

Technical Aspects:
Creating Macros for Aleph
(Using Macro Express or Other 3rd Party Languages)

Linda Moss

Harvard College Library Technical Services

NAAUG
7 June 2005

Table of Contents

I.	Intro	3
II.	Basics – Things to keep in mind even for simple macros.	5
III.	Using Text Files	7
IV.	Using Subroutines or Macros Calling Other Macros	9
V.	Library Macros: Genlib	11
VI.	Macro Express “Control” Variable Overview.	12
VII.	Timing Loops and Error Checking in Macro Express	15
VIII.	Controls in Aleph 16.02 with Macro Express	17
IX.	Capturing Right-Pane Controls in the ACQ Module.	18
X.	Using Visual Basic Scripts	20
XI.	Appendix A: Wait For Aleph Node To Open script	21
XII.	Appendix B: Pull Subfield From MARC Line	22

I. Intro

Introduction

Why use macros in the first place? They can save time, improve ergonomics by reducing mouse-clicks and keystrokes, improve the consistency of data entry, and save the mental energy of your staff by reducing frustration and aggravation at workflows that are more difficult than they need to be. But there are also dangers—it is possible to charge headfirst into a swamp. A little bit of thought at the beginning, and periodic reevaluation of the needs and solutions can save much aggravation and wasted time.

Why Use Macros?

First, why should (or shouldn't) you use macros? Some possible reasons:

- Save Time
- Ergonomics: Save keystrokes
- Consistency of data
- Reduce frustration, thus conserving mental energy for the real work.

When planning a macro system, it is important to think about (and periodically re-visit) which of these reasons are most important, because trade-offs between these goals can affect decisions. Similarly, when evaluating the effectiveness of a macro system, all of these factors should be taken into consideration, because there may have been benefits in an area other than what was anticipated.

Biases of This Document

This document is biased by the following realities at our workplace:

- We use Macro Express, and thus unavoidably, many of the technical considerations given in this document are specific to that software. (There is also much here that is generally applicable)
- We have a large macro user-base spread out over several divisions and even departments. Because of this, we have centralized methods for development and distribution; this bias will be apparent in this document.
- Because of the number of users, groups of users use shared networked MEX files. This limits our use of some activation methods that may be useful to other locations.
- We have written macros primarily for the Cataloging and Acquisitions modules.
- We have one implementation of Aleph serving nearly 100 libraries. Some of the problems we've solved with macros could also be solved by changing Aleph installation options, but with such a wide-spread system, this is not always possible for us.

Designing Macros for Others to Use

When writing simple macros for only oneself, they can be off-the-cuff, whatever-works-fastest. But once you add greater complexity or other people to the equation (even a small handful of people you work with regularly) the dynamics change. Here are some things to take into consideration when using macros for anything other than simple individual use:

Iterative process: One of the most important things to realize is that designing a macro suite is an iterative process, not a one-time project. As you use the macros, you will come up with ways to improve them, and ideas for new macros. Dependencies will change: you will upgrade Aleph, or your workflows will need to

adjust to external circumstances. Bugs in the macros will be discovered and need to be fixed. Something else your macro relies on will change. Before you begin, be aware that the process will need to be continuous, or at least periodically revisited.

Centrality: You don't want five copies of a macro being debugged and edited by five different people. This is five times the time and effort of having one person fix or enhance the macro and then redistributing it. If each macro has a definite "owner" (whether it is the same person for all macros or not) then each bug will only need to be fixed once and all enhancements will immediately be available to all users of the macro.

Reliability: When a macro will be used by more than just its creator, reliability becomes much more important. The macros' idiosyncrasies will be much less intuitive, and trying to fix problems on an ad-hoc basis becomes more difficult and expensive. Thus, more in-code error-checking and robustness become necessary.

Intuitiveness: Another difference between writing macros for yourself and writing them for other people is that much more thought and effort need to go into the user interface when others are going to use your macros. What seems painfully obvious to the creator of a macro may be surprisingly counterintuitive to someone else.

Coding practices: Good coding practices are about writing code that goes beyond "just working". It is about code that is more robustly reliable, readable, and easier to maintain. This will save time debugging, updating, and maintaining macros. Coding practices become increasingly important as macros grow more complex and widespread: start with the basics, and be aware of more careful standards to apply as they become necessary or desirable.

The truth about debugging and maintenance: Debugging and maintenance take more time than the initial coding. This is a reality that the software development world has come to terms with that you need to be aware of as well. Anything that speeds debugging and simplifies maintenance is likely to save more time than things that merely speed up initial development.

Design vs. Implementation

It is important to separate the concept of "design" (what the macro does) from the concept of "implementation" (how it does it), even if the same person is doing both the designing and the implementing. Why keep these separate?

- **They require different skills and knowledge:** Design requires an understanding of the process and workflows being automated, the standards involved, workplace policies and the abilities of the users. Implementation requires a technical knowledge of the tools involved, and how to best use them to solve the technical problems presented by the design specs.
- **Different considerations:** Even if the same person is doing both the design and implementation, the two activities should still be separate, as they require different considerations. And as with all good craftsmanship, form should follow function. When the design and implementation stages are mixed, it is too easy to work backwards-- trying to base the user needs on what the code can do.

That said-- the cycle of design and implementation, like everything else, is iterative. During the design phase, make it clear which parts of the design are crucial, and which are optional (or flexible within given parameters). This will greatly aid the implementation. Likewise, difficulties with implementation may require the design to be revisited. Also, ideas generated during the implementation may be worthwhile to add to the design. Don't be scared to switch between design and implementation, but always know which you are working on at any given time.

Types of Macro

The possibilities for macro-writing cover a wide spectrum, so it is helpful to look at the different needs presented by different type of macros. One possible way to categorize macros is:

- Simple Macros
- Workflow/Processing Macros
- Library Macros

Simple Macros:

These macros perform **one simple task** such as typing a Unicode character or often-used string, increasing the size of a window, or issuing a single command. They are **small in scope**, generally between three and three-dozen lines long, and tend to be **self-contained**: less likely to call other macros, or to use external files than more complex macros. While coding practices are less crucial than with more complex macros, you should still take the basics into consideration.

Workflow / Processing Macros:

These macros perform multi-part **workflow** oriented tasks, or complex **processing** of MARC code or other data. They tend to be **cooperative**, calling other macros and/or using external text files or applications to help in their work. **Coding practices** become much more important here, due to the greater complexity.

Library Macros:

These are macros that are never user-invoked but are **called only by other macros**. They allow for widespread **code reuse** when the same task has to be done by many macros (two examples from our code-base: pulling a subfield out of a MARC line, or pressing a button and checking for error messages. When done well, having a library of “subroutine” macros gives the following advantages:

- Speed writing the macros
- Fewer errors writing macros
- Easier to debug
- Easier to update
- Cleaner, easier to read code.

II. Basics – Things to Keep in Mind Even for Simple Macros

Here are some basic coding practices that one should keep in mind even when programming simple, small-scope macros. Implementing these practices now can save much pulling of hair later. Begin here, and add suggestions from other sections as your macro use grows more complex and/or widespread. Some of these recommendations are specific to Macro Express, while others are more generally applicable. (Please note: What Macro Express call a “Remark” most other languages call a “Comment”.)

With all of these suggestions: If your macro has more than two dozen lines, and/or uses external files or calls other macros or applications, then please see the other sections for more advice on coding practices.

Coding Suggestions

Variables and naming:

- Avoid single-digit variables, as they are more difficult to search for.
- Naming macros: It can be helpful to include a prefix in each macro name, so that related macros sort together as groups.

- If your macro language allows it, (Macro Express does not), then give your variables intuitive names that avoid abbreviations. (Abbreviations can be more difficult to interpret later than one would expect.)

Comments in Code:

- If you cannot give a variable an intuitive name, then add a comment at the first (and each non-proximate subsequent) use, giving a brief explanation of what it is for.
- In the Macro Express Scripting Editor, variables or code are sometimes hidden **inside** a line, and you will either need to double-click to reveal it or look for it in the Direct Editor. Whenever you have information (especially a variable, filename, etc.) hidden inside a ME line, a comment should be used to reveal it, since most macro writers will be working in the Scripting Editor.
- Give a brief description of what each segment of the code is trying to do.
- Longer macros can be subdivided visually by using comments to designate segments:


```
//=====
// Name of New Segment
//=====
```

 This is especially useful in languages like Macro Express that don't allow actual subroutines, which are a better method of segmenting code logically.

Comments in Header:

It is useful to have a comments/remarks header at the top of the document that puts in one easily accessible place some of the information one may want to discover about the document. Some helpful things to include in the header are (we will add to this list in later sections):

- **Purpose:** What the macro is trying to accomplish. This will be obvious to you as you write it, but to someone looking back six months from not, it may not be as clear as it currently seems!.
- **Applications used:** This not only helps understand the macro better, but can be useful when the application in question is updated and one needs to know which macros need to be updated to match.
- **Variables:** If there are only a few variables in the program, list them all and what they are used for. Ideas about when to list variables in more complex macros are discussed later.

Versioning

As macros are debugged and updated, much consternation can be avoided if you can always tell what version of the macro you are working with. Nothing is more frustrating than having an extensive debugging session finally reveal that the problem was that an older version of the macro or a sub-macro was being used! Ways to make it clear whether or not you have the code you think you have:

Revision history in Notes tab: It can also be helpful to keep a revision history in the Notes tab, including a Version number, and the date and reason for any changes. Beware: when you copy a macro in Macro Express, the Notes tab is NOT copied with it automatically.

Version number in name: You can include a version number in the name, such as myMacro_v2.0 or myMacro_2005_06_02. This has implications for other macros in suites where macros call other macros, as shown in the relevant section below.

III. Using Text Files

Sometimes, it is useful for macros to use external text files. These can be temporary processing files, preferences/settings files, or more permanent look-up tables that store information externally to the code.

Temporary Working Files

These are files that are created by your macro to aid in processing; they are either deleted when the macro is finished, or abandoned to be overwritten the next time the macro is run. There are three types of file processing that we will look at:

Reading the Whole File into or out of a Text Variable

This is done using the “Variable Set String → Set Value From File” and “Variable Modify String → Tab 2 → Save to Text File” commands. Possible reasons for doing this include:

- Saving info for use later in the macro
- Reading information that was output by another application
- Creating output intended to be read by another application

Reading a File in Line-by-Line

There are many applications for this, including some tricks to get around the lack of arrays in the Macro Express language. One common use is to have the macro copy a bibliographic record to the clipboard in Aleph, save the clipboard to a file, and then read the file in line-by-line to process.

Line-by-line file processing uses the “Text File Begin Process” and “Text File End Process” commands.

Reading a Delimited File

This allows you to parse comma-separated values into separate Variables. It can also be used to find a value in a table stored as a text file. The relevant text processing commands are “ASCII File Begin Process” and “ASCII File End Process”.

Important Note: The Recycle Bin

Note that if you frequently create and delete files with macros, you will need to empty your Windows Recycle Bin periodically, as this is where the files will end up. We have discovered that if the recycling bin gets too full without being emptied, it can cause performance problems. (very severe performance problems, in the case of a few users who had Recycle Bins with over 50,000 items).

Preferences/Settings Files

These are files that save a particular user’s preferences, settings, or current state of what he or she is working on. This makes macros more flexible by allowing the same macro to behave slightly differently for different individuals or groups. One use of settings files is to simplify input: If the macro needs to ask the user for information that doesn’t change often it can be stored in the settings file so that the macro doesn’t have to ask each time. Additionally, different groups of users can have different information stored. Sometimes it is appropriate to have a separate macro just to update the setting file in a user-friendly manner.

.INI files are particularly useful for this if your language has commands for reading to and from .INI files, because they allow you to retrieve single values from a file with many such value-strings. See the Macro Express “Variable Set String → Set Value From INI File” and “Variable Modify String → Tab 2 → Save to INI File”

Look-Up Tables

It can sometimes be helpful to store information externally to the code. Possible reasons for storing information in lookup files include:

- *Allows more data to be stored:* We have a macro that looks up a MARC country code for a given country. If we put all this data into the macro, the macro would be enormous!
- *Easier to view and update.* Updating or changing a macro would involve digging through the code, worrying what else it would affect, etc. A text file, especially with good comments explaining structure, is much easier to expand or update.
- *Easier to process:* Sometimes the things that make the temporary working files useful are also useful in more permanent lookup files.

General: Error-Free Use of Text Files

- Instead of “hard-coding” a filename or EXE name into the code, set a variable at the top of your script to the file name. Then, use that variable anywhere where you need to refer to that file. It will be much easier and less error-prone during updating if the file name ever changes, or moves to a different directory.
- **Checking for file existence:** Use the “If File Exists”, “If File Does Not Exist”, “If Directory Exists” and/or “If Directory Does Not Exist” commands before using a file/directory to make sure it still exists where you think it does. Warning: Version 3.0.4.1 of Macro Express will CRASH if you use “if file exists” and the DIRECTORY the file was supposed to be in does not exist. So, sometimes it is best to store the directory and file name in separate variables, and check for the existence of the directory before checking for the existence of the file.

```
// Set the path and file name in variables.
Variable Set String %T80% "C:\This\is\The\path where\we expect\to _find_the_file\"
Variable Set String %T81% "theFile.txt"
//
//
// First, verify the directory exists.
If Not Folder Exists "%T80%"
    // Use a TextBox to give an error message, and then stop the macro.
    Macro Stop
End If
//
// File: %T80%\%T81%. (Macro Express Hides everything before the \ until you
double-click it)
If Not File Exists "%T81%"
    // Use a TextBox to give an error message, and then stop the macro.
    Macro Stop
End If
//
// We now know the directory and the file both exist. So we proceed with the macro
// Using %T80%\%T81% every time we need to refer to the file in the code.
```

- **Comment in header:** Place a comment in the header telling what file is used, where it exists, and whether the macro creates it or expects it to exist already.
- **Distribution issues:** Each type of file has its own distribution issue. Files that will exist locally on the user’s machine need to be in a path that exists (or can be created) on each user’s machine. Look-up files may be handled differently: we store files that pre-exist and are constant between users on a central networked drive, and point the macro at the networked location to use them. Another possibility is to have each user make sure the files exist on his or her local drive.

- **Comment lines in text files.** It is possible to add comment lines to your text files—lines that the macro will ignore when processing the file, but that allow information to be placed for a human reader of the file. This may include information about what macro(s) run the file, what sort of data it contains and how it is formatted, and things that need to be taken into consideration for the file to be edited. We do this in look-up files by putting a hash-mark (#) at the beginning of each line we wish to be a comment, for example:

```
# This is a lookup file for the macro shown below. Lines
#   containing a "#" will be ignored.
#
# File Structure:
#   The first three characters are the two-digit numeric
#   value a number followed by a space. The rest of the line
#   gives the number's textual equivalent.
#
01 One
02 Two
03 Three
# etc, etc, etc
```

and then when reading the file, use the code:

```
// Text file T80\T81. Current line will be read into variable T20.
Text File Begin Process: "%T81%"
  If Variable %T20% contains "#"
  Else
    // Code for processing the text file goes here. Only those lines that
    // don't contain a "#" will be processed.
  End If
Text File End Process
```

IV. Using Subroutines or Macros Calling Other Macros

Another trick that helps as macros grow more complex is to have macros call other macros. Macro Express does not have a provision for subroutines, but calling other macros can be a serviceable substitute. Using subroutines or multiple macros can be useful for a number of reasons:

- **Reusable code:** Code that is used more than once, or by more than one macro, needs to be written only once, debugged once, and updated in one place.
- **Clarifies the logic:** Used well, separating some code out into sub-macros can make the code easier to read.
- **Functional:** Sometimes it is the easiest, or possibly only, way to accomplish something. Some tricks and “cheats” in Macro Express include using sub-macros to avoid deeply nested if-statements, or to put a nested loop into a sub-macro to provide an easy way to jump out of the nest.

General Principles

Cohesion & Coupling

When dividing code into classes, files, subroutines, or other structures, two important principles to keep in mind are those of *cohesion* and *coupling*. “Coupling” is the principle that like code should be with like code. “Cohesion” is the principle that non-like code should not be with non-like code.

For instance, if you have everything in one file, you have tight coupling (everything is in the same file, so all like code **MUST** be in the same file) but weak cohesion (everything is mixed together, regardless of how different it is). Contrarily, if you have every single statement in a separate file, you will have strong

cohesion (everything within a given file is similar to itself) but weak coupling (there's a lot of similar code spread out among lots of different files).

The ideal is to have a balance between these two, as much as possible. For instance, trying to contain the code that interacts with the Aleph GUI in one file, the code that simply processes strings could be in separate routines based on what processing is being done, and code that interacts with OCLC Connexion or RLIN21 could be similarly contained.

One result of this is to minimize (within reason) the variables and other information that need to be passed between units of code. This allows one to look at the larger picture of the code structure, looking at the code-units and how they interact without worrying about what's happening inside.

Encapsulation

However, in order to really “not worry about what's happening inside” the various code-units, you need assurance that the code in one unit isn't having unintended effects on the code in other units. This is called encapsulation: where each code unit is protected from outside interference.

Object-oriented programming languages provide mechanisms to enforce encapsulation automatically; Macro Express does not. For instance, all variables are global in scope, so one macro setting the T38 variable will affect all macros in the suite that use the T38 variable.

In Macro Express, as we describe below, we can build a sort of pseudo-encapsulation by segmenting the namespace: that is, deciding which ranges of variables are available to which macros for what purpose, and documenting these decisions with comments/remarks in the code. Also, clearly defining and documenting each macro's “inputs” and “outputs” can help avoid later confusion.

Input and Output

When writing a macro that will be called by other macros, keep in mind:

- What information (variables) you want to pass TO the called macro. These are the INPUTS.
- What, if any, information (variables) you want the called macro to RETURN to the calling macro. These are the OUTPUTS.
- What, if anything, you want the called macro to DO other than passing variables back to the calling macro. In programming terminology, this is sometimes referred to as a “SIDE EFFECT”, a hold-over term from the time when computer programs were primarily just about processing data.

It is a good idea to list intentional variable INPUT and OUTPUT values in remarks at the top of the macro. An INPUT is any variable that is set outside the current macro, but used within it. An OUTPUT is any variable that is set within the macro, but used outside of it.

Besides input and output variables, there are other potential connections between programs to watch out for: creation, change, deletion, or reading of a text file, or changing the state of the Aleph GUI. The rule of thumb is to be aware of everything a macro relies on being there when it begins to run, and everything it does that may affect other macros.

Variable Namespaces

We've talked about the intentional connections between macros. Because all variables in Macro Express are global, there are also unintentional connections. A subroutine macro, for instance, may change a value that the calling macro wasn't expecting to change.

One way to prevent the unintentional re-setting of variables is to segment the variable namespace. For instance, deciding that macros that are called by many macros, but don't call other macros themselves, will use only variables in the 90's range, and then avoiding that range of variables for other macros. Or setting

aside the 80's range for variables that will be used across a suite of macros, and then assigning each macro in the suite a range of variables that it can use for internal variables.

Comments / Remarks in the Code

In addition to the suggestions in above sections, when using macro suites where macros call one another, it is helpful to put the following into comments/remarks:

In the Header:

- If the macro is designed to be called by another macro, the header should list inputs, outputs, and what range of other variables are used in the macro.
- If the macro calls other macros, these should be listed in the header as well.

In Code:

- When another macro is called, it helps to add comments reminding what the inputs and outputs of that macro are, as well as what it does.

Naming Macro Express Macros

Hierarchical, storable names: In Macro Express, it helps to use hierarchical names that sort well in the MEX file window. For instance:

- If a suite of macros is actually one system—i.e., macros in the suite call and rely on other macros in the suite, begin the name of each macro in the suite with the same prefix, followed by an underscore.
- It also helps to differentiate “invokable” macros (the ones that the user actually calls, via command key, short key, pop-up menu, or other method) from “subroutine” macros, which are called only by other macros. We do this by following the suite prefix with `_sub_` on subroutine macros. It may be helpful to also add `_inv_` to invokable macros.
- After these prefixes comes the actual macro name.

Version numbers in names: We mentioned in an above section how appending a version-number suffix to a name can avoid confusion later. While still a good idea in suites of interacting macros, it is a bit more work.

In a suite of closely related macros, when you update the version-name of a macro, you need to update every macro that calls it to refer to the new version (which may cause a small updating cascade, as those macros have a version-number change). This takes a few extra minutes, but it has the following advantages:

- All macros have the version number in the name, which prevents the debugging confusion that can arise when you think you are using a more recent version of the macro than you actually are.
- It is obvious by looking in the code of a macro which version of a sub-macro it is actually calling. This also can save debugging time.

V. Library Macros: Genlib

Genlib is our library of macros that are called by multiple macros and macro suites—bits of code that are used over and over again, so it is easier to write and debug them only once. Reusable code that only needs to be debugged in one place speeds production, improves code readability, and simplifies the updating and debugging of code.

These are some of the principles we followed when building our Genlib library. Many of these points are based specifically on the abilities and limitations of Macro Express.

Not calling other macros: In order to make the effects of the Genlib macros as predictable as possible, our Genlib macros do not call other macros. If our system became complex enough, it would be very possible to implement a library of interconnected macros that call other macros; but this should not be done unless the benefit is great enough to justify the added complexity.

Variable namespace: We've delegated the 90's range of variables to Genlib macros. Genlib macros can ONLY use variables in the 90s range. Similarly, other macros eschew the 90's range except to set inputs and read outputs from Genlibs. Because of this, we can drop any Genlib into any other macro without worrying about overwriting variables.

Macro naming: In accordance with the alphabetical-hierarchical naming policy described above, all Genlib macros begin with the Genlib prefix. As our collection of Genlib macros grows, we are adapting the naming scheme to include a second more specific prefix:

- ◇ **Genlib_gui_** macros interact with some application's GUI. Even if they are generally used with Aleph, these macros are applicable to most GUI applications. Examples:
 - Entering text in an edit box, making sure it overwrites any text already present.
- ◇ **Genlib_Aleph_** macros work specifically with the Aleph GUI. Examples:
 - Clicking a button and checking for Aleph error boxes and bubble windows.
- ◇ **Genlib_text_** macros simply do text processing
 - Pulling a subfield from a line of text in MARC-field format
 - Removing non-filing characters from a line of text in MARC-field format
 - Getting the first X characters in the Yth word. (Used to build derived OCLC searches)
- ◇ **Genlib_file_** macros interact with the file system.
 - Verify that a directory exists, and create it if it does not.

Version name suffixes: For Genlib's we use a letter rather than a number. This is to prevent confusion that may arise if people erroneously expect the Genlib version number to match the version number of the suite that is calling it. Genlib's should be changed very rarely, since it is difficult to tell which macros will be affected. Debugging is generally OK, especially if the bug will adversely affect other macros calling the Genlib. But great care should be taken in adding or changing functionality.

VI. Macro Express "Control" Variable Overview

What You can Do with Control Variables

Setting a "Control" variable in Macro Express give ME the "handle" for that control, allowing it to do certain things with it, including those described below:

- **See if a control is visible:** Possible uses include using this ability to see whether a particular window is on the screen or verify what node of Aleph you are in.
- **See if a control is active:** This can be used to check a window's state or to verify a control before trying to use it. Also, it can be used to help prevent timing issues: if you capture a button control before clicking it, you can wait until the button has become inactive (verifying that Aleph has finished processing the button press and is ready to accept new input) before continuing the macro.
- **Enter text in an edit box:** You don't need to capture the edit-box control's handle to enter text into it, as long as you can ensure that it has focus. But capturing it first allows you to issue a command to make sure that the text you are entering overwrites any existing text the control may have.

- **Push a button:** Many buttons in Aleph (and elsewhere) have command-key alternatives that can be used instead, and sometimes this is the easiest and most efficient alternative, just make sure the correct pane has focus! Command keys in Aleph are local to the pane they appear in. For some Aleph buttons it is good to capture it first so you can wait for it to become inactive, as described above.
- **See what text a control contains:** This is useful to verify that the correct button is focused when the macro issues a “Mouse click” command, however, some controls on Aleph do not allow their text to be viewed in this way.
- **See if a control has focus**
- **Set focus to a control**
- **Mouse-click on a control**

“Get Control” vs. “Capture Control” in Macro Express

There are two ways to set a “Control” variable in Macro Express, “Get Control” and “Capture Control”. “Get Control” works by example: you show Macro Express the control in the other application while you write the macro, and it saves the control's signature. This works for applications that have “statically allocated” controls: that is, they have the same signature each time the application is run. Most of the Aleph 16.02 error windows are “statically allocated”.

However, some applications (including the main GUI of each of the Aleph 16.x modules) have “dynamically allocated” controls. This means that the signature of the control is not the same between runs. It has to be “captured” while the program is running, using the “Capture Control” command.

“Getting” a Control Variable in Macro Express

To “Get” a control for Macro Express, do the following:

- 1) Open the application you want your macro to use
- 2) Find the control you want to be placed in the variable
- 3) Enter your Macro Express script, and add a “Get Control” command
- 4) A “Get Control” dialog box will appear. Drag the bulls-eye onto the control you wish to capture.
- 5) Tell both dialogs to save.

That’s it!

“Capturing” a Control Variable in Macro Express

Capturing a control on the fly is a bit more complicated. You can capture a control if one of these three things is true:

- 1) It has the focus
- 2) You know its screen or window coordinates or
- 3) The mouse-cursor is over it.

Ways of Getting the Focus to a Control

- **User sets focus:** For some macros, you can assume that the user has already set the focus to the control that you will want the macro to act on.
- **Automatically:** Sometimes, you can count on the focus going to a particular control when you perform some action, such as hitting a command key. It is a good idea to give a time delay of 200 milliseconds after landing on the control to give the focus a chance to “take”, and then to verify that you really are on the control you expect (more on how to do this later).
- **By tabbing:** If the macro can ascertain where the focus is, and how many tab-stops it is to the control you want, you can text-type “Tab”s with “Wait for Text to Play Back” commands until you reach the correct control. Warning: If there are circumstances that will change the number of inactive controls between where you start and where you land, it can throw off your count.
- **“Set focus to control”:** After the control has been obtained with either “Capture Control” or “Get Control”, you can put it in focus at any time with the “Set Focus to Control” command.

Finding X,Y Coordinates for a Control

TO find controls by their X, Y coordinates, you need to know where the controls are **anchored**. The controls remain the same distance from their anchor point regardless of how a window or pane is resized. A common anchor point for controls is the upper left-hand corner of the parent window.

In Aleph 16.x, the multiple panes make the anchor points a bit less straight-forward. See more on finding the anchor points in Aleph below.

Mouse Cursor Over Control

As a last resort, you can ask the user to put the mouse over a control and then hit the shift key to continue. You should still check to verify the user selected the control that you expected.

Edit Box Window and Static Text Controls

A pop-up window can be a “control” as well. For some applications, Aleph included, many error messages will have a window with the same control signature. This window will have a static text box and a button that also have statically allocated control signatures.

You can “Get” the window and static text control using the simpler “Get Control” option. When checking for error messages, you can:

- 1) See if the Window control is visible
- 2) Get the text of the static text control
- 3) Use the “If Variable Contains” command do compare the error message to possible expected values, and respond accordingly.

Verifying You Have Landed on the Correct Control

Reading Control Text

If the control you are looking for has a known text value, you can verify that you have found the correct one with the “Variable Get Control Text” command. Some things to keep in mind:

- Controls where the text contains an underlined character to indicate a command-key option (such as the “Duplicate” button) will parse as having an ampersand before the underlined letter (“D&uplicate”).

- This command works best on static text controls and button controls. It will usually work on editable text boxes, but many of the text boxes in Aleph aren't compatible with the command. It will almost never work on list boxes.

Checking Control Size

This is a sanity-check that is useful in some cases. For instance, in Aleph, selecting a pane will put the focus into the pane's home control. But if the controls in the pane are all inactive, the focus will be on the pane itself. To know which situation has occurred, you can capture the control you are on, and check its width. If the width is more than would be reasonable for an edit box, it is probably the pane that is selected, meaning that the controls are disabled.

Finding the Edit Control's Label

Knowing whether or not you are on the correct edit-box can be more difficult. It may not have any text you can check, and Aleph won't always let Macro Express read its edit boxes. But in extended-run macros especially, it is very important to check: if there was an error, you want the macro to stop quickly, not continue performing unknown tasks.

The Label of an edit box will usually be a fixed distance to the left:

- 1) Get the upper-left coordinates of the box you are on, using the "Variable Set Integer" command.
- 2) Go the proper distance to the left, and a little bit down
- 3) Capture the control at this location and read its text to verify that this was the label you expected.

The latest version of Macro Express also contains a command to save the control class to a variable. This can also be used in ensuring that focus is on the correct control.

VII. Timing Loops and Error Checking in Macro Express

Basics: Built-in ME Commands

Wait for Text to Play Back

Follow every "Text Type" command with a "Wait for Text to Play Back" command. Break large "Text Type" commands into multiple, smaller "Text Type" commands interspersed with "Wait for Text to Play Back" commands.

Delay / Wait for Time to Elapse / Pause

Delay

The "Delay" command lets the macro wait a number of seconds or milliseconds before continuing. If Macro Express continues too quickly after certain operations, it can create timing errors because the application it is trying to interact with hasn't had a chance to respond yet.

Wait for Time to Elapse

We don't recommend using this variant of delay, as it locks Macro Express so that you can't stop the macro. It does allow up to a week of waiting time and uses no CPU cycles while waiting, but in the meanwhile no macros can be used and it can't be canceled.

Pause

This one is good for debugging. See the Macro Express documentation.

Wait for Mouse Click or Key Press

- Key press: This pauses the macro until the user presses a key specified in the command.
- Left, center, or right mouse button: This pauses the macro until the user clicks on the mouse button specified in the command.

“Wait for Control...”

- Gain / Loose Focus
- Become visible / Invisible
- Become enabled / Disabled

Other “Wait for...” Commands

- Window Appear / Loose Focus
- Program to terminate
- Text
- Web page to load
- File to exist / Be ready

Waiting for an Unspecified Duration

When you are waiting for an uncertain duration, especially if you don't know whether the event you are waiting for will happen, you need to be a bit more creative. The code below waits for the current window title to change (which will happen when a window opens or closes)

```
//
Variable Set String %T10% from Window Title
//
// Variable N12 tells us which loop we're on.
Repeat Start (Repeat 40 times)
//
// If the window title has changed, stop waiting.
If Not Window Title "%T10%" on top
    Text Box Display: The window title has changed!
    Repeat Exit
End If
//
// See if we're at the end of our waiting loop.
If Variable %N12% >= 40
    Text Box Display: Finished waiting, and the window still hasn't changed.
    Repeat Exit
End If
//
// Wait a bit more. 100 milliseconds x 40 iterations means we have
// a maximum wait of approximately four seconds.
Delay 100 Milliseconds
//
Repeat End
```

Checking for Bubble Windows

The bubble error window in Aleph is a special case, as it appears as a control rather than a window. It is virtually impossible to check for in 16.01, but in 16.02 improvements were made that allow it to be captured via “Get Control”. Unfortunately, it is impossible for the macro to pull the actual text of the error message, since “Get Control Text” will only return “Bubble Error”.

VIII. Controls in Aleph 16.02 with Macro Express

Using Command Keys

Many things that can be done with Button Controls in Aleph can also be done with Command Keys, and this is sometimes the easiest and most efficient way to issue commands. Things to keep in mind:

- The command keys are pane-specific. Therefore, you should verify that the proper pane has focus before invoking a command key, or you could have unexpected effects!
- Menu command key equivalents are accessible from everywhere EXCEPT when the pane has a conflicting command-key of its own. Therefore, it is usually best to put a pane in focus that is known not to have a conflicting command-key before invoking a menu command by command-key.

To press a command key in Macro Express, you use the “Text Type” and “Wait For Text to Play Back” commands. For instance, below is the Macro Express code for typing Ctrl-d:

```
Text Type: <CTRLD>d<CTRLU>
Wait Text Playback
```

Using the <CTRLD>d<CTRLU> is advised over the (theoretically) equivalent <CTRL>d, because the latter can occasionally cause timing issues.

Putting a Pane in Focus / Selecting a Tab on a Pane

Ex Libris gave us a wonderful addition to Aleph 16.02 that is a godsend for macro-writers: each of the panes has a command-key equivalent. To put a pane in focus, text-type the following commands as described above:

```
Left Pane: <CTRLD>1<CTRLU>
Top Right Pane <CTRLD>2<CTRLU>
Lower Right Pane <CTRLD>3<CTRLU>
```

(see Aleph documentation for more).

To Select a Tab on the Current Pane:

Use ALT and the tab number. For example, to go to the second tab:

```
Text Type: <CTRLD>2<CTRLU>
Wait Text Playback
```

Navigating between Aleph Nodes

- 1) Set focus to left-hand (navigation) pane
Text Type: <CTRLD>1<CTRLU>
Wait Text Playback
- 2) Verify radio button is “functional” rather than “overview”
Text Type: <CTRLD><ALTD>1<ALTU><CTRLU>
Wait Text Playback
- 3) Set pane to correct tab using F-key equivalent. This will be F2 for the order tab, F5 for invoice,
Text Type: <F2>
Wait Text Playback

Note: The function-key equivalents for each of the left-pane ACQ tabs are:

- F2 for the Order tab
- F5 for the Invoice tab
- F6 for the Administration tab
- F7 for the Index tab
- F8 for the serial tab
- F9 for the find tab

- 4) Use command key to navigate to the correct node. This is Ctrl-Alt-[the letter in square brackets next to the proper item on the navigation pane]:
Text Type: <CTRLD><ALTD>I<ALTU><CTRLU>
Wait Text Playback
- 5) Wait for the node to open, and verify that you have arrived: we don't know how long it will take for the node to open, and sometimes it can take up to a few seconds. See below for one way that the macro can wait for, and verify, arrival. Appendix A shows the Genlib we use for this purpose.

IX. Capturing Right-Pane Controls in the ACQ Module

Most of the following assumes Version 16.02. There are drastic differences between the 15.x and 16.x versions of Aleph that affect macros—particularly in the recognition of windows and the capturing of controls.

Overview: Where the Controls are Anchored

Button Controls

Button controls in the Aleph ACQ module right-hand panes are anchored to the top and right of the pane. The x-coordinate for the right-hand edge of the window/pane can be found by adding the value for the left of the window to that for the width.

The top of the top-right-pane is a fixed distance from the top of the window, but the top of the bottom pane is difficult to find by X,Y, so we will propose another method below.

Edit Boxes and Combo Boxes

Edit boxes and combo boxes in the Aleph ACQ module are anchored to the top and left of the pane. In both right-panes, the left edge of the pane will be a function of the left-side of the window and the width of the left-hand pane—but we propose ways of finding the controls below that do not require you to determine the left-pane width.

Tabbing to an Control from the Home Control

- 1) Use the command-key sequence to go to the correct node, pane, and tab
- 2) Verify that you have landed on the home control, and that it is visible
- 3) Tab the proper number of times
- 4) Verify that you have landed on the correct control.

When this will not work:

- If the home control was inactive, you will not have landed where you expect.
- If a control between the home control and the control you are looking for was non-tab-able (due, for instance, to being inactive) then you will not have landed on the control you think you have.
- If the control you are seeking is inactive, you may have difficulty landing on it and/or capturing it.

Finding an Edit/Combo Box with X,Y Relative to the Home Control

- 1) Use the command-key sequence to go to the correct node, pane, and tab
- 2) Verify that you have landed on the home control, and that it is visible
- 3) Get the X,Y coordinates of the upper-left hand corner of the control you landed on
- 4) Add/subtract the proper amount to take you to the next control
- 5) Add/subtract a few pixel's worth of buffer
- 6) Capture the control at this location, and verify that you are on the correct control.

When this will work: For edit/combo boxes in the same pane as the home control. Especially useful if you need to capture several controls on the same pane; it saves you from having to tab. It is also useful when tabbing is unreliable.

When this will not work: For buttons on the right-hand edge of the pane.

Upper Right Only: Find an Edit/Combo as X,Y Relative to the Top of the Window and the Right-Edge of the Left-Hand Pane

- 1) Find the top edge of the window
- 2) Subtract the proper amount, and save the value
- 3) Find the left edge of the window
- 4) Set the focus to the left-pane, and get the control width
- 5) Add this width to the left edge of the window
- 6) Add the proper amount for the control location, and save the value
- 7) Get the control at the proper X,Y coordinates, and verify that it is the correct one

When this will work: In the upper-right hand pane, for controls anchored at the upper-left hand corner of the pane.

When this will not work: This will not work for the lower pane, or for buttons in the right edge of the upper pane. Also, it will not work if the Aleph window is small enough that the top-bars are folded over.

Getting X,Y Coordinates for a Button in the Upper-Right-Hand Pane

- 1) Get the left edge of the window and the width of the window and add them together to get the right edge of the window
- 2) Subtract the appropriate amount
- 3) Get the top edge of the window, and subtract the appropriate amount
- 4) Find the control at these coordinates and verify that it is the correct one with "Get Control Text"

When this will work: For buttons in the upper right hand pane.

When this will not work: The "Update" button on the invoice creation tab cannot be captured with X,Y coordinates. It must be tabbed to. Cannot capture any button via its X,Y coordinates if it is inactive.

Getting X,Y Coordinates for a Button in the Lower-Right Hand Pane

- 1) Get the right edge of the window as described above, and subtract the appropriate amount
- 2) Get the home control of the pane
- 3) Add or subtract the proper amount from the home-controls y value
- 4) Capture and verify the control

When this will work: For buttons in either the upper-right or lower-right hand pane of the Aleph ACQ module.

When this will not work: Inactive buttons.

X. Using Visual Basic Scripts

Why Use Visual Basic Scripts

- **User interface elements:** The user interface elements of Macro Express are incredibly limited, especially when writing macros to simplify user input. One of the things we use Visual Basic scripts for it to create more comprehensive input forms with more room for clear text messages.
- **Unicode compliance:** If your records are only in English, this will not be a problem, but as soon as you try to include names and titles in other languages you will become frustrated with Macro Express' lack of Unicode compliance. Visual Basic is Unicode compliant, and allows us to do text processing without losing diacritics or non-Latin characters.
- **Text processing capability:** Even if you work exclusively in English, the text-processing abilities of Visual Basic, Perl, C, and other languages are much more powerful than those of Macro Express, and will allow you to do greater processing of MARC records.

Why (or Why Not) Visual Basic?

We use Visual Basic for these additional capacities rather than another programming language because:

- It is already installed
- It is good for quickly building a simple user interface
- Non-programmers tend to be less intimidated on hearing "Visual Basic" than "C++"
- Because our use of it is trivial (from a software development perspective), many of the standard arguments and complaints against VB are mitigated.

We considered PERL, because it is good for text processing, but we would have needed to install it on every user's machine. Libraries with a smaller number of staff members using macros may want to consider this as an alternative. PERL has powerful text-processing abilities, is useful in many contexts, and is downloadable free from <http://www.perl.org>.

Writing Macros That Call VB Scripts

- **Running across a network:** We've discovered that given our network settings, we are unable to run our VB exe's across a network. Therefore, when the macro is run, it checks to see if the (small) EXE file has been copied locally to a designated local directory, and copies it there if it has not. We are then able to run the script locally.
- **Transferring info between Macro Express and VB:** There are two ways to do this: Command line parameters and text files. We use both, for varying purposes.
- **Waiting for VB script to finish:** When you run a Visual Basic script from a macro, you need to wait until it finishes. We do this by having the VB script write out a "Done.txt" file at a designated location. Before the macro calls the executable, it deletes this file if it exists. After running the executable, it waits for this file to reappear before continuing. This file can contain any information that the VB script is passing back to the Macro Express file.
- **Versioning:** Just as with macros, it is important to know what version of an EXE or DLL you are looking at. We do this by suffixing each EXE and DLL with the yyyyymmdd date that the code was released.

XI. Appendix A: Wait for Aleph Node to Open

```
// # Purpose:
//      * This macro tries to verify that a particular node in Aleph has opened by
//      looking for a button with a certain label to appear a given distance
//      from the upper right-hand corner of the Aleph window. (In Aleph 16.02,
//      buttons in the upper right-hand panel are always a fixed distance from
//      the top and right of the Aleph window, so long as the window is
//      maximized or close to it.)
//
// # Inputs
//      N90: Distance left of the right-hand edge of the window.
//      N91: Distance down from the top edge of the window
//      T90: Expected Text
//      T91: The error message we should give the user if control not found
//           after an extended wait
//
// # Outputs
//      C90: The control we found
//      T92: The (actual) text of that control. (C90)
//
// Internal Variables:
//      N93, N94, N95, N96
//      N98, N99
//
Delay 200 Milliseconds
//
// Set N12 to the X value of the right edge of the window by adding the left
// edge of the window to its width.
Variable Set Integer %N93% from Left of Window
Variable Set Integer %N94% from Width of Window
Variable Modify Integer: %N95% = %N93% + %N94%
//
// Set N22 to the Top edge of the Aleph window.
Variable Set Integer %N20% from Top of Window
//
// Get the coordinates of the control we wish to find
//
// Set N19 to the X coordinate of the button
Variable Modify Integer: %N96% = %N95% - %N90%
// Set N29 to the Y coordinate of the button
Variable Modify Integer: %N99% = %N98% + %N91%
//
//
// Now, we keep grabbing the control at (N19, N29) until it matches
// what we're expecting.
// (Try for up to five seconds)
Variable Set Integer %N93% to 0
Repeat Start (Repeat 50 times)
//
// Capture the control at (N19, N29) and grab the control text
Capture Control to %C90%
Variable Get Control Text: %C90% to %T92%
//
// See if the control text contains the expected phrase. If so, we're here!
If Variable %T92% contains variable %T90%
    Macro Return
    Repeat Exit
End If
//
// Wait a bit and try again.
Delay 100 Milliseconds
//
Repeat End
//
// If we're still here, ask the user to help.
If Message: "Macro Focus Lost while waiting for node to open"
```

```

// Macro calls itself ("recursion") in order to verify that focus has, in fact,
// been regained.
// .....
Macro Run: Genlib_WaitForAlephNodeToOpen_vA
// .....
Macro Return
Else
Macro Stop
End If
//

```

XII. Appendix B: Pull Subfield From MARC Line

```

// # Purpose: Given a line from a marc file, returns the requested subfield.
// * If the (T91) field character requested is:
// --> More than one character: Macro complains and dies.
// --> Blank or whitespace only: Macro returns Marc line with first
// eight characters truncated
// --> A $ sign: Returns Marc line, with first eight characters
// truncated and field designators ($$x) replaced by spaces.
// --> A single character other than "$" or " ", it will try to extract
// that subfield into T93
// * If it exists once, the contents of the subfield will be returned.
// * If it exists and repeats, all instances will be returned,
// separated by spaces.
// * If it does not exist, a blank string will be returned.
//
// # Inputs:
// T90: A line from a marc file
// T91: The subfield we want. (A space pulls any info that's not subfielded,
// such as the 008 linw)
//
// # Outputs:
// T93: Contents of the subfield.
//
// # Internal Variables
// N99, N98, N97
// T99
//
// T91 must be a single character. If it's not, complain and die.
Variable Modify String: Trim %T91%
Variable Set Integer %N99% from Length of Variable %T91%
If Variable %N99% > 1
Text Box Display: BUG:T91 not single character
Macro Stop
End If
//
// =====
// If T91 is a space, just chop off the first 8 characters,
// =====
//
If Variable %T91% = " "
Variable Modify String: Copy %T90% to %T93%
Variable Modify String: Delete Part of %T93%
Variable Modify String: Trim %T93%
Macro Return
End If
//
// =====
// If T91 is a $, chop of the first eight characters,
// and replace subfield delimiters with spaces.
// =====
//
If Variable %T91% = "$"
Variable Modify String: Copy %T90% to %T93%
Variable Modify String: Delete Part of %T93%
// It should be safe to assume there will be fewer than 99 subfields in the
// field.

```

```

Repeat Start (Repeat 99 times)
  If Variable %T93% contains "$$"
    Variable Set Integer %N98% from Position of Text in Variable %T93%
    Variable Modify Integer: %N98% = %N98% + 2
    Variable Modify String: Copy Part of %T93% to %T99%
    Replace "$$%T99%" with " " in %T93%
  Else
    Repeat Exit
  End If
Repeat End
Variable Modify String: Trim %T93%
Macro Return
End If
//
//
// =====
//       If the subfield is present, extract the contents;
//       otherwise, return an empty string.
// =====
//
Variable Set String %T99% "$$%T91%"
If Variable %T90% contains "%T99%"
  // Set N97 based on the position of %T99% within T90
  Variable Set Integer %N97% from Position of Text in Variable %T90%
  // Copy the full line to T93
  Variable Modify String: Copy %T90% to %T93%
  // Delete everything BEFORE the contents of the field (including the field
marker)
  Variable Modify Integer: %N97% = %N97% + 2
  Variable Modify String: Delete Part of %T93%
  // If the field repeats, we want to replace the subfield markers with spaces
  Replace "%T99%" with " " in %T93%
  If Variable %T93% contains "$$"
    // Delete everything AFTER the contents of the field
    Variable Set Integer %N97% from Position of Text in Variable %T93%
    Variable Modify Integer: Dec (%N97%)
    Variable Modify String: Copy Part of %T93% to %T93%
  End If
  Variable Modify String: Trim %T93%
  Macro Return
Else
  // Subfield NOT present. Just return a blank string.
  Variable Set String %T93% ""
  Macro Return
End If
//
//
Text Box Display: BUG- Macro reached end without returning

```