CHAPTER 11

Make

Introduction

A typical program consists of object files from multiple source files. It is tedious and error-prone to manually compile and link the files togcthcr. and especially to remember which files were changed. Also. if a header file changes, then you will have to recompile all source files that include the header tile. By using imake and makefile, you lct imake handles all this details for you. Imake is a make utility for managing dependencies between a set of files. You create a description file that describes the dcpcndcncies between the set of files in a program, and invoke imake to compile files that have changed or compile files that include a header file that has changed since the last time the program was built.

The IDE's Project Builder makes this even easier by generating the makefile' and dependencies for you automatically. This chapter is only for those who want to write their own makefiles, or those who are not using the IDE.

Imake reads an input file containing a listing of dependencies between files and associated rules to maintain the dependencies. The format is generally a target file name, followed by a list of files that it is dependent upon, followed by a set of commands to be used to recreate the target from the dependents. Each dependent is in its own right a target, and so the maintenance of each dependent is performed recursively, before attempting to maintain the current target. If after processing its all of its dependencies, a target he is found either to be missing, or to be older than any of its dependency files. imakc uses the supplied commands or an implicit rule to rebuild it.

The input file defaults to "makefile," but you may override it with a command line option -f <filename>.

If no target is specified on the command line. imake uses the first target defined in makcfilc.

Examples of Using imake

You may use the file sample. mak in the examples directory as a starting point for writing your own makefile. It looks something like this

```
# you can create the icc1I.opt file by using the IDE
# Option->Compiler->SaveToDisk command. Its contents
# look something like this:
#
#CFLAGS = -c -Id:\icc 11\include
#LFLAGS = -btext:Ox1800   -lfp -Ld:\icc 11\lib
#.SUFFIXES:    .c .o .s .s19
#.c.o:
#iccI1
$(CFLAGS) $<
.RESPONSE:
          icc11
include icc11.opt
FILES = foo.o bar.o # list of files, replace with your list
all: $(FILES)
icc11-o myprog $(LFLAGS) $(FILES)
# the above creates a myprog.s19
```

If you are using the IDE, set up your compiler options and then create the `icc11.opt` file by using "Option->Compiler->SaveToDisk" command. Otherwise, uncomment the lines starting with "CFLAGS" down to the line above "include" and delete the "include" line. You will need to make changes to CFLAGS and LFLAGS (although the content of LFLAGS can also bc set by using the ICC11_LINKER_OPTS variable).

All you need is to replace the list of files in the "FILES" variable and change the name of the output file. By using a makefile, you can modify any file in the file list and iimake will recompile any changed file for you. You can even setup imake to compile a source file if a dependency header file changes. For example, you may specify that a source file includes a header file so if that header file changes, the source file should be recompiled:

```
foo.o:    header.h
```

The rest of this chapter describes how imake works in detail.

Using the description file 'makefile'.

When more than one -f <filename> argument pair appears, imake uses the concatenation of those tiles, in order of appearance.

Rule Derivation

If a target has no makefile entry, or if its entry has no rule. imake attempts to derive a rule by each of the following methods:

- implicit rules, read in from a user-supplied makefile.
- standard implicit rules typically read in from the file default `.mk`.
- the rule from the .DEFAULT: entry target, if there is such an entry in the makefile.

If there is no makefile entry for a target, and no rule can he derived for building it.' and if no file by that name is present. imake issues an error message and stops.

Startup Processing

When imake first starts, it reads the environment setting of MAKEFLAGS and scans all present options. Then it reads the command line for a list of options, after which it reads in a default makefile that typically contains predefined macro definitions and target entries for implicit rules. If present, imake uses the file default .mk in the current directory. Othcrwisc it looks for this file along the search path.

Finally, imakc reads in all macro definitions from the command line. These override macro definitions in the makefile.

makefile Components

The makefile(s) may contain a mixture **of comment** lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by escaping the NEWLINE with a backslash '\'.

Comment

A comment line is any line whose first non-space character is a '#'. The comment ends at the next unescaped NEWLINE.

Include

An include line is used to include the text of another makefile. The first seven letters of the line is the word "include" followed by a space. The string that follows is taken as a filename (without double quotes) to include at this line.

If you are using the IDE, you may save the compiler settings that you set in the IDE "Option->Compiler" page in a file so that you can use the same options in your makefile.

Macro

A macro definition line has the form of 'WORD=text...'. The word to the left of the equal sign (without surrounding white space) is the macro name. Text to the right is the value of the macro. Leading white space between the = and the first word of the value is ignored. A word break following the = is implied. Trailing white space (up to but not including a comment character) is included in the value.

Macros are referenced with a $. The following character, or the parenthesized ( ) or bracketed { } string, is interpreted as a macro reference. imake expands the reference (including the $) by replacing it with the macro's value, If a macro contains another macro, the interior one is expanded first. Note that this may lead to infinite expansion, if a macro references itself.

Predefined Macros

The MAKE **macro is** special. It has the value "imake" by default, and temporarily overrides the -n option (which means no execution mode: i.e. commands are printed but not executed) for any line in which it is referred to. This allows nested invocations of imake written as:

```
$(MAKE)..
```

to run recursively, with the -n flag in effect for all commands hut imake.

There are several dynamically maintained macros that arc useful as abbreviations within rules. They arc shown here as references; it is best not to define them explicitly.

- $* refers to the basename of the current target, derived as if selected for use with an implicit rule.
- $< refers to the name of a dependency file, derived as if selected for usc with an implicit rule.
- $@ refers to the name of the current target.

Bccausc imakc assigns $< and $* as it would for implicit rules (according to the suffixes list and the directory contents), they may be unreliable when used within explicit target entries.

A line of the form 'WORD += text...' is used to append the given text to the **end** of **a macro.** The += must be surrounded by white space.

Target Rule

A target entry in the makefile has the following format:

```
target:  dependency...
            rule
```

The first line contains the name of a target, or a list of target names separated by white space. This may be followed by a dependency, or a dependency list that imake checks in order. Subsequent lines in the target entry begin with a space or TAB, and contain shell commands. These commands comprise a rule for building the target.

If a target is named in more than one colon-terminated target entry, the dependencies and rules'are added to form the target's complete dependency list and rule list

To rebuild a target, imake expands macros, strips off initial TABs, and passes each command line to a shell for execution. The first line that does not begin with a space, TAB or # begins another target or macro definition. Macros are expanded during input, for target lines. All other lines have macro expansion delayed until absolutely required.

Special Targets

When used in a makefile, the following target names perform special-functions. Many of them act as commands to imake.

- .DEFAULT

  The rule for this target is used to process a target when there is no other entry for it, and no rule for building it. imake ignores any dependencies for this target.
- .DONE

  imake processes this target and its dependencies after all other targets arc built.
- .IGNORE

  imake ignores non-zero error codes returned from commands.
- .INIT

  This target and its dependencies are built before any other targets are processed.
- .SILENT

  imake does not echo commands before executing them.
- SUFFIXES

  This denotes the suffixes list for selecting implicit rules.
- .RESPONSE

  indicates that the following command can take a response file (i.e. @-file) if the command gets too long. For example,

  ```
  .RESPONSE:
          ilink
  ```

  says that if a command line for ilink **gets too long, then** imake **will put the** command line into a temporary file and use the @-file option with these commands.

Rule

When processing rules, the first non-space character may imply special handling. Lines beginning with the following special characters are handled as follows:

-

  imake ignores any nonzero error code. Normally, imake terminates when a command returns a nonzero status, unless the -i option or the .IGNORE target is used.

- @

  imake does not print the command line before executing it. Normally. each line is displayed before being executed. unless the -s option or the .SILENT target is used.

When any combination of - or @ appear as the first characters after the spaces or TABs, all apply. None are passed to the shell.

Implicit Rules

A target file name is made of a basename and a suffix. The suffix may be null. When a target has no explicit target entry, imake looks for an implicit target made of an element from the suffixes list concatenated with the suffix of the target. If such an implicit target exists, a dependency file name consisting of the basename and the suffix from the suffix list is recursively made. If successful, the implicit rule is invoked to build the target.

An implicit rule is a target of the form:

```
.DsTs:
        rule
```

where .Ts is the suffix of the target. .Ds is the suffix of the dependency file. and 'rule' is the implicit rule for building such a target from such a dependency file.

The Suffix List

The suffix list is given as the list of dependencies for the .SUFFIXES: special-function target. The default list is contained in the SUFFIXES macro. You can define additional SUFFIXES: targets; a SUFFIXES target with no dependencies clears the list of suffixes. Order is significant within the list; imake selects a rule that corresponds to the target's suffix and the first dependency-file suffix found in the list. To place suffixes at the head of the list, clear the list and replace it with the new suffixes, followed by, the default list:

```
.SUFFIXES:
.SUFFIXES: suffixes $(SUFFIXES)
```

Examples

This makefile says that pgm.exe depends on two files a.o and b.o, and that they in turn depend on their corresponding source files (a.c and b.c) along with a common file incl.h:

```
pgm.exe: a.o b.o
        $(CC) a.o b.o -o $@
a.o: incl.h a.c
        $(CC) -c a.c
b.o: incl.h b.c
        $(CC) -c b.c
```

The following makefile uses implicit rules to express the same dependencies:

```
pgm.exe: a.o b.o
        $(CC) a.o b.o -o pgm.exe
a.o b.o: incl.h
```

ICC11IDE - Windows Integrated Development Environment

Parameters to a Windows program such as ICC 11 IDE can be set by modifying the "properties" of the program icon.

Format:

```
icc11 ide <ini file directory>
```

Normally ICC 11 IDE uses `icc11.ini` in your Windows directory to store option settings (or the project file if a project is active). This does not work well if the Windows directory is write-protected such as in a network environment. You may specify an arbitrary directory where the IDE stores the ini file as a parameter to the program.

The setup configuration file icc11wiz.ini is normally stored in the directory where the IDE resides (e.g. c:\icc\bin). If you specify a parameter to the IDE, then that directory is used instead.

ICC11 - Compiler Driver

You usually only interact with the compiler driver and do not need to use the other tools individually when you are compiling. The compiler driver takes your input files and processes them &cording to your specified options. the default options, and the input file types. Some options are passed to the compiler passes directly, such as the -D switch to define macro names. In any case, you can pass any option you want to a particular compiler pass by using the -W option.

If ***any of*** the passes returns a failure code, the compiler driver aborts with the **sub-**program failure code.

You may request the compilation process to stop after a particular pass. For example, -S means compile to assembly only.

Unknown options and file types are passed directly to the linker.

Format:

```
icc11 [ options ] file1 file2 . . .
```

The following options arc recognized. Unrecognized options are passed to the linker.

-A Warn about function declarations without prototypes, and other non-conformance with strict ANSI C rcquircmcnts.

-c Produce object files only. Do not link.

-D<name>[<=def>]

Define a preprocessor macro name. If no definition is given, then the value 1 is implied. For example,

```
iccl 1-Dfoo=bar -Dbaz foo.c
```

is the same as writing

```
#define foo     bar
#define baz     1
```

in the source file.

-E Preprocess the input C files only. Do not compile, assemble or link. The generated output has the same name as the input C file but with a .i extension.

-e Turn on the preprocessor extension which accepts C++ style comments.

-g Produce debugging information for NoICE11.-IInclude path. The preprocessor uses include paths to search for system include files (ones that are enclosed in < and >). You may supply multiple -I options. In addition. the preprocessor searches the **directory** specified by the environment variable ICC11_INCLUDE after these paths.

-l Emit interspersed C and assembly code. Implies -g.

-l<f>

Link in the library file lib<f>.a. For example, use -lfp11 to include the floating point capable printf function,

-L<dir>

Specify the directory in which to search for the library files, crt11.o and end11.o.

-o <file>

Name the output executable S record file. The executable has the default extension.s19 (or .ihx).

-P Generate function declarations with prototypes from the input file This is useful when the input file uses old style K&R function declarations without prototypes and you wish to convert the file to strict ANSI C conformance.

-R Do not link in startup file crt11.o and end11.o.

-s Be silent. By default, if you specify more than one input file, the driver prints out each file name as it is processed.

-S Produce assembly files only. Do not assemble or link.

-U<name>

Undefine a preprocessor macro name, For example

```
icc11 -Ufoo
```

is the same as writing

```
#undef foo
```

in the source file.

-v Be verbose. Print out the command lines used to invoke the other passes and print out version information. If you specify -v more than once, then it prints the commands but does not actually execute them.

-w Suppress warning diagnostics such as unreferenced variables.

-W<pass><arg>

Pass an argument to a particular compiler pass. <pass> must be p, f, a. or l; referring to the preprocessor (icpp), the parser and code generator (iccom 11), the assembler (ias6811), and the linker (ilink) respectively. <arg> is passed directly to the compiler so you will need to specify the switch character '-' if it is a switch option. For example:

```
icc11 -Wl-m foo.c
```

passes the -m switch to the linker.

ICPP - C Macro Preprocessor

The preprocessor reads the input file, processes the macro preprocessor directives in the tile, and writes the output.

Format:

```
icpp    [ options ]<input file> [ <output file> ]
```

The input file usually has a .c extension and the output file has an .i extension. The following are the valid options. If no output file is specified, then the prcproccsscd text is written to standard output.

-11

Assume an HC 11 target. Use the ICC11 specific environment variables and files. If specified, this must be the first argument to icpp. The default is to use ICC 11 environment variables.

-D<name>[<=def>]

Define a preprocessor macro name. If no definition is given, then the value 1 is implied. For example,

```
icpp -Dfoo=bar -Dbaz foo.c
```

is the same as writing

```
#define foo     bar
#define baz     1
```

in the source file.

-E Ignore errors and return success status except when files cannot hc written.

-c Turn on the preprocessor extension which accepts C++ style comments.

-I Include path. The preprocessor uses include paths to search for system include files (ones that are enclosed in < and >). You may supply multiple -I options. In addition, the preprocessor searches the directory specified by the environment

-U<name> Undefine a preprocessor macro name. For example

```
icpp -Ufoo
```

is the same as writing

```
#undef foo
```

in the source file.

ICCOM11 - Compiler Parser and Code Generator

This is the main piece of the compiler system. It takes a preprocessed C input file and generates an assembly output file. The compiler accepts ANSI C language and not the older K & R style C.

Format:

```
        iccom11 [ options ] <input file> [<output file> ]
```

Typically, the input file has a .i extension and the output has a .s extension. If you do not specify an output file, then the output is written to standard output. Note that if a C program does not have any preprocessor directive, you may pass it to iccom 11 directly.

The valid options arc.

-A Warn about function declarations wrthout prototypes, and other non-conformance with strict ANSI C requirements.

-data:<name>

Use <name> instead of "data" for the name of the data section. Note: data not put in the "data" data section will not get intialized by the startup code.

-e<number>

Set the error limit. The compiler aborts when the number of errors reaches the error limit. The default is 20.

-g Produce debugging information. Emit P&E compatible debug map file and other debugging information.

-l Emit interspersed C and assembly code, The source file must have a .c extension.

-P Generate function declarations with prototypes from the input file. This is useful when the input file uses old style K&R function declarations without prototypes and you wish to convert the file to strict ANSI C conformance.

-text:<name>

Use <name> instead of "text" for the name of the text section.

-w Suppress warning diagnostics such as unreferenced variables.

IAS6811 - Assembler

The assembler processes an assembly file as input and produces a relocatable object file as output. Assembler **directives** include conditionals and inclusion of other files. The assembler also generates a listing file (.lis) with the assembly code and the relative code address. Use the linker option -m to generate a full listing with final addresses.

Format:

```
ias6811 [ options ] <input file>
```

The options arc:

-o<file>

Specify the name of the output file. The default is the base name of the input with a .o extension.

ILINK - Linker

Ilink combines object files together to form an executable image. It automatically includes the start-up file crt11.o, end11.o and the library file libc11.a. The linker searches library files last after the object files are processed, so you may place a library anywhere on the link command line.

You may either use the -L flag or the environment variable **ICC-LIB** to specify where the start-up and the library files are. You may also put linker command line options in the environment variable **ICC11_LINKER_OPTS**. For example,

```
set ICC11_LINKER_OPTS=-btext:0x8000 -bdata:0x2000
```

Options contained in the environment variable are read before the options specified in the command line.

Format:

```
ilink [ options ] <file1> <file2> ...
```

Valid options are:

-II

Assume an HC11 target, Use the ICC11 specific environment variables and files. If specified, this must be the first argument to ilink. The default is to use ICC11 environment variables.

-btext:<start address>[.end address][:<start address>[.<end address>]]*

Set the address ranges of the text section. Default is 0 to 0xFFFF.

-bdata:<start address>[.end address][:<start address>[.<end address>]]*

Set the address ranges of the data section. Default is immediately after the text section to 0xFFFF.

-b<section name>:<start address>[.end address][:<start address>[.<end address>]]*

Set the address ranges of the named section. Default is immediately after the last section encountered.

-d<symbol>:<value>

Define a linker symbol which can be used to satisfy a unresolved reference or as a value for defining a section (-b flag) address.

-g Generate a .map file for use with P&E Microcomputer compatible debugger. For ICC11, also generate a .cmd file for NoICE11.

-i Generate an Intel hex format file instead of the default Motorola S record file. The extension is .ihx instead of .s19. Not valid for ICC 16.

-l<f>

Link in the library tile lib<f>.a.

-L<dir>

Specify the directory in which to search for the library files. including crt 11 .o and end.o.

-m Create a map file with a name based on the output file but with a .mp extension and create a .lst file from concatenated .lis files with final addresses.

-R Do not link in startup file crt 11.o and end 11.o.

-o<file>

Specify the name of the output file. which will automatically be given a .s19 extension unless -i is also spccificd. The default name is the base name of the first input file name.

-s<old>:<new>

Treat the section named <new> as if it has the name <old>.

-u<startup tile>

Specify a non-default startup file. The default is crt 11.o

-w Do not emit warning messages.

ILIB - Library Archiver

The library archiver creates and manipulates libraries of object modules.

Format:

```
ilib [ options ] <library archive> <object file 1> <object file 2> .
```

Note that the library tile (e.g., libc.a) must appear before any object files, after the options. Valid options are:

-a Add the object modules to the archive. Create the archive if it does not exist. If the object module already exists in the archive, it is replaced.

-t Print the names of the object modules in the archive.

-x Extract the named object modules from the archive. This overwrites any existing object files of the same name.

-d Delete the object modules from the archive.

Examples:

To place a new-version of putchar.o into the library:

```
ilib -a libc11.a putchar.o
```

To list the contents of a library:

```
ilib -t libc11.a
```

IAS11CVT - Converts Motorola Syntax File

Ias11cvt converts a file written in Motorola assembly syntax to one in ias6811 syntax. No manual adjustment is usually needed after the conversion. The original comments are retained whenever possible.

Format:

```
ias11cvt <infile> [<outfile>]
```

The options are:

<infile>

Name of the input tile

<outfile>

Name of the output file. If not specified, then the output is written to stdout.

IMAKE - Make Utility

The make utility is used to maintain, update, and regenerate groups of programs.

Format:

```
imake [-f filename] [ options ] [target . ..] [macro=value . ..]
```

If no makefile is specified with a-f option, make reads a file named 'makefile'. if it exists. If no target is specified on the command line, make uses the first target defined in maketile. If there is no makefile entry for a target, and no rule can be derived for building it, and if no file by that name is present, make issues an error message and stops.

-f <makefile>

Use the description file 'makefile'. A - as the makefile argument denotes the standard input. The contents of 'makcfile', when present, override the standard set of implicit rules and predefined macros. When more than one -f makefile argument pair appears, make uses the concatenation of those files, in order of appearance.

-d Display the reasons why make chooses to rebuild a target: make displays any and all dependencies that are newer.

-F Force all target updates. Build target and all dependencies even when no update is needed according to file time/date.

-i Ignore error codes returned by commands. Equivalent to the special-function target .IGNORE:.

-k Abandon building the current target as soon as an error code is returned during building. Continue with other targets.,

-n No execution mode. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line contains a reference to the $(MAKE) macro, that line is always executed.

-r Do not read in the default file (default.mk).

-s Silent mode. Do not print command lines before executing them. Equivalent to the special-function target SILENT:.

-S Undo the effect of the -k option. With this switch, the -k option is undone, which means that any error code returned by a child process during building will halt make and display the error code.

-t Touch the target files (bringing them up to date) rather than performing their rules.

-v List the current version number of `make`.

macro=value

Macro definition. This definition remains fixed for the make invocation. It overrides any regular definition for the specified macro within the makefile itself.