Dynamic Partial Reconfiguration for Fault-Tolerance in automotive ECUs

384.157 Labor SoC Design, WS 2018

Constantin Schieber	1228774
Rupert Schorn	1325700
Andreas Hirtenlehner	1327273
Peter Schober	1355178

March 5, 2019

Contents

1 Introduction **1**

1.1 Design Overview 1

1.2 Required tools, IPs and packages 3

2 Cortex M1 on Xilinx **5**

2.1 Usage of Cortex M1 in Vivado 5

2.1.1 Code via Memory Initialization File 5

2.2 How to mbed OS 5

3 Communication Protocol **7**

3.1 Protocol 7

3.2 Bus monitor 8

4 Test setup **10**

4.1 Communication link 10

4.2 ECU, THS, MCU and FBU 11

4.3 Connections 11

5 Partial Reconfiguration **12**

5.1 Limitations imposed by partial reconfiguration 12

5.1.1 No block diagram support 12

5.1.2 Global synthesis / implementation runs are mandatory 12

5.1.3 processor configuration access port (PCAP) / internal configuration access port (ICAP) on the Zynq-7000 13

5.1.4 ICAP primitive instantiation 13

5.1.5 Read from SD-Card 13

5.2 Integration Overview 14

5.2.1 Packaged IP 14

5.2.2 Loading the Partial Reconfiguration (PR) module 14

6 Results **16**

6.0.1 Resource Usage 16

6.0.2 Open Issues 16

6.0.3 Instability of Voltage Supply during PR 17

7 Conclusion **18**
7.1 Future Work 18

Abstract

In this lab, fail-safe mechanisms for Electronic Control Units (ECUs) are explored on the basis of PR in Field Programmable Gate Arrays (FPGAs). The introduced design contains typical characteristics of an automotive system. Communication between the different sub-systems is realized by a specific link (including protocol) and connects the FPGA to the ECUs that handle operations during normal conditions. ECUs are monitored by a bus monitor in the FPGA for nominal operation characteristics and can be replaced by the instantiation of the very same ECU within the FPGA in case of failure.

1 Introduction

Our ECUs are based on ARM Cortex-M1/M3 controllers. Both of them became recently available as an IP-package for Xilinx based FPGAs. By using a common hardware base and a common software middleware we were able to streamline the development of the control-software which enables a faster development process for new applications and reduces the overhead of their functional verification.

1.1 Design Overview

The assumed system consists of three regular separate control units that are connected through a communication link. Figure 1.1 shows the basic components of the assumed design.

Figure 1.1: Basic concept for Failsafe ECU

The realized scenario reads a throttle position and controls an engine after some data conversion. The ECU is responsible for gathering the throttle position data, measured and provided by the throttle sensor (THS). After the ECU converts the throttle position data into engine control data, it forwards these data to the motor control unit (MCU), which is responsible for controlling the engine. A fourth unit acts as a fallback unit in case of a failure of one of the regular control units. This fallback unit is realized by using

PR in an FPGA. Figure 1.2 displays a sequence diagram assuming normal operation. In this case, the fallback unit doesn't have to perform any active functionality, it just has to passively monitor the data transfer on the communication link to detect possible errors.

Figure 1.2: Sequence diagram for normal operation

The fallback unit has to monitor all transferred data on the bus and detect any failures (e.g. timeout, error flags, ...). Once an error is detected, a certain partial reconfiguration is triggered by the bus monitor module, which is part of the fallback unit. After the reconfiguration is finished, the fallback unit completely takes over the functionality of the faulty component. Due to the fail silent assumption, the faulty device will not affect the behaviour of the system. Figure 1.3 shows a sequence diagram including a faulty MCU. The bus monitor detects that the MCU is running into a timeout and triggers a PR to take over its functionality. After finishing the reconfiguration, normal operation takes over again (see figure 1.2), but the fallback unit serves as MCU now.

Figure 1.3: Sequence diagram for MCU failure

1.2 Required tools, intellectual properties (IPs) and packages

- Vivado 2018.3 (with Partial Reconfiguration license)
- Keil Microcontroller Development Kit (Windows only)

- ARM Cortex M1 IP for Vivado
- UART IP by Martin Mosbeck
- ARM Mbed OS

2 Cortex M1 on Xilinx

Description of the setup of the Cortex M1 core. [1] [2]

2.1 Usage of Cortex M1 in Vivado

Only global implementation / synthesis runs are permitted to obtain a working bitstream. If OOC (Out of Context) runs are used, everything except for the Cortex M1 will work fine. The Cortex M1 will enter a hard-fault state which does not allow recovery. This is indicated by a high bit on the *Lockup* port of the processor.

Also trivial, follow tutorial that is provided on ARM Website and adapt to own needs.

2.1.1 Code via Memory Initialization File

To update the Memory Initialization File for a core, a new package with the updated *bram_a7.hex* file in the corresponding core (*/cores/cm1_*/m1_for_arty_a7.srcs/sources_1/imports/m1_for_arty_a7/m1_for_arty_a7/* [3]) has to be generated and the IP in the block design has to be updated.

2.2 How to mbed OS

To achieve a common code base for the application code, Mbed OS, a real-time operating system (RTOS) from ARM, was used (Fig 2.1). Since there is no support for the Xilinx Cortex M1 core, the board support package (BSP) provided by Xilinx was ported to Mbed OS [4]. The port includes the RTOS kernel and the GPIO and UART modules. The firmware for our FPGA design [**?**] can be found on GitHub [5]. For using Mbed OS with a new design, the BSP has to be exported from the Vivado SDK and copied to */mbed-os/targets/TARGET_Xilinx/TARGET_XC7Z020/device/standalone_bsp/* in [5]. Support for targeting the STM-Boards with *Visual Studio Code* and *gcc* was added due to the code size limitations that are imposed through the Keil platform.

Figure 2.1: Software structure.

3 Communication Protocol

This section describes the communication protocol used for command and data exchange between bus participants. Further, the bus monitor module, which is part of the fallback unit (FBU) is explained.

3.1 Protocol

In initial system designs the usage of the CAN protocol was intended. This approach was discarded, because no license for the Xilinx CAN IP core could be obtained. Since no stable open source CAN IP core was found, a new protocol was defined.

In our approach a simple master-slave protocol was defined to enable communication between the master (ECU, FBU) and the slaves (THS, MCU, FBU). The protocol consists of command and data packets, each of them contains just one byte. No error protection measures were taken into account. Each bus participant needs a unique address. Since this design just contains three control units, two bits as address field are sufficient. Figure 3.1 shows the composition of a control and data packets.

Figure 3.1: Composition of control and data packets

A control packet is indicated by two leading zeroes, followed by destination address (DEST) and source address (SRC). The last two bits (CTRL) determine a further classification of the control packet, as shown in table 3.1.

A control packet can be used to request data, send an acknowledge or indicate errors.

CTRL		Function
0	0	Request data
0	1	Acknowledge
1	0	Error 1
1	1	Error 2

Table 3.1: Control packet classification

Only the master is able to send a request data command, the remaining control packet classifications can be used by the slaves, too.

A data packet starts with the destination address (DEST), followed by a 6 bit wide data field (DATA). It can be sent by the master and by slaves (if a request data command was received previously). If sent by a master, the slave has to answer with an acknowledge.

Since no measures to detect or correct data transmission errors are used, a sending retry mechanism is implemented. If the master doesn't receive an answer from a slave, it re-sends the packet up to a configurable number of attempts.

3.2 Bus monitor

The bus monitor module is part of the FBU and is responsible for monitoring the traffic on the communication link to detect faulty units. Therefore it has to implement the protocol described in section 3.1. The bus monitor is implemented as VHDL module and is implemented in the static area of the FPGA. It detects the following incidents:

- Timeout occurence
- Faulty packets
  - Wrong address
  - Error flag set (CTRL field)

Seperate timeouts are configurable for master and slaves. The bus monitor also implements the retry mechanism according to section 3.1.

The interfaces between the bus monitor, the reconfiguration controller and the bus is shown in figure 3.2.

Since the system uses UART as physical communication layer (see section **??**), the UART IP (Martin Mosbeck) is used to retrieve data from the bus, which is forwarded to the watchdog unit within the bus monitor module. The watchdog is responsible for error and timeout detection. In case of an error or timeout detection, the address of the faulty unit is passed to the reconfiguration controller (using one-hot coding) to trigger a PR. A gate circuit is used to pass the universal asynchronous receiver transmitter (UART) receive (Rx) and transmit (Tx) line to the dynamic area once a reconfiguration was triggered. This is a safety measure to just allow transmission by the FBU if a faulty unit has to be replaced. It is assumed, that just one bus participant can be replaced by the

Figure 3.2: Bus monitor interfacing

FBU during system runtime, so once a reconfiguration was done, the bus monitoring is stopped.

The bus monitor is also used to control output pins of the FBU if no bus participant has to be replaced, or if a bus participant that doesn't require any output pins is replaced. This is necessary, because the FPGA which is used in this test setup pulls (unconfigured) output pins to low, which is not acceptable in our system (see section **??**).

4 Test setup

This section describes the used evaluation boards and bus transceivers for the test setup. Further, the test harness, realized by a breadboard is described. Figure 4.1 shows a picture of the test setup, containing the breadboard (ECU, THS, MCU, bus transceivers, switch, LED) and the Zedboard (FBU). All components are supplied by the $3.3V$ power supply on the Zedboard.

Figure 4.1: Zedboard and STM32 hardware setup

4.1 Communication link

In this approach CAN bus transceiver breakout boards, based on SN65HVD230 [6] are used as physical layer below the communication protocol, described in section 3.1. Figure 4.2 shows a basic design for the physical bus setup.

Each bus participant uses a CAN bus transceiver module to retrieve data from the (differential) bus through a UART interface.

Figure 4.2: CAN bus transceivers

4.2 ECU, THS, MCU and FBU

The ECU, THS and MCU components are all implemented on Nucleo-64 Boards. The bus master (ECU) is based on a Cortex M3 core [7] available on a Nucleo-64 F103RB board [8], whereas the slaves (THS, MCU) are based on Cortex M0 cores [9] on Nucleo-64 F072RB boards [8].

As sensor input a simple push button switch (simulating throttle position) is used, whereas as actuator output (simulating engine data) a LED is used.

The FBU is implemented on a Zedboard [10], which is a Zynq 7000 Evaluation Board.

4.3 Connections

To connect all Nucleo-64 boards, bus transceivers, push button and LED a breadboard was used. The ZedBoard and the breadboard are connected via a ribbon cable (Agile Mixed Signaling connector on the Zedboard). Since the FBU must be able to use the sensor (push button) and actuator (LED), those peripherals are shared between THS and FBU (sensor) and between MCU and FBU.

5 Partial Reconfiguration

This section reasons about design choices and encountered obstacles during the development process.

5.1 Limitations imposed by partial reconfiguration

PR does impose some limitations on the design process, a brief description of the encountered limitations and how they were handled is given in the following.

5.1.1 No block diagram support

The PR workflow as implemented by Xilinx in Vivado does not allow the Reconfigurable Partition (RP) to be present in a block diagram. Only hdl files are eligible for the PR process.

To solve this problem without the loss of comfort that is provided by the usage of block diagrams (mainly the connection of different signals between modules) we decided to transfer our Cortex Module into an IP-Package. This IP-Package can then be instantiated as a register-transfer-level (RTL) module alongside an existing block diagram. Only the signal connections between the IP and the block diagram have to be declared manually then.

Figure 5.1: Enable PR by the separation of the block diagram and the RP

Due to this, the originally planned AXI-Bus interface for the RP needs to be connected manually. This has proven to be very error-prone in earlier evaluation steps and was then replaced by an interface that only exposed required signals. An AXI-Bus interface may very well work though, as it probably failed due to other, unrelated errors.

5.1.2 Global synthesis / implementation runs are mandatory

PR works with global synthesis and implementation runs only. This imposes restrictions on the naming of files within packages. While out-of-context (OOC) runs allow the same instance names for different IPs a global run requires distinct file and instance names for everything that is included in the project.

Global synthesis and implementation runs are also necessary to include the memory initialization file for the Cortex processor.

5.1.3 PCAP / ICAP on the Zynq-7000

As the Zynq-7000 was used for the prototyping process our first choice was the usage of the PCAP for writing partial bitstreams to the FPGA as it is the most straight forward (and well documented) way for this platform.

Due to design considerations (scaling well for production vs fast prototyping) a switch to using the ICAP was made. For this, the PCAP needs to be actively deactivated after the boot of the processing system ([11], page 218).

5.1.4 ICAP primitive instantiation

Instantiation of the ICAP as a hardware primitive in VHDL is documented in the 7series libraries guide ([12], page 178). For completeness sake, the component definition is listed below as it does not exist directly in the documentation. The comments the generic parameters that were used in this project.

```
component ICAPE2 is
generic (
    DEVICE_ID    : std_logic_vector(31 downto 0); -- X"23727093"
    ICAP_WIDTH   : string; --"X32"
    SIM_CFG_FILE_NAME    : string -- "None"
);
port (
    O      : out std_logic_vector(31 downto 0);
    CLK    : in std_logic;
    CSIB   : in std_logic;
    I      : in std_logic_vector(31 downto 0);
    RDWRB    : inout std_logic
);
end component ICAPE2;
```

Listing 5.1: Component definition of the ICAPE2.

5.1.5 Read from SD-Card

The SD-Card was used to provide a bootable image with the default configuration and all partial bitstreams. Reads from the SD-Card may fail, resulting in a non-functioning or only partially functioning system. The following steps should be performed to troubleshoot the problem.

- If the system is working in its original configuration and only fails the partial reconfiguration, the file names of the bitstreams should be checked. Only a maximum of 8 characters (+3 for the extension) for the file name are permitted by default.
- Slow (e.g. overwriting everything with zeros) reformat of the SD-Card may be tried.

- Smaller SD-Card should be tried.

5.2 Integration Overview

Due to our problems with the AXI-Bus it was decided to put all relevant peripheral communication into the RP. This eliminates the need of implementing an AXI-Interface at the cost of an increased size of the RP.

The following modules are included in the RP:

- AXI-GPIO
- AXI-UartLite
- Cortex M1

Figure 5.2: Data path for loading a partial bitstream

[13], [14] Usage of ICAP.

5.2.1 Packaged IP

The PR IP was packaged according to the guide in [15]. To avoid naming conflicts, all block designs of the different IPs need to be named differently before this process.

5.2.2 Loading the PR module

The module is loaded through the PR-Controller IP that is provided by Vivado. This module is connected via the AXI memory interface and loads the partial bitstream from DDR. At start-up, the partial bitstreams are loaded form the SD-Card into the DDR through the Zynq Processing System (PS) as it already needs to be used for the deactivation of the PCAP. Figure 5.2 shows how the different modules and memory entities are related to each other. Listing 5.2 shows the currently assigned address space.

```
#define PARTIAL_ECU_ADDR        0x400000
#define PARTIAL_THROTTLE_ADDR    0x800000
#define PARTIAL_ENGINE_ADDR      0x120000
#define PARTIAL_BLANK_ADDR      0x160000

```

```
/* In Words */
#define PARTIAL_ECU_BITFILE_LEN 736718
#define PARTIAL_THROTTLE_BITFILE_LEN 736718
#define PARTIAL_ENGINE_BITFILE_LEN 736718
#define PARTIAL_BLANK_BITFILE_LEN 736718
```

Listing 5.2: Address and Length of the partial bitstreams in DDR memory.

6 Results

Results would usually encompass benchmarks on the following things:

- Time to reconfigure
- Time to fault detection
- Time to fault mitigation
- Fault Mitigation success rate

Time constraints and a hardware set-up that is not yet ready to be benchmarked extensively (e.g. benchmarking infrastructure missing, stability problems) only allow us to give us a basic overview over the used resources.

6.0.1 Resource Usage

The design currently uses around 11% of the Zynqs available hardware resources, a detailed key is given in table 6.0.1. Most of the resource usage can be accounted to the RP as it contains the Cortex-M1 with its high demand for BRAM elements. This demand is easily visible in the implemented design (figure 6.1).

6.0.2 Open Issues

- Partial reconfiguration fails for one bitstream and results in a voltage drop, see section 6.0.3.
- Partial Reconfiguration Controller only works with a manual reset after start-up

Type	Used	Available	Percent [%] used
LUT	4827	53200	9.07
LUTRAM	289	17400	1.66
FF	4832	106400	4.54
BRAM	16	140	11.42
DSP	3	220	1.36
IO	25	200	12.50
BUFG	3	32	9.37
MMCM	1	4	25.00

Table 6.1: Resource utilization, post-implemntation

Figure 6.1: Implemented design, RP in the right bottom corner.

6.0.3 Instability of Voltage Supply during PR

During the process of partial reconfiguration for the Cortex M1 engine module a voltage drop (2V) occurs on the hardware setup.

Figure 6.2

7 Conclusion

This work has explored the possibility of using partial reconfiguration in FPGAs to provide efficient redundancy in an automotive system. Instead of providing a redundant hardware entity for every critical module, one single FPGA provides dynamic redundancy for each of these modules. To avoid over-commitment with regards to resource usage (space and power) partial reconfiguration is used. By using the newly available Cortex M1 IPs, a streamlined software development process is possible. The same software can be executed on the cores in the FPGA as well as on the actual hardware with minimal adaptions in the build-process. This reduces the amount of testing and tool-chain adaptions that need to be performed. We demonstrated these concepts on the Zynq-7000 and with three Cortex M1 Central Processing Units (CPUs) that were connected with a bus.

7.1 Future Work

Based on this work a more heterogeneous set of critical hardware could be provided with redundancy. A good first step would be to include the Cortex M3 CPU that couldn't be included in this project due to time constraints. The bus monitor could be extended with a more sophisticated fault detection algorithm, which could also mean to employ a more sophisticated bus protocol. Measurements with regards to the system performance (e.g. time to reconfigure, time to detect fault, time to mitigate fault, power usage ...) should be considered also. Furthermore the Mbed OS port could be extended by other hardware modules. A first draft for the Timer and I2C modules ia already implemented.

Bibliography

[1] ARM, "Arm® Cortex®-M1 DesignStart™ FPGA-Xilinx edition User Guide," 2018. [Online]. Available: https://static.docs.arm.com/100211/0001/arm_cortex_m1_designstart_fpga_xilinx_edition_ug_100211_0001_00_en.pdf?_ga=2.121336076.523725493.1550477241-1711067005.1549732081

[2] ——, "Cortex-M1 Technical Reference Manual." [Online]. Available: https://static.docs.arm.com/ddi0413/d/DDI0413D_cortexm1_r1p0_trm.pdf

[3] C. Schieber, R. Schorn, A. Hirtenlehner, P. Schober, "FPGA Design Repository," 2019. [Online]. Available: https://github.com/SoC-Lab/zedboard_cortex_m1

[4] A. Hirtenlehner, "Mbed OS Port for Xilinx Cortex M1," 2019. [Online]. Available: https://github.com/andi-h/mbed-os/tree/XC7Z020

[5] C. Schieber, R. Schorn, A. Hirtenlehner, P. Schober, "MCU Firmware Repository," 2019. [Online]. Available: https://github.com/SoC-Lab/Firmware

[6] T. Instruments, "CAN Bus transceiver," 2018. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn65hvd230.pdf

[7] STMicroelectronics, "STM32F103RB," 2015. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f103rb.pdf

[8] ——, "STM32 Nucleo-64 F072RB/F103RB," 2017. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf

[9] ——, "STM32F072RB," 2017. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f072rb.pdf

[10] AvNet, "ZedBoard," 2014. [Online]. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[11] Xilinx, "Zynq 7000 Technical Manual," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[12] ——, "7series libraries guide," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf

[13] ——, "Vivado Design Suite User Guide: Partial Reconfiguration (UG909)," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_

manuals/xilinx2018_3/ug909-vivado-partial-reconfiguration.pdf

[14] ——, "Vivado Design Suite Tutorial: Partial Reconfiguration (UG947)," 2018. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2018_3/ug947-vivado-partial-reconfiguration-tutorial.pdf

[15] ——, "ug1118-vivado-creating-packaging-custom-ip.pdf." [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1118-vivado-creating-packaging-custom-ip.pdf