# Korn Shell Custom Builtins

***Finnbarr P. Murphy***
***(fpm@un-ix.com)***

The majority of UNIX shells are not designed for extensibility or embeddability. The current exception is the 1993 version of the Korn shell, written by David Korn and generally referred to as ***ksh93,*** which includes support for runtime linking of libraries and custom builtins and accessing shell internals.

It is very difficult, however, to find good information or examples of how to implement ***ksh93*** custom builtins. The source code to ***ksh93*** has virtually no comments and the supplied documentation is extremely terse and often conflicts with other parts of the documentation or the source code itself.

This article is an attempt to show by example how to write your own ***ksh93*** custom builtins. You are expected to be reasonably proficient in the C language and the use of the ***gcc*** compiler/linker. However, before we start, it is important to note that custom builtins can only be implemented on operating systems that support dynamic loading of shared objects into the current running process since, internally, a custom builtin is invoked as a C routine by ***ksh93.*** Fortunately most modern operating systems provide this feature via the ***dlopen(), dlsym(), dlerror()*** and ***dlclose()*** APIs.

Why bother implementing ***ksh93*** custom builtins? The answer lies in fact that custom builtins are inherently much faster and require less system resources than an equivalent routine which uses other standalone commands and utilities. A custom builtin executes in the same process as the shell, i.e. it does not create a separate sub-process using ***fork()*** and ***exec()***. Thus a significant improvement in performance can occur since the process creation overhead is eliminated. The author of ***ksh93***, Dave Korn, reported that on a SUN OS 4.1 the time to run ***wc*** on a file of about 1000 bytes was about 50 times less when using the ***wc*** built-in command.

In most cases ***ksh93*** comes with the ***libcmd*** library which contains the custom builtins as listed in the following table:

| | | | | | | |
|---|---|---|---|---|---|---|
| ***basename*** | ***cmp*** | ***dirname*** | ***head*** | ***mkdir*** | ***rev*** | ***Tee*** |
| ***Cat*** | ***comm*** | ***expr*** | ***Id*** | ***mkfifo*** | ***rm*** | ***Tty*** |
| ***chgrp*** | ***Cp*** | ***fmt*** | ***Join*** | ***mv*** | ***rmdir*** | ***Uname*** |
| ***chmod*** | ***cut*** | ***fold*** | ***Ln*** | ***paste*** | ***sty*** | ***Uniq*** |
| ***chown*** | ***date*** | ***getconf*** | ***logname*** | ***pathchk*** | ***tail*** | ***Wc*** |

The above custom builtins were written so as to have no side effects on ***ksh93*** or its environment, and are identical to that of an equivalent stand-alone command. Using these builtins has the advantage that startup time is significantly reduced and shell internals are accessable.

Custom builtins that have side effects on ***ksh93*** or its environment can be also written. This is usually done to extend an application domain. For example, there are two extensions to ***ksh93*** that can be used to write GUI applications as shell script. One is ***dtksh*** (Desktop Korn Shell) which was written by Steve Pendergrast at Novell and is included with the Common Desktop Environment, CDE. The other is ***tksh*** which was written by Dave Korn's son, Jeffrey. ***tksh*** implements the ***tcl*** scripting language as an extension to ***ksh93*** so that both ***tcl*** and ***ksh*** scripts can run in the same address space.

There are two ways to create and install ***ksh93*** custom builtins. In both cases, the custom builtin is loaded into ***ksh93*** using the ***builtin*** command. Which method you use is entirely up to you. The easiest way is to write a shared library containing one or more functions whose names are ***b_xxxx*** where ***xxxx*** is the name of the custom builtin. The function ***b_xxxx*** takes three arguments. The first two are the same as for the main() function in a C program. The third argument is a pointer which points to the current shell context. The second way is to write a shared library containing a function named ***lib_init()***. This function is called with an argument of 0 when the shared library is loaded. This function can add custom builtins with the ***sh_addbuiltin()*** function.

I believe that the best way to learn about a new feature is to actually write code which uses the new feature. Following are a number of examples which demonstrate how to write custom builtins, starting with a few relatively simple examples and then some more complex examples which access and modify ***ksh93*** internals. All examples were written and tested using ***ksh93*** version M 93s+ 2008-01-31 and CentOS 5.0 but should compile and work on any modern UNIX or Linux operating system.

## Example 1

Suppose you wish to write a simple custom builtin called ***hello*** which takes one argument <***string***> and outputs "***hello there*** <***string***>" to ***stdout***.

```
/*
 *  Example 1 – hello.   Based on a published example by David
Korn
 *
 */

#include <stdio.h>

int
b_hello(int argc, char *argv[], void *extra)
{
    if (argc != 2) {
         fprintf(stderr,"Usage: hello arg\n");
         return(2);
    }

    printf("Hello there %s\n",argv[1]);
    return(0);
```

```
}
```

Next compile ***hello.c*** and create a shared library ***libhello.so*** containing ***hello***:

```
$ gcc -fPIC -g -c hello.c
$ gcc -shared -W1,-soname,libhello.so -o libhello.so hello.o –lc
```

Some operating systems (Solaris Intel for example) do not require you to build a shared library and support the direct loading of ***hello.o***. However the majority of operating systems require you to create a shared library as we have done for this example. Note the use of the ***–fPIC*** flag to indicate position independent code should be produced. Unlike relocatable code, position independent code can be copied to any memory location without modification and executed.

To actually use the ***hello*** custom builtin, you must make it available to ***ksh93*** using the ***ksh93 builtin*** command**.**

```
$ builtin -f ./libhello.so hello
```

If you are unfamiliar with the ***builtin*** command, you can type ***builtin –man*** or ***builtin –help*** for more information or read the ***ksh93*** man page.

You can then use the ***hello*** custom builtin just like you would use any other command or shell feature:

```
$ hello joe
Hello there joe
$ hello "joe smith"
Hello there joe smith
$ hello
Usage: hello arg
$
```

Note that the ***hello*** custom builtin will show up when you list builtins using the ***builtin*** command

```
$ builtin
  …..
  hello
  ……
$
```

but not when you list special builtins using the ***builtin –s*** option.

To remove the ***hello*** builtin, use the ***builtin –d*** option:

```
$ builtin –d  hello
$ hello joe
/bin/ksh93: hello: not found [No such file or directory]
$
```

Removing a custom builtin does not necessarily release the associated shared library.

Internally ***hello*** named ***b_hello()*** and takes 3 arguments. As previously discussed custom builtins are generally required to start with "***b_***" (There is an exception which will be discussed in a later example.) The arguments ***argc*** and ***argv*** act just like in a ***main()*** function. The third argument is the current ***context*** of ***ksh93*** and is generally not used as another mechanism, ***sh_getinterp(),*** is provided to access the current content.

Instead of ***exit***, use ***return*** to terminate a custom builtin. The return value becomes the exit status of the builtin and can be queried using ***$?*** A return value of 0 indicates success with > 0 indicating failure. If you allocate any resources such as memory, all such resources used must be carefully freed before terminating the custom builtin.

Custom builtins can call functions from the standard C library, the AST (Advanced Software Technology) ***libast*** library, interface functions provided by ***ksh93***, and your own C libraries. You should avoid using any global symbols beginning with ***sh_***, ***.nv_***, and ***ed_*** or ***BSH_*** since these are reserved for use by ***ksh93*** itself.

If you move ***libhello.so*** to where the shared libraries normally reside for your particular operating system, typically ***/usr/lib***, you can load the ***hello*** custom builtin as follows

```
$ builtin –f hello hello
```

as ***ksh93*** automatically adds a ***lib*** prefix and ***.so*** suffix to the name of the library specified using the ***builtin –f*** option.

It is often desirable to automatically load a custom builtin the first time that it is referenced. For example, the first time the custom builtin ***hello*** is invoked, ***ksh93*** should load and execute it, whereas for subsequent invocations ***ksh93*** should just execute the ***hello*** custom builtin. This can be done by creating a file named ***hello*** as follows:

```
function hello
{
        unset -f hello
        builtin -f /home/joe/libhello.so hello
        hello "$@"
}
```

This file must to be placed in a directory that is in your FPATH environmental variable. In addition, the full pathname to the shared library containing the ***hello*** custom builtin should be specified so that the run time loader can find this shared library no matter where ***hello*** is invoked.

There are alternative ways to locating and invoking builtins using a ***.paths*** file. See the ksh93 man page for further information.

## Example 2

Here is another simple example of a custom builtin. This custom builtin uppercases the first character of the string argument.

```
/*
 * Example 2 – firstcap.  Uppercase first character of string
 *
 */

#include <stdio.h>
#include <ctype.h>

int
b_firstcap(int argc, char *argv[], void *extra)
{
    int c;
    char *s;

    if (argc != 2) {
         fprintf(stderr,"Usage: firstcap arg\n");
         return(2);
    }

    s = argv[1];
    c = *s++;

    printf("%c%s\n", toupper(c), s);

    return(0);
}
```

Assuming you created a library called ***libfirstcap.so*** and placed this library in the default directory for shared libraries you can load and use this custom builtin as follows.

```
$ builtin -f firstcap firstcap
$ firstcap joe
Joe
$ firstcap united
United
$
```

## Example 3

This example extends the previous examples in a number of ways. First of all the AST header **<*shell.h*>** is included. Life as a custom builtin developer is much easier when this header is included. It ensures that other necessary AST headers are included so that C prototypes are provided and calls to ***stdio*** routines are re-mapped to use the equivalent but safer ***sfio*** (AST Safe Fast Input Output) routines. The maintainers of ***ksh93*** regard the ***stdio*** routines as having too many weaknesses to be used safely by ***ksh93***.

Second, if your shared library contains a function named ***lib_init()***, this function is invoked with argument value of 0 when the shared library is loaded. As previously discussed, ***lib_init()*** can load builtins using ***sh_addbuiltin()***. In this case there is no restriction on the name of a custom builtin function name. Note that in the current version of ***ksh93***, ***lib_init()*** can take a second argument, ***void *context***, but this seems to do nothing.

```
/*
 *  Example 3 – hello, goodbye.  Use <shell.h> and lib_init
 *
 */

#include <ast/shell.h>

int
fpm_goodbye(int argc, char *argv[], void *extra)
{
    if (argc != 2) {
         printf("Usage: goodbye arg\n");
         return(2);
    }

    printf("Goodbye %s\n", argv[1]);

    return(0);
}

int
b_hello(int argc, char *argv[], void *extra
{
    if (argc != 2) {
         printf("Usage: hello arg\n");
         return(2);
    }

    printf("Hello %s\n", argv[1]);

    return(0);
}

void
lib_init(int c, void *context)
{
    /* automatically load goodbye when library is loaded */
    sh_addbuiltin("goodbye", fpm_goodbye, 0);
}
```

Assuming you built a library called ***libhello.so*** which contains the above code, and placed this library in the shared library location, you can access the two custom builtins, ***hello*** and ***goodbye*** as follows:

```
$ builtin –f hello hello
$ hello Joe
Hello Joe
```

```
$ goodbye Joe
Goodbye Joe
$
```

To remove both custom builtins at the same time you must specify both on the command line as follows

```
$ builtin –d hello –d goodbye
$
```

Note that ***goodbye*** was not named internally as ***b_goodbye*** as in previous examples but as ***fpm_goodbye***. The "***b_***" function prefix naming requirement is relaxed when a custom builtin is loaded via ***lib_int()*** and ***sh_addbuiltin().***

## Example 4

If for some reason you do not wish to use the AST ***sfio*** routines, you should include the header ***<shcmd.h>*** and then call ***LIB_INIT(context)*** which will initialize the shell context so that ***ksh93*** knows that you are not using the ***sfio*** routines.

```
/*
 *  Example 4 – hello, goodbye.  Use <shcmd.h> and LIB_INIT
 *
 */

#include <ast/shcmd.h>
#include <stdio.h>

int
b_goodbye(int argc, char *argv[], void *extra)
{
    if (argc != 2) {
         printf("Usage: goodbye arg\n");
         return(2);
    }

    printf("Goodbye %s\n", argv[1]);

    return(0);
}

int
b_hello(int argc, char *argv[], void *extra)
{
    if (argc != 2) {
         printf("Usage: hello arg\n");
         return(2);
    }

    printf("Hello %s\n", argv[1]);

    return(0);
```

```
}

void
lib_init(int c, void *context)
{
    LIB_INIT(context);

    sh_addbuiltin("hello", b_hello, 0);
    sh_addbuiltin("goodbye", b_goodbye, 0);
}
```

Assuming that you build a shared library called ***libhello.so***, both custom builtins are automatically loaded when the shared library is loaded i.e.

```
$ builtin –f libhello.so
```

## Example 5

One of the advantages of ***ksh93*** over other shells is the fact that online help is provided for most commands using ***-?***, ***–man*** or ***–help***. Self-documenting code is supported in custom builtins but is very poorly documented. All custom builtins can generate their own manual page in several formats. While this documentation takes up space in the shared library and loaded image, the benefits outweigh the cost in my opinion.

The following example shows how to include built-in help and documentation for a custom builtin.

```
#include <ast/shell.h>

#define SH_DICT "libgoodbye"

static const char usage_goodbye[] =
    "[-?\n@(#)$Id: goodbye 2008-04-06 $\n]"
    "[-author?Finnbarr P. Murphy <fpmATun-ixDOTcom>]"
    "[-licence?http://www.opensource.org/licenses/cpl1.0.txt]"
    "[+NAME?goodbye - output goodbye message]"
    "[+DESCRIPTION?\bgoodbye\b outputs either a short or
extended"
    "message to stdout.]"
    "[x:xtended?Output extended message]"
    "\n"
    "\nname\n"
    "\n"
    "[+EXIT STATUS?] {"
         "[+0?Success.]"
         "[+>0?An error occurred.]"
    "}"
    "[+SEE ALSO?\bprint\b(1), \becho\b(1)]"
;

int
b_goodbye (int argc, char *argv[], void *extra)
```

```
{
     register int n, extend=0;

     while (n = optget(argv, usage_goodbye)) switch(n) {
          case 'x':
              extend=1;
              break;
          case ':':
              error(2, "%s", opt_info.arg);
              break;
          case '?':
              errormsg(SH_DICT,
                       ERROR_usage(2), "%s", opt_info.arg);
              break;
      }

      argc -= opt_info.index;
      argv += opt_info.index;

      if (argc != 1)
           errormsg(SH_DICT,
                    ERROR_usage(2), "%s", optusage((char *)0));

      if (extend)
           sfprintf(sfstdout, "Goodbye for now %s\n", *argv);
      else
          sfprintf(sfstdout, "Goodbye %s\n", *argv);

       return(0);
}

void
lib_init(int c, void *context)
{
     sh_addbuiltin("goodbye", b_goodbye, 0);
}
```

*Example 6*

To prevent memory leaks in your custom builtins you should avoid using ***malloc()*** and ***calloc()*** and similar routines. The preferred way to obtain, free and manage memory space when required in a custom builtin is to use the ***libast stk(3)*** routines. Memory leaks can arise when a custom builtin does not free all of its allocated memory upon return or is interrupted by a signal such as SIGINT before it can do so.

This example shows how to encrypt a string using a numeric key. It uses a relatively simple XOR operation to encrypt the string and is commonly known as the Vernam cipher or XOR encryption. Since it requires memory space to perform this operation, it uses the ***stkcopy()*** routine to get the required space. This memory is guaranteed to be freed by ***libast*** when the custom builtin exits. No additional code is required to free up the allocated memory.

```
/*
```

```
 * Example 6 – strcrypt
 */

#include <shell.h>
#include <stk.h>

#define SH_DICT "strcrypt"

static const char usage_strcrypt[] =
   "[-?\n@(#)$Id: strcrypt 2008-05-04 $\n]"
   "[-author?Finnbarr P. Murphy <fpmAThotmailDOTcom>]"
   "[-licence?http://www.opensource.org/licenses/cpl1.0.txt]"
   "[+NAME?strcrypt - encrypt string using numeric key]"
   "[+DESCRIPTION?\bencrypt\b a string using a numeric key.
        Note uses XOR to do encryption. Only works when
        numeric key used. Use same numeric key to decrypt.]"
   "[+OPTIONS?none.]"
   "\n"
   "\nstring key\n"
   "\n"
   "[+EXIT STATUS?] {"
        "[+0?Success.]"
        "[+>0?An error occurred.]"
   "}"
   "[+SEE ALSO?\bcrypt\b(2)]"
;

int
b_strcrypt(int argc, char *argv[], void *extra)
{
    int i, sl, pl, c;
    char *v, *s, *p;

    while (i = optget(argv, usage_strcrypt)) switch(i) {
         case ':':
           error(2, "%s", opt_info.arg);
           break;
         case '?':
           errormsg(SH_DICT, ERROR_usage(2), "%s", opt_info.arg);
           break;
    }
    argc -= opt_info.index;
    argv += opt_info.index;

    if (argc != 2)
        errormsg(SH_DICT, ERROR_usage(2), "%s", optusage((char *)0));

    s  = argv[0];
    sl = strlen(s);
    p  = argv[1];
    pl = strlen(p);

    /* copy string onto stack so it can be modified */
    if (!(v = (char *)stkcopy(stkstd, s)))
        error(3, "stkcopy failed");

    for (i = 0; i < sl; i++) {
```

```
            c = s[i] ^ toupper(p[i%pl]);
            if (c != 0)
               v[i] = (char)c;
        }

        sfprintf(sfstdout,"%s", v);

        return(0);
    }
```

With this custom builtin, a string of text (argument 1) is encrypted using the supplied numeric key (argument 2). To decrypt the encrypted string, simply invoke the custom builtin again using the encrypted string and same numeric key.

```
$ strcrypt “hello” 1478
YQ[T^
$ strcrypt “YQ[T^“ 1478
hello
$
```

Note that ***libast*** also contains a set of routines whose names start with ***stak*** which provide similar functionality. However the ***stak(3)*** routines are marked deprecated and should not be used in custom builtins.

*Example 7*

This example demonstrates a different approach to returning information from a custom builtin. The purpose of this custom builtin is to set the value of a specified variable (argument 1) to the size of the specified file (argument 2).

```
$ statsize myfilesize /usr/bin/ksh
$ print ${myfilesize}
1157295
$
```

This custom builtin uses the ***libast nval(3)*** name-value routines to access the internals of ***ksh93*** and set up a variable with the specifed name (i.e. ***myfilesize***) and value of ***1157295*** as returned by ***stat(2)*** for the file ***/usr/bin/ksh***. See the ***nval(3)*** documentation for more information.

```
/*
 * Example 7 - statsize
 */

#include <shell.h>
#include <nval.h>

#define SH_DICT "statsize"

static const char usage_statsize[] =
   "[-?\n@(#)$Id: stat 2008-05-03 $\n]"
   "[-author?Finnbarr P. Murphy <fpmAThotmailDOTcom>]"
```

```
    "[-licence?http://www.opensource.org/licenses/cpl1.0.txt]"
    "[+NAME?statsize - assign size of file to variable]"
    "[+DESCRIPTION?\bstat\b assigns the size of the specified file"
         "(in bytes) to the specified variable.]"
    "[+OPTIONS?none.]"
    "\n"
    "[+EXIT STATUS?] {"
         "[+0?Success.]"
         "[+>0?An error occurred.]"
    "}"
    "[+SEE ALSO?\bstat\b(2)]"
;

int
b_statsize(int argc, char *argv[], void *extra)
{
    Namval_t *nvp = (Namval_t *)NULL;
    Shell_t *shp = (Shell_t *)NULL;
    struct stat st;
    long d;
    register int n;

    while (n = optget(argv, usage_statsize)) switch(n) {
         case ':':
           error(2, "%s", opt_info.arg);
           break;
         case '?':
           errormsg(SH_DICT, ERROR_usage(2), "%s", opt_info.arg);
           break;
    }
    argc -= opt_info.index;
    argv += opt_info.index;

    if (argc != 2)
        errormsg(SH_DICT, ERROR_usage(2), "%s", optusage((char*)0));

    /* get current shell context */
    shp = sh_getinterp();

    /* retrieve information about file */
    stat(argv[1], &st);
    /* assign size of file to long */
    d = (long) st.st_size;

    /* access the variables tree and add specified variable */
    nvp = nv_open(argv[0], shp->var_tree,
                  NV_NOARRAY|NV_VARNAME|NV_NOASSIGN);
    if (!nv_isnull(nvp))
        nv_unset(nvp);
    nv_putval(nvp, (char *)&d, NV_INTEGER|NV_RDONLY);
    nv_close(nvp);

    return(0);
}
```

This custom builtin could easily be extended to provide much more information about a file using command line options. I will leave it up to you, the reader, to do this.

## Example 8

This example also uses the ***nval(3)*** routines to access ***ksh93*** internals and print out more information about the specified shell variable.

```
$ showvar HOME
Value: /home/fpm, Flags: 12288 NV_EXPORT NV_IMPORT
$ integer i=123
$ showvar i
Value: 123, Flags: 10 NV_UINT64 NV_UTOL
```

See the ***libast*** header **<*nval.h*>** for more information about the different flags which can be associated with each variable. Note that some flags are overloaded so that they mean different things according to how they are OR'ed with other flags.

```
/*
 * Example 8 – showvar
 */

#include <shell.h>
#include <nval.h>

#define SH_DICT "showvar"

static const char usage_showvar[] =
   "[-?\n@(#)$Id: showvar 2008-05-04 $\n]"
   "[-author?Finnbarr P. Murphy <fpmAThotmailDOTcom>]"
   "[-licence?http://www.opensource.org/licenses/cpl1.0.txt]"
   "[+NAME?showvar - display variable details]"
   "[+DESCRIPTION?\bshowvar\b displays details about the
                  specified variable.]"
   "[+OPTIONS?none.]"
   "\n"
   "\nvariable_name\n"
   "\n"
   "[+EXIT STATUS?] {"
        "[+0?Success.]"
        "[+>0?An error occurred.]"
   "}"
   "[+SEE ALSO?\bstat\b(2)]"
;

struct Flag {
   int flag;
   char *name;
};

/* Note: not a complete list of all possible flags */
struct Flag Flags[] = {
   NV_ARRAY,   "NV_ARRAY",
   NV_BINARY,  "NV_BINARY",
```

```
    NV_EXPORT,  "NV_EXPORT",
    NV_HOST,    "NV_HOST",
    NV_IMPORT,  "NV_IMPORT",
    NV_LJUST,   "NV_LJUST",
    NV_LTOU,    "NV_LTOU",
    NV_RAW,     "NV_RAW",
    NV_RDONLY,  "NV_RDONLY",
    NV_REF,     "NV_REF",
    NV_RJUST,   "NV_RJUST",
    NV_TABLE,   "NV_TABLE",
    NV_TAGGED,  "NV_TAGGED",
    NV_UTOL,    "NV_UTOL",
    NV_ZFILL,   "NV_ZFILL",
    0,          (char *)NULL
};

struct Flag IntFlags[] = {
    NV_LTOU|NV_UTOL|NV_INTEGER,   "NV_UINT64",
    NV_LTOU|NV_RJUST|NV_INTEGER,  "NV_UINT16",
    NV_RJUST|NV_ZFILL|NV_INTEGER, "NV_FLOAT",
    NV_UTOL|NV_ZFILL|NV_INTEGER,  "NV_LDOUBLE",
    NV_RJUST|NV_INTEGER,          "NV_INT16",
    NV_LTOU|NV_INTEGER,           "NV_UINT32",
    NV_UTOL|NV_INTEGER,           "NV_INT64",
    NV_RJUST,                     "NV_SHORT",
    NV_UTOL,            "NV_LONG",    /* long long /long double */
    NV_LTOU,            "NV_UNSIGN",
    NV_ZFILL,           "NV_DOUBLE",  /* floating point */
    NV_LJUST,           "NV_EXPNOTE", /* scientific notation */
    NV_INTEGER,         "NV_INT32(NV_INTEGER)",
    0,                            (char *)NULL
};

int
b_showvar(int argc, char *argv[], void *extra)
{
    Shell_t *shp = (Shell_t *)NULL;
    Namval_t *nvp = (Namval_t *)NULL;
    char *ptr = (char *)NULL;
    int i;

    while (i = optget(argv, usage_showvar)) switch(i) {
         case ':':
           error(2, "%s", opt_info.arg);
           break;
         case '?':
           errormsg(SH_DICT, ERROR_usage(2), "%s", opt_info.arg);
           break;
    }
    argc -= opt_info.index;
    argv += opt_info.index;

    if (argc != 1)
        errormsg(SH_DICT, ERROR_usage(2), "%s", optusage((char*)0));

    /* get current shell context */
    shp = sh_getinterp();
```

```
    if ((nvp = nv_search(*argv, shp->var_tree, 0)) == NULL) {
         errormsg(SH_DICT, ERROR_exit(1),
                      "%s: variable not found", *argv);
        return(1);
    }

    if ((ptr = nv_getval(nvp)) == NULL) {
        errormsg(SH_DICT, ERROR_exit(3),
                "%s: variable is NULL", *argv);
        return(1);
    }

    sfprintf(sfstdout,
             "Value: %s, Flags: %d", ptr, (int)nvp->nvflag);
    if ((int)nvp->nvflag & NV_INTEGER) {
       for (i=0; IntFlags[i].name != NULL; i++) {
          if ((int)nvp->nvflag & IntFlags[i].flag) {
              sfprintf(sfstdout, " %s", IntFlags[i].name);
              break;
          }
       }
    }
    for (i=0; Flags[i].name != NULL; i++) {
        if ((int)nvp->nvflag & Flags[i].flag)
            sfprintf(sfstdout, " %s", Flags[i].name);
    }

    sfprintf(sfstdout,"\n");
    nv_close(nvp);

    return(0);
}
```

## In Conclusion

Custom builtins can be used to extend ***ksh93*** in many useful ways just as Perl modules are used to extend Perl and Python modules are used to extend Python. To date this has not happened with ***ksh93.*** I believe that this is due to the complete lack of good documentation on how to write custom builtins.

This article is but a brief introduction on the subject. If you want to really learn how to write custom builtins, I strongly recommend that you download the ***ast-ksh*** source code from ***http://www.research.att.com/sw/tools/uwin/*** and study it. You should also read "***Guidelines for writing ksh-93 built-in commands***" by David Korn (filename: ***builtins.mm***) which is located in the top-level directory of the ***ksh93*** source tree and examine the source code for the ***ksh93*** builtins provided in the ***…/src/cmd/ksh93/bltins*** directory and the source code for the ***libcmd*** builtins at ***…/src/lib/libcmd.*** An additional resource is the OpenSolaris ***ksh93*** integration project which includes a number of extra custom builtins i.e. ***poll, open, close, dup, tmpfile, stat*** and ***rewind.***