A New Approach To Going GUI

By William Baker

One of the big issues in converting character-based applications to GUI is translating existing screens into a GUI format. Visual PRO/5™ Rev. 1.x offers two migration paths. One path uses the Resource Editor to lay out GUI windows that replicate the old character-based screens. These screen layouts are stored in files called resource files. While this method is acceptable for converting small applications, it becomes much more difficult when you get past 50 or 100 screens. The other migration path puts GUI mnemonics in your programs to create GUI screens at run time. This generally means replacing '@' mnemonics, though it is possible to automate this process to some extent. These two migration techniques can be mixed by creating one or more standard minimal GUI windows with Resource Editor. Then, by using mnemonics in your programs, you can change the window title and populate the window with the labels, input fields, checkboxes, and GUI controls. A sample prototype window appears below.

Sample Prototype Window

Kilauea offers yet another path through this conversion jungle with a new component called ResCompiler, which is short for resource compiler. This utility reads a description of one or more windows contained in an ASCII file and compiles it into a resource file used by your programs at run time. This ASCII file is called an "ASCII resource file" to distinguish it from the "regular" resource files utilized by the Resource Editor and the Visual PRO/5 interpreter. I'll abbreviate it as ARF, although the official BASIS International Ltd.™ file extension is .arc.

When you team up ResCompiler with the totally redesigned version of the Resource Editor, now called ResBuilder™, you have a powerful new screen conversion strategy.

ASCII Resource File Basics

A complete description of the ARF syntax would more than fill up this magazine. It's documented on Early Access 1 diskettes, and you can download the documentation from the BASIS International homepage at www.basis.com.

Just to give the flavor of the syntax, take a look at this sample:

```
  VERSION "3.0"
// ASCII resource file definition for standard
// prototype window
WINDOW 1 "Standard Prototype Window" 20 20 640 480
BEGIN
ENTERASTAB
    BUTTON 1 "Ok" 190 420 100 25
BEGIN
DEFAULT
END
    BUTTON 2 "Cancel" 350 420 100 25
END
```

This ARF will compile into a binary resource file with one window that looks like the prototype screen at the beginning of this article. The window is defined on the third line, followed by the options and controls for that window. BEGIN/END pairs are used to group descriptors. The indents are entirely optional and are shown here to improve readability.

In most cases, it should be easier to automatically generate ARFs than to generate lots of mnemonics code, and using resource files will save you some program size. I'll go over three scenarios for converting character-based screens to ARFs, briefly discussing tactics for each one.

Scenario 1 - Existing Screen Layout Library

If you are fortunate enough to already have your character-based screens described in a consistent format, then you probably have the ingredients for a relatively easy conversion process. All you have to do is transplant the data from your existing library into the ARF syntax.

Be aware that the default unit of measure in ARF syntax is pixels. If you want your coordinates and dimensions to be dimensioned in pixels, you will have to convert character positions to pixels. For example, if your old X coordinates are based on 80-column screens and you're going to GUI screens that are 640 pixels across, you need to multiply the old values by 640/80, or 8, to get the dimensions in pixels.

If you want to switch the unit of measure in the ARF to chars or semichars, use the following syntax:

```
  #define UNITS_CHARS 1 //Use chars
#define UNITS_SEMICHARS 2 //Use semi-chars
```

For a complete description of units of measure, see page 7 of the *PRO/5, Visual PRO/5, BBxPROGRESSION/4® GUI Guide.*

Scenario 2 - Examine Existing Code

If you don't have screen layouts defined in a file, there is still a good possibility that your program has enough structure for a program to

examine your code and generate an ARF file. For example, if the code that prints labels on the screen is always in the same routine, or always starts at a certain line number, you could write a program that examines the PRINT statements in that section and writes, for example, corresponding entries in the ARF to place static text.

```
  PRINT @(10,10)+"BASIS International Ltd.",
```

might be converted to

```
  STATICTEXT 101 "BASIS International Ltd." 10 10 24 1
```

assuming chars as the unit of measure.

Likewise, you could scan for input verbs or called input routines to get the location and length of input fields. Each of these occurrences would generate an ARF EDIT entry, for example:

```
EDIT 101 10 10 24 1
   BEGIN
   INITIALCONTENTS "BASIS International Ltd."
   END
```

Scenario 3 - Read The Screen

Screen Scraping

If neither of these approaches will work for your application, the last alternative is to write a program that can read character-based screens and translate the characters and positions to an ARF file. This program would be CALLed from existing programs. If you use a common input routine, the CALL could be initiated from a hotkey in the input routine. If you don't use a common input routine, you would insert the CALL at the end of the input program that controls the screen you want to read. The object is to read the screen when it's as full of data as possible so you can generate the most accurate ARF file. The exact logic you'll need for this screen-scraping program will vary with the conventions you use for your screens, but some pieces will be universal.

The 'TS' mnemonic will store the characters and display attributes of every position in the scroll region from position 0,0 to the current cursor position. This sample will read 80 columns by 24 rows into the string variable SCREEN$:

```
  PRINT @(80,24),
INPUT @(80,24),'EE','GS'+'TR',SCREEN$,'GE','BE',
```

The information about each position is stored in four bytes, one byte each for background color, foreground color, display attributes, and

the value of the printed character. These bytes might be defined in a string template with the following command:

```
  DIM ATTR$:"BCOLOR:U(1),FCOLOR:U(1),ATTRIBS:U(1),CHAR:C(1)"
```

Then you need to write a loop to evaluate each character and its attributes. For example:

```
  FOR ROW=0 TO 79
FOR COL=0 TO 23
LET ATTR$=A$(1,4); LET A$=A$(5)
REM Routines to analyze one screen position
    ...
NEXT COL
NEXT ROW
```

The details of this loop will vary with your screens. If the top few lines of your screens always include the same information in the same format - company name, date, time, horizontal lines - you might want to skip analyzing these rows. Likewise, if row 24 always contains a user prompt that isn't appropriate for a GUI screen, you will want to skip that row.

The analysis needed for each position, and the logic for building the ARF file, will naturally vary with your screen. If all labels end with colons, then you can treat the text left of colons as labels and the text immediately right of colons as input fields. If your input fields are enclosed in brackets, then the positions between brackets can be counted as input fields and all other text defined as static text in the ARF file.

If your screens distinguish constant text and input data with different colors, underlines, high and low intensity, or some other display attribute, you will need to parse each screen position to figure out how the critical attribute is set. This leads you into each bit checking with the AND() function. Bit checking could fill another article, so I'll summarize by showing you these four tests for display attributes:

```
  REM Check for high intensity
IF ASC(AND(ATTR.ATTRIBS$,$01$))
THEN PRINT "High Intensity"
REM Check for reverse video
IF ASC(AND(ATTR.ATTRIBS$,$02$))
THEN PRINT "Reverse video"
REM Check for underline
IF ASC(AND(ATTR.ATTRIBS$,$04$))
THEN PRINT "Underline"
REM Check for blinking
IF ASC(AND(ATTR.ATTRIBS$,$08$))
THEN PRINT "Blinking"
```

Keep in mind that any screen position can carry more than one, or none, of these attributes.

The following tests will check for the foreground colors blue, green, and red. The tests for background color would be identical except that they would test the variable ATTR.BCOLOR$, using our sample template.

```
  REM Check for blue
IF DEC(AND($03$,ATTR.FCOLOR$)) THEN PRINT "Blue"
REM check for green
IF DEC(AND($0C$,ATTR.FCOLOR$)) THEN PRINT "Green"
REM Check for red
IF DEC(AND($30$,ATTR.FCOLOR$)) THEN print "Red"
```

A color attribute that includes blue and green will be cyan on the screen; blue and red make magenta; green and red make yellow; all three will make white. No color attribute yields black.

Expanding Horizons

After working with ARFs and ResBuilder for a while, you will find other uses for ARFs. For example, you will now have a de facto library of screen layouts, and all of its advantages will be available to you. For example, you could implement global changes to your screens by modifying ARFs with a text editor or simple BBx® program.

ASCII resource files and ResCompiler are powerful tools, and this article describes only a small portion of their potential. The preprocessor capabilities of ResCompiler, alone will give you more flexibility. Any BBx developer with character-based code and customers on Windows platforms should take advantage of these tools.