DaVinci / OMAP Software Design Workshop

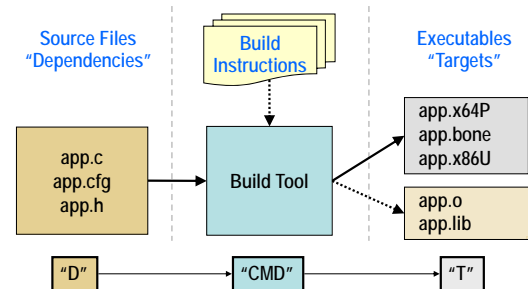| Introduction | 1. Video System Concepts |
	2. DaVinci Hardware Overview
	3. DaVinci S/W Foundation
	4. Software Development Tool Introduction
Application Coding	**5. Building Programs with make**
	6. Device Driver Introduction
	7. Video Drivers : V4L2 and FBdev
	8. Multi-Thread System
Codec Engine	9. Local Codecs - Using a Given Engine
	10. Local Codecs - Building a New Engine
	11. Remote Codecs - Using a Given DSP Server
	12. Remote Codecs - Building a New DSP Server
Algorithms	13. xDAIS and xDM Authoring
	14. (Optional) Using DMA in Algorithms

Outline

◆ Brief overview of gcc for compiling and linking

◆ Understand how to build targets using make

◆ Use rules and variables (built-in, user-defined) in makefiles

◆ Learn how to add "convenience" and "debug" rules

Outline

◆ **Big Picture – Why use make?**

◆ **Creating/Using a Makefile**

◆ **Using Variables and Printing Debug Info**

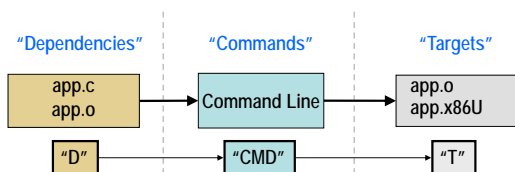◆ **Wildcards and Pattern Substitution**

◆ **Basic Makefile Code Review**

Build Overview

Source Files "Dependencies"   Build Instructions   Executables "Targets"

app.c
app.cfg
app.h   →   Build Tool   →   app.x64P
                               app.bone
                               app.x86U

                               app.o
                               app.lib

"D"   →   "CMD"   →   "T"

◆ Build Tool Goals:
   1. Build _executable_ (target) from _input files_ (dependencies) using _build instructions_ (commands)
   2. Build for _multiple targets_ at once (e.g. ARM, X86, DSP)

◆ Solution: command line (e.g. cl6x, gcc) or _scripting tool_ (make, etc.)

Looking at gcc commands...

Command Line (Examples 1-2)

"Dependencies"   "Commands"   "Targets"

app.c
app.o   →   Command Line   →   app.o
                                app.x86U

"D"   →   "CMD"   →   "T"

◆ Example 1: create an _object_ file (app.o) from an input file (app.c)

   `gcc –g –c app.c –o app.o`

◆ Example 2: create an _executable_ (app.x86U) from an object file (app.o)

   | gcc | –g | app.o | –o | app.x86U |
   | command | flag | dependency | flag | target |

   -c = compile only
   -g = build with debug enabled
   -o = output filename

◆ Might be _more convenient_ to place commands in a _script/batch file...makefile..._

Outline

◆ **Big Picture – Why use make?**

◆ **Creating/Using a Makefile**

◆ **Using Variables and Printing Debug Info**

◆ **Wildcards and Pattern Substitution**

◆ **Basic Makefile Code Review**

Basic Makefile with Rules

"Dependencies" Makefile "Targets"
app.c / app.o → make → app.o / app.x86U
"D" → "CMD" → "T"
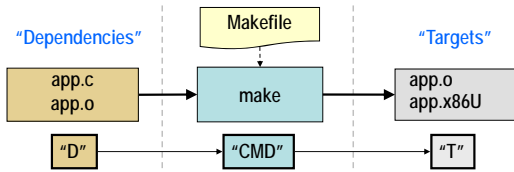
- One of the more common *"scripting"* tools is **make**
- **make** uses "rules" to specify build commands, dependencies and targets
- Generically, a *RULE* looks like this: TARGET: DEPENDENCY [TAB] COMMANDS…
- Remember Example 2? Let's make this into a simple Makefile rule:
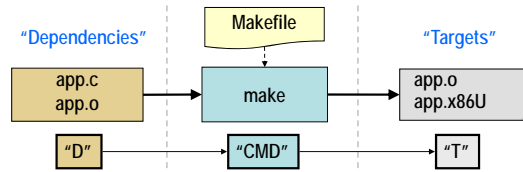
gcc –g app.o –o app.x86U
command flag dependency flag target

- Becomes….

```
app.x86U: app.o
        gcc –g app.o –o app.x86U
```
} *RULE*

Creating Your First Makefile

"Dependencies" Makefile "Targets"
app.c / app.o → make → app.o / app.x86U
"D" → "CMD" → "T"

Command Lines
```
gcc –c –g app.c –o app.o
gcc –g app.o –o app.x86U
```
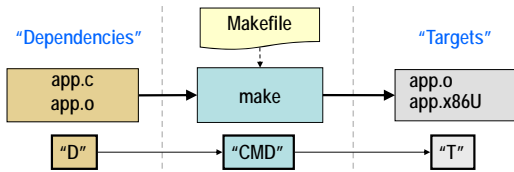
Makefile
```
# Makefile for app.x86U (goal)

app.x86U: app.o
        gcc –g app.o –o app.x86U

app.o:    app.c
        gcc –g –c app.c –o app.o
```

Running make

"Dependencies" Makefile "Targets"
app.c / app.o → make → app.o / app.x86U
"D" → "CMD" → "T"

- To run **make**, you can use the following commands:
  - make   *(assumes the makefile name is "Makefile", runs FIRST rule only)*
  - make app.x86U   *(specifies name of "rule" – e.g. app.x86U)*
  - make –f my_makefile   *(can use custom name for Makefile via forcing flag… -f)*
- **make** looks at *timestamps* for each target and dependency. If the target is *newer* than its dependencies, the rule (and associated commands) will not be executed.
- To "rebuild all", use the *"clean"* rule to remove intermediate/executable files…

Looking at convenience rules...

"Convenience" Rules

"D" → "CMD" → "T"

- Convenience rules (e.g. all, clean, install) can be added to your makefile to make *building/debug easier*.
- For example, a *"clean"* rule can delete/remove existing intermediate and executable files prior to running make again.
- If the rule's target is NOT a file, use the *.PHONY* directive to tell make not to search for that target filename (it's a phony target).
- Let's look at three common convenience rules (to use, type "*make clean*"):

"Build All Targets"
```
.PHONY: all
all:    app.x86U …(all "goals" listed here)
```

"Remove Unwanted Files"
```
.PHONY: clean
clean:
        rm –rf app.o
        rm –rf app.x86U
```

"Copy Executable to the install directory"
```
.PHONY:  install
install: app.bone
        scp app.bone root@bone:.
```

Note: "all" rule is usually the first rule because if you type "make", only the first rule is executed

---

make Rules Summary

"D" → "CMD" → "T"

- 3 common uses of rules include:
  - [.x] – final executable
  - [.o] – intermediate/supporting rules
  - [.PHONY] – convenience rules such as clean, all, install
- Examples:

.x
```
app.x86U: app.o
        gcc –g app.o –o app.x86U
```

.o
```
app.o:    app.c
        gcc –g –c app.c –o app.o
```

.PHONY
```
.PHONY:   clean
clean:
        rm –rf app.x86U
```

- Run:
  - make   *(assumes makefile name is "makefile" or "Makefile" and runs the first rule only)*
  - make app.x86U   *(runs the rule for app.x86U and all supporting rules)*
  - make clean

Outline

Using Built-in Variables

"D" ─────► "CMD" ─────► "T"

- *Simplify your makefile* by using these built-in make variables:
  - $@ = Target
  - $^ = All Dependencies
  - $< = 1st Dependency Only
- *Scope* of variables used is the current rule only.
- Example:

Original makefile…
```
app.x86U: app.o
          gcc –g app.o –o app.x86U
```

Becomes…
```
app.x86U: app.o
          gcc –g  $^   –o  $@
```

User-Defined Variables & Include Files

"D" ─────► "CMD" ─────► "T"

- *User-defined variables* simplify your makefile and make it more readable.
- *Include files* can contain, for example, *path statements* for build tools. We use this method to place absolute paths into one file.
- If "**-include path.mak**" is used, the "**-**" tells make to keep going if errors exist.
- Examples:

makefile
```
include path.mak

CC := $(CC_DIR)gcc

CFLAGS := -g

LINK_FLAGS := -o


app.x86U: app.o
    $(CC) $(CFLAGS) $^ $(LINK_FLAGS) $@
```

path.mak
```
CC_DIR := /usr/bin/
...
# other paths go here…
```

Printing Debug/Warning Info

"D" ─────► "CMD" ─────► "T"

- Two common commands for printing info to stdout window:
  - echo – command line only, flexible printing options ("@" suppresses echo of "echo")
  - $(warning) – can be placed *anywhere in makefile* – provides filename, line number, and message
- Examples:

```
app.x86U: app.o
    $(CC) $(CFLAGS) $^ $(LINK_FLAGS) $@
    @echo
    @echo $@ built successfully; echo

$(warning Source Files: $(C_SRCS))
app.x86U: app.o $(warning now evaluating dep's)
    $(CC) $(CFLAGS) $^ $(LINK_FLAGS) $@
    $(warning $@ built successfully)
```

- $(warning) does *not* interrupt make execution
- A similar function: "$(error)" stops make and prints the error message.

Quiz

- Fill in the blanks below assuming (start with .o rule first):
  - Final "goal" is to build: main.x86U
  - Source files are: main.c, main.h
  - Variables are: CC *(for gcc)*, CFLAGS *(for compiler flags)*

```
CC := gcc
CFLAGS := -g
# .x rule
_____ : _____
          _____  _____  ___  -o ___

# .o rule
_____ : _____  _____
          _____  _____ -c ___ -o ___
```

Quiz

- Fill in the blanks below assuming (start with .o rule first):
  - Final "goal" is to build: main.x86U
  - Source files are: main.c, main.h
  - Variables are: CC *(for gcc)*, CFLAGS *(for compiler flags)*

```
CC := gcc
CFLAGS := -g
# .x rule
main.x86U : main.o
          $(CC)  $(CFLAGS)  $^  -o  $@

# .o rule
main.o : main.c   main.h
          $(CC)  $(CFLAGS) -c  $<  -o  $@
```

- Could $< be used in the .x rule? What about $^ in the .o rule?

Outline

- **Big Picture – Why use make?**
- **Creating/Using a Makefile**
- **Using Variables and Printing Debug Info**
- **Wildcards and Pattern Substitution**
- **Basic Makefile Code Review**

Using "Wildcards"

◆ *Wildcards (*)* can be used in the *command* of a rule. For example:

```
clean :
    rm –rf *.o
```
Removes all .o files in the current directory.

◆ Wildcards (*) can also be used in a *dependency*. For example:

```
print : *.c
    lpr –p $?
```
Prints all .c files that have changed since the last print.
Note: automatic var "$?" used to print only changed files

◆ However, wildcards (*) can *NOT* be used in *variable declarations*. For example:

```
OBJS := *.o
```
*OBJS = the string value " *.o " – not what you wanted*

To set a *variable* equal to a list of object files, use the following *wildcard function*:

```
OBJS := $(wildcard *.o)
```

Simplify Your MakeFile Using "%"

◆ Using pattern matching (or pattern substitution) can help *simplify your makefile* and help you remove explicit arguments. For example:

Original Makefile
```
app.x86U: app.o main.o
    $(CC) $(CFLAGS) $^ -o $@

app.o: app.c
    $(CC) $(CFLAGS) -c $^ -o $@

main.o: main.c
    $(CC) $(CFLAGS) -c $^ -o $@
```

Makefile Using Pattern Matching
```
app.x86U: app.o main.o
    $(CC) $(CFLAGS) $^ -o $@

%.o: %.c
    $(CC) $(CFLAGS) -c $^ -o $@
```

◆ The .x rule depends on the .o files being built – that's what kicks off the .o rules

◆ % is a shortcut for $(patsubst …), e.g. $(patsubst .c, .o)

Outline

◆ **Big Picture – Why use make?**

◆ **Creating/Using a Makefile**

◆ **Using Variables and Printing Debug Info**

◆ **Wildcards and Pattern Substitution**

◆ **Basic Makefile Code Review**

Basic make Makefile – Review (1)

Include file that contains tool paths (e.g. the path to gcc)
```
# -----------------------
# ------ includes -------
# -----------------------
include ./path.mak
```

User-defined variables
```
# -------------------------------
# ------ user-defined vars -------
# -------------------------------
CC := $(X86_GCC_DIR)gcc
CFLAGS := -g
LINKER_FLAGS := -lstdc++
```
In **exercises/make**

"all" rule
```
# -----------------------
# ------ make all -------
# -----------------------
.PHONY: all
all:    app.x86U
```

Main "goal" of makefile… rule for app.x86U
```
# ----------------------------------
# ------ executable rule (.x) -------
# ----------------------------------
app.x86U: app.o
        $(CC) $(CFLAGS) $(LINKER_FLAGS) $^ -o $@
        @echo; echo $@ successfully created; echo
```

Intermediate .o rule. Notice the use of pattern matching.
```
# -------------------------------------------------
# ------ intermediate object files rule (.o) -------
# -------------------------------------------------
%.o: %.c
        $(CC) $(CFLAGS) -c $^ -o $@
```

Basic make Makefile – Review (2)

"clean" rule. Removes all files created by this makefile. Note the use of .PHONY.
```
# ----------------------
# ----- clean all ------
# ----------------------
.PHONY: clean
clean:
        rm -rf app.x86U
        rm -rf app.o
```

"printvars" rule used for debug. In this case, it echos the value of variables such as "CC", "CFLAGS", etc.
```
# -------------------------------------
# ----- basic debug for makefile ------
# -----     example only        ------
# -------------------------------------
.PHONY:   printvars
printvars:
        @echo CC           = $(CC)
        @echo X86_GCC_DIR  = $(X86_GCC_DIR)
        @echo CFLAGS       = $(CFLAGS)
        @echo LINKER_FLAGS = $(LINKER_FLAGS)
```

Outline

◆ **Big Picture – Why use make?**

◆ **Creating/Using a Makefile**

◆ **Using Variables and Printing Debug Info**

◆ **Wildcards and Pattern Substitution**

◆ **Basic Makefile Code Review**