A. INTRODUCTION

1. SDSoC – Software Defined System on Chip

- **A "***C/C++ to HDL function***" converter** (invokes ***Vivado HLS*** in background)
- **A C/C++ IDE** to compile source code to object code running on ARM CPU (invokes ***GNU tool chain*** in background)
- **A data mover instantiator** to transfer data between PS and PL

  SDSoC linkers invokes tools within **Vivado Design Suite** to compile the system into bitstream

Summary

SDSoC development environment includes

- Vivado
- Vivado HLS
- SDSoC tools
  - Eclipse/CDT –based GUI
  - Command line tools
  - ARM GNU toolchain

2. SD card image

- SDSoC generates a complete system running Linux, FreeRTOS or Standard Alone in SD card format.

- A SD card image (for Linux) consists
  - A boot image BOOT.BIN including
    - FPGA bit-stream
    - FSBL – First Stage Boot Loader
    - Boot program (Uboot)
  - Application binary (*.elf)
  - Linux image
    - uImage
    - devicetree.dtb
    - uramdisk.image.gz

Summary

By adding `-target-os <...>` in Makefile, can direct SDSoC compiler generates target OS

- Linux (by default)
- FreeRTOS `freertos`
- Baremetal `standalone`

3. Design Flow

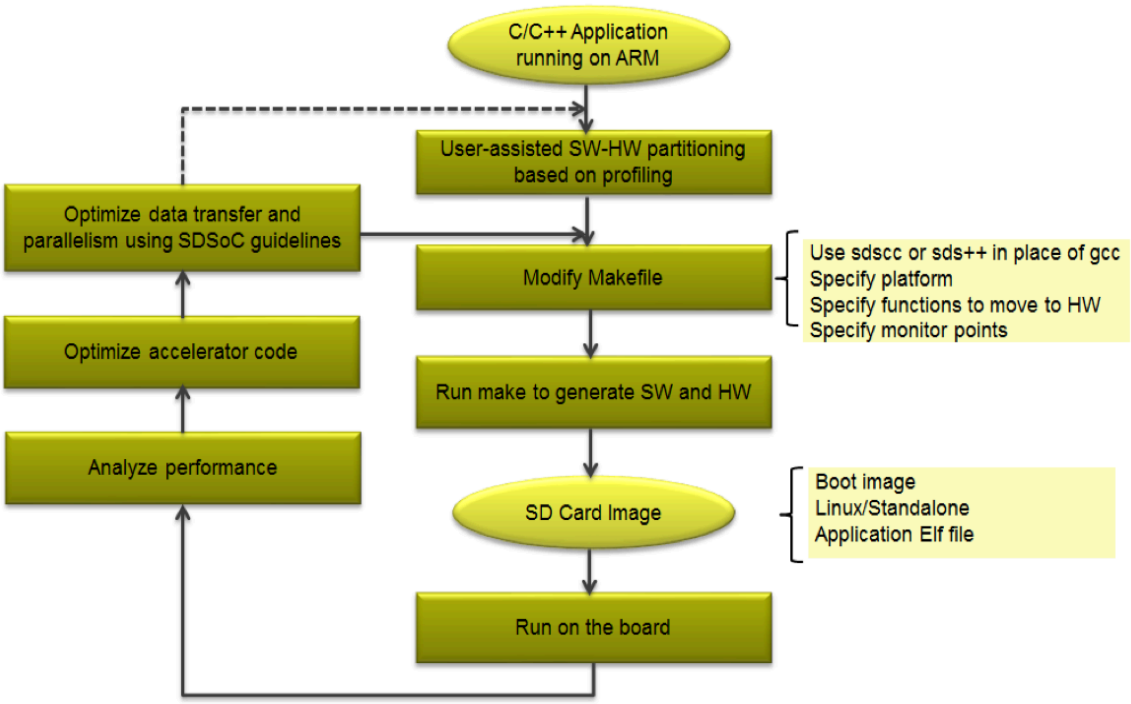


Figure 1: Top-level user design flow for using the SDSoC design environment

a. How to invoke SDSoC ?

SDSoC can be invoked from either:

- Command line (well-suited to scripting flows)
- Make file
- Eclipse-based GUI (interactive features to simplify development)

b. Makefile for SDSoC ?

- A Make file drives the compilation and link process from *source code* to *full SD card image*
- Look at "Basic of Make file" to know basic components of a Makefile

```
APPSOURCES =  mmult.cpp mmult_accel.cpp
EXECUTABLE  =  mmult.elf

SDSFLAGS     =  -sds-pf  zed  \
                -sds-hw  mmult_accel  mmult_accel.cpp  -sds-end \
                -poll-mode 1

CC             =  sds++  ${SDSFLAGS}

CFLAGS        =  -Wall -O3 -c

LFLAGS         =  -O3 -sds-pf zed

OBJECTS := $(APPSOURCES:.cpp=.o)

.PHONY: all

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
        ${CC} ${LFLAGS} ${OBJECTS} -o $@

%.o: %.cpp
        ${CC} ${CFLAGS} $< -o $@

clean:
        ${RM} ${EXECUTABLE} ${OBJECTS}

ultraclean: clean
        ${RM} ${EXECUTABLE}.bit
        ${RM} -rf _sds sd_card
```

Figure 2: Makefile example

Important parameters

- **SDSoC flag**

```
SDSFLAGS = -sds-pf   name_of_platform \
           -sds-hw   funct_name   file_name   -sds-end \
           synchronization_method \
           … …
```

`-sds-pf <…>`	specifies a target platform **zc702** **zed** **zc706**

	microzed
`-sds-hw <…> -sds-end`	specifies name of the top-level function to transfer into hardware and the file contains that function
`-target-os <…>`	Linux by default (do not need to specify) **standalone** **freertos**
`-poll-mode 1`	Synchronization method

Example

```
SDSFLAGS = -sds-pf zed   \
           -sds-hw mmult_accel mmult_accel.cpp  -sds-end \
           -poll-mode 1
```

Remark

- Each top-level function, which is compiled for hardware, MUST reside in a separated file
- That file can contain sub-functions of the top-level function

- **Compiler macro** *to target ARM CPU in Zyn*q

```
CC    =    compiler_name   ${SDSFLAGS}
```

Supporting SDSoC compiler

- **sdscc** #for compiling C
- **sds++** #for compiling C++

Example

```
CC = sds++ ${SDSFLAGS}
```

Remark

- Underneath the hood, SDSoC automatically invokes the arm-gcc compiler to target the ARM CPU
- Can invoke the arm-gcc compiler directly but not recommend

B. SDSoC Compilation and System Linking

1. SDSoC compilation and system linking process

- *Step 1*
  SDSoC compiler **calls Vivado HLS** to do cross-compile a function to programmable logic
  - SDSoC analyzes application code to ***determine data communication patterns and transfer requirement***
  - SDSoC ***builds an AXI-based data motion network*** in hardware based on above requirement. Using standard AXI IP for transport

- *Step 2*
  Caller automatically integrates "device drivers" and additional code to transport data. (Note: Callee is the hardware component)

- *Step 3*
  SDSoC generates ***connectivity description*** (channel-based data model) for system

- *Step 4*
  SDSoC generates a "***stub function***". SDSoC changes the "Caller code" to call the "stub function" instead of calling the function compiled into Hardware

- *Step 5*
  SDSoC calls the **system linker** tool to ***generate*** "***Vivado IPI project/TCL***" from the connectivity description got from STEP 3, then to ***generate the bitstream***

- *Step 6*
  SDSoC **calls the ARM GNU** compiler on the rest of the code and the "stub function" and links them with predefined SW API libraries ***to generate the ELF file***

- *Step 7*
  Generating a SD card image including
  - the bitstream
  - the ELF file
  - the prebuilt Linux kernel image

2. Connectivity framework (CF)

SDSoC design environment builds upon a Connectivity Framework

CF supports multilingual, heterogeneous computing

Connectivity framework consists of:

- An abstract "***channel-based data model***". It is a high-level description of logical and physical connections between system components (Hardware and Software)
- Software APIs for data transfer and allocation
- A "***system linking***" tool for generating a Vivado-based hardware system

3. Stub function

- What is a “Stub function”?
  In the original code, data is passed to “hardware function” by arguments like normal function call.
  However, in the implementation, obviously, data transfer must obey specific protocol.
  That why in the final implementation, application code will call a “***special function***” rather than original “hardware function”
  That special function named “Stub function”

- “Stub function” helps to:
  - Synchronize control between CPU and Hardware
  - Transfer data between Hardware and Software components

- How could “Stub function” do so?
  “Stub function” calls SW APIs defined by CF layer

- Important APIs
  `cf_send_i` sends data to accelerators
  `cf_receive_i` receives the result from the accelerator.
  `cf_wait` waits until a transaction completes

  Note
  - A “stub function” only uses 3 above APIs to communicate btw HW and SW regardless of actual data communication method (Streaming or memory-map …)
    - ⇨ The code of “Stub function” is ***completely independent*** of the actual communication protocol

  - Underneath the hood, the implementation of above APIs is automatically generated based on communication protocol which is provided by the system_linker tool

Example of Stub Function

- Firstly, the application sends commands to control the hardware component. Then, wait until this transaction completes
- Secondly, it sends data to ports of the hardware component
- Thirdly, it receives result from hardware component

```
void _p0_mmult_accel (float in_A[A_NROWS*A_NCOLS],
                      float in_B[A_NCOLS*B_NCOLS],
                      float out_C[A_NROWS*B_NCOLS])
{
  int start_seq[3];
  start_seq[0] = 0x00000003;
  start_seq[1] = 0x00010001;
  start_seq[2] = 0x00020000;
  cf_request_handle_t request_swinst_mmult_accel_cmd;
  cf_send_i(&(swinst_mmult_accel.cmd_mmult_accel), start_seq, 3*sizeof(int),
&request_swinst_mmult_accel_cmd);
  cf_wait(request_swinst_mmult_accel_cmd);

  cf_send_i(&(swinst_mmult_accel.in_A_PORTA), in_A, 1024 * 4, &request_0);
  cf_send_i(&(swinst_mmult_accel.in_B_PORTA), in_B, 1024 * 4, &request_1);

  size_t num_out_C_PORTA;
  cf_receive_i(&(swinst_mmult_accel.out_C_PORTA), out_C, 1024 * 4,
&num_out_C_PORTA, &request_2);

  cf_wait(request_0);
  cf_wait(request_1);
  cf_wait(request_2);
}
```

Table 3: The replacement stub function generated by the SDSoC design environment

C. SDSoC-Based System Optimization

1. Memory allocation for efficient data transfer

- Standard C **`malloc`** allocates memory that is contiguous in the virtual memory space, but may not contiguous in the physical memory => High overhead for data transfer

- ***Solution***
  SDSoC provides the following 2 APIs that encapsulate the contiguous memory allocation

```
sds_alloc (size_t size)      - To allocate memory
sds_free (void *memptr)      - To free the memory
```

Libraries

```
#include "stdlib.h"      - Include first to provide the size_t type
#include "sds_lib.h"     - For above APIs
```

2. Clock frequency

Add following options to **`sdscc/sds++`** in the **makefile**

```
-clkid n       - specifies the clock ID n should be used for hw accelerator
-dmclkid n     - specifies the clk ID to use for data motion network
```

Note: SDSoC 2014.4 allows to specify only a single clock ID for both the hardware functions and the data motion network

3. Task level pipelining

- SDSoC compilers allows to pipeline multiple calls to **an accelerator** to overlap the data transfer

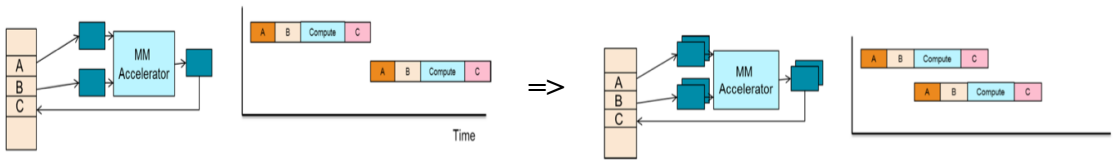


Figure 2: Sequential execution of matrix multiply calls

Figure 3: Pipelined execution of matrix multiply calls

- Underneath the hood, multiple buffers are generated (extra BRAM) to store data of overlapped calls
- Advantage: Reduce latency with small extra resource

SDSoC pragma (2 important pragmas)

```
#pragma SDS async(ID)
```

- The function call that immediately follows should be executed asynchronously
- The processor initiates the call but continues with its own execution rather than waiting for the call to finish

`sds_wait(ID)` or `#pragma SDS wait(ID)`

- Used to wait for an asynchronous call to complete
- Assume: Tasks' completions are in the same order as their issues
- Each ID has one queue to keep results

- ***ID*** specifies the ***unique ID of the accelerator*** that is used to execute the call
  - The same ID is used in a subsequent wait statement to synchronize with the accelerator
  - Different ID will create different instances of the accelerator

Note

From PL side, a call (hardware instance) can access its inputs and write its outputs anytime between the start of the call and the corresponding wait

⇨ In the application, the arguments of a call should be accessed ***only after*** the `sds_wait`

Example

```
for (i = 0; i < NUM_TESTS; i++) {
  /*< Step 1: Fill up the pipeline stage        */
  for (vec = 0; vec < pipeline_depth; vec++) {
  #pragma SDS async(1)
    mmult_accel(tin1Buf[vec], tin2Buf[vec], toutBufHw[vec]);
  }

  /*< Step 2: A new task is issued when an      *
   *          earlier task has finished         */
  for (vec = pipeline_depth; vec < NUM_VECTORS; vec++) {
  #pragma SDS wait(1)
  #pragma SDS async(1)
    mmult_accel(tin1Buf[vec], tin2Buf[vec], toutBufHw[vec]);
  }

  /*< Step 3: Wait for all calls in the pipeline *
   *          to complete                       */
  for (vec = 0; vec < pipeline_depth; vec++) {
  #pragma SDS wait(1)
  }
}
  /*< Step 4: Get the result                    */
return mmult_result_check(toutBufSw, toutBufHw);
```

D. Libraries, Application modes

1. Using an IP library

- Step 1: MUST include header file of an IP in the source code
- Step 2: In the Make file,
  - ADD the path to the header file using the **–I** switch
  - LINK against the library, use the –L and –I switches

```
>> sdscc –c -I<path to header> -o main.o main.c
```

```
>>sdscc -${PLATFORM} ${OBJECT} -L<path to lib> -l<lib> -o <app elf file>
```

Example

```
CC = sdscc -sds-pf zc702
main.o: main.c
	${CC} -c -I./include $< -o $@

fir.elf: main.o
	${CC} main.o -L./lib/hw -lfir -lm -o $@
```

2. Using HLS library

- Only need to INCLUDE header file of HLS library
- Do not need to link or add PATH like “Using IP library”

3. Standard Alone

Changing in the makefile

```
CFLAGS = -target-os standalone
LFLAGS = -target-os standalone
```

Limitation:

- Does not support multi-threading, virtual memory or address protection
- File I/O: not using usual C APIs, but instead through a special API using libxilffs. MUST disable DCache before doing any file operation

E. File I/O

1. In Linux

Can use usual C APIs like normal linux applications.

2. Standard Alone mode

Not using usual C APIs, but instead through a special API using **libxilffs**.

MUST disable DCache before doing any file operation

```
#include "ff.h"
#include "xil_cache.h"

FIL fil_in, fil_out;              // File obj
FRESULT Res;                      // File status
char *SD_in, *SD_out;
char FileName_in[32] = "input.yuv";
char FileName_out[32] = "output.yuv";
SD_in = (char *)FileName_in;
SD_out = (char *)FileName_out;
Xil_DCacheFlush();
Xil_DCacheDisable();
Res = f_open(&fil_in, SD_in, FA_OPEN_EXISTING  | FA_READ);
Res = f_open(&fil_out, SD_out, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
f_read(&fil_in, y[frame][row], bytes_per_cc * cols, &NumBytesRead);
f_close(&fil_in);
f_close(&fil_out);
```

F. HLS-BASED CODING [HARDWARE IMPLEMENTATION]

1. Function Arguments – Accelerators' interfaces

Each argurment of a hardware function will be transferred to a corresponding AXI interface.

- **AXI stream – Array argument**
  - Array arguments are mapped to either
    - Ap_fifo
    - Bram interfaces
  - Limitation of SDSoC v2014.4: it be able to automatically transfers up to SIXTEEN array arguments (8 inputs, 8 outputs) to AXIS
  - If #input or #output arguments > 8 => MUST explicitly code axis interfaces in your HLS code

```
#ifdef __SDSVHLS__
void correlation_accel_v2(      int     number_of_days,                                 /* CPU in*/
                                int     number_of_indices,                              /* CPU in*/

                                ap_axiu<32,1,1,1> in_indices[MAX_NUM_INDICES * MAX_NUM_DAYS],
                                ap_axiu<32,1,1,1> out_correlation[MAX_NUM_INDICES / 2 * (MAX_NUM_INDICES - 1)]
#else
void correlation_accel_v2(
                                // const int            number_of_days,                  /* CPU in*/
                                // const int            number_of_indices,               /* CPU in*/
                                // hls::stream<float>   &in_index,                       /*  Input*/
                                // hls::stream<axis_t> &out_correlation                  /* Output*/
                                // )

                                int     number_of_days,                                 /* CPU in*/
                                int     number_of_indices,                              /* CPU in*/

                                float   in_indices[MAX_NUM_INDICES * MAX_NUM_DAYS],     /*  Input*/
                                float   out_correlation[MAX_NUM_INDICES / 2 * (MAX_NUM_INDICES - 1)]
                                )
{

#ifdef __SDSVHLS__
#pragma HLS interface axis port=out_correlation
#pragma HLS interface axis  port=in_index
#pragma HLS interface ap_ctrl_none port=return

    volatile ap_axiu<32,1,1,1> tmp1;
    union {
          int ival;
          float floatval;
     } conv1;
#endif
```

  - Code in SDSoC includes 2 parts:
    - One is used by Vivado HLS to generate Hardware
    - One is used by GNU toolchain to simulate functional behavior of system
  - Union {} is the way to convert from unsigned type to float type with low cost.

- **AXI memory map – Array argument**
  - NOTE: every HLS function containing an arrgument that maps to an AXIMM MASTER requires **a return value** or **other output scalar**

```
25  int correlation_accel_v1(
26      int     number_of_days,                                 /* CPU in*/
27      int     number_of_indices,                              /* CPU in*/
28
29      float   in_indices[MAX_NUM_INDICES * MAX_NUM_DAYS],     /*  Input*/
30      float   out_correlation[MAX_NUM_INDICES / 2 * (MAX_NUM_INDICES - 1)]
31  )
32  {
33      /* AXI Master*/
34      #pragma HLS INTERFACE m_axi depth=2520000   offset=direct port=in_indices
35      #pragma HLS INTERFACE m_axi depth=49995000  offset=direct port=out_correlation
36      /* Need a scalar return for AXI Master  */
37      #pragma HLS INTERFACE ap_ctrl_hs port=return
```

  - –offset : to apply an address offset
    - off: Does not apply an offset address. This is the default.
    - direct: Adds a 32-bit port to the design for applying an address offset.
    - slave: Adds a 32-bit register inside the AXI4-Lite interface for applying an address offset.
  - –depth : size of memory's chunk which AXIMM will access
  - Do not need to declare the base address, it will be handled automatically by the tool

- **AXI Lite – Scalar argument**
  - DO NOT NEED to use any pragmas for specifying AXI LITE interfaces
  - SDSoC automatically inserts the axis_accelerator_adapter for axilite control

2. Data Mover – Port declaration

Following pragmas are put in the header file *.h above hardware function.

a. Choosing PS port

```
#pragma SDS data sys_port(input: AFI, output: ACP)
```

  - AFI : HP0- 3 (Asynchronous FIFO Interface – High Performance port)
  - ACP: ACP port
  - Do not need to specify for GP port

b. Choosing data mover

```
#pragma SDS data data_mover(input: AXIDMA_SG, output:
AXIDMA_SIMPLE, out:AXIFIFO)
```

  - AXIDMA_SG : Axi DMA with Scatter Gather mode
  - AXIDMA_SIMPLE: Axi DMA with Simple mode
  - AXIFIFO: data mover is assigned to the M_AXI_GPx port

Note: Data mover will be specified indirectly by specify memory allocation method.

  - Malloc() : AXIDMA_SG
  - Sds_alloc() : AXIDMA_SIMPLE

c. Choosing runtime data copy size between PS and PL

```
#pragma SDS data copy(0:N)
```

It is used together with AXIS, it specifies the size of transferred data in runtime.

Example: In the header file,

```
#pragma SDS data copy(in_indices[0: number_of_indices * number_of_days])
#pragma SDS data copy(out_correlation[0: number_of_indices * (number_of_indices - 1)/2])
#pragma SDS data sys_port(in_indices:AFI, out_correlation:AFI)
#pragma SDS data data_mover(in_indices: AXIDMA_SIMPLE, out_correlation: AXIDMA_SIMPLE)
int correlation_accel_v2(
    int     number_of_days,                     /* CPU in*/
    int     number_of_indices,                  /* CPU in*/
    float   in_indices[MAX_NUM_INDICES * MAX_NUM_DAYS],         /*  Input*/
    float   out_correlation[MAX_NUM_INDICES / 2 * (MAX_NUM_INDICES - 1)]
);
```

Important note:

- In version 2014.4, to transfer data which has a size > 8MB, we need to configure DMA under Scatter Gather mode; while allocating data with sds_alloc API

H. NOTES WHEN WORKING WITH BAREMETAL AND SDSOC

1. Manually programming FPGA and running application on Zedboard

Sometimes, SDK or SDSoC cannot program FPGA or run the application due to some unknown problems. Here is the way to overcome it

Three important files are needed to run an application on Zedboard

- ***.bit** bitstream file including hardware configuration of the Zynq PL
- ***.elf** execution file of the application
- **ps7_init.tcl** initialization file for the Zynq PS

Locating in workspace of the project.
Find their relative paths by these instructions
*find . -name *.bit*
*find . -name *.elf*
find . -name ps7_init.tcl

Manually programming FPGA and running application on Zedboard by following this procedure

source **[vivado_setting.sh file]**

-- open Xilinx Microprocessor Debugger
xmd

-- program hardware
fpga -f **[path_to_bitstream_file]**

-- connect debugging system
connect arm hw

-- initialization for zynq ps
source **[path_to_tcl file]**
ps7_ init

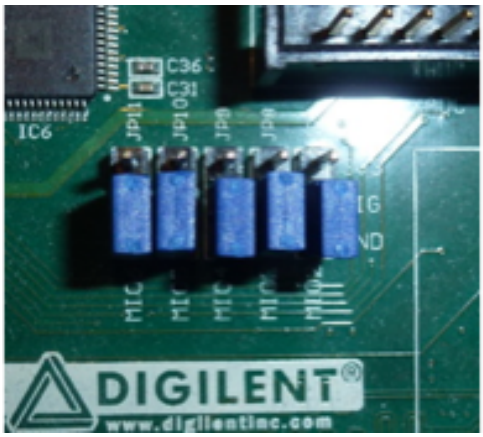
-- load application to memory through data cable ()
dow **[path_to_elf_file]**

-- run application
run

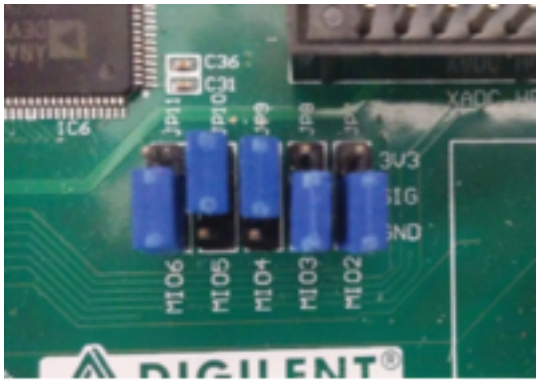
2. Board setup

Jumpers need to be set correctly based on running mode of applications

Standard Alone



Linux



When working with minicom to communicate with UART port

- Must run as system administrator
- To know which port are connecting with UART using *dmesg* command
- Should turn off Hardware/Software control flow
- Baud rate for UART1 port of Zedboard, by default, is **115200**

sudo minicom -D /dev/ttyACM0 -b 115200 -s

Choose Serial Port Setup > Turn off HW/SW flow control

I. Basic of Make File

A ***Makefile*** composes from following components:

- Comments
- Rules
- Directives
- Macro
- Response files

1. Comments

Syntax

```
# comments …
```

Example

```
# Makefile for Opus Make 6.1
#
# Compiler: Microsoft C 6.0
# Linker: Microsoft Link 5.10
```

2. Rules

Tells "MAKE" both ***when*** and ***how*** to make a file

Classification:

- Explicit rules: Supplied explicitly in the Makefile
- Inference rules: Generalize the make process

Syntax

```
target  : prerequisites
          recipes
          ...
```

Example

```
main.o : main.c defs.h
      cc -c main.c
```

Explain

Target *can be either*	**Prerequisite**	**Recipe or Shell lines**
- Name of a file that is generated by a program: o Executable or object files - An action to carry out, such as '***clean'***, called PHONY target	Inputs to create target - Sources	- Action that make carries out. - A recipe may have more than one command, either on the same line or each on its own line.

a. Phony target

Targets are just actions (do not refer to files) are called phony target

b. Dependency lines (*when* to build a target)

Syntax

```
Target : prerequisites
```

Example

```
project.exe : main.obj io.obj
```

At runtime, `project.exe` is rebuilt WHEN timestamp of `main.obj` or `io.obj` is newer than `project.exe`

Remark

- "Make process" is recursive since, it, firstly, check timestamp of sources => Ensure sources always be updated prior target
- Additional dependencies,

Example	Or using additional dependencies

`main.obj : main.c def.h`
`io.obj : io.c def.h`
	`main.obj : main.c`
	`io.obj : io.c`
	`main.obj io.obj : def.h`

c. Shell lines (or Recipes - how to build, update a target)

- **Need** to put **a tab character at the beginning of every recipe line!**
  - To change prefix with a character other than tab, can set the **.RECIPEPREFIX** variable
- Bear in mind: "MAKE" does not know anything about how the recipes work.
  All "MAKE" does is **execute** the recipe when the target file needs to be updated

Example

```
main.o : main.c defs.h
        cc -c main.c
```

When either timestamp of `main.c` or `defs.h` is newer than timestamp of `main.o`, `main.o` needs to be updated. "Make" will execute the command `cc -c main.c`

3. Macro

Assignment symbol: "="

Calling macro: "$(…)"

or "${…}"

To replace repeated text

a. Static macro

Without using Macro

```
project.exe : main.obj io.obj
      tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib

main.obj : main.c
      bcc -ms -c main.c

io.obj : io.c
      bcc -ms -c io.c

main.obj io.obj : def.h
```

Using Macro

```
# Macro declaration
OBJS   = main.obj io.obj
MODEL  = s
CC     = bcc
CFLAGS = -m$(MODEL)

# Makefile content
project.exe : $(OBJS)
      tlink c0$(MODEL) $(OBJS), project.exe,, c$(MODEL) /Lf:\bc\lib

main.obj : main.c
      $(CC) $(CFLAGS) -c main.c

io.obj : io.c
      $(CC) $(CFLAGS) -c io.c

$(OBJS) : incl.h
```

Remark

- Macro can be declared on the Command Line
  Example:

```
make CFLAGS=-ms
```

or

```
make "CFLAGS=-ms -z -p"
```

```
# macro containing spaces must be
#       enclosed in a bracket
```

b. Runtime Macro or Dynamic macro

Value of these macros is set dynamically

```
.TARGET         #return current target
.SOURCE         #return the first source of the explicit sources (from an inference rule)
.SOURCES        #list all sources
```