Tutorial for Adding New Commands to yab

jan_64

August 11, 2006

1 Introduction

yab is based on the yabasic interpreter[1]. On the yabasic website you will find additional sources how to enhance the interpreter (titled "Guide into the Guts of Yabasic"). Nevertheless, this document should give an in-depth introduction too.

You need some basic knowledge about yab, C, C++ and probably the BeAPI for adding new commands to yab. Knowledge about flex and bison are not necessary.

2 Designing a Command

A typical yab command normally is either a procedure (function returning `void`), a function returning a number (`double` or `int`) or a function returning a string (`char*`).

A command consists of

- the command words (lexical entities)
- the command rule
- the C wrapping function calling the C++ method
- the C++ method

This will be discussed in detail in the upcoming sections.

[1] `http://www.yabasic.de`

2.1 The Command Words

The words needed by a command (tokens) are defined in the file `yabasic.flex`. Please introduce only new command words, when the existing are not sufficent to describe your new command. Browse through the file to find all yabasic and yab commands.

A command word consists of the word itself (in capital letters) and the token it returns. The token is most often simply the word itself with a `t` in front of it. Example:

```
BUTTON return tBUTTON;
```

Note: There are exceptions for the token name, e.g. `tGET` represents the command word `GET$` while `tGETNUM` represents `GET`.

New tokens have then to be declared in the file `yabasic.bison` too by simply adding the token name in the token list at the beginning of the file.

2.2 The Command Rule

The actual command rule (grammar) has to be added to the file `yabasic.bison`. If you scroll through the file you get a good idea how the grammar should look like.

Basically three sections in this file are interesting: the section for procedures, the section for numerical functions and the section for string functions. We will investigate the differences of these functions in the next two subsections.

But first we have a look at the similarities. A command rule is written with a leading pipe `|` followed by the tokens and the arguments. At the very end, a C function identifier follows.

Example:

```
| tBUTTON coordinates to coordinates ',' string_expression ','
string_expression ',' string_expression
{add_command(cBUTTON,NULL);}
```

Here, `coordinates` is a substitution for `expression ',' expression` and `to` is a substitution for the command `TO` (which is just a `','` anyway).

So basically there are two types of arguments: `expression` is a number (of type `double`) and `string_expression` is a string (of type `char*`). They

can be separated by commas ’,’. Brackets (’(’ and ’)’) are possible too, but I have not used them often.

Thus, in this example the BUTTON command has 7 arguments, 4 numbers for the coordinates and 3 strings that will contain the own ID, the button text and the view ID.

2.3 Procedures

As all procedures, the mentioned example does not return a value (it is a `void` function in C). It gets an internal identifier named `cBUTTON`. This identifier has to be declared in the file `yabasic.h`. Just check the list of other similar identifiers for the location in the file.

Furthermore, in `yabasic.h` the name of the C function that is added to the file `graphic.c` has to be declared. Procedures always get a `struct command *, YabInterface *` as argument. The command struct contains information about the yab arguments, line number etc. Thus, in `graphic.c` the C function will simply forward these information to the main C++ class (`YabInterface`).

Just before adding the function in `graphic.c` it has to be added in `main.c` too. There, add it to the `switch` that calls the functions according to their identifier. E.g.:

```
      case cBUTTON:
              createbutton(current, yab); DONE;
```

A typical `void` function in `graphic.c` looks like the following example:

```
void createbutton(struct command *cmd, YabInterface* yab)
{
  double x1,y1,x2,y2;
  char *id, *title, *view;

  view = pop(stSTRING)->pointer;
  title = pop(stSTRING)->pointer;
  id = pop(stSTRING)->pointer;
  y2=pop(stNUMBER)->value;
  x2=pop(stNUMBER)->value;
  y1=pop(stNUMBER)->value;
```

```
  x1=pop(stNUMBER)->value;

  yi_SetCurrentLineNumber(cmd->line, (const char*)cmd->lib->s, yab);
  yi_CreateButton(x1,y1,x2,y2, id, title, view, yab);
}
```

In our example, first the 7 yab arguments are retrieved from the `command struct`. Note: the arguments are stored on a stack, so you have to retrieve them in inverse order! Here, e.g. `y1` is retrieved before `x1`. Also, strings and numbers have different `pop` calls.

Numbers can be either `double` or `int` but for coordinates, always `double` should be used.

The current line number is passed on to the `YabInterface` class by calling `yi_SetCurrentLineNumber`. This line is the same for all `void` functions.

Finally, the arguments are passed on to the `YabInterface` class by calling `yi_CreateButton`.

2.4 Functions

Functions that either return a number or a string are implemented differently. First they get a different identifier starting with an `f` e.g. `fLISTBOXGETNUM`. This identifier has to be declared in `yabasic.h` in the `enum functions`. Note: the functions are sorted by the number of their arguments!

Additionaly, the name of the C function that is added to the file `graphic.c` has to be declared in `yabasic.h` too. Differently to procedures, these functions get their argument set immediatly, e.g.:

```
int listboxgetnum(const char*, YabInterface *yab, int line,
                  const char* libname);
```

`listboxgetnum` returns an `int` while taking a string as first argument. The further arguments `YabInterface *yab, int line, const char* libname` have to be added to provide the `YabInterface` class further information.

Other than procedures, the stack retrieval of arguments and the function call are all done in the file `function.c`. There, the yab arguments are retrieved from the stack and forwarded to the wrapper function in `graphic.c`. Example:

```
case fLISTBOXGETNUM:
    str=a1->pointer;
    value = listboxgetnum(str, yab, linenum, current->lib->s);
    result = stNUMBER;
    break;
```

The string argument is retrieved by `a1->pointer`. Numerical arguments are addressed by e.g. `a3->value` (not in this example). The arguments are numbered from `a1` to `a6`. More arguments are currently not supported.

Here, the result is stored as a number. For strings, the result should be stored in `pointer` and `result = stSTRING;` has to be set. Check the other commands in this file for further examples.

Finally, the wrapper function has to be implemented in `graphic.c`. For our example, this looks like the following code:

```
int listboxgetnum(const char* id, YabInterface *yab, int line,
                  const char* libname)
{
        yi_SetCurrentLineNumber(line, libname, yab);
        return yi_ListboxGetNum(id, yab);
}
```

The current line number is passed on to the `YabInterface` class by calling `yi_SetCurrentLineNumber`. This line is the same for all functions. The returned number is then simply passed on by `yi_ListboxGetNum`.

Note: strings have to be copied with `my_strdup`, e.g. `return my_strdup((char*)yi_CheckMessages(yab));` It is up to you that strings still exist in memory when copied!

3 The C++ Class YabInterface

3.1 Adding a Method

After the above described overhead, we are ready to actually write the new method. The method has a C++ name and a wrapper function with an external name starting with `yi_`. Both have to be defined in `YabInterface.h` and implemented in `YabInterface.cpp`.

The wrapper function takes the pointer to the `YabInterface` object and calls the main method:

```
void yi_CreateButton(double x1,double y1,double x2,double y2,
                     const char* id, const char* title,
                     const char* view, YabInterface* yab)
{
  yab->CreateButton(BRect(x1,y1,x2,y2), id, _L(title), view);
}
```

Note the `_L()` macro that is used on all text that should be translated automagically by the ZETA local kit. It will kick in when the `LOCALIZE` command was used. So please use this macro whenever possible, to allow simple localization.

The method itself is part of the `YabInterface` class which is derived from `BApplication`. You can always expect `BApplication` up and running (with the interpreter in an own thread). Thus, all `BApplication` methods are directly accessable from your method.

```
void YabInterface::CreateButton(BRect frame, const char* id,
                          const char* title, const char* view)
{
  // code here
}
```

3.2 Accessing yab Data Structures

yab stores various information in list objects. The probably most wanted list is the list of available views. These are stored in the `YabList` object `viewList`. To initialize a new widget, it is often sufficient to find the parent view. This is done by calling `YabList::GetView(const char*)`:

```
YabView *myView = cast_as((BView*)viewList->GetView(view), YabView);
if(myView)
{
  YabWindow *w = cast_as(myView->Window(), YabWindow);
  if(w)
  {
    w->Lock();
    // initialize widget here
    w->Unlock();
```

```
  }
  else
    ErrorGen("Unable to lock window");
}
else
  Error(view, "VIEW");
```

New widgets should allow some useful layouting. Please refer to the commands for BUTTON and LISTBOX to understand how different kinds of layouting are used in yab.

If you want to find a specific widget on an unknown view, you have to cycle through the views to look for the view containing your widget. Such a loop looks like this (under the condition that `MyWidget` is derived from `BView`):

```
YabView *myView = NULL;
MyWidget *myWidget = NULL;
for(int i=0; i<viewList->CountItems(); i++)
{
  myView = cast_as((BView*)viewList->ItemAt(i), YabView);
  if(myView)
  {
    YabWindow *w = cast_as(myView->Window(), YabWindow);
    if(w)
    {
      w->Lock();
      myWidget = cast_as(myView->FindView(id), MyWidget);
      if(myWidget)
      {
        // do something with myWidget
        w->Unlock();
        return;
      }
      w->Unlock();
    }
  }
}
Error(id, "MyWidget");
```

Note: the `return` is set after something has been done with the widget and after unlocking the window again. This allows the lazy error checking at the end of the loop.

3.3 Some Short Remarks

At the end some short remarks about...

- `BUILD` *macros:* Some commands have `BUILD` macros that allow to disable whole parts of yab on compiling. This is used for the build factory to produce smaller code size. Unused libraries and code parts are simply left out when they are not needed. Make use of these macros whenever you enhance commands that have these macros.
- *Drawing:* Drawing commands are a bit tricky. They offer drawing on a view, on a bitmap and a canvas. Especially the view drawing has a own storage system. Investigate the other drawing commands to understand how they work.
- *Own classes:* Adding own classes is nice, but remember to add this new information in the makefiles (R5 and ZETA!) too.

4 Summary

To give you a checklist of what has to be done in short, have a look at the summary:

- add new command word (token) in `yabasic.flex` if necessary
- add command rule (grammar) in `yabasic.bison`
- add command identifer and C method name in `yabasic.h`
- add command call in either `function.c` or `main.c`
- add wrapper function in `graphic.c`
- add C to C++ wrapper functions in `YabInterface.h` and `YabInterface.cpp`
- add C++ method in `YabInterface.h` and `YabInterface.cpp`