Flash on Pi

Der "flash" wird in 3 generellen Schritten durchgeführt:

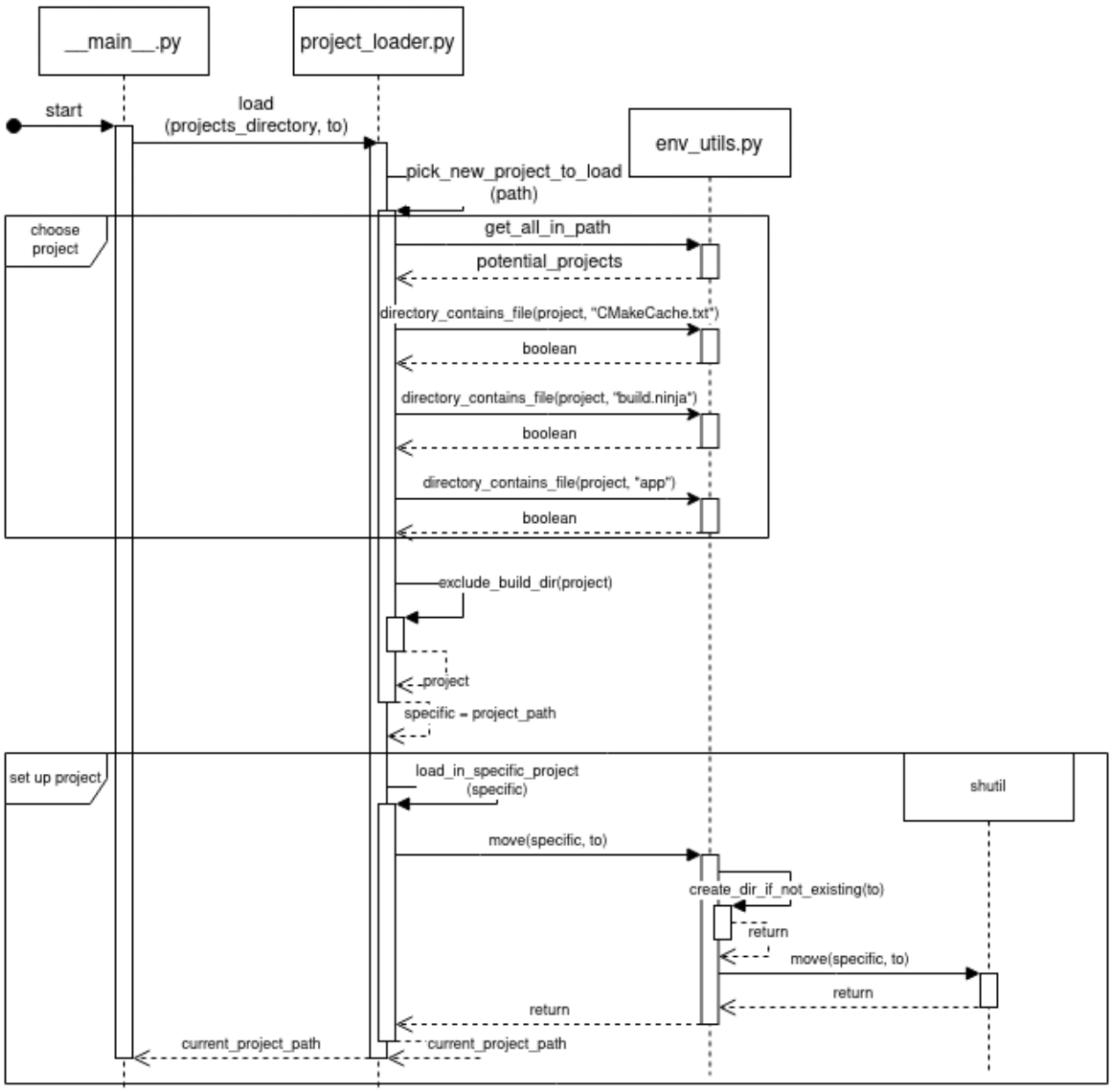
1. laden des projektes
2. "flash" vorbereitung
3. "flash" des Projektes

1. Laden des Projektes (project_loader)

Der Project Loader wählt zunächst das zu flashende Projekt aus und schiebt es in den lokalen "Bearbeitungs"-Ordner.

Damit ein Projekt als "zu bearbeiten" gesehen werden kann, muss dieses mindestens 3 Sachen beinhalten:

- eine "CMakeCache.txt" Datei
- eine "build.ninja" Datei
- eine "app" Directory



(Das Sequenzdiagram zeigt hier nur die Möglichkeit an, dass ein spezifisches Projekt noch nicht ausgewählt wurde. Wenn dies jedoch getan wurde, wird nur der "set up project" Teil durchgeführt und alles andere übersprungen.)

Von dort aus an setzen die "flash"-Vorbereitungsscripts ein.

2. "flash"-Vorbereitung

Da der `build` prozess auf einem anderem System geschehen ist, muss der `flash` Prozess zunächst vorbereitet werden. Diese Vorbereitung beinhaltet einzelne Schritte, welche die Verlinkung zu bestimmten Pfaden abändert, sodass diese sich im Einklang mit dem System vom Pi verhalten. Die gebrauchten Pfade (Verlinkungen), die geändert werden müssen, können sich praktischerweise in einer Datei wiederfinden.

CMakeCache.txt

Diese Datei kann man fast als Informationsausgabe vom `build` Prozess bezeichnen. Sie beinhaltet viele verschiedene Informationen, welche unteranderem auch die Umgebung umfasst, in welcher der `build` Prozess stattfand(generiert wird sie vom Tool: `cmake`).

Zephyr Build System

Die für uns wichtigsten Informationsteile zum vorbereiten des flash-prozesses sind folgende:

- ```
  // Application Binary Directory
  APPLICATION_BINARY_DIR:PATH=/home/user/zephyr/samples/basic/fade_led/build
  ```
- ```
  // Application Source Directory
  APPLICATION_SOURCE_DIR:PATH=/home/user/zephyr/samples/basic/fade_led
  ```
- ```
  //Path to CMake executable.
  CMAKE_COMMAND:INTERNAL=/usr/bin/cmake
  ```
- ```
  //Zephyr SDK install directory
  ZEPHYR_SDK_INSTALL_DIR:PATH=/home/user/zephyr-sdk-0.13.2
  ```
- ```
  //Zephyr base
  ZEPHYR_BASE:PATH=/home/user/zephyrproject/zephyr
  ```

Vorgang der Vorbereitung

Um das build-directory für den `flash` vorzubereiten, werden diese Informationsschnipsel in Zusammenarbeit mit dem gegeben Informationen aus der `.env` Datei genutzt, um Teile aus Dateien umzuschreiben.

Um das Umändern der Verlinkungen durchzuführen verwende ich eine kleine Menge an Python-Scripten.

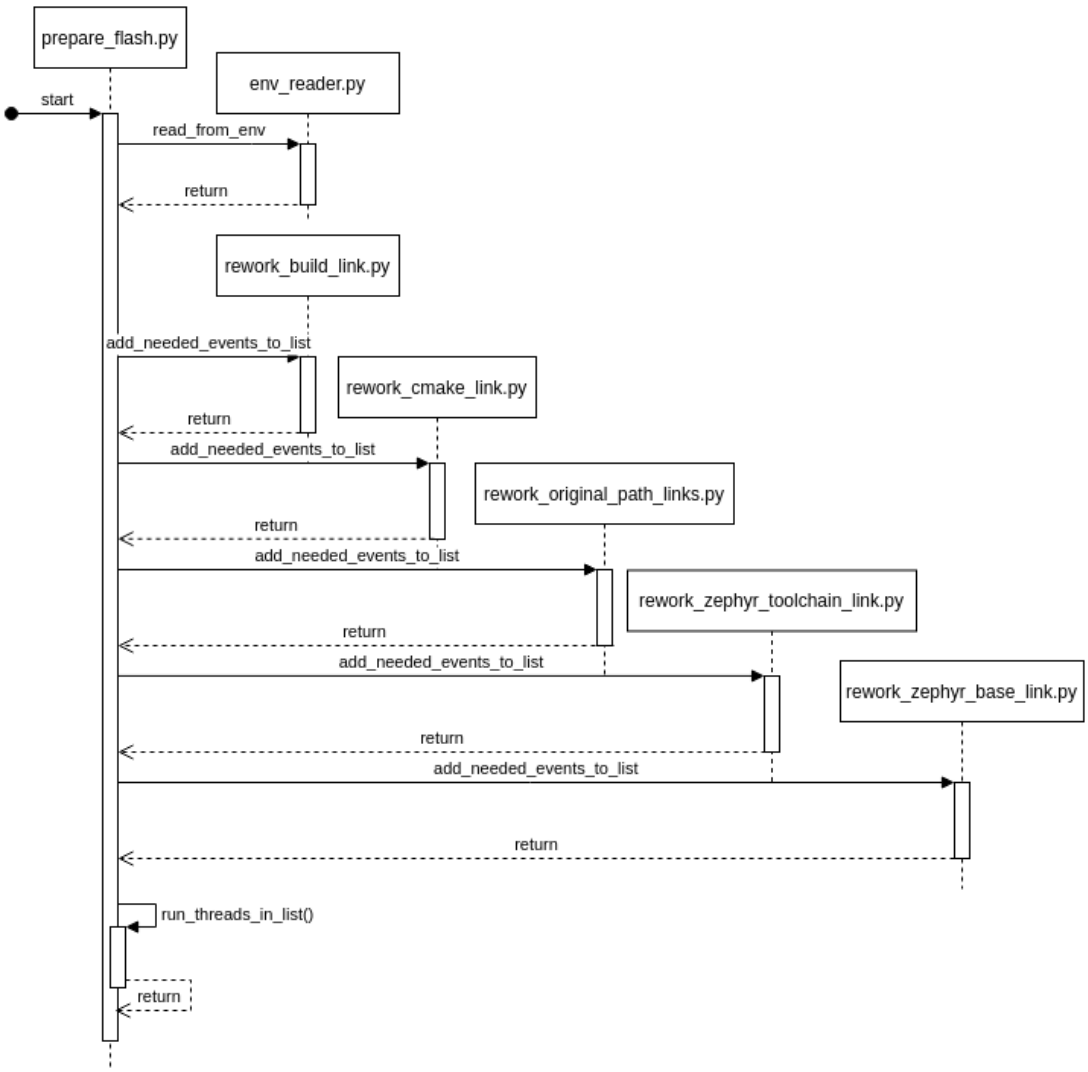
```
├── flash_preparation
│   ├── prepare_flash.py
│   └── rework_scripts
│       ├── remove_build_zephyr_directory.py
│       ├── rework_build_link.py
```
```

```
│       ├── rework_cmake_link.py
│       ├── rework_original_path_links.py
│       ├── rework_zephyr_base_link.py
│       └── rework_zephyr_toolchain_link.py
│
└── specific_utils
    └── env_utils.py
```

prepare_flash.py

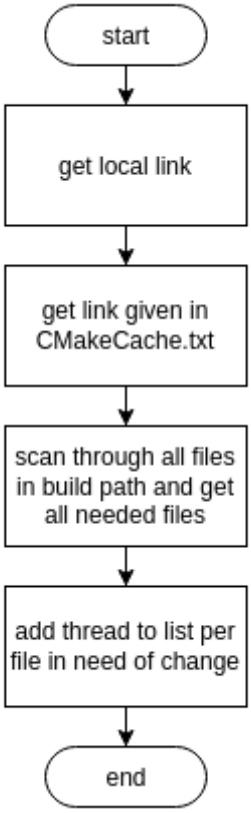
Das Vorbereitungsscript (`prepare_flash.py`) beinhaltet den Startpunkt und durchgeht folgende Schritte:

1. lesen der Konfigurationsdatei .env
2. Vorbereiten der gebrauchten Datei änderungen:
    1. build link
    2. cmake link
    3. project path link
    4. toolchain link
    5. zephyr base link
3. durchführen der gebrauchten Änderungen



`rework` Scripts

Die `rework`-scripts haben alle im allgemeinen den selben Ablauf.



1. `get local link`
   - dieser Schritt kann entweder ganz einfach sein (dieser Wert wird schon mit als Parameter übergeben / er ist immer gleicht), oder nutzt kleine System Calls (Bspw: `whereis cmake`)
   - ```
     # returns the local cmake installation/call path
     def get_local_cmake_install():
         # command to be executed (command finds cmake installation path)
         command = ["whereis", "cmake"]
         # command execution with given output to console put in variable
         res = subprocess.check_output(command)
         # returns first found path
         return res.decode("utf-8").split(" ")[1]
     ```

2. `get link given in CMakeCache.txt`
   - Wie schon oben beschrieben, befinden sich alle gebrauchten Informationen um alle Dateien umzuschreiben in der `CMakeCache.txt` Datei. Um sie dort raus zu bekommen, muss die Datei Zeile für Zeile durchgangen und mit einem `regex`-Pattern abgeglichen werden. Wie die Daten in der Datei gelesen werden sollen, sagt uns Datei selber in den ersten Zeilen.
   - ```
     # The syntax for the file is as follows:
     # KEY:TYPE=VALUE
     # KEY is the name of a variable in the cache.
     # TYPE is a hint to GUIs for the type of VALUE, DO NOT EDIT TYPE!.
     # VALUE is the current value for the KEY.
     ```

- Von hier aus muss nun nur noch der `Key` zur gebrauchten `value` gesucht und gefunden werden.
- ```
  # returning the original dir where the build took place
  def get_original_cmake_install_dir(filepath):
      # Opening the file
      with open(filepath, "r") as file:
          while True:
              # Get next line from file
              line = file.readline()

              # serach in the read line for the needed key
              if re.search(key_to_search_for, line):
                  # clean the line from the key and any newlines and
  return it
                  return line.replace(key_to_search_for, "").replace("\n",
  "")

              # if line is empty
              # end of file is reached
              if not line:
                  print("no original dir found")
                  exit(1)
  ```

3. `scan through all files in build path and get all needed files`
   - Dieser Scan ist sehr primitiv gehalten, kann jedoch in späterer Zeit noch verfeinert werden. Bisher geht er nur duch alle existenten Dateien durch, merk sich jede mit lese und schreib rechten, und gibt diese zurück.
   - ```
     def get_all_filepaths_in_path(local_path="."):
         files = list()
         scan = os.scandir(local_path)
         for entry in scan:
             # noinspection PyUnresolvedReferences
             if entry.is_dir():
                 for inner_entry in get_all_filepaths_in_path(entry.path):
                     files.append(inner_entry)
             elif entry.is_file():
                 files.append(entry.path)

         return files
     ```
   - ACHTUNG: diese Funktion befindet sich in `env_utils.py`
4. `add thread to list per file in need of change`
   - Für jede bei Scan gefundene Datei wir nun ein `thread-object` erstellt, welches den funktions-handle für die Ersetzung, den Dateinamen, der zu ersetzende Wert und den Ersatz-Wert beinhaltet (zuzüglich kann auch noch eingestellt werden, ob eine Konsolen-Ausgabe stattfinden soll, oder nicht)

- ```
  def add_thread_event_to_list(_list, _filename, _old, _new,
  withOutput=False):
      thread = threading.Thread(
          target=replace_all_in_file, args=(_filename, _old, _new,
  withOutput)
      )

      _list.append(thread)
  ```
  - ACHTUNG: diese Funktion befindet sich in `env_utils.py`

3. "flash" des Projektes

Dank der nun durchgagenen Vorbereitungen kann das Projekt direkt ge"flashed" werden. Dafür sind nur ein paar Codezeilen gebraucht.

```
import subprocess

def flash(path):
    env_utils.change_to_dir(path)

    command = ["west", "flash"]  # command to be executed
    res = subprocess.check_output(command)  # system command
    return res.decode("utf-8").split(" ")[1]
    pass
```

Extras

`env_utils.py`

Die `Environment Utilities` bestehen aus unterschiedlichen Codeteilen welche im gesamten Projekt immer öfters verwendet werden.
```

Name	Parameterliste	Rückgabewert	Beschreibung
`get_all_filepaths_in_path`	local_path: str (default: .)	files: Array(str)	Gibt Pfade zu allen schreib/lesbaren Dateien zurück, die sich im gegebenen Pfad befinden
`replace_all_in_file`	filename: str, old: str, new: str, withOutput: boolean (default: False)	None	Geht durch die gegebene Datei durch, und ersetzt alle Vorkommnisse von `old` mit `new` und gibt wahlweise dazu eine Konsolenausgabe
`change_to_dir`	directory: str	None	Ändert den Arbeitsbereich des laufenden Skriptes zum gegebene Pfad (`cd` wird verwendet)
`add_thread_event_to_list`	_list: list(), _filename: str, _old: str, _new: str, withOutput: boolean (default: False)	None	Die gegebenen Daten werden in ein Thread-Objekt gepackt, welches beim starten die Funktion `replace_all_in_file` mit den gegebenen Parametern ausführt, und der angegebenen Liste hinzugefügt
`get_all_in_path`	local_path: str (default: .)	entries: Array(str)	Gibt Pfade zu allen schreib/lesbaren Einträge zurück, die sich im gegebenen Pfad befinden
`remove_dir`	path: str	None	löscht den Ordner des angegebene Pfades(recursively)
`move`	target: str, to :str, overwrite: boolean (default: False)	None	verschiebt Daten von "target" nach "to" (Ziel kann überschrieben werden) wirft FileExistsError, wenn das "to" directory schon existiert aber nicht überschrieben werden soll
`copy_dir`	target: str, to :str, overwrite: boolean (default: False)	None	kopiert einen Ordner("target") nach "to" (Ziel kann überschrieben werden) wirft FileExistsError, wenn das "to" directory schon existiert aber nicht überschrieben werden soll
`create_dir_if_not_existing`	directory: str	None	kreiert einen neuen Ordner, wenn er an der gegeben stelle noch nicht existieret

Name	Parameterliste	Rückgabewert	Beschreibung
`get_all_dir_in_path`	path: str	list(str)	gibt alle existenten Ordner im gegebenen Pfad wieder
`directory_contains_file`	path: str, searched_file: str	boolean	gibt an, ob die angegebene Datei sich irgendwo im Pfad befindet

`env_reader.py`

Der `env_reader` ist ein kleiner File-Scanner, welcher eine Datei einliest, und Daten in Form eines `Dictonary` ausgibt.
Das Format ist ein simples `Key=Value` Prinzip. Das genutzte `regex`-Pattern ist folgendes: `^(.+)=(.+)$`

```
# reads a line in and finds the first instance of a search result for the given
regex pattern
# and then adds it to the directory (key given is the first grouping of the value
pattern)
def set_value_if_existent(_dict: dict, regex: re, string: str):
    for result in regex.finditer(string):
        _dict[result.group(1)] = result.group(2)
    pass


# configuration reader for rework scripts
def read_from_env(path):
    # check if file exists
    if not os.path.exists(path):
        IOError(f"Configuration file could not be found, Path: {path} doesn't
exist")
        exit(1)

    # open file
    with open(path, "r") as file:
        # check if file readable
        if not file.readable():
            IOError(f"Configuration file could not be read, Path: {path} ")

        while True:
            # Get next line from file
            line = file.readline()

            # if line is empty
            # end of file is reached
            if not line:
                return return_dir

            set_value_if_existent(return_dir, regex_value_pattern, line)
```

Beispiel einer .env Datei

```
zephyr=/home/user/zephyrproject/zephyr
cmake=/usr/bin/cmake
toolchain=/home/user/zephyr-sdk-0.13.2

original_projects_path=/home/note/zephyrproject/zephyr/projects
project_holder=/home/note/PycharmProjects/westflashs/projects_holder
```

Welche weiteren Lösungen wurden erwogen, ggf. verworfen und warum?

Es wurden folgene Ideeen während dieses Teils des Projektes erwogen/verworfen:

- "west" forken, und die gegebenen scripte in selber einbauen
  - Vorteile:
    - Es gäbe die Möglichkeit einen Pull Request auf das offizielle "west" Repository zu machen, um, nach weiterer Entwicklung, die Funktion offiziell für jeden zu ermöglichen
  - Status:
    - wurde verworfen
  - Grund:
    - der Ansatz wurde bis zu einem bestimment Teil durchgeführt, jedoch schien dieser Ansatz ab einem bestimment Punkt zu Zeitaufwendig, wurde daher verworfen
- eigenes Tool zum flashen erstellen (west nachbauen)
  - Vorteile:
    - Der Raspberry Pi ist eine sehr "low-powerd" Maschiene, daher wäre eine Umgebung mit weniger unnützen "overhead" vielleicht effizienter, vorallem wenn die Umgebung nur zum flashen geeignet sein soll
  - Status:
    - wurde erwogen
  - Grund:
    - Wegen Zeitaufwand wurde die Idee schon sehr früh verworfen
- erstellen eines Dockers, welches automatisch eine Entwicklungsumgebung mit den nötigen Tools aufbaut um das Projekt auf das angeschlossene Gerät zu flashen
  - Vorteile:
    - Wenn man eine Dockerumgebung für das flashen (und vieleicht auch noch zum builden) verwendet, müssten man keine Umgebung auf dem Host-System installieren. Was viel Zeit spart.
  - Status:
    - wurde erwogen
  - Grund:
    - nicht Teil der Aufgabenstellung (wahrscheinlich zu klein als einzelnes Projekt)

Was wären die nächsten weiterführenden Schritte?

Das hier entworfene und entwickelte Projekt ist sehr simpel gehalten worden, und das zu gutem Grund. Es soll als grundlegendes, in verschiedenen Orten anzuwendendes Tool dienen, welches man in Zukunft für diverse Anwendungsmöglichkeiten weiterentwickeln kann (sei es zur möglichen Reparatur von Projekten ,als schnelles Script zu test zwecken oder zum erstellen eines Dockers, welcher verteilt werden kann, damit jeder die selbe Umgebung besitzt).

Einen weiteren Punkt, den man hier im Hinterkopf behalten sollte, ist, dass diese Scripts von einer einzelnen Person erstellt wurden, welche sich zum Anfangszeitpunkt dieses Projektes noch nie mit "west" befasst hat. Jemand anderes mit mehr Wissen in diesem Feld (vielleicht ein direkter Developer von west oder Zephyr) könnte mit Sicherheit ein besseres Tool bauen. Der hier genutze Ansatz ist nur sehr grob und hat Duzende an Verbesserungsmöglichkeiten. Jedoch durch zeitliche Probleme können diese, aus der Sicht des Authors, nicht hier mit hereingenommen werden.