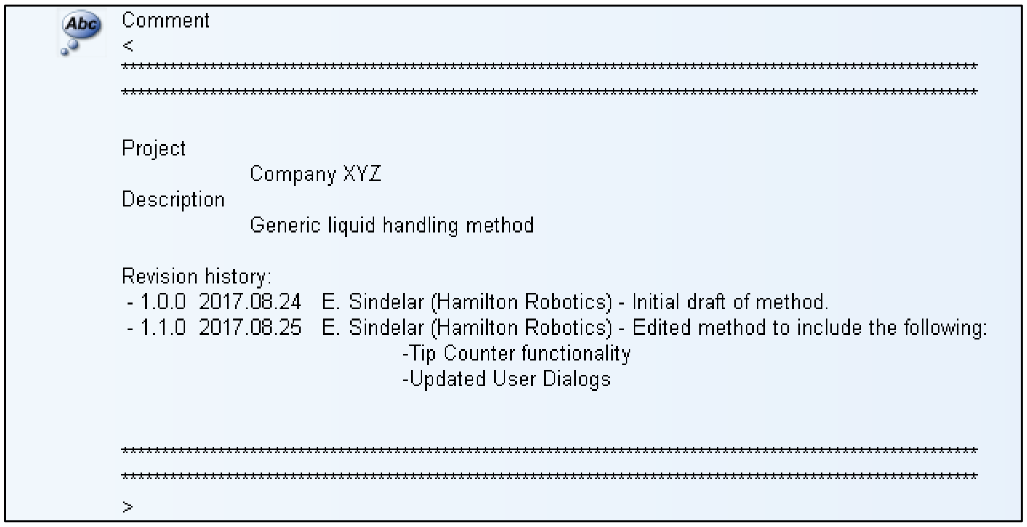
```
Comment
<=======================================
Initialize System
=======================================>
Initialize (Single Step) on ML_STAR
Always initialize: Off
3 return value(s) .
Comment
<=======================================
User Input
=======================================>
Custom Dialog from Custom Dialog Steps
Dialog Title: "User Input"
If, Else
(t_intReturnDialog is equal to 2)
Else
Comment
<=======================================
Load Carriers
=======================================>
Load from Microlab® STAR Smart Steps
Instrument short name 'ML_STAR', load '1' sequence(s):
- 'ML_STAR.AllPlates'
```

**Most comments should be traced to the log file**, especially comments that separate different sections in a method or sub-method. Comments that are used to explain steps to other users do not need to be traced. For example, include untraced comments for math operations, sequence manipulation, and complex file handling.



```
2018-10-02 10:43:02> USER : Trace - complete; ==================================================
2018-10-02 10:43:02> USER : Trace - complete; Initialize System
2018-10-02 10:43:02> USER : Trace - complete; ==================================================
2018-10-02 10:43:02> Microlab® STAR : Initialize (Single Step) - start;
2018-10-02 10:43:08> Microlab® STAR : Initialize (Single Step) - complete;  > channel 1: Waste,
2018-10-02 10:43:08> Microlab® STAR : Move Auto Load (Single Step) - start;
2018-10-02 10:43:08> Microlab® STAR : Move Auto Load (Single Step) - complete;  > position: 30
2018-10-02 10:43:08> USER : Trace - complete; ==================================================
2018-10-02 10:43:08> USER : Trace - complete; User Input
2018-10-02 10:43:08> USER : Trace - complete; ==================================================
2018-10-02 10:43:09> SYSTEM : Custom Dialog - start; <User Input>
2018-10-02 10:43:10> SYSTEM : Custom Dialog - complete; <User Input>, Closed with: <1>
2018-10-02 10:43:10> USER : Trace - complete; *****Number of Plates: 1, Number of Positions: 96,
2018-10-02 10:43:10> USER : Trace - complete; ==================================================
2018-10-02 10:43:10> USER : Trace - complete; Load Carriers
2018-10-02 10:43:10> USER : Trace - complete; ==================================================
2018-10-02 10:43:10> SYSTEM : Load - start;
2018-10-02 10:43:10> SYSTEM : Edit Sequence Dialog - start; <Load Labware>,   Prompt: <Load the
2018-10-02 10:43:11> SYSTEM : Edit Sequence Dialog - complete; <Load Labware>
2018-10-02 10:43:11> Microlab® STAR : Load Carrier (Single Step) - start;
2018-10-02 10:43:13> Microlab® STAR : Load Carrier (Single Step) - complete;  > PLT_CAR_L5AC_A00_
2018-10-02 10:43:13> SYSTEM : Load - complete;
2018-10-02 10:43:13> USER : Trace - complete; ==================================================
2018-10-02 10:43:13> USER : Trace - complete; Transfer Reagent from Trough to Plate
2018-10-02 10:43:13> USER : Trace - complete; ==================================================
2018-10-02 10:43:13> Microlab® STAR : CO-RE 96 Head Aspirate - start;
2018-10-02 10:43:13> Microlab® STAR : CO-RE 96 Head Tip Pick Up (Single Step) - start;
2018-10-02 10:43:14> Microlab® STAR : CO-RE 96 Head Tip Pick Up (Single Step) - complete;  > CO-
2018-10-02 10:43:14> Microlab® STAR : CO-RE 96 Head Aspirate (Single Step) - start;
2018-10-02 10:43:16> Microlab® STAR : CO-RE 96 Head Aspirate (Single Step) - complete;  > CO-RE
2018-10-02 10:43:16> Microlab® STAR : CO-RE 96 Head Aspirate - complete;
2018-10-02 10:43:16> Microlab® STAR : CO-RE 96 Head Dispense - start;
2018-10-02 10:43:16> Microlab® STAR : CO-RE 96 Head Dispense (Single Step) - start;
2018-10-02 10:43:17> Microlab® STAR : CO-RE 96 Head Dispense (Single Step) - complete;  > CO-RE
2018-10-02 10:43:17> Microlab® STAR : CO-RE 96 Head Dispense - complete;
2018-10-02 10:43:17> Microlab® STAR : CO-RE 96 Head Tip Eject (Single Step) - start;
2018-10-02 10:43:18> Microlab® STAR : CO-RE 96 Head Tip Eject (Single Step) - complete;  > CO-RE
2018-10-02 10:43:18> USER : Trace - complete; ==================================================
2018-10-02 10:43:18> USER : Trace - complete; Generate Mapping File
2018-10-02 10:43:18> USER : Trace - complete; ==================================================
2018-10-02 10:43:18> Data Handling Steps : Generate Mapping File - start;
2018-10-02 10:43:18> SYSTEM : HSLMapReport::AddFilterSequence - progress;  > Add positions under
```

**Use a comment to track all version changes for a method or sub-method library that includes a short description and its version history.**



```
Comment
<
*****************************************************************************************************************
*****************************************************************************************************************

Project
                 Company XYZ
Description
                 Generic liquid handling method

Revision history:
 - 1.0.0  2017.08.24   E. Sindelar (Hamilton Robotics) - Initial draft of method.
 - 1.1.0  2017.08.25   E. Sindelar (Hamilton Robotics) - Edited method to include the following:
                                  -Tip Counter functionality
                                  -Updated User Dialogs


*****************************************************************************************************************
*****************************************************************************************************************
>
```

**Write all comments in English.** Additional comments in the local language can be added, but for easy international distribution of methods, write them in English as well. GUIs and user dialogs should remain in the language that is easiest for the routine user.

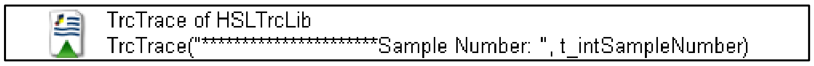
## 4.2 Tracing

**Using trace commands is a basic requirement for easier support and troubleshooting.** Each instrument automatically traces some instrument functions, but many programming tasks are not traced by default. To ensure a complete record of the program’s execution, trace functions must be manually added.

**In general, always trace the following**:

- Variables
- Comments
- Sequences made or sorted during runtime
- Arrays
- Sub-method function name

**Make the traces more visible by tracing multiple variables in one command and by inserting special characters.**



```
TrcTrace of HSLTrcLib
TrcTrace("************************Sample Number: ", t_intSampleNumber)
```

**Use different trace modes when necessary.** For example, certain trace functions provide a debugging mode which produces a lot of entries in the log file. This level of detail is helpful for troubleshooting, but not necessary for routine use.

**There are multiple trace libraries, some of which are included with the software by default**. It is recommended to use the ASW TraceLevel Library as it offers the most robust functionality.

## 4.3 Variables

**Variables are necessary for making a dynamic and user-friendly program.** However, variables should only be used for values that are repeated or that can change. Otherwise, fixed values should be used.

### 4.3.1 Naming Variables

**The name of the variable should describe what it contains, along with its scope and type.** For consistency, all variable names should follow a set format:

[scope]_[type]VariableName

The scope of a variable indicates its visibility and usage in or across methods and sub-methods. There are three main scope types:

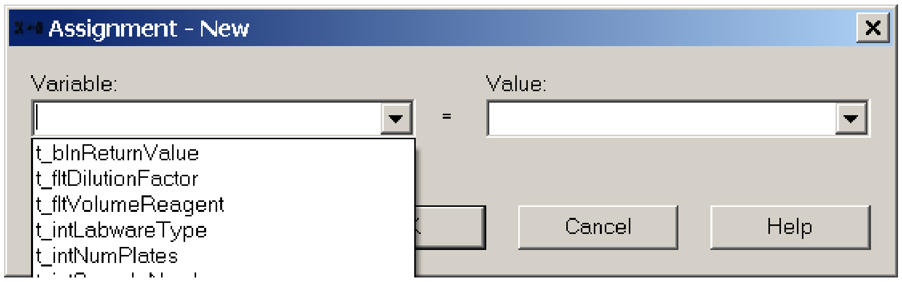
- Task-Local: The variable is visible in the main method and its sub-methods.
- Local: The variable is visible only within the sub-method where it was defined.
- Global: The variable is visible across methods and is used by the Scheduler software and other programs.

**A scope prefix is used to readily identify the scope of the variable.** Local scope variables that are used as input and output parameters for functions have their own specific prefix.

| Scope | Prefix |
|---|---|
| Task-local | t_ |
| Local | l_ |
| Global | g<Namespace>_ |
| Function input parameter | i_ |
| Function output parameter | o_ |

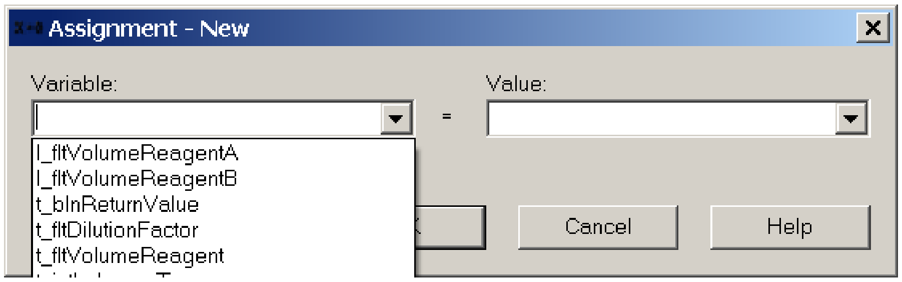
**This convention allows for better organization and distinction of the variables when developing in different parts of the method.** While previous revisions of the guidelines identified Task-local variables with an underscore as a prefix and local variables without any prefix, the updated naming convention makes the scope more explicit for all variables. The previous approach may still be used as deemed necessary.

Main method:



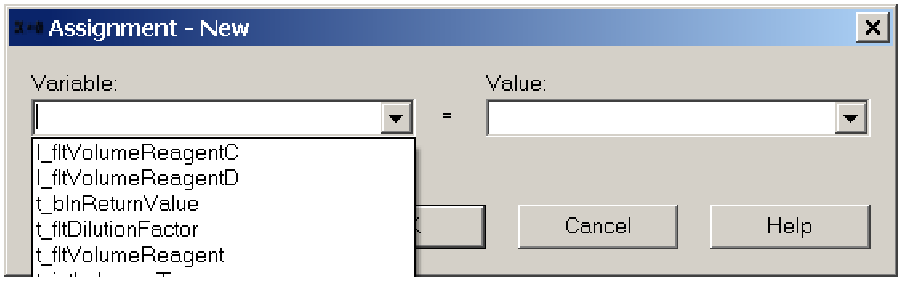
```
t_blnReturnValue
t_fltDilutionFactor
t_fltVolumeReagent
t_intLabwareType
t_intNumPlates
```

Sub-method 1:



```
l_fltVolumeReagentA
l_fltVolumeReagentB
t_blnReturnValue
t_fltDilutionFactor
t_fltVolumeReagent
```

Sub-method 2:



```
l_fltVolumeReagentC
l_fltVolumeReagentD
t_blnReturnValue
t_fltDilutionFactor
t_fltVolumeReagent
```

**The variable type is also indicated using a prefix.**

| Variable Type | Prefix |
|---|---|
| Integer | int |
| String | str |
| Float | flt |
| Boolean | bln |
| Object | obj |
| Array | arr |
| Pipetting Sequence | seq |
| Transport Sequence | trp |
| File, timer, or device handle | hdl |
| Other variable types | var |

**For the actual name, use a short description in Pascal case** (no spaces or other characters, capitalize each word). In general, use one or two words to keep the names concise.

- t_intProcessedSamples: Task-local, integer
- l_strFilePath; Local, string
- i_fltPipettingVolume: function input, float
- o_arrSampleBarcodes: function output, array
- io_seqTips: function input and output, sequence
- gWorkflow_blnSimulateShaker: global, Boolean
- l_hdlWorklist: local, file handle

**The following conventions are also helpful for consistent, clear variable naming:**

- Use variable names in English
- Refrain from using abbreviations unless they are very common
- Avoid numbers unless they help clarify the content of the variable
- Prioritize readability and clarity over brevity
- Refrain from using all caps except for device names like “ML_STAR”

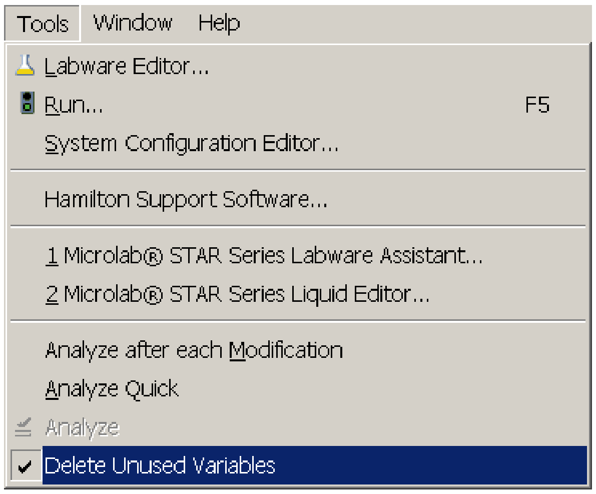
### 4.3.2 Scoping Variables

**The scope of variables should be as small as possible.** Avoid using global variables. Exceptions may include sub-method libraries and scheduler methods.

### 4.3.3 Declaring Variables

**Variables should be defined at the beginning of the method** or in a local sub-method for quick access. Grouping them is also recommended, especially when declaring a long list of variables – refer to section 4.4 for details. Once a variable is declared, refrain from typing it out in all other parameter input fields. Instead, select the variable from the drop-down menu. This approach helps prevent the addition of mistyped variables.

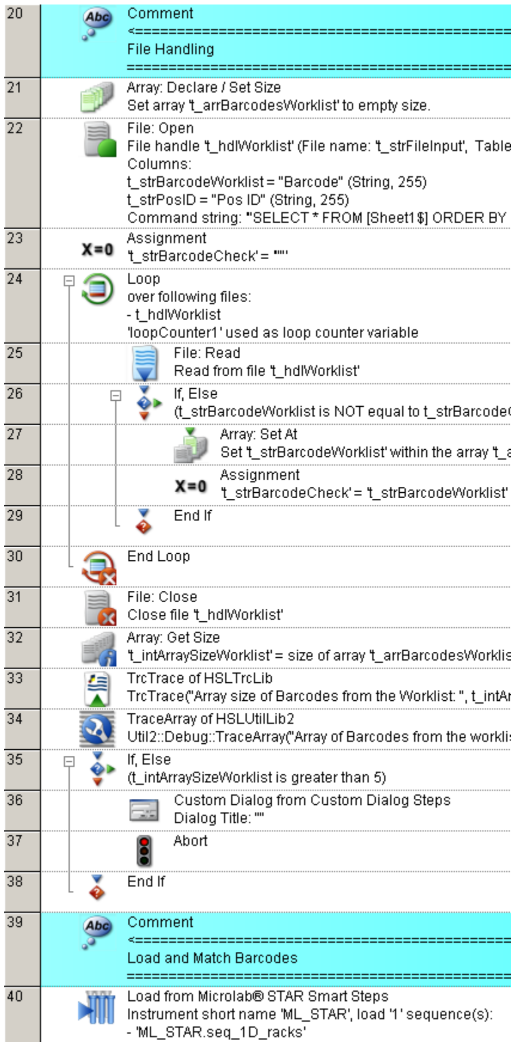
**Do not use the Variable window to set starting values.** This window is not visible by default when starting VENUS, which can cause confusion when it contains variable assignments. Delete any unused variables from the Variable window to avoid confusion while programming. Enabling Delete Unused Variables in the Method Editor also facilitates this function.


Tools
Window
Help
Labware Editor...
Run... F5
System Configuration Editor...
Hamilton Support Software...
1 Microlab® STAR Series Labware Assistant...
2 Microlab® STAR Series Liquid Editor...
Analyze after each Modification
Analyze Quick
Analyze
Delete Unused Variables

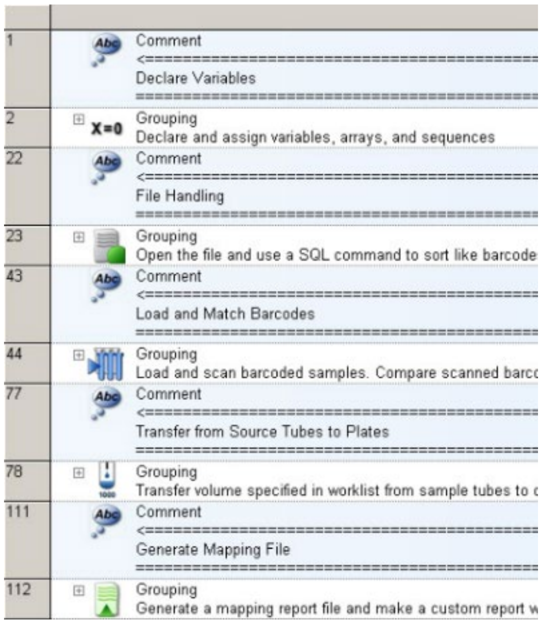
**Trace all variables that are not automatically traced to the log file.** Refer to section 4.2 for details on tracing.

## 4.4 Grouping

**Grouping helps organize methods by collapsing sections of related steps.** Use grouping to hide complex sections or long lists of variables.

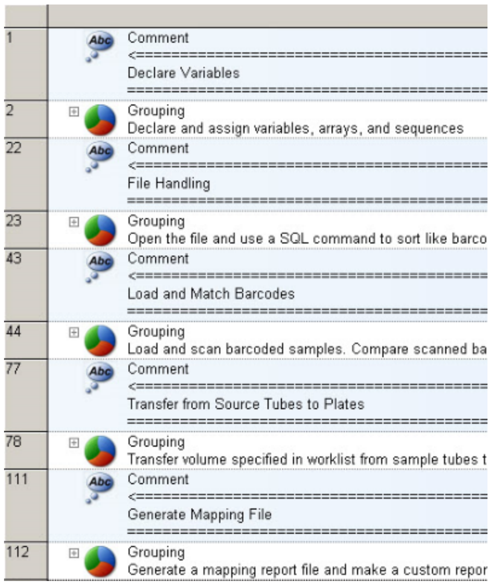


```
Comment
<===============================================
File Handling
================================================
Array: Declare / Set Size
Set array 't_arrBarcodesWorklist' to empty size.
File: Open
File handle 't_hdlWorklist' (File name: 't_strFileInput',  Table
Columns:
t_strBarcodeWorklist = "Barcode" (String, 255)
t_strPosID = "Pos ID" (String, 255)
Command string: "SELECT * FROM [Sheet1$] ORDER BY
Assignment
't_strBarcodeCheck' = ""'
Loop
over following files:
- t_hdlWorklist
'loopCounter1' used as loop counter variable
    File: Read
    Read from file 't_hdlWorklist'
    If, Else
    (t_strBarcodeWorklist is NOT equal to t_strBarcode
        Array: Set At
        Set 't_strBarcodeWorklist' within the array 't_
        Assignment
        't_strBarcodeCheck' = 't_strBarcodeWorklist'
    End If
End Loop
File: Close
Close file 't_hdlWorklist'
Array: Get Size
't_intArraySizeWorklist' = size of array 't_arrBarcodesWorklis
TrcTrace of HSLTrcLib
TrcTrace("Array size of Barcodes from the Worklist: ", t_intA
TraceArray of HSLUtilLib2
Util2::Debug::TraceArray("Array of Barcodes from the workli
If, Else
(t_intArraySizeWorklist is greater than 5)
    Custom Dialog from Custom Dialog Steps
    Dialog Title: ""
    Abort
End If
Comment
<===============================================
Load and Match Barcodes
================================================
Load from Microlab® STAR Smart Steps
Instrument short name 'ML_STAR', load '1' sequence(s):
- 'ML_STAR.seq_1D_racks'
```

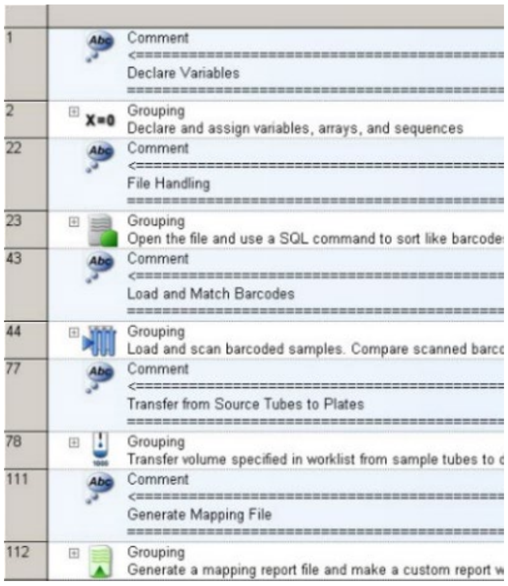


```
Comment
<=============================================
Declare Variables
==============================================
Grouping
Declare and assign variables, arrays, and sequences
Comment
<=============================================
File Handling
==============================================
Grouping
Open the file and use a SQL command to sort like barcode
Comment
<=============================================
Load and Match Barcodes
==============================================
Grouping
Load and scan barcoded samples. Compare scanned barc
Comment
<=============================================
Transfer from Source Tubes to Plates
==============================================
Grouping
Transfer volume specified in worklist from sample tubes to
Comment
<=============================================
Generate Mapping File
==============================================
Grouping
Generate a mapping report file and make a custom report w
```

**Differentiate groups by assigning different icons from those available in the Hamilton\Graphic folder.** For example, use the assignment icon for groups of variables, or the barcode icon for barcode matching logic.



| | |
|---|---|
| 1 | Comment<br><=========================================<br>Declare Variables<br>========================================= |
| 2 | Grouping<br>Declare and assign variables, arrays, and sequences |
| 22 | Comment<br><=========================================<br>File Handling<br>========================================= |
| 23 | Grouping<br>Open the file and use a SQL command to sort like barco |
| 43 | Comment<br><=========================================<br>Load and Match Barcodes<br>========================================= |
| 44 | Grouping<br>Load and scan barcoded samples. Compare scanned ba |
| 77 | Comment<br><=========================================<br>Transfer from Source Tubes to Plates<br>========================================= |
| 78 | Grouping<br>Transfer volume specified in worklist from sample tubes t |
| 111 | Comment<br><=========================================<br>Generate Mapping File<br>========================================= |
| 112 | Grouping<br>Generate a mapping report file and make a custom repor |



| | |
|---|---|
| 1 | Comment<br><=========================================<br>Declare Variables<br>========================================= |
| 2 | Grouping<br>Declare and assign variables, arrays, and sequences |
| 22 | Comment<br><=========================================<br>File Handling<br>========================================= |
| 23 | Grouping<br>Open the file and use a SQL command to sort like barcode |
| 43 | Comment<br><=========================================<br>Load and Match Barcodes<br>========================================= |
| 44 | Grouping<br>Load and scan barcoded samples. Compare scanned barc |
| 77 | Comment<br><=========================================<br>Transfer from Source Tubes to Plates<br>========================================= |
| 78 | Grouping<br>Transfer volume specified in worklist from sample tubes to c |
| 111 | Comment<br><=========================================<br>Generate Mapping File<br>========================================= |
| 112 | Grouping<br>Generate a mapping report file and make a custom report w |

**Keep in mind that grouping is not a substitute for comments.** Use comments in addition to grouping to clearly indicate and trace different sections of the method.

## 4.5 Naming Labware, Sequences, and System Decks

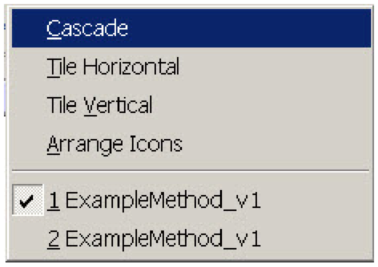
### 4.5.1 Labware and Sequences

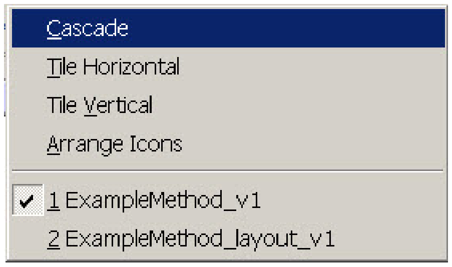
**The default names for labware added to a deck are unclear, and sometimes inaccurate.** Changing these names makes them more identifiable both in the method and in log files, since only the name in the deck layout is traced.

**For example, instead of a default name like "Cos96Rd_0001", a name like "SourcePlate96_01" can be used.** This approach allows for easier changes to labware and clearer default names for the automatically created sequences. For custom sequences, follow the naming conventions in section 4.3.1. Remove unused sequences to avoid clutter and confusion during programming.

### 4.5.2 System Deck

**If the system deck is assigned to only one method, then assign it the same name as the method with a suffix like "_layout"** to differentiate it from the method. If one system deck is designed to accommodate multiple methods, then name it based on the overall process with a suffix like "_layout".

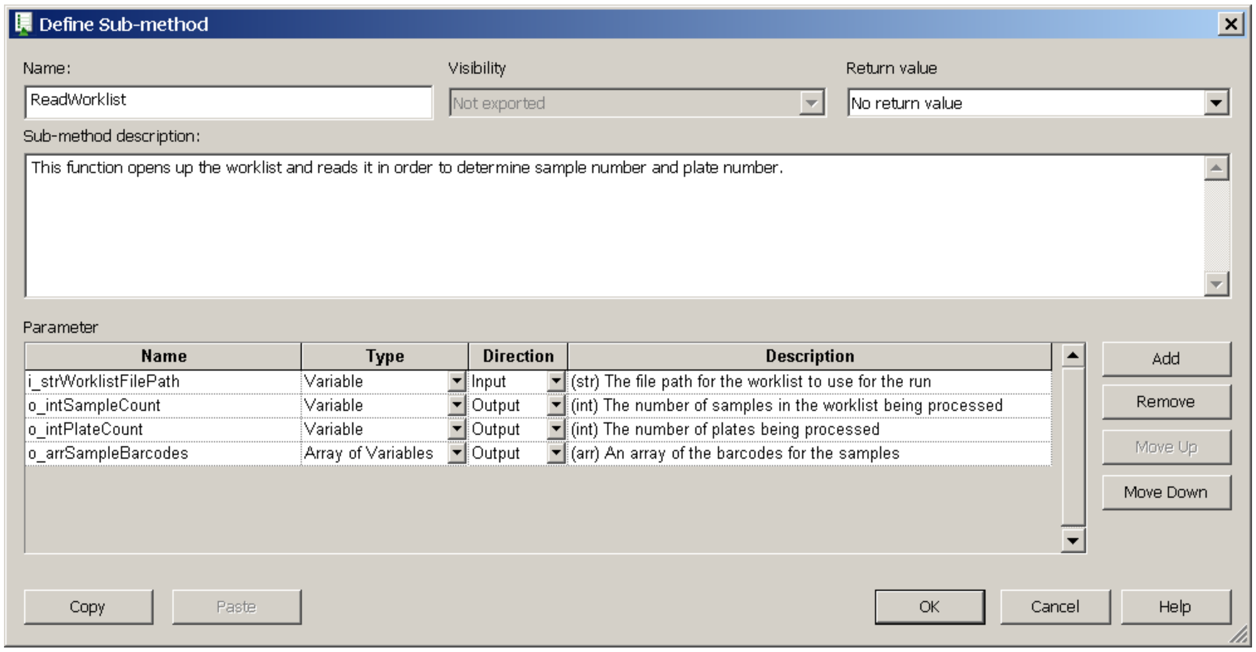




## 4.6 Sub-Methods and Libraries

**One way to organize methods with multiple lengthy transfer steps is to separate sections into sub-methods.** This approach allows for better readability and clarity of the method.

**Refrain from creating sub-methods with lots of parameters.** In general, sub-methods and sub-method library functions should be limited to a maximum of 8 input/output parameters. Sub-method libraries should also be limited to 20 functions. Sub-methods should be annotated with descriptions and include acceptable expected input ranges and output values.


Define Sub-method

Name: ReadWorklist

Visibility: Not exported

Return value: No return value

Sub-method description:

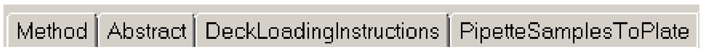
This function opens up the worklist and reads it in order to determine sample number and plate number.

Parameter

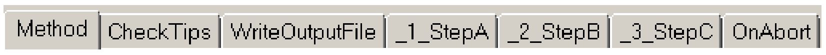
| Name | Type | Direction | Description |
|---|---|---|---|
| i_strWorklistFilePath | Variable | Input | (str) The file path for the worklist to use for the run |
| o_intSampleCount | Variable | Output | (int) The number of samples in the worklist being processed |
| o_intPlateCount | Variable | Output | (int) The number of plates being processed |
| o_arrSampleBarcodes | Array of Variables | Output | (arr) An array of the barcodes for the samples |

Sub-method libraries should include a return value to indicate success/fail. The return could be a Boolean value for success/fail or an error code value if the function is for a device command.

**As with variables, use a short description in Pascal case** to name each sub-method. In general, use no more than four words to keep the name concise.

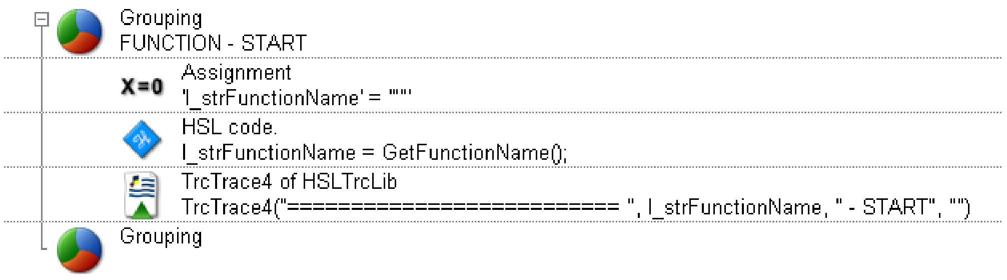

Method | Abstract | DeckLoadingInstructions | PipetteSamplesToPlate

Refrain from using underscores unless needed to interrupt the order that the sub-methods are arranged.


Method | CheckTips | WriteOutputFile | _1_StepA | _2_StepB | _3_StepC | OnAbort

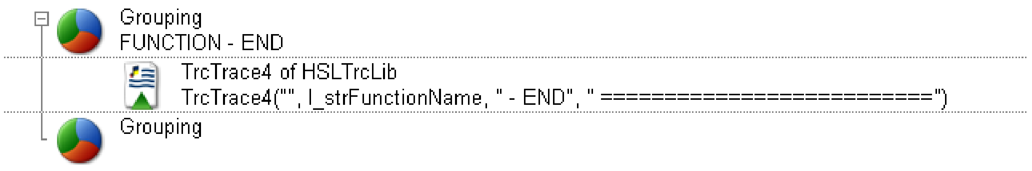
All sub-methods should implement and trace the name of the sub-method and indicate the start and end of the function. See below for example code that can be re-used for any sub-method:

Example code to include at the start of the sub-method:



```
Grouping
FUNCTION - START
    Assignment
    'l_strFunctionName' = ""
    HSL code.
    l_strFunctionName = GetFunctionName();
    TrcTrace4 of HSLTrcLib
    TrcTrace4("========================= ", l_strFunctionName, " - START", "")
Grouping
```

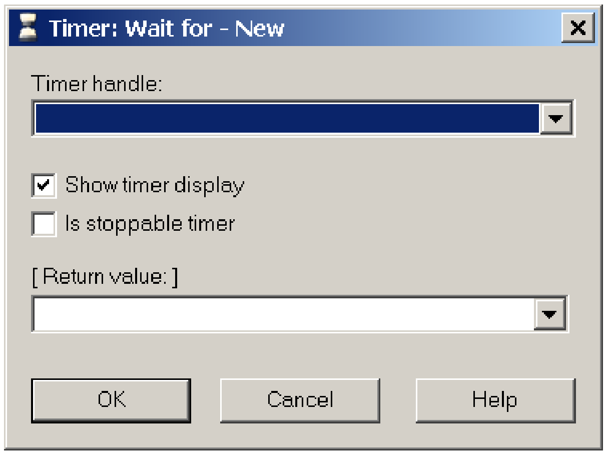
Example code to include at the end of the sub-method:



```
Grouping
FUNCTION - END
    TrcTrace4 of HSLTrcLib
    TrcTrace4("", l_strFunctionName, " - END", " =========================")
Grouping
```

## 4.7 Timers

**When implementing timers in a method during development, avoid enabling the stoppable timer option.** If left enabled, it could allow for the possibility of human error during an actual run. Instead, enable timers with a short value of 1 second when in development or simulation mode. The timer value can be conditionally assigned with a variable. This approach allows for the verification of set and wait for timer functions without having to incur the actual wait time required for the process step.



Timer: Wait for - New

Timer handle:

☑ Show timer display

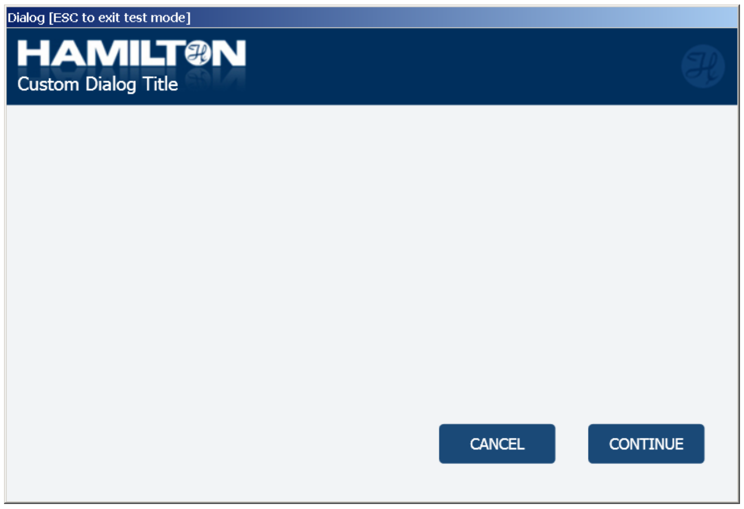
☐ Is stoppable timer

[ Return value: ]

## 4.8 User Dialogs

**Custom dialogs can greatly improve the appearance and usability of the method.** The use of some visual libraries may be required for ease of use and advanced functionality.

**The standard Hamilton template is a useful starting point.** Use the default icons to identify alerts and statuses to the user. Adjust the dialog and font size as necessary for clarity and use the Group Box function to call out inputs and outputs more clearly.

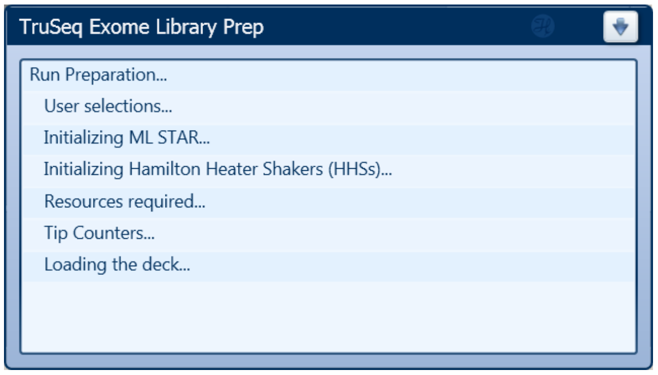


**When the method contains several custom dialogs in succession, include an option to allow the user to go back to previous dialogs.** Allowing a user to do so improves usability, as it allows the user to correct their inputs without having to restart the method.

**ASW Standard Dialogs may be used as an alternative under certain circumstances:**

- The desired functionality is not available through normal custom dialogs
- The method requires a high level of standardization (such as when the method will be widely distributed)

**The ASW Status Dialog can be helpful for lengthy methods with many steps and sections.** The dialog provides runtime updates immediately to the user. Including this function requires more effort, but it can help with the adoption and usage of the method.



TruSeq Exome Library Prep

Run Preparation...
  User selections...
  Initializing ML STAR...
  Initializing Hamilton Heater Shakers (HHSs)...
  Resources required...
  Tip Counters...
  Loading the deck...

## 4.9 File Handling

**Make a copy of any input files and work from the copy instead of the original file.** Working from the copy can prevent errors that could occur if the user manipulates the original file during runtime.

**Make sure that folders for any output files have suitable access rights.** The Hamilton\Logfiles folder is a good default, since the VENUS software must have read/write access to that folder to function properly. If the output file must go to a network or other location, save the file to the Logfiles folder first, then make the copy to the destination folder. This approach helps ensure the original copy will always be created and recovered in case the copy command encounters a conflict.

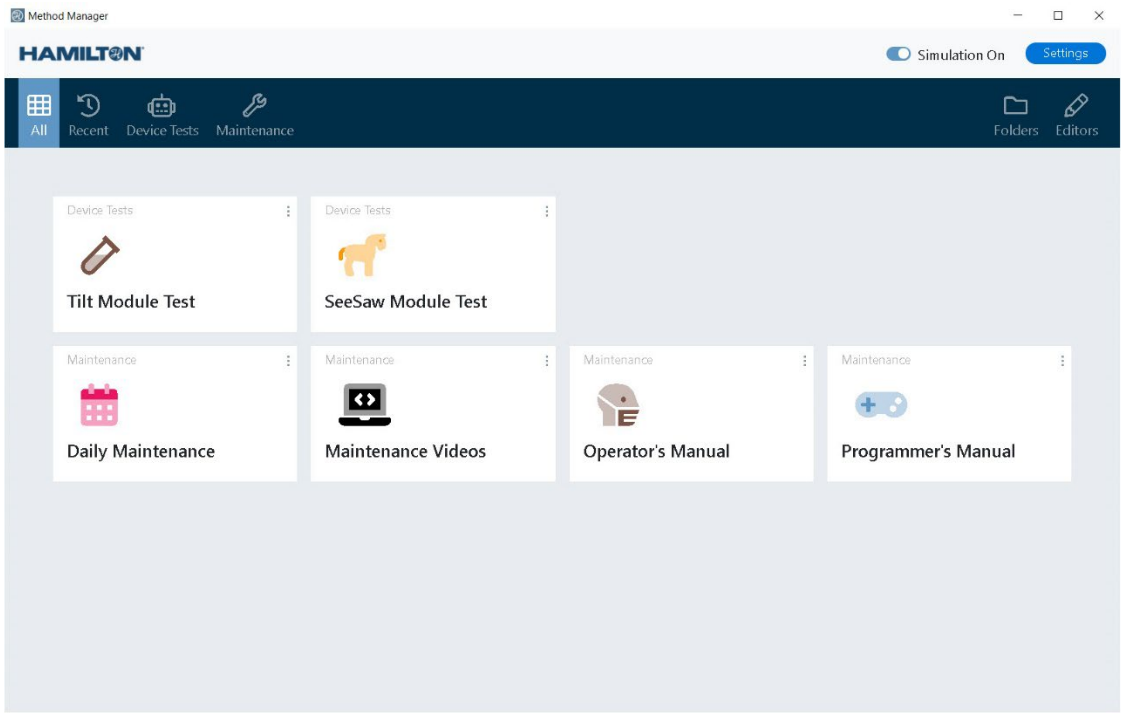
**When using default File handling commands in VENUS, incorporate the Error Handling by the User function to capture any error that results in a type cancel.** The use of this function allows the programmer to provide a more informative and clear user output upon error and the ability to try again.

## 4.10 GUIs and Visual Libraries

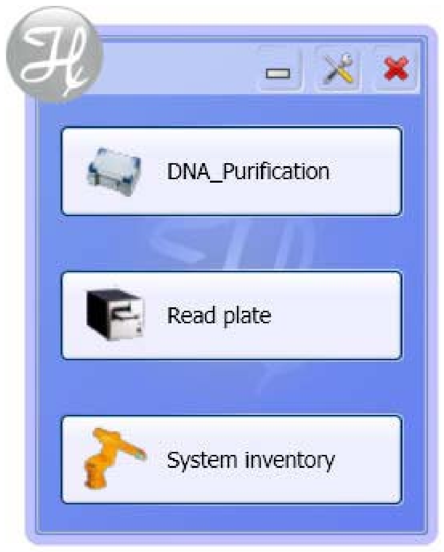
**Hamilton's VENUS software is not intuitive to navigate on its own.** The editors provide little direction once they are opened, and existing files are saved by default to directories that can be difficult to find. The use of a graphic user interface (GUI) can help organize method files and provide a more intuitive interface for the user.

**Several GUIs are available, such as the Hamilton Method Manager 2 and the Application Launcher.** These GUIs require their own installers, which are located in the online Resource Center. For methods used with a formal product, a formally released and version-controlled GUI software must be used.

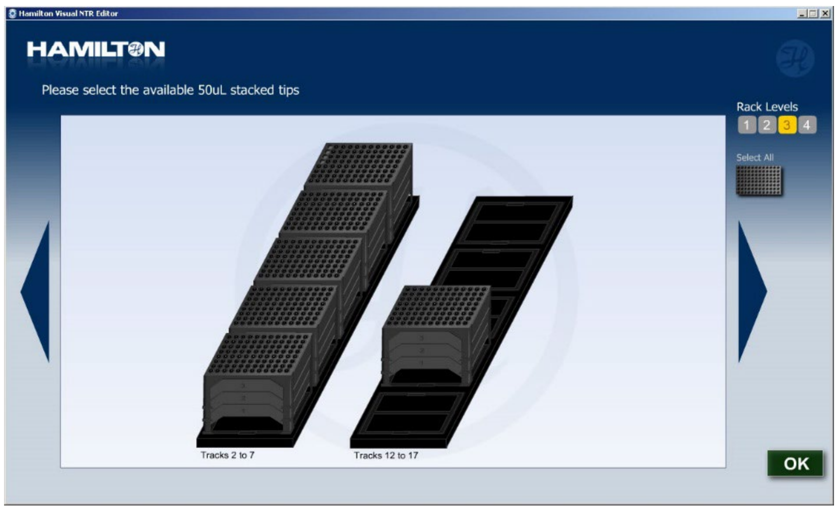
**The Method Manager 2 is a desktop application that is intuitively designed to simplify user interactions with VENUS.** This latest version allows one to launch and organize methods, access editors and folders, and manage log files more effectively. This version has more modern Hamilton branding and support for full/widescreen views and touch screens.



**The Application Launcher is a program that offers a list of shortcuts to run VENUS methods for the STAR, NIMBUS, or Microlab VANTAGE.** Pre-set and customizable icons can be configured for each method. Different command line options can be set to control the execution of the called program.



**When using NTRs, plate stacking, or specific plate or tip mapping, consider using Visual libraries.** These libraries include a GUI component during runtime, but also offer advanced functionality to control and manage labware and sequences.


Hamilton Visual NTR Editor
HAMILTON
Please select the available 50uL stacked tips
Rack Levels
1
2
3
4
Select All
Tracks 2 to 7
Tracks 12 to 17
OK

# 5 Saving and Exporting

**The following sections describe the best practices for saving and exporting methods.** In general, version control and maintaining backups are the most critical practices.

## 5.1 Saving

### 5.1.1 Folder Structure

**Save methods, custom libraries, and labware within their respective Hamilton directory folders.** All project-specific files should be grouped in subfolders for each project. Keep files needed for simulation runs and archived drafts of methods in clearly labelled folders within these subfolders.

### 5.1.2 Versioning and File Naming

**All method and sub-method library names should include a suffix identifying the version using a nomenclature that describes top, middle, and low-level changes.**

| **Nomenclature = X.Y.Z (example: 0.2.5)** | |
|---|---|
| X | **Top-level Version Control**<br><br>Will require major changes to the method or sub-method library when upgrading from one X rev to another. Little to no similarities to the previous top-level number. Examples: overhaul of the entire method structure, addition of new hardware or integrated devices, addition of new workflow steps. |
| Y | **Mid-level Version Control**<br><br>May require some changes or parameters corrections but will allow upgrade with easy implementation. Mostly like the previous version with some changes that would require updating the method. Examples: adjustments to the method structure, corrections to entire segments of the method/workflow. |
| Z | **Low-level Version Control**<br><br>Should require no changes or parameter corrections and will be almost identical to the previous version. Examples: typos, variable assignment corrections, volume calculation corrections. |

**Track all version changes within a sub-method titled "_VersionHistory" within the method or sub-method library.**

**Use generic names when possible to make sharing methods easier.** The names should describe the project or application the method is for. Avoid using company names for folder, method and labware files. All method names should include a suffix with the version number as detailed above; when a new version is saved, move any old versions to an archive folder.

**Follow the naming convention used for default labware when naming new labware definitions.** Include a prefix of the manufacturer and a brief description of the labware. For example, a Falcon branded 96-well flat bottom plate would be named "Fal_96FlatBottom". More detailed info like the part number should go in the description field in the labware definition.

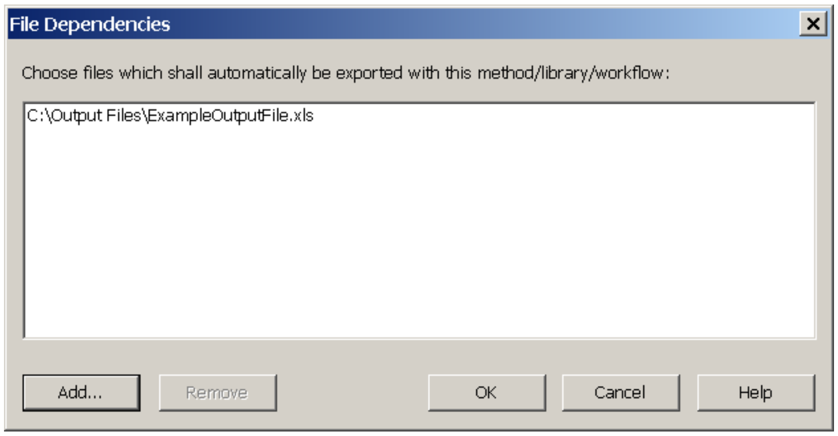
**For larger projects, consider the use of a git repository for sharing and version control of methods and their associated files.**

### 5.1.3 Installation Folder

Maintain all materials necessary to recreate the system setup in the default Hamilton installation directory in a subfolder named Hamilton Software and Manuals. Store all installers for the main software, libraries, and drivers in this subfolder. Transfer electronic copies of the manuals and periodically export method files here during development. Adding a shortcut to this folder on the desktop of the computer is helpful for recreating the system setup on another computer.

## 5.2 Exporting

**Export the method as a *.pkg file and include the original Hamilton files upon export.** Include any dependencies and file paths needed by the method in order for it to run properly when imported.


File Dependencies

Choose files which shall automatically be exported with this method/library/workflow:

C:\Output Files\ExampleOutputFile.xls

**Export methods to local drives before sharing them to shared drives**, since exporting directly to shared drives can corrupt files. Certain files such as liquid classes may not be properly exported, so save backups of these files in their respective editors.