Installation Guide for Version 8 of Icon on UNIX Systems*

Ralph E. Griswold

TR 90-2f

January 1, 1990; last modified April 3, 1990

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant CCR-8901573.

Installation Guide for Version 8 of Icon on UNIX Systems

1. Introduction

Version 8 is the current version of Icon and replaces Version 7.5. Version 8 contains several new features and improvements to the implementation [1]. Most changes to the language are upward compatible with earlier versions of Icon. Icon programs may need to be recompiled, however, when Version 8 is installed.

This report provides the information necessary to install Version 8 of Icon on computers running UNIX. For other operating systems, see [2]. The installation process for Version 8 is very similar to that for Version 7.5.

The implementation of Icon is designed so that it can be installed, largely automatically, on a variety of computers running different versions of UNIX. This is accomplished by providing configuration information that tailors the installation to specific computers and versions of UNIX. Appendix A contains a list of supported configurations.

These systems are referred to as ''supported'' in this report. Some of these originated under earlier versions of Icon, and not all of these have been tested yet under Version 8. The systems marked with an asterisk have been tested under Version 7.5 or 8 and are referred to as ''tested'' in this report. Not all of these have been tested under Version 8, so minor difficulties are possible.

If your system is a tested one, the installation of Version 8 of Icon should be as simple as issuing a few make commands. If your system is supported but untested, you may be able to install it without modification, but if problems show up, you may have to make minor modifications in configuration files. If your system is not in this list, it may have been added since this report was written. See Section 2.1 for information on how to get a current list of configurations and their statuses. In some cases, there may be partial configuration information. If the configuration information for your system is partial or lacking altogether, you still may be able to install Version 8 of Icon by providing the information yourself, using other configurations are a guide. See Section 3.

2. The Installation Process

There are only a few steps needed to install Icon proper. In addition to Icon itself, there are a number of optional components that can be installed: a program library [3], a personalized interpreter system [4], a variant translator system [5], and a memory-monitoring system [6]. You may want to review the technical reports describing these optional components before beginning the installation. In any event, the installation of optional components can be done separately after Icon itself is installed.

There are Makefile entries for most steps. Those steps are marked by asterisks. Steps that are optional are enclosed in brackets.

Icon Proper

1. Decide where to unload Icon.
2. Unload the Icon hierarchy at the selected place.

[3.*] Check the status of the configuration for your system.

4. Set up paths.

5.* Configure the source code for your system.

6.* Check the size of a header file; if it is not large enough, adjust a configuration parameter and start again at Step 5.

7.* Compile Icon.

8.∗ Install the compiled files.

9.∗ Run some simple tests to be sure Icon is working.

[10.∗] Run a test suite.

The Icon Program Library

[1.∗] Compile the Icon program library

[2.∗] Test the Icon program library

[3.] Copy the Icon program library to a public place.

The Icon Personalized Interpreter

[1.∗] Build the Icon personalized interpreter system.

[2.∗] Test the Icon personalized interpreter system.

[3.] Copy the personalized interpreter system to a public place.

The Icon Variant Translator System

[1.∗] Test the Icon variant translator system.

[2.] Copy the variant translator system to a public place.

The Icon Memory-Monitoring System

[1.∗] Build the monitoring programs.

[2.∗] Test the monitoring programs.

Benchmarking

[1.∗] Timing test programs.

Finishing Up

[1.] Install documentation for the various components of Icon.

[2.∗] Remove files that are no longer needed.

2.1 Installing Icon Proper

Step 1: Deciding Where to Unload Icon

The default location for all files, including executable binaries, is in the directory /usr/icon/v8. You can unload the distribution in another area, or move the files later, but the installation is easiest if the default location is used. If you decide not to put Icon at the default location, read the discussion at Step 4 before going on.

In the balance of this report, relative paths and the location of files are given with respect to the location into which the Icon hierarchy is unloaded. For example, a reference to make is with respect to the Makefile at the top level of this hierarchy (/usr/icon/v8/Makefile for the default location). Similarly, config/unix corresponds to /usr/icon/v8/config/unix for the default location.

Step 2: Unloading the Files

The distribution consists of a hierarchy, which is rooted in ‘‘.’’. Icon is distributed in a variety of formats. It requires about 4.5MB of disk space when unloaded.

The usual distribution medium is magnetic tape, although it is also available on cartridges and diskettes.

Tapes: The Icon system is provided on tape in *tar* or *cpio* format, recorded at 1600 or 6250 bpi. The format and recording density are marked on the label on the tape.

To unload the tape, cd to the directory that is to hold the Icon hierarchy (the default location is /usr/icon/v8 as mentioned above) and mount the tape. The precise *tar* or *cpio* command to unload the distribution tape depends on

your local environment. On a VAX running 4.*n*bsd, use the following command for a 1600 bpi *tar* distribution tape:

```
tar x
```

Similarly, on a VAX running System V with a 6250 bpi *cpio* tape, use:

```
cpio –icdB </dev/rmt/0h
```

The c (compatibility) and B (blocked) options are essential.

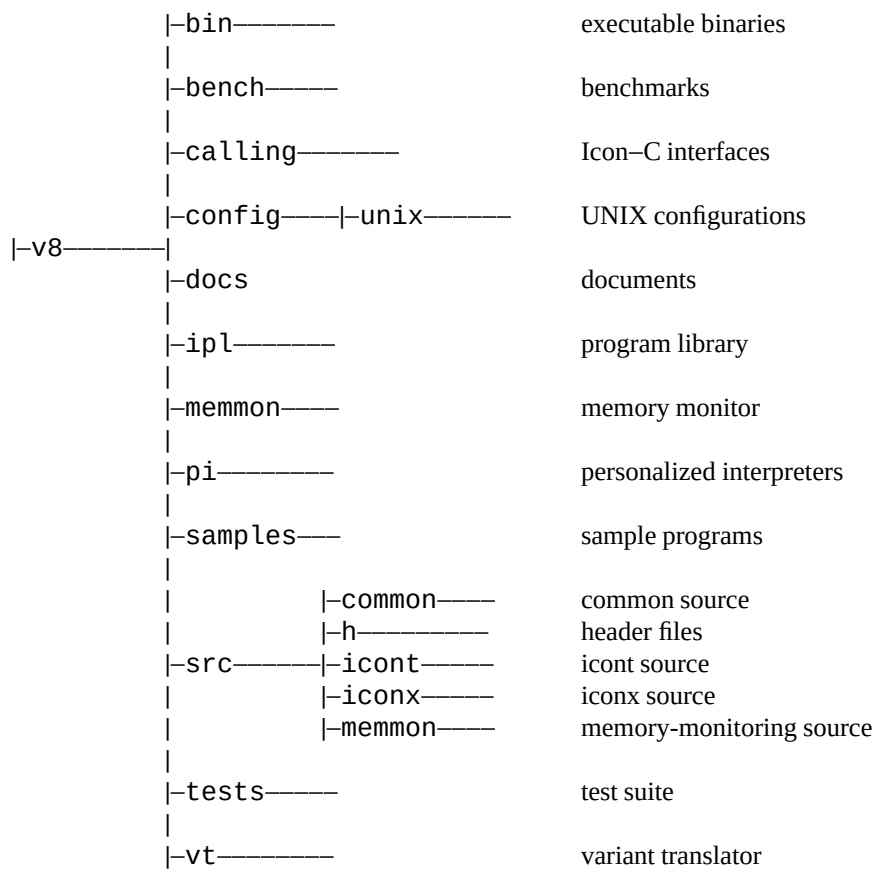
Cartridges: Data cartridges are functionally equivalent to magnetic tapes, but they are not blocked. For example, on a Sun Workstation with a *cpio* cartridge, cd to the directory that is to hold the Icon hierarchy and use

```
cpio –icd </dev/rst0
```

Diskettes: Diskettes contain *cpio* files on diskettes in MS-DOS format. Copy the ∗.cpi files on the diskettes to the directory that is to hold the Icon hierarchy and use a script such as the following:

```
for i in ∗.cpi
do
   cpio –icd <$i.cpi
done
```

After the distribution files are unloaded, the resulting hierarchy should look like this:



In some cases there are subdirectories not shown above.

Step 3: Checking the Status of the Configuration for Your System

You may wish to check the status of the configuration for your system. This can be done by

```
make Status name=name
```

where *name* is one of those given in the table in Appendix A. For example,

```
make Status name=vax_bsd
```

lists the status of the configuration for a VAX running BSD UNIX.

In many cases, the status information was provided by the person who first installed Icon on the system in question. The information may be old and possibly inaccurate; use it as a guideline only.

There are some supported systems for which not all features of Icon are implemented. If the status information shows this for your system, proceed with the installation, but you may wish to implement the missing features later. For this, see Section 3 after completing the basic installation.

Step 4: Setting Up Paths

If you unloaded Version 8 of Icon at the default location and plan to leave executable binaries at their default locations, skip this step. Otherwise, you need to change path specifications in your configuration directory.

There are three paths used in the installation of Icon that are given by defined constants:

RootPath	The root of the Icon hierarchy; used by scripts that build personalized interpreters and variant translators to locate Icon source code.
IcontPath	The location of the Icon command-line processor, icont.
IconxPath	The location of the Icon run-time executor, iconx.

The default paths for most supported configurations are:

```
#define RootPath        "/usr/icon/v8"
#define IcontPath       "/usr/icon/v8/bin/icont"
#define IconxPath       "/usr/icon/v8/bin/iconx"
```

They are slightly different for a few configurations that have non-standard naming conventions.

The location of iconx is particularly important, since compiled Icon programs do not stand alone but must find iconx to run. To make this easy, the path specified in IconxPath is hardwired into compiled Icon programs. This means, however, that the value of IconxPath, which must be set before Icon is compiled, is inherited by all subsequently compiled Icon programs. If iconx is moved to another place, the hardwired path is invalidated.

There are ways around this, however. If the environment variable ICONX is set, its value overrides the hardwired path. Furthermore, if ICONX is not set and iconx is not found on the hardwired path, the user's PATH environment variable is searched for iconx. In fact, it is possible to configure Icon to disable the use of hardwired paths. See Section 3 if you want to do this. Nonetheless, it is advisable to chose an appropriate value for IconxPath.

If you decide to change the default paths, you need to edit the file paths.h in the configuration directory for your system. The directory config/unix contains a subdirectory for each supported system. For example, config/unix/sun3 contains the configuration information for the Sun-3 Workstation. To get to the configuration information for your system,

```
cd config/unix/name
```

where *name* is the name of your system. For example, if you want the Icon hierarchy in /usr/irving/v8 and have the binaries in /usr/local/icon, edit paths.h to be

```
#define RootPath        "/usr/irving/v8"
#define IcontPath       "/usr/local/icon/icont"
#define IconxPath       "/usr/local/icon/iconx"
```

Caution: If you are using a previous version of Icon and put iconx where the previous version was, all user programs will have to be recompiled, since iconx for Version 8 is incompatible with earlier versions of iconx.

Step 5: Configuring Icon for Your System

Configuring Icon creates a number of files for general use. Before starting the configuration, be sure your umask is set so that these files will be accessible.

To configure Icon for your system, do

```
make Configure name=name
```

where *name* is the name of your system as described above. For example,

```
make Configure name=vax_bsd
```

configures Version 8 of Icon for a VAX running BSD UNIX.

Step 6: Checking the Size of a Header File

Translating and linking an Icon program with icont produces an *icode* file, which can then be run. In order to make icode files executable, a bootstrap header, iconx.hdr, is provided. The size of iconx.hdr varies from system to system and is determined by the defined constant MaxHdr, which is given in a configuration file. If value of MaxHdr is not large enough, the compilation of icont terminates with an error message. To be sure that MaxHdr is large enough for your system, do

```
make Header
```

This compiles the header file and lists its size, followed by the value of MaxHdr. For example, on a VAX BSD system, typical output from this make is

```
cc –O –c ixhdr.c
cc –O –N ixhdr.o –o iconx.hdr
strip iconx.hdr
–rwxrwxr–x  1 icon          1912 Jan 10 18:32 iconx.hdr
#define MaxHdr  1950
```

The last two lines are what are important. In this example, MaxHdr is 1500 and the size of the header file is 1492 — that is, MaxHdr is large enough.

If you find MaxHdr is not large enough for your system, edit config/unix/*name*/define.h and change the value of MaxHdr there to an appropriate value (where *name* is the name of your system as given above). It is advisable to leave a little spare room; some systems even require the value of MaxHdr to be rounded up. Don't worry about that at this point, but if icode files fail to execute, come back to this step and increase MaxHdr.

If you change MaxHdr, you must go back and start over with Step 5.

Step 7: Compiling Icon

Next, compile Icon by

```
make Icon
```

This takes a while. There may be warning messages on some systems, but there should be no fatal errors.

Step 8: Installing Icon

To install Icon, do

```
make Install
```

Among other things, this copies icont and iconx to the locations specified in IcontPath and IconxPath, respectively.

Step 9: Doing Some Simple Tests

For supported systems that compile and install without apparent difficulty, a few simple tests usually are sufficient to confirm that Icon is running properly. The following does the job:

```
make Samples
```

This test compares local program output with the expected output. There should be no differences. If there are none, you presumably have a running Version 8 Icon.

Note: If Icon fails to run at all, this may be because there is not enough ''static'' space for it to start up. If this happens, check define.h in your configuration directory. If it contains a definition for MaxStatSize, try doubling it, and start over with Step 5. If define.h does not contain a definition for MaxStatSize, add one such as

```
#define MaxStatSize 20480
```

and go back to Step 5. If this solves the problem, you may wish to reduce MaxStatSize to a smaller value that works in order to conserve memory. If this does not solve the problem, try increasing MaxStatSize even more (it is unlikely that much larger values will help).

Step 10: Extensive Testing

If you want to runs more extensive tests, do

```
make Test–all
```

This takes quite a while and does a lot of work. Some differences are to be expected, since tests include date, time, and local host information. There also may be insignificant differences in the format of floating-point numbers and the order of random numbers. In addition to Test–all there are some individual tests of optional features. See the main Makefile for more information about the tests.

2.2 Icon Program Library

The Icon program library contains a variety of programs and procedures. This library not only is useful in its own right, but it provides numerous examples of programming techniques which may be helpful to novice Icon programmers. While this library is not necessary for running Icon programs, most sites install it.

In addition to the library proper, the directory ipl/idol contains an object-oriented version of Icon written in Icon. Go to that directory for more information.

Step 1: Building the Icon Program Library

To build the Icon program library, do

```
make Ipl
```

This puts compiled programs in ipl/icode and translated procedures in ipl/ucode.

Step 2: Testing the Icon Program Library

To test the library, do

```
make Test–ipl
```

No differences should show.

Step 3: Installing the Icon Program Library

You can copy the executable programs in ipl/icode and the translated procedures in ipl/ucode to public locations to make them more accessible, although they can be used from any location that is readable by the user.

2.3 Personalized Interpreters

The personalized interpreter system allows an individual to build a private copy of Icon's run-time system, which then can be modified.

Personalized interpreters are somewhat specialized and the typical Icon programmer has no need for them. However, if your site has a need for tailored versions of Icon, this system may be useful.

Step 1: Building the Personalized Interpreter System

To build the personalized interpreter system, do

```
make PI
```

Step 2: Testing the Personalized Interpreter System

For testing, do

```
make Test–pi
```

There may be some warning messages during compilation, but there should be no fatal errors.

Step 3: Installing the Personalized Interpreter System

Personalized interpreter directories are constructed by the shell script icon_pi. You therefore may wish to place it in a public location:

```
cp icon_pi location
```

2.4 Variant Translators

The variant translator system facilitates the construction of preprocessors for variants of the Icon programming language. This facility is even more specialized than the personalized interpreter system, but some forthcoming tools related to measuring the performance and behavior of Icon programs may use the variant translator system.

The variant translator system requires a version of *yacc(1)* with large regions. You may have to tailor your version of *yacc(1)* for this. See [5]. On systems with a limited amount of memory, this may not work at all. If there is a problem, it will show up during testing.

There is no separate step for building the variant translator system. However, Icon must be installed before testing the variant translator system.

Step 1: Testing the Variant Translator System

For testing, do

```
make Test–vt
```

There may be warning messages during compilation, but there should be no fatal errors.

Step 2: Installing the Variant Translator System

To put icon_vt, the shell script that builds variant translator directories into a public place, do

```
cp icon_vt location
```

2.5 Memory Monitoring

Step 1: Building the Monitoring Programs

To build the memory-monitoring programs, do

```
make MemMon
```

Step 2: Testing the Memory-Monitoring System

For testing, do

```
make Test–memmon
```

There will be differences in date lines and in some monitoring data because of different memory locations, but there should not be extensive differences.

2.6 Benchmarking

Test programs are provided for benchmarking Version 8 of Icon. To perform the benchmarks, go to the subdirectory bench and do

```
make benchmark
```

See also the other material in that subdirectory. It contains a form that you can use to record your benchmarks with the Icon Project (see Section 4).

2.7 Finishing Up

Step 1: Installing Documentation

After Icon and any optional components have be installed, you may wish to install the appropriate manual pages in the standard location on your system. The manual pages are in the docs directory:

icont.1	manual page for Icon proper
icon_pi.1	manual page for the Icon personalized interpreter system
icon_vt.1	manual page for the Icon variant translator system
memmon.1	manual page for using the Icon memory-monitoring system
memmon.5	manual page for memory-monitoring data

The docs directory also contains machine-readable copies of technical reports related to Version 8 of Icon.

Step 2: Cleaning Up

You can remove object files and test results by

```
make Clean
```

You also can remove source files, but think twice about this, since source files may be useful to persons using personalized interpreters and variant translators. In addition, you can remove files related to optional components of the Icon system that you do not need. If you are tight on space, you may wish to remove documents as well.

3. Configuring Version 8 for a New UNIX System

Version 8 of Icon assumes that C *int*s are 16, 32, or 64 bits long. If your system violates this assumption, don't try to go on — but check back with us, since we are may be able to provide some advice on how to proceed.

There are 13 steps in installing Icon for a new system:

1.∗ Build a configuration directory.

2. Edit a configuration file to provide appropriate definitions for your system.

3. Edit Makefile headers.

4.∗ Perform the installation as described in Section 2.

5.∗ Perform extensive testing.

[6.] Possibly provide assembly-language code for integer overflow checking.

[7.] Implement and test co-expressions.

[8.] Install the personalized interpreter system.

[9.] Test the variant translator system.

[10.] Install and test the memory-monitoring system.

[11.] Run benchmarks.

12. Provide status information in your configuration directory.

13. Send the contents of your configuration directory to the Icon Project.

Step 1: Building a New Configuration Directory

First select a name for your system. For compatibility with tools used at the Icon Project, the name should be in lowercase and consist of a mnemonic for the computer, which may be followed by an underscore and a mnemonic for the operating system, if there is more than one operating system for the computer. Examples are vax_bsd and vax_sysv.

To build and initialize a new configuration directory,

```
make System name=name
```

where *name* is the name of your system.

As a result, the subdirectory *name* will contain the following files:

define.h	main configuration file
paths.h	paths
icont.hdr	flags for the icont, common, and memmon Makefiles
iconx.hdr	flags and other definitions for the iconx Makefile
pi.hdr	flags for the personalized-interpreter Makefile
vt.hdr	flags for the variant-translator Makefile
rswitch.c	co-expression context switch
Ranlib	library randomizer for personalized interpreters

Alternatively, if there is a supported configuration for a system than is similar to yours, you may wish to copy the files from that configuration.

To work on your configuration files,

```
cd config/unix/name
```

Step 2: Editing the Main Configuration File, define.h

There are many defined constants in the source code for Icon that vary from system to system. Default values are provided for most of these so that the usual cases are handled automatically. The file define.h contains C preprocessor definitions for parameters that differ from the defaults or that must be provided on an individual basis. The initial contents of this file as produced in Step 1 above are for a ''vanilla'' system with the commonest values for parameters. If your system closely approximates a ''vanilla'' system, you will have few changes to make to define.h. Over the range of possible systems, there are many possibilities as described below.

The definitions are grouped into categories so that any necessary changes to define.h can be approached in a logical way.

ANSI Standard C: Icon preprocessor directives use string concatenation and substitution of arguments within quotation marks. By default, the ''old-fashioned'', *ad hoc* method of accomplishing this in UNIX preprocessors is used. A different method is specified in the ANSI C draft standard [7]. The ANSI C draft standard also uses void * in place of the older char * for pointers to ''generic storage''.

If your C compiler supports the ANSI C draft standard, add

```
#define Standard
```

to define.h.

Alternatively, you can define StandardPP or StandardC if your preprocessor is standard but your compiler isn't, or vice versa.

C sizing and alignment: There are four constants that relate to the size of C data and alignment:

IntBits	(default: 32)
WordBits	(default: 32)
Double	(default: undefined)

IntBits is the number of bits in a C *int*. It may be 16, 32, or 64. WordBits is the number of bits in a C *long* (Icon's ''word''). It may be 32 or 64. If your C library expects *double*s to be aligned at double-word boundaries, add

```
#define Double
```

to define.h.

Most computers have downward-growing C stacks, for which stack addresses decrease as values are pushed. If you have an upward-growing stack, for which stack addresses increase as values are pushed, add

```
#define UpStack
```

to define.h.

The alignment, in words, of stacks used by co-expressions is controlled by

StackAlign	(default: 2)

If your system needs a different alignment, provide an appropriate definition in define.h.

Floating-point arithmetic: There are three optional definitions related to floating-point arithmetic:

Big	(default: 9007199254740092.)
LogHuge	(default: 309)
Precision	(default: 10)

The values of Big, LogHuge, and Precision give, respectively, the largest floating-point number that does not lose precision, the maximum base-10 exponent + 1 of a floating-point number, and the number of digits provided in the string representation of a floating-point number. If the default values given above do not suit the floating-point arithmetic on your system, add appropriate definitions to define.h.

Include file location: The location of the include file time.h varies from system to system. Its default location is <time.h>. If it resides at a different location on your system (such as <sys/time.h>), add an appropriate definition of SysTime to define.h, as in

```
#define SysTime <sys/time.h>
```

If the location is incorrect, a fatal error will occur during the compilation of src/common/time.c.

The use of this definition also depends on your C preprocessor making macro substitutions in #include directives. Most preprocessors do, but if yours does not, edit src/common/time.c and replace SysTime there by the appropriate value. If you have to do this, make a note to come back later and place the definition under the control of conditional compilation as described in Step 4.

Run-time routines: The support for some run-time routines varies from system to system. The related constants are:

IconGcvt	(default: undefined)
IconQsort	(default: undefined)
index	(default: undefined)
rindex	(default: undefined)

If IconGcvt and IconQsort are defined, versions of *gcvt(3)* and *qsort(3)* in the Icon system are used in place of the routines normally provided in the C run-time system. These constants only need to be defined if the versions of these routines in your run-time system are defective or missing.

Different versions of UNIX use different names for the routines for locating substrings within strings. The source code for Icon uses index and rindex. The other possibilities are strchr and strrchr. If your system uses the

latter names, add

```
#define index strchr
#define rindex strrchr
```

to define.h.

Host identification: The identification of the host computer as given by the Icon keyword &host needs to be specified in define.h. The definition

```
#define HostStr "unknown host"
```

is provided in define.h initially. This definition should be changed to an appropriate value for your system.

Alternatively, some systems provide direct mechanisms for specifying the host in a standard way. In this case, remove the definition of HostStr and provide an alternative as follows:

On some versions of UNIX, notably Version 7 and 4.1bsd, the file /usr/include/whoami.h contains the host name. If your system has this file and you want to use this name, add

```
#define WhoHost
```

to define.h.

Some versions of UNIX, notably 4.2bsd and 4.3bsd, provide the host name via the *gethostname(2)* system call. If your system supports this system call and you want to use this name, add

```
#define GetHost
```

to define.h.

Some versions of UNIX, such as System V, provide the host name via the *uname(2)* system call. If your system supports this call and you want to use this name, add

```
#define UtsName
```

to define.h.

Note: Only one of these methods of specifying the host name can be used.

Hardwired paths: As mentioned in Section 2.1, a hardwired path normally is used for finding iconx. This feature can be removed by adding

```
#define NoFixedPaths
```

to define.h.

Storage management: Icon includes its own versions of *malloc(3)*, *calloc(3)*, *realloc(3)*, and *free(3)* so that it can manage its storage region without interference from allocation by the operating system. Normally, Icon's versions of these routines are loaded instead of the system library routines.

Leave things are they are in the initial configuration, but if your system insists on loading its own library routines, multiple definitions will occur as a result of the *ld* in src/iconx. If multiple definitions occur, go back and add

```
#define IconAlloc
```

to define.h. This definition causes Icon's routines to be named differently to avoid collision with the system routine names.

One possible effect of this definition is to interfere with Icon's expansion of its memory region in case the initial values for allocated storage are not large enough to accommodate a program that produces a lot of data. This problem appears in the form of run-time errors 305-307. Users can get around this problem on a case-by-case basis by increasing the initial values for allocated storage by setting environment variables [8].

Icon's dynamic storage allocation system uses three contiguous memory regions that it expands if necessary. This method relies on the use of *brk(2)* and *sbrk(2)* and the system treatment of user memory space as one logically contiguous region. This may not work on some systems that treat memory as segmented or do not support *brk()* and *sbrk()*. On such systems, it may be necessary to add

```
#define FixedRegions
```

to define.h. The effect of this definition is to assign fixed-sized regions for Icon's use. These regions may not be shared or expanded and all of available memory may not be used. This option should be used only if necessary.

The bootstrap header: As described In Section 2.1, Step 6, a bootstrap header is used to make icode files executable. The space reserved for the header is determined by

```
#define MaxHdr          (default: 4096)
```

On some systems, many routines may be included in the header even if they are not needed. Start by assuming this is not a problem, but if MaxHeader has to be made impractically large, you can eliminate the header by adding

```
#define NoHeader
```

to define.h. *Note:* If NoHeader is defined, the value of MaxHdr is irrelevant.

The effect of this definition is to render Icon programs non-executable. Instead, they must be run by using the –x option after the program name when icont is used, as in

```
icont prog.icn –x
```

Such a program also can be run as an argument of iconx, as in

```
iconx prog
```

where prog is the result of translating and linking prog.icn as in the previous example.

Storage regions: The sizes of Icon's run-time storage regions for allocated blocks and strings normally are the same for all implementations. However, different values can be set:

MaxStatSize	(default: 20480 if co-expressions are enabled, else 1024)
MaxAbrSize	(default: 65000)
MaxStrSize	(default: 65000)

Since users can override the set values with environment variables, it is unwise to change them from their defaults except in unusual cases.

The sizes for Icon's main interpreter stack and co-expression stacks also can be set:

MStackSize	(default: 10000)
StackSize	(default: 2000)

As for the block and string storage regions, it is unwise to change the default values except in unusual cases.

Finally, with fixed-regions storage management, a list used for pointers to strings during garbage collection, can be sized:

QualLstSize	(default: 5000)

Like the sizes above, this one normally is best left unchanged.

Dynamic hashing:

Four parameters configure the implementation of tables and sets:

HSlots	Initial number of hash buckets; it must be a power of 2
HSegs	Maximum number of hash bucket segments
MaxHLoad	Maximum allowable loading factor
MinHLoad	Minimum loading factor for new structures

The default values (listed below) are appropriate for most systems. If you want to change the values, read the discussion that follows.

Every set or table starts with HSlots hash buckets, using one bucket segment. When the average hash bucket exceeds MaxHLoad entries, the number of buckets is doubled and one more segment is consumed. This repeats until HSegs segments are in use; after that, structure still grows but no more hash buckets are added.

MinHLoad is used only when copying a set or table or when creating a new set through the intersection, union, or difference of two other sets. In these cases a new set may be more lightly loaded than otherwise, but never less than MinHLoad if it exceeds a single bucket segment.

For all machines, the default load factors are 5 for MaxHLoad and 1 for MinHLoad. Because splitting or combining buckets halves or doubles the load factor, MinHLoad should be no more than half MaxHLoad. The average number of elements in a hash bucket over the life of a structure is about 2/3×MaxHLoad, assuming the structure is not so huge as to be limited by HSegs. Increasing MaxHLoad delays the creation of new hash buckets, reducing memory demands at the expense of increased search times. It has no effect on the memory requirements of minimally-sized structures.

HSlots and HSegs interact to determine the minimum size of a structure and its maximum efficient capacity. The size of an empty set or table is directly related to HSegs+HSlots; smaller values of these parameters reduce the memory needs of programs using many small structures. Doubling HSlots delays the onset of the first structure reorganization until twice as many elements have been inserted. It also doubles the capacity of a structure, as does increasing HSegs by 1.

The maximum number of hash buckets is HSlots×(2^(HSegs−1)). A structure can be considered ''full'' when it contains MaxHLoad times that many entries; beyond that, lookup times gradually increase as more elements are added. Until a structure becomes full, the values of HSlots and HSegs do not affect lookup times.

For machines with 16-bit *int*s, the defaults are 4 for HSlots and 6 for HSegs. Sets and tables grow from 4 hash buckets to a maximum of 128, and become full at 640 elements. For other machines, the defaults are 8 for HSlots and 10 for HSegs. Sets and tables grow from 8 hash buckets to a maximum of 4096, and become full at 20480 elements.

Memory monitoring: The number of bytes for reporting block sizes in allocation history files produced by memory monitoring [6] is determined by

```
MMUnits          (default: WordSize)
```

A smaller value may be needed if the size of any block in Icon's allocated block region is not an even multiple of WordSize. This occurs, for example, on computers with 80-bit (1-1/2 word) floating-point numbers, in which case the value of MMUnits should be defined to be 2.

Clock Rate: Hz defines the units returned by the *times()* function call. Check the man page for this function on your system. If it says that times are returned in terms of 1/60 second, no action is needed. Otherwise, define Hz in define.h to be the number of *times()* units in one second.

The man page may refer you to an additional file such as /usr/include/sys/param.h. If so, check the value there, and define Hz accordingly.

Executable Images: If you have a BSD UNIX system and want to enable the function save(s), which allows an executable image of a running Icon program to be saved [9], add

```
#define ExecImages
```

to define.h.

External functions and calling Icon from C: Version 8 of Icon provides a mechanism for calling C functions from Icon programs [10]. Such functions are called external functions. The mechanism for calling C functions from Icon normally is enabled. It can be disabled by adding

```
#define NoExternalFunctions
```

to define.h.

It also is possible to call an Icon program from C [10]. This feature normally is disabled. It can be enabled by adding

```
#define IconCalling
```

to define.h.

The ability to call an Icon program from C is incompatible with executable images. If ExecImages is defined, IconCalling has no effect.

Large integers: Version 8 of Icon supports arithmetic on integers of arbitrarily large magnitude. This features increases the size of iconx by 15-20%. It may be disabled by adding

```
#define NoLargeInts
```

to define.h.

Co-expressions: The implementation of co-expressions requires an assembly-language routine. Initially, define.h contains

```
#define NoCoexpr
```

This definition disables co-expressions. Leave this definition in for the first round, although you may want to remove it later and implement these features (see Step 7).

Step 3: Makefile Headers

The file icont.hdr provides headers for Makefiles in the source directories src/common, src/icont and src/memmon. The file iconx.hdr provides a header for the Makefile in src/iconx. These headers are prepended to the standard bodies for the Makefiles during configuration.

These headers serve to specify flags for *cc(1)* and *ld(1)* via CFLAGS, LDFLAGS, and LIBS. If your C optimizer is robust, you may wish to start with

```
CFLAGS= -O
```

in all these headers. If you encounter problems during testing, suspect your optimizer first and try compiling Icon without the -O flag.

Other *cc* and *ld* flags vary considerably from system to system. You may want to review your local manual pages for these processors and look at the header files in the other configuration areas.

LIBS is provided in case you need to load system routines after everything else in the *ld* step.

There another definition in iconx.hdr, RSWITCH, which depends on whether the local co-expression context switch is written in C or assembly language. The initial value of this definition is rswitch.c and a dummy C routine is provided. To start out, leave this definition as it is; the default routine can be replaced later. See Step 7.

The file pi.hdr provides a header for the personalized interpreter Makefile (which is named Pimakefile). In addition to the usual *cc* and *ld* flags, you should provide definitions for XCFLAGS and XLDFLAGS that are the same as those for CFLAGS and LDFLAGS in icont.hdr. This assures that the header file in the personalized interpreter is the same size as the one in the regular version of Icon.

The file vt.hdr provides a header for the variant translator Makefile (which is named Vtmake2). It should have the same *cc* and *ld* flags as icont.hdr.

Step 4: Installation

Once you have edited the files as described in the previous steps, proceed with the installation as described in Steps 5 through 9 at the beginning of Section 2. You may need to iterate if problems show up. If you make a change in a configuration file after a compilation, be sure to perform the configuration step again; some aspects of the configuration are far-reaching and not obvious.

Note: The configuration system is designed to avoid the need for modifications to the distributed source code for Version 8. However, you may run into problems that require modifications to the source code itself. If you need to modify the source code, do it under the control of conditional compilation keyed to the name of your system. Add

```
#define NAME
```

to define.h, where *NAME* is an all-uppercase name that identifies your system. For example, the define.h for Sun Workstations contains

```
#define SUN
```

Then use

```
#ifdef NAME
        .
        .
        .
#endif                                  /* NAME */
```

or similar constructions where you need local source-code modifications. For example, this technique can be used to handle the problem that may arise with SysTime, described in Step 2. Note that nested #ifdefs may be needed in places where there are several different local modifications.

It is important to be consistent and careful about the use of such conditional compilations; if done properly, your modifications can be backed into the master version of the source code at the Icon Project and will be in place for you when subsequent versions are released. See Step 11.

If you need to add C functions to your implementation, put them in tlocal.c for icont and in rlocal.c for iconx.

Step 5: Testing

More testing is recommended for a new installation than for one that has been successfully installed elsewhere. You should do Step 10 at the beginning of Section 2:

```
make Test-all
```

Step 6: Overflow Checking

The code to check integer overflow in iconx is written in C. Some improvement in performance may be obtained by replacing this C code by assembly-language code. This is entirely optional. If you want to do it, add

```
#define AsmOver
```

to define.h and provide assembly-language routines add(), sub(), mul(), and neg() using the corresponding C routines at the end of rmisc.c as examples.

If your C compiler supports the asm directive, place your code in rlocal.c in the UNIX section under control of an appropriate defined symbol that identifies your system. If you need to define such a symbol, place the definition is define.h.

Step 7: Co-Expressions

Once Icon is working properly, you may wish to implement co-expressions. *Note:* If your system does not allow the C stack to be at an arbitrary place in memory, there is probably little hope of implementing co-expressions. If you do not implement co-expressions, the only effect will be that Icon programs that attempt to use co-expressions will terminate with an error message.

All aspects of co-expression creation and activation are written in C in Version 8 except for a routine, coswitch, that is needed for context switching. This routine requires assembly language, since it must manipulate hardware registers. It either can be written as a C routine with asm directives or as an assembly language routine.

When a new configuration directory is set up, a file rswitch.c is provided with a version of coswitch that results in error termination if an Icon program attempts to activate a co-expression. If you implement coswitch in C, modify rswitch.c. Alternatively, if you implement coswitch in assembly language, place it in a new file, rswitch.s.

Calls to the context switch have the form coswitch(old_cs,new_cs,first), where old_cs is a pointer to an array of words that contain C state information for the current co-expression, new_cs is a pointer to an array of words that hold C state information for a co-expression to be activated, and first is 1 or 0, depending on whether or not the new co-expression has or has not been activated before. The zeroth element of a C state array always contains the hardware stack pointer (*sp*) for that co-expression. The other elements can be used to save any C frame pointers and any other registers your C compiler expects to be preserved across calls.

The default size of the array for the C state is 15. This number may be changed by adding

```
#define CStateSize n
```

to define.h, where *n* is the number of elements needed.

The first thing coswitch does is to save the current pointers and registers in the old_cs array. Then it tests first. If first is zero, coswitch sets *sp* from new_cs[0], clears the C frame pointers, and *calls* interp. If first is not zero, it loads the (previously saved) *sp*, C frame pointers, and registers from new_cs and returns.

Written in C, coswitch has the form:

```
/*
 * coswitch
 */
coswitch(old_cs, new_cs, first)
long *old_cs, *new_cs;
int first;
{
                                        .
                                        .
                /* save sp, frame pointers, and other registers in old_cs */
                                        .
                                        .
    if (first == 0) {                                   /* this is first activation */
                                        .
                                        .
                /* load sp from new_cs[0] and clear frame pointers */
                                        .
                                        .
        interp(0, 0);
        syserr("interp() returned in coswitch");
        }

    else {
                                        .
                                        .
                /* load sp, frame pointers, and other registers from new_cs */
                                        .
                                        .
        }
}
```

Appendix B contains a listing of coswitch for a VAX running BSD. Other examples are contained in the configuration directories in config/unix.

After you implement coswitch, remove the #define NoCoexpr from define.h and replace rswitch.c or rswitch.s in your configuration directory as described above. The configuration process will copy your file to the appropriate place prior to compilation. If you use rswitch.s, change the definition of RSWITCH in iconx.hdr to

```
RSWITCH=rswitch.s
```

If your assembler requires special flags, add an appropriate definition for OFLAGS to iconx.hdr.

To test your context switch,

```
make Test-coexpr
```

Ideally, there should be no differences in the comparison of outputs.

If you have trouble with your context switch, the first thing to do is double-check the registers that your C compiler expects to be preserved across calls — different C compilers on the same computer may have different requirements.

Another possible source of problems is built-in stack checking. Co-expressions rely on being able to specify an arbitrary region of memory for the C stack. If your C compiler generates code for stack probes that expects the C stack to be at a specific location, you may need to disable this code (via a C compiler option) or replace it with something more appropriate.

Step 8: Personalized Interpreters

The personalized interpreter system uses *ar(1)*. On most UNIX systems, it is necessary to use *ranlib(1)* so that the loader can access the archive. The script Ranlib that is provided when a new configuration directory is initialized contains calls of *ranlib* for this purpose.

Some UNIX systems, notably System V, handle this problem directly in *ar(1)* and do not have *ranlib(1)*. If your system does not use *ranlib(1)*, change Ranlib to an empty script by

```
echo "" >Ranlib
```

in your configuration directory.

Test your personalized interpreter system as described in Section 2.3.

Step 9: Variant Translators

Normally no work is needed for variant translators on a newly configured system. Test them as described in Section 2.4.

Step 10: Memory Monitoring

Normally no work is needed for memory monitoring on a newly configured system. Test it as described in Section 2.5.

Step 11: Benchmarking

Run the benchmarks as described in Section 2.6.

Step 12: Status Information

Each configuration directory contains a file named status that describes the state of the configuration. A placeholder is provided when a new configuration directory is set up. When your configuration is complete, edit status appropriately, using a status file in another configuration directory as a model.

Step 13: Sending Your Configuration Information to the Icon Project

When your newly installed system is complete, send a copy of any files you modified (both in your configuration directory and in the source code, if changes there were necessary) to the Icon Project as given in Section 4.

Your changes will be added to the master Icon system and will be available in future releases of Icon.

Files can be sent on any convenient medium, such as MS-DOS diskettes or magnetic tape in *tar* or *cpio* format. Electronic mail also can be used.

4. Communicating with the Icon Project

If you run into problems with the installation of Version 8 of Icon, contact the Icon Project:

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, AZ 85721
U.S.A.

(602) 621-4049

icon-project@cs.arizona.edu (Internet)
… {uunet, allegra, noao}!arizona!icon-project (uucp)

Please also let us know if you have any suggestions for improvements to the installation process or corrections or refinements to configuration files for supported systems.

References

1. R. E. Griswold, *Installation Guide for Version 8 of Icon on UNIX Systems*, The Univ. of Arizona Tech. Rep. 90-2, 1990.
2. R. E. Griswold, *Transporting Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-5, 1990.
3. R. E. Griswold, *The Icon Program Library*, The Univ. of Arizona Tech. Rep. 90-7, 1990.
4. R. E. Griswold, *Personalized Interpreters for Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-3, 1990.
5. R. E. Griswold and K. Walker, *Variant Translators for Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-4, 1990.
6. G. M. Townsend, *The Icon Memory Monitoring System*, The Univ. of Arizona Icon Project Document IPD113, 1990.
7. Technical Committee X3J11, *Draft Proposed American National Standard for Information Systems — Programming Language C*, 1988.
8. R. E. Griswold, *ICONT(1)*, manual page for *UNIX Programmer's Manual*, The Univ. of Arizona Icon Project Document IPD109, 1990.
9. R. E. Griswold, *Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-1, 1990.
10. R. E. Griswold, *Icon-C Calling Interfaces*, The Univ. of Arizona Tech. Rep. 90-8, 1990.

Appendix A — Supported Configurations

Asterisks make ‘‘supported’’ configurations that have been tested under Version 7.5 or 8.

computer	*UNIX system*	*name*
Amdahl	UTS	amdahl_uts
Apollo Workstation*	BSD	domain_bsd
Astronautics ZS-1	UNIX	zs1
AT&T 3B1 (UNIX PC)	System III	unixpc
AT&T 3B2	System V	att3b_2
AT&T 3B5	System V	att3b_5
AT&T 3B15	System V	att3b_15
AT&T 3B20	System V	att3b_20
AT&T 3B4000	System V	att3b_4000
AT&T 6386	System V	att6386
CDC Cyber	NOS/VE	cdc_vxve
Celerity	4.2BSD	celerity_bsd
Codata 3400	Unisis	codata
Convergent MegaFrame	CTIX	mega
Convex C-1/2	BSD	convex_bsd
Cray-2	UNICOS	Cray-2
DecStation 3100*	Ultrix	dec_3100
DG AViiON*	System V	aviion
DIAB	D-NIX	diab_dnix
Elxsi-6400*	BSD	elxsi_bsd
Encore	UMAX	multimax_bsd
Gould Powernode	UTX	gould_pn
HP 9000/330*	HP-UX	hp9000_s300
HP 9000/500	HP-UX	hp9000_s500
HP 9000/800	HP-US	hp9000_s800
IBM PS/2	AIX	ps2_aix
IBM RT Workstation	ACIS	rtpc_acis
IBM RT Workstation	AIX	rtpc_aix
Intel 286*	XENIX 286	i286_xenix
Intel 386*	System V	i386_sysv
Intel 386*	XENIX 386	i386_xenix
Iris 4D/20*	BSD	iris4d
Intergraph Clipper	System V	clix
Masscomp 5500	System V	masscomp
Microport V/AT	System V	microport
MIPS/r3000*	System V	mips
Motorola 8000/400	System V	mot_8000
Multifow Trace	UNIX	trace
NeXT*	Mach	next
Plexus P60	System V	plexus
Pyramid 90x	4.2BSD	pyramid_bsd
Ridge 32	ROS	ridge
Sequent Balance 8000	Dynix	balance_dynix2
Sequent Symmetry*	Dynix	symmetry
Siemens MX500	SINIX	mx_sinix
Sun 2 Workstation	SunOS	sun2
Sun 3 Workstation*	SunOS	sun3

Sun 3 with 68881	SunOS	sun3_68881
Sun 386i	SunOS	sun386i
Sun 4 Workstation*	SunOS	sun4
Unisys 7000/40	4.3BSD	tahoe_bsd
VAX-11	4.1BSD	vax_41_bsd
VAX-11*	4.2BSD and 4.3BSD	vax_bsd
VAX-11	System V	vax_sysv
VAX-11	Ultrix	vax_ultrix
VAX-11	9th Edition	vax_v9

Appendix B — A Sample Co-Expression Context Switch

The co-expression context switch for a VAX running BSD is:

```
coswitch(old_cs, new_cs, first)
int *old_cs, *new_cs;
int first;

{
    asm("   movl 4(ap),r0");
    asm("   movl 8(ap),r1");
    asm("   movl sp,0(r0)");
    asm("   movl fp,4(r0)");
    asm("   movl ap,8(r0)");
    asm("   movl r11,16(r0)");
    asm("   movl r10,20(r0)");
    asm("   movl r9,24(r0)");
    asm("   movl r8,28(r0)");
    asm("   movl r7,32(r0)");
    asm("   movl r6,36(r0)");

    if (first == 0) {           /* this is first activation */
        asm("   movl 0(r1),sp");
        asm("   clrl fp");
        asm("   clrl ap");
        interp(0, 0);
        syserr("interp() returned in coswitch");
        }
```