Qt Embedded Days
13-14 April 2021
bulding devices with Qt

DATA MODUL
THE EMBEDDED EXPERTS
KDAB

Simulator Training for Hardware Pilots

Alexander Trica & Kevin Krammer

Agenda

- About us
- Project/System Overview
- Needs and Challenges
- Simulator Architecture
- Implementation
- Lessons & Conclusions

Qt Embedded Days
13-14 April 2021
bulding devices with Qt

About Us

DATA MODUL
THE EMBEDDED EXPERTS

KDAB

Alexander Trica

- Head of Embedded ARM and PCAP Touch Development at Data Modul
- Project Lead Embedded Hardware and BSP Development

Kevin Krammer

- Senior Software Engineer and Trainer at KDAB
- Technical Project Lead HMI Development

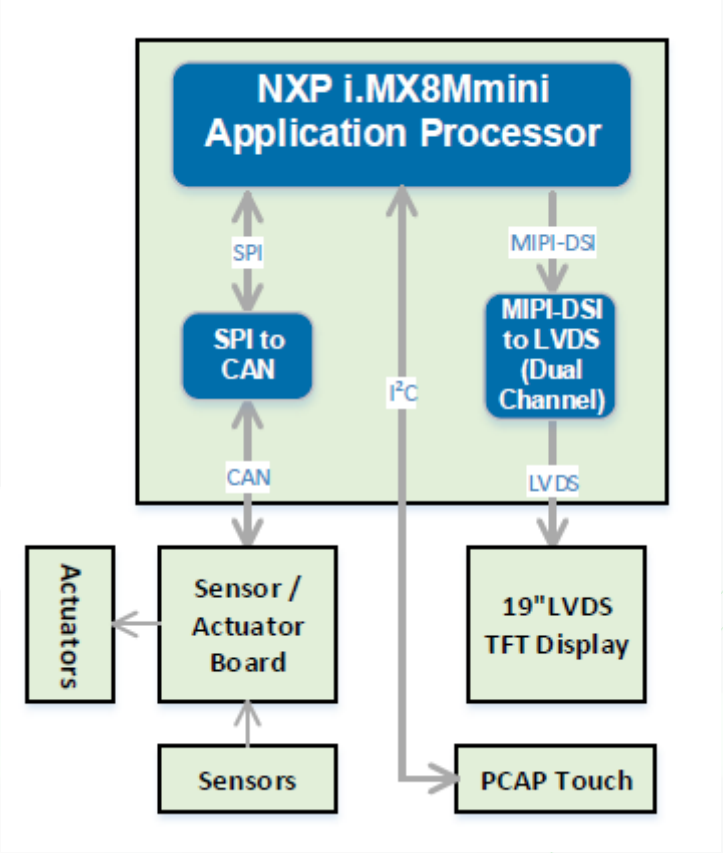
System Design

HW:

- Single Board Computer using i.MX8Mmini NXP application processor running Linux
- 19” bar-type stretched LVDS TFT display
- PCAP-Touch connected via I²C
- Sensor / Actuator Board with Cortex-M4 by STM running FreeRTOS
- Actuators and sensors

Interaction

- User inputs via PCAP Touch based on the UI shown on the TFT display
- Start of application use cases that may trigger actuators or read out sensor data



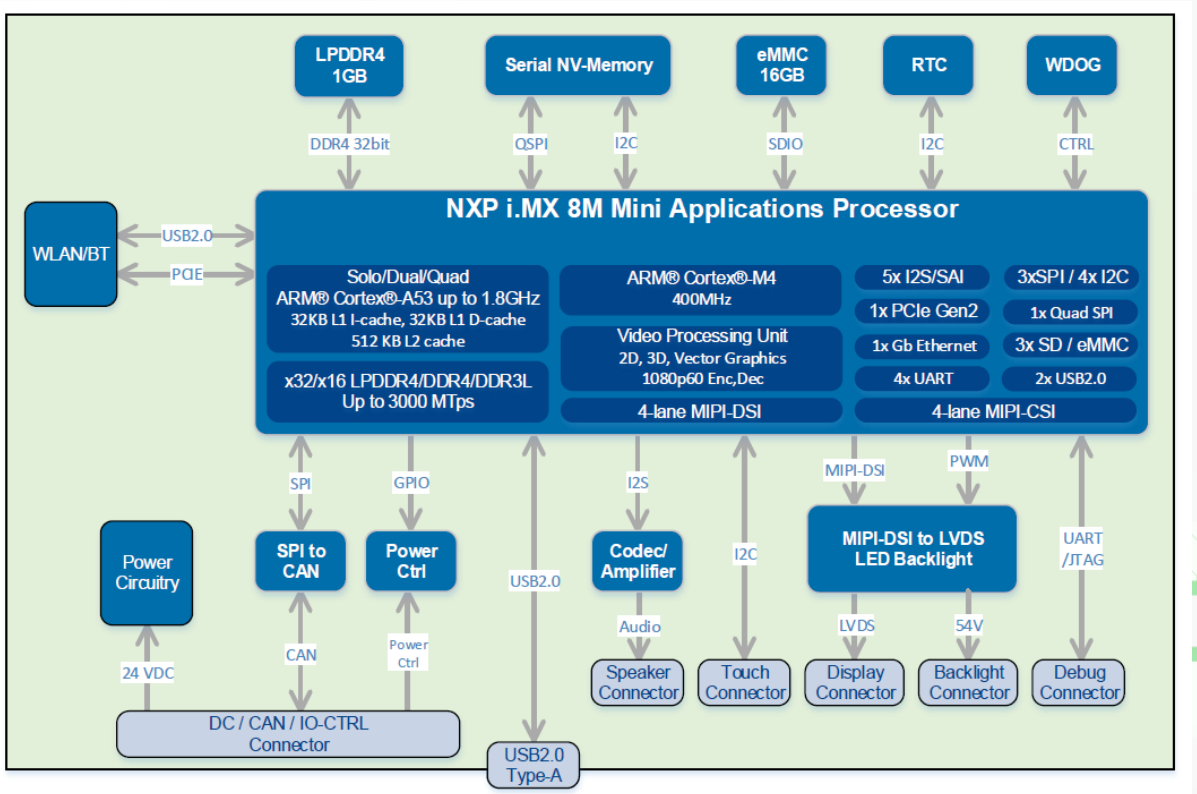
Single Board Computer

HW:

- i.MX8Mmini NXP application processor
- Dual Cortex A53 @ 1.8GHz
- Single Cortex M4
- 1GB LPDDR4
- 16GB EMMC
- 19” bar-type stretched TFT

OS:

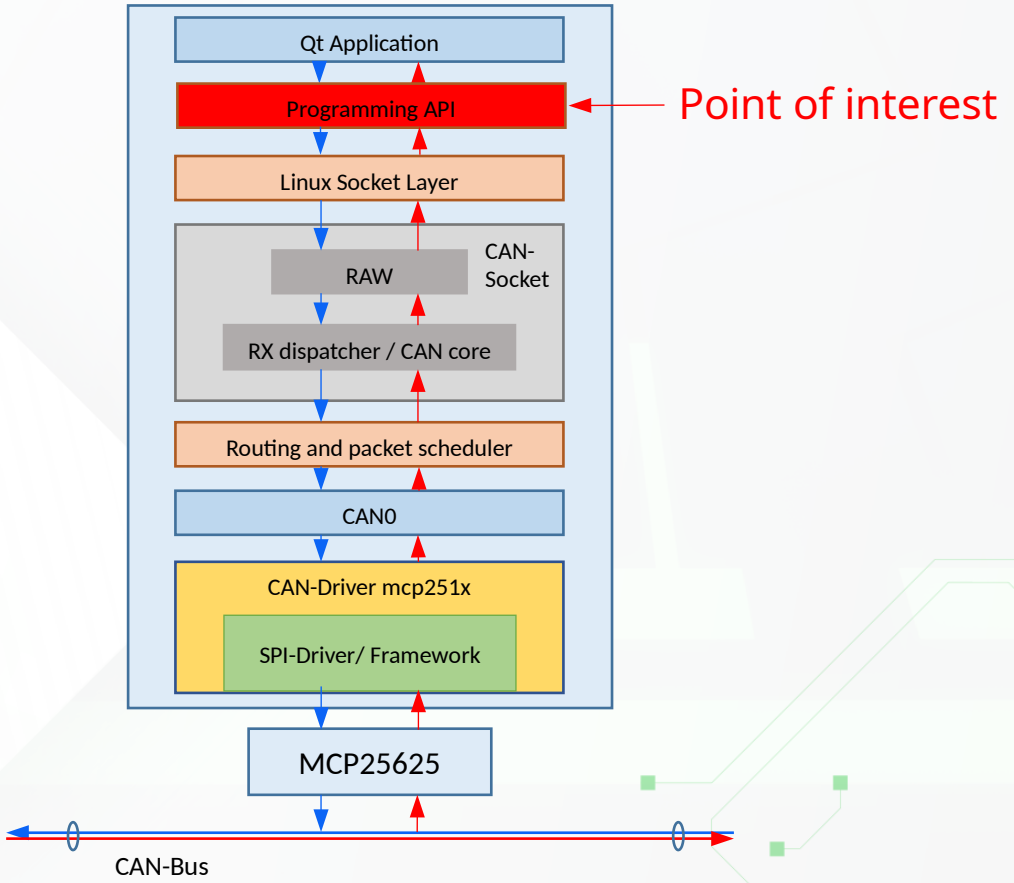
- Mainline Linux kernel 5.10
- YOCTO 3.1 (Dunfell, LTS)
- Qt 5.12



CAN Interface on OS Level

HW:

- SPI-to-CAN since the i.MX8Mmini has no native CAN interface (like the i.MX6)
- SPI Framework needed because of MCP25625
- CAN Socket has been added to the existing Linux networking subsystem
- Socket layer as highest layer in kernel space
- API abstraction layer that handles a higher level protocol between actuator/sensor board and application
- RPC calls for actuators
- Reading data points for sensor data



Based on https://www.can-cia.org/fileadmin/resources/documents/proceedings/2012_kleine-budde.pdf and extended by Alexander Trica and Elmar Albert to cover for MCP25625 and programming API

Needs & Challenges

- Automated Testing
  - Be able to run Unit/Component/System Tests of code that communicates with the actuator/sensor board
  - For Blackbox UI Testing with Squish
  - Be able to run these on the CI in virtualized environments
- Faster development by avoiding build for and deploy to target
- Better control over values, timing, errors than HW provided demo mode

Needs & Challenges

- HW is in development during application SW development
    - Development on the latest HW is not always possible
- CAN communication protocol is under development during application SW development
- 4 sets of hardware vs. 8 developers in 3 different countries
    - Amplified by Covid-19 and working from home

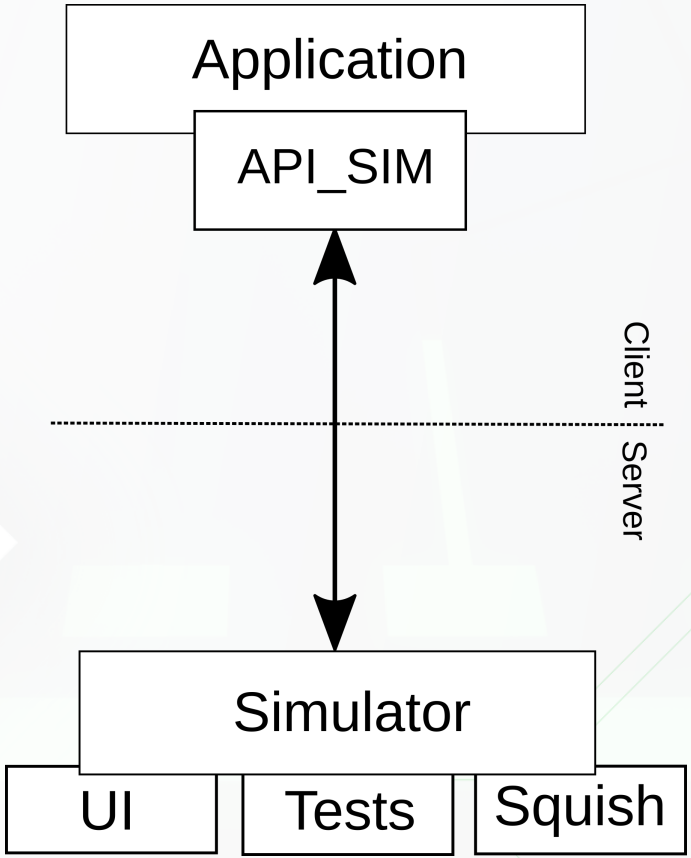
Simulator Architecture

Server/Client

- Simulator has Server role
- Can be run in-process and out-of-process
- Can use simulator desktop UI with SW deployed on target
- Simulator can keep state across SW restarts

Frontend/Backend

- Functionality in shared library
- QtWidgets based UI
- QObject API with signals for requests
- Property API for Squish integration



Implementation

Communication

- TCP/IP
- Google Protocol Buffers
- Simple, custom message framing

Build-Time API Replacement

- API helpfully already as C++ Interfaces
- Simulator Client implements these
- Build selected with QMake config option
- #ifdef only when necessary

```
1  boardsimulator {
2      SOURCES += boardapi_simulator.cpp \
3                 sim.pp.cc
4
5      DEFINES += SIMULATOR_ENABLED
6
7      LIBS += lib/libprotobuf.a
8  }  else {
9      LIBS += -lboardapi # board vendor library
10 }
```

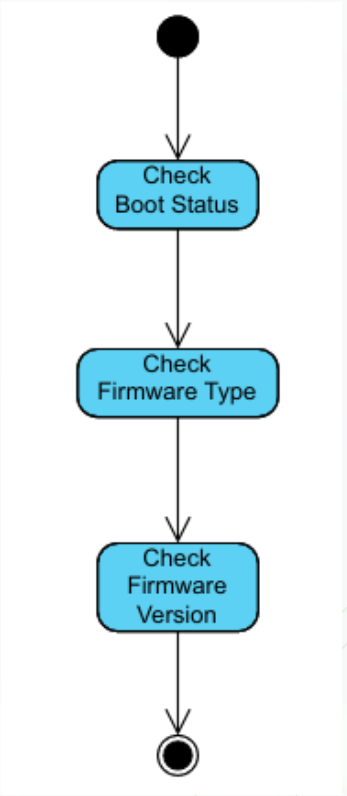
Example

Synchronization between HMI and Board

- Request/Response driven State Machine
    - HMI sends requests
    - Board/Simulator responds

In this example

- Board has already booted
- It is running the Production firmware
- It is running the right firmware version
- Code snippets are simplified
    - Only looks at the firmware type request/response



API


```
1  class BoardApiIface
2  {
3      virtual void getFirmwareTypeInformation(std::function<void(bool success, const FirmwareType&)>) = 0;
4  };
```

```
1  class BoardApi : public BoardApiIface
2  {
3      void getFirmwareTypeInformation(std::function<void(bool success, const FirmwareType&)>) override;
4  };
```

```
1  class BoardSimulator : public QObject
2  {
3      Q_OBJECT
4  public:
5      void sendFirmwareTypeResponse(bool success, const FirmwareType&);
6
7  signals:
8      void firmwareTypeRequestReceived();
9  };
```

Protobuf


```
package sim;

message GetFirmwareTypeRequest { }

message GetFirmwareTypeResponse {
    bool success = 1;
    uint32 firmwareType = 2;
}
```

```
namespace sim
{
    class GetFirmwareTypeRequest : public google::protobuf::Message
    {
    };

    class GetFirmwareTypeResponse : public google::protobuf::Message
    {
    public:
        bool success() const;
        void set_success(bool value);

        uint32_t firmwaretype() const;
        void set_firmwaretype(uin32_t value);
    };
}
```

Client Implementation


```
1  void BoardApi::getFirmwareTypeInformation(std::function<void(bool success, const FirmwareType&)> c)
2  {
3      m_firmwareTypeCallback = c;
4
5      sim::GetFirmwareTypeRequest s;
6      sendMessage(s);
7  }
8
9  void BoardApi::simulatorResponseReceived(const google::protobuf::Any &any)
10 {
11     if (any.Is<sim::GetFirmwareTypeResponse>()) {
12         sim::GetFirmwareTypeResponse s;
13         any.UnpackTo(&s);
14
15         if (m_firmwareTypeCallback != nullptr) {
16             m_firmwareTypeCallback(s.success(), static_cast<FirmwareType>(s.firmwaretype()));
17             m_firmwareTypeCallback = nullptr;
18         }
19     }
20 }
```

Simulator Implementation


```
1  void BoardSimulator::sendFirmwareTypeResponse(bool success, const FirmwareType &type)
2  {
3      sim::GetFirmwareTypeResponse s;
4      s.set_success(success);
5      s.set_firmwaretype(static_cast<int>(type));
6      sendMessage(s);
7  }
8
9  void BoardSimulator::apiRequestReceived(const google::protobuf::Any &any)
10 {
11     if (any.Is<sim::GetFirmwareTypeRequest>()) {
12         emit getFirmwareTypeRequestReceived();
13     }
14 }
```

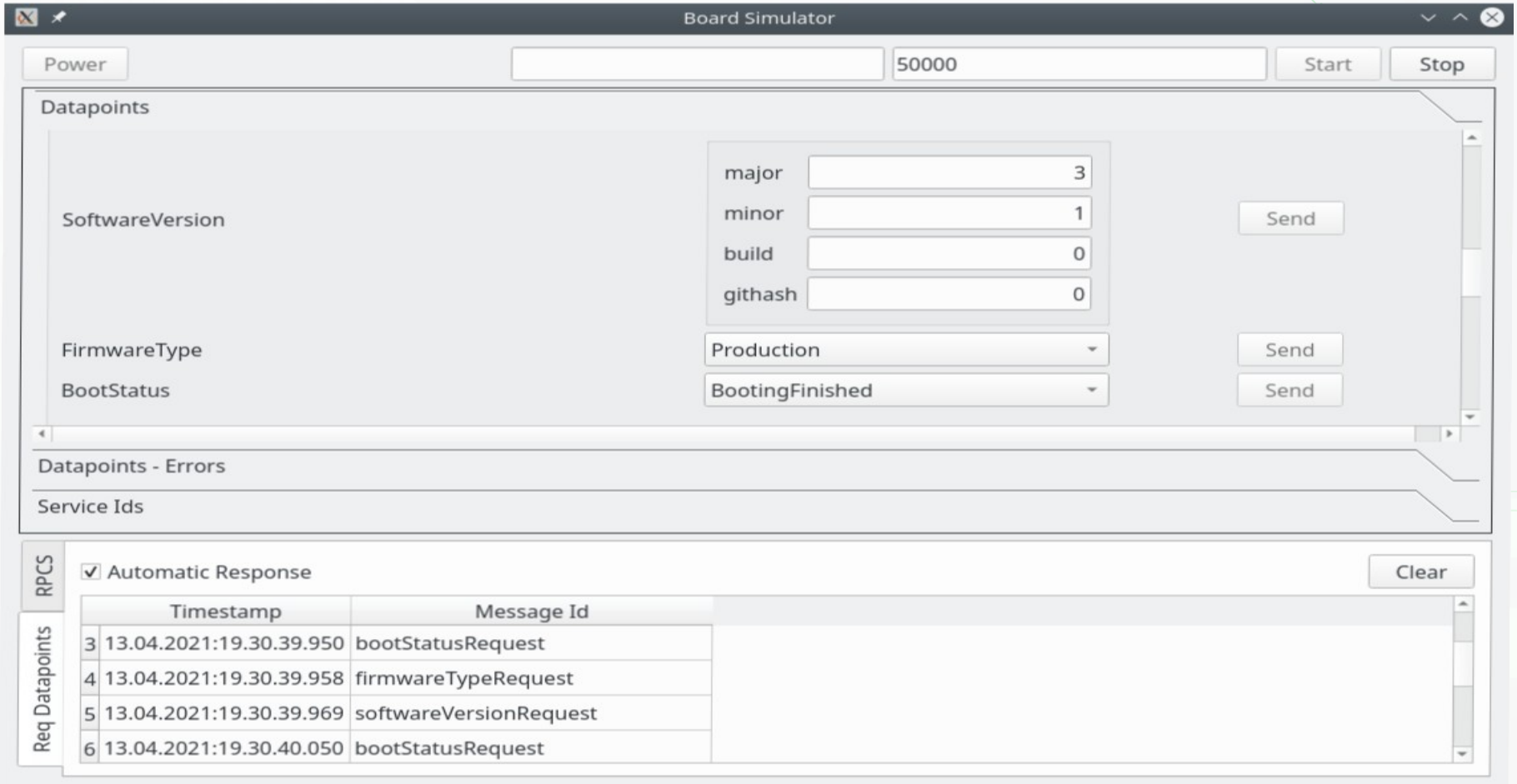
Use by State Machine


```
1  QState *checkFirmwareTypeState = new QState();
2  m_machine->addState(checkFirmwareTypeState);
3
4  connect(checkFirmwareTypeState, &QState::entered, this, [=] () {
5
6      m_api.getFirmwareTypeInformation([=] (bool success, const FirmwareType &data) {
7          if (data != Production) {
8              emit firmwareTypeCheckMismatch();
9          } else {
10             emit firmwareTypeCheckDone();
11         }
12     }
13 }
14
15 checkFirmwareTypeState->addTransition(this, &SyncStateMachine::firmwareTypeCheckDone, checkFirmwareVersionState);
16 checkFirmwareTypeState->addTransition(this, &SyncStateMachine::firmwareTypeCheckMismatch, flashBoardState);
```

Use by Unit Test


```
1  // SIM: setup
2  QSignalSpy firmwareTypeRequestSpy(m_simulator, &BoardSimulator::firmwareTypeRequestReceived);
3
4  // SYNC: setup
5  QSignalSpy firmwareTypeDoneSpy(stateMachine, &SyncStateMachine::firmwareTypeCheckDone);
6
7
8  // SIM: check that the state machine requested the firmware type
9  QTRY_COMPARE_WITH_TIMEOUT(firmwareTypeRequestSpy.count(), 1, kWAIT);
10
11 // and answer with production firmware
12 m_simulator->sendFirmwareTypeResponse(true /*success*/, Production);
13
14
15 // SYNC: check we have received firmware type
16 QTRY_COMPARE_WITH_TIMEOUT(firmwareTypeDoneSpy.count(), 1, kWAIT);
```

Simulator UI



Lessons & Conclusions

- Works very well!
  - Solves the HW availability issues for most development needs
  - Great coverage of various scenarios with automated tests
- TCP/IP a bit problematic for parallel test execution
  - Maybe consider a simple transport layer using delayed method invocation as a build-time or run-time option
- Google Protocol Buffers easy to use, a bit tricky as a dependency
  - Pre-Built binaries (platform & compiler specific) in VCS
  - Would be nice to have something with integrated build or package managed

Thank You!

ATrica@data-modul.com

kevin.krammer@kdab.com