The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1996
Number 36

In this issue …

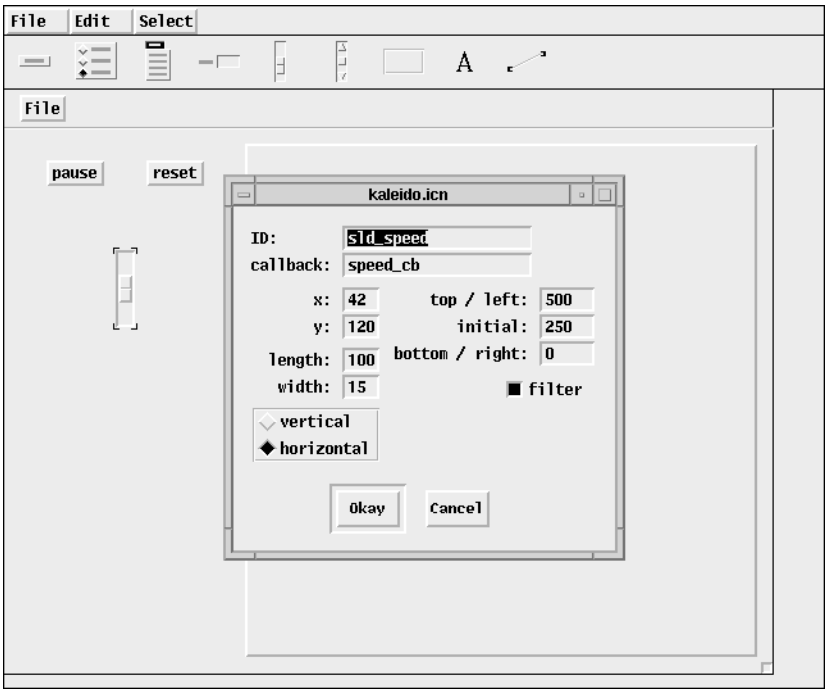
Building a Visual Interface

This is the fourth article in the series describing how visual interfaces are built using VIB. We hope you are still with us.

In the last article on this subject, we added a menu and two buttons to the interface for the kaleidoscope application. The four sliders are next.

The image above shows a newly created slider for controlling the speed of this display and its dialog box after editing. Since the dialog has not yet been dismissed, the newly created slider is shown in its original size and orientation.

We've changed the default vertical orientation to horizontal and set the range from 500 to 0, anticipating that the left end of the slider will



The Edited Slider Dialog

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

correspond to “slow” and the right end to “fast”.

We set the filter toggle so that intermediate events are filtered out. Thus, the program gets a callback only when the mouse button is released to “let go” of the slider. If the intermediate events are not filtered out, there are callbacks for every movement of the slider.

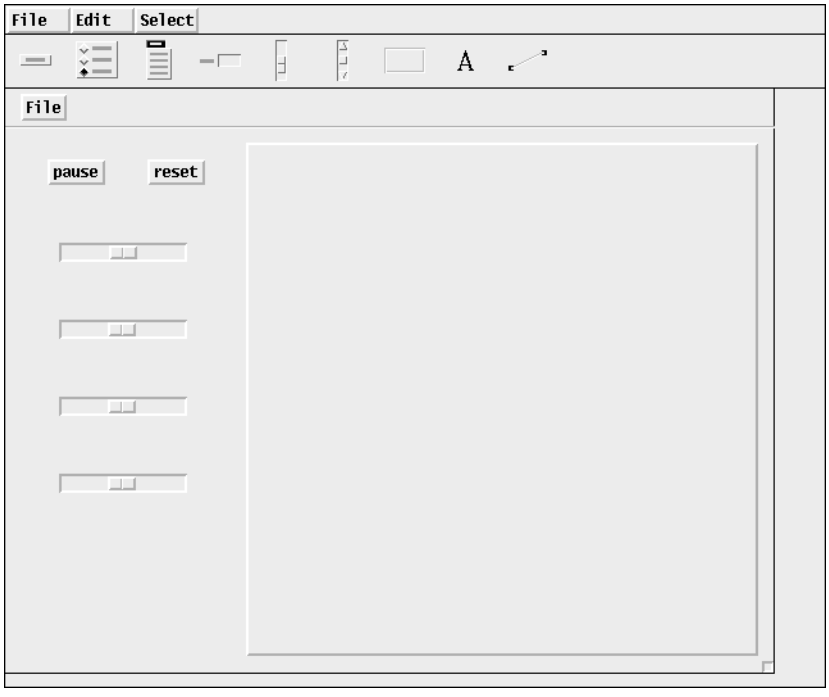
This deserves some discussion. Since most of the values associated with the kaleidoscope sliders require the display to be restarted, callbacks for intermediate slider positions are not useful and would provide no visual feedback for the user. In fact, if events are not filtered out, there is no way for the application to know which event is the last one when the user is manipulating the slider.

One Slider in Place

On the other hand, in an application that allows viewing a portion of a large image through a small pane, intermediate slider events to position the image should not be filtered out — instead, each movement of the slider can be used to reposition the portion of the image being viewed so that the user can see what’s going on and when to release the slider.

Of course, you may not know whether filtering is appropriate or not until you get into the details of writing the application code. As with other aspects of a visual interface, it’s easy to go back and change the attributes of a slider.

The image above shows the slider after it has been positioned. Getting the size and position of the slider just right may take some experimentation.



All Sliders in Place

Three more sliders are needed. We could repeat the process we used for the first slider, but we can save some work by making copies of the first slider. Entering @C when a vidget is selected makes a copy of the selected vidget. @C stands for entering c with the meta key held down. This is a common convention for interfaces built with VIB.

The new vidget won’t be where we want it, and we’ll have to change some of its attributes, but it will be the same size as the vidget from which we made the copy, which is what we want in our layout.

The image at the left shows the four sliders in place. The interface is taking shape; at this point the results should be satisfying.

The radio buttons are next. As is the case for menus, three radio buttons are provided by default. Adding and deleting radio buttons and changing their names is similar to the process for menu items.

The image at the right shows a newly created set of radio buttons and the dialog after it has been edited. The results are shown in the next image.

We've saved the labels until last for a good reason: We couldn't be sure the sliders were where we wanted them until the radio buttons were in place. Twelve labels are needed and moving labels around after creating them is a lot of work — we want to be sure that all the buttons and sliders to be labeled are just where we want them. This is one of several things you're likely to "learn the hard way", as we did.

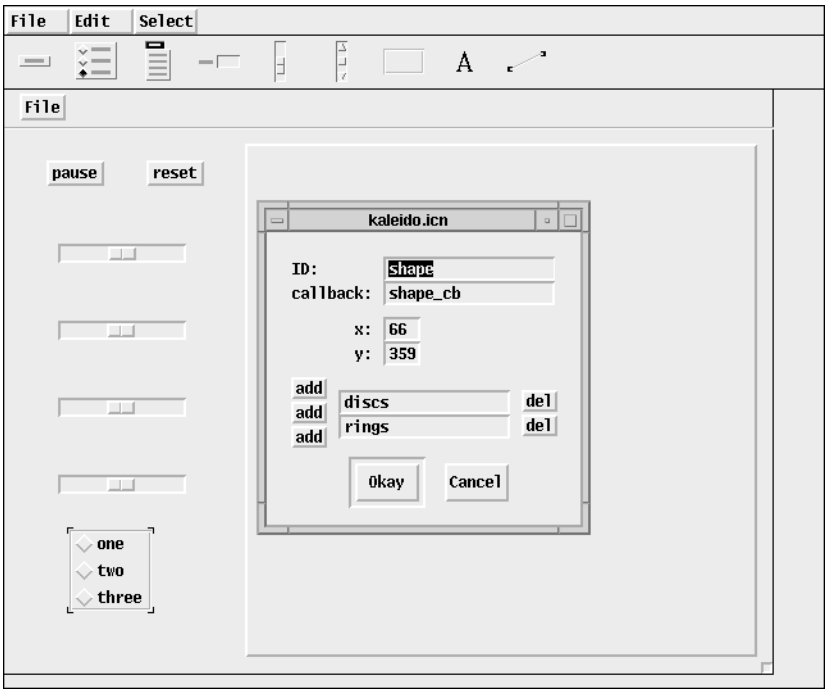
As we mentioned in an earlier article, labels are vidgets that don't accept events and produce no callbacks. Other than that, they are created and manipulated like any other vidgets.

The image at the top of the next page shows a newly created label vidget and its dialog box before editing. We'll use this label to identify the speed slider and need to change its text (label) accordingly. The result is shown on the next page.

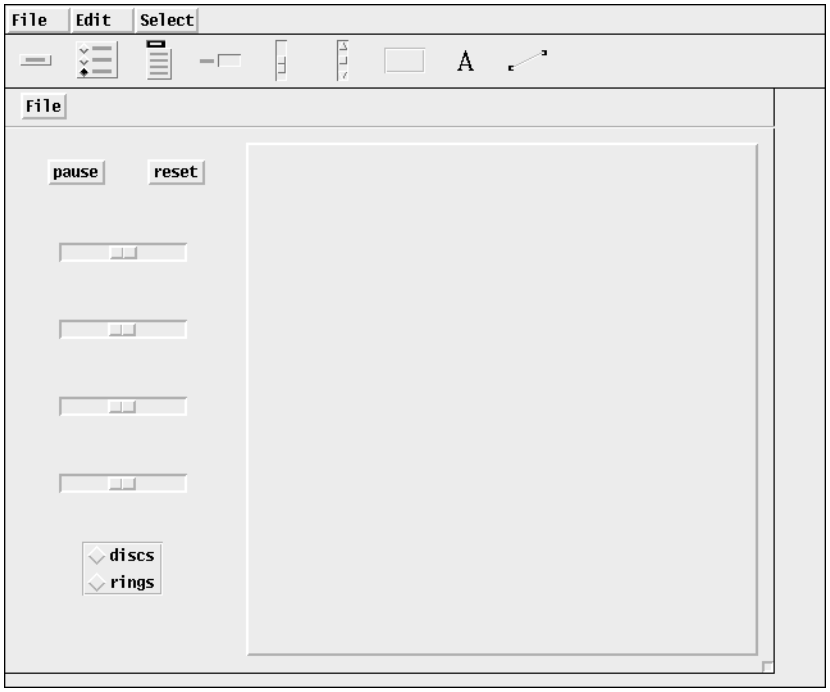
We won't bore you with all the details, but once one label is created, others can be made by copying and editing as we did for sliders.

It's a good idea to think about this before starting. Four of the labels identify sliders and might be created and positioned first.

After that's complete, the labels for the ends of the sliders can be done,



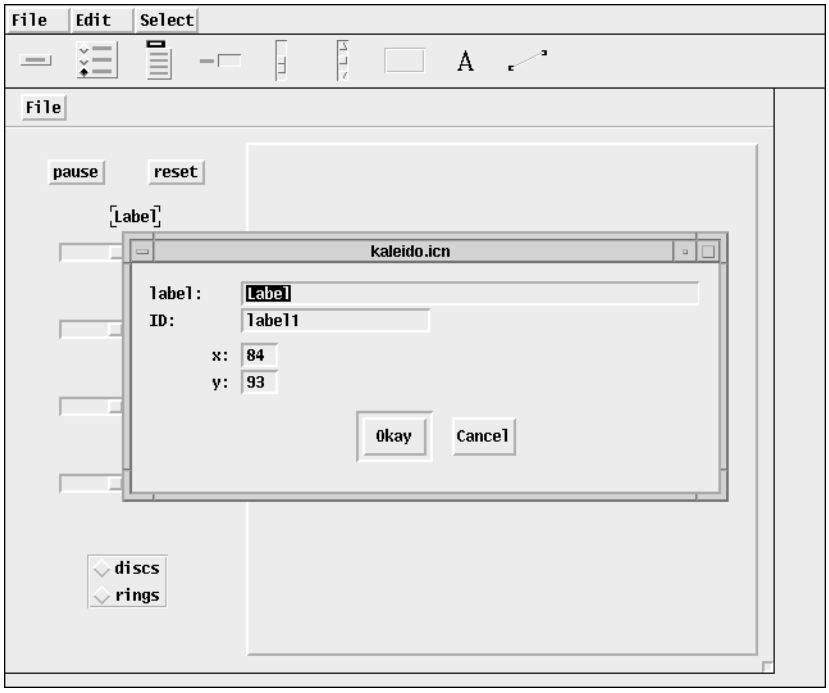
Configuring the Radio Buttons



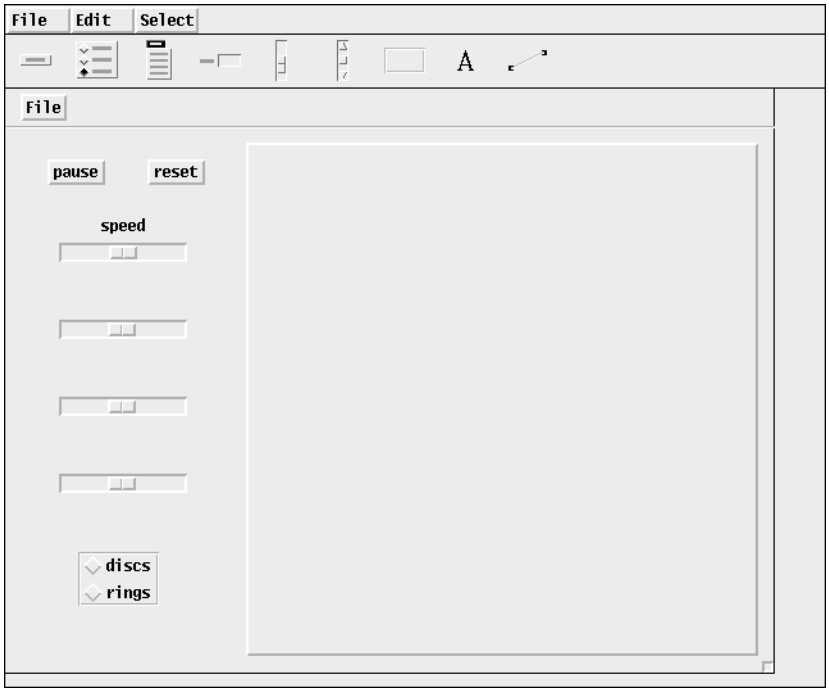
The Radio Buttons in Place

noting the fact that two sets of three have the same text.

We've created all the vidgets and they are at least approximately where we want them. That doesn't mean the interface will never change; as the application develops, new functionality may require additions or changes to the interface. With



A Label Dialog



One Label in Place

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

a good foundation, though, future changes will not be so hard.

The completed interface is shown at the top of the next page.

Next Time

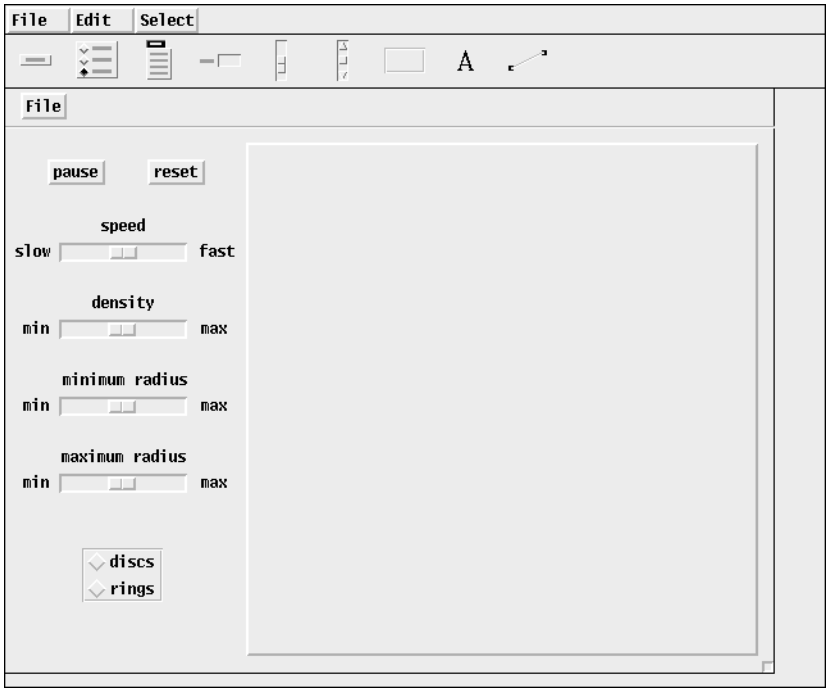
The interface in VIB looks like it will look when the application is run. It's possible, however, to see the application "in action" without leaving VIB — what user actions on different vidgets do, when callbacks occur, what they are, and so on.

We'll cover these matters in the next issue of the Analyst and then finish up with VIB by describing its menus (which we've not needed yet) and the kind of code it produces.

Subscription Renewal

For many of you, this issue is the last in your present subscription to the Analyst. If so, you'll find a renewal form in the center of this issue. Renew now so that you won't miss an issue.

Your prompt renewal also helps us by reducing the number of follow-up notices we have to send. Knowing where we stand on subscriptions also lets us plan our budget for the next fiscal year.



The Complete Kaleidoscope Interface

Quiz

Several Icon functions provide default values for omitted or null-valued arguments. This little quiz is designed to test your knowledge of these defaults. The answers are on page 12.

Suppose that

```
word := "Casablanca"
```

1. What are the results produced by the following expressions? If an expression fails or causes a run-time error, note that.

```
left(word)
right(word)
center(word)
map(word)
repl(word)
```

2. What is the default for s2 in open(s1, s2)?
3. What is the default for i in sort(X, i)?
4. What does stop() (with no arguments) write?

Loading C Functions Dynamically

Dynamic loading provides a way to use functions coded in C in an Icon program without modifying the Icon system itself. The C code is compiled and placed in a library, then loaded from the library when the Icon program runs.

Version 9.0 of Icon introduced dynamic loading. It is supported on UNIX systems that provide the System V loader interface specified by the C header file <dlsym.h>, including systems from Sun, Digital, and SGI.

We will start by showing different ways to load a C function from the Icon program library. After that, we'll discuss how to write a new function and we'll go through the steps that are needed to make use of it.

Program Library Functions

The Icon program library [1] includes an assortment of loadable UNIX interfaces and special-purpose functions. Here is a sampling:

bitcount(i)	count the bits set in an integer
chmod(s, i)	change the permissions of a file
fpoll(f, i)	poll a file for input, with timeout
getpid()	return the process identification number
kill(i1, i2)	send a signal to a process
lgconv(i)	convert a large integer to a string
tconnect(s, i)	connect a file to a TCP port
umask(i)	change the process permission mask

The full set of functions can be found in the library's cfuncs directory. Documentation and code are also available on-line on the Web [2]. We'll use the bitcount() function in our first examples.

Prebuilt C libraries are included with the binary distributions of Icon for SunOS, Solaris, Digital UNIX, and SGI Irix. We'll assume that a binary distribution has been unpacked into a directory named /icon, and so the function library file is named /icon/bin/libcfunc.so.

Loading a Function

The built-in Icon function loadfunc(libname, funcname) loads the C function funcname() from the library file libname and returns a procedure value. If the function cannot be loaded, the program is terminated.

If loadfunc(libname, "myfunc") produces p, then

p(arguments)	calls myfunc() with a list of arguments
type(p)	returns "procedure"
image(p)	returns "function myfunc"
proc("myfunc")	returns p
proc("myfunc", 0)	fails

The following program loads the function bitcount() and assigns it to a global variable of the same name. Assigning it to a global variable makes it available to other procedures, although that's not needed here. The bitcount() function returns the number of bits that are set in the binary representation of an integer.

```
$define Library "/icon/bin/libcfunc.so"

global bitcount

procedure main()
  local i

  bitcount := loadfunc(Library, "bitcount")
  every i := 250 to 260 do
    write(i, " ", bitcount(i))

end
```

When this program is run, it lists the integers from 250 to 260 along with their bit counts:

```
250  6
251  7
252  6
253  7
254  7
255  8
256  1
257  2
258  2
259  3
260  2
```

Loading from a Path

Embedding a file name such as /icon/bin/libcfunc.so in the program is undesirable. An alternative is for the program to find the library file using information from the program environment.

The Icon library procedure pathload(libname, funcname) searches the set of directories given by the FPATH environment variable to find libname and load funcname. As is usual in Icon path searching, the current directory is searched first. If the function cannot be loaded, the program is terminated.

The pathload() procedure is included by linking pathfind from the Icon program library. Using pathload(), the example program becomes:

```
$define Library "libcfunc.so"

link pathfind
global bitcount

procedure main()
  local i

  bitcount := pathload(Library, "bitcount")
  every i := 250 to 260 do
    write(i, " ", bitcount(i))

end
```

FPATH must be set before the program is run. A suggested value for FPATH is given by the Setup script that is run when installing UNIX binaries of Icon. For our hypothetical configuration, FPATH could be set by

```
setenv FPATH "/icon/bin/"
```

Implicit Function Loading

It is possible to encapsulate the loading process so that the body of an Icon program is unaware that it is calling a C function. Consider this example:

```
$define Library "libcfunc.so"

link pathfind

procedure main()
  local i

  every i := 250 to 260 do
    write(i, " ", bitcount(i))

end

procedure bitcount(n)

  bitcount := pathload(Library, "bitcount")
  return bitcount(n)

end
```

First of all, notice that there is no longer a global declaration for bitcount, and that the main procedure no longer calls pathload(). As far as the main procedure is concerned, bitcount() is just another procedure to call, with no special requirements. This is a nice simplification.

The new bitcount() procedure is a bit tricky, though. To understand it, you must know that an Icon procedure declaration creates a global variable with an initial value of that procedure. A

global variable is subject to assignment.

When main() calls bitcount() for the first time, the bitcount() procedure loads the bitcount() C function from the library. The result is assigned to the global variable bitcount, replacing the current procedure value. Consequently, all subsequent calls to bitcount() use the loaded function.

The first call to bitcount() remains incomplete after loading the function; the bits of n still must be counted. So, following loading, the procedure calls bitcount(n). Although this looks like a recursive call, it isn't — the call uses the current value of the global variable bitcount, and so it calls the loaded C function. The bits of n are counted and returned, completing the first call.

After the first time, calls to bitcount() go directly to the loaded code. The Icon procedure bitcount() is no longer accessible.

Implicit Library Loading

The Icon program library provides an implicit loading procedure for each of the C functions in the library. Small procedures like the bitcount() procedure shown above are included by linking cfunc. Using the library interface procedure, our example now can be simplified to this:

```
link cfunc

procedure main()
   local i

   every i := 250 to 260 do
      write(i, " ", bitcount(i))

end
```

The link cfunc declaration is the only hint that bitcount() is written in C. Of course, FPATH must still be set to run this program.

Making Connections

The bit counting example doesn't really illustrate the full potential of using C functions in an Icon program. Bit counting, after all, can be done in Icon. Here's something that can't.

The library function tconnect(host, port) establishes a TCP connection to a specified port number on an Internet host. TCP is a communication protocol used by telnet programs, news servers, Web servers, and many other network services.

The following program makes a connection to the Icon Web server and writes the contents of the Icon home page — in its original HTML markup language, of course.

```
link cfunc

procedure main()
   local f

   f := tconnect("www.cs.arizona.edu", 80)
   writes(f, "GET /icon/ HTTP/1.0\n\n")
   flush(f)
   seek(f, 1)
   while write(read(f))

end
```

The tconnect() call establishes the connection and returns a file that is open for both input and output. The internet host www.cs.arizona.edu is our department's Web server. Port 80 is used by most Web servers, including ours.

The program then transmits a request for the /icon/ Web page. The details of the request string are specified by the "Hypertext Transfer Protocol" [3], which we won't discuss here.

The flush() call ensures that all the data is actually sent, and then the seek() call resets the file in preparation for a switch from output to input. In this situation seek() does not actually reposition the file, but it's required when switching modes.

Finally, lines are read and echoed until an end-of-file is received.

Writing Loadable C Functions

We've seen how functions can be loaded from the library; now let's consider how to write them.

Because the Icon system expects C functions to implement a certain interface, dynamic loading usually requires specially written C functions. In general, it is not possible to use an existing C function without writing an intermediate "glue" function.

C functions must deal with the data types used by the Icon run-time system, notably the "descriptors" that represent all Icon values. While an understanding of the Icon run-time system [4, 5] is helpful, it is possible to create useful functions by modeling them after existing library functions. Integer and string values are most easily handled.

A loadable C function has the prototype

```
int funcname(int argc, descriptor *argv)
```

where argc is the number of arguments and argv is an array of argument descriptors. The first element, argv[0], is used to return an Icon value, and is initialized to a descriptor for the null value. This element is not included in the count argc. The actual arguments begin with argv[1].

If the C function returns zero, the call from Icon succeeds. A negative value indicates failure. If a positive value is returned, it is interpreted as an error number and a fatal error with that number is signalled. In this case, if argv[0] is non-null, it is reported as the "offending value". There is no way for a C function to suspend, and no way to indicate a null value as an offending value in the case of an error.

Interface Macros

The C file icall.h contains a set of macros for use in writing loadable functions. Documentation is included as comments. This file is not included in binary distributions of Icon but can be found in the cfuncs directory in the source code of the Icon program library. Alternatively, it can be loaded from the Web [2]. Macros are provided for:

- inspecting the type of an Icon value
- validating the type of an argument
- converting an Icon value into a C value
- returning a C value in Icon form
- failing or signaling an error

Most macros deal with integers or strings. Some support also is provided for handling real and file values.

Counting Bits, Again

For a concrete example of a C function, we will revisit the bitcount() function used earlier and look at its source code:

```
#include "icall.h"

int bitcount(int argc, descriptor *argv)
{
   unsigned long v;
   int n;

   ArgInteger(1);

   v = IntegerVal(argv[1]);
   n = 0;
   while (v != 0) {
      n += v & 1;
      v >>= 1;
      }
   RetInteger(n);
}
```

Like all loadable functions, bitcount() is an integer function with two parameters, argc and argv.

The ArgInteger macro call verifies that argument 1 is a simple integer. (Large integers are typically rejected by C functions because of the extra work involved.) If argument 1 is missing or has the wrong type, ArgInteger makes the function return error code 101 (integer expected or out of range).

The IntegerVal macro call extracts the value of the first argument.

In each pass through the while loop, the low-order bit of v is extracted (v & 1), added to n, and shifted off (v >>= 1). When no more nonzero bits are left, the loop exits. Note that v is declared unsigned to ensure that only zero bits are inserted by the shift operation.

The RetInteger macro call returns the value of n as an Icon integer.

Preparing a Library

To be used in an Icon program, a C function must be built and installed in a library. Compilation comes first, usually involving a command such as

```
cc –c bitcount.c
```

to produce an object file bitcount.o. We're assuming that icall.h has been copied to the current directory, although other methods are also possible. The –c option causes a relocatable object file to be produced instead of a stand-alone executable program. Other options, such as optimization options, also could be specified.

A C function can be loaded only from a "shared library". Even if there is just one function, it must be placed in a library. Library names conventionally end with a .so suffix.

It seems that every system has a different way to create libraries, usually involving special flags to cc or ld. The program library file mklib.sh is a shell script that embodies our understanding of shared library creation. It takes one argument naming the library to be created and one or more additional arguments listing object file names. For example, the command

```
mklib.sh mylib.so bitcount.o
```

creates a file mylib.so containing the functions read from bitcount.o. Like icall.h, mklib.sh is available in the program library source code or from the Web.

Conclusion

We will close with a review of the key points:

- Icon can load C functions on many UNIX systems.
- The C functions must be tailored to Icon's requirements.
- Each function must be loaded before it can be called.
- A simple Icon procedure can be used to hide the loading details.
- pathload() searches FPATH to find a function library.
- Some useful functions are provided in the Icon program library.

We'd be pleased to hear of any interesting applications you find for dynamic function loading, and additional contributions to the Icon program library always are welcome.

References

1. R. E. Griswold and G. M. Townsend, *The Icon Program Library; Version 9.2* , Department of Computer Science, The University of Arizona, Icon Project Document IPD272, 1996.

2. The Icon Project, "Program Library Index: Loadable C Functions", Department of Computer Science, The University of Arizona, WWW http://www.cs.arizona.edu/icon/library/ccfuncs.html.

3. T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol — HTTP / 1.0", work in progress, Internet Engineering Task Force, February 19, 1996, WWW http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-v10-spec-05.html.

4. R. E. Griswold and M. T. Griswold,*The Implementation of the Icon Programming Language,* Princeton University Press, Princeton, New Jersey, 1986.

5. R. E. Griswold, *Supplementary Information for the Implementation of Version 9 of Icon,* Department of Computer Science, The University of Arizona, Icon Project Document IPD239, 1995.

Icon Glossary

This is second part of a glossary of Icon terms. There will be one or two more parts, depending on how much space we have in upcoming Analysts. When the glossary is complete, we'll provide a copy as a supplement to the Analyst.

If you have questions or suggestions about material in the glossary, please let us know.

activation: evaluation of a **co-expression**.

allocation: the process of providing space in **memory** for values created during **program execution**. See also: **garbage collection**.

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–project@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

and

Bright Forest Publishers

Tucson Arizona

alternation: a **control structure** that **generates** the results of its first **operand** followed by the results of its second **operand**. See also **disjunction**.

argument: an expression that provides a value for a **function** or **procedure call**; sometimes used to mean **operand**.

associativity: the order in which like **operators** are evaluated in the absence of parentheses. Associativity can be left-to-right, in which case the first (left-most) **operator** is evaluated first or right-to-left, in which case the last (right-most) **operator** is evaluated first.

augmented assignment: assignment combined with a **binary operation**. The **binary operation** is performed on the value of the **left-operand variable** and the value of the **right operand**, and then the result is assigned to the **left-operand variable**.

built-in: a feature that is part of the Icon programming language, as opposed to a feature written in Icon.

case expression: a **control structure** in which an **expression** to evaluate is selected depending on a value.

co-expression: an **expression** coupled with an environment for its execution. If the **expression** is a **generator**, its **results** can be obtained one at a time by **activation**.

character: the elementary unit from which **strings** and **csets** are composed. Characters are used to represent letters, digits, punctuation marks, and so forth. Characters are represented internally by small nonnegative integers (typically 8 bits). Some characters have associated **glyphs**. Icon has no character **data type**.

collating sequence: the sorting order for **strings** imposed by the internal representation of **characters**.

command-line argument: a string given after the program name when Icon is invoked from a command line. Command-line arguments are passed to the main **procedure** as a list of **strings** in its first **argument**.

comparison operation: a **binary operation** that compares two values according to a specified criterion. A comparison operation succeeds and returns the value of its right **operand** if the criterion is satisfied. Otherwise it **fails**. See also **numerical comparison**, **lexical comparison**, and **value comparison**.

conjunction: a **binary operation** that evaluates its **operands** but performs no computation on them; used to test if two **expressions** both **succeed**. Conjunction has the effect of logical *and*. See also: **mutual evaluation** and **disjunction**.

control character: a **character** that has special interpretation in an input/output context. For example, **linefeed** or **newline**.

control structure: an expression whose evaluation may alter the otherwise sequential order of evaluation of **expressions**.

data type: a designation that identifies values that share common properties and **operations**. Icon has 12 data types: **co-expression**, **cset**, **file**, **integer**, **list**, **null**, **procedure**, **real**, **set**, **string**, **table**, and **window**. In addition, each **record declaration** defines a data type. The term data type often is shortened to **type**.

declaration: a component of a program that specifies its properties and structure. There are seven kinds of declarations: **global**, **local**, **static**, **procedure**, **record**, **link**, and **invocable**.

default case clause: a component of a case expression that contains an **expression** that is evaluated if no other **expression** is selected in a **case expression**. A default case clause is indicated by the reserved word default.

default value: a value that is provided in place of an **omitted** or **null-valued argument** of a **function**.

default table value: a value specified when a **table** is created that serves as the value corresponding to **keys** that are not in the **table**.

define directive: a **preprocessor directive** that associates a symbol (name) with a string so that the string is substituted for subsequent uses of the symbol in the program.

disjunction: logical *or*; used to describe the effect of **alternation**. See also **conjunction**.

environment variable: a named attribute of the system environment under which a program runs. Environment variables can be used to specify the size of Icon's **memory regions**, the locations of **libraries**, and so forth.

error: a condition or situation that is invalid. Errors may occur during **translation**, **linking**, **compiling**, or **execution**. An error in **translation** prevents **linking**. An error in **linking** prevents the production of an **icode** file. An error that

occurs during **execution** is called a **run-time error**. See also **error conversion**.

error conversion: changing **run-time errors** to expression **failure** rather than program **termination**. This is accomplished by setting a **program state** using a **keyword**.

escape sequence: a sequence of **characters** in a **string** or **cset literal** that encodes a single **character**. Escape sequences usually are used for **characters** that cannot be given **literally**.

function: a **built-in procedure.**

garbage collection: the process of reclaiming space in **memory** that has been **allocated** but no longer needed**.** Garbage collection occurs automatically when insufficient space remains for **allocation.** Garbage collection can be forced by collect()**.**

global variable: a **variable** whose value is accessible throughout the entire program and from the beginning of **execution** to the end.

glyph: a symbol such as a letter, digit, or punctuation mark.

heterogeneous structure: a **structure** whose **elements** have different **types**.

homogeneous structure: a **structure** all of whose **elements** have the same **type**.

initial clause: an optional component of a **procedure** that contains **expressions** that are evaluated only on the first **invocation** of the **procedure**.

invocable declaration: a **declaration** that specifies that **procedures** are to be included when a program is **linked**, even if there is no explicit reference to them in the program. Such procedures may be called using **string invocation**.

invocation: the evaluation of a **procedure** or **function**. Invocation and **call** are sometimes used synonymously.

keyword: An ampersand (&) followed by a string of letters that has a special meaning. Some keywords are **variables**.

lexical comparison: comparison of **strings** "alphabetically" according to the numerical values used to represent **characters.** Also called **string comparison**. See also **collating sequence**.

library module: a file consisting of one or more **procedures** or other declarations that have been **translated** into **ucode** so that they may be incorporated in a program by **linking**.

limitation: restricting the number of times a **generator** is **resumed**. Limitation can be specified by a **control structure** or because of the syntactic context in which the **generator** appears. See also **bounded expression**.

line terminator: a **character** or pair of **characters** that is used by convention to mark the end a line of text in a file. In UNIX, the line terminator is a **linefeed character**; on the Macintosh, it is the **return character**; in DOS, it is a **linefeed character** followed by a **return character**. Other platforms generally use one of these conventions. See also: **newline character**.

link declaration: a **declaration** that causes a **library module** to be included in a program during **linking**.

literal: a sequence of **characters** in a source program that directly represents a value, as the **integer** 1 and the **string** "hello".

local variable: a **variable** that is accessible only to the **procedure** in which it is **declared** and during a single **invocation** of the **procedure**. Local variables are created when a **procedure** is **invoked** and are destroyed when the **procedure returns** or **fails**, but not when the **procedure suspends**. See also: **global variable** and **static variable**.

matching function: a **function** that returns a portion of the **subject** in **string scanning**. The term can be extended to include matching **procedures**.

memory: the space in which a program and the **objects** it creates are stored. Memory is implemented in RAM. Also called **storage**.

memory region: a portion of **memory** used for storing Icon values. There are separate memory regions for **strings** and for other **objects**. Also called **storage region**.

mixed-mode arithmetic: arithmetic on a combination of **integers** and **real numbers** to produce a **real number**. Any arithmetic operation that has a **real operand** produces a **real** value.

mutual evaluation: an **expression** consisting of an **argument list**, but with no **function** or **procedure**. A mutual evaluation **expression succeeds** only if all the **expressions** in the **argument list succeed**. The **result** of a specific **argument** can be selected by an **integer** preceding the **argument list**.

newline character: the single **character** used to

represent a **line terminator** in **Icon** regardless of the actual representation used in the underlying system.

object: in the most general sense, any value. More specifically, a value that is represented by a **pointer** to **memory**. These are **strings**, **csets**, **files**, **real numbers**, **large integers**, **co-expressions**, **procedures**, **windows**, and **data structures**. Sometimes the term object is used for just **data structures**.

operand: an expression that provides a value for an **operation**. See also **argument**.

operation: an expression that is part of the **built-in** computational repertoire of Icon and cast in the form of an **operator** and **operands**. Sometimes used in a broader sense to include **function** and **procedure calls** to characterize **expressions** that are not **control structures**.

operator: a symbol consisting of one or more characters that designates an **operation**.

parameter: an **identifier** in a **procedure declaration** that provides a **variable** to which a value is passed when the **procedure** is called. **Parameters** are **local variables**.

passing arguments: the **assignment** of **argument** values in a **procedure call** to the **parameters** of the **procedure**.

Answers to the Quiz

1. The second argument of left(), right(), and center(), which determines the length of the result, defaults to 1, so the answers for these are:

```
left(word)      "C"
right(word)     "a"
center(word)    "l"
```

In the case of center(word), if word has an even number of characters, as it does here, the result is the character to the right of center.

While 1 may seem like a useless default for field length, it actually can be useful. For example, left(s) can be thought of as producing the left-most character of s. Programs written using these functions in this way may be difficult for others to understand, however.

The second and third arguments of map() default to upper- and lowercase letters, respectively, so the answer is

```
map(word)     "casablanca"
```

The next one may surprise you; there is no default for the number of times to replicate a string, so repl(word) causes a run-time error. This seems to us like an inconsistency in language design; a default of 1 would have been better.

2. The default for s2 in open(s1, s2) is "rt". The "r" should be familiar. The "t" enables the translated mode for input, so that line terminators automatically are converted to newline characters. This is only an issue for platforms such as MS-DOS and the Macintosh, where line terminators are not newline characters. The converse operation is performed on output. The result is to make line terminators transparent.

3. The default for i in sort(X, i) is 1. For tables, the result is a list of two-element key / value lists. For other types, the second argument is not used.

4. stop() is tricky. Except for the fact that stop() terminates program execution and writes to standard error output instead of standard output by default, stop() treats its arguments in the same way write() does — it writes them in succession and adds a line terminator. With no arguments, write() writes a a blank (empty) line, and stop() does too. Such a blank line usually causes no harm, but it may be disconcerting when output is to the screen.

What's Coming Up

We have lots of things on deck. There's another article in the series on building visual interfaces, another article on versum numbers, an article on dynamic analysis that takes a different approach from past ones, and more of the glossary.