Guidelines for writing `ksh-93` built-in commands

David G. Korn

One of the features of `ksh93`, the latest version of `ksh`, is the ability to add built-in commands at run time. This feature only works on operating systems that have the ability to load and link code into the current process at run time. Some examples of the systems that have this feature are Linux, System V Release 4, Solaris, Sun OS, HP-UX Release 8 and above, AIX 3.2 and above, and Microsoft Windows systems.

This memo describes how to write and compile programs that can be loaded into `ksh` at run time as built-in commands.

1. INTRODUCTION

A built-in command is executed without creating a separate process. Instead, the command is invoked as a C function by `ksh`. If this function has no side effects in the shell process, then the behavior of this built-in is identical to that of the equivalent stand-alone command. The primary difference in this case is performance. The overhead of process creation is eliminated. For commands of short duration, the effect can be dramatic. For example, on SUN OS 4.1, the time to run `wc` on a small file of about 1000 bytes, runs about 50 times faster as a built-in command.

In addition, built-in commands may have side effects on the shell environment. This is usually done to extend the application domain for shell programming. For example, there is a group of X-windows extension built-ins that make heavy use of the shell variable namespace. These built-ins are added at run time and result in a windowing shell that can be used to write X-windows applications.

While there are definite advantages to adding built-in commands, there are some disadvantages as well. Since the built-in command and `ksh` share the same address space, a coding error in the built-in program may affect the behavior of `ksh`; perhaps causing it to core dump or hang. Debugging is also more complex since your code is now a part of a larger entity. The isolation provided by a separate process guarantees that all resources used by the command will be freed when the command completes. Resources used by a built-in must be meticulously maintained and freed. Also, since the address space of `ksh` will be larger when built-in are loaded, it may increase the time it takes `ksh` to fork() and exec() non-built-in commands. It makes no sense to add a built-in command that takes a long time to run or that is run only once, since the performance benefits will be negligible. Built-ins that have side effects in the current shell environment have the disadvantage of increasing the coupling between the built-in and `ksh`, making the overall system less modular and more monolithic.

Despite these drawbacks, in many cases extending `ksh` by adding built-in commands makes sense and allows reuse of the shell scripting ability in an application specific domain. This memo describes how to write `ksh` extensions.

2. WRITING BUILT-IN COMMANDS

There is a development kit available for writing `ksh` built-ins as part of the AST (AT&T Software Technology) Toolkit. The development kit has three directories, `include`, `lib`, and `bin`. It is best to set the value of the environment variable `PACKAGE_ast` to the pathname of the directory containing the development kit. The `include` directory contains a sub-directory named `ast` that contains interface prototypes for functions that you can call from built-ins. The `lib` directory contains the `ast` library and a library named `cmd` that contains a version of several of the standard POSIX[1] utilities that can be made run time built-ins. The `lib/ksh` directory contains shared libraries that implement other `ksh` built-ins. The `bin` directory contains build tools such as `nmake`[2]. To add built-ins at runtime, it is necessary to build a shared library containing one or more built-ins that you wish to add. The built-ins are then added by running `builtin –f` *shared_lib*. Since the procedure for building share libraries is system dependent, it is best to use `nmake` using the sample nmake makefile below as a prototype. The AST Toolkit also contains some examples of built-in libraries under the `src/cmd/kshlib` directory.

There are two ways to code adding built-ins. One method is to replace the function `main` with a function `b_`*name*, where *name* is the name of the built-in you wish to define. A built-in command has a calling convention similar to the `main` function of a program, `int main(int argc, char *argv[])`. except that it takes a third argument of type `Shbltin_t*` which can be passed as `NULL` if it is not used. The definition for `Shbltin_t*` is in `<ast/shcmd.h>`. Instead of `exit`, you need to use `return` to terminate your command. The return value will become the exit status of the command. The `open` built-in, installed in `lib/ksh` in the AST Toolkit, uses this method. The `Shbltin_t` structure contains a field named `shp` which is a pointer the the shell data that is needed for `shell` library callbacks. It also contains the fields, `shrun`, `shtrap`, `shexit`, and `shbltin` that are function pointers to the `shell` library functions `sh_run`, `sh_trap sh_exit`, and `sh_addbuiltin`, respectively. These functions can be invoked without the need for runtime symbol lookup when the shell is statically linked with `libshell`.

The alternative method is to create a function `lib_init` and use the `Shbltin_t.shbltin()` function to add one or more built-ins. The `lib_init` function will be called with two arguments. The first argument will be 0 when the library is loaded and the second argument will be of type `Shbltin_t*`. The `dbm_t` and `dss` shell built-ins use this method.

No matter which way you add built-ins you should add the line `SHLIB(`*identifier*`)` as the last line of one of the built-in source file, where *identifier* is any C identifier. This line provides version information to the shell `builtin` command that it uses to verify compatibility between the built-in and `ksh` implementation versions. `builtin` fails with a diagnostic on version mismatch. The diagnostic helps determine whether `ksh` is out of date and requires an upgrade or the built-in is out of date and requires recompilation.

The steps necessary to create and add a run time built-in are illustrated in the following simple example. Suppose you wish to add a built-in command named `hello` which requires one argument and prints the word hello followed by its argument. First, write the following program in the file `hello.c`:

Exhibit 1

```
#include      <stdio.h> int b_hello(int argc, char *argv[], void *context) {
        if(argc != 2)
        {
                fprintf(stderr,"Usage: hello arg\n");
                return(2);
        }
        printf("hello %s\n",argv[1]);
        return(0); } SHLIB(hello)
```

Next, the program needs to be compiled. If you are building with AT&T `nmake` use the following `Makefile`:

Exhibit 2

:PACKAGE: --shared ast hello plugin=ksh :LIBRARY: hello.c and run `nmake install` to compile, link, and install the built-in shared library in `lib/ksh/` under `PACKAGE_ast`. If the built-in extension uses several `.c` files, list all of these on the `:LIBRARY:` line.

Otherwise you will have to compile `hello.c` with an option to pick up the AST include directory (since the AST `<stdio.h>` is required for `ksh` compatibility) and options required for generating shared libraries. For example, on Linux use this to compile:

Exhibit 3

cc -fpic -I$PACKAGE_ast/include/ast -c hello.c and use the appropriate link line. It really is best to use `nmake` because the 2 line Makefile above will work on all systems that have `ksh` installed.

If you have several built-ins, it is desirable to build a shared library that contains them all.

The final step is using the built-in. This can be done with the `ksh` command `builtin`. To load the shared library `libhello.so` from the current directory and add the built-in `hello`, invoke the command,

Exhibit 4

builtin -f ./libhello.so hello The shared library prefix (`lib` here) and suffix (`.so` here) be omitted; the shell will add an appropriate suffix for the system that it is loading from. If you install the shared library in `lib/ksh/`, where `../lib/ksh/` is a directory on **$PATH**, the command

Exhibit 5

builtin -f hello hello will automatically find, load and install the built-in on any system. Once this command has been invoked, you can invoke `hello` as you do any other command. If you are using `lib_init` method to add built-ins then no arguments follow the `-f` option.

It is often desirable to make a command *built-in* the first time that it is referenced. The first time `hello` is invoked, `ksh` should load and execute it, whereas for subsequent invocations `ksh` should just execute the built-in. This can be done by creating a file named `hello` with the following contents:

Exhibit 6

function hello {
    unset -f hello
    builtin -f hello hello
    hello "$@" } This file `hello` needs to be placed in a directory that is in your **FPATH** variable, and the built-in shared library should be installed in `lib/ksh/`, as described above.

3. CODING REQUIREMENTS AND CONVENTIONS

As mentioned above, the entry point for built-ins must either be of the form `b_`*name* or else be loaded from a function named `lib_init`. Your built-ins can call functions from the standard C library, the `ast` library, interface functions provided by `ksh`, and your own functions. You should avoid using any global symbols beginning with **sh_**, **nv_**, and **ed_** since these are used by `ksh` itself. `#define` constants in `ksh` interface files use symbols beginning with `SH_` and `NV_`, so avoid using names beginning with these too.

3.1 Header Files

The development kit provides a portable interface to the C library and to libast. The header files in the development kit are compatible with K&R C[3], ANSI-C[4], and C++[5].

The best thing to do is to include the header file `<shell.h>`. This header file causes the `<ast.h>` header, the `<error.h>` header and the `<stak.h>` header to be included as well as defining prototypes for functions that you can call to get shell services for your builtins. The header file `<ast.h>` provides prototypes for many **libast** functions and all the symbol and function definitions from the ANSI-C headers, `<stddef.h>`, `<stdlib.h>`, `<stdarg.h>`, `<limits.h>`, and `<string.h>`. It also provides all the symbols and definitions for the POSIX[6] headers `<sys/types.h>`, `<fcntl.h>`, and `<unistd.h>`. You should include `<ast.h>` instead of one or more of these headers. The `<error.h>` header provides the interface to the error and option parsing routines defined below. The `<stak.h>` header provides the interface to the memory allocation routines described below.

Programs that want to use the information in `<sys/stat.h>` should include the file `<ls.h>` instead. This provides the complete POSIX interface to `stat()` related functions even on non-POSIX systems.

3.2 Input/Output

`ksh` uses **sfio**, the Safe/Fast I/O library[7], to perform all I/O operations. The **sfio** library, which is part of **libast**, provides a superset of the functionality provided by the standard I/O library defined in ANSI-C. If none of the additional functionality is required, and if you are not familiar with **sfio** and you do not want to spend the time learning it, then you can use `sfio` via the `stdio` library interface. The development kit contains the header `<stdio.h>` which maps `stdio` calls to `sfio` calls. In most instances the mapping is done by macros or inline functions so that there is no overhead. The man page for the `sfio` library is in an Appendix.

However, there are some very nice extensions and performance improvements in `sfio` and if you plan any major extensions I recommend that you use it natively.

3.3 Error Handling

For error messages it is best to use the `ast` library function `errormsg()` rather that sending output to `stderr` or the equivalent `sfstderr` directly. Using `errormsg()` will make error message appear more uniform to the user. Furthermore, using `errormsg()` should make it easier to do error message translation for other locales in future versions of `ksh`.

The first argument to `errormsg()` specifies the dictionary in which the string will be searched for translation. The second argument to `errormsg()` contains that error type and value. The third argument is a *printf* style format and the remaining arguments are arguments to be printed as part of the message. A new-line is inserted at the end of each message and therefore, should not appear as part of the format string. The second argument should be one of the following:

`ERROR_exit(`*n*`)`: If *n* is not-zero, the builtin will exit value *n* after printing the message.

`ERROR_system(`*n*`)`: Exit builtin with exit value *n* after printing the message. The message will display the message corresponding to `errno` enclosed within `[ ]` at the end of the message.

`ERROR_usage(`*n*`)`: Will generate a usage message and exit. If *n* is non-zero, the exit value will be 2. Otherwise the exit value will be 0.

`ERROR_debug(`*n*`)`: Will print a level *n* debugging message and will then continue.

`ERROR_warn(`*n*`)`: Prints a warning message. *n* is ignored.

3.4 Option Parsing

The first thing that a built-in should do is to check the arguments for correctness and to print any usage messages on standard error. For consistency with the rest of `ksh`, it is best to use the `libast` functions `optget()` and `optusage()`for this purpose. The header `<error.h>` includes prototypes for these functions. The `optget()` function is similar to the System V C library function `getopt()`, but provides some additional capabilities. Built-ins that use `optget()` provide a more consistent user interface.

The `optget()` function is invoked as

Exhibit 7

int optget(char **argv*[], const char **optstring*) where `argv` is the argument list and `optstring` is a string that specifies the allowable arguments and additional information that is used to format *usage* messages. In fact a complete man page in `troff` or `html` can be generated by passing a usage string as described by the `getopts` command. Like `getopt()`, single letter options are represented by the letter itself, and options that take a string argument are followed by the `:` character. Option strings have the following special characters:

`:` Used after a letter option to indicate that the option takes an option argument. The variable `opt_info.arg` will point to this value after the given argument is encountered.

`#` Used after a letter option to indicate that the option can only take a numerical value. The variable `opt_info.num` will contain this value after the given argument is encountered.

`?` Used after a `:` or `#` (and after the optional `?`) to indicate the the preceding option argument is not required.

`[...]` After a `:` or `#`, the characters contained inside the brackets are used to identify the option argument when generating a *usage* message.

space The remainder of the string will only be used when generating usage messages.

The `optget()` function returns the matching option letter if one of the legal option is matched. Otherwise, `optget()` returns

`':'` If there is an error. In this case the variable `opt_info.arg` contains the error string.

`0` Indicates the end of options. The variable `opt_info.index` contains the number of arguments processed.

`'?'` A usage message has been required. You normally call `optusage()` to generate and display the usage message.

The following is an example of the option parsing portion of the `wc` utility.

Exhibit 8

```
#include <shell.h> while(1) switch(n=optget(argv,"xf:[file]")) {     case 'f':          file =
opt_info.arg;          break;      case  ':':          error(ERROR_exit(0),
opt_info.arg);         break;      case  '?':          error(ERROR_usage(2),
opt_info.arg);        break; }
```

3.5 Storage Management

It is important that any memory used by your built-in be returned. Otherwise, if your built-in is called frequently, `ksh` will eventually run out of memory. You should avoid using `malloc()` for memory that must be freed before returning from you built-in, because by default, `ksh` will terminate you built-in in the event of an interrupt and the memory will not be freed.

The best way to to allocate variable sized storage is through calls to the **stak** library which is included in **libast** and which is used extensively by `ksh` itself. Objects allocated with the `stakalloc()` function are freed when you function completes or aborts. The **stak** library provides a convenient way to build variable length strings and other objects dynamically. The man page for the **stak** library is contained in the Appendix.

Before `ksh` calls each built-in command, it saves the current stack location and restores it after it returns. It is not necessary to save and restore the stack location in the `b_` entry function, but you may want to write functions that use this stack are restore it when leaving the function. The following coding convention will do this in an efficient manner:

Exhibit 9

```
yourfunction() {
        char        *savebase;
        int         saveoffset;
        if(saveoffset=staktell())
           savebase = stakfreeze(0);
        ...
        if(saveoffset)
           stakset(savebase,saveoffset);
        else
           stakseek(0); }
```

4. CALLING `ksh` SERVICES

Some of the more interesting applications are those that extend the functionality of `ksh` in application specific directions. A prime example of this is the X-windows extension which adds builtins to create and delete widgets. The **nval** library is used to interface with the shell name space. The **shell** library is used to access other shell services.

4.1 The nval library

A great deal of power is derived from the ability to use portions of the hierarchal variable namespace provided by `ksh-93` and turn these names into active objects.

The **nval** library is used to interface with shell variables. A man page for this file is provided in an Appendix. You need to include the header `<nval.h>` to access the functions defined in the **nval** library. All the functions provided by the **nval** library begin with the prefix `nv_`. Each shell variable is an object in an associative table that is referenced by name. The type `Namval_t*` is pointer to a shell variable. To operate on a shell variable, you first get a handle to the variable with the `nv_open()` function and then supply the handle returned as the first argument of the function that provides an operation on the variable. You must call `nv_close()` when you are finished using this handle so that the space can be freed once the value is unset. The two most frequent operations are to get the value of the variable, and to assign value to the variable. The `nv_getval()` returns a pointer the the value of the variable. In some cases the pointer returned is to a region that will be overwritten by the next `nv_getval()` call so that if the value isn't used immediately, it should be copied. Many variables can also generate a numeric value. The `nv_getnum()` function returns a numeric value for the given variable pointer, calling the arithmetic evaluator if necessary.

The `nv_putval()` function is used to assign a new value to a given variable. The second argument to `putval()` is the value to be assigned and the third argument is a *flag* which is used in interpreting the second argument.

Each shell variable can have one or more attributes. The `nv_isattr()` is used to test for the existence of one or more attributes. See the appendix for a complete list of attributes.

By default, each shell variable passively stores the string you give with with `nv_putval()`, and returns the value with `getval()`. However, it is possible to turn any node into an active entity by assigning functions to it that will be called whenever `nv_putval()` and/or `nv_getval()` is called. In fact there are up to five functions that can associated with each variable to override the default actions. The type `Namfun_t` is used to define these functions. Only those that are non-`NULL` override the default actions. To override the default actions, you must allocate an instance of `Namfun_t`, and then assign the functions that you wish to override. The `putval()` function is called by the `nv_putval()` function. A `NULL` for the *value* argument indicates a request to unset the variable. The *type* argument might contain the `NV_INTEGER` bit so you should be prepared to do a conversion if necessary. The `getval()` function is called by `nv_getval()` value and must return a string. The `getnum()` function is called by by the arithmetic evaluator and must return double. If omitted, then it will call `nv_getval()` and convert the result to a number.

The functionality of a variable can further be increased by adding discipline functions that can be associated with the variable. A discipline function allows a script that uses your variable to define functions whose name is *varname*`.`*discname* where *varname* is the name of the variable, and *discname* is the name of the discipline. When the user defines such a function, the `settrap()` function will be called with the name of the discipline and a pointer to the parse tree corresponding to the discipline function. The application determines when these functions are actually executed. By default, `ksh` defines `get`, `set`, and `unset` as discipline functions.

In addition, it is possible to provide a data area that will be passed as an argument to each of these functions whenever any of these functions are called. To have private data, you need to define and allocate a structure that looks like

Exhibit 10

```
struct yours {
        Namfun_t  fun;          your_data_fields; };
```

4.2 The shell library

There are several functions that are used by `ksh` itself that can also be called from built-in commands. The man page for these routines are in the Appendix.

The `sh_addbuiltin()` function can be used to add or delete builtin commands. It takes the name of the built-in, the address of the function that implements the built-in, and a `void*` pointer that will be passed to this function as the third agument whenever it is invoked. If the function address is `NULL`, the specified built-in will be deleted. However, special built-in functions cannot be deleted or modified.

The `sh_fmtq()` function takes a string and returns a string that is quoted as necessary so that it can be used as shell input. This function is used to implement the `%q` option of the shell built-in `printf` command.

The `sh_parse()` function returns a parse tree corresponding to a give file stream. The tree can be executed by supplying it as the first argument to the `sh_trap()` function and giving a value of `1` as the second argument. Alternatively, the `sh_trap()` function can parse and execute a string by passing the string as the first argument and giving `0` as the second argument.

The `sh_isoption()` function can be used to set to see whether one or more of the option settings is enabled.

REFERENCES

1. *POSIX − Part 2: Shell and Utilities*, IEEE Std 1003.2-1992, ISO/IEC 9945-2:1993.
2. Glenn Fowler, *A Case for make*, Software - Practice and Experience, Vol. 20 No. S1, pp. 30-46, June 1990.
3. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1978.
4. American National Standard for Information Systems − Programming Language − C, ANSI X3.159-1989.
5. Bjarne Stroustroup, *C++*, Addison Wesley, xxxx
6. *POSIX − Part 1: System Application Program Interface*, IEEE Std 1003.1-1990, ISO/IEC 9945-1:1990.
7. David Korn and Kiem-Phong Vo, *SFIO - A Safe/Fast Input/Output library*, Proceedings of the Summer Usenix, pp. , 1991.