Realisatie stappen Proof of Concept

Dafuq heb ik gedaan?

# BMPTK

## Makefile.inc

Om de aansturing van de esp32 te kunnen maken moeten een aantal zaken toegevoegd worden in de BMPTK build tool. In de voorbeelden die gegeven worden zijn soms stukken code weg gehaald voor leesbaarheid van het verslag. De volledige code is te vinden op de GitHub repository[[1]](#footnote-1) van dit onderzoek.

ifeq ($(TARGET),esp\_devkit\_v1)

CHIP ?= esp\_wroom\_32

# Clock speed: 240 MHz

XTAL ?= 240000

Endif

*Code voorbeeld 1 - Target selectie*

In het bestand “Makefile.inc” is een nieuwe target aangemaakt, zie Code voorbeeld 1. Aangezien voor dit onderzoek de ESP32 DevKit v1 gebruikt is heeft de target naam “esp\_devkit\_v1” gekregen. Op dit programmeerbordje zit de “esp\_wroom\_32” chip die een klok snelheid (XTAL) heeft van 240 MHz.

Nadat de target gemaakt is, wordt gekeken wat voor de gedefinieerde chip moet gebeuren, zie Code voorbeeld 2. Om de correcte chip te vinden wordt een ifeq() functie gebruikt waarna de define “Xtensa\_LX6” aangeroepen wordt. Nadat daar alle vlaggen goed zijn gezet, worden de ROM, RAM en STACK grootte gedefinieerd.

ifeq ($(CHIP),esp\_wroom\_32)

$(eval $(Xtensa\_LX6))

ROM\_SIZE ?= 448k

RAM\_SIZE ?= 520k

STACK\_SIZE ?= 200000

DEFINES +=

Endif

*Code voorbeeld 2 – Chip selectie*

Binnen de Xtensa\_LX6 define worden meerdere vlaggen gezet, zie Code voorbeeld 3. Eerst wordt de naam van de Application binary interface van de esp32 (ESP\_EABI), de locatie hiervan (GCC\_LX6) en de prefix van de uitvoerbare bestanden (PREFIX) gedefinieerd. Wanneer deze gedefinieerd zijn wordt de define “Xtensa\_LX” aangeroepen. Daarna worden een aantal vlaggen gevuld waarbij het beginadres van de ROM (ROM\_START) en RAM (RAM\_START) gedefinieerd. Ook worden hier bepaalde parameters gespecificeerd die meegegeven dienen te worden aan de compiler en commando’s die uitgevoerd dienen te worden voor het checken van de seriële poort of het runnen van het programma.

define Xtensa\_LX6

$(eval ESP\_EABI := xtensa-esp32-elf)

$(eval GCC\_LX6 := …)

$(eval PREFIX := $(GCC\_LX6)/$(ESP\_EABI)-)

$(eval $(Xtensa\_LX))

ROM\_START ?= 0x40000000

RAM\_START ?= 0x40070000

PORT\_CHECK ?= $(CHECK\_PORT) \

$(SERIAL\_PORT)

RESULTS += $(BIN)

…….

RUN := …

DEFINES += -DDONT\_USE\_CMSIS\_INIT

DEFINES += -DBMPTK\_INCLUDE\_CHIP

RUN\_TERMINAL ?= …

endef

*Code voorbeeld 3 – Xtensa\_LX6 Define*

De Xtensa\_LX define doet eigenlijk vrij weinig. Binnen deze define wordt de Xtensa define aangeroepen en kunnen extra argumenten meegegeven worden voor het builden van het programma.

define Xtensa\_LX

$(eval $(Xtensa))

ARCH\_FLAGS :=

endef

*Code voorbeeld 4 – Xtensa Define*

Binnen de Xtensa define worden de meest belangrijke vlaggen gevuld, zie code voorbeeld 5. Hierbij wordt eerst de Embedded define aangeroepen waarbij de basis functionaliteiten gezet worden voor embedded systemen. Daarna worden onder andere de benodigde source files, header files, linkerscriptfiles en libraries gevult. Het vinden van de correcte source en header files gebeurd door twee verschillende functies die gemaakt zijn: “GenerateComponentInfo” (zie code voorbeeld 6) en “GenerateComponentSources” (zie code voorbeeld 7).

define Xtensa

$(eval $(Embedded))

……

ESP\_CORE := $(BMPTK)/targets/esp/esp32

INCLUDES += -I$(PROJECT\_PATH)/build/include

……

$(foreach component, \

$(COMPONENTS), \

$(eval $(call GenerateComponentInfo,$(component))))

$(foreach component, \

$(SOURCE\_COMPONENTS), \

$(eval $(call GenerateComponentSources,$(component))))

LINKERSCRIPT := esp32\_out.ld

COMMON += -D BOOTLOADER\_BUILD=1

CORE\_FLAGS += -nostartfiles

ALL\_LIBRARIES += …

LIBS += -L"$(ESP\_CORE)/lib/"

LDFLAGS += -L$(ESP\_CORE)/lib/

…….

OBJ += bmptk\_startup\_esp32.o

CPU ?= XtensaLX6

ROOT := call\_start\_cpu0

TARGET\_C\_FLAGS += -mlongcalls

TARGET\_CPP\_FLAGS += -mlongcalls

endef

*Code voorbeeld 5 – Xtensa Define*

In de functie GenerateComponentInfo wordt informatie vergaard over de header files van de verschillende benodigde componenten uit de ESP-IDF. Deze functie wordt aangeroepen voor elk component dat benodigd is voor het programma. Vervolgens wordt gekeken in welke mappen van het component de headerfiles staan die benodigd zijn en wordt het volledige pad ernaartoe opgeslagen in de vlag “INCLUDES”. Daarna wordt gekeken of bepaalde compileer commando’s benodigd zijn en zo nodig toegevoegd aan de vlag “LDFLAGS”.

define GenerateComponentInfo

COMPONENT\_ADD\_INCLUDEDIRS :=

COMPONENT\_SRCDIRS :=

$(eval COMPONENT\_PATH := …)

COMPONENT\_PATHS += $(COMPONENT\_PATH)

$(eval include $(COMPONENT\_PATH)/component.mk)

$(eval COMPONENT\_INCLUDE\_DIRS := )

$(foreach dir, $(COMPONENT\_ADD\_INCLUDEDIRS), \

$(eval COMPONENT\_INCLUDE\_DIRS += \

$(subst /.,,$(COMPONENT\_PATH)/$(dir)))\

$(eval INCLUDES += \

-I$(subst /.,,$(COMPONENT\_PATH)/$(dir)))\

)

$(eval LDFLAGS += $(COMPONENT\_CFLAGS))

$(eval CLEAN += SOURCES)

endef

*Code voorbeeld 6 – Xtensa Define*

define GenerateComponentSources

$(eval COMPONENT\_PATH := …)

$(eval include $(COMPONENT\_PATH)/component.mk)

$(eval COMPONENT\_INCLUDE\_DIRS := )

$(foreach dir, $(COMPONENT\_ADD\_INCLUDEDIRS), \

$(eval COMPONENT\_INCLUDE\_DIRS += \

$(subst /.,,$(COMPONENT\_PATH)/$(dir))) \

$(eval INCLUDES += \

-I$(subst /.,,$(COMPONENT\_PATH)/$(dir))) \

)

$(eval COMPONENT\_SOURCE\_DIRS :=)

$(foreach dir, $(COMPONENT\_SRCDIRS), \

$(eval COMPONENT\_SOURCE\_DIRS += \

$(subst /.,,$(COMPONENT\_PATH)/$(dir)))\

$(eval SEARCH += \

-I$(subst /.,,$(COMPONENT\_PATH)/$(dir)))\

)

$(foreach src, $(COMPONENT\_SOURCES), \

$(eval SOURCES += $(src))\

)

endef

*Code voorbeeld 7 – Xtensa Define*

De functie GenerateComponentSources zorgt voor de verzameling van informatie over de verschillende source files die benodigd zijn voor het component. Hierbij wordt ook weer gekeken naar de mappen waar de source files in staan en wat het volledige pad hiervan is. Deze worden dan ook toegevoegd aan de “INCLUDES” vlag. Daarnaast worden de paden ook toegevoegd aan de vlag “SEARCH”. Deze vlag bevat namelijk de paden waar de compiler moet zoeken voor de verschillende source files. Als laatste worden de benodigde source files toegevoegd aan de vlag “SOURCES”, deze vlag bevat namelijk de namen van de verschillende source files die omgezet dienen te worden tot objecten.

## Bmptk.mk

In het originele build systeem van de ESP-IDF wordt het bestand project.mk, dat in de make folder van het ESP-IDF staat, vanaf de Makefile in de workspace aangeroepen. Alleen heeft project.mk een aantal problemen. Zo kan het alleen op een linux omgeving aangeroepen worden of vanuit een Linux gebaseerde command promt op Windows. Dit komt aangezien in project.mk Linux commando’s gebruikt worden voor bijvoorbeeld het aanmaken van mappen of kijken of bestanden al bestaan. Daarnaast verbied project.mk dat ‘:’ in padnamen voorkomen, aangezien dit problemen geeft met de verschillende targets die in de Makefiles aangeroepen worden. Om deze redenen is gekozen om dit bestand te herschrijven tot “bmptk.mk”. Hierbij zijn de Linux commando’s voor o.a. het aanmaken van mappen in de build folder van het project omgevormd tot een python script dat uitgevoerd wordt. Ook is in bmptk.mk het gebruikt van ‘:’ toegestaan in padnamen, waardoor het vanuit het Windows command promt uitgevoerd kan worden.

Naast deze essentiële omvormingen zijn ook nog twee extra debug functionaliteiten toegevoegd. Wanneer de target “see-defined-variables” als make commando aangeroepen wordt, worden alle gedefinieerde vlaggen in bmptk.mk uitgeprint op het scherm. En met het commando “see-var <naam>” kan de waarde die in deze vlag zit bekeken worden. Hierdoor kunnen soms problemen dat vlaggenverkeerd gevuld worden snel gevonden worden door de programmeur.

# Problemen

**Probleem:** Letters in objectnamen verdwenen

Tijdens de realisatie van het Proof of Concept deed zich bij de linker een vreemd probleem voor. Bij het zoeken naar de benodigde objecten verdwenen soms letters, waardoor de linker sommige objecten niet kon vinden. Zo zocht de linker bijvoorbeeld naar “dport\_acess.o” in plaats van “dport\_access.o”. De rede van dit probleem is dat waarschijnlijk de commando’s die uitgevoerd werden het maximale letteraantal overschreden. Deze commando’s bevatte namelijk zeer veel argumenten en paden naar verschillende bestanden. Om dit probleem te kunnen oplossen, zijn de benodigde componenten voor de basis aansturing van de esp32 samengevoegd tot één component “bmptk-core”. Hierdoor verminderde het aantal paden dat in het commando moest staan en verdwenen de letters niet meer.

**Probleem:** Multiple target pattern error

Tijdens de realisatie van het Proof of Concept was een van de eerste problemen waar tegenaan gestuit werd, de “Multiple target pattern error”. Dit probleem doet zich voor wanneer bij de dependensies van een target een ‘:’ voorkomt, zoals te zien in het onderstaande voorbeeld. De dubbelepunt geeft aan dat dit het einde is van de target die gespecificeerd wordt. Wanneer een dubbelepunt bij de dependensies komt de staan ziet GNU Make dit als het einde van een tweede (multiple) target. Hierbij is gekozen om dit op te lossen met een tussenstap. Binnen de target wordt een functie aangeroepen waarbij handmatig de dependensie gecontroleerd word.

Make: **C:/**user/project

…

*Code voorbeeld – dubbelepunt in target dependensie*

**Probleem:** Missing separator

Het krijgen van een “Missing separator” error kan vaak veel onduidelijkheid geven. Meestal betekend dit dat ergens in de code een tab neergezet is, in plaats van vier spaties. Binnen defines moet namelijk gebruik gemaakt worden van spaties en binnen targets moeten tabs gebruikt worden. GNU Make is hier zeer gevoelig voor. Echter wordt deze error gegeven op de aanroep van de eerste functie. Hier kan zijn dat die functie een andere functie aan roept, die weer een andere functie aanroept, waar de fout in zit. Dit kan daardoor soms veel zoekwerk zijn als deze error gegeven wordt.

1. GitHub repository van het onderzoek: https://github.com/renedekluis/onderzoeksemester\_18-19 [↑](#footnote-ref-1)