

icc Command

Related tools: ld, qar, del

Overview

When V4R3 rolled around and the ILE-C and a native C++ compiler came along that would process source statements in stream files I thought “great now I can port all that free Unix code to the AS/400 without worrying about moving everything to libraries and converting make files to CL programs, and etc.” Of course when I tried it there were still a lot of problems to overcome so I decided to do something about it. The result of that is a set of four QSHHELL utilities that will let me take a Unix make file and it’s source, put them in stream files on the AS/400 and build a program the same way I would on Unix.

In a Unix environment I would normally do something like this:

```
cc -g -o BigHelloFromRochester aBigHelloFromRochester.c
```

This would compile a source file from the current directory call aBigHelloFromRochester.c and create an executable file called BigHelloFromRochester in the current directory. If you were to try the same command on the AS/400 from the QSHHELL environment on V4R3 or later the command would be able to process the source file but because of the long name it wouldn’t know how to create the module or program objects.

To ameliorate this situation I created a new shell utility called icc. It would be used like this from the QSHHELL:

```
icc -o BigHelloFromRochester aBigHelloFromRochester.c
```

Assume that the file aBigHelloFromRochester.c is in a directory called test and that there is an AS/400 library called test. Then, when this command is run the following happens (at least this happens the first time it is run, we’ll go over what happens on subsequent invocations next):

1. icc looks for a symbolic link called aBigHelloFromRochester.o, assume it doesn’t find one
2. A library/module name is generated.
3. A CRTCMOD command is issued. In this case it looks like this: CRTCMOD
MODULE(TEST/ABIGHELLOF) SRCSTMF('aBigHelloFromRochester.c') DBGVIEW(*ALL)
TEXT('/test/aBigHelloFromRochester.o')
4. Since the compile was successful a symbolic link is created from aBigHelloFromRochester.o in the current directory to the object /QSYS.LIB/TEST.LIB/aBigHelloF.MODULE.
5. icc looks for symbolic link called BigHelloFromRochester, (assume it is not found)
6. A library/program name is generated.

7. A CRTPGM command is issued using the library/program name from step 6: CRTPGM PGM(TEST/BIGHELLOFR) MODULE(TEST/ABIGHELLOF) TEXT('/test/BigHelloFromRochester')
8. Since the CRTPGM was successful a symbolic link is created from BigHelloFromRochester in the current directory to the object /QSYS.LIB/TEST.LIB/BigHelloFr.PGM. In this case the module object is also deleted, since an executable program is the result.
9. I can then invoke the program from the qshell by typing BigHelloFromRochester. (Alternatively I can call it from an OS/400 command line with CALL TEST/BIGHELLOFR

The next time I ran the command the symbolic links that exist are found and the steps to generate the names are not required.

Using Symbolic Links

So the real magic here is to use symbolic links from the root file system (or any other file system that supports stream files) in IFS to objects in the /QSYS.LIB file system. This will let you use the long file names and extensions with the icc utility and have it translate to the name that the OS/400 command expects. “Okay,” you ask, “but where do the library and object names come from?” The names can come from different sources:

10. You can create the links yourself using the ln utility in QSHELL. This method gives you the most control over naming of the objects.
11. You can use the default mapping provided by the QCCMAPOUT service program as described a little later.
12. You can write your own mapping algorithms and replace the QCCMAPOUT service program with your own, also described later in this section.

The default mapping service program QCCMAPOUT uses a very simple mapping algorithm. It first checks for the environment variable OUTPUTDIR to get the library name to use. If the environment variable OUTPUTDIR isn't set, then it will use up to the first 10 characters of the directory name of the immediate parent of the file passed. For example if compiling a file with the following path /proj1/src/applicationServer/serverInitializer.c the library name used for the module object would be application. The name of the module object will also be based on the first 10 characters of the file name, but if the name already exists then the first 9 characters of the name will be appended with the characters 0-9 in sequence until a unique name is found. If none is found then the algorithm gives up. So if a directory contains two source files one called serverInitializer.c and the called serverInitInstance.c then the object names serverInit and serverIni0 will be used.

If this name mapping algorithm doesn't suit your purposes then you are encouraged to write your own. The default source is available in a file called qccmap.c and it should be pretty straightforward to write one that is more appropriate to your needs. The source is available upon request by sending a note to rchqptl@us.ibm.com.

Usage:

icc [option | inputfile]...

Description:

The icc command compiles C and C++ source files, producing module objects and a symbolic link to the module object. It can also be used to create program objects.

Unless the -c option is specified, icc calls the CRTPGM command to produce a single program object.

Input files may be any of the following:

1. file name with .C (upper case only) suffix: C++ source file
2. file name with .CPP or cpp suffix: C++ source file
3. file name with .CC or cc suffix: C++ source file
4. Any other file suffix: c source file
5. file name with .o suffix: sym link to module object or a service program
6. File name with .a suffix: sym link to binding directory

Options:

Options can be flag options or keyword options:

1. Flag options:

- + Treat source files as C++ source code regardless of extension
- c Do not do a CRTPGM. Only compile and produce module object.
- D<name>[=<def>]
Define <name> as in #define directive. If <def> is not specified, 1 is assumed.
- g Produce information for the debugger.
- I<dir> Search in directory <dir> for include files that do not start with an absolute path.
- l<key> Search the specified library file (symbolic link to bnmdir object), where <key> selects the file lib<key>.a.
- L<dir> Search in directory <dir> for files specified by -l<key>.
- M<name> Generate information to be included in a "make" description file; output goes to named file.
- o<name> Name the executable file <name> instead of a.out.
When used with the -c option and one source file, name the object file <name> instead of filename.o.
- O<lvl> Optimize generated code. Level can be 1,2,3,4 corresponding to levels 10, 20, 30 and 40.
If <lvl> not specified default is 4.

- P<name> Specifies the path name for a file to receive the listing produced by the C++ compiler.
- v Displays the OS/400 command to be executed; output goes to stdout.
- z <none,all,noifsio,ifsio,ifs64io> The ifsio option to use with this compile.
Default is ifsio.

2. Keyword options:

Keyword options are specified in one of the following ways:

-q<option>

-q<option>=<parameter>

where <option> is an option name and <parameter> is a parameter value.

Keyword options with no parameters represent switches that may be either on or off. The keyword by itself turns the switch on, and the keyword preceded by the letters NO turns the switch off. For example, -qGEN tells the compiler to produce a module and -qNOGEN tells the compiler not to produce a module. If an option that represents a switch is set more than once, the compiler uses the last setting.

Keyword option and parameter names may appear in either UPPER CASE or lower case letters in the icc command.

C source options without parameters:

*NOAGR

*AGR

*NOEXPMAC

*EXPMAC

*GEN

*NOGEN

*LOGMSG

*NOLOGMSG

*NOPPONLY

*PPONLY

*NOSECLVL

*SECLVL

*NOSHOWINC

*SHOWINC

*SHOWUSR
*SHOWSYS
*NOSHOWSKP
*SHOWSKP
*NOXREF
*XREF
*USRINCPATH
*SYSINCPATH

Options with parameters:

OUTPUT *NONE - No listing is produced
*PRINT - Produce a spooled file listing

CHECKOUT

Checkout options

*NONE	*NOINIT
*USAGE	*INIT
*ALL	*NOPARM
*NOACCURACY	*PARM
*ACCURACY	*NOPORT
*NOENUM	*PORT
*ENUM	*NOPPCHECK
*NOEXTERN	*PPCHECK
*EXTERN	*NOPPTRACE
*NOGENERAL	*PPTRACE
*GENERAL	
*NOGOTO	
*GOTO	
*GENERAL	
*NOGOTO	
*GOTO	

INLINE "*ON *AUTO 250 2000 *YES"
*OFF *NOAUTO 250 2000 *NO"

MODCRTOPT *NOKEEPILDTA
*KEEPILDTA

LANGLVL *SOURCE
*EXTENDED
*SAAL2

*SAA
 *ANSI

SYSIFCOPT *NONE
 *ALL
 *NOIFSIO
 *IFSIO

LOCALETYPE *CLD
 *LOCALE

MARGINS "<left margin> <right margin>"
 SEQCOL *NONE or "<left> <right>"
 FLAG 0 -all
 10 - error and warnings
 30 - error only

MSGLMT "*NOMAX 30"

REPLACE *YES
 *NO

AUT *LIBCRTAUT
 *ALL
 *CHANGE
 *USE
 *EXCLUDE

TGTRLS *CURRENT
 *PRV
 VxRxMx

ENBPFRCOL *PEP
 "*ENTRYEXIT *NOLEAF"
 "*ENTRYEXIT *ALLPRC"
 "*FULL *NOLEAF"
 "*FULL *ALLPRC"
 *NONE

PFROPT *SETFPCA
 *NOSETFPCA
 *NOSTRDONLY
 *STRDONLY

PRFDTA *NOCOL
 *COL

ld Utility

Related tools: icc, qar, del

usage: ld -o name [-svx] [-q value]... [-L directory] [-l<key>] operand ...

The ld utility is used to create shared libraries (service programs).

Arguments

operands The name of the object files(modules) and shared libraries (service programs) to be part of the new shared library(service program).
a “.a” extension

Options:

-o<name> Name the service program <name>. The name of the file must have a .o extension. If the name does not exist a library/object name is generated for the service program and a symbolic link for <name> is created.

 Otherwise the library/object linked by <name> is used.

-s Remove observability of the service program.

-v View the native command executed by the ld utility.

-x Use export file.

 Note: You can use the -q option to specify the export source file (SRCFILE) and member (SRCMBR).

-l<key> Search the specified library file (symbolic link to bnmdir object), where <key> selects the file lib<key>.a.

-L<dir> Search in directory <dir> for files specified by -l<key>.

Keyword options are specified in one of the following ways:

-q<option>
-q<option>=<parameter>
 where <option> is an option name and <parameter> is a parameter value.

Keyword options with no parameters represent switches that may be either on or off. The keyword by itself turns the switch on, and the keyword preceded by the letters NO turns the switch off. For example, -qLIST tells the compiler to produce a listing and -qNOLIST tells the compiler not to produce a listing. If an option that represents a switch is set more than once, the compiler

uses the last setting.

Keyword option and parameter names may appear in either UPPER CASE or lower case letters in the ld command.

qar Utility

Related tools: `icc`, `ld`, `del`

usage: `qar [-cduv] Archive File...`

The qar utility is used to create binding directories. Binding directories can be used somewhat like libraries are used on Unix systems. The big difference is that libraries created with `ar` on Unix are archive files that contain an actual copy of the module to be used. With a binding directory only a reference to a module object or a service program is actually stored in the binding directory.

Arguments

Archive File	Name of the archive file to be used. This should be specified with a “.a” extension
input files	Must be files with the extension of .o, these file can be symbolic links to module objects or other service programs.

Options:

-c	Create a new binding directory if one does not exist. It is not a error to specify -c if a <code>bnddir</code> already exists.
-d	Delete the specified files from the archive.
-u	Add the specified files to the archive. I the file is already in the archive no error is generated.
-v	View the native command executed by the qar utility.

Example 1:

In this example assume that we have the following situation. There are 3 .c files, `serviceProgramMod1.c`, `serviceProgramMod2.c` and `serviceProgramMod3.c`, that will be compiled and linked into a shared library (service program) called `shared.o`. Two other files `l1.c` and `l2.c` are used to build a library archive (`bnddir`) called `lib1.a`. Finally `shared.o` `lib1.a` and 3 additional files: `verylongprogramname.c`, `mainprog2.c` and `mainprog3.c` are used to build a program to be called `myprog`.

at the start:

```
[1]> pwd
/src
$
[2]> ls -l *
/src/lib:
 11.c
 12.c
/src/prog:
mainprog2.c
mainprog3.c
verylongprogramname.c
/src/shared:
serviceProgramModule1.c
serviceProgramModule2.c
serviceProgramModule3.c
```

```
[3]$
>cd shared
$
>pwd
/src/shared
$
[4]> system crtlib shared
CPC2102: Library SHARED created.
[5]> icc -cgv *.c
```

¹ Assume that at the start the current directory is /src.

² The sub directories contain only the .c files.

³ First start with the files in the shared directory.

⁴ An AS/400 library is required to hold the objects created as a result of compiling the .c files. I used the convention of naming the libraries the same as the immediate parent.

⁵ In step 6 I used the icc command to compile all of the .c files in the current directory. the -c option tells the icc command that it should not create a program object, but that it should only create module objects.

The -g option indicates that debuggable modules should be produced. The -v option is used to print a copy of the command that is executed.

```
command = CRTCMOD MODULE(shared/servicePro) SRCSTMF('serviceProgramModule1.c')
DBGVIEW(*ALL) TEXT('/src/shared/serviceProgramModule1.o') OPTION(*LOGMSG)
command = CRTCMOD MODULE(shared/servicePr0) SRCSTMF('serviceProgramModule2.c')
DBGVIEW(*ALL) TEXT('/src/shared/serviceProgramModule2.o') OPTION(*LOGMSG)
command = CRTCMOD MODULE(shared/servicePr1) SRCSTMF('serviceProgramModule3.c')
DBGVIEW(*ALL) TEXT('/src/shared/serviceProgramModule3.o') OPTION(*LOGMSG)
$
```

```
6> ls
```

```
serviceProgramModule1.c
serviceProgramModule1.o
serviceProgramModule2.c
serviceProgramModule2.o
serviceProgramModule3.c
serviceProgramModule3.o
```

```
$
```

```
8> ld -v -o shared.o *.o
```

```
command = CRTSRVPGM SRVPGM(SHARED/shared) MODULE(SHARED/SERVICEPRO
SHARED/SERVICEPR0 SHARED/SERVICEPR1 ) EXPORT( *ALL) TEXT('/src/shared/shared.o')
```

```
$
```

```
9> ls
```

```
serviceProgramModule1.c
serviceProgramModule1.o
serviceProgramModule2.c
serviceProgramModule2.o
serviceProgramModule3.c
serviceProgramModule3.o
shared.o
```

```
$
```

```
10> cd ../lib
```

```
$
```

```
11> ls -l
```

```
11.c
12.c
```

```
$
```

```
12> export -s OUTPUTDIR=shared
```

```
$
```

```
13> icc -cgv *.c
```

⁶After the commands completes the directories now contains the .o files. Note that these files are symbolic links to the module objects just created.

```
command = CRTCMOD MODULE(shared/l1) SRCSTMF('11.c') DBGVIEW(*ALL)
TEXT('/src/lib/11.o') OPTION(*LOGMSG)
command = CRTCMOD MODULE(shared/l2) SRCSTMF('12.c') DBGVIEW(*ALL)
TEXT('/src/lib/12.o') OPTION(*LOGMSG)
$
14 > qar -cuv libchk.a 11.o 12.o
command = CRTBNDDIR BNDDIR(shared/libchk) TEXT('/src/lib/libchk.a')
command = ADDBNDDIRE BNDDIR(shared/libchk) OBJ((SHARED/L1 *MODULE)
(SHARED/L2 *MODULE) )
$
15> ls
libchk.a
11.c
11.o
12.c
12.o
$
16> cd ../prog
$
17> unset OUTPUTDIR
$
18> system crtlib prog
CPC2102: Library PROG created.
$
19> icc -cgv -o mainProg2.o mainProg2.c
command = CRTCMOD MODULE(prog/mainProg2) SRCSTMF('mainProg2.c')
DBGVIEW(*ALL) TEXT('/src/prog/mainProg2.o') OPTION(*LOGMSG)
$
20> icc -cgv -o mainProg3.o mainProg3.c
command = CRTCMOD MODULE(prog/mainProg3) SRCSTMF('mainProg3.c')
DBGVIEW(*ALL) TEXT('/src/prog/mainProg3.o') OPTION(*LOGMSG)
$
21> icc -cgv verylongprogramname.c
command = CRTCMOD MODULE(prog/verylongpr) SRCSTMF('verylongprogramname.c')
DBGVIEW(*ALL) TEXT('/src/prog/verylongprogramname.o') OPTION(*LOGMSG)
$
22> ls
mainprog2.c
mainprog3.c
mainProg2.o
mainProg3.o
verylongprogramname.c
verylongprogramname.o
```

```

$
23> icc -v -o prog -L../lib -lchk *.o ../shared/shared.o
command = CRTPGM PGM(prog/prog) MODULE(PROG/MAINPROG2 PROG/MAINPROG3
PROG/VERYLONGPR ) BNDSRVPGM(SHARED/SHARED ) BNDDIR(SHARED/LIB1)
TEXT('/src/prog/prog')
$
24> ls
  mainprog2.c
  mainprog3.c
  mainProg2.o
  mainProg3.o
  prog
  verylongprogramname.c
  verylongprogramname.o
$
25> prog
Hello from ServiceProgramModule1!
Hello from ServiceProgramModule2!
Hello from ServiceProgramModule3!
Hello from L1!
Hello from L2!
Hello from mainprog2!
Hello from mainProg3!
Good-bye from verylongprogramname!
$
>cd /src
$
26> ls -l -l *
/src/lib:
lrwxrwxrwx 1 JDEIKEN 0 32 Mar 29 16:03 libchk.a -> /qsys.lib/shared.lib/lib1.bnddir
-rwxrwxrwx 1 JDEIKEN 0 76 Mar 29 16:26 l1.c
lrwxrwxrwx 1 JDEIKEN 0 30 Mar 29 16:01 l1.o -> /qsys.lib/shared.lib/l1.module
-rwxrwxrwx 1 JDEIKEN 0 76 Mar 29 16:26 l2.c
lrwxrwxrwx 1 JDEIKEN 0 30 Mar 29 16:01 l2.o -> /qsys.lib/shared.lib/l2.module

/src/prog:
-rwxrwxrwx 1 JDEIKEN 0 86 Mar 29 16:24 mainprog2.c
-rwxrwxrwx 1 JDEIKEN 0 86 Mar 29 16:25 mainprog3.c
lrwxrwxrwx 1 JDEIKEN 0 35 Mar 29 16:09 mainProg2.o -> /qsys.lib/prog.lib/mainProg2.module
lrwxrwxrwx 1 JDEIKEN 0 35 Mar 29 16:09 mainProg3.o -> /qsys.lib/prog.lib/mainProg3.module
lrwxrwxrwx 1 JDEIKEN 0 27 Mar 29 16:20 prog -> /qsys.lib/prog.lib/prog.pgm
-rwxrwxrwx 1 JDEIKEN 0 191 Mar 29 16:22 verylongprogramname.c
lrwxrwxrwx 1 JDEIKEN 0 36 Mar 29 16:11 verylongprogramname.o -> /qsys.lib/prog.lib/verylongpr.module

/src/shared:
-rwxrwxrwx 1 JDEIKEN 0 95 Mar 29 16:25 serviceProgramModule1.c

```

```
lrwxrwxrwx 1 JDEIKEN 0 38 Mar 29 15:55 serviceProgramModule1.o -> /qsys.lib/shared.lib/servicePro.module
-rwxrwxrwx 1 JDEIKEN 0 95 Mar 29 16:25 serviceProgramModule2.c
lrwxrwxrwx 1 JDEIKEN 0 38 Mar 29 15:55 serviceProgramModule2.o->/qsys.lib/shared.lib/servicePr0.module
-rwxrwxrwx 1 JDEIKEN 0 95 Mar 29 16:25 serviceProgramModule3.c
lrwxrwxrwx 1 JDEIKEN 0 38 Mar 29 15:55 serviceProgramModule3.o -> /qsys.lib/shared.lib/servicePr1.module
lrwxrwxrwx 1 JDEIKEN 0 34 Mar 29 15:57 shared.o -> /qsys.lib/SHARED.lib/shared.srvpgm
$
```

del Utility

Related tools: `icc`, `ld`, `qar`

The `del` utility can be used in place of the `rm` utility to remove a symbolic link and the underlying AS/400 object.

usage: `del [-fv] operand...`

options:

`-f`

If the file does not exist, do not display a diagnostic message or modify the exit status to reflect an error.

`-v`

Display the AS/400 dlt commands actually used.

If you don't use this command and use `rm` in your make files instead of `del`, then only the symbolic link is removed and the underlying object is left alone. For example if you create a `.o` file called `test.o` like this:

```
$
icc -c test.c

$
ls -l test.o
0 33 Sep 23 16:09 /jdeiken/lowlvl/test.o -> /qsys.lib/jdeiken.lib/test.module
$
rm test.o
$
icc -c test.c
$
ls -l test.o
0 34 Sep 23 16:10 /jdeiken/lowlvl/test.o -> /qsys.lib/jdeiken.lib/test0.module
```

Note that the second time `icc` is invoked `test.o` is linked to `test0.module`. This is because `icc` will generate a new name if the chosen name already exists in the library.