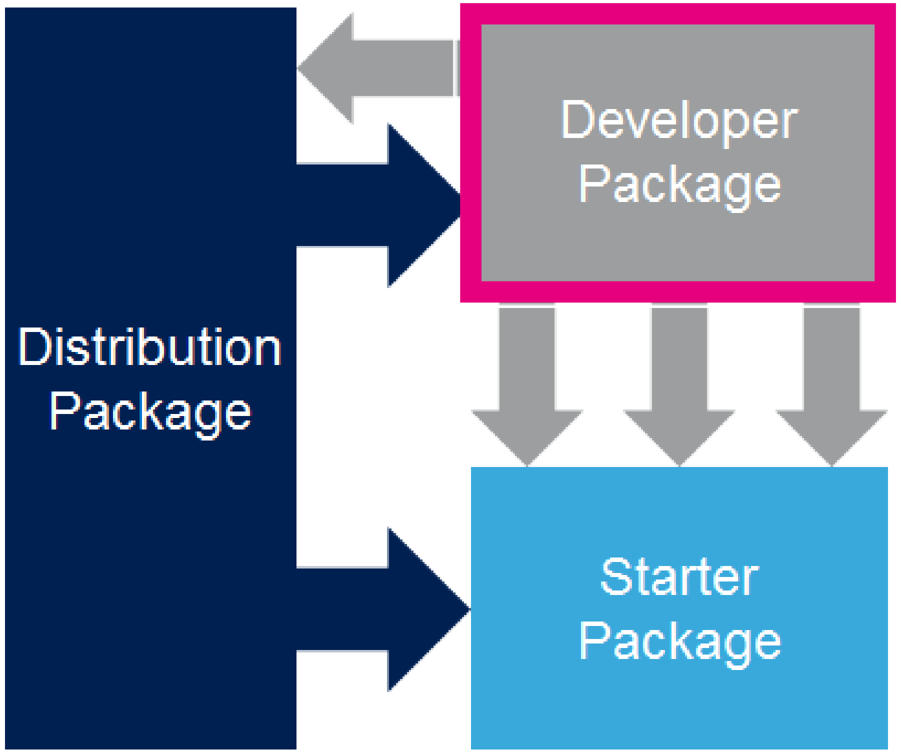
Openstlinux : developer package workshop

Bossen WU

Developer package

- The developer package is a SDK that can be used to generate (TF-A, OPTEE, U-boot and linux) binaries, DTB and customer application.
- The result can be directly used in the starter package or/and integrated inside the Distribution package.
- The SDK used by the developer has been generated by the Distribution package so the customer application can use the services of the package selected in the distribution Package.



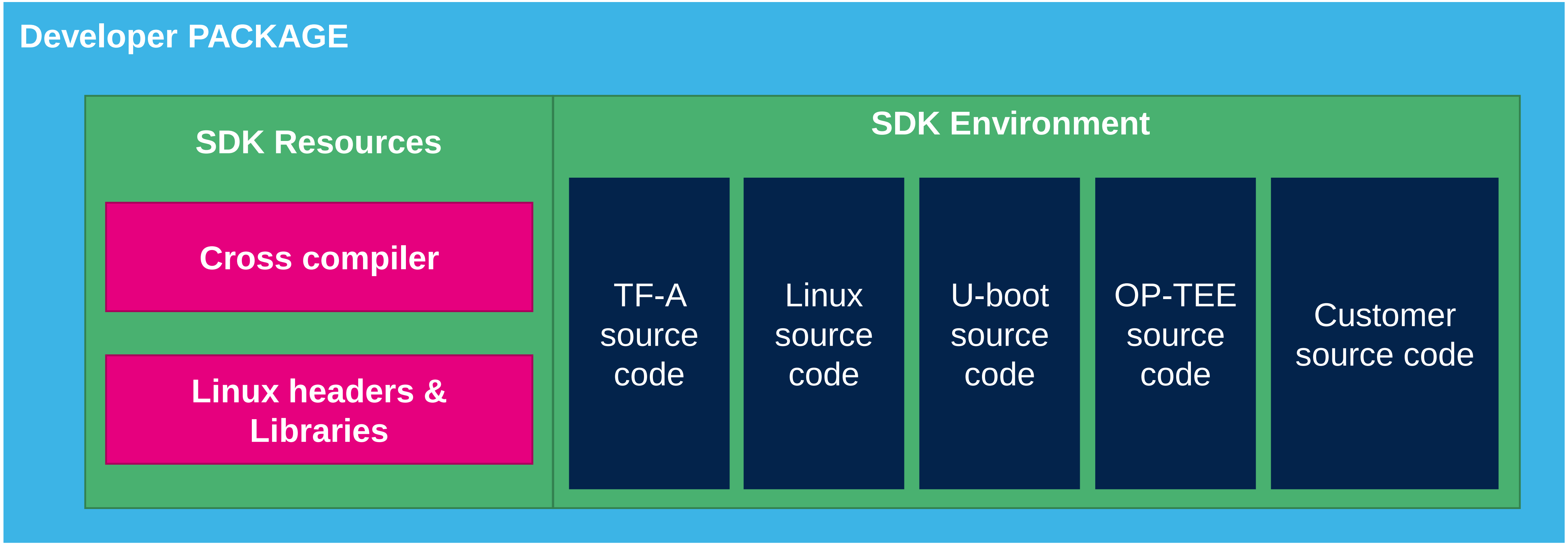
Developer package : installation

- Please follow the wiki page:
  https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package
- Same developer package for all the boards.
- Instead of using package from the wiki download the source code from the Github.
- Compile each binaries and update your board with the result.

life.augmented

Developer package

- The developer package is a SDK environment where we can build software for the STM32MP1 Target



Sdk

- Read the SDK wiki page :
  - https://wiki.st.com/stm32mpu/wiki/SDK_for_OpenSTLinux_distribution
- A SDK is linked to the rootfs/userfs used on the target. If some packages are added in the system the SDK has to be updated. This not a requirement to build TF-A, u-boot, Optee and the kernel source code but it’s very important to build a linux application.
- Have a look inside the sdk folder installation.
  - Where is the gcc cross compiler? Look the sysroot folder, what is the sysroot?

Device tree

- All software layer configuration inside the OpenSTLinux used the device tree.
- Each code source package embedded the documentation about all the possible configuration. In case of trouble the best way is to check the driver.
- As example in the PMIC datasheet there is a feature to keep a SMPS/LDO “ON” during the PMIC reset, we can activate this feature thnks to the device tree, find this device tree property to use it in TF-A.

Tf-a

- Have a look in the TF-A wiki page :
  - https://wiki.st.com/stm32mpu/wiki/STM32MP15_TF-A
- Modify the TF-A source to add a trace. You add a trace inside the ./plat/st/stm32mp1/bl2_plat_setup.c in the function print_reset_reason()
- Build the TF-A :

  unset LDFLAGS

  unset CFLAGS

  make ARM_ARCH_MAJOR=7 ARCH=aarch32 PLAT=stm32mp1 AARCH32_SP=sp_min DTB_FILE_NAME=stm32mp157c-<board>.dtb DEBUG=1
- Load the generated binary and boot the board.
- Open the fdts folders and have a look on device tree

life.augmented

U-boot

- Have a look in the U-boot wiki page :
  - https://wiki.st.com/stm32mpu/wiki/STM32MP15_U-Boot.
  - https://www.denx.de/wiki/U-Boot
- Add a trace with your name inside the U-boot source code in common/board_info.c, show_board_info()
- Re-Build U-boot using :

  make stm32mp15_trusted_defconfig

  make DEVICE_TREE=stm32mp157c-dk2 all
- Load the generated binary and boot the board.

life.augmented

Kernel

- Add a trace with your name inside the Kernel source code.
- Follow steps in
  https://github.com/STMicroelectronics/meta-st-stm32mp/blob/thud/recipes-kernel/linux/linux-stm32mp/README.HOW_TO.txt
- Re-Build kernel :

  cd <directory to kernel source code>

  mkdir -p ../build

  make ARCH=arm O="$PWD/../build" multi_v7_defconfig fragment.config*

  cd <directory to kernel source code>

  - * Build kernel images (uImage and vmlinux) and device tree (dtbs)

    make ARCH=arm uImage vmlinux dtbs LOADADDR=0xC2000040 O="$PWD/../build"

  - * Build kernel module

    make ARCH=arm modules O="$PWD/../build"

Kernel

- * Generate output build artifacts

  make ARCH=arm INSTALL_MOD_PATH="$PWD/../build/install_artifact" modules_install O="$PWD/../build"

  mkdir -p $PWD/../build/install_artifact/boot/

  cp $PWD/../build/arch/arm/boot/uImage $PWD/../build/install_artifact/boot/

  cp $PWD/../build/arch/arm/boot/dts/st.dtb $PWD/../build/install_artifact/boot/*

- load the generated binary and boot the board..

Refer to https://github.com/STMicroelectronics/meta-st-stm32mp/blob/thud/recipes-kernel/linux/linux-stm32mp/README.HOW_TO.txt

7. Update software on board:

Kernel

- Use the menu config to add the debug inside the DMA driver :
  - [*] DMA Engine support ->
  -     [*] DMA Engine debugging
  -         [*]   DMA Engine verbose debugging (NEW)
- Rebuild your kernel.
- Have a look on the trace on the kernel (play a sound or a video)

User application

- Build your own application: openstlinux-hands

✓ It will use the IOCTL interface of GPIOLib framework see [GPIOLib overview].

- On top of this article, all you need to know is available in these files:
  - https://elixir.bootlin.com/linux/latest/source/tools/gpio/gpio-event-mon.c
  - https://elixir.bootlin.com/linux/latest/source/tools/gpio/gpio-hammer.c
  - https://elixir.bootlin.com/linux/latest/source/tools/gpio/gpio-utils.c

- Architecture of the main program is this one:

```
/*  Open device: gpiochip0 for GPIO bank A */
/* request GPIO line: GPIO_A_14 for Led switching and GPIO_A_13 for Button activation*/
/*  Start main loop */
while(1) {
   /* read GPIO_A_13 input event */
   /* process the event received  and update the led value */ }
/*  Close device: gpiochip0 */
```

life.augmented

User application

- Note: GPIO_A_14 and GPIO_A_13 are defined in the device tree of the board but not activated by default (status = "disabled"). Do not change it. If you activate them they will be taken by kernel frameworks and not available for your application.
- Try to develop this application using the Developer Package and run it on the board.
  - The interest here is to understand how to structure both your makefile with your application and do some debug around it.

✓ One solution is provided in your hands-on package in "openstlinux-hands-appli" directory. Please check it.

User application

Lab 2 TIPS:

- To understand "patsubst" function (commonly used in Makefiles): https://www.gnu.org/software/make/manual/html_node/Text-Functions.html
- Makefiles have to be customized matching your needs and your habits of development but the structure is quite similar from one to another.
- You can see the different options of a local make with the tab key:

```
PC $> make <tab>
All     clean     install     openstlinux-hands
```

- Any time you open a new terminal, you will need to run the SDK env setup source to position your tool chain variable. A quick way to check you are with right setup is:

```
PC $> env | grep ARCH
ARCH=arm
```

- Standard debugger for application development is GDB. See [GDB] and especially [gdbgui] usage.

life.augmented

Kernel driver

- The purpose here is to do the same use case as previous one but within the kernel.
- We will need for that the kernel in full source (as provided in the Developer Package delivery).
- Many different ways are possible:
  - built-in kernel module = kernel internal module (part of kernel uimage)
  - kernel external module = module is dynamically loaded after kernel init
  - out-of-tree external module = module source is outside the kernel source tree (over way is in-tree external module)

- We will focus on the last one for this lab.

Kernel driver

- [STM32MP1 Developer Package] article is providing an example of such use case (this gives you a template of your program). The new driver name can be "push_led_driver".
- A specific node will first need to be added to your device tree (device tree of your board):
  - It will initialize the 2 gpios used: GPIO_A_14 and GPIO_A_13. These labels will be used by gpiolib interface.
  - It will create also the link between these gpios and the driver (« compatible » field).
  - You can check [How to control a GPIO in kernel space] for help.

- Your program may need to use these functions:
  - gpiolib: « devm_gpiod_get », « gpiod_to_irq », « gpiod_set_value » and « gpiod_get_value »
  - irqchip: « devm_request_any_context_irq » to handle interrupt generated by GPIO_A_13 button

Kernel driver

- With https://elixir.bootlin.com/linux/latest/source, you will easily find with the identifier search field:
  - The definitions of the functions and associated structures.
  - Many examples of usage provided by kernel community drivers.
  - The best way to use them and manage errors.

- Try to develop this driver using the Developer Package and run it on the board.
  - The interest here is to understand how to structure both your makefile with your driver, the interconnection with the device tree and the relationship with the kernel source.

✓ One solution is provided in your hands-on package in "openstlinux-hands-driver" directory. Please check it.

Kernel driver

TIPS:

- The delivered Linux kernel source code is a git extraction, then by default your module will get a dirty version (not compatible with the Starter Package version). See the "README.HOW_TO.txt" file (".scmversion" file).
- After you updated the ".scmversion" file, you will need to rebuild your kernel first then your module to make it clean.
- Don’t forget to reboot the board to take device tree modifications into account.
- To make your external module probed at boot, use "vi" to create .conf file like below :

```
Board $> vi /etc/modules-load.d/openstlinux-hands-driver.conf
Board $> cat /etc/modules-load.d/openstlinux-hands-driver.conf
openstlinux-hands-driver
```

Practicing school: developer package 11/11

? Q&A session (if possible with support contact):

- Duration: 1h
- Objective: Check all the topics mentioned in Developer Package section were well done and there was no issues. Mandatory steps:
  - SDK installation and use of cross compilation to upload embedded SW programs
  - rebuild the kernel and device tree then update the Starter Package (rapid cycle)
  - makefile organization related to the Developer Package (SDK + source code packages)
- Attendees: developers in Embedded Linux

Releasing your creativity with the STM32

/STM32 @ST_World community.st.com

Famous video here

www.st.com/stm32