



DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2017

Formal Models of Hardware Peripherals

JONATHAN YAO HÅKANSSON

NIKLAS ROSENCRANTZ

Abstract

We consider the security properties of hardware peripherals. We show how formal models of hardware peripherals improve the state-of-the-art security models. We apply formal models of the universal asynchronous transmitter/receiver (UART) and the serial interface RS-232¹. We conduct model-checking using the model-checking tool NuSMV². We do formal verification of the mechanism *altera_avalon_uart*³ including its security policy. We also document our findings about actual hardware configurations based on our theory. The models we use are statechart diagrams and their corresponding formulas written in NuSMV. One of our conclusions is that connections based on ticket-granting security models are secure if we don't assume that there is a powerful malicious eavesdropper who can add, remove and eavesdrop on the data being sent.

¹ IEEE recommended standard 232 <https://en.wikipedia.org/wiki/RS-232>

² Nusmv.fbk.eu New Symbolic Model Checker

³ Altera embedded peripherals
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=68

Terminology and definitions

CPU - A central processing unit which is the center of the computer and performs calculations

FPGA - Field-programmable gate array, a device suitable for hardware prototyping

HID - Human interface device e.g. keyboard, touchscreen, mouse

MMU - A memory management unit

Nios2 - A softcore CPU to which one can download a custom CPU design

RAM - Random access memory

RS-232 - Recommended standard 232 (IEEE) serial interface for UART'

SDRAM - Synchronous dynamic random access memory

SoC - A system on a chip: an integrated circuit that integrates all components of a computer or other electronic systems

UART - Universal asynchronous receiver/transmitter, a hardware for serial data transmission

uClinux - A Linux distribution suitable for embedded systems and FPGA

1. Introduction	5
1.1. Benefits and goal-settings	5
1.2. Practical examples and problems	5
1.3. Architecture of a UART	6
1.4. Interface and Registers	6
1.5. DB-9 Connector	7
1.6. Asynchronous Communication	8
1.7. Physical layer	9
2. Scope and limitations	9
2.1. Scenario	14
2.2. Assessments and Measures	15
3. Specification and Methodology	15
3.1. Formal verification	15
3.2. Model checking	15
3.3. RS-232 Interface	16
3.4. UART transmitter state transitions	17
3.5. UART receiver logic	22
3.6. False Positives and Negatives	23
3.7. Software Programming Model UART Core	24
3.8. Buffer Overflow Exploit	27
4. The Security Traits of the Network	28
4.1. The attacker	28
4.2. Configuration	30
4.3. Choosing the appropriate tools	31
4.4. Methods for Model Checking	32
4.5. Results and Conclusions	33
5. HDL for UART i/o	33
6. UART in SMV format	34
6.1. SMV roles with configuration program	34
7. Specification in .smv files	35
8. NuSMV output example	35
9. Discussion	36

Formal models of hardware peripherals

4

10. Bibliography

39

1. Introduction

In recent years, model-checking has emerged as a means both to test and verify the correctness of hardware and software systems. “Correctness” in this context means that the model satisfies the requirements. The model-checkers can automatically answer the question whether a model satisfies the requirements. We use NuSMV to conduct formal model-checking of hardware peripherals. The hardware peripherals we model are the altera_avalon_uart with the serial interface RS-232.

1.1. Benefits and goal-settings

Many external hardware peripherals have no security model when connected to a computer and vice versa. An operating system will in many cases automatically trust a connected hardware peripherals such as a keyboard, mouse or even a network interface that is connected to the computer.

We cover model-checking of these important security issues using NuSMV, its background and a typical formal model of the UART including its security model, both written in NuSMV. We won’t cover an entire detailed realization and use a simplified model instead⁴.

In practical life, if a user is not secure they will almost always stay insecure about that particular context and about that setting. There are formal models to test software such as unit tests and integration tests and to guarantee data security. Hardware peripherals and their external connections are usually not part of such test and security models. The scope of our project is to build and verify formal models of a common external hardware peripheral that has been connected to the computer such as a typical UART RS-232 configuration. The configuration could mean many different devices (terminals, modem, serial lines, ...) or even emulated devices such as a system emulating an HID (a serially connected component behaving just like a keyboard or a modem while actually being something else).

1.2. Practical examples and problems

A main practical problem is that the operating system often will trust any random physical device that is physically connected to the computer. The operating system will automatically run a program from a trusted physical device, and that program can install malware. Hence, we are not safe when connecting physical peripherals to a computer.

⁴ Altera_avalon_uart

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=68

An inexperienced computer user could also leave the UART interface open for root access and this is the case we can demo with a configuration between two computers. It is also common that the UART port is left open for debugging purposes which means that a hacker can get a login prompt and maybe even login without a password and get a root shell

People breaking into their own devices could also find a serial console left in for debug purposes by the manufacturer available on an unconnected UART.

An example of actual abuse done by hardware peripherals is the “BadUSB” which was a USB device that unauthorized emulated an authenticated user’s keyboard and could issue unauthorized commands by impersonating an authorized user. We examine the ways of securing a computer system from such abuse and similar cases. USB is not similar to UART in practice, but there are similarities in principle and both are used for data in serial transmission.

There are also similar scopes for other kinds of external hardware such as network controllers (NIC) and hardware peripherals that have built-in processors (GPU, FPGA). We have limited our scope to the UART RS-232. Other external hardware security models will be different in details but there will still be similarities for several properties.

1.3. Architecture of a UART

We aim to answer the following questions:

- How reliable is the UART connector?
- What are the inputs of the UART connector?
- What are the outputs?
- Is there internal state?
- How does the internal state change over time and, in particular, in one step?

We look at the specification which should report the memory mapped registers (which are input/output and can affect the state of the UART). An additional input and output is the wire to which the UART is connected. Our model represents some features of the hardware such as the minimal set of necessary features of the `avalon_altera_uart`⁵.

1.4. Interface and Registers

Our UART core provides an Avalon-MM slave interface to the internal register file. The user interface to the UART core consists of six 16-bit registers: control, status, rxdata, txdata, divisor, and end-of-packet. A master peripheral, such as a microprocessor, accesses the registers to control the core and transfer data over the serial connection.

⁵ https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=68

UART (Universal Asynchronous Receiver/Transmitter) is a hardware peripheral (part of an SoC) that is memory mapped and available for use in the context of a program running on a microcontroller. It requires configuration before use, which is generally achieved by writing values into a memory mapped configuration register. The UART can be used to send and receive arbitrary data asynchronously (no clock needed) over two signal wires, TX and RX, respectively.

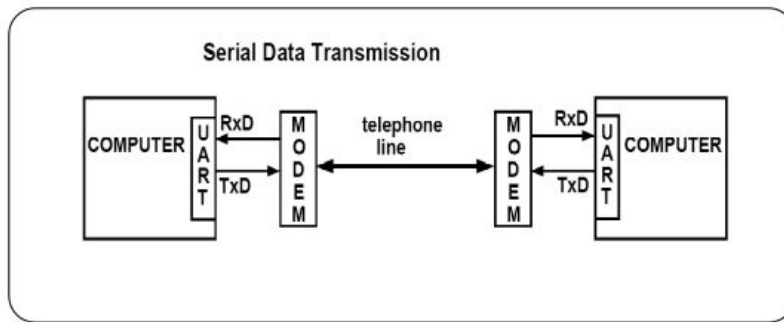


Figure 1. Block diagram of an UART serial transmission.

Simply put, UART is used from the application context within an SoC to send and receive arbitrary data to and from an external device. An RS-232 interface has the following characteristics:

- A 9 pin connector "DB-9"
- Bidirectional full-duplex communication (the PC can send and receive data at the same time).
- Can communicate at a maximum speed of roughly 10 KBytes/s.

The case that we formalize is a connected UART RS-232 device in a security context. The manufacturer is Altera and then name of the UART is `avalon_altera_uart`.

1.5. DB-9 Connector

Many computer users have probably seen a connector like this on the back of a PC.

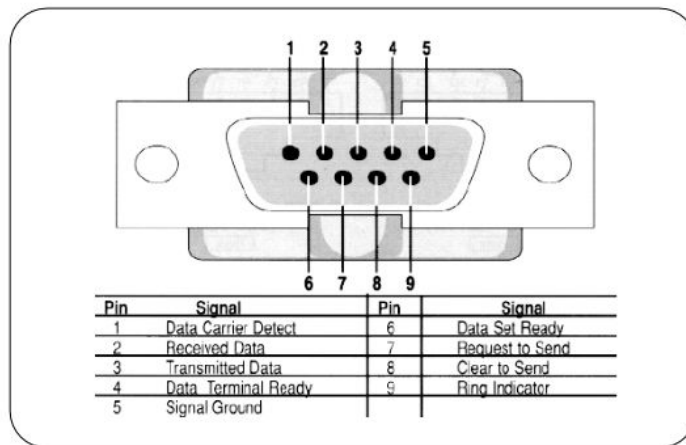


Figure 2. DB-9 connector.

The connector has 9 pins, the 3 relevant pins are:

- pin 2: RxD (receive data).
- pin 3: TxD (transmit data).
- pin 5: GND (ground).

Using these 3 pins, 2 connected peripherals can send and receive data. It also is used for null modem connections.

Data is commonly sent in bytes serialized by the LSB (data bit 0) that is sent first, then bit 1, ... and the MSB (bit 7) last.

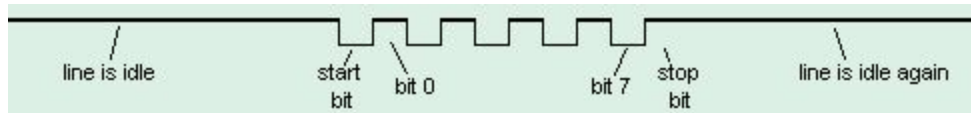
1.6. Asynchronous Communication

The serial interface uses an asynchronous protocol i.e. no clock signal is transmitted along the data. The receiver has to have a way to "time" itself to the incoming data bits.

In the case of RS-232, that's done this way:

1. Both side of the cable agree in advance on the communication parameters (speed, format...). That's done manually before communication starts.
2. The transmitter sends "idle" ("1") when and as long as the line is idle.
3. The transmitter sends "start" ("0") before each byte transmitted, so that the receiver can figure out that a byte is coming.
4. The 8 bits of the byte data are sent.
5. The transmitter sends "stop" ("1") after each byte.

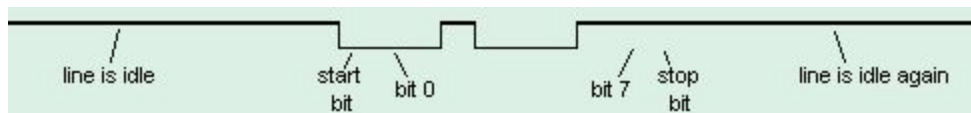
For example a byte 0x55 is transmitted as follows:



Byte 0x55 is 01010101 in binary.

Since it is transmitted LSB (bit-0) first, the line toggles like that: 1-0-1-0-1-0-1-0.

Here's another example:



Here the data is 0xC4.

The bits are harder to see. That illustrates how important it is for the receiver to know at which speed the data is sent.

1.7. Physical layer

The signals on the wires use a positive/negative voltage scheme.

- "1" is transmitted by -10V (or between -5V and -15V).
- "0" is transmitted by +10V (or between 5V and 15V).

So an idle line carries something like -10V.

2. Scope and limitations

We limit the scope of our project to communication via UART RS-232. We build a simplified model that still is realistic without too many details. The RS-232 specification is relatively easy in comparison to modern advanced peripherals. Therefore we choose to build a simplified model that can still be useful for different purposes such as prototyping a security property or serve as a template for adding more details later.

The specific UART we've chosen is the one that is used in Altera FPGA, named `altera_avalon_uart`. In Quartus this UART looks according to the following:

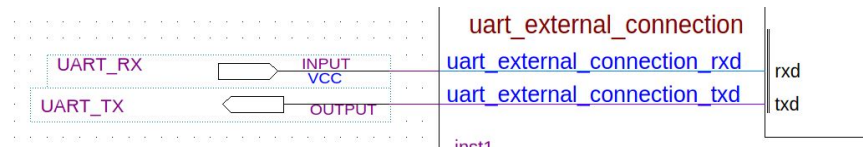


Figure 8. UART hardware in Qsys.

The product specification is according to the following:

Name	altera_avalon_uart
Version	16.1
Author	Altera Corporation
Description	no description
Group	Interface Protocols/Serial
User Guide	https://documentation.altera.com/#/link/sfo1400787952932/iga1401317331859
Release Notes	https://documentation.altera.com/#/link/hco1421698042087/hco1421697689300
Parity	Determines whether the UART core transmits characters with parity checking.
Data bits	Determines the widths of the txdata, rxdata, and endofpacket registers.

Stop bits	Use to terminates a receive transaction at the first stop bit.
Synchronizer stages	Synchronizer of stages
Include CTS/RTS	Include CTS/RTS pins and control register bits
Include end-of-packet	Include end-of-packet register
Baud rate (bps)	standard baud rates for RS-232 connections

The RS-232 settings are provided below for our connection from the Linux host that is used:

- Baud Rate: 115200
- Parity Check Bit: None
- Data Bits: 8
- Stop Bits: 1
- Flow Control (CTS/RTS): Off

The RS-232 standard is used by many specialized and custom-built devices. This list includes some of the more common devices that are connected to the serial port on a PC. Some of these such as modems and serial mice are falling into disuse while others are readily available.

In Quartus our UART looks like the following in QSys.

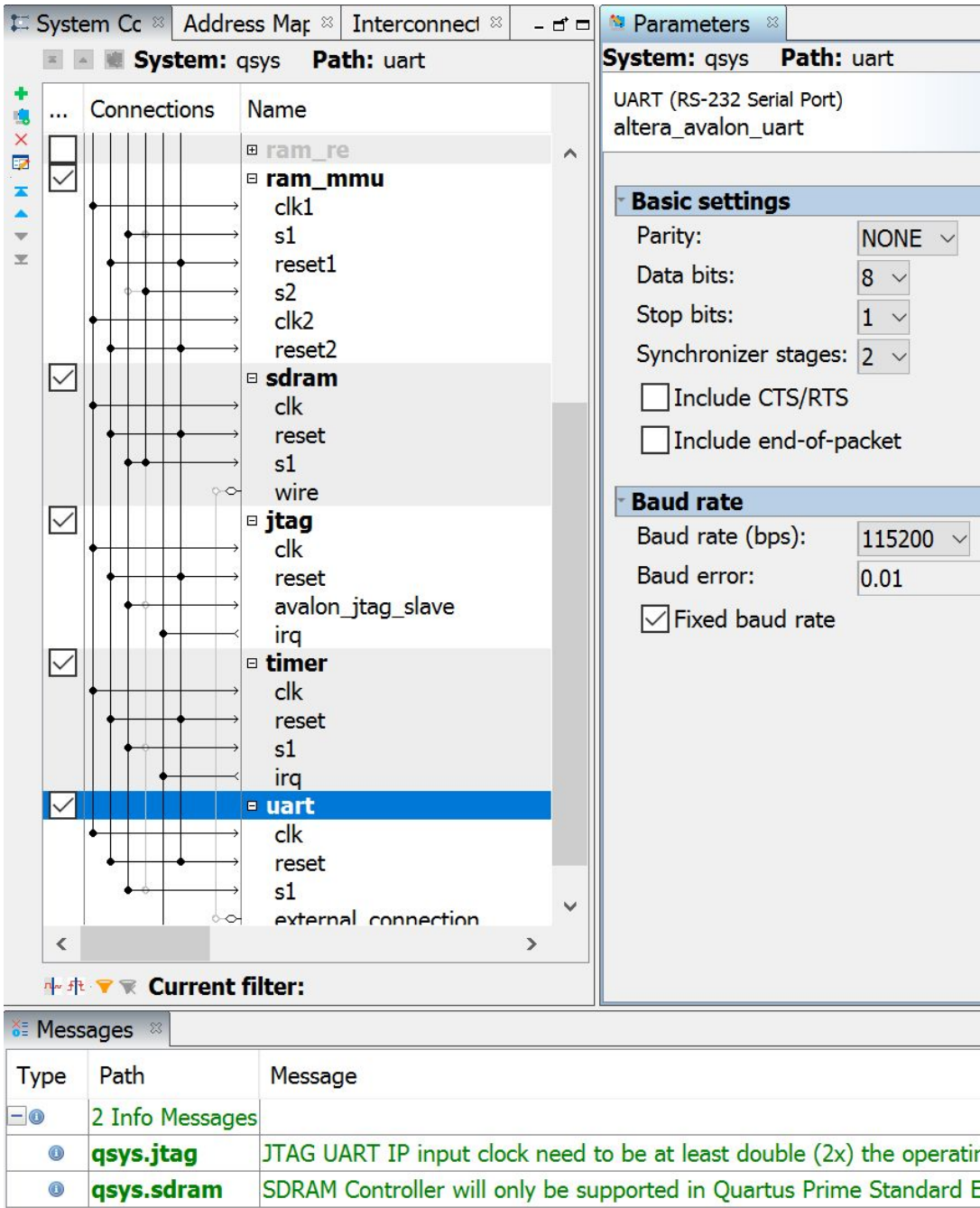


Figure 9. UART connection in Qsys.

Our 32-bit system looks as follows in Quartus. The clock for the SDRAM is to the left and the UART, RAM, MMU and CPU are to the right.

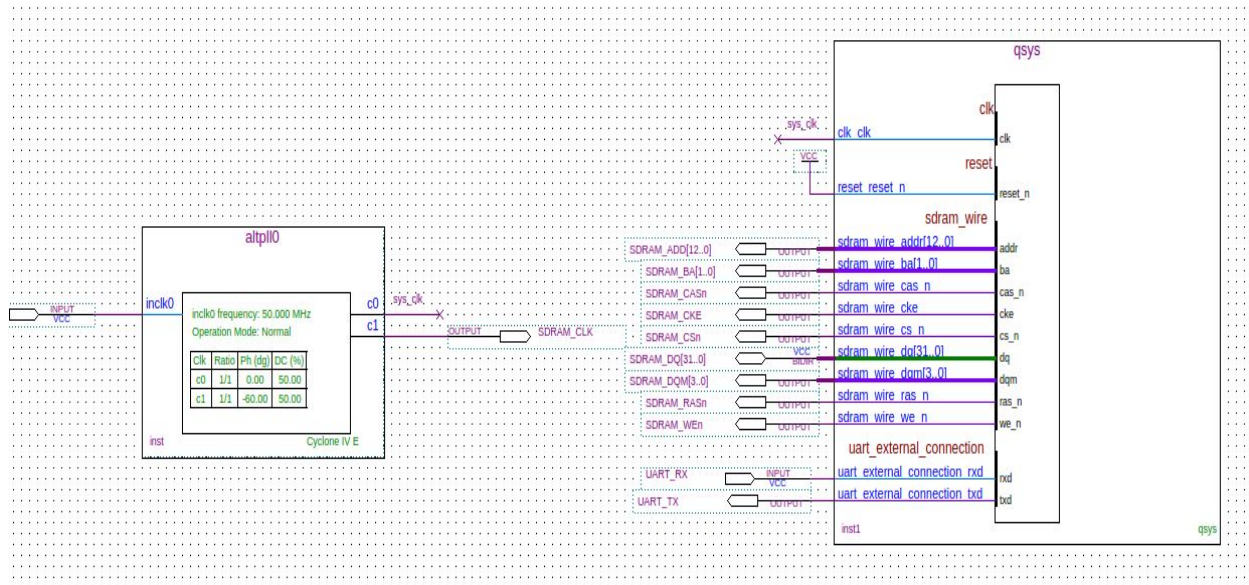


Figure 10. A 32-bit MMU system in Quartus.

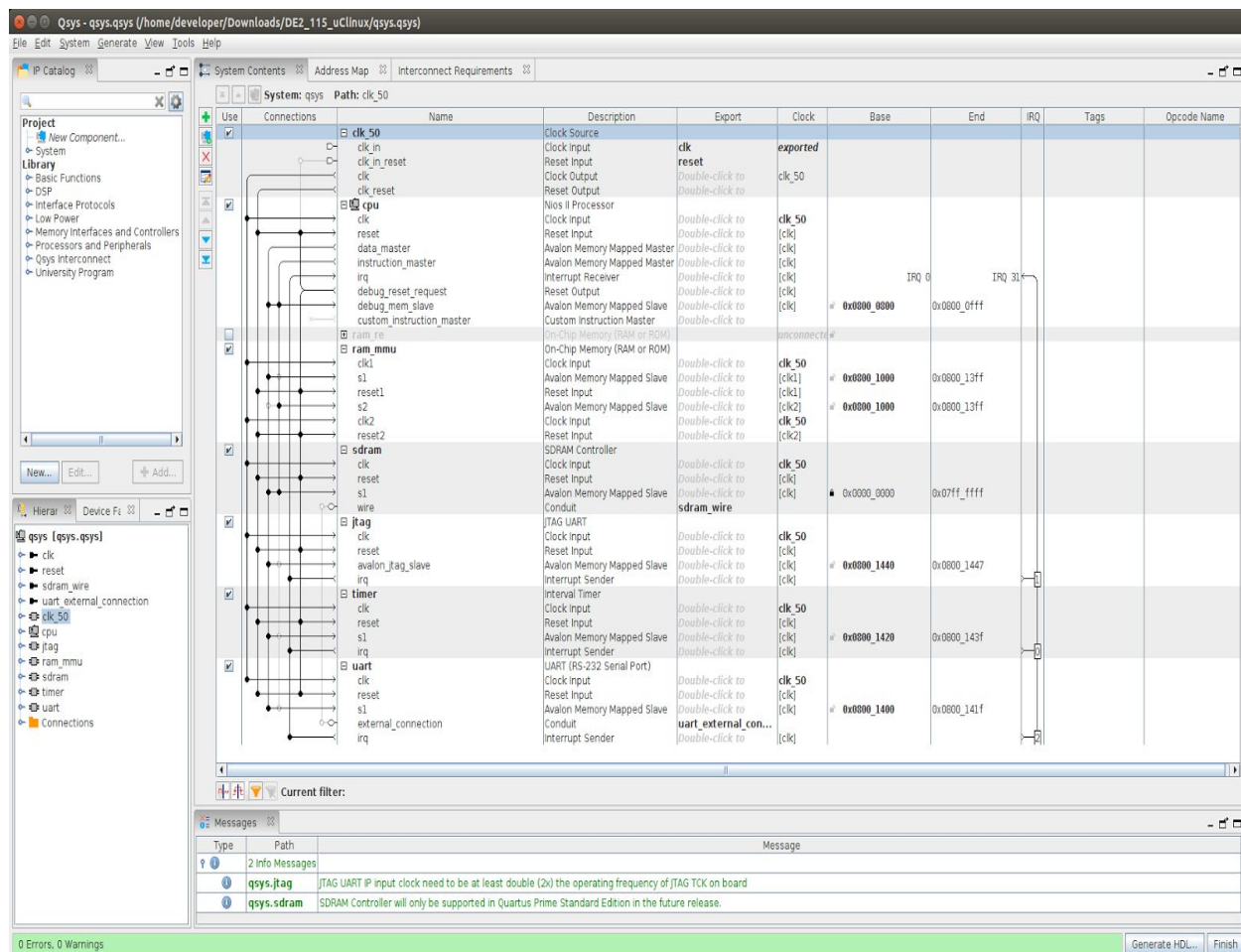


Figure 11. Our MMU system done in Qsys.

When building uClinux we selected the console on Altera UART:

Character devices --->

Serial drivers --->

[] Altera JTAG UART console support

[*] Altera UART console support

2.1. Scenario

Common usage scenarios are that computer components are connected and checked using an abstract security policy implemented with some concrete software or hardware mechanism. A policy is an abstraction of how a security mechanism should be implemented. The mechanism then implements the policy.

2.2. Assessments and Measures

UART enables serial character bitstreams between a computer system or an FPGA and an external peripherals. The UART implements the RS-232 protocol timing, and provides adjustable baud rate, parity, stop, and data bits. The feature set is configurable, allowing designers to implement just the necessary functionality for a given system. The UART provides a memory-mapped interface that allows peripherals (such as a processor) to communicate with the UART by reading and writing control and data registers.

3. Specification and Methodology

Hardware circuits can be described in NuSMV. Hence, we build a behavioral model of a real UART RS-232 peripheral: The *altera_avalon_uart*, and we use model-checking to check the requirements whether a connected peripheral can be trusted.

3.1. Formal verification

We include two models in our scope: The hardware devices and their states (1) and the software (2) (the driver and the os) delegated to be responsible for the security of the configuration. We include software for the reasons of most security vulnerabilities being due to software errors.

3.2. Model checking

Model checking is formally defined as $M \models \Phi$ where M is an automation model and Φ is a logical formula in CTL and LTL.

$M \models \Phi$ means that Φ is true in every structure in which M is true. $M \not\models \Phi$ on the other hand, means Φ can be proved using M as the premises. (In both cases, M is a not necessarily finite set of formulas and Φ is a formula.)

First-order logic simultaneously enjoys the following properties: There is a system of proof for which

- If $M \models \Phi$ then $M \models \Phi$ (soundness)
- If $M \models \Phi$ then $M \models \Phi$ (completeness)
- There is a proof-checking algorithm (effectiveness).
- Fortunately there's a fast-running algorithm.

That last point is in stark contrast to this fact: There is no provability-checking algorithm. One can search for a proof of a first-order formula in such a systematic way that you'll find it if it exists, and you'll search forever if it doesn't. But if we've been searching for millions of years

and it hasn't turned up yet, we still don't know whether the search will go on forever or end next week. These results were proved in the 1930s. The non-existence of an algorithm for deciding whether a formula is provable is called Church's theorem, after Alonzo Church⁶.

If Φ is a true statement about all possible behaviors of M then the model checker confirms it (formal verification). If Φ is a false statement about M the model checker constructs a counterexample to Φ that satisfies $\neg\Phi$.

The model usually includes transitions between states. In model-checking, a transition system can be defined to include an additional labeling function for the states. If the transitions are labeled with elements of Σ , then the transition relation is a subset of $Q \times \Sigma \times Q$, while the transition function maps $Q \times \Sigma$ to $2Q$. If the transition system is deterministic and complete, then the transition function can be taken to map Q (or $Q \times \Sigma$) to Q .

For example, to say that there are transitions labeled σ_1 from q_1 to q_2 and q_3 , we would write

$$\{(q_1, \sigma_1, q_2), (q_1, \sigma_1, q_3)\} \subseteq \rho \text{ or } \{q_2, q_3\} \subseteq \delta(q_1, \sigma_1),$$

where ρ is the transition relation and δ is the transition function. It is also possible to adopt more compact notation and denote by something like $q_1 \rightarrow_{\sigma_1} q_2$ the existence of a transition from q_1 to q_2 labeled σ_1 .

A sequence of transition steps can be executed as an actual use-case. This means that we have a method for formally verifying finite-state systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large statecharts can often be traversed in minutes. This technique has been applied to several large industrial systems.

Since our goal is to guarantee that the UART is a secure communication channel we will identify which parameters and internal states affect which memory addresses are accessed by the UART and model it.

3.3. RS-232 Interface

The UART core implements RS-232 asynchronous transmit and receive logic. The UART core sends and receives serial data via the TXD and RXD ports. The I/O buffers on most Altera FPGA families do not comply with RS-232 voltage levels, and may be damaged if driven directly by signals from an RS-232 connector. To comply with RS-232 voltage signaling specifications, an external level-shifting buffer is required (for example, Maxim MAX3237) between the FPGA I/O pins and the external RS-232 connector. The UART core uses a logic 0

⁶ Alonzo Church's theorem

for mark, and a logic 1 for space. An inverter inside the FPGA can be used to reverse the polarity of any of the RS-232 signals, if necessary.

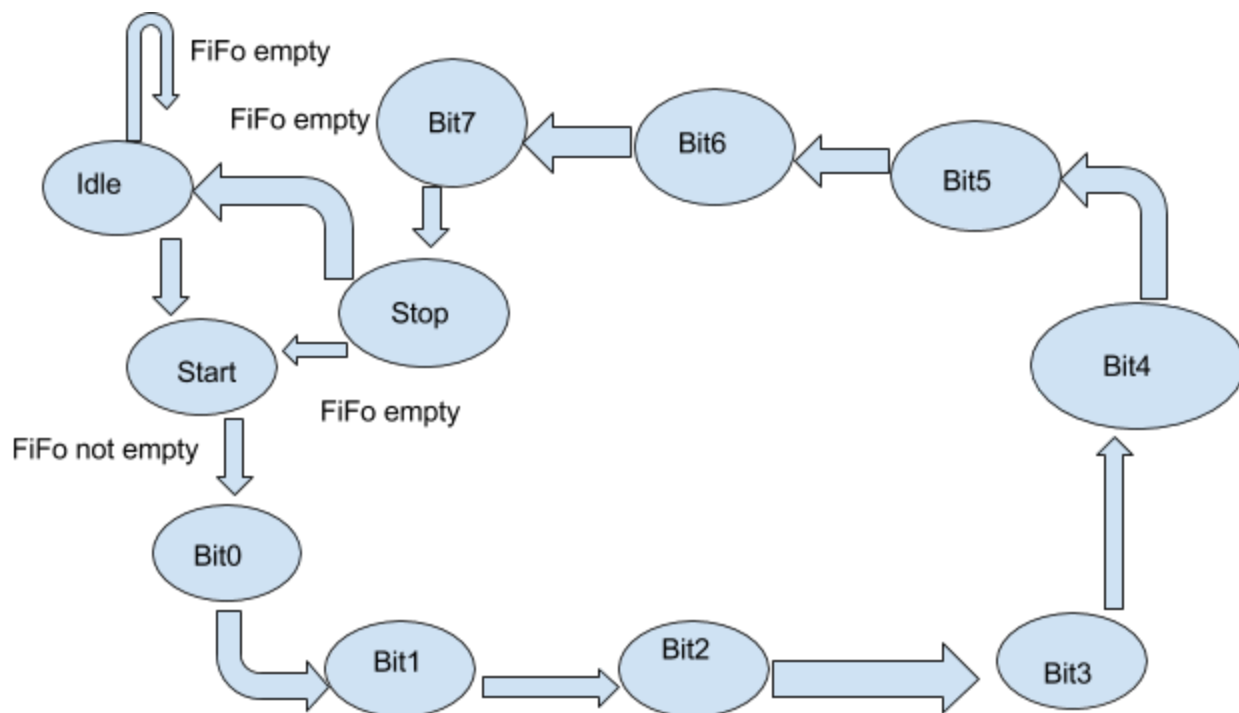


Figure 3. FSM diagram of the UART shift register.

The state machine waits for the CPU or DMA to place an entry in the transmit FIFO. Once there is data present it transmits one Start bit, 8 data bits and finishes with one Stop bit. The bits are sent starting with bit 0. A design requirement is that two bytes can be sent back to back, so the STOP state must go directly to the START state if more data is available.

3.4. UART transmitter state transitions

The UART transmitter consists of a 7-, 8-, or 9-bit txdata holding register and a corresponding 7-, 8-, or 9-bit transmit shift register. Avalon-MM master peripherals write the txdata holding register via the Avalon-MM slave port. The transmit shift register is loaded from the txdata register automatically when a serial transmit shift operation is not currently in progress. The transmit shift register directly feeds the TXD output. Data is shifted out to TXD LSB First. The two registers provide double buffering. A master peripheral can write a new value into the txdata register while the previously written character is being shifted Out. The master peripheral can monitor the transmitter's status by reading the status register's transmitter ready (TRDY), transmitter Shift Register empty (tmt), and transmitter overrun error (TOE) bits. The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial TXD data stream as required by the RS-232 specification.

The signals and states are pictured in the following state diagram.

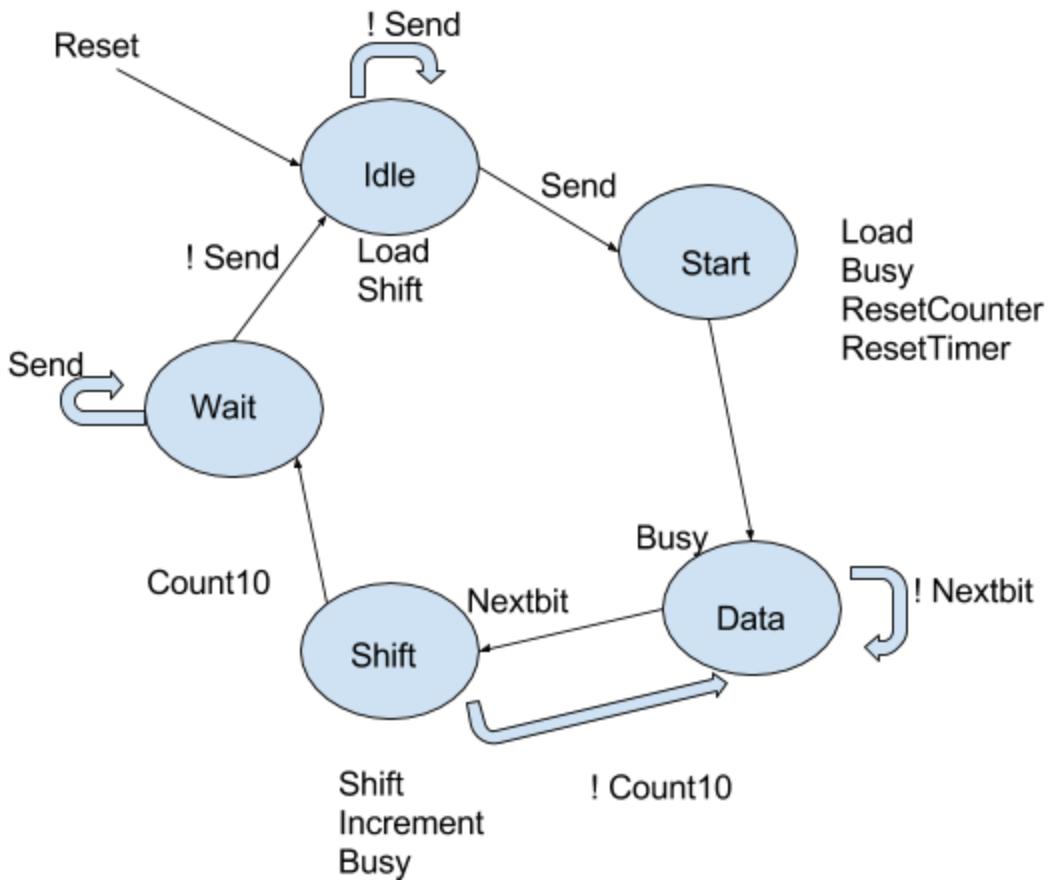


Figure 4. State diagram of UART transmitter.

It is important that the busy signal has no hazards. The corresponding NuSMV code is

```

MODULE main

-- ACTIVITIES

-- 1. Fill in the smv definitions

-- 2. Then run this file in interactive mode to see the states

VAR

-- system outputs

-- the model has states: idle, start, data, shift and wait

    state : {idle, start, data, shift, wait};

    send : boolean;
  
```

```

    load : boolean;

    busy : boolean;

    reset_counter : boolean;

    reset_timer : boolean;

    count10 : 1..10;

    nextbit : boolean;

IVAR

-- system inputs

    input : boolean;

ASSIGN

    init (state) := idle;

    init (send) := TRUE;

    init (nextbit) := TRUE;

    init (count10) := 1;

    next (state) :=

        case

start      state = idle & send : start; -- if state is idle and load=1 then next state is

            state = start & send : data; --if state is start then next state is data

            state = start & nextbit : shift;

            state = data & ! nextbit : data;

            state = shift & count10 < 10: data;

            state = shift & count10 >= 10: wait;

            state = wait : idle;

            TRUE : state; -- if conditions fail, then do not change state

        esac;

    next (load) :=

        case

            (state = start) & send : TRUE;

            TRUE : FALSE; -- otherwise, load is false

        esac;

```

```
next (send) :=
    case
        (state = data) & send : TRUE;
        TRUE : FALSE; -- otherwise, send is false
    esac;

next (count10) :=
    case
        (state = shift) & (count10 < 10): count10 +1;
        TRUE : count10; -- otherwise, go to waiting state
    esac;

next (nextbit) :=
    case
        (state = shift) & (count10 < 10): FALSE;
        TRUE : TRUE; -- otherwise, goto waiting state
    esac;
```

We can step through the model in NuSMV interactive mode as follows

```
$ NuSmV -int
NuSMV > read_model -i uart.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > pick_state -i
```

```
***** AVAILABLE STATES *****
```

```
===== State =====
```

```
0) -----
```

```
state = idle
```

```
send = TRUE
```

```
load = FALSE
```

```
busy = FALSE
```

```
reset_counter = FALSE
```

```
reset_timer = FALSE
```

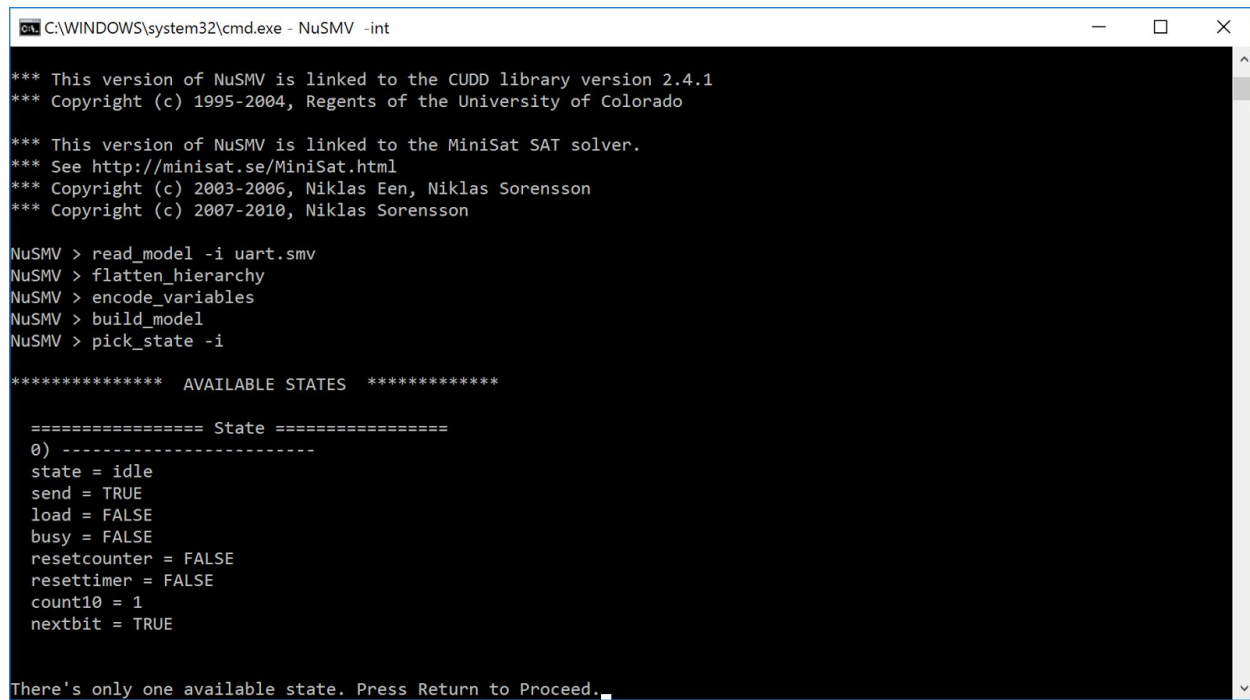
```
count10 = 1
```

```
nextbit = TRUE
```

There's only one available state. Press Return to Proceed.

Chosen state is: 0

NuSMV > simulate -i -k 15



```

C:\WINDOWS\system32\cmd.exe - NuSMV -int

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

NuSMV > read_model -i uart.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > pick_state -i

***** AVAILABLE STATES *****

===== State =====
0) -----
state = idle
send = TRUE
load = FALSE
busy = FALSE
resetcounter = FALSE
resettimer = FALSE
count10 = 1
nextbit = TRUE

There's only one available state. Press Return to Proceed.

```

Figure 5. NuSMV interaction.

3.5. UART receiver logic

The UART serial receiver consists of 2 registers: Bitwise shift register and bitwise holding register. In the case of `altera_avalon_uart`, the memory management master peripheral reads the holding register. The holding register is loaded from the shift register when a new byte is received. These two registers provide double Buffering. The `rxdata` register can hold a previously received character while the subsequent character is being shifted into the receive shift Register. A master peripheral can monitor the receiver's status by reading the status register's read-ready (RRDY), receiver-overflow error (ROE), break detect (BRK), parity error (PE), and framing error (FE) bits. The receiver logic automatically detects the correct number of start, stop, and parity bits in the serial RXD stream as required by the RS-232 Specification. The receiver logic checks for four exceptional conditions, frame error, parity error, receive overrun error, and break, in the received data and sets corresponding status register bits.

We build and check such formal models that can achieve security with a hardware peripheral. We briefly compare model checking with theorem proving. We answer related questions and prioritize the known related problems and relations with software such as operating system and application programs. We describe the delegation of responsibilities in such configurations, what to protect and what the potential hardware and software vulnerabilities are.

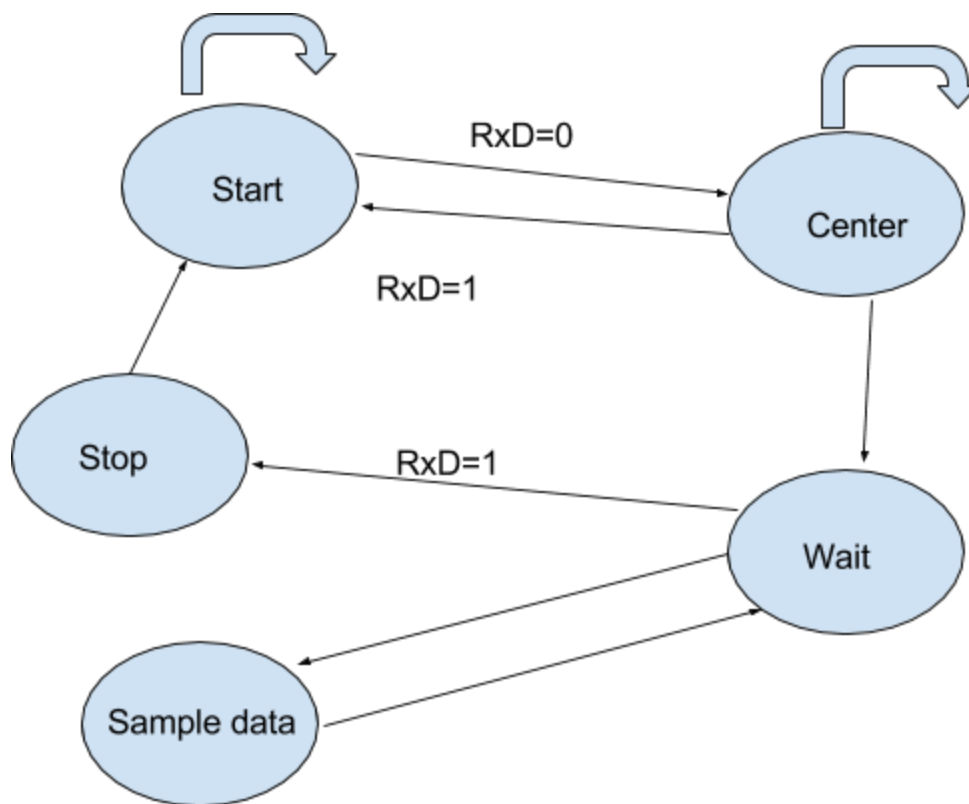


Figure 6. UART receiver state diagram.

3.6. False Positives and Negatives

Bruce Schneier, a famous expert in IT security, has specified a common and general IT security problem as follows:

“Any security expert can invent a cipher that he himself can’t break.”

Schneier’s observation is that everybody can outsmart themselves and therefore you can also be intruded because intruders can get around your security solution in ways you didn’t prepare for. We study the hardware security applications of this problem and the existing and proposed solutions. Recently there has also been reports in the news and media that important and vital Government departments and functions of the state have been intruded in this manner by injecting malware from external hardware peripheral, a hardware peripheral which mistakenly has been trusted.

The common data security patterns often use the placeholder actor names Alice, Bob, Carol, Eve, Mallory, Peggy, Victor, Trent etc for different roles and functions. Typically Alice and Bob are the ones sending data and the other guys are actors with different functions. Will they ever be 100 % safe?

A common problem with theorem proving and model checking today is that theorems and models become increasingly complicated due to more functionalities, more performances and more possible combinations. Therefore it has been proposed that induction proofs can be used instead of checking and proving all possible combinations. In practice an induction proof means verifying a security aspect for a base-case e.g. a 4-bit CPU and then proving it for example for a parallel connection for 2 parallel 4-bits CPU and then the property is proved for 8-bits and likewise for longer words by induction.

The practical problem definition is firstly proving the problem of unauthorized access to the example of a cryptographic key that should be for authorized owner or user only.

We first show that the cryptographic key is accessible and unsafe by policy and mechanism. Then we show that a formal model enforced by a mechanism can make the cryptographic key formally private and secure.

3.7. Software Programming Model UART Core

We can select UART as a console when we build Linux for embedded systems. This flavor of linux is named uClinux.

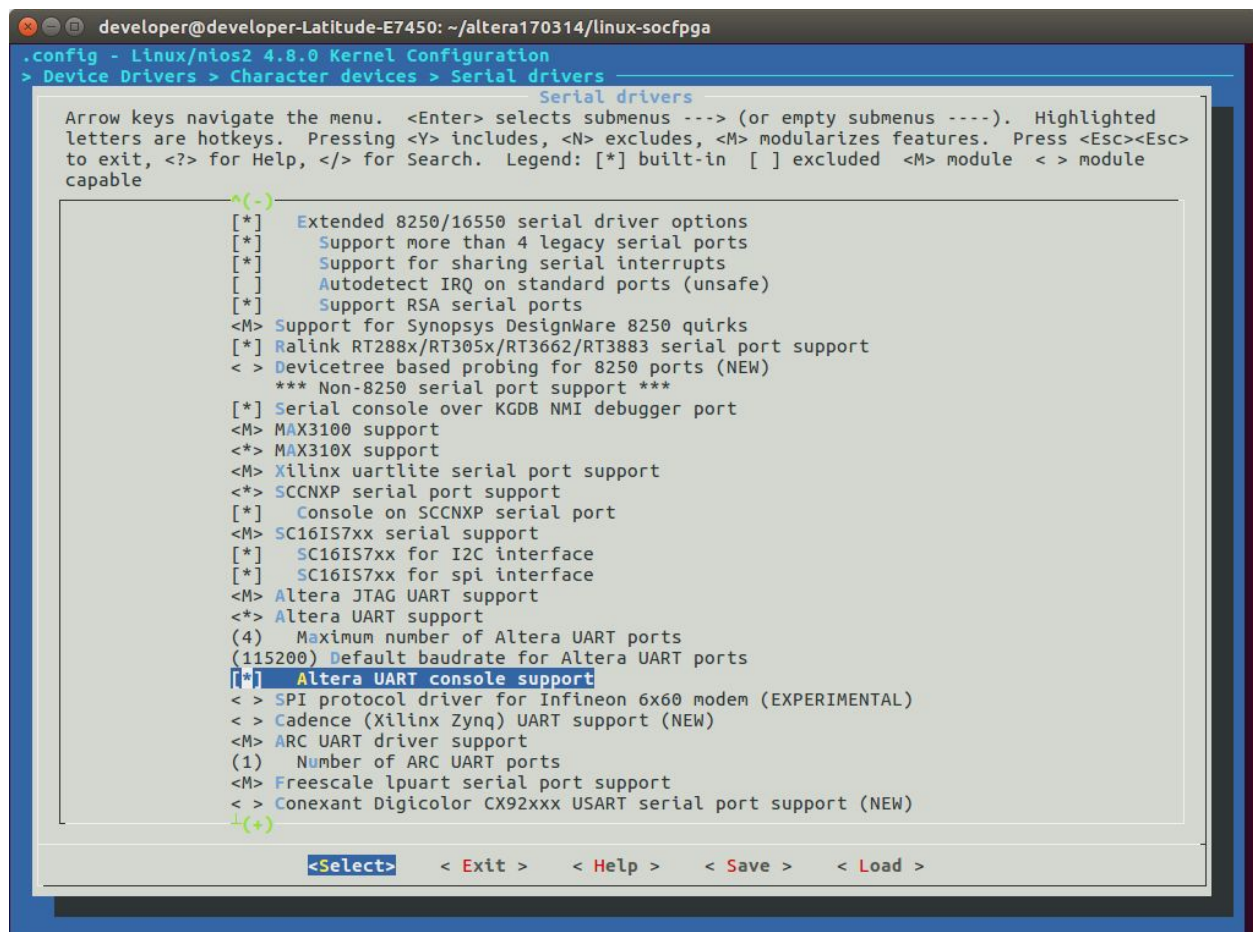


Figure 7. Configure Linux for UART.

The following code demonstrates the simplest possible usage, printing a message to stdout using `printf()`. In this example, the system contains a UART core, and the HAL system library has been configured to use this device for stdout.

Example 3-1: Example: Printing Characters to a UART Core as stdout

```
#include <stdio.h>

int main ()
{
    printf("Hello world.\n");

    return 0;
}
```

```
}
```

The following code demonstrates reading characters from and sending messages to a UART device using the C standard library. In this example, the system contains a UART core named `uart1` that is not necessarily configured as the stdout device. In this case, the program treats the device like any other node in the HAL file system.

Example 3-2: Example: Sending and Receiving Characters

```
/* A simple program that recognizes the characters 't' and 'v' */

#include <stdio.h>

#include <string.h>

int main ()
{
    char* msg = "Detected the character 't'.\n";
    FILE* fp;
    char prompt = 0;

    fp = fopen ("/dev/uart1", "r+"); //Open file for reading and writing
    if (fp)
    {
        while (prompt != 'v')
        { // Loop until we receive a 'v'.
            prompt = getc(fp); // Get a character from the UART.
            if (prompt == 't')
            { // Print a message if character is 't'.
                fwrite (msg, strlen (msg), 1, fp);
            }
        }

        fprintf(fp, "Closing the UART file.\n");
        fclose (fp);
    }

    return 0;
}
```

3.8. Buffer Overflow Exploit

```

#include <string.h>
#include <stdio.h>

void foo (char *bar)
{
    float My_Float = 10.5; // Addr = 0x0023FF4C
    char  c[28];           // Addr = 0x0023FF30

    // Will print 10.500000
    printf("My Float value = %f\n", My_Float);

    /* ~~~~~~
       Memory map:
       @ : c allocated memory
       # : My_Float allocated memory

       *c                *My_Float
       0x0023FF30         0x0023FF4C
       |                 |
       @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#####
       foo("my string is too long !!!!! XXXXX");

       memcpy will put 0x1010C042 (little endian) in My_Float value.
       ~~~~~~*/

    memcpy(c, bar, strlen(bar)); // no bounds checking...

    // Will print 96.031372
    printf("My Float value = %f\n", My_Float);
}

int main (int argc, char **argv)
{
    foo("my string is too long !!!!! \x10\x10\xc0\x42");
    return 0;
}

```

}

4. The Security Traits of the Network

We will handle the authentication property as the security traits of the network.

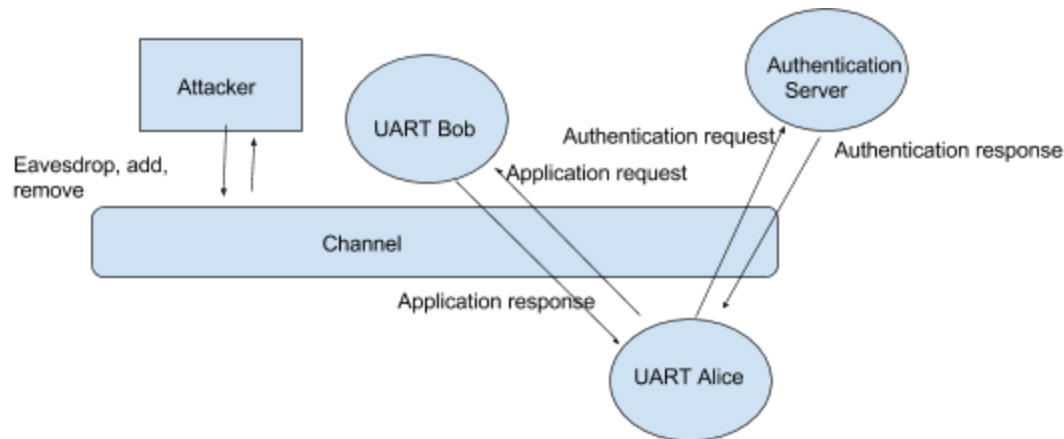


Figure 12. Security model of communication via UART.

We assume that Bob is running embedded uClinux with his FPGA and Alice wants to authenticate with Bob via the UART connection. First Alice must authenticate which is done in connection with the external authentication server that typically grants a time-constrained cryptographic ticket.

4.1. The attacker

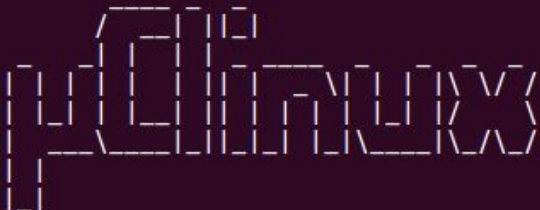
Modeling the attacker requires identification of the aforementioned assumptions that allow an attacker to capture possible behaviors. The attacker is assumed to be powerful in order to verify the security property in the worst case scenario. Therefore, we assume that an attacker has a complete control of the communication channel. In other words, the attacker is able to eavesdrop messages sent over the authentication channel and generate new messages.

The condition is that the authorized parties must know a set of terms (typically, these terms are keys), and integrity, i.e. assurance that an information has not been altered by unauthorized entity.

With the default build options of uClinux for Nios2 the system boots directly to shell (sash) as root with no login prompt. While this is not an issue if you don't need a system console it could become a security issue if you plan to, in example, set up a console in a serial port for

This situation can be avoided with some simple changes to the kernel configuration so that we have some security model by adding regular users whose privileges are restricted instead of root.

```
developer@developer-Latitude-E7450: ~  
Enter the new password (minimum of 5, maximum of 8 characters)  
Please use a combination of upper and lower case letters and numbers.  
Enter new password:  
Re-enter new password:  
passwd[663]: password for `tommy' changed by user `root'  
Password changed.  
\u:\w> login tommy  
Password:  
Welcome to
```



```
For further information check:  
http://www.uclinux.org/
```

```
\u:\w> whoami  
tommy  
\u:\w>
```

Figure 13. uClinux running on FPGA.

We specifically build the formal model of the states of the UART and the states of the system that should be used for formal verification and model checking. Model checking is an examination of all the possible states of a design to determine if any of them violate a specified set of properties, similar to a mathematical proof that covers all possible cases. Model checking is done through mathematical analysis where properties such as assertions are checked against a specification.

It's theoretically possible to use model checking to fully verify the functionality of design, it is typically used for sub-blocks of a design

Equivalence checking is an alternative method of formal verification where one compares two different implementations of a design to see if they are functionally equivalent.

4.2. Configuration

A simplistic outline of our theorem, that we are going to add more details to, can look as follows.

$$\begin{array}{c} \text{Conf} \\ \wedge \\ \models \text{Model} \\ \text{UART} \\ \wedge \\ \models \text{WRITE} \end{array}$$

The above theorem means that if the configuration is done properly and correctly, the UART connection can't allow manipulation of any protected user processes such as web browser or a command-line interpreter. By adding more details and specifics to the above theorem it can be useful without being overly simplistic and without being too complicated.

A simple overview of our model of hardware and its policy enforcement (the driver) can be seen in the following figure.

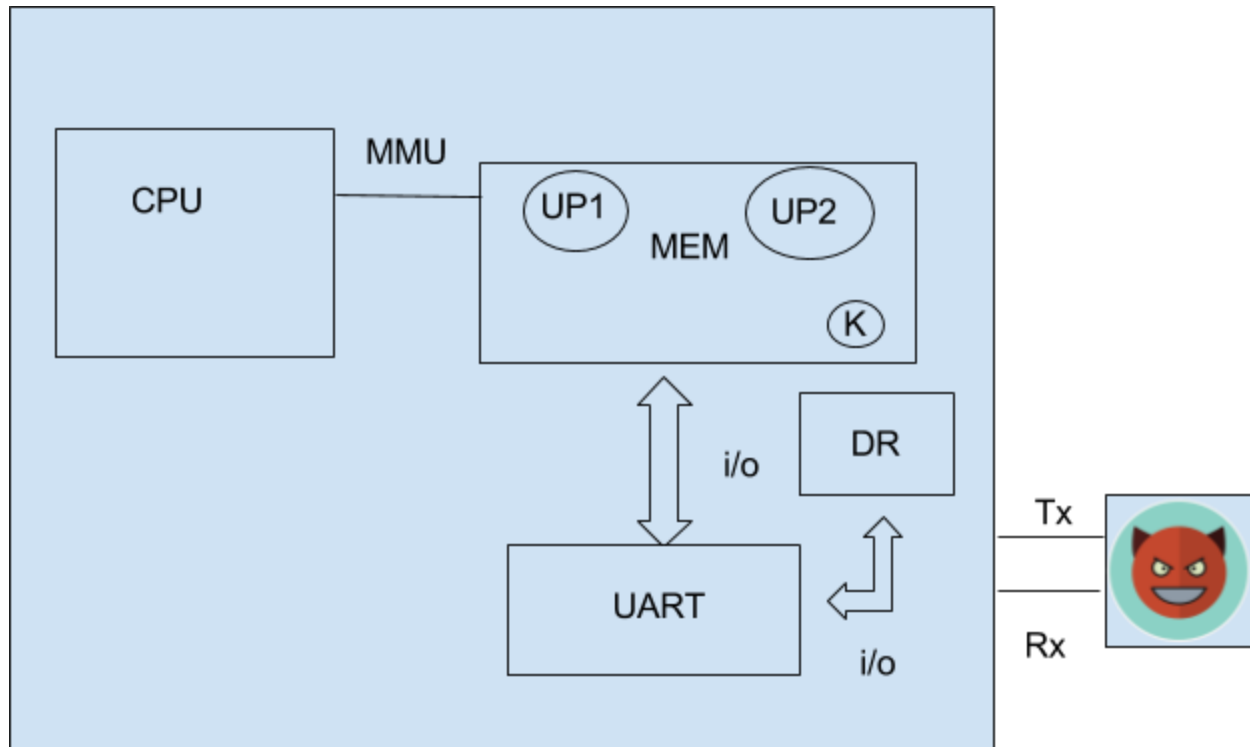


Figure 14. Hardware model overview.

The above is our simplified model of hardware host and the connected external hardware device to illustrate the problem being solved. We added more details to our view in our continued project so that our modelling became more detailed and realistic.

4.3. Choosing the appropriate tools

We chose the appropriate modelling tool NuSMV and also tried Java Path Finder and the Spin model checker to perform model checking of a simplified connected hardware consisting of an UART RS-232 peripheral connected to a computer system and its operating system. We handled the question of responsibility for security: Is the operating system or the hardware responsible?

Our main four problems and questions were (1) that our models were either overly simplistic or much too complicated. We worked on building models that were realistic simplifications without being overly simplistic.

Another problem (2) we to choose the right tools for model checking, which we finally decided to select NuSMV for reasons of wanting to learn a new tool and learning a new language, as well as the tool is well established.

A third problem (3) we had was to correctly identify whether it is hardware or software that should perform the protection mechanism. The responsibility of protecting a system might be the responsibility of the operating system and not the hardware.

Our fourth question (4) was to what extent our project should or could create a new model and a new solution or if we mainly must use existing models and existing solutions.

With the .sof file from Quartus and the uCLinux image we set up our environment:

```
$ nios2-download -g zImage
```

```
$ nios-terminal.exe 1
```

After the above procedures are done we have a development environment where two systems are connected via UART: One Altera DE2-115 which includes the UART connection and via a cable is connected to a peripheral system that tries to gain access via the RS-232 interface that UART enables.

4.4. Methods for Model Checking

One can use Java Path Finder, NuSMV or the Spin model checker for model checking.

The following program, to make us get to know NuSMV, adds a variable configuration and proves that the system is secure for root access provided that the configuration is that the system is closed so that root privileges are not allowed from an open connection that was not checked to begin with.

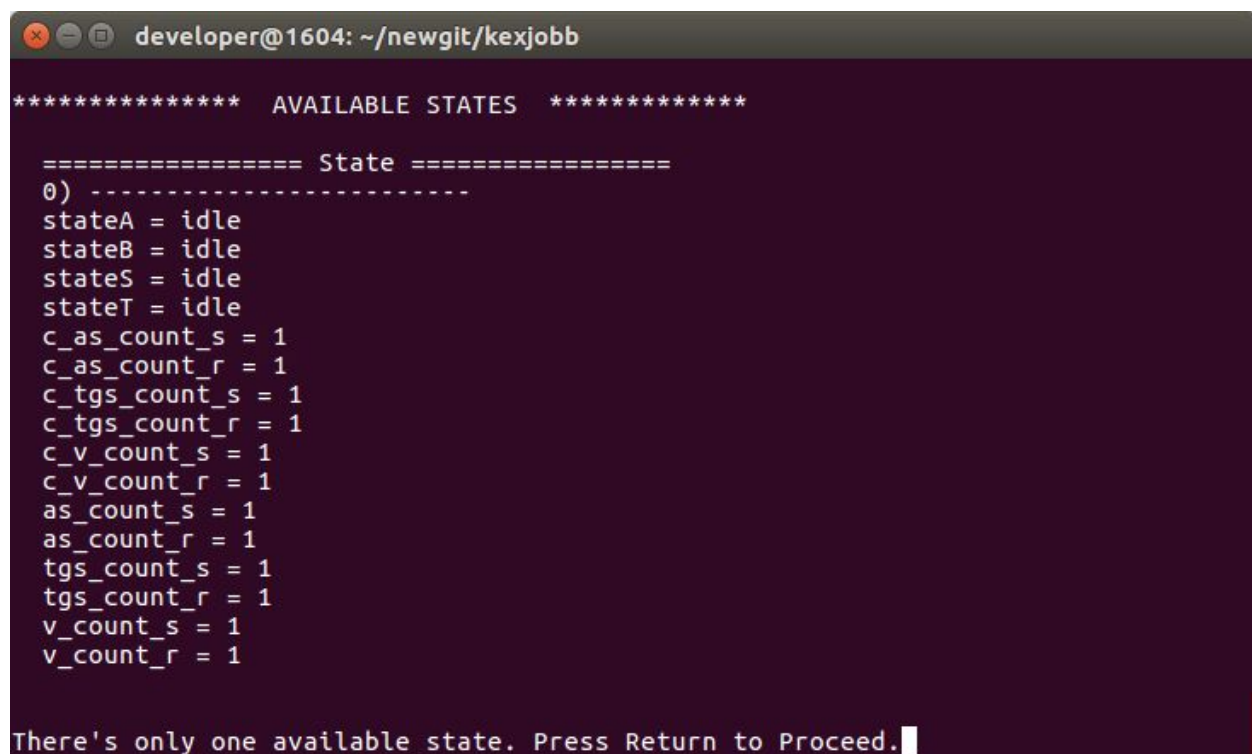
```
MODULE main
VAR
    user : {root, restricted};
    state : {secure, nonsecure};
    config : {open, closed};
ASSIGN
    init (state) := secure;
    init (config) := closed;
    next (state) :=
        case
            config = closed : secure;
            state = secure & (user = root) & (config = open) : nonsecure;
            config = closed : secure;
            TRUE : {secure, nonsecure};
        esac;
SPEC AG ((user = root) -> AF state = nonsecure)
```

The above is supposed to mean that an "open configuration" (to be specified later) is distinct from a "closed configuration" and that the system is not secure if the user is root.

4.5. The proof that communication is secure

The proof is that if we have had an attacker or intrusion, the sum for a ticket would not have been 1 at all places. Some counter would have been 2 or more if there had been an attacker who manipulated the configuration by adding or removing to data being sent.

We see clearly in the model that all counters have the number 1 after a round of data communication.



```

developer@1604: ~/newgit/kexjobb
***** AVAILABLE STATES *****

===== State =====
0) -----
stateA = idle
stateB = idle
states = idle
stateT = idle
c_as_count_s = 1
c_as_count_r = 1
c_tgs_count_s = 1
c_tgs_count_r = 1
c_v_count_s = 1
c_v_count_r = 1
as_count_s = 1
as_count_r = 1
tgs_count_s = 1
tgs_count_r = 1
v_count_s = 1
v_count_r = 1

There's only one available state. Press Return to Proceed.

```

```

-- ACTIVITIES

-- 1. Fill in the smv definitions

-- 2. Then run this file in interactive mode to see the states

--$ NuSMV -int

--NuSMV > read_model -i uart.smv

--NuSMV > flatten_hierarchy

--NuSMV > encode_variables

--NuSMV > build_model

```

```

--NuSMV > pick_state -i

-- NuSMV > simulate -i -k 15

MODULE main

VAR

    stateA : {idle, send1, receive1, send2, receive2, send3, receive3};

    stateB : {idle, send1, receive1};

    stateS : {idle, send1, receive1};

    stateT : {idle, send1, receive1};

    --stateI : {idle, eavesdrop, remove, store, send};

    c_as_count_s : 0 .. 255; --counts the number of messages sent from Alice to Server

    c_as_count_r : 0 .. 255; -- counts the number of messages Alice received from Server

    c_tgs_count_s : 0 .. 255; -- counts the number of messages sent from Alice to Ticket
management

    c_tgs_count_r : 0 .. 255; -- counts the number of messages Alice received from Ticket
management

    c_v_count_s : 0 .. 255; -- counts the number of messages sent from Alice to Bob

    c_v_count_r : 0 .. 255; -- counts the number of messages Alice received from Bob

    as_count_s : 0 .. 255; -- counts the number of messages sent from Server to Alice

    as_count_r : 0 .. 255; -- counts the number of messages received by Server from Alice

    tgs_count_s : 0 .. 255; -- counts the number of messages sent from Ticket management to
Alice

    tgs_count_r : 0 .. 255; -- counts the number of messages received by Ticket management
from Alice

    v_count_s : 0 .. 255; -- counts the number of messages sent from Bob to Alice

    v_count_r : 0 .. 255; -- counts the number of messages received by Bob from Alice

ASSIGN

    init (stateA) := send1; --start by sending to S

    init (stateB) := idle;

    init (stateS) := idle;

    init (stateT) := idle;

    --init (stateI) := idle; --initial states are idle

```

```

init (c_as_count_s) := 0;
init (c_as_count_r) := 0;
init (c_tgs_count_s) := 0;
init (c_tgs_count_r) := 0;
init (c_v_count_s) := 0;
init (c_v_count_r) := 0;
init (as_count_s) := 0;
init (as_count_r) := 0;
init (tgs_count_s) := 0;
init (tgs_count_r) := 0;
init (v_count_s) := 0;
init (v_count_r) := 0;
next (stateA) :=
  case
    stateS = send1 : send2;
    stateB = send1 : receive3;
    --stateA = receive1 : send2;
    stateT = send1 : send3;
    --stateT = receive2 : send3;
    --stateA = idle : send1;
    TRUE : idle;
  esac;
next (stateB) :=
  case
    stateA = send3 : send1;
    --stateB = receive1 : send1;
    TRUE : idle;
  esac;
next (stateS) :=
  case
    stateA = send1 : send1;
    --stateS = receive1 : send1;

```

```

        TRUE : idle;

    esac;

next (stateT) :=

    case

        stateA = send2 : send1;

        --stateT = receive1 : send1;

        TRUE : idle;

    esac;

next (v_count_s) :=

    case

        stateB = send1 & v_count_s < 254 : v_count_s + 1;

        TRUE : 0 + v_count_s;

    esac;

next (v_count_r) :=

    case

        stateB = send1 & v_count_r < 254 : v_count_r + 1;

        TRUE : 0 + v_count_r;

    esac;

next (c_v_count_s) :=

    case

        stateA = send3 & c_v_count_s < 254 : c_v_count_s + 1;

        TRUE : 0 + c_v_count_s;

    esac;

next (c_v_count_r) :=

    case

        stateA = receive3 & c_v_count_r < 254 : c_v_count_r + 1;

        TRUE : 0 + c_v_count_r;

    esac;

next (c_as_count_s) :=

    case

        stateA = send1 & c_as_count_s < 254 : c_as_count_s + 1;

        TRUE : 0 + c_as_count_s;

```

```

    esac;

next (tgs_count_r) :=

    case

        stateT = send1 & tgs_count_r < 254 : tgs_count_r + 1;

        TRUE : 0 + tgs_count_r;

    esac;

next (tgs_count_s) :=

    case

        stateT = send1 & tgs_count_s < 254 : tgs_count_s + 1;

        TRUE : 0 + tgs_count_s;

    esac;

next (c_tgs_count_s) :=

    case

        stateA = send2 & c_tgs_count_s < 254 : c_tgs_count_s + 1;

        TRUE : 0 + c_tgs_count_s;

    esac;

next (c_tgs_count_r) :=

    case

        stateA = send2 & c_tgs_count_r < 254 : c_tgs_count_r + 1;

        TRUE : 0 + c_tgs_count_r;

    esac;

next (as_count_s) :=

    case

        statesS = send1 & as_count_s < 254 : as_count_s + 1;

        TRUE : 0 + as_count_s;

    esac;

next (c_as_count_r) :=

    case

        stateA = send2 & c_as_count_r < 254 : c_as_count_r + 1;

        TRUE : 0 + c_as_count_r;

    esac;

next (as_count_r) :=

```

```

    case
        states = send1 & as_count_r < 254 : as_count_r + 1;
        TRUE : 0 + as_count_r;
    esac;

SPEC AG (stateB = send1 -> c_as_count_s = as_count_r)
SPEC AG (stateA = receive3 -> as_count_s = c_as_count_r)

```

4.6. Results and Conclusions

Any physical computer system can be accessed physically while being somewhat unprotected from technical attacks (for example physically pulling out the RAM module and by such means extracting the data from the physical RAM memory) and therefore we conclude that a computer user or the data will not be 100 % safe or completely protected in principle. The safest usage of the altera_avalon_uart today is to not leave to connector open. If the connector is open, like it was with the uClinux installation, root access is easy for anybody with physical access.

If someone has physical access it is game over and they can also pull out the dram modules (possibly after spraying them with freezer spray) and just read out the cryptographic keys, use JTAG, and many promote possible attacks.

5. HDL for UART i/o

We used the following code to implement the altera_avalon_uart hardware in the Altera DE2-115 FPGA. The FPGA includes the Altera Avalon UART to which we can connect an external peripheral with an RS-232 interface.

```

component qsys is
    port (
        ctlk_clk          : in    std_logic          := 'X';
        -- clk

        reset_reset_n     : in    std_logic          := 'X';
        -- reset_n

        sdram_wire_addr    : out    std_logic_vector(12 downto 0);
        -- addr

        sdram_wire_ba      : out    std_logic_vector(1 downto 0);
        -- ba

        sdram_wire_cas_n   : out    std_logic;
        -- cas_n
    )
end component

```

```

        sdram_wire_cke                : out    std_logic;
-- cke

        sdram_wire_cs_n               : out    std_logic;
-- cs_n

        sdram_wire_dq                 : inout  std_logic_vector(31 downto 0) := (others
=> 'X'); -- dq

        sdram_wire_dqm                : out    std_logic_vector(3 downto 0);
-- dqm

        sdram_wire_ras_n              : out    std_logic;
-- ras_n

        sdram_wire_we_n               : out    std_logic;
-- we_n

        uart_external_connection_rxd : in     std_logic                := 'X';
-- rxd

        uart_external_connection_txd : out    std_logic
-- txd

    );

end component qsys;

```

6. UART in SMV format

The following rather trivial NuSMV code checks if user is root or a restricted user and proves that the system is nonsecure if a user is root.

```

MODULE main

VAR

    user : {root, restricted};

    state : {secure, nonsecure};

ASSIGN

    init(state) := secure;

    next(state) := case

        state = secure & (user = root) : nonsecure;

        TRUE : {secure, nonsecure};

    esac;

```


SPEC

```
AG((user = root) -> AF state = nonsecure)
```

6.1. SMV roles with configuration program

The following program adds a variable configuration and proves that the system is secure for root access provided that the configuration is that the system is closed so that root privileges are not allowed from an open connection that was not checked to begin with.

7. Specification in .smv files

One needs to enter the model of the UART in the format of the model-checker. It now remains to describe the transitions. This is done by the assign block. First we need to assign that idle is the initial state.

8. NuSMV output example

```
C:\Users\dataniklas\git\kexjobb170320>NuSMV -int
```

```
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)

*** Enabled add-ons are: compass

*** For more information on NuSMV see <http://nusmv.fbk.eu>

*** or email to <nusmv-users@list.fbk.eu>.

*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1

*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.

*** See http://minisat.se/MiniSat.html

*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson

*** Copyright (c) 2007-2010, Niklas Sorensson
```

```
NuSMV > read_model -i uart.smv
```

```
NuSMV > flatten_hierarchy
```

```
NuSMV > encode_variables
```

```
NuSMV > build_model
```

```
NuSMV > pick_state -i
```

```
***** AVAILABLE STATES *****
```

```
===== State =====
```

```
0) -----
```

```
state = idle
```

```
send = TRUE
```

```
load = FALSE
```

```
busy = FALSE
```

```
reset_counter = FALSE
```

```
reset_timer = FALSE
```

```
count10 = 1
```

```
nextbit = TRUE
```

There's only one available state. Press Return to Proceed.

Chosen state is: 0

```
NuSMV > simulate -i -k 15
```

9. Discussion

We arranged the following subtasks to complete our main tasks.

1. **We identified the problem to solve:** One problem is that device specifications of 600+ pages have no obvious formal proof and that hardware was developed less by security aspects and more by functional properties and performance.

A second problem is that the model we build might either become overly simplistic or too complicated to be feasible.

2. **We built and formalized our model:** Our goal has been to build and formally verify a model of secure hardware peripherals. Our networks, operating systems and applications must interact with secure hardware peripherals such as input/output devices (UART, USB, network controllers), interrupt controllers and coprocessors (GPU, FPGA). The results are from both theoretical models as well as verifying with real hardware peripherals e.g. Altera DE2-115 FPGA and its UART. The theorems and models we use and build are simplified to enable us conclude intermediate results at the intermediate level.
3. **We conducted model-checking:** Using the appropriate selected tools we have written down the details of our theory and our model. We used automatic theorem proving with NuSMV to prove our theory.

A known problem with model checking is the state explosion problem, which is the problem that the number of states in a system grows exponentially by the number of different parameters. It has been told that induction proofs and inductive and recursive techniques could be able to get around the state explosion problem by proving or checking only a small number of cases and then the proof or model is generalized for all possible cases similar to an inductive or recursive proof technique.

4. **We document, discuss and report relevant conclusion(s) and communicate the result(s):** We document and elaborate about our findings. We also present to result for an intended audience of intermediate level, less advanced than current research and more advanced than trivial.

We specified the problem and limited the scope to a small and simplified model of an UART RS-232 consisting of the parts.

UART doesn't have a security model in its current availability. It seems up to the driver and os to delegate and react responsible.

RS232 doesn't specify higher-level about the wire format. It can be a teletype, AT commands for modems, IP over SLIP/PPP, MODBUS, or something stranger like X.25

We evaluated three frameworks to conduct our model-checking of UART and RS-232.

1. Java Path Finder, JPF, is used to verify executable Java ByteCode programs. JPF was created by NASA AMES Research Center. Main focus of JPF uses and executes Java ByteCode and can store states, match restore program states. Main usage for JPF is Model Checking of concurrent programs. You can use JPF as model check of distributed application, model checking of user interfaces, low level program inspection, program instrumentation.
2. NuSMV is well-suited for modelling hardware circuits. It is a language for verification of finite state systems (FSM). This language conducts checking of finite state machines and checks if specifications are correct against CTL called temporal logic. NuSMV is a BDD-based (Binary Decision Diagram) model checker that allows to check finite state systems against specifications in the temporal logic CTL. The software is freely available at <http://nusmv.irst.itc.it/> where you will also be able to find a tutorial and manual. This program makes it able to have finite systems from completely synchronous to completely asynchronous. Data-types are thought to be used as finite state machines and it have only datatypes as boolean, scalar, bit vectors, fixed arrays. NuSMV has following features Interaction, Analysis of invariants, Partitioning methods, LTL Model Checking, PSL Model Checking, SAT-Based Bounded Model Checking.
3. Spin-Model Checking is well-suited for modeling of concurrent systems. It is a verification system that can be used as verification tool for asynchronous process systems. The Main focus of Spin is process interactions and provide abstract from internal sequence computations. Some formal methods that Spin have are:
 - An intuitive program-like notation for design choices.
 - A concise notation for general correctness requirements.
 - A methodology for establishing logical consistency.
 Spin accepts a verification language PROMELA specified in syntax of Linear Temporal Logic.

We have evaluated these languages for the scope of our project. The most appropriate program to prove this problem is NuSMV because of finite state machines and after that the most likely think is JPF because of our wide java knowledge.

In our methodology there are various types of formal verification that we can do. One is theorem proving, which is a proof of a relationship between a specification and an implementation as a theorem in a logic, proved within the framework of a proof calculus. It is used for verifying arithmetic circuits.

A second way of formal verification is model checking which is checking the specification in the form of a logic formula, the truth of which is determined with respect to a semantic model provided by an implementation. Model checking is starting to be used to check small modules in industry.

Equivalence checking is a third way that checks the equivalence of a specification and an implementation. Equivalence checking is the most common industry use of formal verification.

We have chosen to not perform equivalence checking in this project and instead working primarily with building a sufficient model for our needs and then perform model checking using the appropriate tools.

10. Bibliography

- [1] UART RS-232 <https://en.wikipedia.org/wiki/RS-232>
- [2] RS-232 specification http://www-ug.eecg.toronto.edu/msl/nios_devices/dev_rs232uart.html
- [3] <http://nusmv.fbk.eu/NuSMV/>
- [4] <http://spinroot.com/spin/Doc/ieee97.pdf>
- [5] Buffer overflow
<http://www.csc.kth.se/utbildning/kth/kurser/DD2395/dasak06/dokument/F4/F4.pdf>
- [6] Computer Security Dieter Gollmann
- [7] [Ross Anderson, Security Engineering](#)
- [8]
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=68
- [9] http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C11_SerialInterface.htm
- [10] http://chibios.sourceforge.net/docs/hal_stm32f4xx_rm/group__u_a_r_t.html
- [11] Model checking overview <http://www.cs.cmu.edu/~modelcheck/tour.htm>

[12] DESIGN AND IMPLEMENTATION OF CUSTOMIZABLE

<http://www.ijraet.com/pdf17/31.pdf>

[13] http://www.ijcst.org/Volume7/Issue4/p2_7_4.pdf

[14] Security <http://ijireeice.com/upload/2016/may-16/IJIREEICE%2071.pdf>

[15] The serial driver layer <http://www.linuxjournal.com/article/6331>

[16] tty layer <http://marcocorvi.altervista.org/games/lkpe/tty/tty.htm>

[17] altera_uart.c http://lxr.free-electrons.com/source/drivers/tty/serial/altera_uart.c

[18] http://lxr.free-electrons.com/source/drivers/tty/serial/serial_core.c

[19] NuSMV <http://nuseen.sourceforge.net/>