Formal models of hardware peripherals

Jonathan Yao Håkansson, Niklas Rosencrantz

KTH University of Technology

`{jyh,nik}@kth.se`

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 UART and RS-232 serial interfaces and conduct model-checking using NuSMV. We do formal verification of the mechanism *altera_avalon_uart* including its security policy.

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 UART with the serial interface RS-232.

1.1 Coverage

We cover model-checking with NuSMV, its background and a typical formal model of UART including its security model, both written in NuSMV. We won’t cover an entire detailed realization and use a simplified model instead[1].

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

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

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 Formal verification

We include 2 models in our scope: The hardware devices and their states (1) and the software (2) (typically 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.

1.3 Model-checking

Model checking is formally defined as $M \vDash \varphi$ where M is an automation model and φ is a logical formula. 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 that Φ satisfies $!\varphi$. **<TODO: kolla att detta är sant>**

A model checker is a tool which takes as input

- An automaton model M
- A logical formula φ

We can use a model checker to generate counterexamples to formulas (i.e. use cases) with specific structural properties. If φ is a true statement about all possible behaviors of M then the model checker confirms it (proof) If φ is a false statement about M the model checker constructs a counterexample (a sequence). Such a counterexample to Φ satisfies !Φ.

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 2_Q. 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

$$\{(q1,\sigma1,q2),(q1,\sigma1,q3)\} \subseteq \rho \text{ or } \{q2,q3\} \subseteq \delta(q1,\sigma1),$$

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. The technique has been applied to several complex industrial systems such as the Futurebus+ and the PCI local bus protocols.

Since our goal is to guarantee that the UART does not leak a file that is supposed to be protected (such as a cryptographic key) we will identify which parameters and internal states affect which memory addresses are accessed by the UART and model it.

1.4 Architecture of a UART

Now we answer the following questions:

- How reliable are connected hardware peripherals?
- What are the inputs of the UART component?
- 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, it 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 or the hardware such as the minimal set of necessary features of the avalon_altera_uart[2].

1.4.1 Interface and Registers

Our UART core provides an Avalon-MM slave interface to the internal register

[2] RT*M

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.

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.

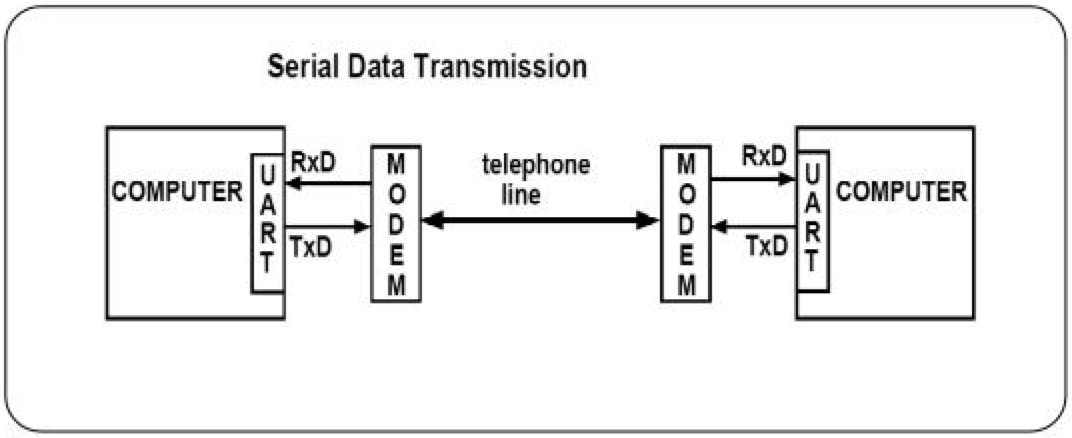


Figure x.x.x Block diagram of a serial transmission

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

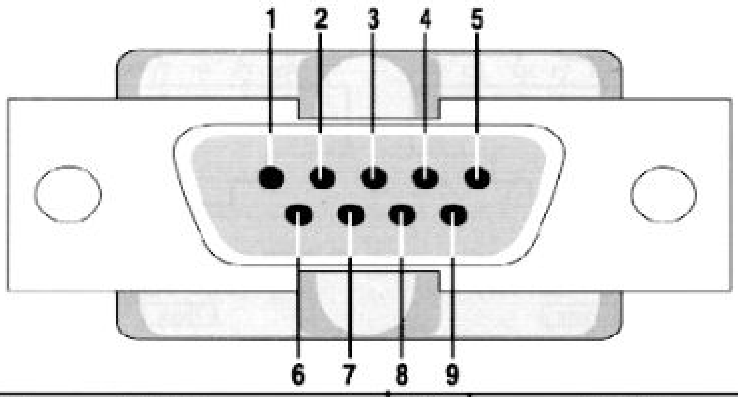
- A 9 pin connector "DB-9"

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

You have probably seen a connector like this on the back of a PC.



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

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.

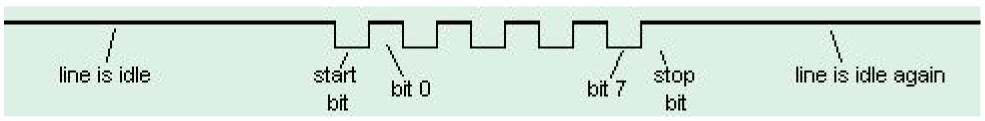
1.6 Asynchronous communication

The 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.

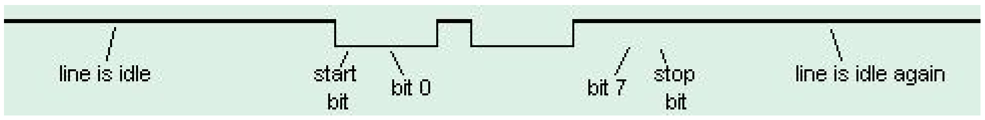
For example a byte 0x55 is transmitted as follows:



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 Benefits and goals

We will answer the questions:

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

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.

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.

2.3 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 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.

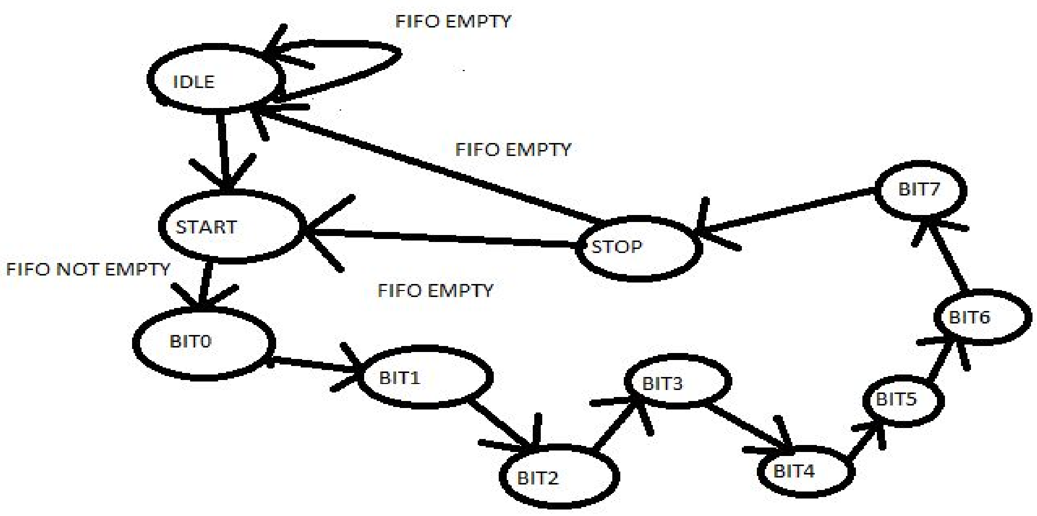


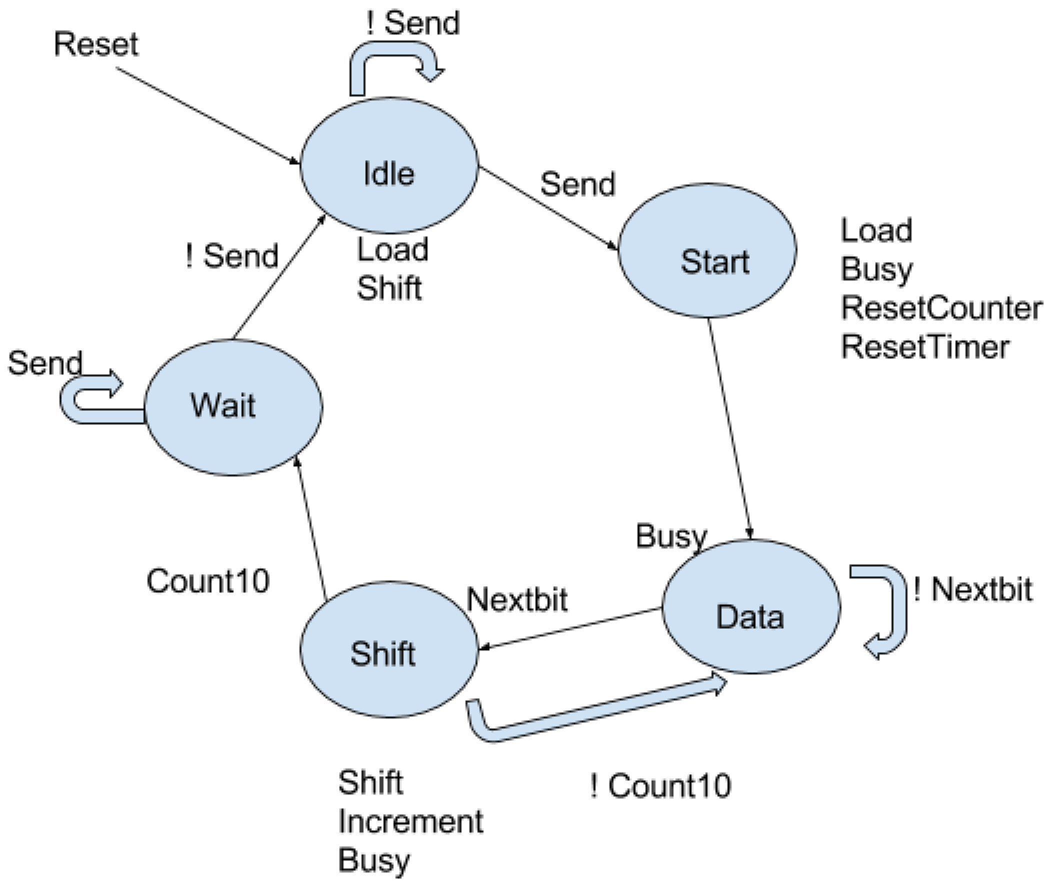
Figure: FSM diagram of UART shift register

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.

2.4 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.

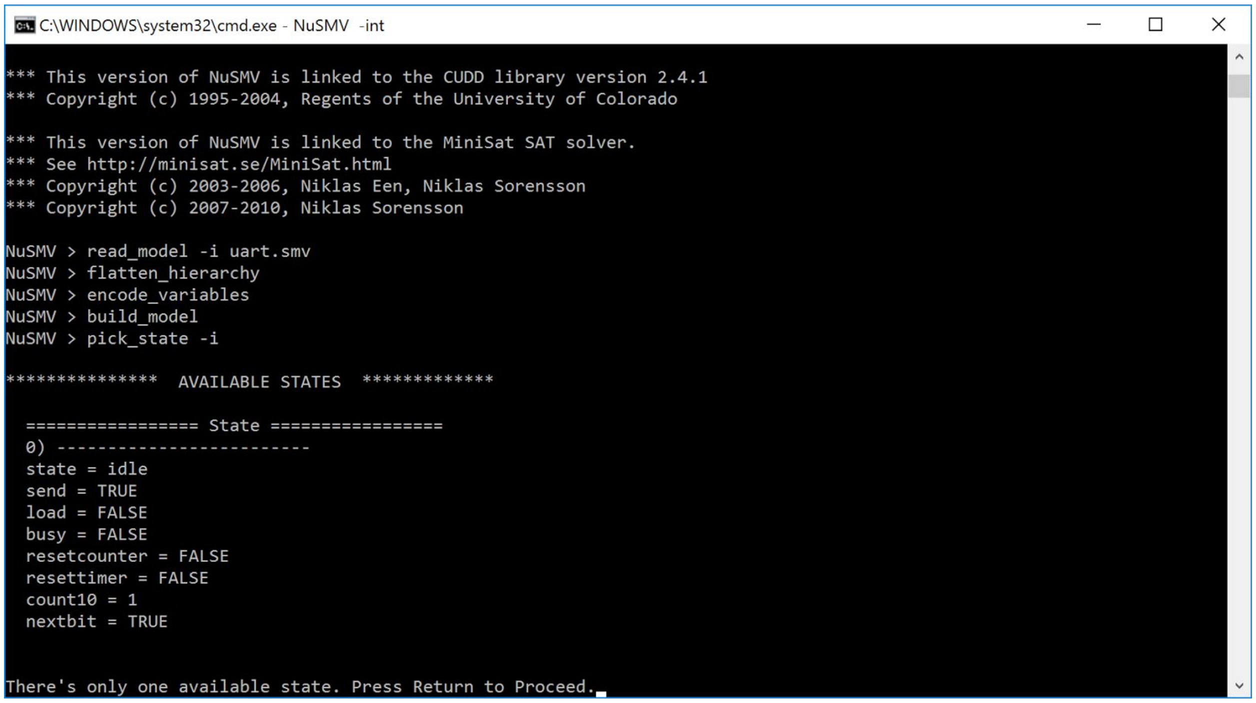


It is important that the busy signal has no hazards. The corresponding NuSMV code is

```
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;

    resetcounter : boolean;

    resettimer : 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

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

                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, goto 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

  resetcounter = FALSE

  resettimer = 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.
```

2.5 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-overrun 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.

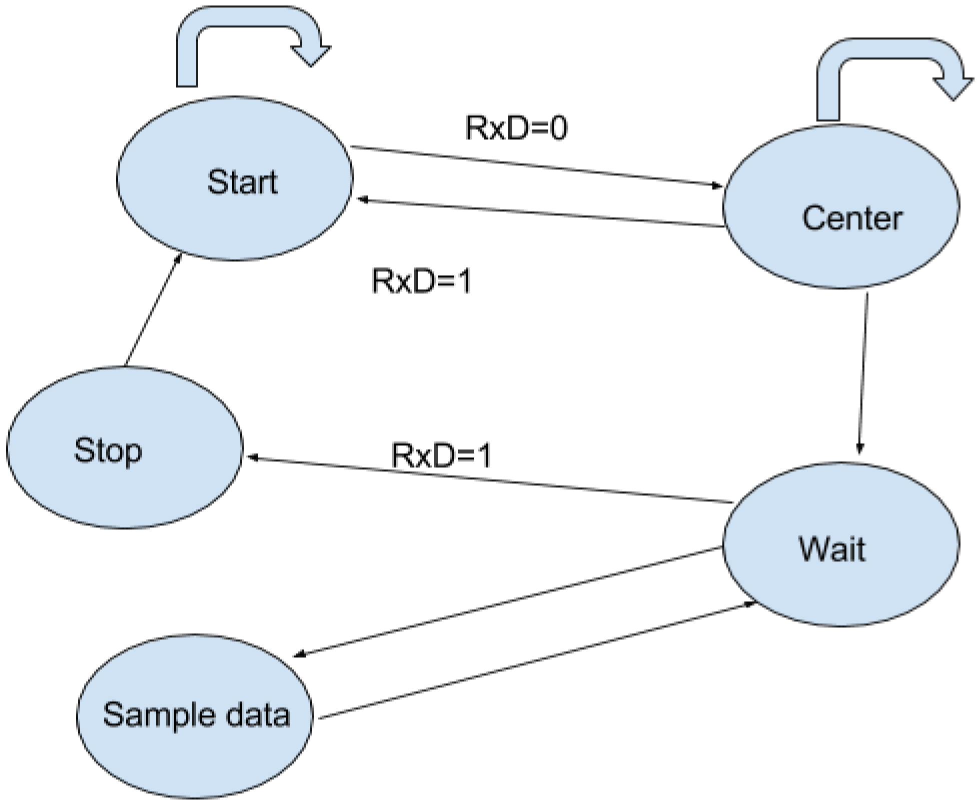


Illustration UART receiver state diagram

2.6 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.

Some not experienced computer user could leave the UART interface open for root access and this is the case we can demo with a configuration between two computers.

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.

2.7 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?

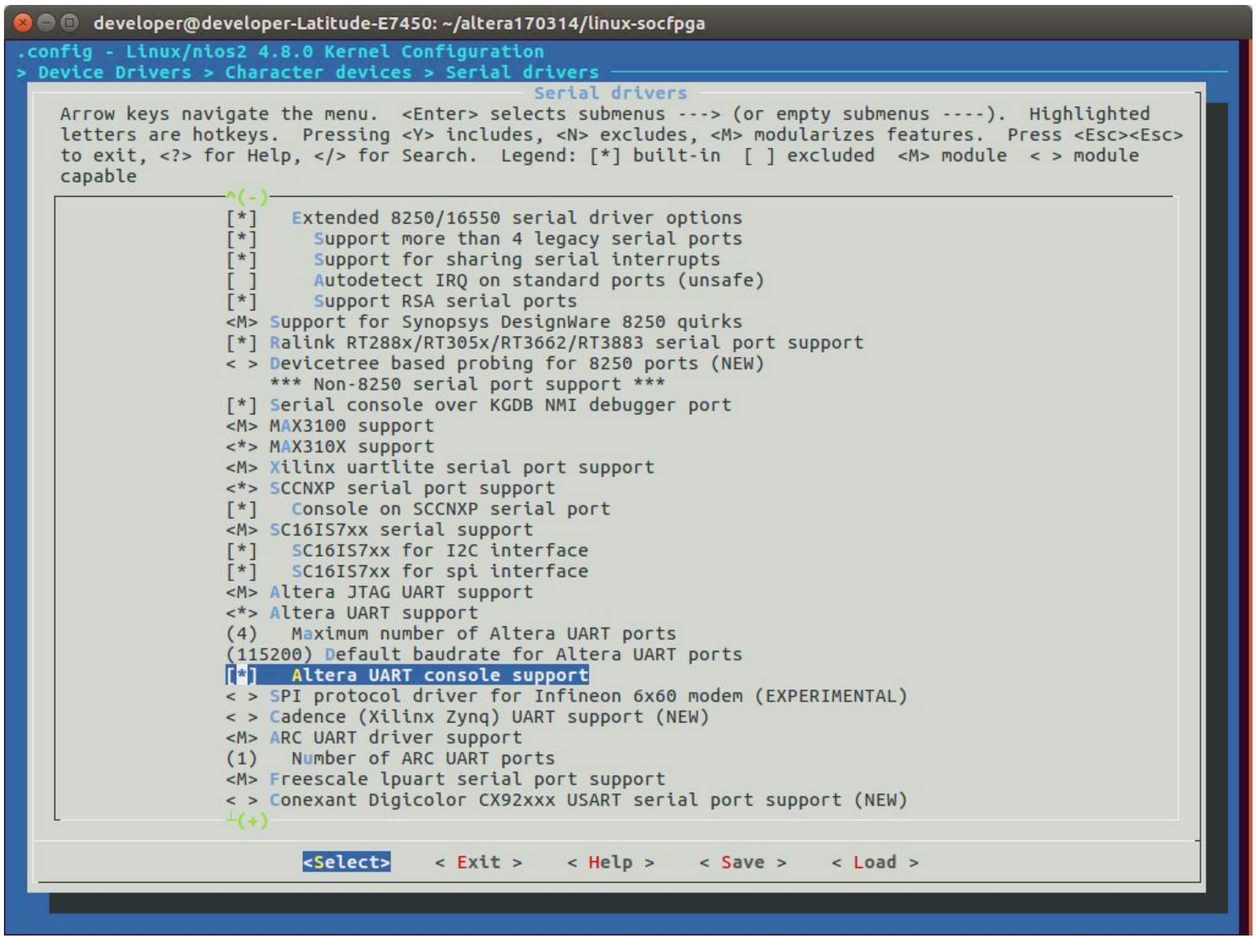
A common problem with theorem proving and model checking today is that theorems and models become increasingly complicated due to more functionalities, more performances and more possible combinations. Therefore it has been proposed that induction proofs can be used instead of checking and proving all possible combinations. In practice an induction proof means proving 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.

2.7.1 Software Programming Model UART Core

We can select UART as a console when we build Linux.



```
developer@developer-Latitude-E7450: ~/altera170314/linux-socfpga
.config - Linux/nios2 4.8.0 Kernel Configuration
> Device Drivers > Character devices > Serial drivers
                              Serial drivers
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted
  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc>
  to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module
  capable
                    ^(-)
                    [*]   Extended 8250/16550 serial driver options
                    [*]     Support more than 4 legacy serial ports
                    [*]     Support for sharing serial interrupts
                    [ ]     Autodetect IRQ on standard ports (unsafe)
                    [*]     Support RSA serial ports
                    <M> Support for Synopsys DesignWare 8250 quirks
                    [*] Ralink RT288x/RT305x/RT3662/RT3883 serial port support
                    < > Devicetree based probing for 8250 ports (NEW)
                        *** Non-8250 serial port support ***
                    [*] Serial console over KGDB NMI debugger port
                    <M> MAX3100 support
                    <*> MAX310X support
                    <M> Xilinx uartlite serial port support
                    <*> SCCNXP serial port support
                    [*]   Console on SCCNXP serial port
                    <M> SC16IS7xx serial support
                    [*]   SC16IS7xx for I2C interface
                    [*]   SC16IS7xx for spi interface
                    <M> Altera JTAG UART support
                    <*> Altera UART support
                    (4)   Maximum number of Altera UART ports
                    (115200) Default baudrate for Altera UART ports
                    [*]   Altera UART console support
                    < > SPI protocol driver for Infineon 6x60 modem (EXPERIMENTAL)
                    < > Cadence (Xilinx Zynq) UART support (NEW)
                    <M> ARC UART driver support
                    (1)   Number of ARC UART ports
                    <M> Freescale lpuart serial port support
                    < > Conexant Digicolor CX92xxx USART serial port support (NEW)
                    └(+)

                <Select>    < Exit >    < Help >    < Save >    < Load >
```

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;

}
```

2.8 Example 3-3 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;
}
```

3 Specification and methodology

We arranged the following sub-goals to complete our main goal.

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 not been 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 (like now) or too complicated to be feasible.

2. **We built and formalized our model:** Our goal is to build and proof-check 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 proofs anas well as checking the models that we build and verifying with real hardware peripherals. The theorems and models we use and build are simplified to enable us conclude intermediate results at the intermediate level. We should combine functional invariants, for example the invariant that our "protected" is always "protected" from unauthorized use, varying the inputs.

3. **We conduct model checking:** Using the appropriate selected tools we write the details of our theory and our model. We can use automatic theorem proving to prove our theory. We can also combine and compare the same scope by checking a formal model of the

hardware peripherals and the connections, as exemplified by a simplified model of a connected UART RS-232.

A known problem with model checking is called 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, report, make relevant conclusion(s) and communicate the result(s):** We document and elaborate about our findings. We will 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 on 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 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

- 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.

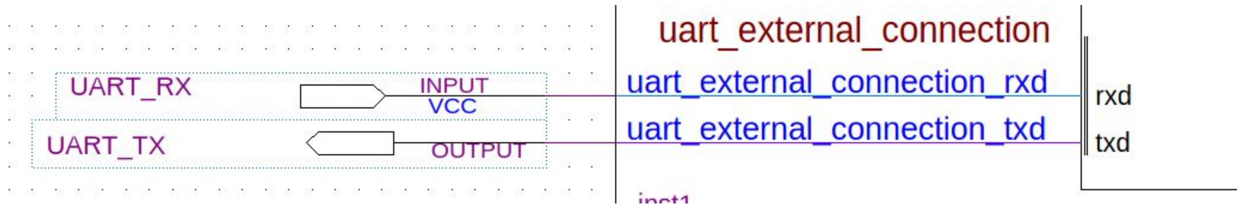
Equivalence checking is a third way that checks the equivalence of a specification and an implementation. Equivalence checking is the most common industry use of formal verification.

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.

3.1 Scope and limitations

We limit the scope of our project to the UART RS-232 and 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, altera_avalon_uart. In Quartus the UART looks as follows.



UART (RS-232 Serial Port)

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

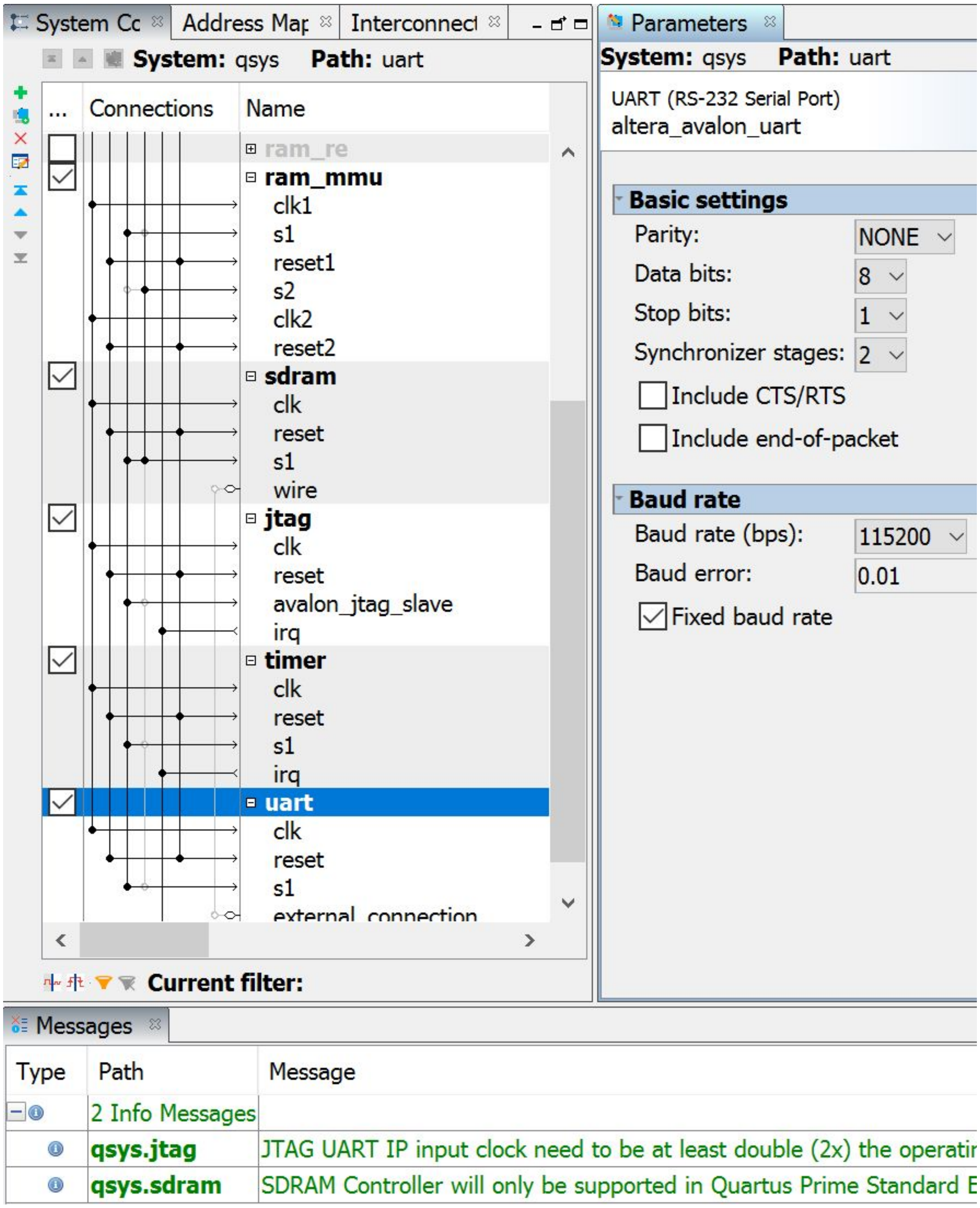
Include CTS/RTS	Include CTS/RTS pins and control register bits
Include end-of-packet	Include end-of-packet register
Baud rate (bps)	standard baud rates for RS-232 connections

The RS-232 settings are provided below for our connection from the PC 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.



...	Connections	Name
☐		⊞ ram_re
✓		⊟ ram_mmu
		clk1
		s1
		reset1
		s2
		clk2
		reset2
✓		⊟ sdram
		clk
		reset
		s1
		wire
✓		⊟ jtag
		clk
		reset
		avalon_jtag_slave
		irq
✓		⊟ timer
		clk
		reset
		s1
		irq
✓		⊟ uart
		clk
		reset
		s1
		external_connection

UART (RS-232 Serial Port)
altera_avalon_uart

Basic settings

Parity:	NONE
Data bits:	8
Stop bits:	1
Synchronizer stages:	2

☐ Include CTS/RTS
☐ Include end-of-packet

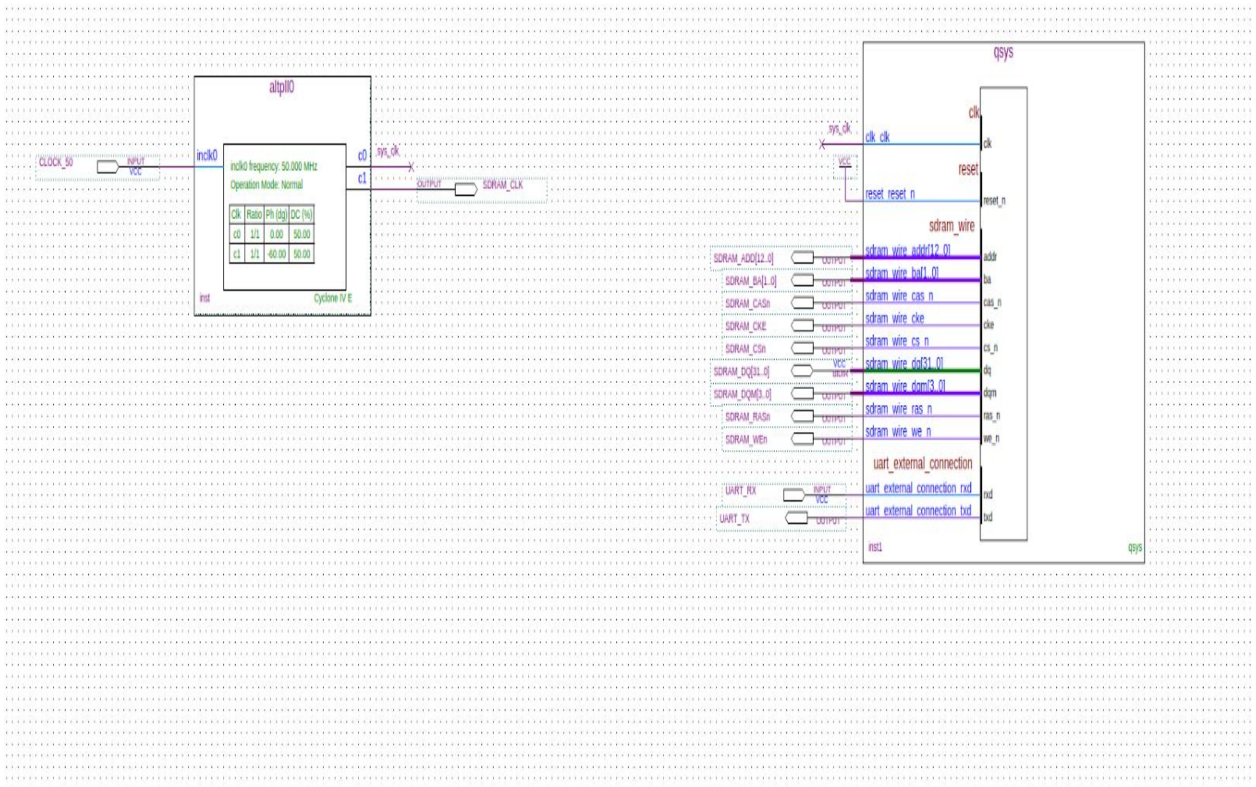
Baud rate

Baud rate (bps):	115200
Baud error:	0.01

✓ Fixed baud rate

Type	Path	Message
	2 Info Messages	
	qsys.jtag	JTAG UART IP input clock need to be at least double (2x) the operatir
	qsys.sdram	SDRAM Controller will only be supported in Quartus Prime Standard E

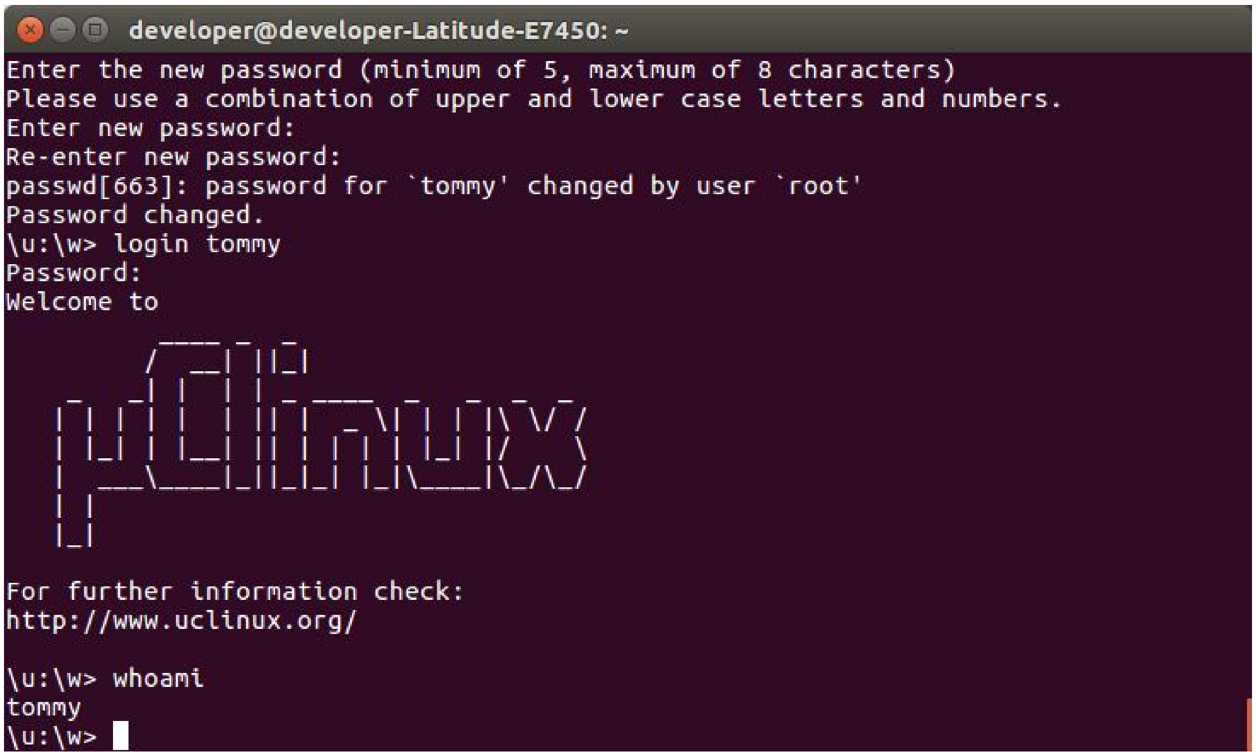
Our 32-bit system looks as follows in Quartus. The clock for the SDRAM is to the left and the UART, RAM, MMU and CPU is to the right.



When building uClinux we selected the console on Altera UART:

Character devices --->
Serial drivers --->
[ ] Altera JTAG UART console support
[*] Altera UART console support

Running embedded Linux with the above system system as a host with a regular UART connection demonstrates that an attacker can get root access just by physically connecting to the UART port with the above configuration and uClinux. There is directly a root access (due to the configuration) and being root one can do anything (e.g. rm -rf / or similar damage or copy any secret data and get the crypto keys from the system) with the embedded system (uClinux).



```
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>
```

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.

3.2 Configuration

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

$$\begin{array}{rl} & \text{Conf} \\ & \wedge \\ \models & \text{Model} \\ & \text{UART} \\ & \wedge \\ \not\models & \text{WRITE} \end{array}$$

The above theorem means that if the configuration is done properly and correctly, the UART connection can't manipulate any protected user processes such as web browser or a command-line interpreter. We are adding more detail and specifics to the theorem so that 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.

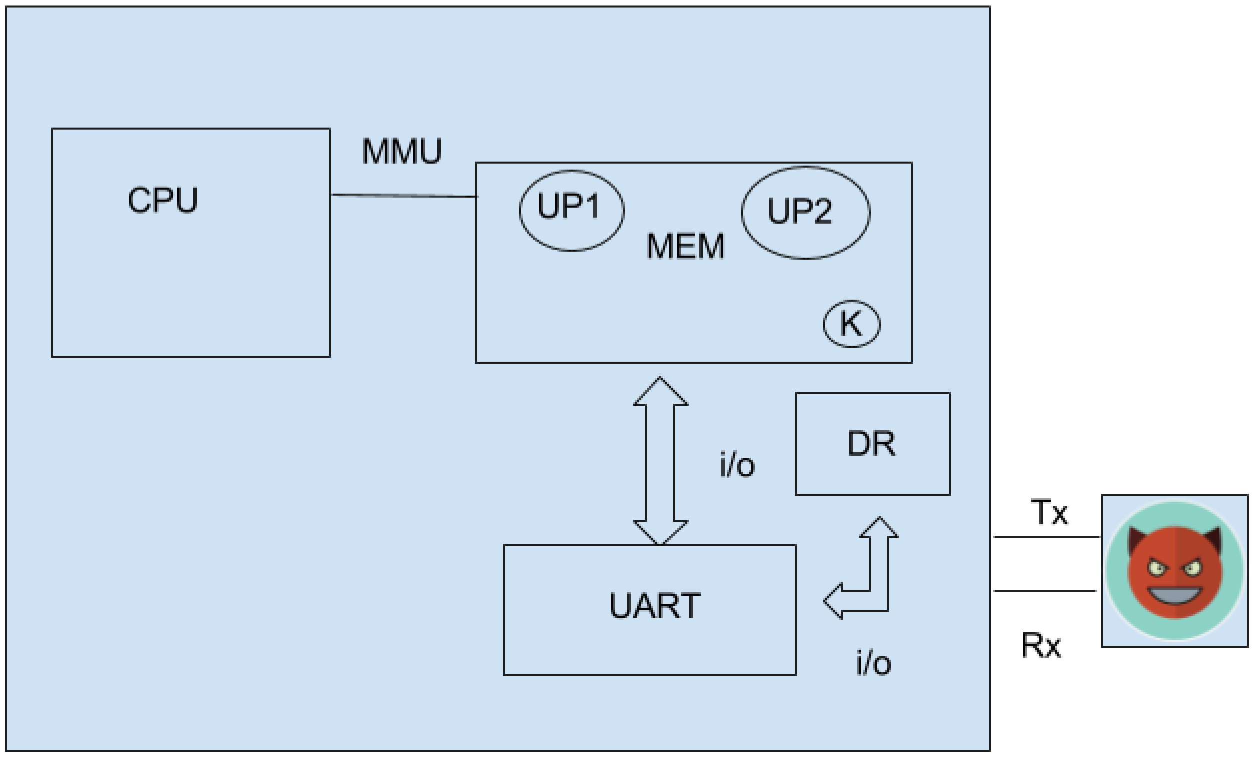


Figure 2. Hardware model overview

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.

3.3 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 of 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.

3.4 Methods for model checking

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

3.5 Results and conclusions

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 ias 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 mote possible attacks.

4 HDL for UART i/o

We used the following code to implement a custom computer with Nios 2 that includes the Altera Avalon UART with which we can connect an external peripheral with an RS-232 interface.

```
        component qsys is

         port (

                clk_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

                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;
```

5 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)
```

5.1 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.

6 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.

7 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 addons 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

  resetcounter = FALSE

  resettimer = FALSE

  count10 = 1

  nextbit = TRUE



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

Chosen state is: 0

NuSMV > simulate -i -k 15
```

8 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] NuSeen http://nuseen.sourceforge.net/