KEIm-CVSoC

RTSP (Real Time Streaming Protocol)

サーバーの構築と実行

目次

1. はじめに

この資料は、KEIm-CVSoC に Debian 9.0 (Stretch) をインストールし、カメラモジュールからの映像を RTSP (Real Time Streaming Protocol) で配信するシステムを作成する手順と実行方法を紹介します。

ボードの設定やピン・アサイン等については **KEIm-CVSoC ハードウェアマニュアル** や **KEIm-CVSoC 開発キット スタートアップガイド** を参照してください。

各種マニュアルおよび **BSP**（Board Support Package）**リファレンスデザイン**（GSRD,GHRD）、**RTSP Server サンプル ソースコード**および**デザイン**は弊社製品サイト https://kd-group.co.jp/product/keim-cvsoc/ からユーザー登録する事で取得可能となっています。

なお、**RTSP Server 用サンプルデザイン SD カード・イメージ**を使用する場合は、必要なデザインおよびソフトウェアはインストール済みですので、「13-3. カメラモジュールでの RTSP サーバー実行」から、また **BSP リファレンスデザイン（GSRD）SD カード・イメージ**を使用する場合は、「11. RTSP サーバーの構築」から操作を行ってください。

2. 使用機材

- KEIm-CVSoC ： https://kd-group.co.jp/product/keim-cvsoc/
  （IO Board 及びカメラモジュール、電源等付属品含む）
- SD Card (16GB 以上 Class 10 を推奨）
- Ethernet
- Quartus Prime 18.1 update 1

< オプション >

- モニター (1280x720@60p , HDMI 入力)
- キーボード & マウス (USB)
- USB カメラ
- iEi 社製 Mustang-M2BM-MX2

3. FPGA デザイン

弊社ダウンロードサイトから KEIm-CVSoC 開発キットリファレンスデザイン（GHRD: SoC FPGA Quartus プロジェクトファイル）を入手し、ローカルフォルダに展開します。本資料では */KEIm-CVSoC* フォルダを作成し、そこに展開しています。

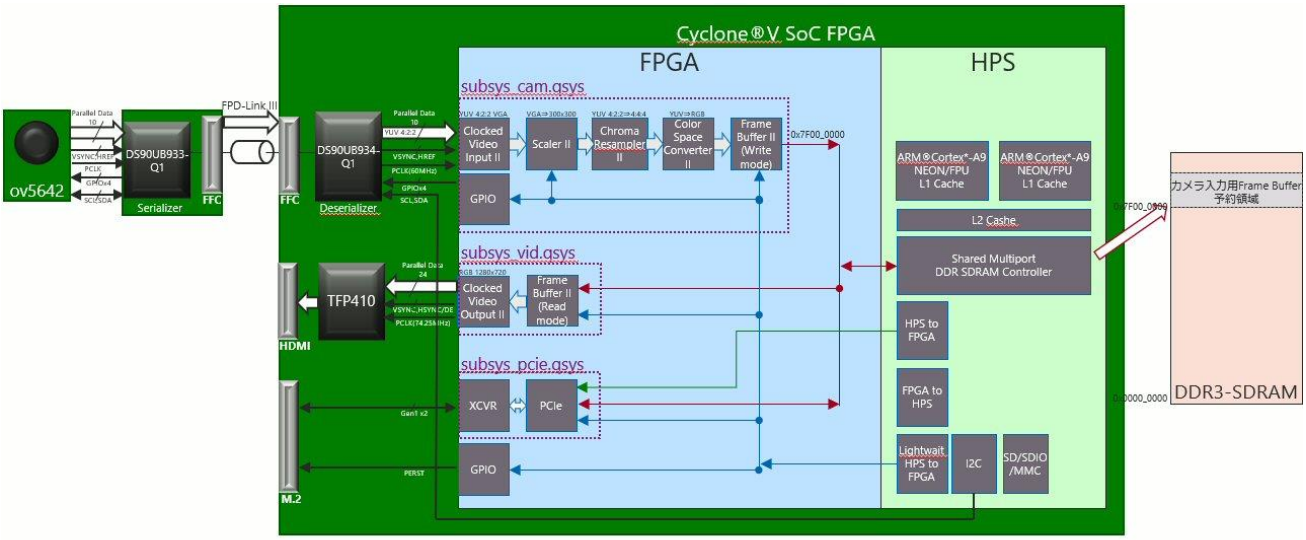
FPGA には主に下記の3つのインターフェースを実装します。

- カメラモジュールからの映像を入力するパラレル I/F
- デスクトップ画面を出力するパラレル I/F

- M.2 に接続するための PCIe I/F

上記の機能ブロックは Platform Designer のサブシステムとして実装されていますので、必要が無い場合比較的簡単に切り離したり、他の機能に影響を及ぼさず変更する事が可能です。

下記にブロック図を示します。



※主要なブロックのみを記載しています。

図 3. ブロック図

FPGA のコンパイルが完了したら SoC EDS Command Shell を起動してください。

FPGA デザインからは HPS 起動時に FPGA をコンフィグレーションするための RBF (Raw Binary File) と、プリローダ、uboot ファイル等が生成されます。

3-1. RBF の作成

SoC EDS Command Shell で Quartus プロジェクトの */output_files* フォルダに移動して下記のコマンドを実行します。RBF (Raw Binary File) が “soc_system.rbf” のファイル名で生成されます。

```
$ quartus_cpf -c -o bitstream_compression=on soc_system.sof soc_system.rbf
```

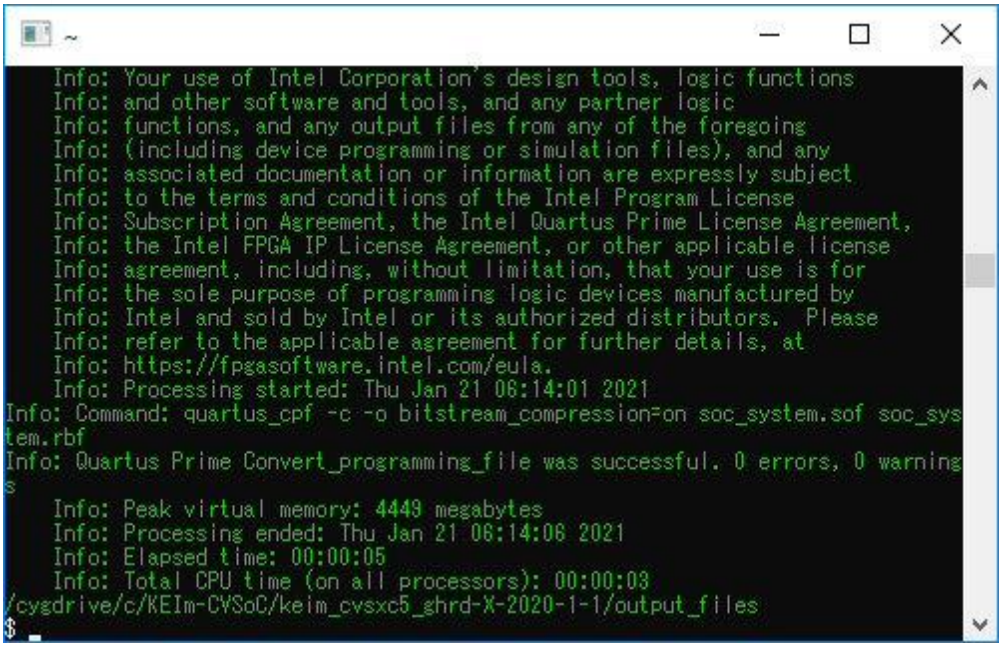


図 3-1. quartus_cpf の実行

3-2. プリローダと uboot の作成

3-2-1. BSP Editor によるソースファイルの Generate

SoC EDS Command Shell で BSP-Editor を起動します。

```
$ bsp-editor &
```

メインメニューの File ⇒ New HPS BSP を起動し、Preloader settings directory: に

<Project Folder>¥hps_isw_handoff¥soc_system_hps_0 を選択します。

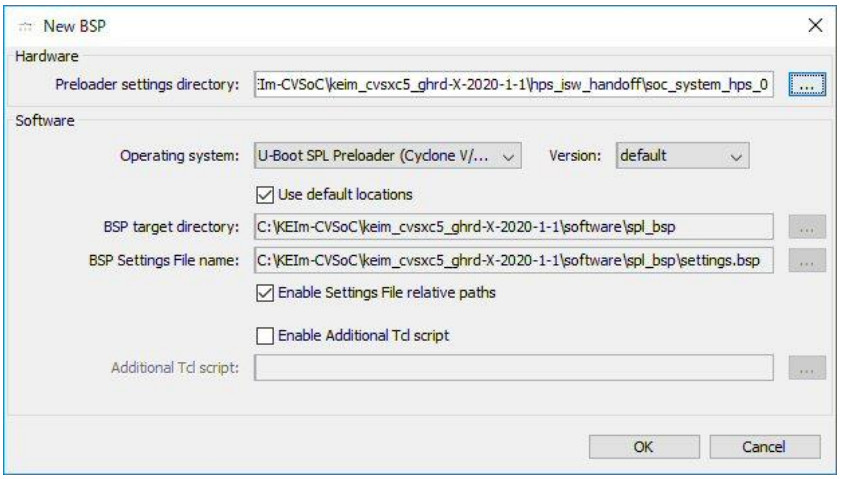


図 3-2-1-1. New BSP 設定

BOOT_FROM_SDMMC にチェックが入っている事を確認して、Generate をクリックすると software フォルダ以下にプリローダと uboot を生成するためのファイル一式が展開されます。

Generate が完了したら Exit をクリックしてウィンドウを閉じてください。

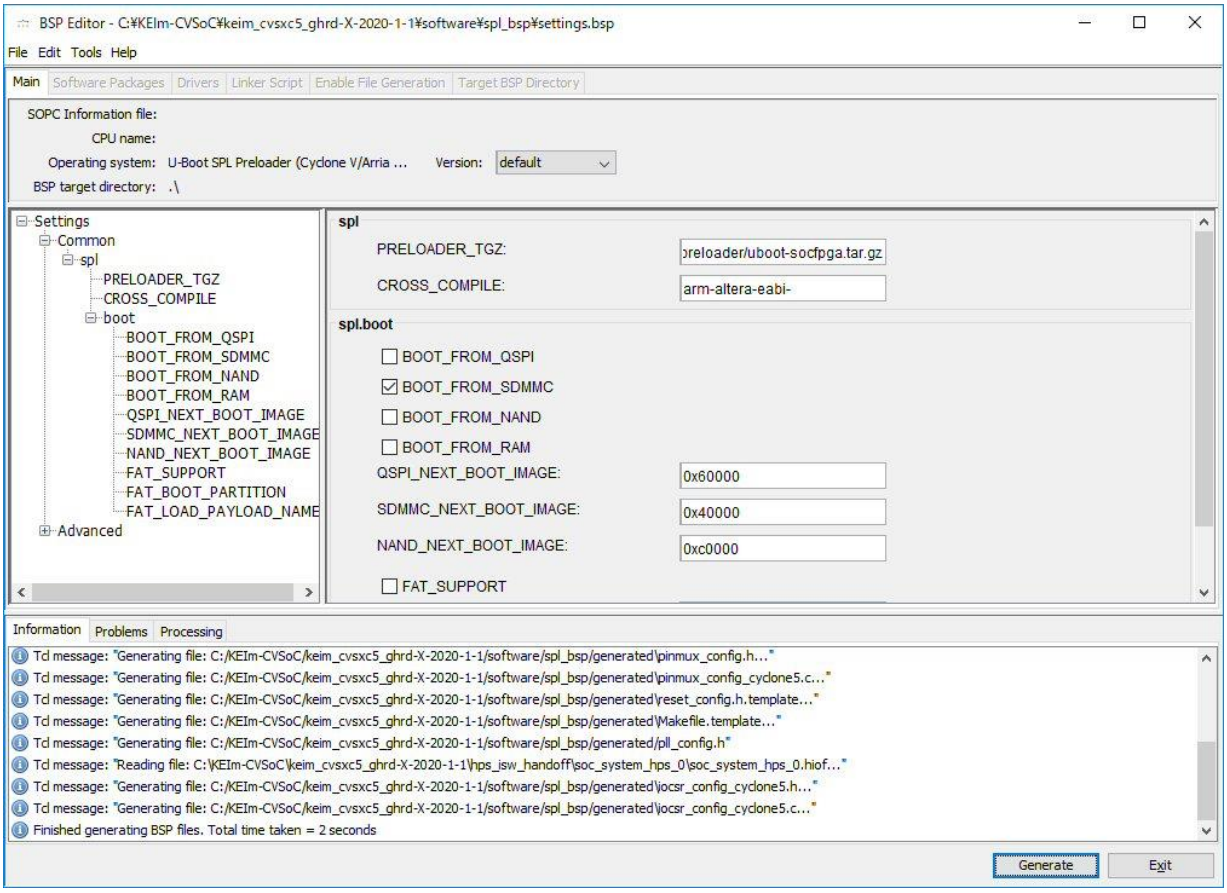


図 3-2-1-2. BSP エディタによる Generate

3-2-2. プリローダの make

コンパイル済の Quartus プロジェクトフォルダの下 *software/spl_bsp* に移動して make します。

```
$ cd <Project Folder>/software/spl_bsp/
$ make
```

もし、下記のようなエラーが発生したら [Unable to make preloader in Windows 10] を参考に変更を行い、software フォルダを削除した後、再度、**3-2-1. BSP Editor によるソースファイルの Generate** から行ってください。

```
$ make
tar zxf /cygdrive/c/intelFPGA/18.1/embedded/host_tools/altera/preloader/uboot-so
cfpga.tar.gz
tar: Error opening archive: Failed to open '/cygdrive/c/intelFPGA/18.1/embedded/
host_tools/altera/preloader/uboot-socfpga.tar.gz'
make: *** [uboot-socfpga/.untar] Error 1
```

図 3-2-2-1. プリローダの make エラー

また、”Permission denied” のエラーが出る場合は、SoC EDS Command Shell を管理者権限で起動してから make を実行してください。

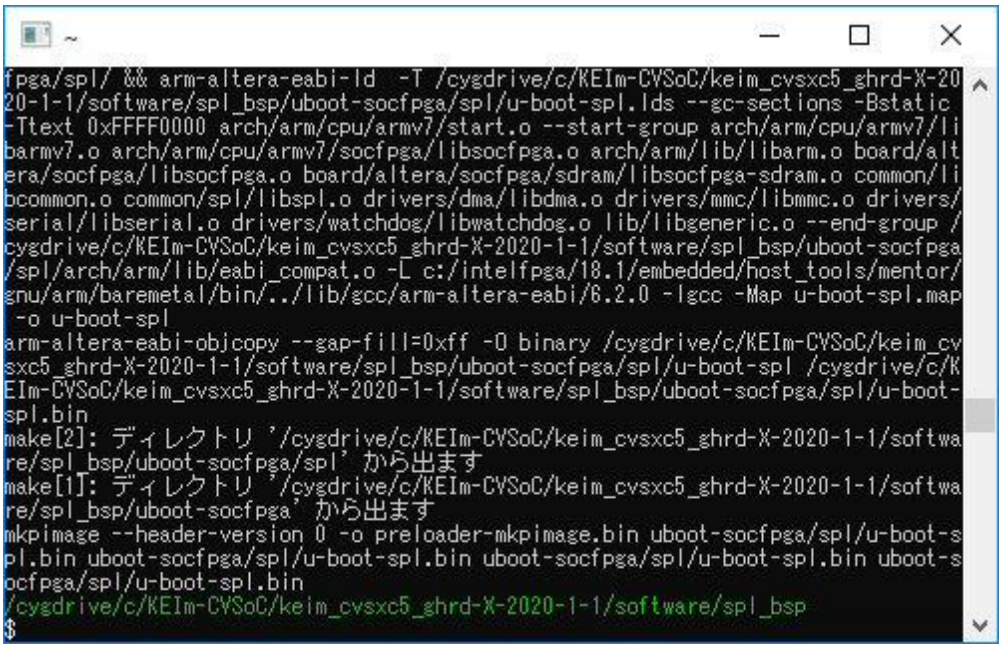


図 3-2-2-2. プリローダの生成

<Project Folder>/software/spl_bsp フォルダに "preloader-mkpimage.bin" が生成されます。

3-2-3. uboot の make

そのまま続けて make します。

```
$ make uboot
```

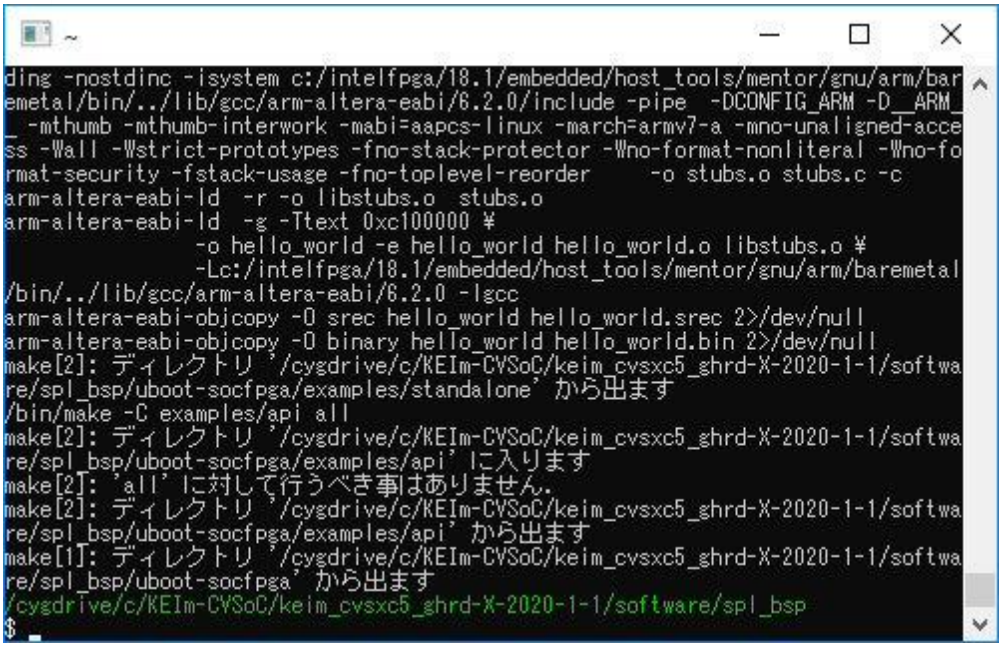


図 3-2-3. u-boot の作成

<Project Folder>/software/spl_bsp/uboot-socfpga フォルダに "u-boot.img" が生成されます。

3-3. デバイス・ツリーの作成

3-3-1. sopc2dts を使った dts ファイルの生成

Linux を起動する際、デバイス・ドライバを適正な状態で起動するためのデバイス・ツリーを作成します。

SoC EDS Command Shell でコンパイル済の Quartus プロジェクトフォルダに移動して sopc2dts を実行します。

```
$ sopc2dts --input soc_system.sopcinfo --output soc_system.dts --type dts --board soc_system_board_info.xml --board hps_common_board_info.xml --bridge-removal all --clocks
```

プロジェクトフォルダに "soc_system.dts" が生成されます。この dts (Device Tree Source) ファイルは現状の HPS 設定の原型のデバイス・ツリーとなりますので、ここから KEIm-CVSoC の付加機能のための変更を加える必要があります。

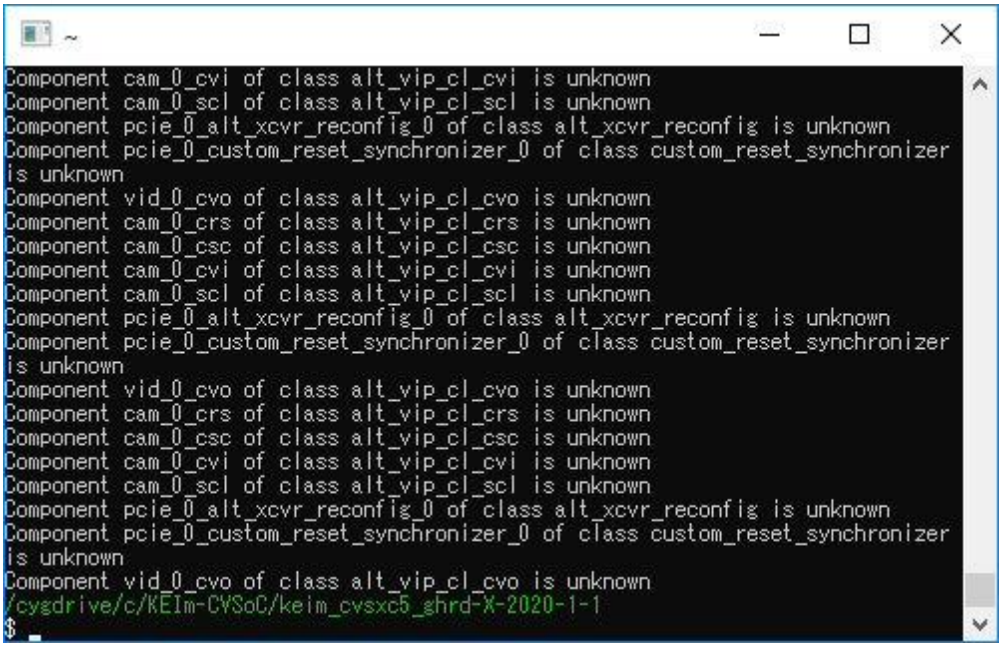


図 3-3-1. dts の生成

変更箇所としては下記のとおりです。

- カメラモジュールからの映像データを SDRAM に書き込む領域を ユーザースペース I/O (以降 UIO) ドライバで確保する
- FPGA に実装された IP コアのレジスタにアクセスする領域を UIO ドライバで確保する
- フレームバッファドライバが、デスクトップ出力用の Frame Buffer II IP core と、カメラ入力用の Frame Buffer II IP core を混同しないようにする
- PCIe ドライバの記述を修正する

3-3-2. カメラ用フレームバッファ領域を記述

memory 下のセンテンスに下記を追記します。

```
reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        frame_buf0: frame_buf@0 {
                compatible = "shared-dma-pool";
                reg = <0x7F000000 0x00800000>;
                no-map;
        };
};
```

ここでは、カメラ入力用フレームバッファを物理アドレスとサイズで領域を予約します。

アドレスの 0x7F00_0000 は、カメラ入力用 Frame Buffer II IP core (subsys_cam > vfb)設定の Frame buffer memory base address で設定されたものです。

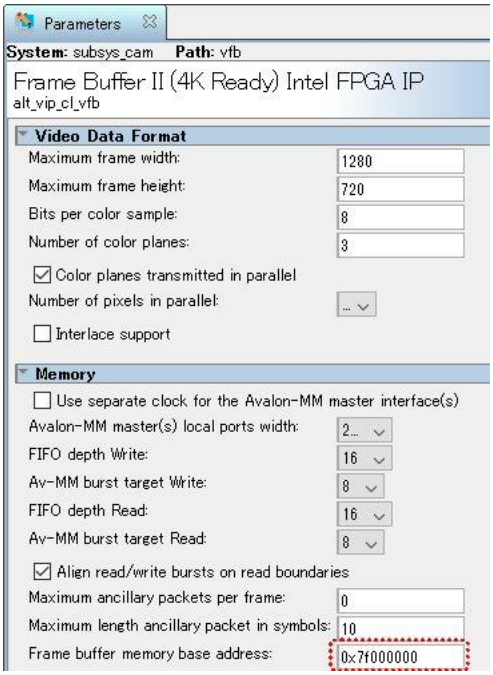


図 3-3-2. Frame Buffer II IP Core Parameters

最大 1280x720 を想定していますので 1280*720*3byte*3frame = 829,400 byte = 0x7E9_000 ≒ 0x80_000 を確保していますが、必要に応じてアドレスやサイズは変更可能です。

3-3-3. UIO ドライバ・アクセス領域を記述

上記のすぐ下に下記を追記します。

```
uio0@0xf7000000 {
        compatible = "generic-uio";
        reg = <0x7F000000 0x00800000>;
};

uio1@0xff200000 {
        compatible = "generic-uio";
        reg = <0xff240000 0x1000>;
        interrupt-parent = <&hps_0_arm_gic_0>;
        interrupts = <0 46 4>;
};
```

ここでは、UIO ドライバでアクセスできる物理アドレスとその領域サイズを宣言しています。
uio0 はカメラ入力映像用フレームバッファ、uio1 は FPGA に実装された IP コアのレジスタに HPS からアクセスするためのものです。

3-3-4. カメラ用フレームバッファの変更

本実装では、カメラ入力の他にデスクトップ(HDMI)出力用に2つの Frame Buffer II IP core を使用しています。

Linux Kernel では出力用フレームバッファのドライバは標準で用意されていますが、カメラ入力用に使用されるフレームバッファは用意されていないため Frame Buffer II IP core を誤認識しないようにカメラ入力用のデバイス名を変更しておく必要があります。

```
cam_0_vfb: vip@0x100040000 {
    compatible = "altr,vip-18.1", "altr,vip-frame-writer-2.0";
    reg = <0x00000001 0x00040000 0x00000040>;
    interrupt-parent = <&hps_0_arm_gic_0>;
    interrupts = <0 46 4>;
    clocks = <&cam_0_clk_0>;
```

“altr,vip-frame-buffer-2.0“ を別の名前 (例:"altr,vip-frame-**writer**-2.0") に変更します。

3-3-5. PCIe のクロックを修正(オプション)

sopc2dts で生成された dts ファイルをそのまま dtb に変換すると、PCIe のクロック部分でエラーになりますので、エラーにならないように修正します。

```
pcie_0_msgdma_0: msgdma@0x1000140c0 {
    compatible = "altr,msgdma-18.1", "altr,msgdma-1.0";
    reg = <0x00000001 0x000140c0 0x00000020>,
          <0x00000001 0x000140e0 0x00000010>;
    reg-names = "csr", "descriptor_slave";
    interrupt-parent = <&hps_0_arm_gic_0>;
    interrupts = <0 44 4>;
    clocks = <0>;
}; //end msgdma@0x1000140c0 (pcie_0_msgdma_0)
```

clocks = <&pcie_0_custom_reset_synchronizer_0>; を clocks = **<0>**; に変更します。

3-3-6. dts ファイルを dtb ファイルに変換

SoC EDS Command Shell でコンパイル済のプロジェクトフォルダに移動して dtc を実行します。

前項で変更した dts ファイルを dtb (Device Tree Blob) ファイルに変換します。

```
$  dtc -I dts -O dtb -o soc_system.dtb soc_system.dts
```

プロジェクトフォルダに "soc_system.dtb" が生成されます。

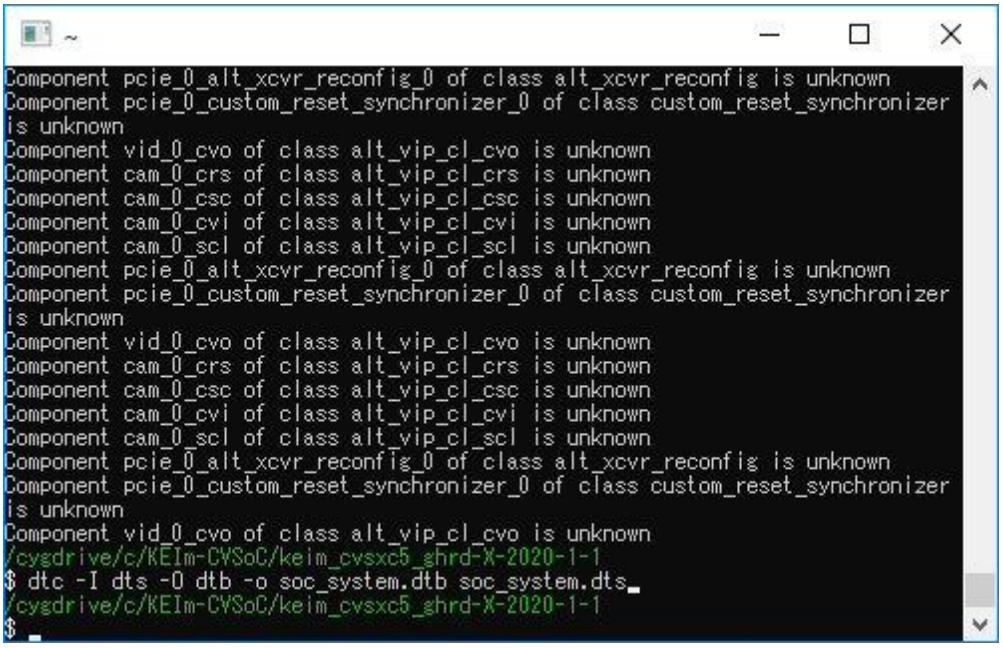


図 3-3-6. soc_system.dtb の作成

4. ブート・スクリプトの作成

4-1. ブートスクリプト・テキスト・ファイルの作成

スクリプト・テキスト・ファイル (boot.script) を新規に作成し、下記のテキストを追加します。

```
fatload mmc 0:1 $fpgadata soc_system.rbf;
fpga load 0 $fpgadata $filesize;
setenv fdtimage soc_system.dtb;
setenv mmcboot 'setenv bootargs console=ttyS0,115200 uio_pdrv_genirq.of_id=generic-uio root=${mmcroot} rw
rootwait;bootz ${loadaddr} - ${fdtaddr}';
run bridge_enable_handoff;
mw.l 0xFF220140 0x1 1;
mw.l 0xFF220140 0x0 1;
run mmcload;
run mmcboot;
```

※1

※2

※1 "uio_pdrv_genirq.of_id=generic-uio" は Linux 起動時に UIO ドライバを自動的に組み込むためのもの。

※2 PCIe (M.2) のハードリセットを行うもので、Mustang-M2BM-MX2 を使用する場合には必須。(オプション)

4-2. ブートスクリプト・ファイルの作成

mkimage を使ってブートスクリプト・ファイルを作成します。

```
$ mkimage -A arm -O linux -T script -C none -a 0 -e 0 -n "My script" -d boot.script u-boot.scr
```

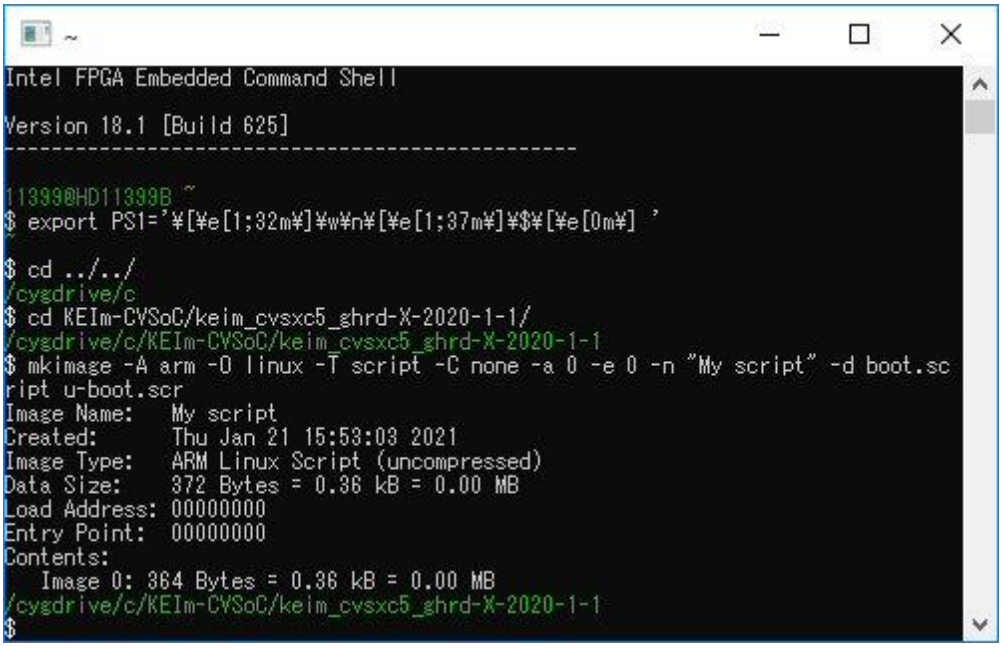


図 4-2. u-boot.scr の生成

プロジェクトフォルダに "u-boot.scr" が生成されます。

5. Linux Kernel

Linux Kernel の make は Linux 環境でクロスコンパイルします。本資料では x86 PC に Ubuntu 18.04 をインストールして操作を行います。x86 系以外のプラットフォームをお使いの場合や Debian 系以外のディストリビューションをお使いの場合は、ファイルの入手先やコマンド等が異なる場合がありますのでご注意ください。

Linux Kernel ソースは Intel® が提供している Github から入手しますが、保守メンテナンス等により本資料に記載されているバージョンが入手出来ない場合もありますので、その場合は入手出来たバージョンに読み替えて操作するか、tag を参照して過去のバージョンを入手する必要があります。

また、Kernel をコンパイルする場合、Toolchain が必要となりますが、本資料では Linaro プロジェクトが提供している GNU Toolchain を使用しています。もし、他の Toolchain をお使いになる場合は、それに合わせた設定を行ってください。

5-1. Toolchain のダウンロードと展開

Linaro プロジェクトが提供している GNU Toolchain をダウンロードします。

ダウンロードするフォルダはどこでも構いませんが、本資料では *~/KEIm-CVSoC/* としています。

```
$ cd ~/KEIm-CVSoC
$ wget https://releases.linaro.org/components/toolchain/binaries/7.5-2019.12/arm-linux-gnueabihf/gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf.tar.xz
$ tar Jxfv gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf.tar.xz
```

```
ihf/7.5.0/f951
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/liblto_plugin.so.0.0.0
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/cc1plus
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/liblto_plugin.so
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/lto1
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/plugin/
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/plugin/gengtype
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/mkinstalldirs
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/fixinc.sh
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/fixincl
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/mkheaders
~/KEIm-CVSoC$
```

図 5-1. Toolchain をダウンロード

5-2. 環境変数の設定

クロスコンパイル環境のため、環境変数に Toolchain のパスと CPU アーキテクチャを設定します。

```
$ export CROSS_COMPILE=~/KEIm-CVSoC/gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
$ export ARCH=arm
```

5-3. Kernel ソースの取得

Intel® が提供している Github から Linux Kernel ソースを取得します。

ダウンロードするフォルダはどこでも構いませんが、本資料では *~/KEIm-CVSoC/* としています。

```
$ cd ~/KEIm-CVSoC
$ git clone https://github.com/altera-opensource/linux-socfpga
```

```
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/plugin/gengtype
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/mkinstalldirs
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/fixinc.sh
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/fixincl
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf/libexec/gcc/arm-linux-gnueab
ihf/7.5.0/install-tools/mkheaders
~/KEIm-CVSoC$ export CROSS_COMPILE=~/KEIm-CVSoC/gcc-linaro-7.5.0-2019.12-x86_64_
arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
~/KEIm-CVSoC$ export ARCH=arm
~/KEIm-CVSoC$ cd ~/KEIm-CVSoC
~/KEIm-CVSoC$ git clone https://github.com/altera-opensource/linux-socfpga
Cloning into 'linux-socfpga'...
remote: Enumerating objects: 7865441, done.
remote: Total 7865441 (delta 0), reused 0 (delta 0), pack-reused 7865441
Receiving objects: 100% (7865441/7865441), 1.71 GiB | 4.84 MiB/s, done.
Resolving deltas: 100% (6663570/6663570), done.
Checking out files: 100% (65762/65762), done.
~/KEIm-CVSoC$
```

図 5-3. Kernel ソースのダウンロード

5-3-1. Branch の確認

Github には複数の Branch が含まれていますので、その中の1つをチェックアウトします。

ダウンロード・フォルダに移動して Branch の一覧を表示します。

```
$ cd linux-socfpga/
$ git branch -r
```

```
~/KEIm-CVSoC/linux-socfpga$ git branch -r
  origin/HEAD -> origin/socfpga-5.4.74-lts
  origin/socfpga-4.14.126-ltsi-rt
  origin/socfpga-4.14.130-ltsi
  origin/socfpga-5.4.64-lts
  origin/socfpga-5.4.74-lts
  origin/socfpga-5.8
  origin/socfpga-5.9
~/KEIm-CVSoC/linux-socfpga$
```

図 5-3-1. Kernel Branch の確認

5-3-2. チェックアウト

Kernel 5.4.74-lts をチェックアウトします。（バージョンは一例です。異なるバージョンを使用する場合は読み替えてください。）

```
$ git checkout -b test_branch origin/socfpga- 5.4.74-lts
```

5-4. Kernel コンフィグレーションの初期化

Linux Kernel のコンフィグレーションを初期状態にします。

```
$ make socfpga_defconfig
```

```
~/KEIm-CVSoC/linux-socfpga$ make socfpga_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/confdata.o
  HOSTCC  scripts/kconfig/expr.o
  LEX     scripts/kconfig/lexer.lex.c
  YACC    scripts/kconfig/parser.tab.[ch]
  HOSTCC  scripts/kconfig/lexer.lex.o
  HOSTCC  scripts/kconfig/parser.tab.o
  HOSTCC  scripts/kconfig/preprocess.o
  HOSTCC  scripts/kconfig/symbol.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
~/KEIm-CVSoC/linux-socfpga$
```

図 5-4. socfpga_defconfig の実行

5-5. Kernel コンフィグレーションの変更

KEIm-CVSoC を使用する上で必要な設定をメニュー形式で編集します。

```
$ make menuconfig
```

上下キーで項目を選択し、下層のメニューに移動する場合は <Select> を選択した後 'Enter' を、上層のメニューに戻るには 'Esc' または <Exit> を選択した後 'Enter' を押下します。

項目を有効化する場合には 'y' を、無効化する場合には 'n' キーを押下します。

全ての項目の選択が終了し、その設定を確定したい場合、<Save> を選択して 'Enter' を押下します。

menuconfig から抜ける場合は、'Esc' または <Exit> を選択して 'Enter' を押下し続けます。

5-5-1. フレームバッファの有効化

KEIm-CVSoC の IO ボードに実装されている HDMI ポートから映像を出力し、コンソールやデスクトップの出力を有効にします。

設定箇所は下記の階層です。

```
Device Driver
  Graphics support --->
    Frame buffer Drivers --->
      --- Support for frame buffer devices
      <*> Altera VIP Frame Buffer II framebuffer support OF Device
    [*] Bootup logo --->
      --- Bootup logo
      [*] Standard black and while Linux logo
      [*] Standard 16-color Linux logo
      [*] Standard 224-color Linux logo
    Console display driver support --->
      [*] Framebuffer Console support
        [*] Map the console to the primary display device
```

Altera VIP Frame Buffer II framebuffer support OF Device： Frame Buffer II IP コアを使用するデバイス・ドライバを組み込みます。

Bootup logo： 起動時、タックス（Linux オペレーティングシステムの公式マスコットであるペンギンマーク）を表示させます。フレームバッファが正常に動作しているか確認する手段にもなります。

Map the console to the primary display device： 起動ログをディスプレイ出力させる場合に設定します。

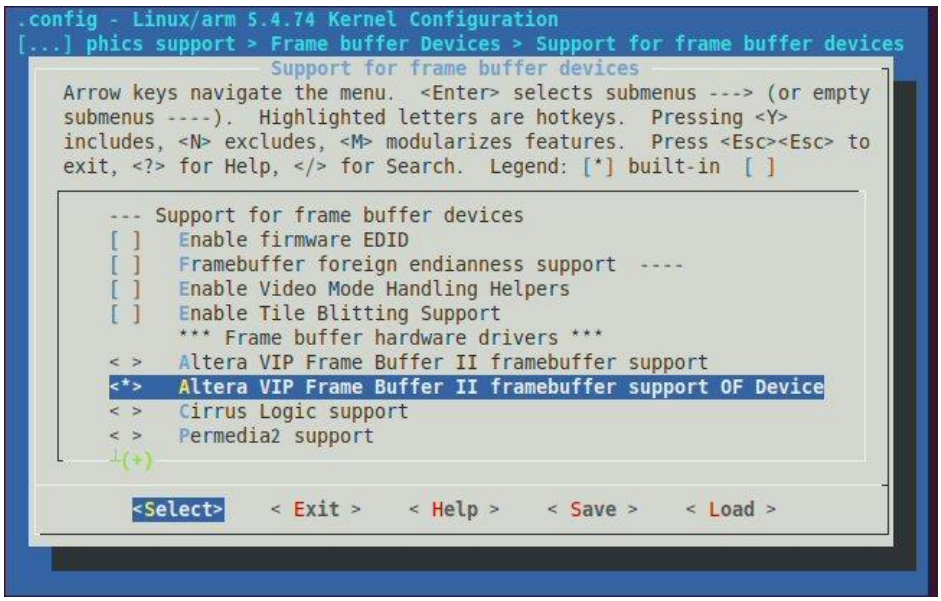


図 5-5-1. フレームバッファの有効化

5-5-2. PCIe (M.2) の有効化（オプション）

KEIm-CVSoC の IO ボードに実装されている M.2 コネクタにデバイスを接続する場合に有効にします。

※ iEi 社製 Mustang-M2BM-MX2 を接続し使用する場合には、下記の設定を行ってください。

```
Device Drivers --->
  [*] PCI support
    [*] Message Signaled Interrupts (MSI and MSI-X)
        PCI controller driver --->
          [*] Altera PCIe controller
          [*]       Altera PCIe MSI feature
  [*] USB support --->
        <*> xHCI HCD (USB 3.0) support
        <*>       Generic xHCI driver for a platform device
```

Altera PCIe controller：Intel® PCIe Controller IP core を使用する場合に有効にします。

Altera PCIe MSI feature：Intel® PCIe Controller IP core の MSI (Message Signaled Interrupts) を有効にします。

xHCI HCD (USB 3.0) support：PCIe デバイスを USB デバイスとして認識させる場合に有効にします。

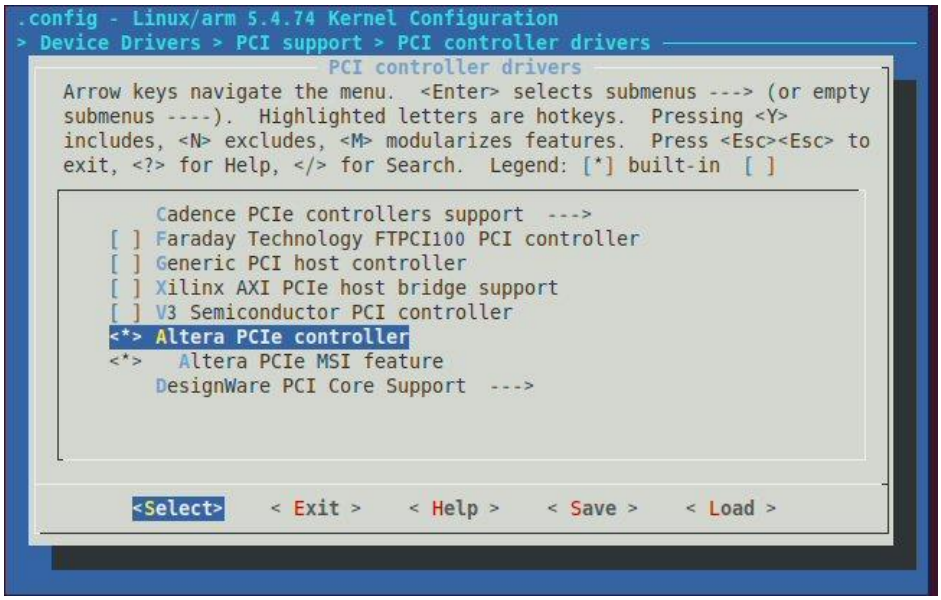


図 5-5-2. PCIe (M.2) の有効化

5-5-3. UVC (USB Video Class) の有効化（オプション）

KEIm-CVSoC の IO ボードに実装されている USB OTG コネクタに Web Camera 等を接続する場合に有効にします。

```
Device Drivers --->
  [*] Multimedia support --->
        [*] Cameras/video grabbers support
        [*] Media USB Adapters --->
              <*> USB Video Class (UVC)
              [*] UVC input events device support
```

Media USB Adapters：USB バスのメディア・ドライバーを有効にします。

UVC input events device support：USB ビデオクラス・デバイスは、ボタンイベントを報告するための入力デバイスを登録します。

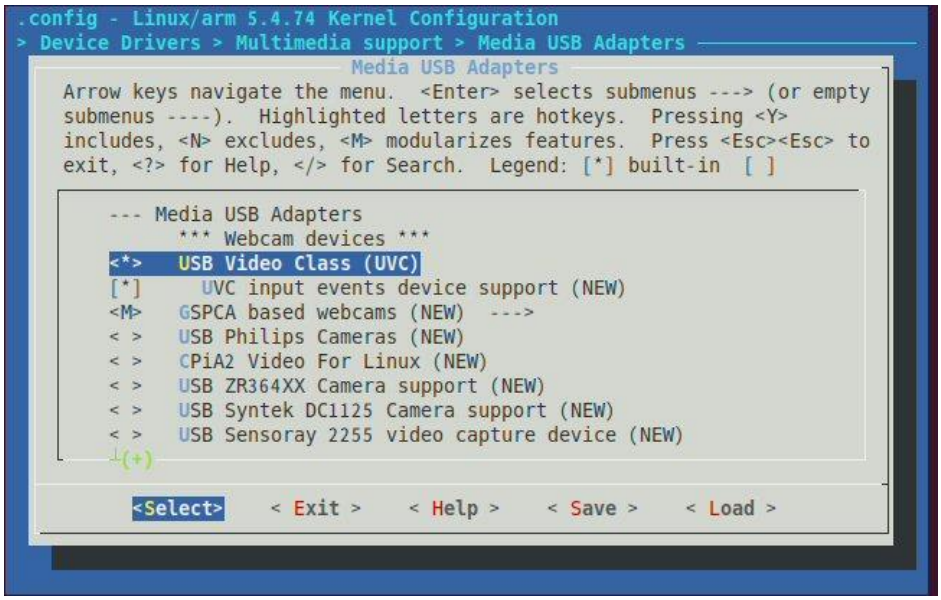


図 5-5-3. UVC (USB Video Class) の有効化

5-5-4. ユーザースペース I/O デバイスの有効化

FPGA に実装されている IP core へのアクセスを ユーザースペース I/O を使って行うため、UIO ドライバを有効にします。

Device Driver

<*> Userspace I/O drivers --->

<*> Userspace I/O platform driver with generic IRQ handling

Userspace I/O platform driver with generic IRQ handling：一般的な割り込み処理コードを含む、ユーザースペース I/O デバイス用のプラットフォームドライバーを組み込みます。

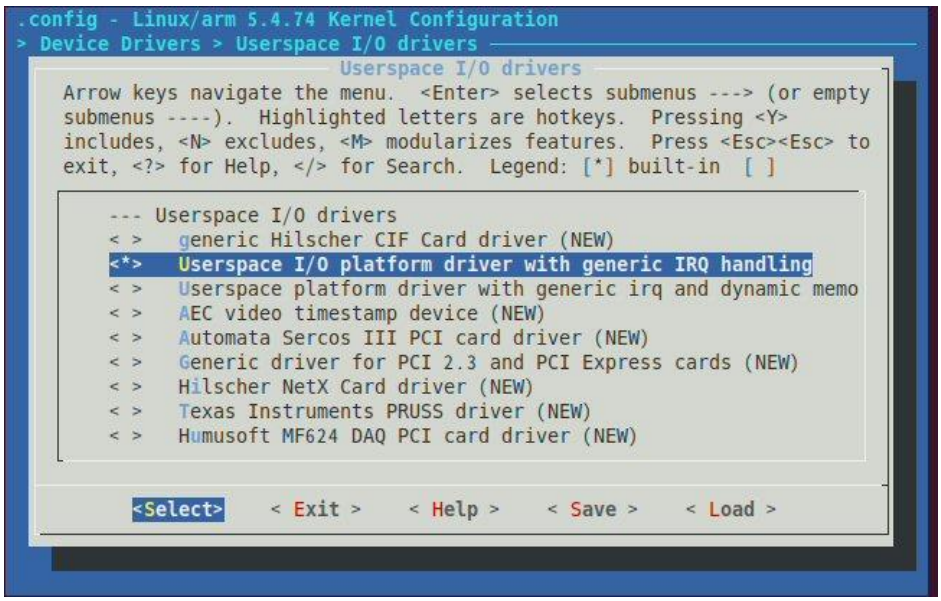


図 5-5-4. ユーザースペース I/O デバイスの有効化

5-5-5. その他の設定

Kernel のバージョンにローカルバージョンを追加しないように、下記の項目を無効にします。

General setup ->

[ ] Automatically append version information to the version strings

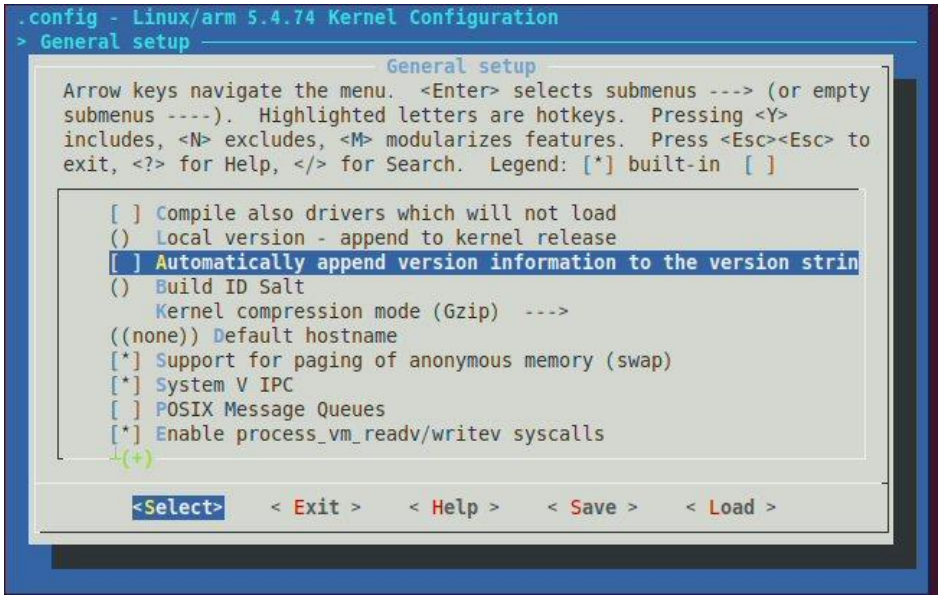


図 5-5-5. その他の設定

5-5-6. コンフィグレーションの保存

下層のメニューの <save> を左右キーで選択し 'Enter' を押下し、確認ダイアログが表示されたら <Ok> を選択して 'Enter' を押下します。

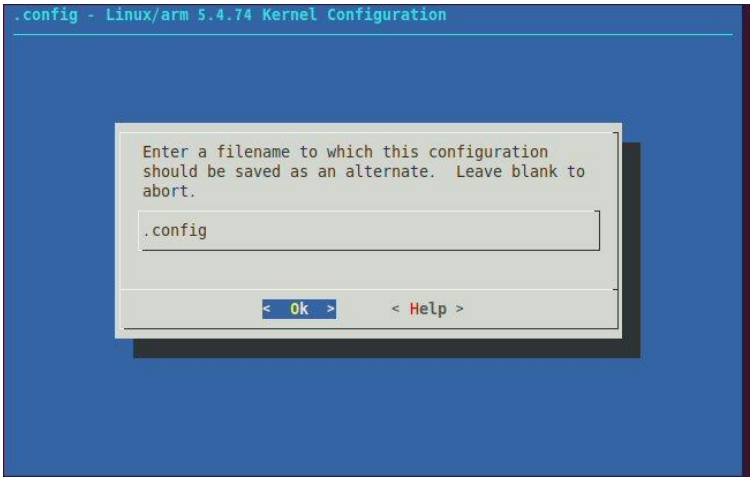


図 5-5-6. コンフィグレーションの保存

保存完了のダイアログが表示されますので、'Enter' を押下して画面を抜けます。

選択画面に戻ったら menuconfig を抜けます。

5-6. Kernel の make

Linux Kernel を make して zImage を作成します。

```
$ make zImage
```

zImage ファイルは *~/KEIm-CVSoC/linux-socfpga/arch/arm/boot* に生成されます。

```
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  LDS     arch/arm/boot/compressed/vmlinux.lds
  AS      arch/arm/boot/compressed/head.o
  GZIP    arch/arm/boot/compressed/piggy_data
  AS      arch/arm/boot/compressed/piggy.o
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  CC      arch/arm/boot/compressed/string.o
  SHIPPED arch/arm/boot/compressed/hyp-stub.S
  AS      arch/arm/boot/compressed/hyp-stub.o
  SHIPPED arch/arm/boot/compressed/lib1funcs.S
  AS      arch/arm/boot/compressed/lib1funcs.o
  SHIPPED arch/arm/boot/compressed/ashldi3.S
  AS      arch/arm/boot/compressed/ashldi3.o
  SHIPPED arch/arm/boot/compressed/bswapsdi2.S
  AS      arch/arm/boot/compressed/bswapsdi2.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
~/KEIm-CVSoC/linux-socfpga$
```

図 5-6. Kernel の make

5-7. Kernel モジュールの生成

ターゲット・システム（動作している KEIm-CVSoC の Linux システム）上でデバイス・ドライバや実行モジュールをコンパイルする際には、Kernel に含まれるソースコードやインクルードファイル等が必要になる場合があり、その時のために Kernel モジュールをあらかじめ生成しておきます。

```
$ make ARCH=arm INSTALL_MOD_PATH=./modroot modules
$ make ARCH=arm INSTALL_MOD_PATH=./modroot modules_install
```

Kernel モジュールは *~/KEIm-CVSoC/linux-socfpga/modroot/lib/modules/* の下に Kernel のバージョンをフォルダ名として生成されます。

5-5-5. その他の設定 で、 ”Automatically append version information to the version strings” をディセーブルにしていますが、これは、ターゲット・システム上でデバイス・ドライバ等をコンパイルした際、互換 Kernel バージョンに “-dirty” が付加され、実行出来なくなってしまう場合があるためです。

ここで生成した Kernel モジュールは、後程、Debian 9.0 Root File System の中に組み込まれる事になります。

```
  CC [M]  drivers/net/ethernet/intel/ixgbe/ixgbe.mod.o
  LD [M]  drivers/net/ethernet/intel/ixgbe/ixgbe.ko
  CC [M]  drivers/net/mdio.mod.o
  LD [M]  drivers/net/mdio.ko
  CC [M]  drivers/nvme/host/nvme-core.mod.o
  LD [M]  drivers/nvme/host/nvme-core.ko
  CC [M]  drivers/nvme/host/nvme.mod.o
  LD [M]  drivers/nvme/host/nvme.ko
~/KEIm-CVSoC/linux-socfpga$ make ARCH=arm INSTALL_MOD_PATH=./modroot modules_ins
tall
  INSTALL drivers/char/hw_random/rng-core.ko
  INSTALL drivers/dma/dmatest.ko
  INSTALL drivers/i2c/algos/i2c-algo-bit.ko
  INSTALL drivers/media/usb/gspca/gspca_main.ko
  INSTALL drivers/misc/altera_ilc.ko
  INSTALL drivers/misc/altera_sysid.ko
  INSTALL drivers/net/ethernet/intel/e1000e/e1000e.ko
  INSTALL drivers/net/ethernet/intel/igb/igb.ko
  INSTALL drivers/net/ethernet/intel/ixgbe/ixgbe.ko
  INSTALL drivers/net/mdio.ko
  INSTALL drivers/nvme/host/nvme-core.ko
  INSTALL drivers/nvme/host/nvme.ko
  DEPMOD  5.4.74+
~/KEIm-CVSoC/linux-socfpga$
```

図 5-7. Kernel モジュールの生成

6. Linux Root File System

KEIm-CVSoC で動作する Linux ディストリビューションは幾つか存在しますが、本資料では Linaro プロジェクトが提供する Debian 9.0 (Stretch) の実装方法を紹介します。

6-1. Root File System のダウンロード

Linaro プロジェクトが提供している Debian 9.0 の Root File System のアーカイブを取得します。

ダウンロードするフォルダはどこでも構いませんが、本資料では 5. 章で使用した~/KEIm-CVSoC/ としています。

```
$ cd ~/KEIm-CVSoC
$ wget https://releases.linaro.org/debian/images/developer-armhf/latest/linaro-stretch-developer-20170706-43.tar.gz
$ sudo tar zxf linaro-stretch-developer-20170706-43.tar.gz
```

Root File System は *~/KEIm-CVSoC/binary* に展開されます。

6-2. Kernel モジュールのコピー

5-7. Kernel モジュールの生成 で作成した Kernel モジュールを Root File System の *lib/modules/* にコピーします。

```
$ cd ~/KEIm-CVSoC/linux-socfpga/modroot/lib/modules/
$ rm 5.4.74+/source
$ sudo cp -Lpr * binary/lib/modules
```

生成された Kernel モジュールのソースファイルはシンボリック・リンクが作成されているだけですので、コピーする際は、”L” オプションを必ず付加してください。

7. SD カード・イメージの作成

KEIm-CVSoC のカードスロットに SD カードを挿入し Linux を起動するためには、一連のファイルを所定のパーティションに書き込んで置く必要があります。

fdisk コマンド等を使って SD カードのパーティションを所定のサイズで作成し、個別にファイルを書き込んで SD カードを作成する事も可能ですが、本資料では、スクリプトを使って一括で SD カード・イメージを作成します。

7-1. 構成ファイルの取得

3. から 6. 章で作成したファイルを ~/KEIm-CVSoC に集めます。

ファイル名	フォルダ	解説
soc_system.rbf	<Project Folder>/output_files	3-1. RBF の作成
preloader-mkpimage.bin	<Project Folder>/software/spl_bsp	3-2-2. プリローダの make
u-boot.img	<Project Folder>/software/spl_bsp/uboot-socfpga	3-2-3. uboot の make
soc_system.dtb	<Project Folder>	3-3-6. dts ファイルを dtb ファイルに変換
u-boot.scr	<Project Folder>	4-2. ブートスクリプト・ファイルの作成
zImage	~/KEIm-CVSoC/linux-socfpga/arch/arm/boot	5-6. Kernel の make
binary/*	~/KEIm-CVSoC	6-2. Kernel モジュールのコピー

7-2. SD カード・イメージ作成スクリプトの取得

Rocketboards.org からスクリプトを取得し、実行属性を付加します。

```
$ cd ~/KEIm-CVSoC
$ wget https://releases.rocketboards.org/release/2020.05/gsrd/tools/make_sdimage_p3.py
$ chmod +x make_sdimage_p3.py
```

7-3. SD カード・イメージ作成

下記を実行して SD カード・イメージを生成します。

```
$ sudo ./make_sdimage_p3.py -f -P preloader-mkpimage.bin,u-boot.img,num=3,format=raw,size=10M,type=A2 -P binary/*,num=2,format=ext3,size=12G -P zImage,u-boot.scr,soc_system.rbf,soc_system.dtb,num=1,format=vfat,size=500M -s 13G -n keim_cvsoc_debian9.img
```

上記のコマンドでは、Root File System を含む ext3 パーティションを 12 GB 確保し、全体では 13GB の SD イメージを作成していますが、必要に応じて増減させる事が可能です。その場合、SD イメージのサイズ (本スプリクトでは 13GB) は、ext3 パーティションのサイズ (本スクリプトでは 12GB) に 510MB 加えたサイズ以上にする必要があります。

~/KEIm-CVSoC に "keim_cvsoc_debian9.img" が作成されます。

8. SD カード・イメージの書き込み

7. SD カード・イメージの作成 で作成した SD カード・イメージ・ファイルを SD カードに書き込みます。

この操作は Linux PC または Windows PC のどちらでも可能ですが、Windows の場合、フリーの書き込みツールをインストールする必要があります。

8-1. Linux PC で書き込む場合

dd コマンドを使用して書き込みます。

```
$ sudo dd if=keim_cvsoc_debian9.img of=/dev/sdb bs=65536
$ sudo sync
```

書き込み先の SD カード・ドライブ (本コマンドでは /dev/sdb) は PC 環境によって異なります。 ls /dev コマンドや fdisk コマンド等で SD カード・ドライブのパスを確認してから実行してください。

ブロックサイズ (本コマンドでは bs=65536) は任意です。省略した場合 512 byte となります。このサイズが小さいと書き込み完了までの時間が長くなりますが、大きく設定し過ぎると PC のメモリを多く消費しますので注意が必要です。

また、dd コマンドが完了してもその時点では書き込みが完了していない場合がありますので sync コマンドで必ず書き込み完了を確認し、更に SD カード・ドライブにインジケータ等があれば点滅していない事を確認した上で、SD カードをアンマウントして抜き出してください。

8-2. Windows PC で書き込む場合

KEIm-CVSoC 開発キットスタートアップガイド の **3.3 SD イメージ書き込み手順** を参照してください。

9. KEIm-CVSoC の起動

8. SD カード・イメージの書き込み で作成した SD カードを KEIm-CVSoC のカードスロットに差し込み起動します。

KEIm-CVSoC 開発キットスタートアップガイド の **3.4 microSD カードの取り出し方と差し込み方** 以降を参照してください。

初回起動時にはターミナル接続は必須です。ターミナルにはブートログが出力され、起動が完了するとコマンド受付待ちのプロンプトが表示されます。

Linaro Debian 9.0 では、ターミナル接続からは root ユーザーとしてログイン無しで入る事ができます。

10. Linux 環境の整備

Linux に最新のパッケージをインストールし、デスクトップ環境を実装します。また、RTSP を実行する上で必要なパッケージのインストールも行います。

10-1.ネットワーク接続の確認

パッケージ等をネットワークからダウンロードしてインストールするためネットワークが外部サイトに接続出来る必要があります。

```
#  ip addr
```

ip コマンドを使って DHCP サーバーから IP アドレスが取得出来ている事を確認します。

固定アドレスを使用する場合は */etc/network/interfaces* に設定する必要がありますが、詳細については本資料では省略します。

10-1-1. プロキシの設定（オプション）

プロキシ・サーバーを経由して外部ネットワークに接続する場合は下記の設定が必要です。

```
#  vi /etc/profile.d/proxy.sh
```

vi エディタで */etc/profile.d/proxy.sh* を開いて下記を記述します。

```
export http_proxy="http://<proxy ip address : port number>"
export https_proxy="http:// <proxy ip address : port number >"
export ftp_proxy="http:// <proxy ip address : port number >"
```

<proxy ip address : port number> にはプロキシ・サーバーの IP アドレスとポート番号を設定します。

```
#  vi /etc/apt/apt.conf
```

vi エディタで */etc/apt/apt.conf* を開いて下記を記述します。

```
Acquire::http::proxy "http:// <proxy ip address : port number>";
Acquire::https::proxy "http:// <proxy ip address : port number>";
```

10-2.upgrade の実行

Linux に最新のパッケージをインストールします。

```
#  apt update
#  apt -y upgrade
```

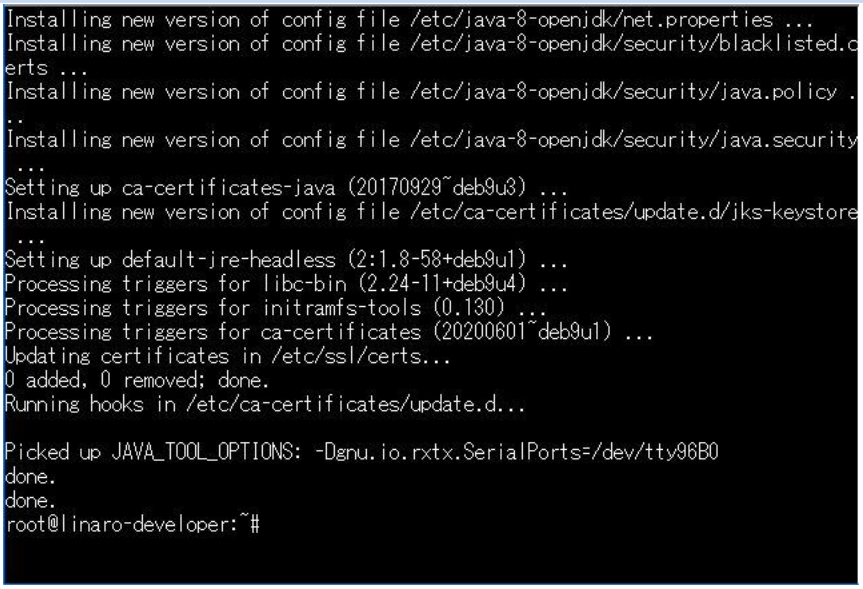


図 10-2. upgrade の実行

10-3.root のパスワード設定

root ユーザーのパスワードを設定します。

```
#  passwd root
```

任意のパスワードを入力します。(同じワードを2回入力します)

図 10-3. root のパスワード設定

10-4.タイムゾーンの設定

日付、時刻を確認し、UTC になっている場合 JST に変更します。

```
#  date
#  timedatectl
#  timedatectl list-timezones | grep -i tokyo
```

設定可能なタイムゾーンに日本時間が存在するか確認します。

```
root@linaro-developer:~# date
Tue Feb  2 04:43:53 UTC 2021
root@linaro-developer:~# timedatectl
      Local time: Tue 2021-02-02 04:43:59 UTC
  Universal time: Tue 2021-02-02 04:43:59 UTC
        RTC time: Tue 2021-02-02 04:43:59
       Time zone: Etc/UTC (UTC, +0000)
 Network time on: yes
NTP synchronized: yes
 RTC in local TZ: no
root@linaro-developer:~# timedatectl list-timezones | grep -i tokyo
Asia/Tokyo
root@linaro-developer:~#
```

図 10-4-1. タイムゾーンの確認

```
#  timedatectl set-timezone Asia/Tokyo
#  date
#  timedatectl
```

タイムゾーンを Asia/Tokyo に変更し、日付、時刻が JST に変更されている事を確認します。

```
root@linaro-developer:~# timedatectl set-timezone Asia/Tokyo
root@linaro-developer:~# date
Tue Feb  2 13:45:54 JST 2021
root@linaro-developer:~# timedatectl
      Local time: Tue 2021-02-02 13:46:00 JST
  Universal time: Tue 2021-02-02 04:46:00 UTC
        RTC time: Tue 2021-02-02 04:46:01
       Time zone: Asia/Tokyo (JST, +0900)
 Network time on: yes
NTP synchronized: yes
 RTC in local TZ: no
root@linaro-developer:~#
```

図 10-4-2. タイムゾーンの設定

10-5.SSH 接続の設定

SSH 接続する際、パスワードで接続できるようにします。

```
#  vi /etc/ssh/sshd_config
```

vi エディタで /etc/ssh/sshd_config を開いて変更します。

PermitRootLogin の項目を “prohibit-password“ から “yes” に変更し、コメントアウト(#) を削除します。

```
# Logging
#SyslogFacility AUTH
#LogLevel INFO

# Authentication:

#LoginGraceTime 2m
PermitRootLogin yes
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10

#PubkeyAuthentication yes
-- INSERT --                                          32,20         14%
```

図 10-5. SSH 接続の設定

10-6.キーボードの設定

デスクトップで使用するキーボードを設定します。日本語キーボードを使う場合キー配列が異なりますので設定を変更する必要があります。

```
#  vi /etc/default/keyboard
```

vi エディタで /etc/default/keyboard を開いて変更します。

XKBMODEL を “pc105” から "jp106" に、XKBLAYOUT を “us” から "jp" に変更します。

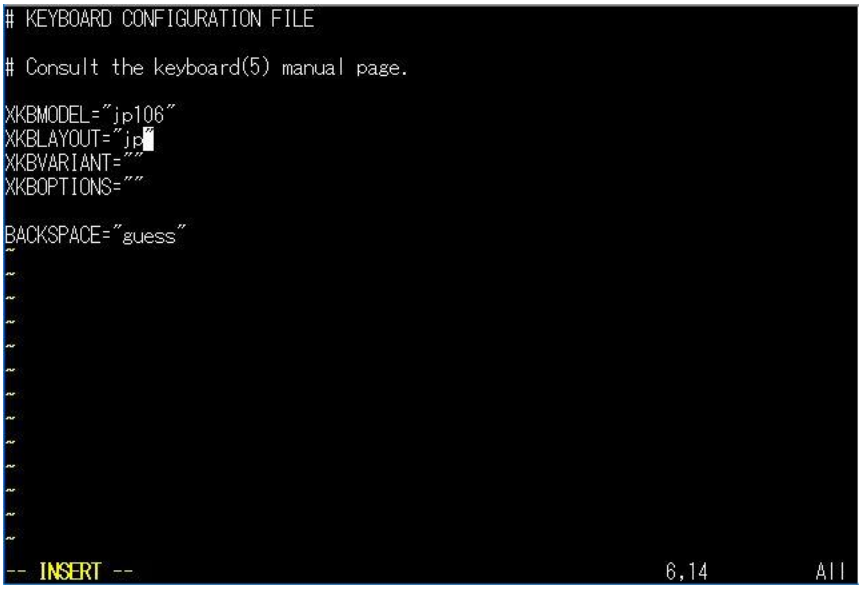


図 10-6. キーボードの設定

10-7.LXDE のインストール

LXDE をインストールしてデスクトップ環境を構築します。

```
#  apt install -y task-lxde-desktop
```

インストールが完了したら reboot してください。

```
aspell-autobuildhash: processing: en [en-variant_2].
aspell-autobuildhash: processing: en [en-w_accents-only].
aspell-autobuildhash: processing: en [en-wo_accents-only].
aspell-autobuildhash: processing: en [en_AU-variant_0].
aspell-autobuildhash: processing: en [en_AU-variant_1].
aspell-autobuildhash: processing: en [en_AU-w_accents-only].
aspell-autobuildhash: processing: en [en_AU-wo_accents-only].
aspell-autobuildhash: processing: en [en_CA-variant_0].
aspell-autobuildhash: processing: en [en_CA-variant_1].
aspell-autobuildhash: processing: en [en_CA-w_accents-only].
aspell-autobuildhash: processing: en [en_CA-wo_accents-only].
aspell-autobuildhash: processing: en [en_GB-ise-w_accents-only].
aspell-autobuildhash: processing: en [en_GB-ise-wo_accents-only].
aspell-autobuildhash: processing: en [en_GB-ize-w_accents-only].
aspell-autobuildhash: processing: en [en_GB-ize-wo_accents-only].
aspell-autobuildhash: processing: en [en_GB-variant_0].
aspell-autobuildhash: processing: en [en_GB-variant_1].
aspell-autobuildhash: processing: en [en_US-w_accents-only].
aspell-autobuildhash: processing: en [en_US-wo_accents-only].
Processing triggers for menu (2.1.47+b1) ...
Processing triggers for libreoffice-common (1:5.2.7-1+deb9u11) ...
root@linaro-developer:~#
```

図 10-7-1. LXDE のインストール

reboot 後、HDMI コネクタに接続されたモニターにログインウィンドウが表示されますが、

ログインユーザー: linaro 、パスワード: linaro でログインできます。

初回ログイン時には下記のようなメッセージが表示されますが、"yes" をクリックして続けてください。

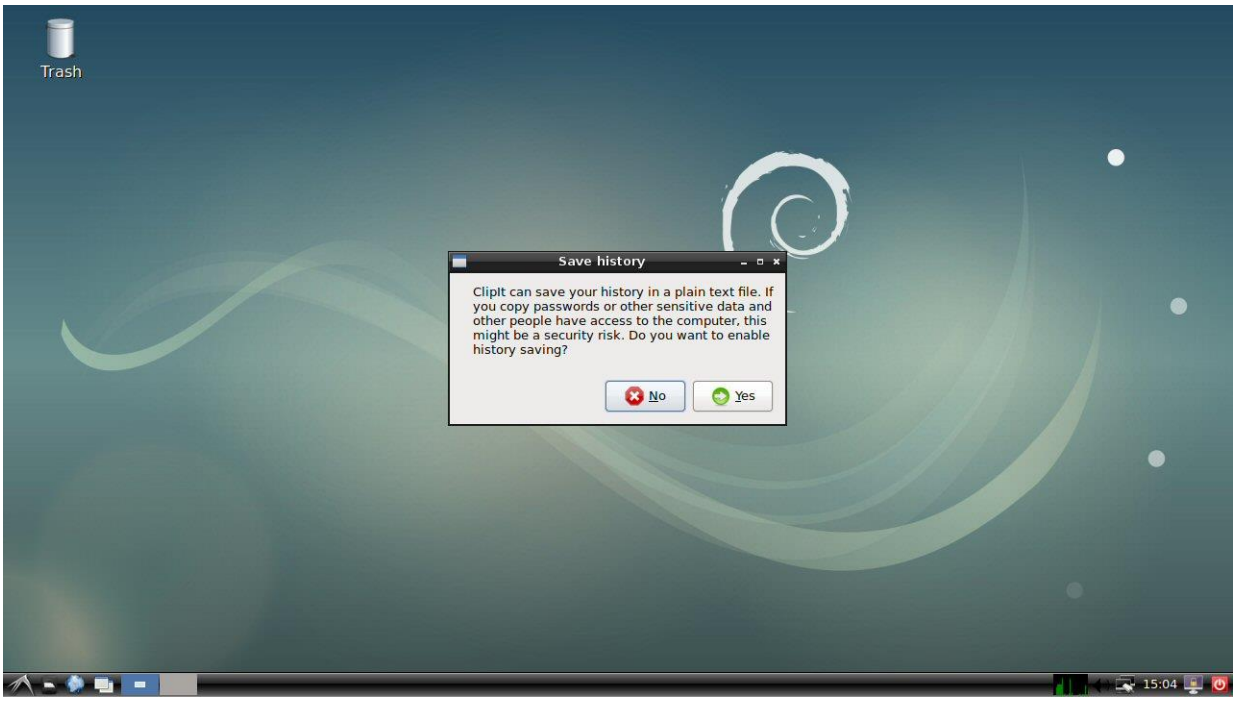


図 10-7-2. デスクトップ起動直後の画面

10-8.追加のパッケージのインストール

RTSP サーバー実装にために必要なパッケージをインストールします。

```
#  apt install -y bison bc autoconf pkg-config qv4l2 cmake
#  apt install -y libtool libelf-dev libssl-dev liblog4cpp5-dev libvpx-dev libjpeg-dev liblivemedia-dev libv4l-dev
```

10-9.インストール状況の確認

下記の機能がインストールされている事を確認します。

機能	確認コマンド	バージョン
Kernel	uname -a	Linux linaro-developer 5.4.74+
Distributor	lsb_release -a	Debian GNU/Linux 9.13 (stretch)
GCC	gcc -v	6.3.0
cmake	cmake --version	3.7.2
python	python --version	2.7.13
python3	python3 --version	3.5.3

バージョンは一例です。上記より新しい場合でも問題ありません。

11. RTSP サーバーの構築

RTSP サーバーを構築するため、各種のモジュールをインストールします。

11-1.Linux Headers のリンクとスクリプトの生成

Linux Headers のシンボリック・リンクを生成し、スクリプトの生成を行います。

```
#  ln -sf /lib/modules/5.4.74+/build /usr/src/linux-headers-5.4.74+
#  cd /usr/src/linux-headers-5.4.74+
#  make scripts
#  make prepare0
```

```
  HOSTCC  scripts/dtc/checks.o
  HOSTCC  scripts/dtc/util.o
  HOSTCC  scripts/dtc/dtc-lexer.lex.o
  HOSTCC  scripts/dtc/dtc-parser.tab.o
  HOSTLD  scripts/dtc/dtc
  HOSTCC  scripts/kallsyms
  HOSTCC  scripts/pnmtologo
  HOSTCC  scripts/conmakehash
  HOSTCC  scripts/recordmcount
  HOSTCC  scripts/sortextable
root@linaro-developer:/usr/src/linux-headers-5.4.74+# make prepare0
  CC      scripts/mod/empty.o
  HOSTCC  scripts/mod/mk_elfconfig
  MKELF   scripts/mod/elfconfig.h
  HOSTCC  scripts/mod/modpost.o
  CC      scripts/mod/devicetable-offsets.s
  HOSTCC  scripts/mod/file2alias.o
  HOSTCC  scripts/mod/sumversion.o
  HOSTLD  scripts/mod/modpost
  CC      kernel/bounds.s
  CC      arch/arm/kernel/asm-offsets.s
  CALL    scripts/checksyscalls.sh
  CALL    scripts/atomic/check-atomics.sh
root@linaro-developer:/usr/src/linux-headers-5.4.74+#
```

図 11-1. Linux Headers のリンクとスクリプトの生成

11-2.v4l2loopback のインストール

v4l2loopback は仮想ビデオデバイスを作成するカーネル・モジュールです。

```
#  cd /usr/src/
#  git clone https://github.com/umlaeute/v4l2loopback.git
#  cd v4l2loopback
#  make
#  make install
#  depmod -a
```

```
Building v4l2-loopback driver...
make -C /lib/modules/`uname -r`/build M=/usr/src/v4l2loopback modules
make[1]: Entering directory '/lib/modules/5.4.74+/build'
  CC [M]  /usr/src/v4l2loopback/v4l2loopback.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /usr/src/v4l2loopback/v4l2loopback.mod.o
  LD [M]  /usr/src/v4l2loopback/v4l2loopback.ko
make[1]: Leaving directory '/lib/modules/5.4.74+/build'
make -C utils
make[1]: Entering directory '/usr/src/v4l2loopback/utils'
cc  -I..   v4l2loopback-ctl.c   -o v4l2loopback-ctl
make[1]: Leaving directory '/usr/src/v4l2loopback/utils'
root@linaro-developer:/usr/src/v4l2loopback# make install
make -C /lib/modules/`uname -r`/build M=/usr/src/v4l2loopback modules_install
make[1]: Entering directory '/lib/modules/5.4.74+/build'
  INSTALL /usr/src/v4l2loopback/v4l2loopback.ko
  DEPMOD  5.4.74+
make[1]: Leaving directory '/lib/modules/5.4.74+/build'

SUCCESS (if you got 'SSL errors' above, you can safely ignore them)

root@linaro-developer:/usr/src/v4l2loopback# #depmod -a
root@linaro-developer:/usr/src/v4l2loopback#
```

図 11-2. v4l2loopback のインストール

11-3.live555 のインストール

live555 は RTP/RTCP, RTSP, SIP でメディアをマルチストリーミングするためのライブラリです。

```
#  cd /usr/src/
#  mkdir live555
#  cd live555
#  wget http://www.live555.com/liveMedia/public/live555-latest.tar.gz -O - | tar xvzf –
#  cd live
#  ./genMakefiles linux
#  make CPPFLAGS=-DALLOW_RTSP_SERVER_PORT_REUSE=1 install
```

図 11-3. live555 のインストール

11-4.v4l2rtspserver のインストール

v4l2rtspserver は Video4Linux デバイスからのストリーマーフィードで、RTSP サーバーをサポートします。

```
#  cd /usr/src/
#  git clone https://github.com/mpromonet/v4l2rtspserver.git
#  cd v4l2rtspserver
#  git tag
#  git checkout refs/tags/v0.2.2
#  cmake .
#  make
#  make install
```

※2021.04.06 時点の最新バージョン v0.2.3 を使用した場合、”cmake .”でエラーが発生したため、v0.2.2 を使用しています。

```
ediaSubsession.cpp.o
[ 98%] Building CXX object CMakeFiles/libv4l2rtspserver.dir/src/ServerMediaSubse
ssion.cpp.o
[ 98%] Building CXX object CMakeFiles/libv4l2rtspserver.dir/src/TSServerMediaSub
session.cpp.o
[ 98%] Building CXX object CMakeFiles/libv4l2rtspserver.dir/src/UnicastServerMed
iaSubsession.cpp.o
[ 99%] Linking CXX static library liblibv4l2rtspserver.a
[ 99%] Built target libv4l2rtspserver
Scanning dependencies of target v4l2rtspserver
[ 99%] Building CXX object CMakeFiles/v4l2rtspserver.dir/main.cpp.o
[100%] Linking CXX executable v4l2rtspserver
[100%] Built target v4l2rtspserver
root@linaro-developer:/usr/src/v4l2rtspserver# make install
[  4%] Built target v4l2wrapper
[ 99%] Built target libv4l2rtspserver
[100%] Built target v4l2rtspserver
Install the project...
-- Install configuration: ""
-- Installing: /lib/systemd/system/v4l2rtspserver.service
-- Installing: /usr/local/bin/v4l2rtspserver
-- Installing: /usr/local/share/v4l2rtspserver/index.html
-- Installing: /usr/local/share/v4l2rtspserver/hls.js/dist/hls.light.min.js
root@linaro-developer:/usr/src/v4l2rtspserver#
```

図 11-4. v4l2rtspserver のインストール

12. USB カメラで RTSP サーバーを起動（オプション）

RTSP サーバーの動作を確認するため、USB カメラでの動作を確認します。

USB カメラが準備出来ない場合、本章を飛ばして次章に進んでください。

12-1.USB カメラ (UVC) の動作確認

USB カメラが用意出来る場合、一旦、USB カメラでの動作を確認します。

ターミナルから USB カメラが認識されている事を “ls /dev” コマンドで確認します。

```
initctl         pts         tty0    tty34   tty6    uio1    video0
input           ptyp0       tty1    tty35   tty60   urandom video1
kmsg            ptyp1       tty10   tty36   tty61   v4l     watchdog
log             ptyp2       tty11   tty37   tty62   vcs     watchdog0
loop-control    ptyp3       tty12   tty38   tty63   vcs1    zero
loop0           ptyp4       tty13   tty39   tty7    vcs2
loop1           ptyp5       tty14   tty4    tty8    vcs3
loop2           ptyp6       tty15   tty40   tty9    vcs4
loop3           ptyp7       tty16   tty41   ttyLCD0 vcs5
```

図 12-1-1. USB カメラ (UVC) の確認

デスクトップのアプリケーションメニューの Sound & Video > Qt V4L2 test Utility を実行して正常に動作する事を確認してください。

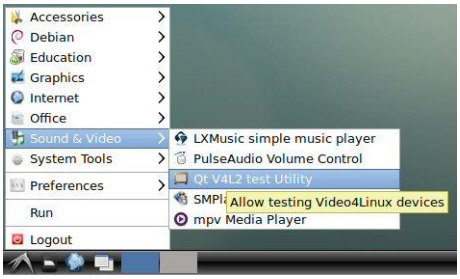


図 12-1-2. Qt V4L2 test Utility の起動

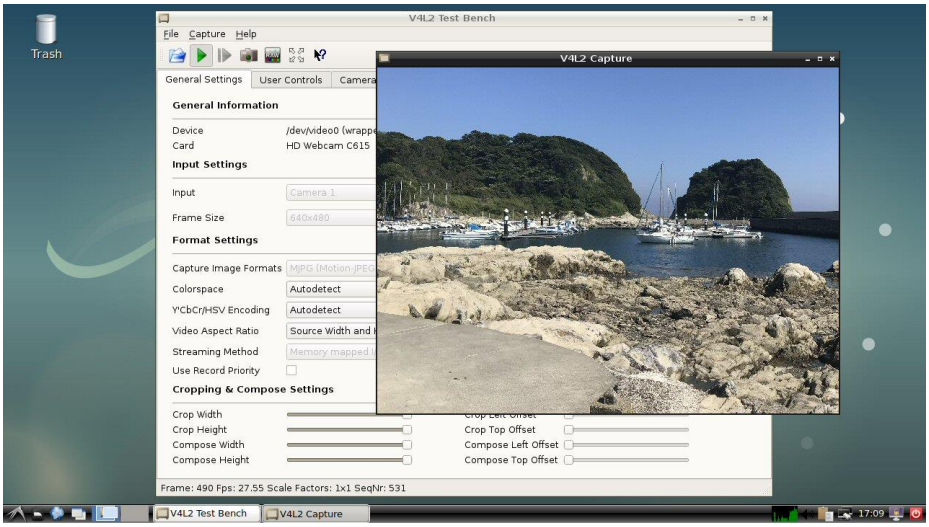


図 12-1-3. Qt V4L2 test Utility の実行

12-2.USB カメラ (UVC) での RTSP サーバー実行

USB カメラで撮影された映像を RTSP で配信します。

下記のコマンドを実行します。

```
#  modprobe v4l2loopback video_nr=10
#  v4l2compress /dev/video0 /dev/video10 -fJPEG &
   ⏎
#  v4l2rtspserver /dev/video10 &
   ⏎
```

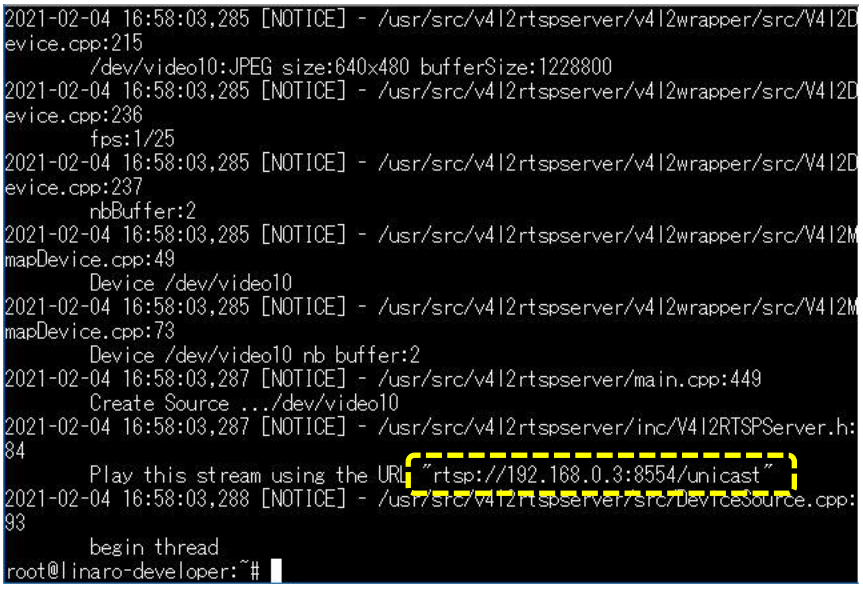


図 12-2-1. USB カメラ (UVC) での RTSP サーバー実行

modprobe v4l2loopback video_nr=10 : /dev/video10 の仮想ビデオデバイスを作成します。

v4l2compress /dev/video0 /dev/video10 -fJPEG & : USB カメラデバイス (/dev/video0) を JPEG 圧縮して仮想ビデオデバイス (/dev/video10) に送ります。RTSP サーバー実行時には動作し続ける必要がありますので、コマンドの最後に “&” を付加します。

v4l2rtspserver /dev/video10 & : 仮想ビデオデバイス(/dev/video10) の映像を RTSP で配信します。受信する場合、サーバーアドレスは、ターミナルに出力されている “rtsp://192.168.0.3:8554/unicast” を使用します。こちらも動作し続ける必要がありますので、コマンドの最後に “&” を付加します。

上記で実行したプロセスを停止させる場合は、 “ps” コマンドで表示された v4l2compress と v4l2rtspserver の PID を “kill” コマンドで終了させてください。

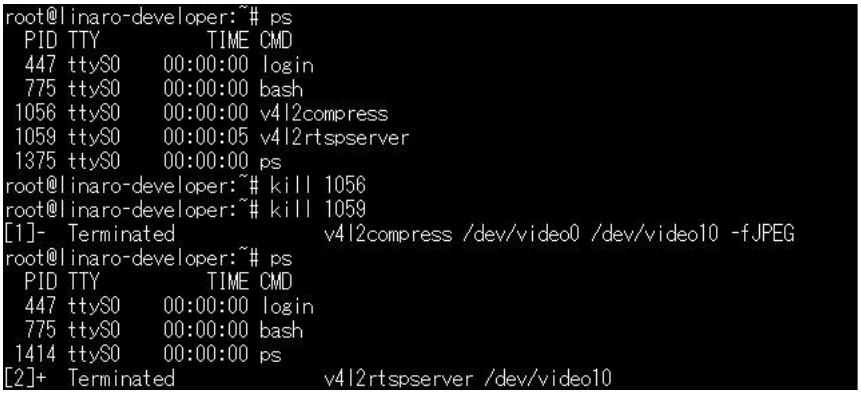


図 12-2-2. 実行中モジュールの停止

12-3.Linux PC での RTSP 受信

RTSP サーバーから配信された映像を Linux PC で再生します。

再生にはツールが必要ですが、本資料では VLC メディア・プレーヤを使用した例を紹介します。

12-3-1. VLC メディア・プレーヤのインストールと実行

下記のコマンドでインストールします。

```
#  sudo apt install -y vlc
```

インストールが完了したら VLC メディア・プレーヤを起動します。

```
#  vlc
```

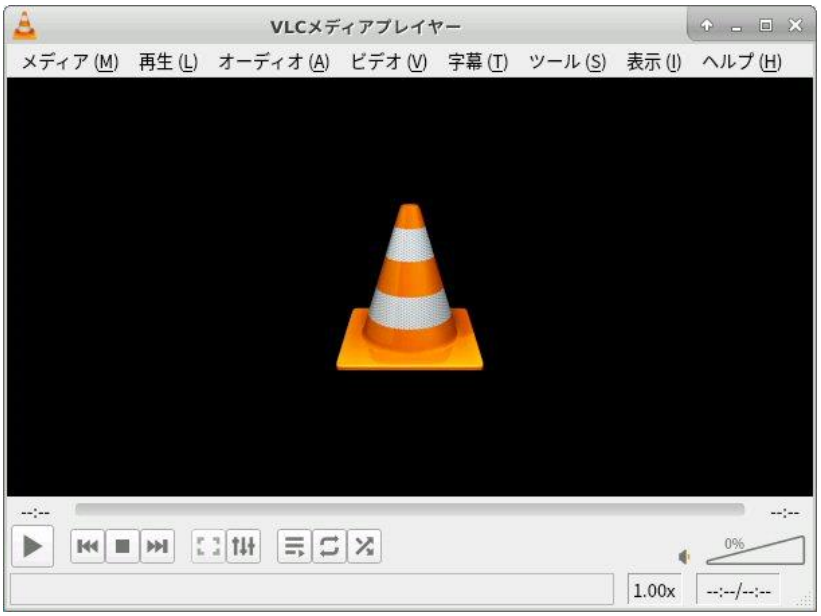


図 12-3-1. VLC メディア・プレーヤ

メインメニューから、”メディア” → ”ネットワークストリームを開く(N)...” を選択し、ネットワーク・タブの “ネットワークの URL を入力してください” に、**12-2. USB カメラ (UVC) での RTSP サーバー実行** で示された URL を入力し、再生ボタンをクリックします。

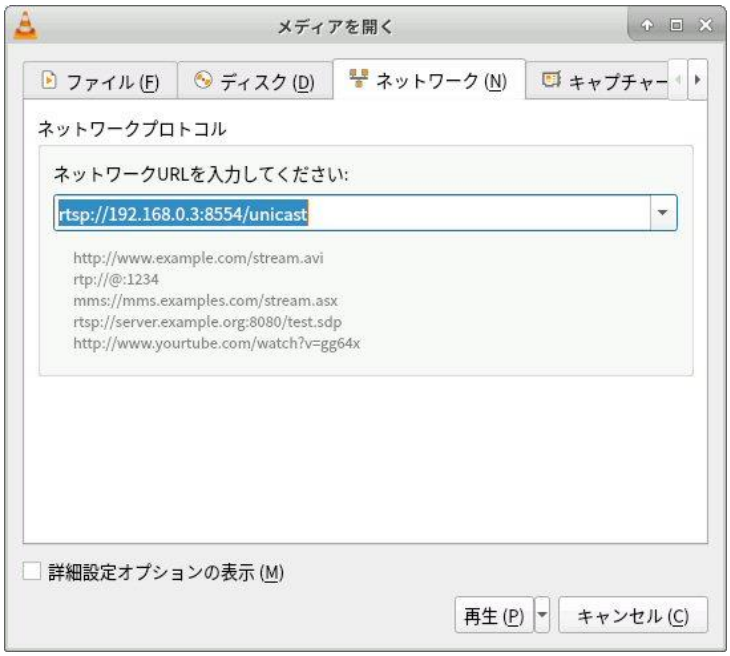


図 12-3-2. ネットワークストリームを開く

すると KEIm-CVSoC に接続されたカメラの映像を確認する事ができます。

図 12-3-3. RTSP サーバーからの映像を再生

13. カメラモジュールからの映像を RTSP で配信

カメラモジュールで撮影された映像を RTSP で配信します。

本資料では処理の詳細については説明しておりませんのでご了承ください。

13-1.v4l2tools のインストール

v4l2tools は V4L2 を利用したシンプルで便利なサンプル・コードとライブラリです。このソースコードを流用してカメラモジュールからの映像をビデオデバイスに出力する処理を作成します。

```
#  cd /usr/src/
#  git clone https://github.com/mpromonet/v4l2tools.git
#  cd v4l2tools
#  make
#  make install
```

図 13-1. v4l2tools のインストール

make / make install が正常終了した事を確認してください。

13-2.カメラモジュールからビデオデバイスへの出力

カメラモジュールからの映像データは、デバイス・ツリーで設定されたフレームバッファ用メモリリージョンに書き込まれます。このデータを仮想ビデオデバイスに送り込む処理を実装します。

13-2-1. ファイルの転送

SSH などを使ってダウンロードした RTSP Server 用ソフトウェアを下記のフォルダに転送します。

ファイル	転送先フォルダ	概要
v4l2cam2dev.cpp	/usr/src/v4l2tools/src	カメラモジュールの映像を仮想デバイスに出力するメイン処理
cam_init.cpp	/usr/src/v4l2tools/src	カメラモジュールに関連した初期化処理
ov5642.h	/usr/src/v4l2tools/inlude	ov5642 の初期設定のためのレジスタ値

13-2-2. makefile の変更

v4l2tools の makefile を変更して、v4l2cam2dev.cpp をコンパイル、インストール出来るようにします。

- 先頭行の ALL_PROGS に v4l2cam2dev を追加

```
ALL_PROGS = v4l2copy v4l2convert_yuv v4l2source_yuv v4l2dump v4l2compress
```

⇓

```
ALL_PROGS = v4l2copy v4l2convert_yuv v4l2source_yuv v4l2dump v4l2compress v4l2cam2dev
```

- 102 行目あたりに v4l2cam2dev の記述を追加

```
libv4l2wrapper.a:
	git submodule update --init v4l2wrapper
	make -C v4l2wrapper
	mv v4l2wrapper/libv4l2wrapper.a .
	make -C v4l2wrapper clean

# read V4L2 capture -> write V4L2 output
v4l2copy: src/v4l2copy.cpp	libv4l2wrapper.a
	$(CXX) -o $@ $(CFLAGS) $^ $(LDFLAGS)
```

⇓

```
libv4l2wrapper.a:
	git submodule update --init v4l2wrapper
	make -C v4l2wrapper
	mv v4l2wrapper/libv4l2wrapper.a .
	make -C v4l2wrapper clean

# read Camera Module -> V4L2 device
v4l2cam2dev: src/v4l2cam2dev.cpp src/cam_init.cpp libv4l2wrapper.a
	$(CXX) -o $@ $(CFLAGS) $^ $(LDFLAGS)

# read V4L2 capture -> write V4L2 output
v4l2copy: src/v4l2copy.cpp	libv4l2wrapper.a
	$(CXX) -o $@ $(CFLAGS) $^ $(LDFLAGS)
```

13-2-3. v4l2cam2dev のインストール

v4l2cam2dev.cpp を make / make install します。

```
#  make
#  make install
```

```
root@linaro-developer:/usr/src/v4l2tools# make
ARCH=armv7l
with log4cpp
g++ -o v4l2cam2dev -std=c++11 -W -Wall -pthread -g -pipe  -I include -DHAVE_LOG4
CPP -I /usr/include -I v4l2wrapper/inc -DHAVE_VPX -DHAVE_JPEG src/v4l2cam2dev.cp
p src/cam_init.cpp libv4l2wrapper.a -llog4cpp  -lvpx -ljpeg
root@linaro-developer:/usr/src/v4l2tools# make install
ARCH=armv7l
with log4cpp
mkdir -p /usr/bin
install -D -m 0755 v4l2copy v4l2convert_yuv v4l2source_yuv v4l2dump v4l2compress
 v4l2cam2dev v4l2uncompress_jpeg /usr/bin
root@linaro-developer:/usr/src/v4l2tools#
```

図 13-2-3. v4l2cam2dev のインストール

13-3.カメラモジュールでの RTSP サーバー実行

カメラモジュールで撮影された映像を RTSP で配信します。

下記のコマンドを実行します。

```
#  modprobe v4l2loopback video_nr=10,11
#  v4l2cam2dev /dev/video10 &
   ⏎
#  v4l2compress /dev/video10 /dev/video11 -fJPEG &
   ⏎
#  v4l2rtspserver /dev/video11 &
   ⏎
```

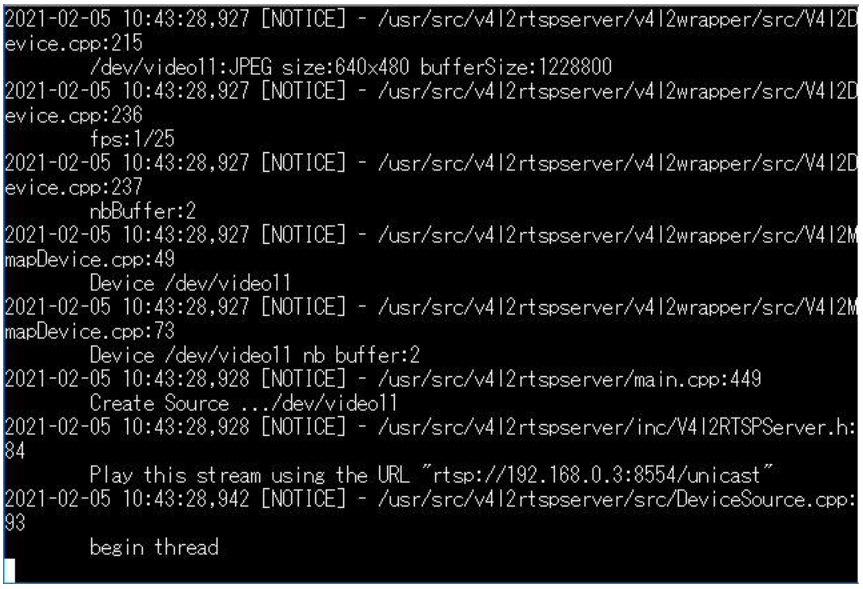


図 13-3. カメラモジュールでの RTSP サーバー実行

modprobe v4l2loopback video_nr=10,11 : /dev/video10 と /dev/video11 の2つの仮想ビデオデバイスを作成します。/dev/video10 はカメラモジュールからの映像を出力する仮想ビデオデバイスで、/dev/video11 は JPEG 圧縮後の映像を出力する仮想ビデオデバイスです。

v4l2cam2dev /dev/video10 & : 本項で作成したモジュールです。カメラモジュールからの映像を仮想ビデオデバイス (/dev/video10) に出力します。RTSP サーバー実行時には動作し続ける必要がありますので、コマンドの最後に “&” を付加します。

v4l2compress /dev/video10 /dev/video11 -fJPEG & : カメラモジュール (/dev/video10) を JPEG 圧縮して仮想ビデオデバイス (/dev/video11) に送ります。RTSP サーバー実行時には動作し続ける必要がありますので、コマンドの最後に “&” を付加します。

v4l2rtspserver /dev/video11 & : 仮想ビデオデバイス(/dev/video11) の映像を RTSP で配信します。受信する場合、サーバーアドレスは、ターミナルに出力されている “rtsp://192.168.0.3:8554/unicast” を使用します。こちらも動作し続ける必要がありますので、コマンドの最後に “&” を付加します。

なお、v4l2cam2dev では出力解像度を指定する事が出来ます。 デフォルトでは 640 x 480 (VGA) ですが、コマンドの後ろに -W と -H を指定する事で映像の幅と高さを指定する事が出来ます。

例） 解像度を 320x240 (QVGA) で出力する場合

```
#  v4l2cam2dev /dev/video10 -W 320 -H 240 &
```

13-4.Linux PC での RTSP 受信

12-3. Linux PC での RTSP 受信 を参照して動作を確認してください。

改版履歴

Revision	更新日	概要
0.1	2021/02/08	初版
1.0	2021/04/06	HP 公開版