

30 Von Icon auf C-Bibliotheken zugreifen

30.1 Motivation

30.1.1 Geschichte

Es gibt ein paar Programmiersprachen, die mehr als 40 Jahre hinter sich haben – und immer noch Verwendung finden, wenn auch einige davon nur noch in Nischen zu finden sind. C allerdings hat es z.B. geschafft, selbst für die Entwicklung von Betriebssystemen (UNIX-Kernel für PDP-11 und einige andere) eingesetzt zu werden.

- Assembler [ab 1948] (spezifisch für jeden Prozessor)
- BASIC [1964] - Einsteigersprache
- C [ab 1969](in Form von C, C++, C#, JAVA)
- Cobol [1960] – Branchensoftware für Banken
- Fortran [ab 1954] – Naturwissenschaften
- Pascal [1971] (in Form von Pascal, Turbo-Pascal, Modula, Oberon, Delphi, Lazarus)
- Prolog [1972]
- ähem, Icon [1977] (in Form von Icon, Idol, JCon, ObjectIcon, Unicon, Goaldi [2015])

Übrigens: Selbst die auf dem Raspberry Pi so beliebte Programmiersprache Python basiert auf Konzepten der Programmiersprache Icon...

Es ist keine Überraschung, dass mehr Leute mehr Programme und Bibliotheken in C und Pascal erstellt haben, als dies von den wenigen Programmierern in Icon hätte erreicht werden können.

So ist es keine Überraschung, dass sich Icon-Programmierer Gedanken gemacht haben, wie man sich die Arbeit der Programmierer von Bibliotheken in anderen Programmiersprachen zunutze machen könnte. Ergebnis dieser Arbeit sind Konzepte, Schnittstellen, Tools sowie Funktionen & Programme, die auf externe Bibliotheken anderer Programmiersprachen zugreifen können.

In diesem Tutorial geht es um die Einbindung von Bibliotheken, die in der Programmiersprache C geschrieben wurden sowie um die Vermittlung tiefgehender Grundlagen, die mit denen zum Aufrufen von Bibliotheken in C++ (Teil 31) und anderen Programmiersprachen (Teil 32) identisch sind.

30.1.2 Was ist eigentlich eine Bibliothek

Unter einer Bibliothek (Library) versteht man in der Software-Entwicklung eine Sammlung von Routinen, Funktionen, Prozeduren, Strukturen, Klassen, Objekten, Methoden, …, die man einsetzen kann,

- ohne den Code dahinter zu kennen
- ohne den Code anpassen zu müssen – ich sage sogar: ohne den Code anpassen zu können, weil die Bibliotheken oft nicht quelloffen sind
- ohne wissen zu müssen, in welcher Programmiersprache die Bibliothek einst erstellt wurde

Das Einzige, was der Entwickler, der in der Programmiersprache A entwickelt, wissen muss, um eine Bibliothek zu nutzen, die irgendwann jemand in der Programmiersprache B geschrieben hat, ist:

- welche Datentypen, Strukturen, Klassen, Objekte zur Verfügung gestellt werden,
- welche Funktionen, Prozeduren, Methoden inkl. Argumente und Datentypen zur Verfügung gestellt werden,
- wie die Aufrufkonventionen definiert sind,
- wie die Datentypen der Programmiersprache A in die in der Programmiersprache B geschriebenen Bibliothek einfließen,
- wie die Ergebnisse aus der in der Programmiersprache B geschriebenen Bibliothek in die Datentypen der Programmiersprache A rückgewandelt werden.

Wir erinnern uns, das wir in Icon zwar Datentypen kennen, aber Variablen nicht deklariert werden.

So ist in Icon so etwas möglich:

```
x := "Hallo"     # Zeichenkette
x := sizeof(x)   # Ganzzahl
x *:= &pi        # irrationale Dezimalzahl
```

Bei Einsatz fast aller Programmiersprachen, die ich kennen gelernt habe, erfährt man nach dem Compilieren solcher Zeilen so einiges über das Potential an Fehlermeldungen, zu der der jeweilige Compiler in der Lage ist.

30.2 Von Icon-Programmen heraus auf C-Funktionen zugreifen

30.2.1 Beispiele innerhalb der IPL

Die Icon-Program-Library (IPL) beinhaltet eine Ansammlung ladbarer Unix-Schnittstellen und Funktionen von besonderem Nutzen. Einige gibt es zur allg. Verwendung, einige zur Veranschaulichung und einige erfüllen spezielle Zwecke. Es handelt sind um Folgende:

- `bitcount(i)` zählt die gesetzten Bits in einer Ganzzahl
- `chmod(s,i)` ändert die Zugriffsrechte einer Datei
- `fpoll(f, i)` pollt eine Datei für Eingabe inkl. Auszeit, d.h. prüft auf Inhalt und wartet ggf. darauf
- `getpid()` gibt die Prozess-ID zurück
- `kill(u1, i2)` sendet ein Signal zu einem Prozess
- `lgconv(i)` konvertiert eine große Ganzzahl (im Sinne von Large Integer Arithmetik – LIA) in eine Zeichenkette
- `tconnnect(s, i)` verbindet eine Datei mit einem TCP-Port
- `umask(i)` ändert die Prozess-Zugriffsmaske

Der vollständige Funktionscode kann im Verzeichnis `/home/pi/icon9_51/ipl/cfuns` gefunden werden. Dokumentation und Code gibt es unter www.cs.arizona.edu/icon/library/cfuncs.htm.

30.2.2 Dynamisches Laden

In der Programmiersprache Icon gibt es verschiedene Techniken, wie man auf Funktionen aus Bibliotheken, die in anderen Programmiersprachen geschrieben wurden, zugreifen kann.

30.2.2.1 Dynamisches Laden ganzer Bibliotheken

Die Technik *Dynamisches Laden* erlaubt es Icon-Programmen, in C geschriebene Funktionen ohne Änderung des Icon-Systems zu laden. Dabei wurde der C-Code compiliert und in einer Library abgelegt und wird aus der Library geladen, sobald das Icon-Programm läuft. Hier geht es um die Anwendung und den Nutzen sowie die Konstruktion solcher Funktionen.

Wie wir als angehende Icon-Jünger schließlich Bibliotheken in der Programmiersprache C erstellen können, zeige ich in Abschnitt 30.3.

Die Funktion `bitcount()` ist für eine detaillierte Behandlung ein gutes Beispiel.

Listing home/pi/icon9_51/bin/test_bitcount0.icn

```
1  link cfunc
2
3  procedure main()
4     local i
5     every i := 250 to 260 do
6        write(i, "  ", bitcount(i))
7  end
```

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

Abbildung 30-1: Programmausgabe

30.2.2.2 Einzelne Funktionen gezielt laden

Die eingebaute Funktion `loadfunc(libname, funcname)` lädt die C-Funktion `funcname()` aus der Library `libname` und gibt einen Wert des Datentyps `procedure` zurück. Falls die Funktion nicht geladen werden konnte, wird das Programm beendet.

Falls `loadfunc(libname, "myfunc")` die Variable `p` vom Datentyp `procedure` erzeugt, dann gilt:

- `p(argumente)` ruft `myfunc()` mit einer Argumentenliste auf
- `type(p)` gibt `"procedure"` zurück
- `image(p)` gibt `"function myfunc"` zurück
- `proc("myfunc")` gibt `p` zurück
- `proc("myfunc", 0)` scheitert

Das folgende Programm lädt die Funktion `bitcount()` und weist es einer globalen Variablen des gleichen Namens zu. Durch Zuweisung zu einer globalen Variable wird sie anderen Prozeduren zur Verfügung gestellt – auch wenn dies hier nicht benötigt wird, denn unser Beispielprogramm hat nur eine einzige Prozedur.

Listing home/pi/icon9_51/bin/test_bitcount1.icn

```
$define Library "/home/pi/icon9_51/bin/libcfunc.so"

global bitcount

procedure main()
   local i
   bitcount := loadfunc(Library, "bitcount")
   every i := 250 to 260 do write(i, " ", bitcount(i))
end
```

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

Abbildung 30-2: Screenshot

OK. Das Programm `test_bitcount1.icn` funktioniert. Aber, ehrlich gesagt, das Wahre ist eine solche Lösung nicht, denn sie ist ohne Änderung des sog. Magic-String zur Deklaration der Library nicht so ohne Weiteres von einem Computer auf einen anderen übertragbar. Also suchen wir nach einer anderen Lösung.

Eine Programmiersprache, die unter verschiedenen Betriebssystemen eingesetzt werden kann, entfaltet ihre wahre Stärke dann, wenn der Quellcode möglichst ohne betriebssystembedingte Anpassungen übernommen werden kann. Dies könnte man zwar durch Code wie diesen erreichen:

```
$ifdef _LINUX
   $define Library
"/home/pi/icon9_51/bin/libcfunc.so"
$endif
$ifdef _MS_DOS
   $define Library "c:/WINICON/bin/libcfunc.so"
$endif
```

Aber auch diese Lösung ist nicht allgemeingültig innerhalb der Linux- und der Windows-Welt übertragbar, denn es hängt von der jeweiligen Installation des Betriebssystems ab. Bei dem einem heißt das Home-Verzeichnis unter Linux /home/pi bei dem anderen /home/meins oder sonstwas.

30.2.2.3 Aus einem Pfad laden

Eine Bibliothek mit einem Dateinamen wie `"/home/pi/icon9_51/bin/libfunc.so"` einzubetten, ist also unerwünscht. Eine Alternative für das Programm besteht darin, die Bibliothek unter Nutzung der Umgebungsvariablen zu finden.

Die Icon-Library-Prozedur `pathload(libname, funcname)` durchsucht eine Menge an Verzeichnissen, die von der Umgebungsvariablen `FPATH` angegeben wird, um `libname` zu finden und `funcname` zu laden. Wie beim Durchsuchen von Pfaden in Icon allgemein üblich wird das aktuelle Verzeichnis zuerst durchsucht. Falls die Funktion nicht geladen werden kann, wird das Programm beendet. Die Prozedur `pathload()` wird durch Einbinden von `io` verfügbar. So erhalten wir das Programm `test_bitcount2.icn`.

Anmerkung zu `link io`: Auf der Icon-Homepage www.cs.arizona.edu/icon gibt es mehrere Dokumente, die sich mit dem Laden von Libraries aus einem Pfad heraus beschäftigen. Diese Beispiele verwenden `link pathfind` anstelle des von mir gewählten `link io`. Als ich mich vor Jahren schon einmal mit dieser für mich vollkommen abgefahrenen Technik beschäftigt habe, kam ich da über frustrierend gescheiterten Erstversuchen nicht hinaus. Mit dem deutlich besseren Wissen und Fähigkeiten von heute, habe ich mich da noch einmal herangewagt. Bei Verwendung der Zeile `link pathfind` kam eine Fehlermeldung, dass `pathfind.u1` nicht gefunden werden konnte. Daraufhin habe ich das Verzeichnis `ipl` nach einer Datei pathfind.icn durchsucht. Diese Datei gab es nicht – was die Fehlermeldung erklärt! Diese Datei gab es auch unter der gesamten Seite www.cs.arizona.edu/icon nicht. Aber der Befehl `pathload` ist auf dem Icon-Material enthalten – als Funktion innerhalb `io.icn`. Deswegen habe ich es mit dieser Datei versucht, indem ich die vorcompilierte Dateien `io.u1` bzw. `io.u2` mittels `link io` eingebunden habe. Das Ergebnis war erfolgreich!

Listing home/pi/icon9_51/bin/test_bitcount2.icn

```
1   $define Library "libcfunc.so"
2
3   link io
4
5   global bitcount
6
7   procedure main()
8      local i
9      bitcount := pathload(Library, "bitcount")
10     every i := 250 to 260 do write(i, " ", bitcount(i))
11  end
```

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

Abbildung 30-3: Screenshot

Voreinstellungsgemäß enthält die Umgebungsvariable `FPATH` das aktuelle Verzeichnis und das installierte Verzeichnis `/home/pi/icon9_51/ipl`. Um eine Library zu finden, die sich woanders befindet, muss `FPATH` ausdrücklich vor dem Programmstart gesetzt werden.

Wer weiß noch, wie das ging? Ja, ich weiß, lang, lang ist's her... Im Teil 1 des Icon-Tutorials war's gewesen:

```
sudo nano /home/pi/.bashrc
```

und dort folgende Zeilen anhängen (falls noch nicht geschehen), ggf. speichern und Reboot:

```
echo "set FPATH"
FPATH="/home/pi/icon9_51/ipl/packs/loadfunc /home/pi/icon9_51/ipl/packs/loadfuncpp
/home/pi/icon9_51/ipl/cfuncs"
echo $FPATH
export FPATH
```

30.2.2.4 Funktion implizit laden

Es ist möglich, den Ladevorgang zu verschachteln, so dass der Rumpf eines Icon-Programmes in Unkenntnis verbleibt, dass es überhaupt eine C-Funktion aufruft. Betrachte dieses Beispiel:

Listing home/pi/icon9_51/bin/test_bitcount3.icn

```
1   $define Library "libcfunc.so"
2
3   link io
4
5   procedure main()
6      local i
7
8      every i := 250 to 260 do write(i, " ", bitcount(i))
9   end
10
11  procedure bitcount(n)
12     bitcount := pathload(Library, "bitcount")
13     return bitcount(n)
14  end
```

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

Abbildung 30-4: Screenshot

Beachte als Allererstes, dass es keine globale Deklaration von `bitcount` mehr gibt, und dass die Prozedur `main()` nicht mehr `pathload()` aufruft. Was die Prozedur `main()` betrifft, ist `bitcount` nur eine andere aufzurufende Prozedur, ohne spezielle Forderungen. Dies ist eine nette Vereinfachung!

Dagegen ist die neue Prozedur `bitcount()` ein wenig trickreich. Um sie zu verstehen, musst Du wissen, dass eine Icon-Prozedur-Deklaration eine globale Variable mit einem anfänglichen Wert dieser Prozedur erzeugt. Eine globale Variable ist Gegenstand einer Zuweisung. Sobald `main()` die Prozedur `bitcount()` zum allerersten Mal aufruft, lädt die Prozedur `bitcount()` die C-Funktion `bitcount` aus der Library. Das Ergebnis wird der globalen Prozedur-Variablen `bitcount` zugewiesen, wodurch der aktuelle Prozedurwert ersetzt wird.

Demzufolge verwenden alle nachfolgenden Aufrufe von `bitcount()` die geladene Funktion.

Der erste Aufruf von `bitcount()` verbleibt nach dem Laden der Funktion jedoch unvollständig; die

Bits von `n` müssen noch gezählt werden. Daher ruft die Icon-Prozedur `bitcount()` nach dem Laden der C-Funktion `bitcount(n)` aus der Library eben diese auf. Obwohl dies wie ein rekursiver Aufruf aussieht, ist er keiner – der Aufruf verwendet den aktuellen Wert der globalen Variablen `bitcount` und deswegen ruft es die geladene Funktion auf. Die Bits von `n` werden gezählt und zurückgegeben und vervollständigen auf diese Weise den ersten Aufruf.

Nach dem ersten Mal gehen Aufrufe von `bitcount()` direkt zum geladenen Code. Die Icon-Prozedur `bitcount()` ist nicht mehr zugänglich.

Listing home/pi/icon9_51/bin/test_bitcount4.icn

```
1  link cfunc
2
3  procedure main()
4     local i
5     every i := 250 to 260 do write(i, " ", bitcount(i))
6  end
```

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

Abbildung 30-5: Screenshot

30.2.2.5 Library implizit laden

Die Icon-Programm-Library (IPL) stellt für jede C-Funktion eine implizite Lade-Prozedur zur Verfügung. Kleine Prozeduren wie die oben gezeigte Prozedur `bitcount()` werden durch Einbinden von `cfunc` eingeschlossen. Unter Verwendung der Library-Schnittstellen-Prozeduren kann unser Beispiel vereinfacht werden. Die Anweisung `link cfunc` ist der einzige Hinweis, dass `bitcount` in C geschrieben wurde.

30.2.2.6 Verbindungen aufbauen

Die bisher vorgestellten Beispiele, Bits zu zählen, veranschaulichen das vollständige Potential von C-Funktionen in einem Icon-Programm nicht annähernd. Denn Bitzählen kann man schließlich auch in Icon programmieren.

Deswegen kommt jetzt ein kleines Beisiel, das so in dieser Form in (reinem) Icon nicht möglich ist.

Die Library-Funktion `tconnect(host, port)` stellt eine TCP-Verbindung zu einem angegebenen Port auf einem Internet-Host her. TCP ist ein Kommunikationsprotokoll, das von Telnet-Programmen, Newsservern, Webservern und vielen anderen Netzwerkdiensten verwendet wird.

Das folgende Programm stellt eine Verbindung zum Icon-Web-Server her und schreibt die Inhalte der Icon-Homepage – natürlich im HTML-Code (s. Abb. 30-6, Abb. 30-7).

Der Aufruf von `tconnect()` stellt die Verbindung her und gibt ein Datei-Handle zu einer Datei zurück, das sowohl für die Eingabe als auch für die Ausgabe geöffnet ist. Der Internet-Host www.cs.arizona.edu ist der Webserver der Abteilung Computer-Wissenschaften der Universität Arizona. (Port 80 ist die Standard-Portnummer für Webserver.) Das Programm überträgt dann eine Anfrage für die Icon-Webseite. Die Details für eine Request-Zeichenkette sind im Hypertext-Transfer-Protokoll festgelegt, und werden hier nicht weiter besprochen (nähere Details siehe auch www.w3org/Protocols/rfc2616/rfc2616.html).

Listing home/pi/icon9_51/bin/test_tconnect.icn

```
1  link cfunc
2
3  procedure main()
4     local f
5
6     f := tconnect("www.cs.arizona.edu", 80)
7     writes(f, "GET /icon/ HTTP/1.0\n\n")
8     flush(f)
9     seek(f, 1)
10    while write(read(f))
11 end
```

Der Aufruf von `flush()` stellt sicher, dass alle Daten tatsächlich gesendet werden, danach setzt der Aufruf von `seek()` die Datei in Vorbereitung für das Umschalten von Ausgabe zu Eingabe zurück. In dieser Situation führt `seek()` tatsächlich keine Repositionierung der Datei durch, aber es wird verlangt, sobald die Betriebsarten umgeschaltet werden.

Schließlich werden die Zeilen bis zum Ende der Datei gelesen und wiedergegeben.

```
HTTP/1.1 200 OK
Date: Sun, 03 May 2015 19:41:49 GMT
Server: Apache
Last-Modified: Sat, 25 Apr 2015 23:28:56 GMT
Accept-Ranges: bytes
Content-Length: 3653
Vary: Accept-Encoding
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
    <META NAME=viewport CONTENT="width=device-width, initial-scale=1">
    <TITLE>The Icon Programming Language</TITLE>
    <LINK REL="STYLESHEET" TYPE="text/css" HREF="istyle.css">
    <LINK REL="SHORTCUT ICON" HREF="shortcut.gif">
    <STYLE TYPE="text/css"> H2 { padding-top: 1.3em; margin-top: 0; } </STYLE>
</HEAD>
<BODY>

<P><A HREF="http://www.cs.arizona.edu/icon/"> <IMG SRC="wwwcube.gif"
ALT="[Icon home]" WIDTH=128 HEIGHT=144 BORDER=0 ALIGN=RIGHT> </A>

<H1>The Icon Programming Language</H1>

<P>
Icon is a high-level, general-purpose programming language
<!-- runs on Unix-based systems including Linux, BSD, Macintosh, and Cygwin.
Icon is especially noted for its --> with
novel features including string scanning and goal-directed evaluation.

<P><A HREF="current/"><STRONG>Download Icon for Unix</STRONG></A>
(including Linux, Macintosh, and Cygwin)
<BR><A HREF="v95w.htm"><STRONG>Download Icon for Microsoft Windows</STRONG></A>

<H3> Latest news </H3>
<UL>
<LI> 17 Feb 2015:
    A binary-only release for <A HREF="v95w.htm">Windows</A>,
    without graphics, is now available.
<LI> 31 Dec 2014:
    On Mac OS X Yosemite, export DISPLAY=:0 to avoid graphics problems.
<LI> 4 Dec 2014:
    Icon source code is now on
    <A HREF="https://github.com/gtownsend/icon">GitHub</A>.
<LI> 31 Jul 2014:
    Version 9.5.1 has been revalidated on Mac OS X 10.9.4
    and Ubuntu 14.04.
<LI> 18 Mar 2014:
    Many more old reports have been added to the historical
    <A HREF="docs/docs.htm">tech report</A> collection.
<LI> 15 Aug 2013:
    Non-graphics Mac binaries are
    <A HREF="current/">available</A> for systems without X11.
<LI> 7 Jun 2013:
    There is a new <A HREF="mlist.htm">Icon-language mailing list</A>.
<LI> 6 Jun 2013:
    <A HREF="v951/">Version 9.5.1 of Icon</A> is now available.
</UL>

<HR>

<DIV CLASS=twocols>
```

Abbildung 30-6: Screenshot zu test_tconnect.icn

```
<DIV CLASS=together>
<H2>Getting Started with Icon</H2>
<UL>
<LI>Ralph Griswold's <A HREF="docs/ipd266.htm">Overview</A>
<LI>Dave Hanson's <A HREF="intro.htm">Brief Introduction</A>
<LI>Bill Mitchell's <A HREF="http://www.mitchellsoftwareengineering.com/icon/">
   introduction and slides</A>
<LI>John Shipman's <A HREF="http://www.nmt.edu/tcc/help/lang/icon/">
   Tutorial</A> (at NM Tech)
</UL>
</DIV>

<DIV CLASS=together>
<H2>Current Documentation</H2>
<UL>
<LI><A HREF="faq.htm">Frequently Asked Questions</A>
<LI><A HREF="books.htm">Books About Icon</A>
<LI><A HREF="current/docguide.htm">Current Release Documentation</A>
<LI><A HREF="library/ipl.htm">Program Library Indexes</A>
<LI><A HREF="refernce/ref.htm">Language Reference</A>
</UL>
</DIV>
<DIV CLASS=together>
<H2>Older Documentation</H2>
<UL>
<LI><A HREF="docs/docs.htm">Technical Reports</A>
<LI><A HREF="inl/inl.htm">The Icon Newsletter</A>
<LI><A HREF="analyst/ia.htm">The Icon Analyst</A>
<LI><A HREF="progcorn.htm">Programming Corner</A>
</UL>
<UL>
<LI><A HREF="progvis/progvis.htm">Program Visualization</A>
<LI><A HREF="oddsends/other.htm">Odds and Ends</A>
</UL>
</DIV>
<DIV CLASS=together>
<H2>Miscellany</H2>
<UL>

<LI><A HREF="mlist.htm">Icon-language mailing list</A>
<LI><A HREF="implver.htm">Implementations</A>
<LI><A HREF="ftp://ftp.cs.arizona.edu/icon/">Icon FTP Site</A>
<LI><A HREF="links.htm">Icon Links</A>
</UL>

<UL>
<LI><A HREF="http://www.cs.arizona.edu/">Department of Computer Science</A>
<LI><A href="http://www.cs.arizona.edu/news/articles/200610-memorial.html">
    Ralph E. Griswold Memorial Endowment</a>
</UL>
</DIV>

</DIV>

<HR>
Contact us:
<A HREF="mailto:icon-project@cs.arizona.edu">icon-project@cs.arizona.edu</A>
</BODY>
</HTML>
```

Abbildung 30-7: Screenshot zu test_tconnect.icn

30.3 Ladbare C-Funktionen schreiben

30.3.1 Einführung

Lasst uns jetzt die Konstruktion von Library-Funktionen in C betrachen. Weil das Icon-System von C-Funktionen erwartet, dass sie eine gewisse Schnittstelle implementieren, erfordert dynamisches Laden speziell geschriebene C-Funktionen. Allgemein ist es nicht möglich, eine vorhandene C-Funktion zu nutzen, ohne eine dazwischenliegende „Klebe"-Funktion zu schreiben.

C-Funktionen müssen Datentypen behandeln, die vom Icon-Laufzeitsystem verwendet werden, insbesondere sog. *Deskriptoren*, die alle Icon-Werte repräsentieren. Während ein Verständnis von Icons Laufzeitsystem (näheres siehe unter: www.cs.arizona.edu/icon/ibsale.htm) hilfreich ist, ist es möglich, nützliche Funktionen zu erzeugen, indem vorhandene Library-Funktionen nachmodelliert werden. Ganzzahlen und Zeichenketten werden am einfachsten behandelt.

Eine ladbare C-Funktion hat den Prototyp

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

wobei `argc` die Anzahl der Argumente darstellt und `argv` ein Array von Argument-Deskriptoren angibt. Das erste Element `argv[0]` wird verwendet, um einen Icon-Wert zurückzugeben und wird zu einem Deskriptor für den Null-Wert initialisiert.

Dieses Element ist in der Anzahl `argc` nicht eingeschlossen. Die tatsächlichen Argumente beginnen mit `argv[1]`.

Anmerkung: Arrays beginnen in der Programmiersprache Icon mit dem Index 1 – in der Programmiersprache C dagegen mit dem Index 0.

Falls die C-Funktion 0 zurückgibt, dann ist der Aufruf aus Icon heraus erfolgreich. Ein negativer Wert zeigt ein Scheitern an. Falls ein positiver Wert zurückgegeben wird, wird er als Fehlernummer interpretiert und ein fataler Fehler mit dieser Nummer wird signalisiert. In diesem Fall, falls `argv[0]` ungleich `&null` ist, wird er als der angetroffene Wert betrachtet. Für eine C-Funktion gibt es keinen Weg auszusetzen und keinen Weg einen null-Wert als einen angetroffenen Wert im Fall eines Fehlers anzuzeigen.

30.3.2 Schnittstellen-Makros: `icall.h`

Die C-Datei `icall.h` enthält eine Menge Makros zum Schreiben ladbarer Funktionen. Eine Dokumentation ist in den Kommentaren enthalten. Diese Datei befindet sich im Verzeichnis `cfuncs` im Quellcode der Icon-Programm-Library. Alternativ kann sie heruntergeladen werden:

```
wget www.cs.arizona.edu/icon/library/src/cfuncs/icall.h
```

Listing home/pi/icon9_51/ipl/cfuncs/icall.h

```
1  /*
2  ############################################################################
3  #
4  #   File:     icall.h
5  #
6  #   Subject:  Definitions for external C functions
7  #
8  #   Author:   Gregg M. Townsend
9  #
10 #   Date:     October 29, 2009
11 #
12 ############################################################################
13 #
14 #    This file is in the public domain.
15 #
16 ############################################################################
17 #
18 #    Contributors: Kostas Oikonomou, Carl Sturtivant
19 #
20 ############################################################################
21 #
22 #    These definitions assist in writing external C functions for use with
23 #    Version 9 of Icon.
24 #
25 ############################################################################
26 #
27 #    From Icon, loadfunc(libfile, funcname) loads a C function of the form
28 #   int func(int argc, descriptor argv[])
29 #    where "descriptor" is the structure type defined here.  The C
30 #    function returns -1 to fail, 0 to succeed, or a positive integer
31 #    to report an error.  Argv[1] through argv[argc] are the incoming
32 #    arguments; the return value on success (or the offending value
```

```
#   in case of error) is stored in argv[0].
#
#   In the macro descriptions below, d is a descriptor value, typically
#   a member of the argv array.  IMPORTANT: many macros assume that the
#   C function's parameters are named "argc" and "argv" as noted above.
#
############################################################################
#
#   IconType(d) returns one of the characters {cfinprsCEILRST} indicating
#   the type of a value based on the key on page 273 of the Blue Book (The
#   Icon Programming Language).  The character E indicates external data;
#   The character I indicates a large (multiprecision) integer.
#
#   Only a few of these types (i, r, f, s, E) are easily manipulated in C.
#   Given that the type has been verified, the following macros return
#   a value from a descriptor in C terms:
#
#  IntegerVal(d)  value of a integer (type 'i') as a C long
#  RealVal(d)     value of a real (type 'r') as a C double
#  FileVal(d)     value of a file (type 'f') as a C FILE pointer
#  FileStat(d)    status field of a file
#  StringVal(d)   value of a string (type 's') as a C char pointer
#        (copied if necessary to add \0 for termination)
#
#  StringAddr(d)  address of possibly unterminated string
#  StringLen(d)   length of string
#
#  ListLen(d)     length of list
#  ExternalBlock(d) address of heap block for external data
#
#   These macros check the type of an argument, converting if necessary,
#   and returning an error code if the argument is wrong:
#
#  ArgInteger(i)      check that argv[i] is an integer
#  ArgReal(i)         check that argv[i] is a real number
#  ArgString(i)       check that argv[i] is a string
#  ArgList(i)         check that argv[i] is a list
#  ArgExternal(i,f)   check that argv[i] is an external w/ funcblock f
#
#   Caveats:
#      Allocation failure is not detected.
#
############################################################################
#
#   These macros return from the C function back to Icon code:
#
#  Return             return argv[0] (initially &null)
#  RetArg(i)      return argv[i]
#  RetNull()      return &null
#  RetInteger(i)      return integer value i
#  RetReal(v)         return real value v
#  RetFile(fp,status,name)return (newly opened) file
#  RetExternal(e)     return block at addr e made by alcexternal()
#  RetString(s)       return null-terminated string s
#  RetStringN(s, n)   return string s whose length is n
#  RetAlcString(s, n) return already-allocated string
#  RetConstString(s)  return constant string s
#  RetConstStringN(s, n)  return constant string s of length n
#  Fail           return failure status
#  Error(n)       return error code n
#  ArgError(i,n)      return argv[i] as offending value for error n
#
############################################################################
 */

#include <stdio.h>
#include <limits.h>

#if INT_MAX == 32767
#define WordSize 16
#elif LONG_MAX == 2147483647L
#define WordSize 32
#else
#define WordSize 64
#endif

#if WordSize <= 32
#define F_Nqual    0x80000000      /* set if NOT string qualifier */
#define F_Var      0x40000000      /* set if variable */
#define F_Ptr      0x10000000      /* set if value field is pointer */
#define F_Typecode 0x20000000      /* set if dword includes type code */
#else
#define F_Nqual    0x8000000000000000   /* set if NOT string qualifier */
```

```
116 #define F_Var      0x4000000000000000   /* set if variable */
117 #define F_Ptr      0x1000000000000000   /* set if value field is pointer */
118 #define F_Typecode 0x2000000000000000   /* set if dword includes type code */
119 #endif
120
121 #define D_Typecode    (F_Nqual | F_Typecode)
122
123 #define T_Null         0       /* null value */
124 #define T_Integer  1       /* integer */
125 #define T_Real         3       /* real number */
126 #define T_File         5       /* file, including window */
127 #define T_External    19       /* externally defined data */
128
129 #define D_Null        (T_Null     | D_Typecode)
130 #define D_Integer (T_Integer  | D_Typecode)
131 #define D_Real        (T_Real     | D_Typecode | F_Ptr)
132 #define D_File        (T_File     | D_Typecode | F_Ptr)
133 #define D_External    (T_External | D_Typecode | F_Ptr)
134
135 #define Fs_Read       0001     /* file open for reading */
136 #define Fs_Write  0002     /* file open for writing */
137 #define Fs_Pipe       0020     /* file is a [popen] pipe */
138 #define Fs_Window 0400     /* file is a window */
139
140
141 typedef long word;
142 typedef struct { word dword, vword; } descriptor;
143 typedef struct { word title; double rval; } realblock;
144 typedef struct { word title; FILE *fp; word stat; descriptor fname; } fileblock;
145 typedef struct { word title, size, id; void *head, *tail; } listblock;
146
147 typedef struct externalblock {      /* standard header for external block */
148    word title, size, id;
149    struct funclist *funcs;
150    word data[];
151 } externalblock;
152
153 typedef struct funclist {
154    int (*extlcmp)  (int argc, descriptor argv[]);
155    int (*extlcopy) (int argc, descriptor argv[]);
156    int (*extlname) (int argc, descriptor argv[]);
157    int (*extlimage)(int argc, descriptor argv[]);
158 } funclist;
159
160
161 char *alcstr(char *s, word len);
162 realblock *alcreal(double v);
163 fileblock *alcfile(FILE *fp, int stat, descriptor *name);
164 externalblock *alcexternal(long nbytes, funclist *f, void *data);
165
166 int cnv_c_str(descriptor *s, descriptor *d);
167 int cnv_int(descriptor *s, descriptor *d);
168 int cnv_real(descriptor *s, descriptor *d);
169 int cnv_str(descriptor *s, descriptor *d);
170 double getdbl(descriptor *d);
171
172 extern descriptor nulldesc;         /* null descriptor */
173
174
175 #define IconType(d) ((d).dword>=0 ? 's' : "niIrcfpRL.S.T.....CE"[(d).dword&31])
176
177
178 #define IntegerVal(d) ((d).vword)
179
180 #define RealVal(d) getdbl(&(d))
181
182 #define FileVal(d) (((fileblock *)((d).vword))->fp)
183 #define FileStat(d) (((fileblock *)((d).vword))->stat)
184
185 #define StringAddr(d) ((char *)(d).vword)
186 #define StringLen(d) ((d).dword)
187
188 #define StringVal(d) \
189 (*(char*)((d).vword+(d).dword) ? cnv_c_str(&(d),&(d)) : 0, (char*)((d).vword))
190
191 #define ListLen(d) (((listblock *)((d).vword))->size)
192
193 #define ExternalBlock(d) ((externalblock *)(d).vword)
194
195
196 #define ArgInteger(i) do { if (argc < (i)) Error(101); \
197 if (!cnv_int(&argv[i],&argv[i])) ArgError(i,101); } while (0)
198
```

```
199 #define ArgReal(i) do { if (argc < (i)) Error(102); \
200 if (!cnv_real(&argv[i],&argv[i])) ArgError(i,102); } while (0)
201
202 #define ArgString(i) do { if (argc < (i)) Error(103); \
203 if (!cnv_str(&argv[i],&argv[i])) ArgError(i,103); } while (0)
204
205 #define ArgList(i) \
206 do {if (argc < (i)) Error(108); \
207 if (IconType(argv[i]) != 'L') ArgError(i,108); } while(0)
208
209 #define ArgExternal(i,f) \
210 do {if (argc < (i)) Error(131); \
211 if (IconType(argv[i]) != 'E') ArgError(i,131); \
212 if (ExternalBlock(argv[i])->funclist != (f)) ArgError(i,132); \
213 } while(0)
214
215
216 #define RetArg(i) return (argv[0] = argv[i], 0)
217
218 #define RetNull() return (argv->dword = D_Null, argv->vword = 0)
219
220 #define RetInteger(i) return (argv->dword = D_Integer, argv->vword = i, 0)
221
222 #define RetReal(v) return (argv->dword=D_Real, argv->vword=(word)alcreal(v), 0)
223
224 #define RetFile(fp,stat,name) \
225 do { descriptor dd; dd.vword = (word)alcstr(name, dd.dword = strlen(name)); \
226    argv->dword = D_File; argv->vword = (word)alcfile(fp, stat, &dd); \
227    return 0; } while (0)
228
229 #define RetExternal(e) return (argv->dword=D_External, argv->vword=(word)(e), 0)
230
231 #define RetString(s) \
232 do { word n = strlen(s); \
233 argv->dword = n; argv->vword = (word)alcstr(s,n); return 0; } while (0)
234
235 #define RetStringN(s,n) \
236 do { argv->dword = n; argv->vword = (word)alcstr(s,n); return 0; } while (0)
237
238 #define RetConstString(s) return (argv->dword=strlen(s), argv->vword=(word)s, 0)
239
240 #define RetConstStringN(s,n) return (argv->dword=n, argv->vword=(word)s, 0)
241
242 #define RetAlcString(s,n) return (argv->dword=n, argv->vword=(word)s, 0)
243
244
245 #define Fail return -1
246 #define Return return 0
247 #define Error(n) return n
248 #define ArgError(i,n) return (argv[0] = argv[i], n)
```

Makros werden für:

- den Typ eines Icon-Wertes zu prüfen
- den Typ eines Argumentes zu bestätigen
- einen Icon-Wert in einen C-Wert umzuwandeln
- einen C-Wert im Icon-Format zurückzugeben
- Scheitern oder einen Fehler signalisieren

zur Verfügung gestellt. Die meisten Makros behandeln Ganzzahlen oder Zeichenketten. Ein wenig Unterstützung wird ebenso für die Behandlung von Fließpunkt-Werten und Datei-Werten bereitgestellt. Die Makros erwarten von den C-Argumenten, dass sie mit den Namen `argc` und `argv` deklariert werden.

30.3.3 Eine individuelle Library erstellen

30.3.3.1 Nochmals Bitcounting

Als konkretes Beispiel für eine C-Funktion betrachten wir den Quellcode der vorhin verwendeten Funktion `bitcount()`.

Listing home/pi/icon9_51/ipl/cfuncs/bitcount.c

```
1  #include "icall.h"
2
3  int bitcount(int argc, descriptor *argv)
4  {  unsigned long v;
5     int n;
6
7     ArgInteger(1);
8     v = IntegerVal(argv[1]);
9     n = 0;
10    while (v != 0)
11    {     n += v & 1;
12          v >> 1;
13    }
14    RetInteger(n);
15 }
```

Wie alle ladbaren Funktionen ist auch `bitcount()` eine Ganzzahl-Funktion mit den beiden Parame-

tern `argc` und `argv`.

Der Makro-Aufruf `ArgInteger` verifiziert, dass das Argument 1 eine einfache Ganzzahl ist. Große Ganzzahlen werden von C-Funkionen typischerweise wegen aufwändiger Extra-Arbeit zurückgewiesen. Falls das Argument 1 fehlt oder vom falschen Datentyp ist, dann bewirkt `ArgInteger`, dass die Funktion den Fehlercode 101 (`Ganzzahl erwartet oder außerhalb des Bereiches`) zurückgibt.

Der Makro-Aufruf `IntegerVal` extrahiert den Wert des ersten Argumentes. In jedem Durchlauf der `while`-Schleife wird das niedrigwertige Bit von `v` herausgezogen (`v & 1`), zu `n` addiert und herausgeschoben (`v >>= 1`). Sobald keine Bits ungleich 0 zurückbleiben, wird die Schleife verlassen. Beachte, dass `v` als nicht vorzeichenbehaftet deklariert wurde, um sicherzustellen, dass nur 0-Bits durch die Shift-Operation eingefügt werden.

Der Makro-Aufruf `RetInteger` gibt den Wert von `v` als einen Icon-Ganzzahl-Wert zurück.

30.3.3.2 Code zur Objektdatei compilieren

Um in einem Icon-Programm benutzt werden zu können, muss eine C-Funktion aufgebaut und in einer Library installiert sein. Zuerst erfolgt die Compilierung des Quellcodes zu einer Objektdatei `bitcount.o`, üblicherweise unter Beteiligung eines Kommandos wie:

```
cc -c -fPIC bitcount.c -o bitcount.o
```

Die Option `-c` bewirkt, dass eine verschiebbare Objektdatei anstelle eines eigenständigen ausführbaren Programmes erzeugt wird. Falls sich `icall.h` nicht im aktuellen Verzeichnis befindet, dann wird ein zusätzlicher Ort benötigt, um dessen Position anzugeben. Andere Optionen wie z.B. Optimierungsoptionen können ebenso angegeben werden.

Abb. 30-8 ist unter Einsatz der Option -v entstanden, um überhaupt eine Ausgabe von cc zu erhalten. Nach Eingabe von

```
ls -all bitcount.o
```

erhält man

```
-rw-r--r-- 1 pi pi 1100 Mai 10 15:38 bitcount.o
```

Daraus folgt, dass der Compiler `cc` erfolgreich die Objektdatei `bitcount.o` erstellt hat. Wären Compilierfehler aufgetreten, dann wären zum Einen diese Compilierfehler angezeigt und zum Anderen keine Objektdatei erzeugt worden.

30.3.3.3 Objektdatei zur Library umwandeln

Die Library kann auf zwei Möglichkeiten erzeugt werden:

a) durch den C-Compiler

Eine C-Funktion kann nur von einer verteilten Bibliothek (*shared Library*) geladen werden. Sogar, wenn es nur eine Funktion gibt, muss sie in eine Library verfrachtet werden. Library-Namen enden übereinkunftsgemäß mit `.so`. Es scheint so, dass jedes Betriebssystem einen unterschiedlichen Weg aufweist, Libraries zu erzeugen, üblicherweise unter Beteiligung besonderer Flags zu `cc` oder `ld`.

Auf einem Linux-System wie dem Raspberry Pi erhält man eine Library (Datei mit der Endung `.so`) mit folgendem Linux-Kommando:

```
cc bitcount.o -shared -o mylib.so
```

Abb. 30-9 ist unter Einsatz der Option -v entstanden, um überhaupt eine Ausgabe von cc zu erhalten. Nach Eingabe von

```
ls -all mylib.so
```

erhält man

```
-rwxr-xr-x 1 pi pi 4924 Mai 10 16:19 mylib.so
```

b) durch Nutzung eines Skriptes

Das Shell-Skript `mklib.sh` enthält unser betriebssystem-übergreifendes Verständnis der Erzeugung verteilter Libraries. Es nimmt ein Argument, das die Library benennt, die erzeugt werden soll, und ein oder mehrere Argument(e), das/die Dateinamen auflisten.

Z.B. erzeugt das Kommand

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

eine Datei `mylib.so`, die die Funktionen enthält, die aus `bitcount.o` gelesen wurden. Wie `icall.h` ist auch `mklib.sh` im Quellcode der Programm-Library oder vom Web verfügbar (s. Listing).

```
wget www.cs.arizona.edu/icon/library/src/cfuncs/mklib.sh
```

Und auch hier erhalten wir eine Bildschirmausgabe von

```
+ cc .shared -o mylib.so -fPIC bitcount.o
```

nach dessen Ablauf wir uns vom Ergebnis überzeugen, indem wir eingeben

```
ls -all mylib.so
```

und als Ergebnis diese Ausgabe erhalten:

```
-rwxr-xr-x 1 pi pi 4924 Mai 10 16:26 mylib.so
```

30.3.3.4 Library direkt aus dem Quellcode erzeugen

Die .so-Datei lässt sich auch direkt aus dem Quellcode erzeugen durch

```
cc -shared -o mylib.so -fPIC bitcount.c
```

Der Inhalt von Abb. 30-10 ist wiederum mittels der Option -v entstanden.

```
Using built-in specs.
COLLECT_GCC=cc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/arm-linux-gnueabihf/4.6/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: ../src/configure -v --with-pkgversion='Debian 4.6.3-14+rpi1' --with-
bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++
--prefix=/usr --program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib
--libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-
dir=/usr/include/c++/4.6 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu
--enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin
--enable-objc-gc --disable-sjlj-exceptions --with-arch=armv6 --with-fpu=vfp --with-float=hard
--enable-checking=release --build=arm-linux-gnueabihf --host=arm-linux-gnueabihf --target=arm-
linux-gnueabihf
Thread model: posix
gcc version 4.6.3 (Debian 4.6.3-14+rpi1)
COLLECT_GCC_OPTIONS='-v' '-c' '-fPIC' '-o' 'bitcount.o' '-march=armv6' '-mfloat-abi=hard' '-
mfpu=vfp'
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/cc1 -quiet -v -imultilib . -imultiarch arm-linux-gnueabihf
bitcount.c -quiet -dumpbase bitcount.c -march=armv6 -mfloat-abi=hard -mfpu=vfp -auxbase-strip
bitcount.o -version -fPIC -o /tmp/ccZaoq1g.s
GNU C (Debian 4.6.3-14+rpi1) version 4.6.3 (arm-linux-gnueabihf)
   compiled by GNU C version 4.6.3, GMP version 5.0.5, MPFR version 3.1.0-p10, MPC version 0.9
GGC heuristics: --param ggc-min-expand=59 --param ggc-min-heapsize=55717
ignoring nonexistent directory "/usr/local/include/arm-linux-gnueabihf"
ignoring nonexistent directory "/usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../../arm-linux-
gnueabihf/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/include
 /usr/local/include
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/include-fixed
 /usr/include/arm-linux-gnueabihf
 /usr/include
End of search list.
GNU C (Debian 4.6.3-14+rpi1) version 4.6.3 (arm-linux-gnueabihf)
   compiled by GNU C version 4.6.3, GMP version 5.0.5, MPFR version 3.1.0-p10, MPC version 0.9
GGC heuristics: --param ggc-min-expand=59 --param ggc-min-heapsize=55717
Compiler executable checksum: cf78bfe2d99d4ca6e0637950db1b4acd
COLLECT_GCC_OPTIONS='-v' '-c' '-fPIC' '-o' 'bitcount.o' '-march=armv6' '-mfloat-abi=hard' '-
mfpu=vfp'
 as -march=armv6 -mfloat-abi=hard -mfpu=vfp -meabi=5 -o bitcount.o /tmp/ccZaoq1g.s
COMPILER_PATH=/usr/lib/gcc/arm-linux-gnueabihf/4.6/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/:/usr/lib/gcc/arm-linux-gnueabihf/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/:/usr/lib/gcc/arm-linux-gnueabihf/
LIBRARY_PATH=/usr/lib/gcc/arm-linux-gnueabihf/4.6/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../../arm-linux-gnueabihf/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../../:/lib/arm-linux-gnueabihf/:/lib/:/usr/lib/arm-linux-gnueabihf/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-c' '-fPIC' '-o' 'bitcount.o' '-march=armv6' '-mfloat-abi=hard' '-
mfpu=vfp'
```

Abbildung 30-8: Screenshot zu `cc -v -c -fPIC bitcount.c -o bitcount.o`

```
Using built-in specs.
COLLECT_GCC=cc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/arm-linux-gnueabihf/4.6/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: ../src/configure -v --with-pkgversion='Debian 4.6.3-14+rpi1' --with-
bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++
--prefix=/usr --program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib
--libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-
dir=/usr/include/c++/4.6 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu
--enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin
--enable-objc-gc --disable-sjlj-exceptions --with-arch=armv6 --with-fpu=vfp --with-float=hard
--enable-checking=release --build=arm-linux-gnueabihf --host=arm-linux-gnueabihf --target=arm-
linux-gnueabihf
Thread model: posix
gcc version 4.6.3 (Debian 4.6.3-14+rpi1)
COMPILER_PATH=/usr/lib/gcc/arm-linux-gnueabihf/4.6/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/:/usr/lib/gcc/arm-linux-gnueabihf/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/:/usr/lib/gcc/arm-linux-gnueabihf/
LIBRARY_PATH=/usr/lib/gcc/arm-linux-gnueabihf/4.6/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../../arm-linux-gnueabihf/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../../:/lib/arm-linux-gnueabihf/:/lib/:/usr/lib/arm-linux-gnueabihf/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-shared' '-o' 'mylib.so' '-march=armv6' '-mfloat-abi=hard' '-mfpu=vfp'
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/collect2 --sysroot=/ --build-id --no-add-needed --eh-frame-
hdr -shared -dynamic-linker /lib/ld-linux-armhf.so.3 -X --hash-style=both -m armelf_linux_eabi -o
mylib.so /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf/crti.o
/usr/lib/gcc/arm-linux-gnueabihf/4.6/crtbeginS.o -L/usr/lib/gcc/arm-linux-gnueabihf/4.6
-L/usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf -L/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../.. -L/lib/arm-linux-gnueabihf -L/usr/lib/arm-linux-gnueabihf bitcount.o -lgcc
--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/arm-
linux-gnueabihf/4.6/crtendS.o /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-
gnueabihf/crtn.o
```

Abbildung 30-9: Screenshot zu `cc -v bitcount.o -shared -o mylib.so`

Listing home/pi/icon9_51/ipl/cfuncs/mklib.sh

```
1  #!/bin/sh
2  #
3  #  mklib libname.so obj.o ...
4
5  CC=${CC-cc}
6  BIN=${BIN-../../bin}
7
8  LIBNAME=${1?"usage: $0 libname obj..."}
9  shift
10
11 SYS=`uname -s`
12 set -x
13 case "$SYS" in
14    Linux*|*BSD*|GNU*)
15       $CC -shared -o $LIBNAME -fPIC "$@";;
16    CYGWIN*)
17       # move the win32 import library for iconx.exe callbacks
18       # created when iconx.exe was built
19       if [ -e $BIN/../src/runtime/iconx.a ]; then
20          mv $BIN/../src/runtime/iconx.a $BIN
21       fi
22       $CC -shared -Wl,--enable-auto-import -o $LIBNAME "$@" $BIN/iconx.a;;
23    Darwin*)
24       $CC -bundle -undefined suppress -flat_namespace -o $LIBNAME "$@";;
25    SunOS*)
26       $CC $CFLAGS -G -o $LIBNAME "$@" -lc -lsocket;;
27    HP-UX*)
28       ld -b -o $LIBNAME "$@";;
29    IRIX*)
30       ld -shared -o $LIBNAME "$@";;
31    OSF*)
32       ld -shared -expect_unresolved '*' -o $LIBNAME "$@" -lc;;
33    AIX*)
34       # this may not be quite right; it doesn't seem to work yet...
35       ld -bM:SRE -berok -bexpall -bnoentry -bnox -bnogc -brtl -o $LIBNAME "$@";;
36    *)
37       set -
38       echo 1>&2 "don't know how to make libraries under $SYS"
39       exit 1;;
40 esac
```

```
Using built-in specs.
COLLECT_GCC=cc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/arm-linux-gnueabihf/4.6/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: ../src/configure -v --with-pkgversion='Debian 4.6.3-14+rpi1' --with-
bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++
--prefix=/usr --program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib
--libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-
dir=/usr/include/c++/4.6 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu
--enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin
--enable-objc-gc --disable-sjlj-exceptions --with-arch=armv6 --with-fpu=vfp --with-float=hard
--enable-checking=release --build=arm-linux-gnueabihf --host=arm-linux-gnueabihf --target=arm-
linux-gnueabihf
Thread model: posix
gcc version 4.6.3 (Debian 4.6.3-14+rpi1)
COLLECT_GCC_OPTIONS='-v' '-shared' '-o' 'mylib.so' '-fPIC' '-march=armv6' '-mfloat-abi=hard' '-
mfpu=vfp'
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/cc1 -quiet -v -imultilib . -imultiarch arm-linux-gnueabihf
bitcount.c -quiet -dumpbase bitcount.c -march=armv6 -mfloat-abi=hard -mfpu=vfp -auxbase bitcount
-version -fPIC -o /tmp/ccgYqHkJ.s
GNU C (Debian 4.6.3-14+rpi1) version 4.6.3 (arm-linux-gnueabihf)
   compiled by GNU C version 4.6.3, GMP version 5.0.5, MPFR version 3.1.0-p10, MPC version 0.9
GGC heuristics: --param ggc-min-expand=59 --param ggc-min-heapsize=55717
ignoring nonexistent directory "/usr/local/include/arm-linux-gnueabihf"
ignoring nonexistent directory "/usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../../arm-linux-
gnueabihf/include"
#include "..." search starts here:
#include <...> search starts here: cc -shared -o mylib.so -fPIC bitcount.c
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/include
 /usr/local/include
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/include-fixed
 /usr/include/arm-linux-gnueabihf
 /usr/include
End of search list.
GNU C (Debian 4.6.3-14+rpi1) version 4.6.3 (arm-linux-gnueabihf)
   compiled by GNU C version 4.6.3, GMP version 5.0.5, MPFR version 3.1.0-p10, MPC version 0.9
GGC heuristics: --param ggc-min-expand=59 --param ggc-min-heapsize=55717
Compiler executable checksum: cf78bfe2d99d4ca6e0637950db1b4acd
COLLECT_GCC_OPTIONS='-v' '-shared' '-o' 'mylib.so' '-fPIC' '-march=armv6' '-mfloat-abi=hard' '-
mfpu=vfp'
 as -march=armv6 -mfloat-abi=hard -mfpu=vfp -meabi=5 -o /tmp/cc3dPaMU.o /tmp/ccgYqHkJ.s
COMPILER_PATH=/usr/lib/gcc/arm-linux-gnueabihf/4.6/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/:/usr/lib/gcc/arm-linux-gnueabihf/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/:/usr/lib/gcc/arm-linux-gnueabihf/
LIBRARY_PATH=/usr/lib/gcc/arm-linux-gnueabihf/4.6/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../../arm-linux-gnueabihf/:/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../../:/lib/arm-linux-gnueabihf/:/lib/:/usr/lib/arm-linux-gnueabihf/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-shared' '-o' 'mylib.so' '-fPIC' '-march=armv6' '-mfloat-abi=hard' '-
mfpu=vfp'
 /usr/lib/gcc/arm-linux-gnueabihf/4.6/collect2 --sysroot=/ --build-id --no-add-needed --eh-frame-
hdr -shared -dynamic-linker /lib/ld-linux-armhf.so.3 -X --hash-style=both -m armelf_linux_eabi -o
mylib.so /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf/crti.o
/usr/lib/gcc/arm-linux-gnueabihf/4.6/crtbeginS.o -L/usr/lib/gcc/arm-linux-gnueabihf/4.6
-L/usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf -L/usr/lib/gcc/arm-linux-
gnueabihf/4.6/../../.. -L/lib/arm-linux-gnueabihf -L/usr/lib/arm-linux-gnueabihf /tmp/cc3dPaMU.o
-lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/arm-linux-gnueabihf/4.6/crtendS.o /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-
linux-gnueabihf/crtn.o
```

Abbildung 30-10: Screenshot zu `cc -shared -o mylib.so -fPIC bitcount.c`

30.3.4 Erzeugung der Library `libcfunc.so`

Wer die bereits des öferen genutzte Library `libcfunc.so` erzeugen oder erweitern möchte, erstellt einfach wie in Abschnitt 30.3.3 vorgestellt einen Quellcode in C, berücksichtigt Schnittstellen, damit das in der Programmiersprache

Icon geschriebene Programm Parameter übergeben und erhalten kann, und compiliert diesen Code zu einer Objektdatei. Bis hier also nichts Neues.

Um jedoch alle vorhandenen Objektdateien in `libcfuns.so` zusammenzufassen, ist noch ein einfacher Schritt zusätzlich erforderlich.

```
cd /home/pi/ipl/cfunc
make
```

`make` öffnet die Datei `Makefile` im aktuellen Verzeichnis. Diese Datei wird üblicherweise für einen bestimmten Zweck erstellt. Wir nutzen hier einfach die bereits enthaltene Datei und passen diese allerhöchstens an, indem wir unsere eigene Objektdatei ergänzen.

Die neue Bibliothek `libcfuncs.so` wird mit einem kleinen Testporogramm auf Funktionsfähigkeit gestestet und dann in das Verzeichnis `/home/pi/icon9_51/bin` kopiert, da sich diese Blibliothek standardmäßig dort befindet und innerhalb der in diesem Tutorial-Teil vorgestellten Befehle als Erstes (nach dem aktuellen Verzeichnis) durchsucht wird.

Listing home/pi/icon9_51/ipl/cfuncs/Makefile

```
1  #  Makefile for the dynamically loaded C function library.
2  #
3  #  If building with the compiler (instead of the interpreter)
4  #  use the "-fs" option to avoid problems.
5
6
7  include ../../Makedefs
8
9  ICONT = icont
10 IFLAGS = -us
11
12 FUNCLIB = libcfunc.so
13
14 .SUFFIXES: .c .o
15 .c.o:             ; $(CC) $(CFLAGS) $(CFDYN) -c $<
16
17 FUNCS = bitcount.o external.o files.o fpoll.o internal.o lgconv.o osf.o \
18    pack.o ppm.o process.o tconnect.o
19 CSRC = $(FUNCS:.o=.c)
20
21
22 default: cfunc.u2 $(FUNCLIB)
23
24
25 # library
26
27 $(FUNCLIB):  $(FUNCS) mklib.sh
28     CC="$(CC)" CFLAGS="$(CFLAGS)" BIN="../../bin" \
29         sh mklib.sh $(FUNCLIB) $(FUNCS)
30 $(FUNCS):    icall.h
31
32
33 # Icon interface
34
35 cfunc.u2:    cfunc.icn
36     $(ICONT) $(IFLAGS) -c cfunc.icn
37 cfunc.icn:   $(CSRC) mkfunc.sh
38     sh mkfunc.sh $(FUNCLIB) $(FUNCS) >cfunc.icn
39
40
41 # cleanup
42
43 clean Clean:
44     rm -f $(FUNCLIB) *.o *.u? *.so so_locations cfunc.icn
```

#	Erläuterungen
12	Definition des Namens der zu erzeugenden Library mit der Endung `.so`.
17-18	Auflistung der zu berücksichtigenden Objektdateien. Weitere Zeilen werden durch das Zeichen \ angegeben.

Dieses Shell-Skript macht Gebrauch von einem anderen Shellskript `mkfunc.sh`, das unter anderem die Datei `cfunc.icn` erzeugt. Damit schließt sich der Kreis – denn diese Datei haben wir bereits in Abschnitt 30.2.2 verwendet. Der Vollständigkeit halber sind das Shell-Skript `mkfunc.sh` als auch die davon erzeugte Icon-Datei `cfunc.icn` aufgeführt.

Listing home/pi/icon9_51/ipl/cfuncs/mkfunc.sh

```
#!/bin/sh
#
#  mkfunc libname file.o ...
#
#  looks at the corresponding C files and generates an Icon procedure
#  corresponding to each C function header that matches the pattern below.
#
#  If a function name begins with "icon_", those characters are removed
#  to form the procedure name.  Otherwise, the name is copied verbatim.

LIB=${1?"usage: $0 libname obj..."}
shift

cat <<ENDHDR
############################################################################
#
#   File:     cfunc.icn
#
#   Subject:  Procedures implemented in C
#
#   Author:   Gregg M. Townsend
#
#   Date:     June 8, 2001
#
############################################################################
#
#      These Icon procedures transparently load and execute functions
#  implemented in C.  Each procedure is a simple stub.  The first call
#  to a stub causes it to replace itself with the corresponding
#  dynamically loaded C function, after which the C function processes
#  the arguments and returns a result (or fails).  Subsequent calls
#  go straight to the C function without involving the Icon stub.
#
#      C functions are loaded from a file "$LIB" that is found by
#  searching \$FPATH.  The default \$FPATH is set by iconx to include
#  this library.
#
############################################################################
#
#  Requires:  Dynamic loading
#
############################################################################

#  DO NOT EDIT THIS FILE DIRECTLY.
#  It was created mechanically by the shell file "$0".
#  Edit that instead.

link io

\$define LIB "$LIB"
ENDHDR

LC_ALL=POSIX
export LC_ALL

for i
do
   FNAME=`basename $i .o`
   echo ""
   echo "# $FNAME.c:"
   sed '
s/ / /g
s/^int  *//
/^[a-z][a-z0-9_]* *(.*argc.*argv.*).*\/\*:.*\*\//!d
s/\([a-z0-9_]*\).*)\(.*\)$/\1(a[])\2@   return(\1:=pathload(LIB,"\1"))!a;end/
s/^[a-z]/procedure &/
s/\([^"]\)icon_/\1/g
s/)[     ]*\/\*\(.*\)\*\/.*@/)              #\1@/
   ' $FNAME.c | tr '@' '\012'
done

echo ""
echo "#---"
```

Listing home/pi/icon9_51/ipl/cfuncs/cfunc.icn

```
1  ############################################################################
2  #
3  #   File:     cfunc.icn
4  #
5  #   Subject:  Procedures implemented in C
6  #
7  #   Author:   Gregg M. Townsend
8  #
9  #   Date:     June 8, 2001
10 #
11 ############################################################################
12 #
13 #      These Icon procedures transparently load and execute functions
14 #   implemented in C.  Each procedure is a simple stub.  The first call
15 #   to a stub causes it to replace itself with the corresponding
16 #   dynamically loaded C function, after which the C function processes
17 #   the arguments and returns a result (or fails).  Subsequent calls
18 #   go straight to the C function without involving the Icon stub.
19 #
20 #      C functions are loaded from a file "libcfunc.so" that is found by
21 #   searching $FPATH.  The default $FPATH is set by iconx to include
22 #   this library.
23 #
24 ############################################################################
25 #
26 #   Requires:  Dynamic loading
27 #
28 ############################################################################
29
30 #  DO NOT EDIT THIS FILE DIRECTLY.
31 #  It was created mechanically by the shell file "mkfunc.sh".
32 #  Edit that instead.
33
34 link io
35
36 $define LIB "libcfunc.so"
37
38 # bitcount.c:
39 procedure bitcount(a[])             #: count bits in an integer
40    return(bitcount:=pathload(LIB,"bitcount"))!a;end
41
42 # external.c:
43 procedure extxmin(a[])              #: create minimal external value
44    return(extxmin:=pathload(LIB,"extxmin"))!a;end
45 procedure extxstr(a[])              #: create string-valued external
46    return(extxstr:=pathload(LIB,"extxstr"))!a;end
47 procedure extxreal(a[])             #: create real-valued external
48    return(extxreal:=pathload(LIB,"extxreal"))!a;end
49
50 # files.c:
51 procedure chmod(a[])            #: change UNIX file permissions
52    return(chmod:=pathload(LIB,"icon_chmod"))!a;end
53 procedure umask(a[])            #: change UNIX permission mask
54    return(umask:=pathload(LIB,"icon_umask"))!a;end
55
56 # fpoll.c:
57 procedure fpoll(a[])            #: await data from file
58    return(fpoll:=pathload(LIB,"fpoll"))!a;end
59
60 # internal.c:
61 procedure dword(a[])            #: return descriptor d-word
62    return(dword:=pathload(LIB,"dword"))!a;end
63 procedure vword(a[])            #: return descriptor v-word
64    return(vword:=pathload(LIB,"vword"))!a;end
65 procedure descriptor(a[])           #: construct descriptor
66    return(descriptor:=pathload(LIB,"icon_descriptor"))!a;end
67 procedure peek(a[])             #: load value from memory
68    return(peek:=pathload(LIB,"peek"))!a;end
69 procedure spy(a[])              #: create spy-port to memory
70    return(spy:=pathload(LIB,"spy"))!a;end
71
72 # lgconv.c:
73 procedure lgconv(a[])               #: convert large integer to string
74    return(lgconv:=pathload(LIB,"lgconv"))!a;end
75
76 # osf.c:
77 procedure osftable(a[])             #: query OSF system table
78    return(osftable:=pathload(LIB,"osftable"))!a;end
79
80 # pack.c:
81 procedure pack(a[])             #: pack integer into bytes
82    return(pack:=pathload(LIB,"pack"))!a;end
```

```
83  procedure unpack(a[])              #: unpack integer from bytes
84     return(unpack:=pathload(LIB,"unpack"))!a;end
85
86  # ppm.c:
87  procedure ppmwidth(a[])            #: extract width of PPM string
88     return(ppmwidth:=pathload(LIB,"ppmwidth"))!a;end
89  procedure ppmheight(a[])           #: extract height of PPM string
90     return(ppmheight:=pathload(LIB,"ppmheight"))!a;end
91  procedure ppmmax(a[])              #: extract max of PPM string
92     return(ppmmax:=pathload(LIB,"ppmmax"))!a;end
93  procedure ppmdata(a[])             #: extract data from PPM string
94     return(ppmdata:=pathload(LIB,"ppmdata"))!a;end
95  procedure ppmstretch(a[])          #: stretch contrast of PPM string
96     return(ppmstretch:=pathload(LIB,"ppmstretch"))!a;end
97  procedure ppmsharpen(a[])          #: sharpen a PPM string
98     return(ppmsharpen:=pathload(LIB,"ppmsharpen"))!a;end
99  procedure ppm3x3(a[])              #: convolve PPM with matrix
100    return(ppm3x3:=pathload(LIB,"ppm3x3"))!a;end
101 procedure ppmimage(a[])            #: dither PPM to Icon image
102    return(ppmimage:=pathload(LIB,"ppmimage"))!a;end
103
104 # process.c:
105 procedure kill(a[])           #: kill process
106    return(kill:=pathload(LIB,"icon_kill"))!a;end
107 procedure getpid(a[])              #: query process ID
108    return(getpid:=pathload(LIB,"icon_getpid"))!a;end
109 procedure getuid(a[])              #: query user ID
110    return(getuid:=pathload(LIB,"icon_getuid"))!a;end
111 procedure getgid(a[])              #: query group ID
112    return(getgid:=pathload(LIB,"icon_getgid"))!a;end
113
114 # tconnect.c:
115 procedure tconnect(a[])            #: connect to TCP socket
116    return(tconnect:=pathload(LIB,"tconnect"))!a;end
117
118 #---
```

30.4 Zusammenfassung

- Icon-Programme können in C geschriebene Funktionen laden und aufrufen
- Die C-Funktionen müssen bzgl. Icon-Forderungen zugeschnitten sein
- Jede Funktion muss geladen worden sein, bevor sie aufgerufen werden kann
- Eine simple Icon-Prozedur kann verwendet werden, um die Lade-Details zu verbergen
- `pathload()` durchscht `FPATH`, um eine Funktionsbibliothek zu finden
- C-Funktionen können in unklaren Externwerten eingebettet werden
- Einige nützliche Funktionen werden in der Icon-Programm-Library (IPL) zur Verfügung gestellt

Ladbare Funktionen können Datenstrukturen erzeugen und zurückgeben, die als ein unklarer externer Typ durch Icon behandelt werden. Die Verwendung von Externwerten wird in Abschnitt 30.5 behandelt.

30.5 Icon-Externwerte

30.5.1 Einführung

Externwerte stellen einen Weg für dynamisch geladene C-Funktionen bereit, um unklare Datenstrukturen zu erzeugen und zu einem Icon-Programm zurückzugeben. Dies erlaubt es dem Zustand, zwischen mehrfachen Aufrufen geladener Funktionen aufrechterhalten zu bleiben.

Die Erzeugung eines Externwertes in C definiert nicht nur die Daten selbst sondern ebenso einige verwandte Merkmale. Diese steuern das Sortierverhalten, Image und andere solche Aspekte. Icon stellt Voreinstellungen zur Verfügung und fügt ebenso eine Serialnummer hinzu, ähnlich wie wir es bereits von von Listen, Mengen und Tabellen her kennen.

Die Angabe von Merkmalen erzeugt effektiv eine Anzahl unterschiedlicher Typen. Jede ladbare Library kann die Typen für ihre eigenen Zwecke festlegen. Aus einer objektorientierten Sichtweise können diese als Untertypen eines gemeinsamen `Extern`-Datentyps angesehen werden (s. auch www.cs.arizona.edu/icon/v951/extvals.htm).

30.5.2 Externwerte in einem Icon-Programm

Externwerte sind bzgl. Icon unklar. Sie werden nur durch dynamisch geladene C-Funktionen erzeugt, jede von denen mag umgekehrt Zugriff auf ihre internen Werten haben. Auf diese Weise ist es möglich, dass eine Sammlung von Icon-Funktionen mit einer besonderen Art von Externwerten arbeiten kann. Jedoch können solche Werte in Icon ausgegeben und geändert (zugewiesen, Prozeduren oder Funktionen übergeben werden, sortiert, kopiert etc.) werden. Das Verhalten eines Externwertes in Icon kann in einem gewissen Ausmaß durch Implementierung der dynamisch geladenen Funktion geändert werden, die benutzt wurde, diesen Externwert zu erzeugen, wie unten beschrieben.

Im Folgenden seien `E`, `E1`, `E2` Externwerte, die von irgendeiner/irgendwelchen geladenen Funktion/en erzeugt wurden. Icon schreibt das folgende Voreinstellungsverhalten solcher Werte vor, welches zur Schau gestellt wird, falls die Funktionen, die sie erzeugt haben, solche zum Zeitpunkt der Erzeugung nicht überschrieben haben. Anderweitiges Verhalten wird in den folgenden Zusammenhängen durch die Erzeuger der Externwerte festgelegt.

Beim Sortieren erscheinen Externwerte immer hinter den Werten aller anderen Datentypen. Innerhalb der Externwerte werden diese zuerst nach dem Typnamen sortiert (was eine Menge durch den Erzeuger darstellt, wird von der Icon-Funktion `type()` zurückgegeben und gibt den Untertyp des Externwertes an). Die voreingestellte Sortierung innerhalb eines Untertyps erfolgt durch die Serialnummer. Nur die Sortierung innerhalb eines Subtyps (d.h. Externwerte mit einem besonderen Typnamen) können durch den Erzeuger überschrieben werden. Siehe den nächsten Abschnitt für eine Beschreibung der relevanten C-Internals. Das voreingestellte Verhalten folgt.

- `type(E)` liefert die Zeichenkette `"external"`
- `image(E)` liefert die Zeichenkette, die Typ, Serialnummer und die Anzahl der Datenworte angibt, z.B. `"external_12(3)"`
- `copy(E)` gibt `E` ohne Kopieren zurück
- `sort()` Externwerte sortieren zuerst nach dem Typnamen, danach nach der Serialnummer
- `E1 === E2` erzeugt `E1`, sobald `E1` und `E2` das gleiche externe Objekt sind; scheitert ansonsten
- `E1 ~=== E2` erzeugt `E2` sobald `E1` und `E2` unterschiedlich sind; scheitert ansonsten.

30.5.3 Externwerte in C erzeugen und nutzen

Die nächsten Abschnitte beschreiben die C-Schnittstelle für einen Leser, der mit der Anwendung ladbarer Funktionen vertraut ist. Ein Externwert wird in Icon durch einen Deskriptor implementiert, der auf einen externen Block zeigt, der mehrere Komponenten enthält. Der Datenbereich und die Funktionsliste sind die wichtigsten dieser Komponenten.

Eine Ganzzahl zeigt die Größe des Datenbereiches an. Dies wird angegeben, sobald der externe Block erzeugt ist. Oft sind externe Daten ein einzelner Zeiger auf andere Daten (ein sog. *Handle*) und deswegen wird nur ein Wort benötigt.

Die Funktionsliste erlaubt es dem Programmierer, die voreingestellten Verhaltensweisen, die im vorherigen Abschnitt aufgeführt wurden, für alle Werte des gleichen Externtyps, zu überschreiben. Diese Liste ist eine Callback-Tabelle, die auf die vom Programmierer definierten C-Funktionen zeigt, wie unten mehr im Detail beschrieben wird. Die Funktionsliste agiert ebenso als eine eindeutige Typ-Bezeichnung, weil Externwerte mit unterschiedlichen Funktionslisten sich wie Werte unterschiedlicher Datentypen verhalten.

Eine dynamisch geladene C-Funktion alloziert einen Externwert durch Einbinden von `ipl/cfuncs/icall.h` und Aufrufen von `alcexternal`:

```
externalblock *alcexternal(long size,
                           funclist *funcs,
                           void *date)
```

alloziert und gibt einen Zeiger auf einen externen Block zurück.

`size` gibt die Anzahl der Bytes im gesamten externen Block an (und als Folge die Größe des Datenblocks darin)

`funcs`, falls nicht null, gibt eine Funktionsliste an,

die das voreingestellte Verhalten überschreibt. Siehe den nächsten Abschnitt.

`Data`, falls nicht null, gibt die Position der Daten an, in die kopiert wird, um den Datenblock zu initialisieren (bis er voll gelaufen ist).

Ein typischer Aufruf könnte sein:

```
blk = alcexternal(sizeof(externalblock) + sizeof(mydata), funcs, &mydata);
```

Das Ergebnis wird durch den Makroaufruf

```
RetExternal(blk);
```

zu Icon zurückgegeben. Der Block kann eventuell durch *Garbage-Collection* befreit werden, falls er nicht gespeichert wird oder falls das Icon-Programm später keinen Zugriff mehr darauf haben sollte.

Eine ladbare Funktion, die einen Externwert als Argument akzeptiert, kann

```
ArgExternal(i, f);
```

aufrufen, um `argv[i]` als Externwert des Typs zu validieren, der zur Funktionsliste `f` zugehörig ist und kann dann

```
blk = ExternalBlock(i);
```

aufrufen, um die Adresse des zugehörigen externen Blocks zu erhalten; die zugehörigen Daten befinden sich bei `blk->data`.

Ein vollständigeres Beispiel kann in der Datei `/home/pi/icon9_51/ipl/cfuncs/external.c` gefunden werden.

Listing home/pi/icon9_51/ ipl/cfuncs/external.c

```
1   /*
2   ############################################################################
3   #
4   #   File:     external.c
5   #
6   #   Subject:  Functions to demonstrate Icon external values
7   #
8   #   Author:   Gregg M. Townsend
9   #
10  #   Date:     May 29, 2013
11  #
12  ############################################################################
13  #
14  #    This file is in the public domain.
15  #
16  ############################################################################
17  #
18  #  These functions demonstrate the use of external values.
19  #
20  #  extxmin() creates a minimal external type
21  #  extxstr(s)creates an external hold a string and trivial checksum
22  #  extxreal(r)    creates a fully customized external type holding a real value
23  #
24  ############################################################################
25  #
26  #  Requires:  Dynamic loading
27  #
28  ############################################################################
29  */
30
31  #include <string.h>
32  #include "icall.h"
33
34  /*
35   * minimal external type with no parameters
36   */
37  int extxmin(int argc, descriptor argv[])   /*: create minimal external value */
38     {
39     RetExternal(alcexternal(0, 0, 0));
40     }
41
42
43  /*
44   * custom external holding a null-terminated string and a trivial checksum
45   */
46
47  /* data area */
48  typedef struct sdata {
49     unsigned short cksum;
50     char string[];
51     } sdata;
52
```

```
53  /* type name returns "xstr" */
54  static int sname(int argc, descriptor argv[]) {
55     RetConstStringN("xstr", 4);
56     }
57
58  /* image returns "xstr_N(cksum:string)" with no special string escapes */
59  static int simage(int argc, descriptor argv[]) {
60     externalblock *xb = ExternalBlock(argv[1]);
61     sdata *d = (sdata*) xb->data;
62     char buffer[1000]; /* not robust against huge strings */
63     RetStringN(buffer,
64        sprintf(buffer, "xstr_%ld(%05d:%s)", xb->id, d->cksum, d->string));
65     }
66
67  /* list of custom functions for constructor */
68  static funclist sfuncs = {
69     NULL, /* cmp */
70     NULL, /* copy */
71     sname,     /* name */
72     simage,    /* image */
73     };
74
75  /* finally, the exported constructor function, extxstr(s) */
76  int extxstr(int argc, descriptor argv[])   /*: create string-valued external */
77     {
78     ArgString(1);
79     int slen = StringLen(argv[1]);
80     externalblock *xb = alcexternal(
81        sizeof(externalblock) + sizeof(sdata) + slen + 1, &sfuncs, 0);
82     sdata *d = (sdata*) xb->data;
83     memcpy(d->string, StringAddr(argv[1]), slen);
84     d->string[slen] = '\0';
85     int cksum = 0;
86     char *p;
87     for (p = d->string; *p; p++)
88        cksum = 37 * cksum + (unsigned char) *p;
89     d->cksum = cksum;
90     RetExternal(xb);
91     }
92
93
94
95  /*
96   * custom real-valued external with lots of trimmings
97   */
98
99  /* data area */
100 typedef struct rdata {
101    float value;
102    } rdata;
103
104 /* comparison function for sorting */
105 static int rcmp(int argc, descriptor argv[]) {
106    externalblock *xb1 = ExternalBlock(argv[1]);
107    externalblock *xb2 = ExternalBlock(argv[2]);
108    rdata *data1 = (rdata*) xb1->data;
109    rdata *data2 = (rdata*) xb2->data;
110    if (data1->value < data2->value) RetInteger(-1);
111    if (data1->value > data2->value) RetInteger(+1);
112    if (xb1->id < xb2->id) RetInteger(-1);
113    if (xb1->id > xb2->id) RetInteger(+1);
114    RetInteger(0);
115    }
116
117 /* copy function duplicates block, getting new serial number */
118 static int rcopy(int argc, descriptor argv[]) {
119    externalblock *xb = ExternalBlock(argv[1]);
120    RetExternal(alcexternal(
121       sizeof(externalblock) + sizeof(rdata), xb->funcs, xb->data));
122    }
123
124 /* type name returns "xreal" */
125 static int rname(int argc, descriptor argv[]) {
126    RetConstStringN("xreal", 5);
127    }
128
129 /* image returns "xreal_N(V)" */
130 static int rimage(int argc, descriptor argv[]) {
131    externalblock *xb = ExternalBlock(argv[1]);
132    rdata *d = (rdata*) xb->data;
133    char buffer[100];
134    RetStringN(buffer,
135       sprintf(buffer, "xreal_%ld(%.1f)", xb->id, d->value));
```

```
136     }
137
138 /* list of custom functions for constructor */
139 static funclist rfuncs = {
140     rcmp, /* cmp */
141     rcopy,     /* copy */
142     rname,     /* name */
143     rimage,    /* image */
144     };
145
146 /* finally, the exported constructor function, extxreal(r) */
147 int extxreal(int argc, descriptor argv[])    /*: create real-valued external */
148     {
149     ArgReal(1);
150     float v = RealVal(argv[1]);
151     RetExternal(alcexternal(
152        sizeof(externalblock) + sizeof(rdata), &rfuncs, &v));
153     }
```

30.5.4 Die Funktionsliste, die `alcexternal` übergeben wird

Die zu einem Externwert zugehörige Funktionsliste ist eine Struktur, die Zeiger auf C-Funktionen enthält. Es erinnert an eine Tabelle oder Klassenpointer für dynamische Methoden-Aufrufe in einer Implementierung einer objektorientierten Programmiersprache. In der Tat ist ein Icon-Externwert viel eher wie ein traditionelles Objekt mit dessen eigenen Daten und Methoden. Typischerweise würde solch eine Funktionsliste statisch sein und unter vielen Icon-Externwerten der gleichen Art ("class" oder "type") verteilt sein.

Jeder Externwert besitzt eine Funktionsliste; eine voreingestellte Liste wird geliefert, falls NULL an `alcexternal` übergeben wird. Ein Null-Eintrag innerhalb einer Funktionsliste erzeugt das voreingestellte Verhalten für die zugehörige Aktion.

Funktionen in der Liste verwenden die gleichen Schnittstellen wie ladbare C-Funktionen. Eingehende Argumente werden beginnend mit `argv[1]` übergeben. Ein Ergebnis wird erzeugt, indem es in `argv[0]` gespeichert und indem 0 als Ergebnis der Funktion zurückgegeben wird. Sobald `extlcmp` aufgerufen wird, hat `argc` einen Wert von 2; für die anderen Funktionen ist `argc` gleich 1.

Die möglichen Funktionen sind:

- `int extlcmp(int argc, descriptor *argv)`

  gibt eine Icon-Ganzzahl zurück, um beim Sortieren zweier Externwerte verwendet zu werden, die beide diese Funktionsliste aufweisen, und daher vom gleichen externen Subttyp gehalten werden. Das Funktionsergebnis sollte negativ sein, falls der erste Externwert für kleiner als der zweite erachtet wird, 0 falls sie gleich erachtet werden und positiv, falls der erste größer als der zweite erachtet wird. Dies überschreibt das voreingestellte Verhalten der Funktion `sort()`, die Serialnummern vergleicht.

- `int extlcopy(int argc, descriptor *argv)`

  gibt einen Externert zurück, der als Kopie von dessen Argument festgelegt wird. Dies überschreibt das voreingestellte Verhalten der Funktion `copy()`, die darin besteht, eine andere Referenz auf den Externwert ohne Kopieren zurückzugeben.

- `int extlname(int argc, descriptor *argv)`

  gibt eine Icon-Zeichenkette zurück, die den Typ des Externwertes benennt. Dies überschreibt das voreingestellte Verhalten der Funktion `type()` und beinflusst somit ebenso die Reihenfolge relativ zu anderen Externwerten bem Sortieren.

- `int extlimage(int arrgc, descriptor *argv)`

  gibt eine Icon-Zeichenkette zurück, um als Image für den Externwert zu dienen. Dies überschreibt das voreingestellte Verhalten der Funktion `image()`.

30.5.5 Details zur Implementierung

Ein Deskriptor für einen Externwert weist ein `vword` auf, das das Bitmuster `D_External` enthält, das Folgendes beinhaltet:

- den Wert `T_External`, der den externen Datenyp anzeigt, zusammen mit
- dem Bit `F_Nqual`, das anzeigt, dass der Wert keine Zeichenkette ist, und

- das Bit `F_Ptr` zeigt an, dass der Zeiger `dword` zum Garbage-Collektor der Pflege bedarf.

Das `dword` enthält einen Zeiger auf einen externen Block. Dieser Block wird als C-Struktur mit einem Zeiger auf einen Funktionsblock imlementiert. Die u.a. Strukturen, zusammen mit Typen und Deskriptor, werden in `ipl/cfuncs/icalll.h` definiert.

```
typedef struct funclist
{   int (*extlcmp) (int argc, descripor *argv);#
    in (*extlcopy) (intargc, descriptor *argv);
    int (*extlname) (int argc, descruptot *argv);
    int (*extlimage) (int argc,  descriptr *argv);
} funclist;
typedef struct externalblock
{   word title;
    word size,
    word id;
    funclist *funcs;
    word data [];
} externalblock;
```

Ein Aufruf von `alcexternal` initialisiert alle Felder eines externen Blcoks. Daten werden ohne irgendwelche Interpretationen kopiert und der Funktionslisten-Zeiger wird gespeichert. Dem Header wird eine angemessene Konstante für den Garbage-Kollektor zugewiesen und die Grösse wird gesetzt. `alcexternal` erhält ein statisches Ergebnis der Anzahl allozierter externer Blöcke aufrecht und dies wird hochgezählt und `id` zugewiesen.

Die folgenden Fehlercodes werden bei Anwendung von Externwerten zugewiesen:

errno	errtext	Bedeutung
131	external expected	Kein Externwert
132	incorrect external type	External von falscher Variante
133	invalid external value	Richtige Variante im falschen Zusammenhang
134	malformed external value	Daten sind falsch, noch nicht ungeeignet

Das war's für heute! Genießt Euer Leben bis zum nächsten Mal!

Was kommt demnächst?

A: Software existiert
B: Ideen bzgl. Inhalt vorhanden
C: Grundlagenforschung

Block Nutzung von in anderen Programmiersprachen geschriebenen Bibliotheken
31 Von Icon auf C++-Bibliotheken zugreifen **C**
32 Von Icon auf Cobol- / D- / Prolog-Bibliotheken zugreifen **C**

Block Sprachsysnthese & Sprachsteuerung
33 TTS & STT: Sprachsysnthese und Sprachsteuerung auf dem Raspberry Pi **A**

Block Tools für die Programmentwicklung
34 Gestaltung graphischer Benutzeroberflächen mit dem Event-Multiplexor (EVMUX) inkl. Programmierung der Ereignisbehandlung **BC**
35 Symbolizer BiLevel **A**
36 Symbolizer Farbpaletten **A**

Block Pseudo-Code-Compiler
37 Pseudo-Code-Compiler **BC**

Block Ressourcenschionende Programmierung
38 CPU-Usage – FreeMem **A**

Block Schulprojekte
39 DiaShower **A**
40 Widerstandsmesser **A**

Block RPi & Arduino
41 Tisch-Sonar **A**
42 SPI am Beispiel AD-Wandler **A**
43 Arduino-Controll-Center **A**
44 Automatischer Filmscanner **C**

Block Pipe-Programmierung
45 Pipe-Programmierung **B**

Block Kernel-Compilierung
46 Der Kernel-Compilator **A**
47 Individuelle Kernel-Programmierung I **B**
48 Individuelle Kernel-Programmierung II **C**
49 Individuelle Kernel-Programmierung III **C**
50 Individuelle Kernel-Programmierung III **C**