The Evolution Of BBx®

By Scott Amspoker

One look at the first BB1 created by MAI reveals that Business Basic has come very far from its early roots. In those days (the 1970s), Business Basic performed the demanding role of host operating system as well as programming language. When BASIS was formed in 1985, Business Basic took another radical step, which we know as BBx. BBx dropped the role of host operating system and focused on being a cross-platform programming environment that ran under another host operating system.

The evolution continued. Business Basic users learned to expect certain kinds of enhancements from one release to the next. Likewise, Business Basic providers learned to play along-adding mostly minor language enhancements-new verbs, functions, and mnemonics. This "feature hacking" became synonymous with progress. Business Basic remained a monolithic environment: the interpreter, development tools, and file system were an inseparable package. It created its own little world for the programmer to enjoy, while ignoring much of what was happening outside its address space. This problem has been addressed in minor, yet useful, ways by allowing pipes to other processes or simple DLL calls.

Today's real world demands more hybrid solutions. How can the "closed" Business Basic system act as one component of a larger integrated solution? A Business Basic compiler might do the trick. Well, a compiler for Business Basic might be an interesting novelty, but only useful to some. You would give up the convenience of dynamic interpretation without gaining the power of a large, full-featured compiled language. (For a more in-depth discussion of multi-tier development, please see "*Pinatubo*: A More Open Business Basic".)

So, Business Basic developers are having to make tough decisions. It appears that their only choices are to stay with Business Basic or leave it. Well, let's not be so hasty. There is some good news. Interpreters are **IN** again.

One of the hottest language tools today is Java. Java programs are typically translated into a machine-independent byte code and executed by a Java Virtual Machine (JVM) interpreter. Sound familiar? Java also borrows bits and pieces from other languages, such as C/C++, that are practical to implement in a dynamic interpreted environment.

When working with Java, you have two primary options. You can create a stand-alone Java application or a Java "applet" which can only run embedded in an "applet viewer" such as a web browser. Could the monolithic BBx language ever be used as a simple runtime engine? Actually...yes!

Pinatubo - The New BBx Engine

At our April dealer conference in Nashville, I showed BASIS's new Business Basic engine. This technology is part of a release code-named *Pinatubo*. *Pinatubo* represents the first time since the first BBx release that our language interpreter has been completely re-written from scratch. We've learned a lot over the years and many of our future enhancements simply would not be possible without a re-design.The internal token structure is re-designed for faster performance as well as supporting many new language features. Don't panic: existing PRO/5 programs will load into the *Pinatubo* engine just fine.

Rather than a giant executable, conference attendees saw a lean-and-mean Business Basic engine implemented as an opaque C++ object. The purpose of the BB engine is to interpret *Pinatubo's* machine-independent byte code. The engine does not require any particular file system, GUI manager, or front end. Such services are hooked into the engine at runtime as necessary by the larger application. The BB engine can be embedded directly into a C++ application, wrapped up as a Component Object Model (COM) object, used within an event handler, or placed anywhere a developer wishes to use it.

This engine will sit at the core of our next-generation Business Basic products as well as appear in non-traditional places. Not only will this technology enable Basis to respond much faster to changes in the industry, it will allow our developers to create their own solutions as well.

BBx code will be executed without a lot of fuss. The traditional "workspace" concept will all but disappear. Code and data will be dynamically managed. A larger application can invoke entire BBx programs or throw in snippets of BBx code for instant execution. Not only can a C++ program directly invoke BBx code within the same process but BBx code can directly invoke the host C++ code. Multiple threads? No problem. And BBx developers will be able to perform various single-stepping modes, set breakpoints, set variable watches, examine the call stack, and do many other interactive debugging operations. This is possible because the new Business Basic engine supports multiple operating modes to assist in the development/debugging process.

Business Basic Syntax - Modernize the Classic

As long as the shiny new BB engine was being unveiled, we decided to seriously revamp the BBx language as well. Although we didn't show every enhancement, we did give conference attendees a sneak peek at some of the more down-to-earth features.

The biggest attention grabber was the option to work without those annoying line numbers. The *Pinatubo* interpreter supports two syntax modes-"line mode" and "free mode". In free mode, line numbers are no longer a part of the program. They're not hidden-they're simply not there any more. Internally, the only difference

between the two modes is the presence of line numbers. Programs in either mode can peacefully coexist in an application. If the developer wishes, a line mode program can be instantly converted into a free mode program in the blink of an eye.

By removing the physical line as a complete syntactical unit, *Pinatubo* is ready to embrace many modern language constructs. For example,

```
if a=b then
    {
    for x=1 to 10
        {
        print "hello";
        a=a+1;
        }
    }
else
    {
    while a<c
        {
        b=b+a;
        a=a+1;
        }
    }
```

One important difference between line mode and free mode is how the program is edited. In line mode, a program is edited piecemeal by adding and removing lines. In free mode, a program is edited as a single unit via an interactive editor (any editor will do). For example, the *Cabezon* integrated development environment (IDE) has a built-in editor. The programmer can make any number of modifications to the program and then RUN it. Within a split second, the program is tokenized and execution begins. If syntax errors exist, the IDE will take the programmer directly to the offending lines of code. (For more information on *Cabezon*, please see "*Cabezon*, the IDE for Business Basic Developers".)

Some readers will wonder how *Pinatubo* will be able to *resume* execution after a free mode program is modified. We recognize this ability as an indispensable feature of BBx. Rest assured that *Pinatubo* will re-synchronize with a modified free mode program and resume execution from the point where it left off. In fact, it's much more liberal about modifying lines with active GOSUB's than previous versions of BBx.

On a less glamorous note, we showed various small feature enhancements in *Pinatubo*. Many of our customers saw these features last year in our traveling demo tours. Here is a snippet of a (meaningless) line mode program showing some new syntactical touches, including enhancements to comments, variables, assignment operators and string expressions.

```
0010 rem test program
0020 // Note the C++ style comment as an alternative to REM
0030 // Variable names can be up to 200 characters long
0040 aaa: print this_is_a_very_very_looooooong_variable
0050 // radix integer constants from AngelFire and C
0060 let hexval=0x7fff,octval=0o37777,binval=0b11011101
0070 // character integer constants
0080 let charval='w'+123
0090 // C-style assignment operators
0100 let a[x*y]+=4
0110 let a$+=b$
0120 let a[all] += b[all]
0130 // more liberal string and substring expressions
0140 let a$=(b$+c$)(1,10),b$=fid(1)(4,6)(x,y)
```

Getting more serious, we've implemented C++ style exception handling (a la BBx, of course). For starters, individual error numbers can be specified:

```
let a=abs(x,err(41,60,27)=xxx,err=yyy)
```

The above statement traps errors 41, 60, and 27 by branching to label xxx. Other errors branch to label yyy. If something other than a simple branch is desired, a block of BBx code may be used instead:

```
let a=abs(x,err={ print "an error happened"; return })
```

In fact, the block of code for an error handler can be thousands of lines and contain error handlers within it. By default, the error handling code resumes at the next statement following the statement containing the error handling code. Therefore, err={} would be considered an "ignore" operation.

As if that wasn't enough, the global SETERR verb has some new competition. The *Pinatubo* engine implements a trap block (similar to the C++ try/catch feature).

```
trap
    {
myloop:
    read record(1)b$;
    write record(2)b$;
    goto myloop;
    } end={},err=myhandler;
```

In the above code, any untrapped error that occurs inside the "trap" block will be subjected to the error branches at the end of the trap block. Trap blocks can be nested within trap blocks. This is a much better way of catching errors within a range of statements than using SETERR.

Unfortunately, the limited space for this article prevents me from covering everything we showed at the conference (much less things we didn't show). Fortunately for BBx developers, they will soon have a much more powerful BBx language to use. Also, their existing code can be embedded in unimagined hybrid environments. My advice-don't throw that old code away!