EmCraft systems

Develop Linux CAN device driver in the Linux i.MX BSPs

Detailed Requirements and Design

rm6798-drad-1_2.doc

RM:	*6798*
Revision:	*1.2*
Date:	*5/12/2024*

TABLE OF CONTENTS

1. OVERVIEW 3

2. REQUIREMENTS 3

2.1. Detailed Requirements 3
2.2. Detailed Non-Requirements 3

3. DESIGN 3

3.1. Detailed Design 3
3.1.1. Design: Demo project 3
3.1.2. Design: Linux CAN Device Driver 4
3.1.3. Design: CANSocket 4
3.1.4. Design: CANSocket Test Suite 4
3.2. Effect on Related Products 4
3.3. Changes to User Documentation 4
3.4. Alternative Design 4

4. TEST PLAN 4

4.1. Secure Download Area 4
4.2. Downloadable Files 5
4.3. Test Set-Up 5
4.3.1. Hardware Setup 5
4.3.2. Software Setup 5
4.4. Detailed Test Plan 7
4.4.1. Test Plan: Demo Project 7
4.4.2. Test Plan: Linux CAN Driver 7
4.4.3. Test Plan: CANSocket 7
4.4.4. Test Plan: CANSocket Test Suite 8

1. Overview

This project develops Linux CAN device driver in the Linux BSP for the i.MX RT processor.

2. Requirements

2.1. Detailed Requirements

The following are the requirements for this project:

1. Provide a Linux demo project combining all the requirements in this project.
    - *Rationale*: Needed to let Customer integrate results of this project into target embedded application.
      Implementation: Section: "Design: Demo project".
      Test: Section: "Test Plan: Demo Project".

2. Develop Linux CAN device driver for the i.MX RT CAN controller.

    - *Rationale*: Explicit Customer requirement.
      Implementation: Section: "Design: Linux CAN Device Driver".
      Test: Section: "Test Plan: Linux CAN Driver".

3. Port CANSocket to the Linux i.MX RT BSP.

    - *Rationale*: Explicit Customer requirement.
      Implementation: Section: "Design: CANSocket".
      Test: Section: "Test Plan: CANSocket".

4. Validate successful execution of the test suite from the SocketCAN package.

    - *Rationale*: Explicit Customer requirement.
      Implementation: Section: "Design: CANSocket Test Suite".
      Test: Section: "Test Plan: CANSocket Test Suite".

2.2. Detailed Non-Requirements

The following are the non-requirements for this project that may otherwise not be obvious:

1. None

3. Design

3.1. Detailed Design

3.1.1. Design: Demo project

This project will enable the required CAN functionality in Linux configuration ("embedded project") called `rootfs`, which resides in a `projects/rootfs` directory, relative to the top of the Linux i.MX RT installation.

3.1.2. Design: Linux CAN Device Driver

The internal i.MX clocks necessary for the CAN interface will be enabled in the kernel. The i.MX RT `flexcan` interface network driver `linux/drivers/net/can/flexcan.c` will be enabled, and the respective changes will be added to the kernel `.dts` file.

The `flexcan` functionality will be enabled in the Linux kernel configuration as follows:

- Go to `Networking support` -> `CAN bus subsystem support` -> `CAN Device Drivers`
- Enable `Platform CAN drivers with Netlink support` (`CONFIG_CAN_DEV`) and `Support for Freescale FLEXCAN based chips` (`CONFIG_CAN_FLEXCAN`)

3.1.3. Design: CANSocket

The CAN socket API, described in details in `linux/Documentation/networking/can.txt`, will be enabled using the `Raw CAN Protocol (raw access with CAN-ID filtering)` and `Broadcast Manager CAN Protocol (with content filtering)` configuration options in the `Networking support` -> `CAN bus subsystem support` configuration menu.

3.1.4. Design: CANSocket Test Suite

The `can-utils` and `can-tests` packages will be used for verification of the functionality implemented in this project.

3.2. Effect on Related Products

This project makes the following updates in the related products:

- None

3.3. Changes to User Documentation

This project updates the following user documents:

- None

3.4. Alternative Design

The following alternative design approaches were considered by this project but then discarded for some reason:

- None

4. Test Plan

4.1. Secure Download Area

The downloadable materials developed by this project are available from a secure Web page on the Emcraft Systems web site. Specifically, proceed to the following URL to download the software materials:

for the i.MX RT1050 BSP release:

- https://www.emcraft.com/imxrtaddon/imxrt1050/can

The page is protected as follows:

- Login: *CONTACT EMCRAFT FOR DETAILS*
- Password: *CONTACT EMCRAFT FOR DETAILS*

for the i.MX RT1170 BSP release:

- https://www.emcraft.com/imxrtaddon/imxrt1170/can

The page is protected as follows:

- Login: *CONTACT EMCRAFT FOR DETAILS*
- Password: *CONTACT EMCRAFT FOR DETAILS*

4.2. Downloadable Files

The following files are available from the secure download area:

- `linux-flexcan.patch` - patch to the Linux kernel sources;
- `projects-flexcan.patch` - patch to the `rootfs` project;
- `rootfs.uImage` - prebuilt bootable Linux image;

Refer to the below sections for the instructions on how to install and use these files.

4.3. Test Set-Up

4.3.1. Hardware Setup

The following hardware setup is required for the i.MX RT1050 boards:

- The i.MX RT1050 EVK board, with the serial console attached as per https://emcraft.com/imxrt1050-evk-board/connecting-serial-console-to-imxrt1050-evk.
- A Linux PC with the `VSCOM USB-CAN` USB to CAN Adapter http://www.vscom.de/vscom-usb-can.html plugged into a USB port on the PC, and the following connections to the i.MX RT1050 EVK board:
    - `DB9.2` of USB-CAN connected to `CANL` J11.3 on the i.MX RT EVK board.
    - `DB9.7` of USB-CAN connected to `CANH` J11.1 on the i.MX RT EVK board.
    - `DB9.3` of USB-CAN connected to `GND` J11.2 on the i.MX RT EVK board.

for the i.MX RT1170 boards:

- The i.MX RT1170 EVK board, with the serial console attached as per https://emcraft.com/imxrt1170-evk-board/connecting-serial-console-to-imxrt1170-evk.
- A Linux PC with the `VSCOM USB-CAN` USB to CAN Adapter http://www.vscom.de/vscom-usb-can.html plugged into a USB port on the PC, and the following connections to the i.MX RT1170 EVK board:
    - `DB9.2` of USB-CAN connected to `CANL` J47.3 on the i.MX RT EVK board.
    - `DB9.7` of USB-CAN connected to `CANH` J47.1 on the i.MX RT EVK board.
    - `DB9.3` of USB-CAN connected to `GND` J47.2 on the i.MX RT EVK board.

4.3.2. Software Setup

The following software setup is required:

1. Download the files listed in Section: "Downloadable Files" to the top of the Linux i.MX RT installation.
2. Install the BSP, as per the respective "Installing and activating cross development environment" document in the "Software" section on the Emcraft site.
3. From the top of the Linux installation, activate the Linux cross-compile environment by running:

```
$ . ./ACTIVATE.sh
```

4. Install U-Boot to the target board.
    - for the IMXRT1050-EVK boards as per https://emcraft.com/imxrt1050-evk-board/installing-uboot-to-imxrt1050-evk-board
    - for the IMXRT1170-EVK boards as per https://emcraft.com/imxrt1170-evk-board/installing-uboot-to-imxrt1170-evk-board
5. From the top of the BSP installation, go to the Linux kernel tree and install the kernel patch, eg:

```
$ cd linux/
$ patch -p1 < ../../linux-flexcan.patch
```

6. From the top of the Linux installation, go to the `projects` sub-directory, and patch the `rootfs` project:

```
$ cd projects/
$ patch -p1 < ../../projects-flexcan.patch
```

7. On the Linux PC intended for execution of the CANsocket test suite, ensure that the following software is installed (Emcraft used Linux PC running the `Fedora 16` (3.1.0-7.fc16.i686.PAE) installation; the other Linux distributives should work too, but may require some additional steps like compilation and installation of the CAN framework kernel modules):
    1. Install `can-utils` package on the Linux PC (commands below are for a Fedora host):

```
$ sudo yum install can-utils
...
$
```

    2. Install and build `can-tests` on the Linux PC:

```
$ cd ~
$ git clone https://github.com/linux-can/can-tests.git
$ cd can-tests
$ make
$ sudo DESTDIR=/usr PREFIX= make install
```

    3. Load the CAN kernel modules on the Linux PC:

```
$ sudo modprobe can
$ sudo modprobe can-raw
$ sudo modprobe slcan
```

8. Connect the VSCOM USB-CAN adapter to the Linux PC and configure it as follows:
    0. Get the VSCOM USB-CAN serial device name (in the example below it is `ttyUSB0`):

```
$ dmesg | tail
[77641.738206] usbcore: registered new interface driver ftdi sio
[77641.739086] usbserial: USB Serial support registered for FTDI USB Serial Device
[77641.747063] ftdi_sio 1-2:1.0: FTDI USB Serial Device converter detected
[77641.747348] usb 1-2: Detected FT232R
[77641.781982] usb 1-2: FTDI USB Serial Device converter now attached to ttyUSB0
[78603.073189] can: controller area network core
[78603.073360] NET: Registered PF CAN protocol family
[78618.877319] can: raw protocol
[78632.316446] CAN device driver interface
```

```
[78632.334423] slcan: serial line CAN interface driver
```

1. Configure the VSCOM USB-CAN adapter to run with a 1Mbps CAN-bus speed (the `-s8` parameter in `slcan_attach`), enable the corresponding network interface:

```
$ sudo slcan attach -o -s8 /dev/ttyUSB0
attached tty /dev/ttyUSB0 to netdevice can0
$ sudo slcand -o -s8 -t hw -S 3000000 /dev/ttyUSB0
$ sudo ifconfig can0 up
```

2. If you have disconnected the VSCOM USB-CAN adapter from the Linux PC, before reconnecting it back run the following command:

```
$ sudo killall slcand
```

4.4. Detailed Test Plan

4.4.1. Test Plan: Demo Project

Perform the following step-wise test procedure:

1. Go to the `projects/rootfs` directory, build the loadable Linux image (`rootfs.uImage`) and copy it to the TFTP directory on the host:

```
$ cd projects/rootfs
$ make
```

2. Boot the loadable Linux image (`rootfs.uImage`) to the target via TFTP and validate that it boots to the Linux shell:

```
=> run netboot
...
TFTP from server 192.168.1.96; our IP address is 192.168.1.86
Filename 'imxrt/rootfs.uImage'.
Load address: 0x80007fc0
Loading: #################################################################
         #################################################################
...
/ #
```

4.4.2. Test Plan: Linux CAN Driver

Perform the following step-wise test procedure:

1. In the kernel bootstrap messages, validate that the CAN driver has been successfully installed and activated:

```
/ # dmesg | grep -i can
CAN device driver interface
can: controller area network core
NET: Registered PF CAN protocol family
can: raw protocol
can: broadcast manager protocol
can: netlink gateway - max hops=1
/ #
```

4.4.3. Test Plan: CANSocket

Perform the following step-wise test procedure:

1. On the target, configure the CAN network:

```
/ # ip link set can0 type can bitrate 1000000
/ # ifconfig can0 up
```

2. Test target to Linux PC transfers:
    - Run the capture utility on the Linux PC:

```
$ candump can0
```

    - Send packets from the target to the host:

```
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.00
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.01
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.02
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.03
```

    - Validate that the packets have been captured on the Linux PC:

```
  can0  12345678   [8]  99 AA BB CC DD EE FF 00
  can0  12345678   [8]  99 AA BB CC DD EE FF 01
  can0  12345678   [8]  99 AA BB CC DD EE FF 02
  can0  12345678   [8]  99 AA BB CC DD EE FF 03
```

    - On the host, stop the capture utility by pressing `Ctrl-C`:

```
^C
$
```

3. Test Linux PC to target transfers:
    - Run the capture utility on the target:

```
/ # candump can0
```

    - Send packets from the Linux PC to the target:

```
$ cansend can0 123abcde#11.22.33.44.56.78.90.01
$ cansend can0 123abcde#11.22.33.44.56.78.90.03
$ cansend can0 123abcde#11.22.33.44.56.78.90.05
$ cansend can0 123abcde#11.22.33.44.56.78.90.07
```

    - Validate that the packets have been captured on the target:

```
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 01
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 03
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 05
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 07
```

    - On the target, stop the capture utility by pressing `Ctrl-C`:

```
^C
/ #
```

4.4.4. Test Plan: CANSocket Test Suite

Perform the following step-wise test procedure:

1. Run the `tst-raw` Linux PC to target test:
    - On the target:

```
/ # tst-raw -i can0
```

- On the Linux PC:

```
$ sudo tst-raw-sendto -i can0
```

- Observe the test data on the target, then press `Ctrl-C` and complete the test:

```
123  [3] 11 22 33
^C
/ #
```

2. Run the `tst-raw` target to Linux PC test:
   - On the Linux PC:

```
$ sudo tst-raw -i can0
```

   - On the target:

```
/ # tst-raw-sendto -i can0
```

   - Observe test data on the Linux PC, then press `Ctrl-C` and complete the test:

```
123  [3] 11 22 33
^C
$
```

3. Run the `tst-packet` Linux PC to target test:
   - On the target:

```
/ # tst-packet -i can0
```

   - On the Linux PC send a packet, then press `Ctrl-C` and complete the test:

```
$ sudo tst-packet -i can0 -s
^C
$
```

   - Observe the test packet on the target, then press `Ctrl-C` and complete the test:

```
123  [2] 11 22
^C
/ #
```

4. Run the `tst-packet` target to Linux PC test:
   - On the Linux PC:

```
$ sudo tst-packet -i can0
```

   - On the target, send a packet, then press `Ctrl-C` and complete the test:

```
/ # tst-packet -i can0 -s
^C
/ #
```

   - Observe the test packet on the Linux PC, then press `Ctrl-C` and complete the test:

```
123  [2] 11 22
^C
$
```

5. Run the `tst-filter` test on the target:

```
/ # tst-filter can0
---
testcase  0 filters : can_id = 0x00000123 can_mask = 0x000007FF
testcase  0 sending patterns ... ok
testcase  0 rx : can id = 0x00000123 rx = 1 rxbits = 1
testcase  0 rx : can id = 0x40000123 rx = 2 rxbits = 17
testcase  0 rx : can id = 0x80000123 rx = 3 rxbits = 273
testcase  0 rx : can_id = 0xC0000123 rx = 4 rxbits = 4369
testcase  0 ok
---
testcase  1 filters : can id = 0x80000123 can mask = 0x000007FF
testcase  1 sending patterns ... ok
testcase  1 rx : can id = 0x00000123 rx = 1 rxbits = 1
testcase  1 rx : can_id = 0x40000123 rx = 2 rxbits = 17
testcase  1 rx : can_id = 0x80000123 rx = 3 rxbits = 273
testcase  1 rx : can id = 0xC0000123 rx = 4 rxbits = 4369
testcase  1 ok
---
testcase  2 filters : can_id = 0x40000123 can_mask = 0x000007FF
testcase  2 sending patterns ... ok
testcase  2 rx : can id = 0x00000123 rx = 1 rxbits = 1
testcase  2 rx : can id = 0x40000123 rx = 2 rxbits = 17
testcase  2 rx : can id = 0x80000123 rx = 3 rxbits = 273
testcase  2 rx : can id = 0xC0000123 rx = 4 rxbits = 4369
testcase  2 ok
---
<...>
---
testcase 15 filters : can id = 0xC0000123 can mask = 0xC00007FF
testcase 15 sending patterns ... ok
testcase 15 rx : can id = 0xC0000123 rx = 1 rxbits = 4096
testcase 15 ok
---
testcase 16 filters : can id = 0x00000123 can mask = 0xDFFFFFFF
testcase 16 sending patterns ... ok
testcase 16 rx : can id = 0x00000123 rx = 1 rxbits = 1
testcase 16 ok
---
testcase 17 filters : can_id = 0x80000123 can_mask = 0xDFFFFFFF
testcase 17 sending patterns ... ok
testcase 17 rx : can id = 0x80000123 rx = 1 rxbits = 256
testcase 17 ok
---
/ #
```

6. Run the `tst-rcv-own-msgs` test on the target:

```
/ # tst-rcv-own-msgs can0
Starting PF CAN frame flow test.
checking socket default settings ... ok.
check loopback 0 recv_own_msgs 0 ... ok.
check loopback 0 recv_own_msgs 1 ... ok.
check loopback 1 recv own msgs 0 ... ok.
check loopback 1 recv own msgs 1 ... ok.
PF_CAN frame flow test was successful.
/ #
```