Configuring the Source Code for Version 8.10 of Icon

Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend

Department of Computer Science, The University of Arizona

1. Background

The implementation of the Icon programming language is large and sophisticated [1-3]. The implementation is, however, written almost entirely in C and RTL [4], a superset of C, for which a translator to C is provided. A small amount of assembly-language code is needed for the context switch used by co-expressions. This code is optional and only affects co-expressions.

There presently are implementations of Icon for the Acorn Archimedes, the Amiga, the Atari ST, the Macintosh, MS-DOS, MVS, OS/2, UNIX, VM/CMS, and VMS.

All implementations of Icon are obtained from the same source code, using conditional compilation and defined constants to select and configure platform-dependent code. Consequently, installing Icon on a new platform is largely a matter of selecting appropriate values for configuration parameters, deciding among alternative definitions, and possibly adding some code that is dependent on the specific computer, operating system, and C compiler used.

This document describes the process of configuring Version 8.10 of the Icon source code for a platform on which it has not previously been installed.

Since there are several existing configurations for UNIX and MS-DOS, configuring a new platform for one of these operating systems is easier than for other platforms. See Sections 5, 6, and 7 for specific information concerning UNIX and MS-DOS platforms.

Building Icon with a new C compiler on an operating system for which Icon has previously been installed usually is a fairly simple task and normally can be done by adjusting a few configuration parameters.

Installing Icon on a new operating system is more complex; read this report carefully, especially Section 8 before undertaking such a project.

2. Requirements

C Data Sizes

Icon places the following requirements on C data sizes:

- *char*s must be 8 bits.
- *int*s must be 16, 32, or 64 bits.
- *long*s and pointers must be 32 or 64 bits.
- All pointers must be the same length.
- *long*s and pointers must be the same length.

If your C data sizes do not meet these requirements, do not attempt to configure Icon.

The C Compiler

The main requirement for implementing Icon is a production-quality C compiler that supports at least the *de facto* ''K&R'' standard [5]. An ANSI C compiler is preferable. The term ''production quality'' implies robustness, correctness, the ability to address large amounts of memory, the ability to handle large files and complicated expressions, and a comprehensive run-time library.

The C preprocessor should conform either to the ANSI C standard [6] or to the *de facto* standard for UNIX C preprocessors. In particular, Icon uses the C preprocessor to concatenate strings and substitute arguments within quotation marks. For the ANSI preprocessor standard, the following definitions are used:

```
#define Cat(x, y) x##y
#define Lit(x) #x
```

For the UNIX *de facto* standard, the following definitions are used:

```
#define Ident(x) x
#define Cat(x, y) Ident(x)y
#define Lit(x) "x"
```

The following program can be used to test these preprocessor facilities:

```
Cat(ma, in)()
{
   printf(Lit(Hello world\n));
}
```

If this program does not compile and print Hello world using one of the sets of definitions above, there is no point in proceeding. Contact the Icon Project for alternative approaches.

Memory

The Icon programming language requires a substantial amount of memory to run. The practical minimum depends somewhat on the platform; 640KB is typical.

File Space

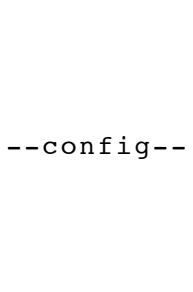
The source code for Icon is large — about 3.5MB. Test programs and other auxiliary files take additional room, as does compilation and testing. While the implementation can be divided into components that can be built separately, this approach may be painful.

3. File Structure

The files for Icon are organized in a hierarchy. The top level, assuming the hierarchy is rooted in icon is:

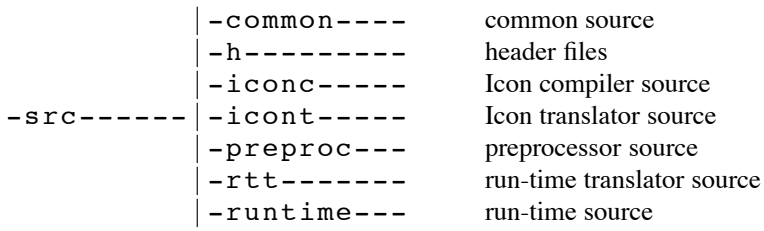
```
            |-bin------     executable binaries and support files
            |-config---     configurations
|-icon----  |-src------     source code
            |-tests----     tests
```

There are several subdirectories in config for different operating systems:



Not all of these subdirectories are included in all distributions of Icon. Some configuration directories contain subdirectories for different platforms. These subdirectories contain various files, depending on the platform.

The directory src contains the source code for various components of Icon.



Some distributions contain other, optional components of Icon. The Icon compiler is not included in all distributions of Icon.

4. Parameters and Definitions

There are many defined constants and macros in the source code for Icon that vary from platform to platform. Over the range of possible platforms, there are many possibilities. A complete list is given in the appendix. *Do not be intimidated by the large number of options listed there;* most are provided only for unusual situations and only a few are needed for any one platform.

The defined constants and macros needed for a specific platform are placed in src/h/define.h. There are many existing define.h files that can be used as guides. One for a ''vanilla'' 32-bit platform is:

```
#define HostStr "new host"
#define NoCoexpr

#define PORT 1
```

HostStr provides the value used in the Icon keyword &host and should be changed as appropriate. NoCoexpr causes Icon to be configured without co-expressions. This definition can be removed when co-expressions are implemented. PORT indicates an implementation for an unspecified operating system. It should be changed to a name for the operating system for the new platform (see Section 8). Other definitions probably need to be added, of course.

5. Configuring Icon for a UNIX Platform

Since Icon has been implemented for many UNIX platforms, the easiest way to configure Icon for a new UNIX platform usually is to copy an existing configuration for a platform that is similar to the new one. A few modifications then often suffice to get Icon running on the new platform.

In addition to define.h, a UNIX configuration also contains headers used to construct Makefiles. These headers are named ∗.hdr. Check these headers for appropriateness.

See also [7] for information concerning the installation of Icon on a UNIX platform.

6. Adding Configuration Information for the X Window System

If your platform has X Window software, you may wish to configure Icon Version 8.10 with X support. Icon's X support consists of a collection of Icon functions that call Xlib, the standard C interface to X. At present, configuration of X Window facilities is provided only for UNIX platforms.

In order to build Icon with these X Window functions, you will need to know what library or libraries are required to link in the X facilities into C programs; this library information is needed when iconx is built and when iconc links a compiled Icon executable. Normally, the answer will be –lX11, but on some platforms additional libraries or alternate paths are required. Consult appropriate manuals to find out what libraries are needed.

If your platform requires the default –lX11, no additional steps are required in creating your configuration. If your platform requires additional libraries, you will need to add files to the configuration directory for your particular system.

The files xiconx.mak and xiconc.def, if they are present, are used during Icon configuration to supply non-default library information to the interpreter and the compiler.

If, for example, your platform requires an additional pseudo-terminal library and a bsd-compatibility package in order to link X applications, you would add an xiconx.mak file with the line

```
XLIB= –L../../bin –lX11 –lpt –lbsd
```

and a corresponding xiconc.def file with the line

```
#define ICONC_XLIB "–lX11 –lpt –lbsd"
```

The former file gets prepended to the Makefile that builds iconx, while the latter file gets included and compiled into iconc when X is configured. Then proceed to the make X-Configure build step.

In order to build Icon with X support, some platforms also will have to specify the location of the X header files. Normally they are in /usr/include/X11; if they are in some other location on your platform, you will need to locate them and identify the appropriate option to add to the C compiler command line, usually –I *path*, where *path* is the directory above the X11 include directory.

For the Icon compiler, this option is added via the COpts macro in define.h for your configuration. The COpts macro must define a quoted C string. For the interpreter, the option is added to the CFLAGS argument of the common.hdr, icont.hdr, runtime.hdr, and xpm.hdr Makefile headers for your configuration.

7. Configuring Icon for an MS-DOS Platform

In the case of MS-DOS, the primary considerations in configuring Icon have to do with the C compiler that is used. There are existing configurations for several 16- and 32-bit C compilers.

The easiest approach to configuring Icon for a new MS-DOS C compiler is to copy an existing configuration for a C compiler that most closely matches the new one.

An MS-DOS configuration includes Makefiles, batch scripts, and response files for linking. These files should be modified for the new platform as appropriate. See [8] for more information concerning the installation of Icon on an MS-DOS platform.

8. Configuring Icon for a New Operating System

The conditional compilation for specific operating systems is concerned primarily with matters such as differences in file naming, the handling of input and output, and environmental factors.

The presently supported operating system are RISC OS for the Acorn Archimedes, AmigaDos, Atari ST TOS, the Macintosh under MPW, MS-DOS, MVS, OS/2, UNIX, and VM/CMS, and VMS. There are hooks for a different operating system. The associated defined symbols are

AMIGA	AmigaDos
ARM	RISC OS for the Acorn Archimedes
ATARI_ST	Atari ST TOS
MACINTOSH	Macintosh
MSDOS	MS-DOS
MVS	MVS
OS2	OS/2
PORT	different operating system
UNIX	UNIX
VM	VM/CMS
VMS	VMS

Conditional compilation uses logical expressions composed from these symbols. An example is:

```
                    .
                    .
                    .
#if MSDOS
                    .
                    .                   /* code for MS-DOS */
                    .
#endif

#if UNIX || VMS
                    .
                    .                   /* code for UNIX and VMS */
                    .
#endif
                    .
                    .
                    .
```

Each symbol is defined to be either 1 (for the target operating system) or 0 (for all other operating systems). This is accomplished by defining the symbol for the target operating system to be 1 in define.h. In config.h, which includes define.h, all other operating-system symbols are defined to be 0.

Logical conditionals with #if are used instead of defined or undefined names with #ifdef to avoid nested conditionals, which become very complicated and difficult to understand when there are several alternative operating systems. Note that it is important not to use #ifdef accidentally in place of #if, since all the names are defined.

The file define.h for a different operating system should initially contain

```
#define PORT 1
```

as indicated in Section 4. You can use PORT during the configuration for a different operating system. Later you should come back and change PORT to some more appropriate name.

Note: The PORT sections contain deliberate syntax errors (so marked) to prevent sections from being overlooked during configuration. These syntax errors must, of course, be removed before compilation.

To make it easy to locate places where there is code that may be dependent on the operating system, such code usually is bracketed by unique comments of the following form:

```
/*
 * The following code is operating-system dependent.
 */
                              .
                              .
                              .
/*
 * End of operating-system specific code.
 */
```

Between these beginning and ending comments, the code for different operating systems is provided using conditional expressions such as those indicated above.

Look through some of the files for such segments to get an idea of what is involved. Each segment contains comments that describe the purpose of the code. In some cases, the most likely code or a suggestion is given in the conditional code under PORT. In some cases, no code will be needed. In others, code for an existing operating system may suffice for the new one.

In any event, code for the new operating system name must be added to each such segment, either by adding it to a logical disjunction to take advantage of existing code for other operating systems, as in

```
#if MSDOS || UNIX || PORT
        .
        .
        .
#endif

#if VMS
        .
        .
        .
#endif
```

and removing the present code for PORT or by filling in the segment with the appropriate code, as in

```
#if PORT
                              .
                              .                    /* code for the new operating system */
                              .
#endif
```

If no code is needed for the target operating system in a particular situation, a comment should be provided so that it is clear that the situation has been considered.

You may find need for code that is operating-system dependent at a place where no such dependency presently exists. If the situation is idiosyncratic to your operating system, which is most likely, simply use a conditional for PORT as shown above. If the situation appears to need different code for several operating systems, add a new segment similar to the other ones, being sure to provide something appropriate for all operating systems.

Do not use #else constructions in these segments; this increases the probability of logical errors and obscures the mutually exclusive nature of operating system differences.

9. Trouble Reports and Feedback

If you run into problems, contact us at the Icon Project:

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

(602) 621-8448 (voice)
(602) 621-4246 (fax)

icon-project@cs.arizona.edu (Internet)
… uunet!arizona!icon-project (uucp)

Please also let us know of any suggestions for improvements to the configuration process.

Once you have completed your installation, please send us copies of any files that you modified so that we can make corresponding changes in the central version of the source code. Once this is done, you can get a new copy of the source code whenever changes or extensions are made to the implementation. Be sure to include documentation on any features that are not implemented in your installation or any changes that would affect users.

References

1. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
2. R. E. Griswold, *Supplementary Information for the Implementation of Version 8 of Icon*, The Univ. of Arizona Icon Project Document IPD112, 1992.
3. R. E. Griswold, *Supplementary Information for the Implementation of Version 8.10 of Icon*, The Univ. of Arizona Icon Project Document IPD215, 1993.
4. K. Walker, *The Run-Time Implementation Language for Version 8.7 of Icon*, The Univ. of Arizona Tech. Rep. 92-18. July 1992.
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
6. *American National Standard for Information Systems — Programming Language - C, ANSI X3.159-1989*, American National Standards Institute, New York, 1990.
7. R. E. Griswold, C. L. Jeffery and G. M. Townsend, *Installing Version 8.10 of Icon on UNIX Platforms*, The Univ. of Arizona Icon Project Document IPD218, 1993.
8. R. E. Griswold, *Building Version 8.10 of Icon for MS-DOS and OS/2*, The Univ. of Arizona Icon Project Document IPD223, 1993.
9. R. E. Griswold, *Icon-C Calling Interfaces; Version 8.10*, The Univ. of Arizona Icon Project Document IPD217, 1993.

Appendix — Configuration Parameters and Definitions

C Compiler Considerations

Although the source code for Icon does not require an ANSI C compiler, installation is likely to be much easier if such a compiler is used. It is necessary to specify that the compiler is ANSI-conformant in order to take advantage of ANSI features.

If your C compiler is completely ANSI-conformant, add

```
#define Standard
```

to define.h.

Alternatively, you can define one or more of the following constants if only a portion of your C compiler is ANSI-conformant:

```
#define StandardPP          /* standard preprocessor */
#define StandardC           /* standard compiler proper */
#define StandardLib         /* standard library */
```

If your C compiler supports the void type but is not ANSI-conformant, add

```
#define VoidType
```

to define.h.

If your C compiler supports function prototypes but is not ANSI-conformant, add

```
#define Prototypes
```

Icon normally defines the C language const with an empty definition to remove its effect. If you have an ANSI C compiler and this definition causes problems, add

```
#define AllowConst
```

to define.h.

On some platforms it may be necessary to provide a different typedef for pointer than is provided by default. For example, on the huge-memory-model implementation of Icon for Microsoft C on MS-DOS, its define.h contains

```
typedef huge void *pointer;
```

If an alternative typedef is used for pointer, add

```
#define PointerDef
```

to define.h to avoid the default one.

Sometimes computing the difference of two pointers causes problems. Pointer differences are computed using the macro DiffPtrs(p1, p2), which has the default definition:

```
#define DiffPtrs(p1, p2) (word)((p1)-(p2))
```

where word is a typedef that is provided automatically and usually is *long int*.

This definition can be overridden in define.h. For example, Microsoft C for the MS-DOS large memory model uses

```
#define DiffPtrs(p1, p2) ((word)(p1)-(word)(p2))
```

If you provide an alternate definitions for pointer differencing, be careful to enclose all arguments in parentheses.

Character Set

The default character set for Icon is ASCII. If you are configuring Icon for a platform that uses the EBCDIC character set, add

```
#define EBCDIC 1
```

to define.h.

Data Sizing and Alignment

There are two constants that relate to the size of C data:

IntBits	(default: 32)
WordBits	(default: 32)

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.

The word alignment of stacks used by co-expressions is controlled by

StackAlign	(default: 2)

If your platform needs a different alignment, provide an appropriate definition in 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.

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 platform, add appropriate definitions to define.h.

Options for Opening Files

The options for opening files with *fopen()* are given by the following constants:

ReadBinary	(default: "rb")
ReadText	(default: "r")
WriteBinary	(default: "wb")
WriteText	(default: "w")

These defaults can be changed by definitions in define.h.

Library Routines

Support from some library routines varies from platform to platform, and names vary. The relevant definitions are:

NoIconGcvt	(default: undefined)
NoSelect	(default: undefined)
SysMem	(default: undefined)
SysOpt	(default: undefined)
index	(default: undefined)
rindex	(default: undefined)

By default, Icon provides its own version of *gcvt()* to minimized the differences in the string representations of floating-point numbers between different platforms. If you prefer to use the version of *gcvt()* in your C library, add

```
#define NoIconGcvt
```

to define.h.

The Icon function delay() uses the C function *select()*. If your C library does not have *select()*, add

```
#define NoSelect
```

to define.h. The effect of this is that delay() fails and does not delay execution.

If SysMem is defined and IntBits == WordBits, the C library routines *memcpy()* and *memset()* are used in place of the corresponding Icon routines *memcopy()* and *memfill()*. SysMem is automatically defined if Standard or StandardLib is defined.

If SysOpt is defined, the C library function *getopt()* is used instead of Icon's private version.

Different C compilers 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 platform uses the latter names, add

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

to define.h.

Icon uses unlink for the routine that deletes a file. The other common name is remove. If your platform uses this name, for example, add

```
#define unlink remove
```

to define.h.

Storage Region Sizes

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

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, 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.

Allocation Sizing

malloc() is used to allocate space for Icon's storage regions. This limits region sizes to the value of the largest *unsigned int*. Some platforms provide alternative allocation routines for allocating larger regions. To change the allocation procedure for regions, add a definition for AllocReg to define.h. For example, the huge-memory-model implementation of Icon for Microsoft C uses the following:

```
#define AllocReg(n) halloc((long)n, sizeof(char))
```

Note: Icon still uses *malloc()* for allocating other blocks. If this is a problem, it may be possible to change this by defining malloc in define.h, as in

```
#define malloc lmalloc
```

where *lmalloc()* is a local routine for allocating large blocks of memory. If this is done, and the size of the allocation is not *unsigned int*, add an appropriate definition for the type by defining AllocType in define.h, such as

```
#define AllocType unsigned long int
```

It is also necessary to add a definition for the limit on the size of an Icon region:

```
#define MaxBlock n
```

where *n* is the maximum size allowed (the default for MaxBlock is MaxUnsigned, the largest *unsigned int*). It generally is not advisable to set MaxBlock to the largest size an alternative allocation routine can return. For the huge-memory-model implementation mentioned above, MaxBlock is 256000.

File Name Suffixes

The suffixes used to identify Icon source programs, ucode files, and icode files may be specified in define.h:

```
#define SourceSuffix    (default: ".icn")
#define U1Suffix        (default: ".u1")
#define U2Suffix        (default: ".u2")
#define USuffix         (default: ".u")
#define IcodeSuffix     (default: "")
#define IcodeASuffix    (default: "")
```

USuffix is used for the abbreviation that icont understands in place of the complete U1Suffix or U2Suffix. IcodeASuffix is an alternative suffix that iconx uses when searching for icode files specified without a suffix. For example, on MS-DOS, IcodeSuffix is ".icx" and IcodeASuffix is ".ICX".

If values other than the defaults are specified, care must be taken not to introduce conflicts or collisions among names of different types of files.

Paths

If icont is given a source program in a directory different from the local one (''current working directory''), there is a question as to where ucode and icode files should be created: in the local directory or in the directory that contains the source program. On most platforms, the appropriate place is in the local directory (the user may not have write permission in the directory that contains the source program). However, on some platforms, the directory that contains the source file is appropriate. By default, the directory for creating new files is the local directory. The other choice can be selected by adding

```
#define TargetDir SourceDir
```

UNIX Bootstrap Headers

A bootstrap header is used to make UNIX icode files executable. The space reserved for the header is determined by

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

On some UNIX platforms, 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.

Command-Line Options

The command-line options that are supported by icont and iconc are defined by Options. The default value (see config.h) will do for most platforms, but an alternative can be included in define.h.

Similarly, the error message produced for erroneous command lines is defined by Usage. The default value, which should correspond to the value of Options, is in config.h, but may be overridden by a definition in define.h.

Environment Variables

If your platform does not support environment variables (via the run-time library routine getenv), add the following line to define.h:

```
#define NoEnvVars
```

This disables Icon's ability to change internal parameters to accommodate special user needs (such as using memory region sizes different from the defaults), but does not otherwise interfere with the use of Icon.

Host Identification

The identification of the host computer as given by the Icon keyword &host needs to be specified in define.h. The usual way to to this is to add

```
#define HostStr "identification"
```

to define.h.

Several alternatives are available on UNIX platforms for host identification. To use one of these, 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.

Exit Codes

Exit codes are determined by the following definitions:

NormalExit	(default: 0)
ErrorExit	(default: 1)

Clock Rate

Hz defines the units returned by the *times()* function call. Check the documentation for this function on your platform. 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 documentation may refer you to an additional file such as /usr/include/sys/param.h. If so, check the value there, and define Hz accordingly.

Dynamic Hashing Constants

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 platforms. 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.

Keyboard Functions

If your platform supports the keyboard functions *getch()*, *getche()*, and *kbhit()*, add

```
#define KeyboardFncs
```

to define.h.

Some UNIX platforms are capable of supporting the keyboard functions. A UNIX platform should be able to to support the keyboard functions if it supports *ioctl()* in a manner that is compatible with either BSD/SunOS or System V. The keyboard functions are enabled by default on platforms known to be compatible, including SunOS, Xenix, System V/386, and a few others.

On other platforms, in addition to adding

```
#define KeyboardFncs
```

to define.h as described above, select *one* of two methods for calling *ioctl()* supported in the source code for Icon, and add it to define.h:

```
#define HaveTioc
```

or

```
#define HaveTermio
```

Use HaveTioc if you have a BSD-based platform and have the system include files <sys/ioctl.h>, <errno.h>, and <sys/signal.h>. The system *ioctl()* call must support TIOCSETN, TIOCGETP, TIOCGETC, and FIONREAD.

Use HaveTermio if you have a System V, Xenix 386, or compatible platform and have the system include files <sys/termio.h>, <sys/ioctl>, <sys/errno.h>, and <sys/signal.h>. The system *ioctl()* call must support TCSETA, TCGETA, and (unless it is a Xenix platform), FIONREAD.

If your platform supports neither of these methods, you may be able to write your own keyboard functions. See src/runtime/fsys.r.

Other Optional Features

Some features of Icon are optional. Some of these normally are enabled, while others normally are disabled. The features that normally are enabled can be disabled to, for example, reduce the size of the executable files. A negative form of definition is used for these, as in

```
#define NoLargeInts
```

which can be added to define.h to disable large-integer arithmetic. It may be necessary to disable large-integer arithmetic on computers with a small amount of memory, since the feature increases the size of the run-time system by 15-20%.

The implementation of co-expressions requires an assembly-language routine.

```
#define NoCoexpr
```

disables co-expressions.

If your platform supports the *system()* function for executing command line, add

```
#define SystemFnc
```

to define.h.

One optional feature of the Icon interpreter that normally is disabled is the ability to call an Icon program from a C function [9]. This feature can be enabled by adding

```
#define IconCalling
```

to define.h.

If you have a BSD UNIX platform and want to enable the function save(s), which allows an executable image of a running Icon program to be saved, add

```
#define ExecImages
```

to define.h.

X Window Facilities

The files needed to build Icon with X Window facilities are not in the same places on all platforms. If Icon fails to build because an include file needed by X cannot be found, it may be necessary to edit src/h/sys.h to reflect the local location.

Some early versions of X Window Systems, notably X11R3, do not support the attribute iconic. If this is the case for your platform, add

```
#define NoIconify
```

to define.h. This disables the attribute iconic, causing references to it to fail.

Compiler Options

The C compiler called by the Icon compiler to process its output defaults to cc. If you want to use a different C compiler, add

```
#define CComp "name"
```

to define.h, where *name* is the name of the C compiler you want the Icon compiler to use. Note the quotation marks surrounding the name. For example, to use Gnu C, add

```
#define CComp "gcc"
```

By default, the C compiler is called with no options. If you want specific options, add

```
#define COpts "options"
```

to define.h. Again, note the quotation marks. For example, to request C optimizations, you might add

```
#define COpts "-O"
```

If your system does not have ranlib, add

```
#define NoRanlib
```

to define.h.

Debugging Code

Icon contains some code to assist in debugging. It is enabled by the definitions

```
#define DeBugTrans      /* debugging code for the translator in icont */
#define DeBugLinker     /* debugging code for the linker in icont */
#define DeBugIconx      /* debugging code for the run-time */
```

All three of these are automatically defined if DeBug is defined. DeBug is defined in define.h as it is distributed, so all debugging code is enabled.

The debugging code for the translator consists of functions for dumping symbol tables (see icont/tsym.c). These functions are rarely needed and there are no calls to them in the source code as it is distributed.

The debugging code for the linker consists of a function for dumping the code region (see icont/lcode.c) and code for generating a debugging file that is a printable image of the icode file produced by the linker. This debugging file, which is produced if the option -L is given on the command line when icont is run, may be useful if icode files are incorrect.

The debugging code for the executor consists of a few validity checks at places where problems have been encountered in the past. It also provides functions for dumping Icon values. See runtime/rmisc.r and

runtime/rmemmgt.r.

When installing Icon on a new operating system, it is advisable to enable the debugging code until Icon is known to be running properly. The code produced is innocuous and adds only a few percent to the size of the executable files. It should be removed by deleting the definition listed above from define.h as the final step in the implementation for a new operating system.