Installation Guide for Version 7.5 of Icon on UNIX Systems*

Ralph E. Griswold

TR 88-6b

Last revised November 1, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

Installation Guide for Version 7.5 of Icon on UNIX Systems

1. Introduction

Version 7 of Icon [1] contains a number of new features and improvements to the implementation. Most changes to the language are upward compatible with Versions 5 and 6 and users should be able to run existing programs under Version 7 with no difficulty.

This report provides the information necessary to install Version 7 of Icon on computers running UNIX. For other operating systems, see [2]. *Note:* If you previously installed Version 6 of Icon, you will find a number of minor changes in the installation procedure for Version 7. Be careful, particularly with respect to paths as described in this report.

The implementation of Version 7 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. As of the date given on the cover of this report, the following configurations were included in the distribution:

computer	*UNIX system*	*name*
Amdahl 580	UTS	amdahl_uts
Apollo Workstation	UNIX	domain_bsd
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
Celerity	Berkeley 4.2bsd	celerity_bsd
Codata 3400	Unisis	codata
Convex C-1/2*		Berkeley bsdconvex_bsd
DIAB	D-NIX	diab_dnix
Encore	UMAX	multimax_bsd
Gould Powernode	UTX	gould_pn
HP 9000/320*	HP-UX	hp9000_s300
HP 9000/500	HP-UX	hp9000
IBM PS/2*	XENIX 386 xenix_386	
IBM RT Workstation	ACIS	rtpc_acis
IBM RT Workstation	AIX	rtpc_aix
IBM XT/AT*	XENIX 286 xenix	
Masscomp 5500	System V	masscomp
Microport V/AT	System V	microport
Microport V/386*	System V	mport386
Motorola 8000/400	System V	mot_8000
Plexus P60	System V	plexus
Pyramid 90x	Berkeley 4.2bsd	pyramid_bsd
Ridge 32	ROS	ridge
Sequent Balance 8000*	Dynix	balance_dynix2
Siemens MX500	SINIX	mx_sinix
Sun-2 Workstation*	UNIX 4.2	sun2
Sun-3 Workstation*	UNIX 4.2	sun3
Sun-3 with 68881*	UNIX 4.2	sun3_68881
Sun-4*	UNIX 4.2	sun4
Unisys 7000/40	Berkeley 4.3bsd	tahoe_bsd
UNIX-PC/3B1*	System III	unixpc
VAX-11	Berkeley 4.1bsd	vax_41_bsd
VAX-11*	Berkeley 4.2bsd and 4.3bsd	vax_bsd
VAX-11*	System V	vax_sysv
VAX-11*	Ultrix	vax_ultrix
VAX-11	9th Edition	vax_v9

These systems are referred to as ‘‘supported’’ in this report. They worked under Version 6 of Icon, but not all of them have been tested yet under Version 7. The systems marked with an asterisk have been tested under Version 7 and are referred to as ‘‘tested’’ in this report.

If your system is a tested one, the installation of Version 7 of Icon should be as simple as issuing a few make commands. If your system is supported but untested, it may install 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 be able to install Version 7 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], and a variant translator system [5]. You may want to review the 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 you want the Icon hierarchy to reside.
- 2. Unload the Icon hierarchy at the selected place.
- [3.∗] Check the status of the configuration for your system.
- 4. If you unloaded Icon at a ‘‘nonstandard’’ place, edit a configuration file that contains path specifications.
- 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 re-do 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.

Finishing Up

- [1.] Install UNIX manual pages for the various components of Icon.
- [2.∗] Remove files that are no longer needed.

2.1 Installing Icon Proper

Step 1: Deciding Where to Put Icon

The standard location for all files, including executable binaries, is in the directory /usr/icon/v7†. You can chose other locations, but if you do, you must edit a file before proceeding. Think twice about this: Executable

†For HP-UX, the standard location is /users/icon/v7.

binary files for Icon are referenced by full path names, and they cannot be easily moved; it is important to select the correct paths at the start. *Note:* 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/v7/Makefile for the standard location). Similarly, config/unix corresponds to /usr/icon/v7/config/unix for the standard location.

Step 2: Unloading the Files

The distribution consists of a hierarchy, which is rooted in ‘‘.’’. Icon is distributed in a variety of formats. The usual distribution media is magnetic tape, although it is also available on Sun cartridges and on 5-¼″ diskettes in two formats.

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 standard location is /usr/icon/v7) 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: UNIX Icon is available on diskettes in two formats: XENIX *tar* and *cpio* on MS-DOS diskettes.

For XENIX *tar*, place the first diskette in drive *n* and do

```
tar xn
```

Repeat this process with the remaining diskettes

For *cpio* on MS-DOS diskettes, 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
```

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 Section 1. For example,

```
make Status name=hp9000
```

lists the status of the configuration for the HP 9000 operating under HP-UX.

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 Version 7 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: Editing Paths

If you unloaded Version 7 of Icon at the standard place, skip this section. Otherwise, you need to edit path specifications in your configuration directory. 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.

The file paths.h, as distributed, is the same for all systems and contains

```
#define RootPath        "/usr/icon/v7"
#define IcontPath       "/usr/icon/v7/bin/icont"
#define TranPath        "/usr/icon/v7/bin/itran"
#define LinkPath        "/usr/icon/v7/bin/ilink"
#define IconxPath       "/usr/icon/v7/bin/iconx"
#define HeaderPath      "/usr/icon/v7/bin/iconx.hdr"
```

RootPath gives the location of the Icon hierarchy; it must be the path of the root directory where the distributed files are located.

The five binary files referenced are:

icont	Icon command processor — all the typical user knows about
itran	Icon translator
ilink	Icon linker
iconx	Icon run–time system
iconx.hdr	bootstrap program that gets Icon programs into execution

The command processor icont calls the other programs.

There are two reasons for changing these paths:

1. If the Icon hierarchy is unloaded in an area other than /usr/icon/v7, you probably want the binary files installed in that area instead of /usr/icon/v7.
2. You may want to install some or all of the binary files in a public area.

For example, if you want to unload Icon in /usr/irving/v7 and have the binaries in /usr/local/icon, edit paths.h to be

```
#define RootPath        "/usr/irving/v7"
#define IcontPath       "/usr/local/icon/icont"
#define TranPath        "/usr/local/icon/itran"
#define LinkPath        "/usr/local/icon/ilink"
#define IconxPath       "/usr/local/icon/iconx"
#define HeaderPath      "/usr/local/icon/iconx.hdr"
```

Caution: If you are using a previous version of Icon, do not put iconx where it is — iconx for Version 7 is incompatible with iconx for previous versions and replacing a previous iconx by Version 7 iconx will invalidate all previously compiled Icon programs.

Step 5: Configuring Icon for Your System

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 7 of Icon for a VAX running Berkeley 4.*n*bsd.

Step 6: Checking the Size of a Header File

Translating and linking an Icon program 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, icode files fail to execute. 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          1492 Jan 13 08:32 iconx.hdr
#define MaxHdr  1500
```

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 (or if it is excessively large), 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's 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 re-do 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
```

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 7 Icon.

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 sometimes are insignificant differences in the format of floating-point numbers, the order of random numbers, and the results of sorting (relating to stability). See the main Makefile for more information if you want to run these tests.

2.2 Icon Program Library

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

Note: The present Icon program library is still Version 6. It runs properly under Version 7, however.

Step 1: Building the Icon Program Library

To build the Icon program library, do

```
make Ipl
```

This puts compiled programs in ipl/progs and translated procedures in ipl/procs.

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/progs and the translated procedures in ipl/procs 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 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 3: 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 Finishing Up

Step 1: Installing Manual Pages

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

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, and if you are tight on space you may wish to remove documents and the directory containing the programs from the Icon language book. See Appendix A, which shows the Icon hierarchy.

3. Configuring Version 7 for a New System

Version 7 of Icon assumes that C *int*s are 16, 32, or 64 bits long. (Icon has not yet been installed on a computer with 64-bit *int*s.) 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 eleven 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 tests.

6. Make any necessary corrections, repeating previous steps as necessary.

[7.] Implement and test co-expressions.

[8.] Implement and test arithmetic overflow checking.

[9.] Implement the personalized interpreter system.

10. Provide status information in your configuration directory.

11. Send the contents of your configuration directory to the Icon Project so that your configuration can be added to the system there for future availability.

Step 1: Building a New Configuration Directory

First you need to select a name for your system. The name should consist of a mnemonic for the computer, which may be followed by an underscore and a mnemonic for the operating system, in case there may be 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 command processor Makefile
iconx.hdr	flags and other definitions for the run-time system Makefile
ilink.hdr	flags for the linker Makefile
itran.hdr	flags for the translator Makefile
pi.hdr	flags for the personalized interpreter Makefile
vt.hdr	flags for the variant translator Makefile
rlocal.c	placeholder for possible assembly-language routines
rover.c	arithmetic overflow checks
rswitch.c	co-expression context switch
Ranlib	library randomizer for personalized interpreters

To work on these files,

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

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

There are many constants in the source code for Icon that vary from system to system. Default values are provided 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.

C preprocessor: Icon preprocessor directives use string concatenation and substitution of arguments within quotation marks. By default, the ‘‘standard’’ method of accomplishing this in UNIX preprocessors is used. A different method is specified in the C draft standard [6]. If your C compiler supports the draft standard, add

```
#define Standard
```

to define.h.

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.

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

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

The values of Big, LogHuge, and Precision give, respectively, the largest floating-point number that does not loose 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. If your system needs a software check for division by floating-point zero, add

```
#define ZeroDivide
```

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 (usually <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/iconx/lmisc.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/iconx/lmisc.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:

Fork	(default: fork)
IconGcvt	(default: undefined)
IconQsort	(default: undefined)
NoAtof	(default: undefined)
SysMem	(default: undefined)
index	(default: undefined)
rindex	(default: undefined)

If your system supports *vfork(2)*, add

```
#define Fork vfork
```

to define.h.

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.

The C run-time routine *atof(3)* normally is used in the Icon linker to convert strings for real literals to corresponding floating-point numbers. If the version of *atof* on your system does not work properly, add

```
#define NoAtof
```

to define.h. This replaces the use of *atof* by in-line conversion code.

If your run-time system includes *memcpy(3)* and *memset(3)*, add

```
#define SysMem
```

to define.h. Otherwise, versions of these routines in the Icon system are used.

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,

```
#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,

```
#define UtsName
```

to define.h.

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

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 [7].

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. They may not be shared or expanded and all of available memory may not be used. This option should be used only if necessary.

The header file: As described In Section 2.1, Step 6, a bootstrap header file is used to make icode files executable. On some systems, particularly UNIX emulators, many routines may be included in the header file by the loader, even if they are not needed.

Start by assuming this is not a problem, but if iconx.hdr is impractically large, you can eliminate the header file 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.

Miscellaneous: There are two other definitions that may be needed in some cases:

Hz	(default: 60)
UpStack	(default: undefined)

If you are running in a 50-hz environment, add

```
#define Hz 50
```

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.

Executable Images: If you have a 4.*n*bsd UNIX system and want to enable the function save(s), which allows an executable image of a running Icon program to be saved [1], add

```
#define ExecImages
```

to define.h.

Optional features: The implementation of co-expressions and arithmetic overflow checking require assembly language routines. Initially, define.h contains

```
#define NoCoexpr
#define NoOver
```

These definitions disable co-expressions and arithmetic overflow checks. Leave these definitions in for the first round, although you may want to remove them later and implement these features (see Steps 7 and 8).

Step 3: Makefile Headers

The files icont.hdr, iconx.hdr, ilink.hdr, and itran.hdr provide headers for Makefiles in the source directories src/icont, src/iconx, and so on. These headers are prepended to the standard bodies for the Makefiles during configuration.

Except for iconx.hdr, these headers serve only to specify flags for *cc(1)* and *ld(1)* via CFLAGS and LDFLAGS. 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.

There are two other definitions in iconx.hdr, RSWITCH and ROVER, which depend on whether the local co-expression context switch and arithmetic overflow checks are written in C or assembly language. The initial values of these definitions are rswitch.c and rover.c, and dummy C routines are provided. To start out, leave these definitions as they are; the default routines can be replaced later. See Steps 7 and 8.

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 itran.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 modifications to the distributed source code for Version 7. 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 central version of the source code at the Icon Project and will be in place for you when subsequent versions are released. See Step 11.

Occasionally, it is necessary to add code that cannot be written in C. The file rlocal.s is provided for such contingencies. It initially is a placeholder that is copied from the configuration file to src/iconx and is compiled and linked when the run-time system is built. This placeholder can be replaced by any necessary assembly-language routines.

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
```

See Makefile for more detailed information about the tests.

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 a co-expression will terminate with an error message.

All aspects of co-expression creation and activation are written in C in Version 7 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, replace the version of this file in your configuration directory. Alternatively, if you implement coswitch in assembly language, place it in a file named 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 number of elements for saving 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)
int *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 coswitch for the VAX. Other examples are contained in the configuration directories in config/unix.

After you implement rswitch, remove the #define NoCoexpr from define.h and replace rswitch.c in your configuration directory by either a new rswitch.c or an assembly language file named rswitch.s. 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 or replace it with something more appropriate.

Step 8: Overflow Checking

C does not provide overflow checking for integer addition, subtraction, or multiplication. Icon, on the other hand, is supposed to check for overflow. This usually requires assembly-language code.

Initially, define.h contains the definition

```
#define NoOver
```

which causes overflow checking to be bypassed.

If you do not want to implement overflow checking, you need do nothing. The only effect will be that overflow will not be detected.

If you want to implement overflow checking, remove the definition of NoOver from your define.h and write routines ckadd, cksub, and ckmul that call fatalerr(-203,0) in the case of overflow. Appendix C contains the overflow checking routine for the VAX. Other examples are contained in the configuration directories in config/unix.

Note: It often is harder to test for overflow for multiplication than for addition and subtraction. A dummy routine that simply returns can be provided for multiplication if this is the case on your system.

If you supply overflow checking routines, put them in the file rover.s in your configuration directory and replace

```
ROVER=rswitch.c
```

by

```
ROVER=rswitch.s
```

in iconx.hdr in your configuration directory. The configuration process then will copy this file to the appropriate place prior to compilation.

To test overflow checking,

```
make Test–over
```

There should be no differences in the comparison of outputs if overflow checking is working properly.

You should also rerun previous tests at this point to make sure that arithmetic still works properly.

Step 9: 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. If the test programs fail to execute, suspect the size of tests/pi/piconx.hdr, the personalized interpreter version of iconx.hdr. If it is larger than MaxHdr, something is wrong. Check the file pi.hdr in your configuration directory as described in Section 2.1.

Step 10: 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 the status files in other configuration directories as models.

Step 11: 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.

In turn, your changes will be added to the central Icon system and will be available to others (as well as to you in future releases of Icon).

Files can be sent on any convenient media, such as 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 7 of Icon, contact the Icon Project:

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

(602) 621-6613

icon–project@arizona.edu
{ihnp4,noao,mcnc,allegra}!arizona!icon-project

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. Griswold, Ralph E., Gregg M. Townsend, and Kenneth Walker, *Version 7 of Icon*, Technical Report TR 88-5, Department of Computer Science, The University of Arizona. 1988.

2. Griswold, Ralph E. *Transporting Version 7 of Icon*, Technical Report TR 88-9, Department of Computer Science, The University of Arizona. 1988.

3. Griswold, Ralph E. *The Icon Program Library; Version 6, Release 1*, Technical Report TR 86-13, Department of Computer Science, The University of Arizona. 1986.

4. Griswold, Ralph E. *Personalized Interpreters for Version 7 of Icon*, Technical Report TR 88-7, Department of Computer Science, The University of Arizona. 1988.

5. Griswold, Ralph E. and Kenneth Walker. *Building Variant Translators for Version 7 of Icon*, Technical Report TR 88-8, Department of Computer Science, The University of Arizona. 1988.

6. Technical Committee X3J11, *Draft Proposed American National Standard for Information Systems — Programming Language C*. 1986.

7. Griswold, Ralph E. *ICONT(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. 1988.

Appendix A — The Distribution Hierarchy

The main directories in the Version 7 hierarchy are:

v7		root of the Version 7 hierarchy
	/bin	standard location for binary files
	/book	programs from the Icon book
	/config/unix	configuration files for UNIX systems
	/docs	source text for documents
	/ipl	Icon program library
	/pi	personalized interpreter system
	/samples	sample programs
	/src	source code for the Icon system
	/tests	test programs
	/vt	variant translator system

A complete directory tree is contained in docs/v7.dtree.

Appendix B — A Sample Co-Expression Context Switch

rswitch.c for the VAX under Berkeley UNIX:

```
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");
        }

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

Appendix C — A Sample Arithmetic Overflow Checking Routine

rover.s for the VAX under Berkeley UNIX:

```
_ckadd:  .word   0
         addl3   4(ap),8(ap),r0     # Perform addition
         jvs     oflow              # Branch if overflow
         ret                        # Return result in r0

_cksub:  .word   0
         subl3   8(ap),4(ap),r0     # Perform subtraction
         jvs     oflow              # Branch if overflow
         ret                        # Return result in r0

_ckmul:  .word   0
         mull3   4(ap),8(ap),r0     # Perform multiplication
         jvs     oflow              # Branch if overflow
         ret                        # Return result in r0

oflow:                              # Got overflow on an operation
         pushl   $0
         pushl   $–203
         calls   $1,_fatalerr       # fatalerr(–203,0)
```