PCBa documentation

Documentation for the entirity of PCBa_infrastructure repository. Available in form of mdbook at code.siemens.com pages

OCII

`OCII`, also known as `OpenController II` or `CPU-1515SP-PC2` is a classic IPC device with `64-bit x86` processor (the same architecture and typical desktop computer).

Harware

- CPU: Intel E3940 quad-core
- Memory: 8 GB DDR3

RTC

Our `OCII` has 2 RTC clocks.

First is the `CMOS RTC` - RTC that would normally be powered by CMOS battery (line typical consumer computer). On `OCII` we can't use this RTC because we cannot have CMOS battery (might be because of the operation temperatures the `OCII` is designed for, not sure).

Since we can't have battery in the device, we have a super-capacitor. However ordinary CMOS clock is too power hungry and would deplete the capacitor too quickly. That is where the second RTC clock comes in - it has much lower power consumption and can last week or two powered by the super-capacitor.

Software overview

Generally speaking, it can run any GPOS (General-Purpose Operating System) such as `Windows 10` or `Linux`.

> **Note**
>
> `OCII` is often also refered to as `PC-based` since it has the same underlying architecture as typical computer.

In our `OCII` team we are responsible (among other things) for creating bundles.

Note

> Bundle is prepared GPOS installation with additional software that is distributed to customers who buy `OCII`. Bundle is written to CFast card and then bundled to the `OCII` (packed into the same box) and sold to the customer.
>
> At the moment, we are creating Windows 10 bundles and Industrial OS bundles (based on Debian 10 at the time of writing).

Each bundle contains GPOS and `SWCPU`. To have both GPOS and `SWCPU` running on single `x86` processor, there is also hypervisor.

Note

> `SWCPU`, also known as `SIMATIC S7-1500 Software Controller`, is software that emulates PLC and can run on `x86` processor.
>
> ---
>
> The SIMATIC S7-1500 Software Controller is a PC-based controller and offers the same functionality as all CPUs of the SIMATIC S7-1500 automation system in a PC-based real-time environment.
>
> ---

Warning

> Siemens sucks at naming things. When you see somewhere `CPU` acronym (by normal people used to describe Central Processing Unit), it often referes to classic PLC or IPC instead.

Note

> Hypervisor definition from wikipedia.org:
>
> ---
>
> A hypervisor (also known as a virtual machine monitor, VMM, or virtualizer) is a type of computer software, firmware or hardware that creates and runs virtual machines.
>
> ---

Note

> Siemens makes multiple hypervisors (at least 3 different ones). I do not know which one we are using.
>
> All I know that that hypervisor we are using is TYPE-1 hypervisor and it is activelly trying to hide it's presence (it is hard to detect).
>
> Also, I do not know if the hypervisor for Windows and IndOS bundles are the same. (likely that they are the same)

The hypervisor assigns 1 CPU core and 1 GB of memory to `SWCPU`, remaining 3 CPU cores and 7 GB of memory to GPOS.

Warning

> The used hypervisor comes with it's own GRUB. Any changes to this GRUB will cause hypervisor crash! Do not touch it!

BIOS

`OCII` uses open-source implementation of BIOS called coreboot (official site at coreboot.org).

For more information look into documentation of system-rescue-remaster project.

Note

> Our implementation of `BIOS` does not have classic boot order / boot sequence. Instead it has only a single boot device set, and it will try too boot from this single device indefinitely.

It is possible to change the boot device, but it is not ideal for automation since keyboard is required. Initially we wanted to use our arduino-keyboard but it is not recognized in the `BIOS`. I thought of trying out the QMK Firmware which should be a proper USB keyboard implementation rather than some hacked together things that is current arduino-keyboard, but I never got time to do it.

Instead of changing the boot device via keyboard, we have asked Johannes Hahn to make a custom `BIOS` for us that would have two boot devices - network boot and CFast card.

This special `BIOS` is stored in out bios-binaries repository along with instructions.

By default, this special `BIOS` tries to boot from network, if not possible then from CFast card. This allows us to automate the bundle testing with CI/CD.

Industrial OS

What is Industrial OS

`Industrial OS`, also known as `IndOS` and `SIMATIC Industrial OS` is a Linux distribution based on Debian.

Note

> At the time of writing there is `IndOS v2.x` based on Debian 10 (Buster) and `IndOS v3.x` based on Debian 11 (Bullseye).
>
> Details regarding Debian releases can be found at wikipedia.org.

Note

> `IndOS` is developed by company called Mentor (I believe Siemens bought them some time ago, but I am not sure).

Warning

> Mentor is known for very long delivery time, inflexibility and not so great support. Often, when you need something, you are often better of doing it yourself.

Building IndOS

If I am not mistaken, `IndOS` is made using either yocto or isar (also at code.siemens.com; it is developed by Siemens). So you can build your own `IndOS` version yourself - but don't, it is a mess. In ideal world, you task Mentor with you requirements, and they do it for you.

What we do here in `OCII` team instead is to take already created `IndOS` image from place called artifactory and alter it. The exact location where these images can be found is written in configuration JSON file in `bundle_generation_industrial_os` repository, specifically src/version_configuration.json.

Note

> The image files have .img extension / file format, and they are just a raw disk images (raw binary copy of disk with partitions and filesystems).
>
> From wikipedia.org description:
>
> ---
>
> A disk image, in computing, is a computer file containing the contents and structure of a disk volume or of an entire data storage device, such as a hard disk drive, tape drive, floppy disk, optical disc, or USB flash drive. A disk image is usually made by creating a sector-by-sector copy of the source medium, thereby perfectly replicating the structure and contents of a storage device independent of the file system.
>
> ---

How we do it

So, we basically take the `IndOS` image file produced by Mentor, we attach said image as Loop device, insert few scripts and things, detach it and call the resulting file our `IndOS bundle`. This is super fast and can be done in few seconds compared to the entire `IndOS` yocto / isar insanity.

In our pipeline it takes longer (around 2 or 3 minutes) because of gitlab runner overhead (spinning up new virtual machine, and so on) and then file transfer over network to NAS (`unsyncable-agony`). But this is still very decent time, especially when compared to Windows bundle building infrastructure that takes hours to produce bundle.

Note

> Loop device, according to Wikipedia is:
>
> ---
>
> In Unix-like operating systems, a loop device, vnd (vnode disk), or lofi (loop file interface) is a pseudo-device that makes a computer file accessible as a block device.
>
> ---
>
> Basically you can take a file and appear it in the system as block device (just like physical disk).
>
> Example of using loop device and image file using BeagleBone Black Debian: Exploring .img Files on Linux.

Detailed description in next chapter Infrastructure for IndOS.

Short version

1. Attach the `IndOS` image file as loop device.
2. Mount partition with `IndOS` installation.
3. Copy over systemd services and executable files into mounted partition.
4. Unmount and detach image.
5. Bundle is now complete.
6. You can place the modified image onto CFast card, and during first boot, it will get all installed.

Note

> `systemd` according to archlinux wikipedia:
>
> ---
>
> systemd is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system. systemd provides aggressive parallelization capabilities, uses socket and D-Bus activation for starting services, offers on-demand starting of daemons, keeps track of processes using Linux control groups, maintains mount and automount points, and implements an elaborate transactional dependency-based service control logic.
>
> ---

The files that we copy over into the bundle are stored in directories `bin`, `config` and `systemd` which are in src/ directory.

The scripts and `systemd` services copied into the bundle are there to install `SWCPU` and other components, and to do some basic configuration during first boot. These steps must be done in specific order.

The reasons why it is done this way:

- it is difficult to install anything into system before first boot
  - during first boot the system expands it's partition to fill all available space on block device
  - during first boot you have to configure usernames, passwords, keyboard layout and things like this (some of this configuration is required for package installation)
- `SWCPU`
  - during it's installation it required user to agree to EULA (we should not agree to EULA or user's behalf)

- it can't be installed before `IndOS` expands it's disk, because it's secondary installer (which also fucks with partition table) conflicts with `IndOS`
- `SWCPU` installation can't run inside virtual environment (I tested this tested), so such the bundle generation would require real hardware, making it long and complicated (like Windows bundles - no thanks)

Warning

> `SWCPU` and hypervisor are such fragile princesses that anything can break them easily.
>
> While I do not like the saying `if it works, do not fix it`, it this case it is a matter or keeping your sanity - do not try to fix it or improve it. Consider yourself warned.

src/bin/

hotfix-time.sh

It can happen that customer's `OCII` has wrong time in RTC (the time is lost when device is without power for more than week or two). This causes errors when package manager (`apt`) attempts to install packages, which appear to be from the future.

Because of this, during the bundle build we generate `/etc/buildtime` text file with the time-stamp when the bundle was built. Then we call this script in all of ours `systemd` service to fix the system time if it is in past compared to content of `/etc/buildtime`.

```
#!/bin/bash

set -Eeuo pipefail

# Fix date if wrong
TIME_NOW=$( date +"%s" )
BUILD_TIME=$( cat ./etc/buildtime )

if [[ ${BUILD_TIME} -gt ${TIME_NOW} ]]; then
    echo "WARNING: System time is in past, time will be moved forward to enable
package installation"
    date -s "@${BUILD_TIME}"
else
    echo "Time is OK"
fi
```

`swcpu-pkg.sh`

This script installs Siemens packages such as `SWCPU` and hypervisor. Each of these comes with their second-stage installer which runs after reboot to fuck with partition table and GRUB and stuff.

```
#!/bin/bash

# shellcheck disable=SC1091

#set -Eeuo pipefail
#set -Eeo pipefail
#       the `install.sh` has undefined variables all over the place

sleep 1
chvt 2
clear

echo "--- START ---"

# Find location of packages
INSTALLERDIR=$( find /root/ -maxdepth 1 -mindepth 1 -type d -name "CPU*" | head -1
)
echo "** INSTALLERDIR: ${INSTALLERDIR}"
cd "${INSTALLERDIR}" || exit 1

ln -s "${INSTALLERDIR}/packages" /usr/bin/packages

# Disable disclaimer
if [[ -f "/etc/skip_eula" ]]; then
    alias "whiptail"="echo 111222;true"
    shopt -s expand_aliases
fi

# Install
echo "--- Start installation ---"
for i in {1..3}; do
    # This ugly re-try code is needed because the installation script introduces
some race-condition that I can't track down
    echo ""
    echo "--- Installation attempt: ${i} ---"
    if [[ -f "/etc/skip_eula" ]]; then
        echo n | source install.sh
    else
        ./install.sh
    fi
    echo "Exit code: $?"
    echo "--- Done ---"

    # Check if packages were installed
    echo "** Checking: siemens-swcpu-man-pages" &&
    dpkg -l siemens-swcpu-man-pages &&
    echo "** Checking: siemens-swcpu-hooks" &&
    dpkg -l siemens-swcpu-hooks &&
    echo "** Checking: simatic-vmm-call-ipc" &&
    dpkg -l simatic-vmm-call-ipc &&
    echo "** Checking: simatic-vmm-vnic-ipc" &&
    dpkg -l simatic-vmm-vnic-ipc &&
    echo "** Checking: vmm-installer" &&
```

```
    dpkg -l vmm-installer &&
    echo "--- Everything is good ---" &&
    break
    #dpkg -l CPU1505SP-installer        # this package should be installed by
`vmm-installer`
    sleep 5
done

echo "--- INSTALLATION COMPLETE ---"
systemctl disable swcpu-pkg.service
reboot
```

Warning

> These Siemens packages are broken, especially the provided `install.sh` script which has totally broken error-handling, because it will report success even when the package failed to install.
>
> I opened a issue in TFS (I can't find it right now), but I do not have to wait for them to fix it, or even fix it myself, so I just re-try the installation 3 times.
>
> The `dpkg -l` commands tests if package is installed, if not try again. 2 tries seems to be the magic number, so for I good measure there is a `for` loop with 3 iterations.

swcpu-configure.sh

```
#!/bin/bash

set -Eeuxo pipefail

echo "--- START ---"
SWCPU_MAC_ADDR=$( s7_vnic_macconfig -m /mnt/swcpu_mount | grep SWCPU | sed -E
's/[A-Z]+ //g' )
SWCPU_IP_ADDR="192.168.73.1"
SWCPU_INTERFACE="enp0s1f1"
LINUX_IP_ADDR="192.168.73.2"
LINUX_MASK="24"

# Configure SWCPU
echo "--- Configure SWCPU ---"

echo "** Temporarily add IP address to Linux VNIC **"
ip addr add "${LINUX_IP_ADDR}/${LINUX_MASK}" dev "${SWCPU_INTERFACE}" || true

echo "** Groups **"
if [[ ! $(getent group failsafe_operators) ]]; then
    groupadd failsafe_operators
fi

if ! groups | grep -q failsafe_operators; then
    gpasswd -a root failsafe_operators
fi

echo "** Configure SWCPU **"
s7_vnic_ipconfig --nic "${SWCPU_INTERFACE}" --mac "${SWCPU_MAC_ADDR}" --setip
"${SWCPU_IP_ADDR}"
s7_resource_configurator -r
"/mnt/swcpu_mount/SWCPU/etc/resource_configurator/OC2_basic_configuration.json"

echo "** Add SWCPU to /etc/hosts file **"
if ! grep -q "vnic-swcpu" /etc/hosts; then
echo "
# VNIC config
${LINUX_IP_ADDR}        vnic-localhost
${SWCPU_IP_ADDR}        vnic-swcpu" >> /etc/hosts;
fi

echo "** Done **"
```

rib-install.sh

```
#!/bin/bash

set -Eeuxo pipefail
RIB_INST="/mnt/swcpu_mount/SWCPU/RIB/RIBSetup"

# Run RIB installer
echo "--- Execute RIB installer ---"
chmod +x "${RIB_INST}"
${RIB_INST}

if [[ -f "/usr/bin/RIB_App" ]]; then
    echo "*** Done ***"
else
    echo "!!! Failed !!!"
    exit 1
fi
```

Long version

The pipeline overview

First, let's look at the stages:

testlint

This stage is there to check all the source-code with linters.

> **Note**
>
> Linter according to Wikipedia:
>
> ---
>
> Lint, or a linter, is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs.
>
> ---

As you can see we run quite a few linters:

You are most likely to come into conflict with `linter-editorconfig` and `linter-python`. For the `editorconfig`, there are plugins for most comonly used editors and IDE's, so use that to have it automatically use predefined styles.

Note

> editorconfig is de-facto a standard. It defines a formatting style for your files and source code.
>
> ---
>
> EditorConfig helps maintain consistent coding styles for multiple developers working on the same project across various editors and IDEs.
>
> The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles.
>
> EditorConfig files are easily readable and they work nicely with version control systems.
>
> ---

Regarding the python linting, I recommend you to run python linter on your local computer before committing any changes. We use pylint.

build

Stuf

```
```

The bundle buidling process starts

The installation is done opening the image file as loop-back device:

```
losetup -fP development-image-industrial-os-ipc.wic.img
```

The image file has following partitioning (`lsblk -f`):

```
NAME                   FSTYPE      FSVER LABEL          UUID
FSAVAIL FSUSE% MOUNTPOINTS
loop0
├─loop0p1              vfat        FAT32 EFI            9988-9437
└─loop0p2              ext4        1.0   root           71ee49ee-c9b5-4c96-
8afe-1a00f212ac2b
```

Or output of `fdisk -l /dev/loop0`:

```
Disk /dev/loop0: 2.9 GiB, 3110080512 bytes, 6074376 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 05584AD5-AD47-4F4A-99BA-FED4A5F830DE

Device        Start     End Sectors  Size Type
/dev/loop0p1   2048  526335  524288  256M EFI System
/dev/loop0p2 526336 6074341 5548006  2.6G Linux filesystem
```

Then the second partition is partition with Industrial OS which is then mounted:

```
mount /dev/loop0p2 /mnt
```

Then SWCPU files are copied over along with custom-made systemd services which do the installation on first boot. Mentioned systemd service are then enabled via chroot:

```
arch-chroot /mnt systemctl enable <service>
```

Unmount and close loop-back device. The bundle is now complete.

Repository structure

- `configuration`
  - arduino-keyboard
    - Program for Arduino DUE which acts like USB keyboar, and recieves commands over network thank to Ethernet shield.
  - arduino-relay
    - Program for Arduino UNO to switch relays to control power to `OCII` devices.
  - codesiemens-runner-dotfiles
    - Dotfiles for code.siemens.com runner (a the moment out of date)
  - flashrom-testing
    - (out of date)
  - system-dotfiles
    - Dotfiles for both NAS (unsyncable-agony) and gitlab runner (sisyphus)
    - **NOTE:** Some files are encrypted with transcrypt because they contain sensitive information (they are versioned as base64).
- `docker-images`
  - docker-archlinux-building
    - Docker image based on ArchLinux with pre-installed packages for compiling and building software.
  - docker-archlinux-documentation
    - Docker image based on ArchLinux with pre-installed packages for creating documentation.
- `gitlab_workshop`
  - Stuff and things for educational purposes.
- `templates`
  - Mostly LaTeX templates for documentation, release notes and additional documents.
- bundle_generation_industrial_os
  - Repository containing everything needed for building and testing Industrial OS bundles.

Note

Dotfiles is a term to describe configuration files of system or programs. For example `~/.bashrc` or `/etc/fstab`.

Sources:

- What are dotfiles?
- Dotfiles – What is a Dotfile and How to Create it in Mac and Linux

Examples:

- My personal dotfiles for system
- My personla dotfiles for user

Note

LaTeX according to latex-project.org:

LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation.
