Version 9 of Icon for MS-DOS

Ralph E. Griswold

Department of Computer Science
The University of Arizona
Tucson, Arizona

IPD247c
March 10, 1996
http://www.cs.arizona.edu/icon/docs/ipd247.html

---

1. Introduction

Version 9 of Icon for MS-DOS should run on any PC or PC clone running MS-DOS 3.0 or higher. A math co-processor is supported and used if present; otherwise software emulation is used. Approximate 500K of free RAM is needed to run Icon satisfactorily. Two versions of Icon's executor system are provided. One supports arithmetic on integers of unlimited magnitude and the other does not. The latter is considerably smaller than the former and can be used if you do not have enough RAM for the former.

This implementation of Icon is in the public domain and may be copied and used without restriction. The Icon Project makes no warranties of any kind as to the correctness of this material or its suitability for any application. The responsibility for the use of Icon lies entirely with the user.

The basic reference for Version 8 of Icon is the second edition of the book The Icon Programming Language [1]. This book is available from the Icon Project at The University of Arizona.

The new features of Version 9 of Icon are described in an accompanying technical report [2].

2. Installing MS-DOS Icon

Three executable binary files are needed to use Icon:

```
icont.exe   translator
iconx.exe   executor
ixhdr.exe   bootstrap header
```

These files should be located at a place on your `PATH` specification. In addition, `icont.exe` and `ixhdr.exe` must be in the same directory.

There are two forms of `icont` and `iconx`: ones without large-integer arithmetic and ones with it. The former are named `icont.exe` and `iconx.exe` as distributed, while the latter are named `icontl.exe` and `iconxl.exe`. The distribution is contained in several files in LHA format. These files have the extension `lzh`. If you do not have a copy of `lha.exe`, execute the self-extracting archive `lha213.exe` on the distribution diskette. This will produce `lha.exe` and documentation.

The distribution files are:

```
docs.lzh     documents
icon.lzh     executable binary files
readme       installation overview and recent notes
samples.lzh  Icon programs and data
```

To install the `.exe` files, set your current directory to the desired place, place the appropriate distribution diskette in a drive, and dearchive the files there using `lha.exe`. For example, using drive `A` to dearchive the executable binary files, the following will do:

```
a:lha x a:icon.lzh
```

The same technique can be used for extracting the remaining files.

Pick the translator and executor that fit your needs, place them at a location on your path, and rename them to `icont.exe` and `iconx.exe` if necessary. For example, if you want the executor that supports large-integer arithmetic, the following will do:

```
rename icontl.exe icont.exe
rename iconxl.exe iconx.exe
```

3. Running MS-DOS Icon -- Basic Information

Files containing Icon programs must have the extension `.icn`. Such files should be plain text files (without line numbers or other extraneous information).

The Icon translator, `icont`, produces an "icode" file that is executed using `iconx.exe`. There are two forms of icode files:

1. Executable icode files that invoke `iconx.exe` automatically when they are run. These icode files have the extension `exe`.

2. Non-executable icode files that are run by `iconx` with the icode file as an argument. These icode files have the extension `icx`.

Non-executable icode files require about 8K less RAM to run than executable files, but they are more awkward to run. Note: Executable icode files do not stand alone; they require `iconx.exe` to run.

Producing Executable Icode Files

Executable icode files are produced by default. An Icon program in the file `prog.icn` is translated by

```
icont prog.icn
```

If your translator is named differently, simply use that name. For example, if your translator is named `icontl.exe`, use

```
icontl prog.icn
```

The result is an icode file with the name `prog.exe`. This file can be run by

```
prog
```

Alternatively, `icont` can be instructed to execute the icode file after translation by appending a `-x` to the command line, as in

```
icont prog.icn -x
```

This only works if your executor is named `iconx.exe`, since the `-x` option looks for this name. In the sections that follow, it is assumed that the executor is named `iconx.exe`.

If `icont` is run with the `-x` option, the file `prog.exe` is left and can be run subsequently using an explicitly named executor as described above.

The extension `.icn` is optional on the command line. For example, it is sufficient to use

```
icont prog
```

Producing Non-Executable Icode Files

To produce a non-executable icode file, use the option `-I`, as in

```
icont -I prog
```

The result is an icode file named `prog.icx`. It can be run by

```
iconx prog
```

Note that the `icx` extension is not necessary.

`iconx` will find an icode file if it is in the current directory or at place given on your `PATH` specification.

Except for the icode file extension and the method of running non-executable icode files, the remarks in the previous section apply.

4. Testing the Installation

There are a few programs on the distribution diskette that can be used for testing the installation and getting a feel for running Icon:

`hello.icn`
: This program prints the Icon version number, time, and date. Run this test as

```
icont hello
hello
```

Note that this can be done in one step with

```
icont hello -x
```

`cross.icn`
: This program prints all the ways that two words intersect in a common character. The file `cross.dat` contains typical data. Run this test as

```
icont cross -x <cross.dat
```

`meander.icn`
: This program prints the "meandering strings" that contain all subsequences of a specified length from a given set of characters. Run this test as

```
icont meander -x <meander.dat
```

`roman.icn`
: This program converts Arabic numerals to Roman numerals. Run this test as

```
icont roman -x
```

and provide some Arabic numbers from your console.

If these tests work, your installation is probably correct and you should have a running version of Icon.

5. More on Running Icon

For simple applications, the instructions for running Icon given in Section 3 may be adequate. The `icont` translator supports a variety of options that may be useful in special situations. There also are several aspects of execution that can be controlled with environment variables. These are listed here. If you are new to Icon, you may wish to skip this section on the first reading but come back to it if you find the need for more control over the translation and execution of Icon programs.

5.1 Arguments

Arguments can be passed to the Icon program by appending them to the command line. Such arguments are passed to the main procedure as a list of strings. For example,

```
iconx prog text.dat log.dat
```

runs the icode file `prog.icx`, passing its main procedure a list of two strings, `"text.dat"` and `"log.dat"`. The program also can be translated and run with these arguments with a single command line by putting the arguments after the `-x`:

```
icont prog -x text.dat log.dat
```

These arguments might be the names of files that `prog.icn` reads. For example, the main procedure might begin as follows:

```
procedure main(args)
in := open(args[1]) | stop("cannot open file")
out := open(args[2]) | stop("cannot open file")
          .
          .
          .
```

5.2 The Translator

The `icont` translator can accept several Icon source files at one time. When several files are given, they are translated and combined into a single icode file whose name is derived from the name of the first file. For example,

```
icont prog1 prog2
```

translates the files `prog1.icn` and `prog2.icn` and produces one icode file, `prog1.exe`.

A name other than the default one for the icode file produced by icont can be specified by using the `-o` option, followed by the desired name. For example,

```
icont -o probe prog
```

produces the icode file named `probe.exe` rather than `prog.exe`.

If the `-c` option is given to `icont`, the translator stops before producing an icode file and intermediate "ucode" files with the extensions left for future use (normally they are deleted). For example,

```
icont -c prog1
```

leaves `prog1.u1` and `prog1.u2`, instead of producing `prog1.exe`. These ucode files can be used in a subsequent `icont` command by using the `.u1` name. This saves translation time subsequently. For example,

```
icont prog2 prog1.u1
```

translates `prog2.icn` and combines the result with the ucode files from a previous translation of `prog1.icn`. Note that only the `.u1` name is given; the `.u2` name is implied. The extension can be abbreviated to `.u`, as in

```
icont prog2 prog1.u
```

Ucode files also can be added to a program using a link declaration.

Icon source programs may be read from standard input. The argument `-` signifies the use of standard input as a source file. In this case, the ucode files are named `stdin.u1` and `stdin.u2` and the icode file is named `stdin.exe`.

The informative messages from the translator can be suppressed by using the `-s` option. Normally, both informative messages and error messages are sent to standard error output.

The `-t` option causes `&trace` to have an initial value of -1 when the icode file is executed. Normally, `&trace` has an initial value of 0.

The option `-u` causes warning messages to be issued for undeclared identifiers in the program.

5.3 Environment Variables

When an icode file is executed, several environment variables are examined to determine execution parameters. The values assigned to these variables should be numbers.

Environment variables are particularly useful in adjusting Icon's storage requirements. Particular care should be taken when changing default values: unreasonable values may cause Icon to malfunction.

The following environment variables can be set to adjust Icon's execution parameters. Their default values are listed in parentheses after the environment variable name:

`TRACE` (undefined)
: This variable initializes the value of `&trace`. If this variable has a value, it overrides the translation-time `-t` option.

`NOERRBUF` (undefined)
: If this variable is set, `&errout` is not buffered.

`STRSIZE` (65000)
: This variable determines the size, in bytes, of the initial region in which strings are stored. If additional string regions are needed, they may be smaller.

`BLKSIZE` (65000)
: This variable determines the size, in bytes, of the initial region in which lists, tables, and other objects are stored. If additional block regions are needed, they may be smaller.

`COEXPSIZE` (2000)
: This variable determines the size, in 32-bit words, of each co-expression block.

`MSTKSIZE` (10000)

This variable determines the size, in words, of the main interpreter stack.

`QLSIZE` (5000)
This variable determines the size, in bytes, of the region used by the garbage collector for pointers to strings.

The maximum region size that can be specified is 65000.

6. Features of MS-DOS Icon

MS-DOS Icon supports all the features of Version 9 of Icon, with the following exceptions and additions:

- Pipes are not supported. A file cannot be opened with the `"p"` option.
- For files opened in the translate mode, the position produced by `seek()` may not reflect the actual byte position because of the translation of carriage-return/line-feed sequences to line-feed characters.
- Path specifications can be entered using either a `/` or a `\`. Examples are:

```
A:\ICON\TEST.ICN
A:/ICON/TEST.ICN
```

- The following MS-DOS device names can be used as file names:

```
console          CON
printer          PRN LST LPT LPT1
auxiliary port   AUX COM RDR PUN
null             NUL NULL
```

  For example,

```
prompt := open("CON", "w")
```

  causes strings written to prompt to be displayed on the console. Use of a null file name means no file is created.

- MS-DOS Icon also has some functions in addition to the standard repertoire. These are described in the next section.

7. MS-DOS Functions

Disclaimer: The following functions provide a gateway to facilities provided by MS-DOS and ROM BIOS. These functions should be used with care, since Icon maintains strict control over its environment (although it uses standard MS-DOS interfaces and does not bypass MS-DOS or ROM BIOS in any way). The descriptions that follow are low-level descriptions. They assume knowledge of the Intel 8086 (8088, 80x86) architecture.

`Int86(L)` generates a hardware interrupt. The input is a list of integer values: `[interrupt number, ax, bx, cx, dx, si, di, es, ds]`. It returns a list of the form `[flags, ax, bx, cx, dx, si, di, es, ds]`. Great care must be taken in using this function. Some things to watch out for are:

- Interrupt functions that alter the stack registers (SS or SP) or alter the MS-DOS memory chain (that is, allocate or deallocate memory in the MS-DOS memory region). When Icon expands memory, it expects the next allocation to be contiguous with the region it currently owns.
- Interrupt functions that operate on files opened by Icon's `open(s)` function -- that is, closing, seeking, reading, or writing.

- The values of ES and DS may not be valid on return.

`InPort(i)` returns an integer from hardware port `i`.

`OutPort(i1, i2)` writes value `i2` to hardware port `i1`.

`GetSpace(i)` allocates a static block of storage outside of Icon's direct control (that is, it is not be affected by garbage collection). This function simply calls the *malloc()* allocation routine and returns the address of the resulting block as a long integer.

`FreeSpace(A)` frees a static block of storage, where `A` is a value returned by `GetSpace(i)`. No check is make to verify that the block was allocated by `GetSpace(i)`. The results are unpredictable if it was not. Arithmetic should not be performed directly on a value returned by `GetSpace(i)`.

`Peek(A, i)` builds a string pointing to the address specified by `A` with a length of `i`, where `A` is either an integer that specifies a linear address value or a list of the form `[segment, offset]` and `i` is a length (default 1). This is the only way of "seeing" the contents of a block of storage allocated by `GetSpace(i)`. Consider the following example:

```
block := Peek(addr := GetSpace(100),100)
```

The value of `block` is a string of length 100 and `addr` is the linear address of the block.
`Peek(A, i)` does not move data into Icon's memory region. Instead, it builds a string qualifier that points to the data.

`Poke(A, s)` copies a string `s` to location `A`, where `A` is an address specified in the same way as for `Peek(A, s)` and `s` is a string to be copied to that storage location. The string is copied directly into storage, byte by byte. No conversion is performed. This is the only way of assigning data to a block of storage allocated by `GetSpace(i)`.

8. Known Bugs

The `-x` option is not supported by some versions of MS-DOS Icon.

MS-DOS memory management sometimes causes problems. Some programs that require a lot of memory may abort or hang when run by the `-x` option to `icont`. This can be avoided running the icode file as a separate step. Also, if there is not enough free RAM, the `system()` function may fail silently or hang.

9. Reporting Problems

Problems with Icon should be noted on a trouble report form (included with the distribution) and sent to:

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

(520) 621-6613 (voice)

(520) 621-4246 (fax)

icon-project@cs.arizona.edu

Acknowledgements

Clint Jeffery and Gregg Townsend collaborated with the author in the development of Version 9. Will Mengarini provided the mechanism for producing executable icode files.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.

2. R. E. Griswold, C. L. Jeffery and G. M. Townsend, *Version 9.1 of the Icon Programming Language*, The Univ. of Arizona Icon Project Document IPD267, 1995.

Icon home page