

EMBEDDED OPERATING SYSTEMS

Embedded Linux on Beaglebone Black

Device tree

- A flexible method to define hardware components
 - Static data, not binary / code
- Specification: 0.4 <https://www.devicetree.org>
- Represents a system
 - As a collection of components
 - Joined together in a hierarchy
 - Tree data structure with nodes
 - That describe the characteristics of said device
- Linux uses device tree for:
 - Platform identification
 - Runtime configuration
 - Device population

Device trees – properties

- Node
 - node-name@unit-address
 - Eg. cpu@0, memory@0x2000
- Properties of nodes
 - compatible
 - model
 - phandle
 - status
 - #address-cells, #size-cells
 - reg

DT prop: compatible

- Decides compatibility of drivers
- Contains 1/more strings
- Defines specific programming model of device
- Recommended format
 - “manufacturer, model”
- Example
 - compatible = “arm-a8, beaglebone”, “arm-a7”;***
 - The kernel first searches for drivers for ARM-A8 Beaglebone
 - If no driver found, it then goes for ARM-A7

DT prop: model

- String specifying manufacturer's model
- Recommended format
 - “manufacturer, model”
- Example:
model = “ti, beaglebone black”;

DT prop: phandle

- Specifies a unique numeric identifier for a node
 - Used by other nodes to “point” to this node

- Example:

```
pic@0x1000
{
    phandle = <1>;
    interrupt-controller;
    reg = <0x1000 0x3000>;
};

another-node@0x2100
{
    interrupt-parent = <1>;
}.
```

DT prop: status

- Indicates operational status of the device node
- Values for status:
 - “okay”
 - Device operational
 - “disabled”
 - Device presently not operational; could be activated
 - “reserved”
 - Device operational; but should not be used
 - “fail”
 - Device failed; unlikely to get repaired
 - “fail-sss”
 - *sss: device-specific string*

DT props: address-cells, size-cells

- Used by nodes having children
 - #address-cells
 - Number of u32 cells needed to encode address in the child's "reg" property
 - #size-cells
 - Number of u32 cells needed to encode size in the child's "reg" property
- Example
 - #address-cells = <1>;***
 - #size-cells = <1>;***

DT prop: reg

- Describes addresses of device's resources
 - As offsets on its parent bus
- Example
 - Assume a device in an SoC has 2 blocks of registers
 - 32-byte block at offset 0x3000
 - 64-byte block at offset 0x5000
 - reg = <0x3000 0x20 0x5000 0x40>;***

Device tree example

```
/dts-v1/;
/{
    model = "TI AM335x BeagleBone";
    compatible = "ti,am33xx";
    #address-cells = <1>;
    #size-cells = <1>;
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a8";
            device_type = "cpu";
            reg = <0>;
        };
    };
    memory@0x80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>; /* 512 MB */
    };
};
```

Toolchains

- Set of tools that
 - Compiles source code into an executable
 - That can run on the target system
- Includes
 - Compiler
 - Assembler
 - Linker
 - Runtime libraries
- Toolchain depends on
 - Target CPU architecture
 - Big or little endian operation
 - Floating point support
 - ABI (Application Binary Interface)

API and ABI

- API
 - Application Programming Interface
 - Calling convention for software components at source level
 - Usually documented as function signatures in header files
- ABI
 - Application Binary Interface
 - Calling convention for passing function call parameters in compiled code (binary level)
 - Could include:
 - How parameters are passed (register/stack)
 - Who cleans parameters from stack (caller/callee)
 - Where the return value is placed
 - How exceptions propagate

ARM “ABI”

- Generally, CPU architectures have a common ABI
 - Except ARM
- ARM had OABI *old*
- Now, ARM has Extended ABI (EABI)
 - If floating point is supported in the hardware
 - EABIHF (hard float)

GNU toolchain

- Components:
 - Binutils
 - Linker, Assembler, etc.
 - GNU Compiler Collection (GCC)
 - Common frontend for C, C++, Fortran, Go, etc.
 - Produces assembly object code
 - C library (glibc)
 - Standardized API based on POSIX specification
 - *Others*
 - *Make, Bison, m4, GDB (debugger), Autotools*

GNU toolchain utils

- gcov
 - Code coverage tool
- gprof
 - Profiler
- gdb
 - Debugger
- objcopy, objdump
 - Object code utils
- readelf
 - Parse ELF binaries
- strip
 - Strip debug symbols
- strings
 - Display printables from binaries
- ar
 - Archiver, creates static libraries
- as
 - Assembler, generates assembly code
- cpp
 - C++ compiler frontend
- gcc
 - C compiler frontend
- ld
 - Linker
- nm
 - Lists symbols in object files

GNU C library components

- **libc**
 - Main C library
 - Contains POSIX functions (printf, open, close, ...)
 - Usually linked in **automatically** by *gcc*
- **libm**
 - Provides math functions (sin, cos, asin, exp, log, ...)
 - Has to be linked in by ***-lm***
- **libpthread**
 - Contains all threading functions (*pthread_*)
 - Has to be linked in by ***-lpthread***
- **librt**
 - Real-time extensions like async io, shared memory
 - Has to be linked by ***-lrt***

Path for libraries ***/usr/lib/x86-linux-gnu/libc.[a,so]****

Linking with libc libraries

- Using the libm library
 - Create a C program that calls the *sin()* function
 - Compile it using *gcc -lm math.c -o math.out*
 - Run *./math.out*
 - Examine *math.out* library linkage using “*ldd*”
 - This is “dynamic” linkage – libraries are pulled in by the loader (*ld-linux*) when *math.out* executes

```
$ ldd ./math.out
linux-vdso.so.1 (0x00007ffe9459a000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f391bb3d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f391b800000)
/lib64/ld-linux-x86-64.so.2 (0x00007f391bc39000)
```

Linking – static and dynamic

- Library code can be linked in 2 ways

- Static

- No dependence on libs on system
 - All lib code pulled into executable
 - Executable size becomes large

\$ gcc -static math.c -lm -o math.out.static

- Dynamic

- Dependence on system libs
 - Library code stays on the system
 - Executable size is smaller

\$ gcc math.c -lm -o math.out.dynamic

- Check executable size and ldd output

```
$ ldd math.out.*
math.out.dynamic:
    linux-vdso.so.1 (0x00007ffcbe3a4000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc9bdfel000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc9bdc00000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fc9be0dd000)
math.out.static:
    not a dynamic executable

$ size math.out.*
   text    data     bss     dec      hex filename
   1583     624         8    2215     8a7 math.out.dynamic
 805972   23240   23016 852228   d0104 math.out.static
```

Linking with our own library

- Static
 - Create object code
 - Archive them using *ar* and *ranlib* to create *.a*
 - Compile against *.a* in *static* mode to get *a.out*
 - Run *a.out* (check *ldd*, *size*)
- Dynamic
 - Create object code using *-fPIC* flag
 - Create *.so* using *shared* mode
 - Export *LD_LIBRARY_PATH*
 - Run *a.out* (check *ldd*, *size*)

Toolchain types

- Native
 - Toolchain runs on same type of system it generates code/executable for
 - Usually used for desktops, servers, workstations
 - Example
 - *gcc (Intel) on Linux on x86 laptop creates a.out for x86*
- Cross
 - Toolchain runs on different system type
 - Code/executable generated on different system type
 - Allows development on powerful systems
 - Example
 - *arm-linux-gcc on Linux on x86 generating code for embedded ARM-based target*

GNU toolchain naming

- GNU uses a prefix for each tool in the toolchain
 - Tuple of 3-4 components separated by dashes
 - CPU architecture (and endian-ness)
 - arm, mips, ppc, etc.
 - el (little endian) / eb (big-endian)
 - Vendor (usually, omitted)
 - Target kernel
 - linux
 - Target OS (userspace component)
 - gnu, musl, etc.
 - ABI: eabi, eabihf, etc.
- Find out using: `$ gcc -dumpmachine`
- Examples
 - ***arm-linux-gnueabihf-gcc***
 - ***arm-linux-gnueabihf-ar***

Toolchain for Beaglebone

- We use GCC for ARM using the EABIHF
*\$ sudo apt-get install **gcc-arm-linux-gnueabi**hf*
- Differentiate between, get details
\$ gcc -v
\$ arm-linux-gnueabi-gcc -v
- Get gcc prefixes
\$ gcc -dumpmachine
\$ arm-linux-gnueabi-gcc -dumpmachine

Compile and cross-compile

- Write a hello-world program
- Compile it for x86
 - `$ gcc a.c -o a.out.x86`
- Cross-compile it for Beaglebone
 - `$ arm-linux-gnueabi-gcc a.c -o a.out.arm`
- Use “*file*” command
 - What differences do you notice?
- Run both on x86
 - What do you observe?
 - Do they run?
- Transfer *a.out.arm* to Beaglebone (*use scp*)
 - Run it on Beaglebone
 - What is the observation?
- Repeat static and dynamic lib linking exercises on Beaglebone
 - Use gcc on Beaglebone

THANK YOU!