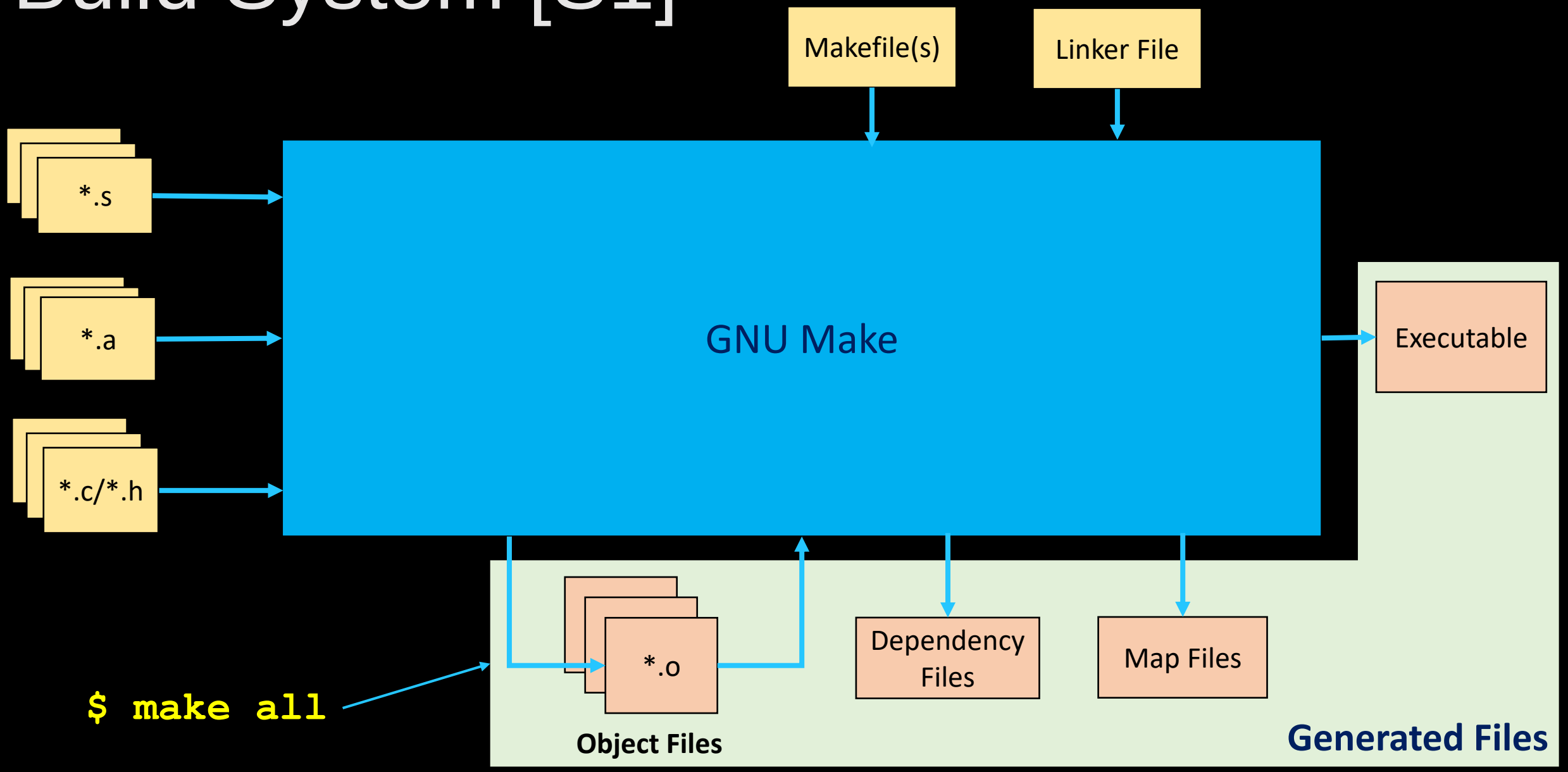
Embedded Software Essentials

Makefiles Part 2

C1 M2 V8

Copyright

Build System [S1]



Makefile Variables [S2]

- Makes Makefile dynamic & eliminates text duplication
  - Variables can use other variables

```
CPU=cortex-m0plus
ARCH=thumb
SPECS=nosys.specs
```

Recursively expanded variables (=)
Variables are expanded when variable is substituted in

```
PLATFORM_FLAGS:=-m$(ARCH) \
                -mcpu=$(CPU) \
                -specs=$(SPECS)
```

Simply Expanded Variables (:=)
Variables are expanded once at time of the variable definition

Pattern Matching [S3]

- Pattern Matching Operator - %
  - Pattern matches a target object rule with an associated source file

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

```
$ make main.o
```

→ **Make uses a pattern match rule to match a target**

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

Pattern Matching [S4]

- Pattern Matches are dynamic

→ **Need a way to track generated object files**

- Can use source variables (SRCS) to generate a list of object files variable (OBJS)

```
OBJS:=$(SRCS:.c=.o)
```

→ **For every *.c file, associate a *.o file with the same name**

```
SRCS:= main.c \
       my_file.c \
       my_memory.c
```

→

```
OBJS:= main.o \
       my_file.o \
       my_memory.o
```

Target Suggestions [S5]

- Targets do NOT have to be a file
  - Need to have a **.PHONY** directive

Can make your own target names!

all – Builds final executable binary

clean – Removes all generated and object files

debug – Builds a debug image with debug symbols enabled

→ **Whatever you want!!!**

```
.PHONY: all
all: main.out
main.out: $(OBJS)
        gcc $(CFLAGS) -o main.out $(OBJS)
```

```
.PHONY: clean
clean:
        rm main.map $(OBJS) main.out
```

Functions & Dynamic Variables [S6]

- Can use make **functions** to process info
  - Output goes into variables
  - shell, file, origin, conditional, etc

```
$(function arguments)
```

- **Shell** functions are one form **command expansion** that can gather data from the system outside of make
  - Use the syntax **$(shell** *command***)**

Shell Command Variables

```
ARCH:=$(shell arch)
CWD:=$(shell pwd)
OS:=$(shell uname)
```

- Use conditional statements to change flags

Example Conditional

```
OS:=$(shell uname -s)
ifeq ($(OS),Linux)
    CC=gcc
endif
```

Overriding Variables [S7]

- Pass input parameters into make to alter build
  - Architecture to build for
  - CPU
  - Platform/Board
  - Compiler Instance
  - Compiler/Linker Options

```
$ make all PLATFORM=msp432
$ make all CPU=cortex-m4
$ make all ARCH=arm
```

Input can set variables or be used conditionally

```
ifeq ($(PLATFORM),MSP)
   CPU=cortex-m4
endif

ifeq($(PLATFORM),FRDM)
   CPU=cortex-m0plus
endif
```

Overriding Variables [S8]

- By making a target variable based, you can change/alter flags for linker or compiler

```
main.elf: main.o file.o foo.o uart.o
      gcc -Wall -I./inc -Wl,-Map=main.map -o main.elf $^

$(TARGET): $(OBJS)
      $(CC) $(CFLAGS) $(INCLUDES) $(LDFLAGS) -o $(TARGET) $(OBJS)
```

Can make our target rules extremely generic yet dynamic!

Special Variables [S9]

- Variables implicitly used by make
  - **CC** – Compiler
  - **CPP** – Preprocessor Program
  - **AS** – Assembler Program
  - **LD** –Linker
  - **CFLAGS** – C program Flags
  - **CPPFLAGS** – C Preprocessor Flags
  - **ASFLAGS** – Flags for Assembler
  - **LDFLAGS** – C program Linker Flags
  - **LDLIBS** – Extra flags for LIbraries

Make has internal rules it uses for targets not defined
You provide the flags

Version Controlled Build System [S10]