WorkLog

Antoine Roffet

25 janvier 2021

Table des matières

1 Introduction **3**

2 Step0 **3**
2.1 Preface by Pr. Olivier Gruber 3
2.2 QEMU 3
2.3 GNU Debugger 3
2.4 Makefile 5
2.4.1 Linker Script 5
2.4.2 ELF Format 5
2.5 Startup Code 5
2.6 Main Code 5
2.7 Test Code – TODO 5
2.7.1 Blocking Uart-Receive 5
2.7.2 Adding Printing 5
2.7.3 Line editing 5

1 Introduction

Ce document est la trace du travail réalisé en cours de M2M en M2 Génie Informatique à l'IM^{2}AG.

2 Step0

Cette section regroupe le travail réalisé sur les premières semaines de TP à partir de l'archive `Step0.zip`.

2.1 Preface by Pr. Olivier Gruber

2.2 QEMU

QEMU est un hyperviseur de type 2, il permet d'executer des machines virtuelles avec ou sans émulation matérielle. Ici,l'émulation est nécessaire pour exécuter les logiciels compilés pour architecture arm sur mon ordinateur (architecture x86_64).

pour executer une machine la commande utilisée est :

```
qemu-system-arm -M versatilepb -m 1M
```

quand qemu est lancé Ctrl-a c donne accès au prompt qemu avec info qtree, on obtient la configuration materielle détaillée

2.3 GNU Debugger

Pour debugger un programme tournant sur qemu, il faut démarrer la machine qemu avec les options souhaitées + -gdb tcp : :1234 -S et exécuter gdb-multiarch dans un autre terminal puis fournir le fichier elf pour charger la table des symboles avec files , puis l'instruction target remote localhost :1234. afin de se connecter au "serveur gdb" de qemu.

GDB cheatsheet - page 1

Running

`# gdb <program> [core dump]`
Start GDB (with optional core dump).

`# gdb --args <program> <args…>`
Start GDB and pass arguments

`# gdb --pid <pid>`
Start GDB and attach to process.

`set args <args...>`
Set arguments to pass to program to be debugged.

`run`
Run the program to be debugged.

`kill`
Kill the running program.

Breakpoints

`break <where>`
Set a new breakpoint.

`delete <breakpoint#>`
Remove a breakpoint.

`clear`
Delete all breakpoints.

`enable <breakpoint#>`
Enable a disabled breakpoint.

`disable <breakpoint#>`
Disable a breakpoint.

Watchpoints

`watch <where>`
Set a new watchpoint.

`delete/enable/disable <watchpoint#>`
Like breakpoints.

<where>

`function_name`
Break/watch the named function.

`line_number`
Break/watch the line number in the current source file.

`file:line_number`
Break/watch the line number in the named source file.

Conditions

`break/watch <where> if <condition>`
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

`condition <breakpoint#> <condition>`
Set/change the condition of an existing break- or watchpoint.

Examining the stack

`backtrace`
`where`
Show call stack.

`backtrace full`
`where full`
Show call stack, also print the local variables in each frame.

`frame <frame#>`
Select the stack frame to operate on.

Stepping

`step`
Go to next instruction (source line), diving into function.

`next`
Go to next instruction (source line) but don't dive into functions.

`finish`
Continue until the current function returns.

`continue`
Continue normal execution.

Variables and memory

`print/format <what>`
Print content of variable/memory location/register.

`display/format <what>`
Like „print“, but print the information after each stepping instruction.

`undisplay <display#>`
Remove the „display“ with the given number.

`enable display <display#>`
`disable display <display#>`
En- or disable the „display“ with the given number.

`x/nfu <address>`
Print memory.
n: How many units to print (default 1).
f: Format character (like „print“).
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

GDB cheatsheet - page 2

Format

a	Pointer.
c	Read as integer, print as character.
d	Integer, signed decimal.
f	Floating point number.
o	Integer, print as octal.
s	Try to treat as C string.
t	Integer, print as binary (*t* = „two“).
u	Integer, unsigned decimal.
x	Integer, print as hexadecimal.

<what>

`expression`
Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).

`file_name::variable_name`
Content of the variable defined in the named file (static variables).

`function::variable_name`
Content of the variable defined in the named function (if on the stack).

`{type}address`
Content at *address*, interpreted as being of the C type *type*.

`$register`
Content of named register. Interesting registers are $esp (stack pointer), $ebp (frame pointer) and $eip (instruction pointer).

Threads

`thread <thread#>`
Chose thread to operate on.

Manipulating the program

`set var <variable_name>=<value>`
Change the content of a variable to the given value.

`return <expression>`
Force the current function to return immediately, passing the given value.

Sources

`directory <directory>`
Add *directory* to the list of directories that is searched for sources.

`list`
`list <filename>:<function>`
`list <filename>:<line_number>`
`list <first>,<last>`
Shows the current or given source context. The *filename* may be omitted. If *last* is omitted the context starting at *start* is printed instead of centered around it.

`set listsize <count>`
Set how many lines to show in „list“.

Signals

`handle <signal> <options>`
Set how to handle signles. Options are:

(no)print: (Don't) print a message when signals occurs.

(no)stop: (Don't) stop the program when signals occurs.

(no)pass: (Don't) pass the signal to the program.

Informations

`disassemble`
`disassemble <where>`
Disassemble the current function or given location.

`info args`
Print the arguments to the function of the current stack frame.

`info breakpoints`
Print informations about the break- and watchpoints.

`info display`
Print informations about the „displays“.

`info locals`
Print the local variables in the currently selected stack frame.

`info sharedlibrary`
List loaded shared libraries.

`info signals`
List all signals and how they are currently handled.

`info threads`
List all threads.

`show directories`
Print all directories in which GDB searches for source files.

`show listsize`
Print how many are shown in the „list“ command.

`whatis variable_name`
Print type of named variable.

2.4 Makefile

1. La toolchain est arm-non-eabi. Ce sont les "options" utilisées pour la compilation afin de compier pour une architecture cible différente de celle de la machine de compilation.
2. VersatileAB et VersatilePB sont 2 machines arm émulées par QEMU. Elle sont de la famille Versatile AB signifie Application Baseboard et PB : Platform Baseboard (cf : elinux page)
3. Un linker script est un fichier de configuration de l'éditeur de lien qui consiste en la construction d'un fichier assembleur contenant l'ensemble des parties de code compilées séparément avec l'insertion des adresses mémoires pour les saut de fonctions et les variables globales (source : Wikipedia). L'option -T permet de définir le fichier linker script à utiliser.
4. Le linker script kernel.ld définit l'adresse de début du programme à 0x10000 et écrit le programme startup.s à cet endroit, puis écrit le reste du code à la suite et écrit les variables globales (*.data).
5. On convertit le kernel en binaire afin que le processeur puisse l'exécuter. (Plus d'infos dans le man)
6. l'option -g de gcc et de arm-none-eabi-as permet de générer les informations nécessaires pour déboguer le programme.
7. L'option "-nostdlib" indique au compilateur de ne pas utiliser la librairie standard ni le startup standard. C'est nécessaire car on veut déployer notre application sur du matériel qui n'a pas d'OS donc pas la stdlib.
8.
9. On ne peut pas utiliser printf car c'est une fonction de la librairie standard stdio.h, hors cette librairies n'est pas présente (cf 7.).

2.4.1 Linker Script

On traduit kernel.elf en kernel.bin pour transformer le fichier texte en binaire exécutable par le CPU de la machine QEMU.

2.4.2 ELF Format

2.5 Startup Code

2.6 Main Code

2.7 Test Code – TODO

2.7.1 Blocking Uart-Receive

2.7.2 Adding Printing

2.7.3 Line editing