# AMASP (ASCII Master Slave Protocol): a Lightweight MODBUS Based Customizable Communication Protocol for General Applications

Andre L. Delai, Alberto N. Miyadaira, Tania C. Lima

***Abstract*—In this paper, we describe AMASP, a proposal for an open standard communication protocol / model whose objective is providing a novel and lightweight solution to the problem of communication between small computers acting in particular on embedded systems, but not limited to them. The protocol establishes addresses for computers or peripherals, allowing them to send messages directly to these addresses. It is based on message interchanging, being the messages composed by ASCII (American Standard Code for Information Interchange) text or binary streams, according to the user customization. AMASP is very customizable using four different packets to send the messages and manage a master / slave communication model. The protocol is designed to work where a direct communication link is established or a bus architecture is available - by serial, USB connections, etc - and it supports to many error checking algorithms that can be used in unsafe links. Due to the focus on simplicity, low overhead and metadata, it is a connectionless protocol and does not support routing.**

***Index Terms*—AMASP, networking, MODBUS, embedded systems, master/slave, protocol.**

## I. Introduction

THE maker culture [1], [2], [3] creates an interest in studying and understanding the technology underlying embedded systems, as well as in using this knowledge to build custom solutions within the domains of industrial automation, domestic automation (domotics), hobby, art, teaching, research, internet of things etc. Some years ago, a new market emerged to meet the growing necessities of this movement, allowing it to expand. This new market brought new computers, such as Arduino [4], [5], [6] and Raspberry Pi [7], [8], very small machines that can be easily programmed using simplified IDEs (Integrated Development Environments) and that provides operating system (e.g. Linux) resources. These low-cost computer platforms, with good support, open design and very active development/user communities are changing the conceptions of the manufacturers towards understanding the open source and hacking movement as a natural tendency in some segments of the market.

Sometimes, a good solution to problems in the aforementioned domains can be an architecture in which low-power computers can work together attacking small parts of the entire problem, as illustrated in Fig. 1. A more powerful computer can control one or more of these small computers working in a modular architecture to build the complete solution. In this case, a master/slave model can be applied, where a communication protocol in which one computer - called master - can control one or more computers known as slaves. The model can also be called as ”primary/secondary”, ”boss/worker” or other names by some manufacturers in the industry.

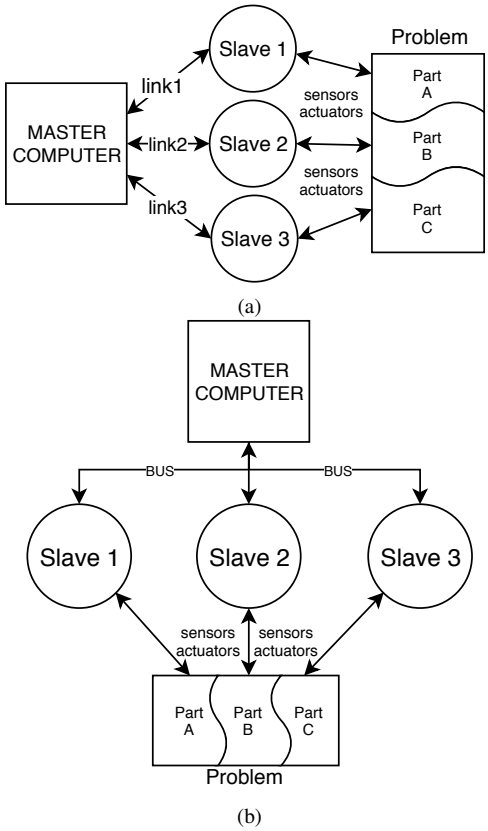


Fig. 1: Examples of a communication architecture. (a) Individual connection lines, (b) Bus architecture

For connecting these computers, in addition to physical connections, it is necessary to have a set of communication rules, known as communication protocols, which makes the talk possible.

The idea for a new protocol came from the necessity

A. L. Delai, T. C. Lima are with the NMI Department , Renato Archer Information Technology Center, Campinas, SP, 13069-901 Brazil e-mail: {aldelai, tclima}@cti.gov.br.

A. N. Miyadaira is with the Federal University of Technology Paraná, Medianeira, PR, 85884-000 Brazil e-mail: miyadaira@utfpr.edu.br

DOI: 10.14209/jcis.2019.1

of building a generic machine-to-machine communication solution for devices developed in the Renato Archer Center [9]. One example is a pen device with haptic technology [10], [11] developed in the center, that is connected to an Arduino board to control an actuator (micro-servo) that returns to the user the position information of a pointer in a screen through the tactile sensitivity of the index finger. The Arduino is connected to a PC, which sets the position of the actuator through a microcontroller analog pin. We intended to design a protocol to work not only with the haptic pen but also with other devices that are build in the center, inserting some level of standardization to the communication increasing the compatibility between the systems.

There are many protocols available today that provide communication rules to send and receive data between computers, industrial and embedded devices, etc. Many of these protocols are very powerful and specialized and in many cases unnecessarily complex for, e.g., basic text or binary message interchange between two computers directly connected. One of the most simple and popular free protocols used in automation today is the MODBUS [12].

Developed by Modicon company (later incorporated by Schneider Electric), the MODBUS protocol [12], [13], [14] was designed to work specifically in industrial automation to control the Modicon PLCs (Programmable Logic Controllers) and RTUs (Remote Terminal Units). It is an application layer messaging protocol, positioned at level 7 of the OSI (Open Systems Interconnection) model [15], which provides client/server communication between devices connected on different types of buses or networks [16]. It uses a frame format (Table I and Table II) based on register access to collect the signals from sensors and to send commands to the actuators in a master-slave model where the slaves are exclusively passive. This means that only the master can initiate transactions (called 'queries'). Slaves respond by supplying the requested data to the master, or by taking the action requested in the query. Typical master devices include host processors and programming panels. Typical slaves include programmable controllers. The master can address individual slaves, or can initiate a broadcast message to all slaves. Slaves return a message (called a 'response') to queries that are addressed to them individually. Responses are not returned to broadcast queries from the master. The Modbus protocol establishes the format for the master's query by placing into it the device (or broadcast) address, a function code defining the requested action, any data to be sent, and an error–checking field. The slave's response message is also constructed using MODBUS protocol. It contains fields confirming the action taken, any data to be returned, and an error–checking field. If an error occurred in receipt of the message, or if the slave is unable to perform the requested action, the slave will construct an error message and send it as its response. [14].

- The Query: The function code in the query informs the addressed slave device what kind of action to perform. The data bytes contain either additional information that the slave will need to perform the function. For example, function code 03 will query the slave to read holding registers and respond with their contents. The data field must contain the information informing the slave which register to start at and how many registers to read. The error check field provides a method for the slave to validate the integrity of the message contents.

- The Response: If the slave makes a normal response, the function code in the response is an echo of the function code in the query. The data bytes contain the data collected by the slave, such as register values or status. If an error occurs, the function code is modified to indicate that the response is an error response, and the data bytes contain a code that describes the error. The error check field allows the master to confirm that the message contents are valid.

MODBUS has been used in automation since the 70s, and it is still very popular in this segment. The classical version has two serial transmission modes, the ASCII (American Standard Code for Information Interchange) and the RTU (Remote Terminal Unit). Operating in serial communication links, both modes utilize asynchronous communication with one character sent at a time with defined framing. The ASCII mode implements a 7-bit character (based on seven-bit ASCII table) to send data, and the RTU an entire byte (8 bits). The most expressive differences between both modes are the way packages are coded and the error checking method (Table I and Table II). MODBUS protocol over serial line takes place at level 2 of the OSI model [17]. The general overview of the ISO MODBUS classification is described in Table III.

TABLE I: Modbus ASCII frame format

| **Start of frame** | **Device Address** | **Function Code** | **Data** | **LRC** | **End of Frame** |
|---|---|---|---|---|---|
| 1 char (:) | 2 chars | 2 chars | n chars | 2 chars | 2 chars (CRLF) |

TABLE II: Modbus RTU frame format

| **Start of frame** | **Device Address** | **Function Code** | **Data** | **CRC** | **End of Frame** |
|---|---|---|---|---|---|
| 4 char times | 8 bits | 8 bits | n x 8 bits | 16 bits | 4 char times |

TABLE III: MODBUS on OSI model

| Layer | OSI Model | |
|---|---|---|
| 7 | Application | MODBUS Application Protocol |
| 6 | Presentation | Empty |
| 5 | Session | Empty |
| 4 | Transport | Empty |
| 3 | Network | Empty |
| 2 | Data Link | MODBUS Master/Slave Protocol |
| 1 | Physical | EIA/TIA-232 or EIA/TIA-485 |

The MODBUS ASCII packet format starts with the ':' character and all other characters in the other fields must be either the numbers 0-9 or the letters A-F since the data is

going to be represented in hexadecimal format but displayed as ASCII characters. For example, function code 03 would be displayed as two ASCII characters '0' and '3'. There is no master identification field in both modes (RTU and ASCII) because the model presumes only one master in the system.

The error checking in ASCII mode is defined by the LCR (Longitudinal Redundancy Check) algorithm. The RTU mode uses a more powerful (and complex) method based on a 16 bits CRC algorithm, which increases the protocol overhead but provides a better method for error checking.

The focus of our communication model are environments where a direct, safety and fast communication is necessary. Where one computer needs to supervise and control other computers and it's peripherals. Easily applicable by students, hobbyists, designers, researchers, etc. A protocol to transport any customized payload types without using complex metadata. Our proposal is an ASCII - American Standard Code for Information Interchange (see appendix A) - based simplified protocol called AMASP (ASCII MAster Slave Protocol), which is inspired by the Modbus ASCII and RTU mode, but with some significant changes. In summary, the chosen design requisites of this new protocol are:

- Simplified design
- Easy understanding for educational purposes
- Easy debugging on serial terminals
- Relative low overhead and meta-data
- Master/slave based model
- Support to generic payloads
- Some customization level
- Good suitability to embedded systems
- Error checking methods for noisy connections

The proposed protocol was designed to suit low-power computers using microcontrollers (e.g. Arduino boards), as well as system-on-a-chip computers (e.g. Raspberry Pi, Beaglebone [18] etc.) and high-performance computers like PCs (Personal Computers). It can be used in embedded systems as a master controlling/monitoring one or more linked slave computers with many peripherals each, providing formatted packets to send messages, to warn the master events occurred in the slave computer as well as communication errors. AMASP is not intended to be a solution to the internet of things field but it can be used as a communication resource for some specific scenarios (e.g. scenarios where protocols like MODBUS are compatible solutions).

This paper is organized as follows. In section II, we describe the elements of the protocol, the communication model, the format of the packets, their fields and the used symbols. In section III, practical examples of use in representative simplified scenarios are shown. Section IV presents an analysis between AMASP and MODBUS protocol differences. A runtime benchmark between 4 different errors check algorithms supported by AMASP is presented in section V. Finally, section VI brings the conclusions about the proposal.

## II. AMASP Protocol Features

Like MODBUS protocol (Table III), AMASP occupies layer 7 of the OSI (Open Systems Interconnection) model, with the serial line version in the layer 2, as illustrated in Fig. 2. It establishes a communication path between linked devices using part of the packet metadata coded in hexadecimal ASCII chars (0 to 9 and A to F). This means that the device IDs (device addresses) are coded using these chars, just like the message length and the error/interrupt codes (see Section III). The exception is the preamble, the end packet chars and the payload (message). Identifying the packet types, a small preamble started by the char '!' (packet beginning) and followed by the char '!', '?', '#' or '~' (the type of the packet itself) is used, and the special chars 'carriage return' and 'line feed' (CR and LF) are defined to signal the end of the packet. The protocol supports message lengths from 1 up to 4096 bytes in a single request or response packet.

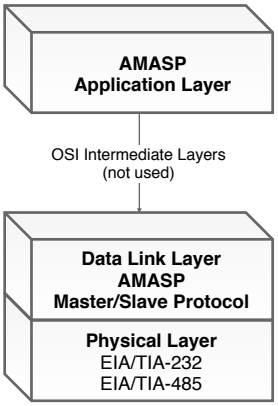


Fig. 2: AMASP OSI Layers

The answer behind the question of why to use the ASCII/Hexadecimal coding is basically that the protocol was designed to be humanly readable, for easy debugging on serial terminals and monitoring software, as well as the MODBUS ASCII mode. This feature generates more overhead in the protocol than the binary version of the MODBUS (RTU) because of the necessity of hexadecimal conversions. These hexadecimal conversions are only for metadata fields, since the payload is inserted in the packet without any kind of conversion, unlike the MODBUS ASCII version where the payload must necessarily be converted into ASCII characters representing hexadecimal values.

AMASP is based on master/slave model but not a traditional one because the slave can, in a specific case, initiate a communication with the master (which will be explained later). We call this 'a pseudo master/slave model'.

A 'device' in AMASP can be recognized as a slave computer or a slave peripheral. It depends on the physical

network topology adopted, as described in Fig. 1. In the first case, where the slaves are connected to individual lines (Fig. 1a), the device IDs are defined as peripheral access addresses. If the topology is based on a bus (Fig. 1b), the device ID can represent access addresses to each slave connected to the bus, so the peripheral access method must be defined inside the payload by the user.

A slave computer may have many different peripherals, like displays, sensors, memories, I/O pins etc. Each of these peripherals have their own particularities in terms of communication and control commands, which it can be treated by customized messages inside the packets. When the device IDs are representing peripheral addresses AMASP can organize them allowing messages to be sent directly to these devices through their respective IDs.

AMASP uses 4 different packet types:

- MRP - Master Request Packet
- SRP - Slave Response Packet
- SIP - Slave Interruption Packet
- CEP - Communication Error Packet

The **master** computer can only send two packet types to the slave: MRP or CEP. The **slave** can use SRP, SIP or CEP to return information to the master. Fig. 3 illustrates the AMASP communication configuration between a master and the associated slave connected by a direct serial line.

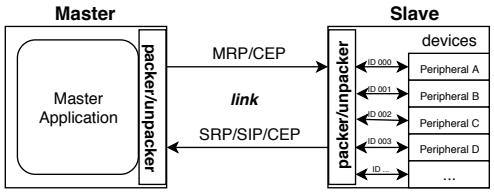


Fig. 3: AMASP Communication model diagram

The **Master Request Packet** is used when the master needs to request or write data to the slave. This happens following the procedure described below:

1) Master sends an MRP to slave
2) Slave must confirm the MRP reception through a **Slave Response Packet (SRP)** or a Communication Error Packet (CEP). The SRP is used only when the master's operation requests data, otherwise, just a simple CEP 00 is used to confirm the reception and the execution of the instruction inside the MRP message (e.g. a write operation in a peripheral). If an error occurs, the returned CEP must contain the appropriate error code (see Table XIII).
3) End of communication

The **Slave Interrupt Packet** must be used only when a slave needs to get the master's attention to an event occurred on it (e.g. a pressed button or a switch state change). The interruption process proceeds as:

1) Slave sends a SIP to the master
2) The master sends a CEP 00 to confirm the reception of the packet and the recognition of the interruption. If an error occurs, the returned CEP must contain the appropriate error code (see Table XIII).
3) The master handles the interruption
4) End of communication

### A. The Master's Request and Slave's Response Packets

Both the MRP and the SRP have the same format (Table IV), but the SRP is always a response to an MRP requisition. Table V and VI shows the component fields of both packets.

TABLE IV: Master request and slave response packet fields

| Packet Type | ECA | Device ID | Msg length | Msg | Error Check | Packet End |
|---|---|---|---|---|---|---|

Msg = Message/payload
ECA = Error Check Algorithm

TABLE V: MRP fields description

| Bytes | Field | Description |
|---|---|---|
| 0..1 | Packet Type | Packet type |
| 2 | ECA | Error Check Algorithm |
| 3..5 | Device ID | Requested device identifier (address) |
| 6..8 | Msg Length | Message size (in bytes) |
| 9..(ML+8) | Msg | Message byte stream (payload) |
| (ML+9..(ML+12) | Error Check | Error Check (16 bits) |
| (ML+13)..(ML+14) | Packet End | Carriage return and line feed chars |

*ML = Message Length

TABLE VI: SRP fields description

| Bytes | Field | Description |
|---|---|---|
| 0..1 | Packet Type | Packet type |
| 2 | ECA | Error Check Algorithm |
| 3..5 | Device ID | Identifier (address) of the responder |
| 6..8 | Msg Length | Message size (in bytes) |
| 9..(ML+8) | Msg | Message byte stream (payload) |
| (ML+9..(ML+12) | Error Check | Error Check (16 bits) |
| (ML+13)..(ML+14) | Packet End | Carriage return and line feed chars |

*ML = Message Length

The **packet ID** has 2 bytes to identify the type of the packet, which is represented by two ASCII chars. Table VII exhibits the symbols and description of the four packet IDs.

TABLE VII: Packet type field description

| Packet ID | Description |
|---|---|
| !? | MRP |
| !# | SRP |
| !! | SIP |
| !∼ | CEP |

In the sequence, the **ECA** field informs the error check algorithm used to generate the value available inside the error check field. This is a one-byte field and the code of the algorithm is an ASCII char which represents hexadecimal values from 0 to F, which means that AMASP can support a total of 16 different error check algorithms. The algorithms

supported in this AMASP version are shown in Table VIII. The algorithms selection was made based on the analysis of the Maxino and Koopman work [19].

TABLE VIII: ECA field description

| ECA | Description |
|---|---|
| 0 | None |
| 1 | XOR 8 bits |
| 2 | Checksum 16 bits |
| 3 | LRC 16 bits |
| 4 | Fletcher 16 bits |
| 5 | CRC 16 bits |
| 6..F | Reserved |

The **device ID** field contains the address of the message receiver device (for MRP packets) or the address of the device that is responding to a request (for SRP packets). The addresses range is from 0 up to 4095. The packet ID 00 is reserved for **broadcasting** where all slaves will receive the same packet. An MRP in broadcasting mode means that the slaves will receive the message and they don't answer the master using any packets. **Message length** represents the size of the message in bytes. **Message field** is the byte stream of the message. The **error check field** uses 4 hex chars to represent a 16-bit redundancy information. This field results from the calculation of the chosen error checking algorithm (defined in the ECA field). Error check algorithms are specifically designed to protect against common types of errors on communication channels. When a receiver receives any AMASP packet, it needs to recalculate the error check data and compare the result to the value code in the error check field. The information inside the packet is valid only if the values match. The error check is calculated over all packet fields, except the end packet chars (carriage return and line feed). When an error check fails the packet is just ignored by the receiver. In safe links (e.g. USB connections), this error check calculation can be disabled to reduce the overhead. The last two bytes are the carriage return and the line feed chars to mark the end of the packet.

### *B. The Slave Interruption Packet*

When a slave computer is working in a process it needs to catch the attention of the master, to communicates an event that occurred in one of its devices, it sends a slave interrupt packet (SIP). This SIP packet (Table IX) contains the device ID and the interrupt code that informs the master about the interrupt subject. If the interrupt code is valid in the system the master must respond to the slave using the CEP packet with the error code 00 (no error) and handle the interruption. If the interruption is not recognized a CEP with error code 02 must be sent back. The SIP uses 2 hexadecimal ASCII chars, which results in 256 different codes per device that can be used by the system. Table X has the description of the SIP fields.

TABLE IX: Slave interruption packet fields

| Packet Type | ECA | Device ID | Int. Code | Error Check | Packet End |
|---|---|---|---|---|---|

TABLE X: SIP fields description

| Bytes | Field | Description |
|---|---|---|
| 0..1 | Packet Type | Packet type |
| 2 | ECA | Error Check Algorithm |
| 3..5 | Device ID | Requested device identifier (address) |
| 6..7 | Interrupt Code | Interrupt type information |
| 8..11 | Error check | Error check information (16 bits) |
| 12..13 | Packet End | Carriage return and line feed chars |

### *C. The Error Communication Packet*

Indicating communication's error or success in the MRP, SRP, or SIP packets, received by one of the computers involved in the communication process, the receiver computer can return a communication error packet (CEP) to the sender computer informing if were a problem in the reception or if it's all fine. The Table XI shows the fields of the CEP packet and the Table XII has the description of all of these fields.

TABLE XI: CEP packet fields

| Packet Type | ECA | Device ID | Error Code | Error Check | Packet End |
|---|---|---|---|---|---|

TABLE XII: CEP fields description

| Byte | Field | Description |
|---|---|---|
| 0..1 | Packet Type | Packet type |
| 2 | ECA | Error Check Algorithm |
| 3..5 | Device ID | Requested device identifier (address) |
| 6..7 | Error Code | Error type information |
| 8..11 | Error check | Error check information (16 bits) |
| 12..13 | Packet End | Carriage return and line feed chars |

The field error code allows 256 different error codes (2 hex chars). In AMASP, the first 4 errors are predefined. The last 252 codes are available to be customized by the user. Table XIII exhibits these codes and their descriptions.

TABLE XIII: Error codes description

| Error code | Description |
|---|---|
| 00 | No errors (reception OK) |
| 01 | Invalid device ID |
| 02 | Invalid interrupt code |
| 03 | Unrecognized message |
| 04..FF | Reserved to the user |

When a packet reception is OK, a 00 code can be sent as a response. In the case that the specified device ID does not exist, a 01 error code is generated. If a sent SIP has an invalid interrupt code (not recognized by the master computer) the 02 error code must be sent back. And finally, when the target device in a slave or a master computer cannot recognize a received message, the 03 code is used to inform the error to the sender. The 04 up to FF error code interval is reserved to the system programmer.

For a better understanding of the particularities of the protocol flowcharts demonstrating the process of receiving packets by the master and slave are shown in Fig. 4 and Fig. 5, respectively.

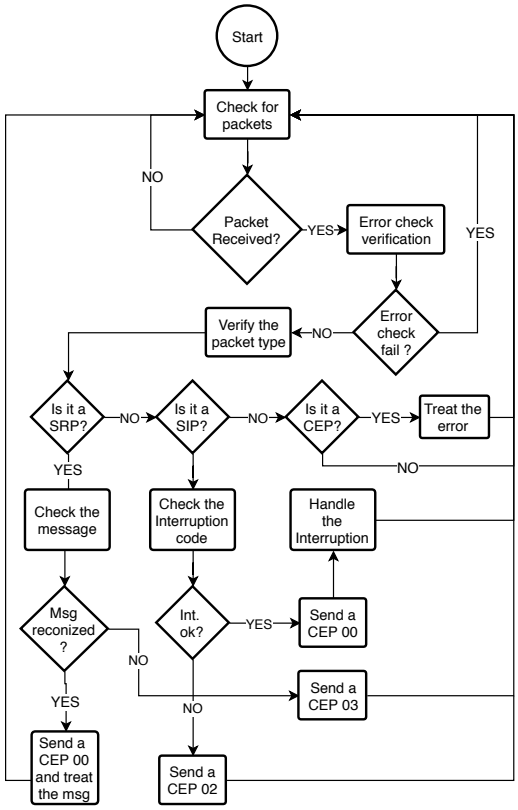


Fig. 4: Master's receiver flow chart

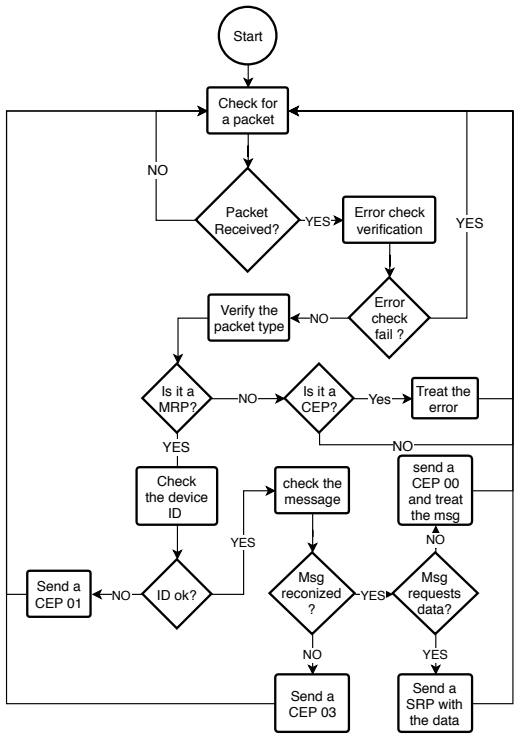


Fig. 5: Slave's receiver flow chart

## III. Practical Examples

### A. Sending a String Message

Assuming a case where the master computer wants to send a text message "Hello Slave!" to the slave computer, and the receiver device in slave has the 00F ID defined by the system programmer as the message interpreter. The MRP assembled package will be as shown in Fig. 6. The error check method selected to the examples was the CRC16.

| PKT TYPE | | ECA | DEVICE ID | | | MSG LENGTH | | | MESSAGE | | | | | | | | | | | | ERROR CHECK | | | | PKT END | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| ! | ? | 5 | 0 | 0 | F | 0 | 0 | C | H | e | l | l | o | | S | l | a | v | e | ! | A | A | 4 | 6 | CR | LF |

Fig. 6: MRP text message example

The slave responds using a SRP packet (Fig. 7)

| PKT TYPE | | ECA | DEVICE ID | | | MSG LENGTH | | | MESSAGE | | | | | | | | | | | | | | | ERROR CHECK | | | | PKT END | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| ! | # | 5 | 0 | 0 | F | 0 | 0 | D | H | e | l | l | o | | M | a | s | t | e | r | ! | D | 5 | B | A | CR | LF |

Fig. 7: SRP text message example

However, if an error occurs, e.g. the message cannot be recognized by the device in the slave, the response must be a CEP packet as in Fig. 8.

| PKT TYPE | | ECA | DEVICE ID | | | ERROR CODE | | ERROR CHECK | | | | PKT END | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| ! | ~ | 5 | 0 | 0 | F | 0 | 3 | 3 | A | 2 | C | CR | LF |

Fig. 8: CEP of a unrecognized message

### B. Generating an Interruption

In an interruption scenario in which, for example, a pressed button event is generated, assuming that the button device is using the 00A ID and that the interrupt code for a pressed button event equals 01, the slave will send a SIP packet to the master (Fig 9).

| PKT TYPE | | ECA | DEVICE ID | | | INT CODE | | ERROR CHECK | | | | PKT END | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| ! | ! | 5 | 0 | 0 | A | 0 | 1 | F | 3 | 1 | 0 | CR | LF |

Fig. 9: Interrupt 01 code example

The master receives the SIP, it recognizes the interruption and replies a CEP as shown is illustrated in Fig. 10.

| PKT TYPE | | ECA | DEVICE ID | | | ERROR CODE | | ERROR CHECK | | | | PKT END | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| ! | ~ | 5 | 0 | 0 | A | 0 | 0 | 3 | A | F | 4 | CR | LF |

Fig. 10: Error code 00 example

### C. Reading a Sensor and Setting an Actuator

As a good "Hello World" test in electronics, we can use AMASP to turn on a LED in the slave computer by pressing a push button. The circuit (Fig. 12) uses the slave input/output pins to receive information from the sensor (button) and provide an action by the actuator (LED). When a button is pressed, as in the latter case, a packet like in Fig. 9 is sent from the slave. The master recognizes the interruption in its programming, but now the answer will be an MRP packet with a message H (the H char). The programming in slave defines the H message (HIGH) in the device ID 00B as a command

to turn on, then the LED will light up. Fig. 11 represents the packet which sends the command to the LED.

| PKT TYPE | ECA | DEVICE ID | | MSG LENGTH | | MESSAGE | ERROR CHECK | | | | PKT END | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| ! | ? | 5 | 0 | 0 | B | 0 | 0 | 1 | H | 9 | 5 | E | C | CR | LF |

Fig. 11: MRP packet to turn on the led

This message in this MRP has no need of data return, so the slave will respond with a CEP 00 if is OK or using another error code if something is wrong.

It is evident that the slave could handle the pressed button event and turn on the led by itself, but in this simple example the intention is showing a process where the decision came from the master.

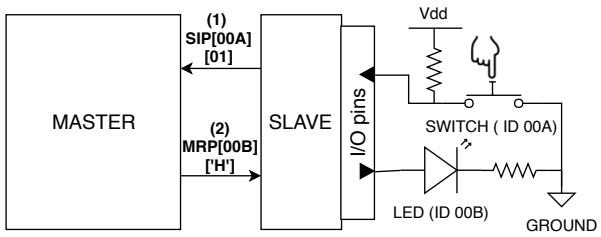


Fig. 12: Turning on a led by a pressed button

## IV. AMASP versus MODBUS Analisys

The MODBUS protocol was first designed to communication between industrial devices. Basically its structure was defined to allow the master read and write on slave's registers. The slaves are originally PLDs (Programmable Logic Devices) with digital and analog devices inside which are accessed by commands or functions codes [16]. The communication is exclusively initiated by the master, the slave has a passive behavior. Monitoring the slaves it is necessary a polling method constantly asking the slaves about its status, which can increase the overhead on the master computer. The type of the frame is basically defined by the function field. In addition to accessing data, function codes also allow diagnostic information frames, such as read exception status, diagnostic, read device identification, etc. The communication between devices using MODBUS must respect these rules being oriented to register access.

AMASP was designed to be a communication protocol with no specific payload formats, which can be fully customized by the users according to each one's own necessities. The packets types are defined by the packet type field and it can be used to transfer data, inform errors and interrupt the master to inform slave's events. It is a communication model that organizes how masters and slaves relate without regulating the way data is trafficked between them. This allows the user to focus only on the payload format and it can be applied up to new protocols giving more flexibility to the project. In this pseudo master/slave model, the master is still in control, but the slave has a resource to communicate to the master some events occurred on it (through interrupt packets), without the necessity of the master's attention all the time. This can be useful when the master has other tasks to do and it still needs to monitor the slaves.

Although AMASP is based on MODBUS, these protocols have significant differences between them, since AMASP has a more generic and user-friendly proposal. Table XIV shows the comparison between the main characteristics of these protocols.

TABLE XIV: MODBUS x AMASP features

| | **MODBUS ASCII** | **MODBUS RTU** | **AMASP** |
|---|---|---|---|
| **Communication hierarchy** | Master/Slave | Master/Slave | Pseudo Master/Slave |
| **Slave status checking method** | Poolling | Poolling | Poolling or Interruption |
| **Payloads/messages format** | Registers (ASCII chars) | Registers (binary data) | Generic |
| **Max. number of Slaves/Devices** | 247 | 247 | 4096 |
| **Masters available** | 1 | 1 | 1 |
| **Broadcast Address** | 0x00 | 0x00 | 0x000 |
| **Error checking** | LRC | CRC-16 | See Table VIII |
| **Frames/packet types** | 1 | 1 | 4 |
| **Frame/Packet metadata** | * up to 17 bytes | *up to 8 bytes | **11 or 14 bytes |
| **Payload capacity** | 2 x 252 chars | 252 bytes | 4096 bytes |
| **Hex. conversion overhead** | Metadata + payload | None | Metadata only |
| **Communication type** | Register access | Register access | Generic |

*Considering register address and range data inside the payload.
**Error/Interrupt packet and Request/Response packet respectively.

### A. Practical Scenario Analysis

Adopting the system described in Fig. 12 and assuming that the system supports both, MODBUS and AMASP protocols at a time, we have a practical scenario for comparison.

The first example we use the protocols to read the actuator status (LED on/off). The slave/device address was defined as 0x0B in both protocols. To MODBUS, the register's address 0x00 is associated with the LED. The zero value in this register means that the LED is off, the one value means that the led is on. Table XV exhibits the frame to get the LED status by MODBUS protocol. The Table XVI represents the packet to get the same information by AMASP.

TABLE XV: MODBUS master frame to get the LED status

| **Query** | | | |
|---|---|---|---|
| **Field Name** | **Ex. (Hex)** | **ASCII Characters** | **RTU** |
| Header | | ":" | None |
| Slave Address | 0B | 0 B | 0000 1011 |
| Function | 03 | 0 3 | 0000 0011 |
| Starting Address Hi | 00 | 0 0 | 0000 0000 |
| Starting Address Lo | 00 | 0 0 | 0000 0000 |
| No. of Registers Hi | 00 | 0 0 | 0000 0000 |
| No. of Registers Lo | 01 | 0 1 | 0000 0001 |
| Error Check | | LRC (2 chars.) | CRC (16 bits) |
| Trailer | | CR LF | None |
| **Total Bytes** | | **17** | **8** |

In the AMASP packet we adopt the message "R" (Read) to be recognized by the slave as a request to inform the LED

status. The number of bytes used by the AMASP packet is between the MODBUS ASCII and RTU versions.

TABLE XVI: AMASP master packet to get the LED status

| **Request** | | |
|---|---|---|
| **Field Name** | **Ex. (Hex)** | **AMASP** |
| Header | | "!?" |
| Device Address | 0B | "00B" |
| Msg Length | 01 | "001" |
| Message (payload) | 52 | "R" |
| Error check | - | Error check (4 chars) |
| End of packet | 03 | CR LF |
| **Total Bytes** | | **15** |

The MODBUS slave response frame to the master's query frame in Table XV is in TableXVII . It uses the same frame header for query and response but with some differences after the function field. In a scenario with multiple slaves connected by a bus (e.g. Fig. 1b) the frame decoding has some particularities. A slave, which is listening to the bus, recognizes a frame addressed to it by the slave address field. It starts the decoding by the frame header recognition and after that decodes the slave address. If the address does not match, the slave ignores the frame, if does, the frame decoding proceeds. Every frame in the bus needs to have its slave address decoded and analyzed by the connected devices.

TABLE XVII: MODBUS response frame from the slave

| **Response** | | | |
|---|---|---|---|
| **Field Name** | **Ex. (Hex)** | **ASCII Characters** | **RTU** |
| Header | | ":" | None |
| Slave Address | 0B | 0 B | 0000 1011 |
| Function | 03 | 0 3 | 0000 0011 |
| Byte Count | 00 | 0 2 | 0000 0000 |
| Data Hi | 00 | 0 0 | 0000 0000 |
| Data Lo | 01 | 0 1 | 0000 0001 |
| Error Check | | LRC (2 chars.) | CRC (16 bits) |
| Trailer | | CR LF | None |
| **Total Bytes** | | **15** | **7** |

The slave response to the AMASP MRP (Table XVI) is shown in Table XVIII. In this case, the header contains the packet type information. In the header analysis, SRPs and SIPs can be ignored by the slaves in the bus, excluding the necessity of slave address decoding and analysis. This feature can be an advantage compared to the ASCII MODBUS because of the hexadecimal conversion overhead to decode the addresses.

TABLE XVIII: AMASP response packet from the slave

| **Response** | | |
|---|---|---|
| **Field Name** | **Ex. (Hex)** | **AMASP** |
| Header | | "!#" |
| Device Address | 0B | "00B" |
| Msg Length | 01 | "001" |
| Message (payload) | 48 | "H" |
| Error check | | CRC (4 chars) |
| End of packet | | CR LF |
| **Total Bytes** | | **15** |

Monitoring the slaves, the master can ask the devices periodically, which is known as polling. AMASP has the possibility of using interrupt packets (SIPs) where the slaves can send small packets to the master to inform events occurred on them. This resource eliminates the number of packets necessary to constantly verify the slaves status.

The Fig.13 exhibits an example with a master monitoring two slaves using the MODBUS protocol and the polling method. No events in slave occurs until $t6$ when the event A occurs in slave 1. This event will be detected only at $t8$ when the response frame informs the master about it. At $t9$ the master sends an action to handle the event A, and the slave confirms the received action as a response at $t10$. After that, at $t11$, another event occurs (event B), but this time in slave 2. As before, it will be informed to the master, which happens at $t13$. In sequence, an action to handle the event is performed by the master ($t14$) and confirmed by the slave ($t15$). After the last occurrence, the polling restarts at $t17$.

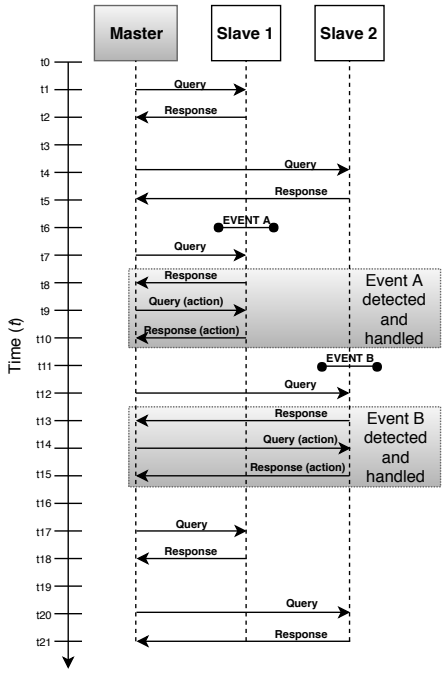


Fig. 13: MODBUS polling method

In Fig.14 we have the same scenario, but now using AMASP and the interrupt method for monitoring the slaves. The master doesn't take any action before the slave 1 informs an event by an interrupt packet (SIP) at $t7$ using a customized code (01) to inform the nature of the event A. The master confirms the interruption by a CEP 00 (no errors) and it sends an MRP to handle the event A at $t14$. The slave 1 confirms the reception of the MRP at $t15$. A new event occurs in slave 2 at $t11$ and it's informed at $t12$ repeating the same steps of the slave 1.

In this particular case, the polling overhead over the master computer is not present and this free processing capacity can be used to execute other tasks.

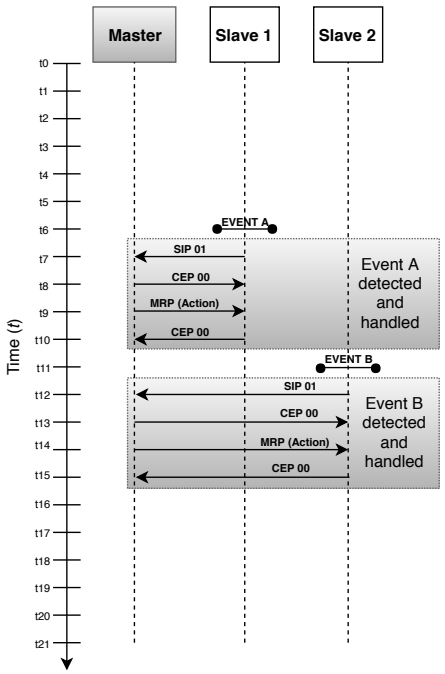


Fig. 14: AMASP interrupt method

Adopting the frame/packet sizes described in Tables XV, XVI, XVII and Table XVIII, we can calculate the number of bytes used to each protocol to monitoring the slaves for the examples on Fig. 13 and Fig. 14. The results can be seen in Table XIX

TABLE XIX: Bytes used by polling and interrupt monitoring methods (based on Fig. 13 and Fig. 14 examples)

| | Polling | Interrupt |
|---|---|---|
| **Protocol** | **Bytes used** | **Bytes used** |
| MODBUS ASCII | 256 | - |
| MODBUS RTU | 120 | - |
| AMASP | 256 | 116 |

Observing the Table XIX we can present a traffic percentage analysis to this particular case. Using the AMASP and interrupt packets, AMASP reduced the byte traffic necessary to monitoring both slaves in 45,81% compared to the MODBUS ASCII, 45.31% to the AMASP by polling and 9.66% to the MODBUS RTU.

## V. Error Check Benchmarks

According to Maxino and Koopman [19], a common way to improve network message data integrity is appending a checksum. Although it is well known that cyclic redundancy codes (CRC) are effective at error detection, many embedded networks employ less effective checksum approaches to reduce computational costs in highly constrained systems. Sometimes such cost/performance trade-offs are justified. However, sometimes designers relinquish error detection effectiveness without gaining commensurate benefits in computational speed increase or memory footprint reduction.

AMASP currently supports 5 different 16-bit error checking algorithms. In the order of low to high computation complexity, they are: the checksum, the XOR, the LRC, the Fletcher and the CRC. In this case, the more complex the algorithm is the less chance of failure to detect errors. The choice of which algorithm will be used depends on the characteristics of the communication line/bus and the computers connected to it. A more efficient error detection algorithm will imply a smaller number of packets sent per unit of time, due to greater computational complexity.

The benchmark was coded in C language, using an Arduino Mega 2560 board [20] and the avr-gcc Atmega cross compiler for Linux. The main features of the Atmega 2560 microcontroller are:

- Microcontroller: ATmega2560
- Flash Memory: 256 KB of which 8 KB used by bootloader
- SRAM: 8 KB
- EEPROM: 4 KB
- Clock Speed: 16 MHz

No operating system in the Arduino Board was used during tests and the specifications of the benchmark are described below:

- The entry of the algorithms are byte streams composed by an AMASP MRP containing 9 metadata bytes + a payload with 8, 64, 128, 256 or 512 bytes.
- In the presented tests we used the micros Arduino API function to time measurement, which returns the number of microseconds since the Arduino board began running a program. This function has a resolution of 4 microseconds. The execution time of the micros function was estimated in 3 microseconds and c it's onsidered in the benchmark of the algorithms.
- Reducing the quantization error, the result for each algorithm is the time average of 100 executions in the same conditions. The execution time of the algorithms was measured by using the average between 100 different payloads filled by random bits (uniform distribution). This is done because the CRC algorithm is sensible to the payload content.
- The time measurement of each algorithm contemplates the execution of the algorithm itself, the algorithm function call and a command to assign the return value to a 16 bits integer variable.

**Algorithm .1** Error Check Benchmark Pseudo-code

```
accTime ← 0
for 1 to 100 do
    payload ← randomBits();
    time ← 0
    for 1 to 100 do
        time ← time + algorithmTime(payload);
    end for
    accTime ← accTime + time/100
end for
return accTime/100
```

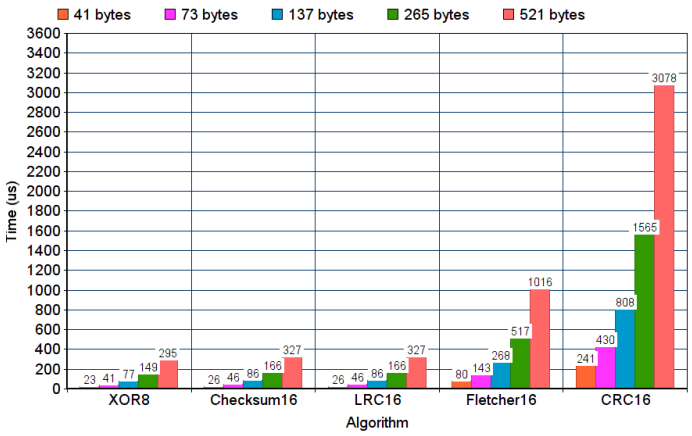


Fig. 15: Error check benchmark in Arduino Mega 2560

The algorithms benchmarks based on runtime show us the XOR as the lightest solution and consequently it allows a greater amount of packets sent per unit of time. However, when analyzing the error detection efficiency [19], the XOR algorithm is one of the worst performance solutions compared to other algorithms. CRC16 is the most efficient in error detection but it is also the most computationally costly (no free lunch).

The choice of the algorithms will depend on the requisites of the system. Powerful processors and networks can provide a satisfactory flow of packets even using the complex CRC16 algorithm. On the other hand, more modest processors can perform satisfactorily if the communication environment does not demand powerful error-control solutions. This is a design decision that the user can make during the configuration of the AMASP protocol.

## VI. Conclusion

AMASP is a proposal to a new open standard protocol and a communication model to general applications between low or high- performance computers in embedded systems or other environments. It is relatively simple, very customizable and a low overhead solution to connect computers in a master/slave model. AMASP mixes and modifies some features from the ASCII and MTU MODBUS versions adding new ones. The possibility to work with interruptions can bring a more efficient solution to monitoring the computers and peripherals in the system than MODBUS in polling mode as demonstrated here.

The support for many different error check algorithms has the advantage to makes the protocol more adaptable to many communication links and computers. Based on the presented checksum tests, users can choose the error check algorithm to their project through a cost-benefit analysis. The ASCII hexadecimal codification of the metadata makes it easier to communication debugging.

Our next steps in this proposal are to optimize the implementation aiming the comparison study between AMASP, MODBUS and other similar protocols in more complex scenarios with many computers and network typologies.

## Appendix A
## Reduced ASCII Table

TABLE XX: Partial ASCII table

| Char | Hex | Dec | Char | Hex | Dec |
|---|---|---|---|---|---|
| LF | 0A | 10 | F | 46 | 70 |
| CR | 0D | 11 | G | 47 | 71 |
| ! | 21 | 33 | H | 48 | 72 |
| # | 23 | 35 | I | 49 | 73 |
| ? | 3F | 63 | J | 4A | 74 |
| ~ | 7E | 126 | K | 4B | 75 |
| 0 | 30 | 48 | L | 4C | 76 |
| 1 | 31 | 49 | M | 4D | 77 |
| 2 | 32 | 50 | N | 4E | 78 |
| 3 | 33 | 51 | O | 4F | 79 |
| 4 | 34 | 52 | P | 50 | 80 |
| 5 | 35 | 53 | Q | 51 | 81 |
| 6 | 36 | 54 | R | 52 | 82 |
| 7 | 37 | 55 | S | 53 | 83 |
| 8 | 38 | 56 | T | 54 | 84 |
| 9 | 39 | 57 | U | 55 | 85 |
| A | 41 | 65 | V | 56 | 86 |
| B | 42 | 66 | W | 57 | 87 |
| C | 43 | 67 | X | 58 | 88 |
| D | 44 | 68 | Y | 59 | 89 |
| E | 45 | 69 | Z | 5A | 90 |

## Acknowledgment

The author would like to thank CNPq (National Council for Scientific and Technological Development - Brazil) for the financial support. Special thanks to engineer Adilson Chinatto (Espectro Engineering LTDA - Brazil), professors Romis Attux and Rafael Ferrari (FEEC Unicamp - Brazil) and engineer Tabata Tomaz (Honda Motor Co. - Brazil).

## References

[1] J. G. Tanenbaum, A. M. Williams, A. Desjardins, and K. Tanenbaum, "Democratizing technology," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI 13*, ACM Press, 2013. doi:10.1145/2470654.2481360.

[2] E. R. Halverson and K. Sheridan, "The maker movement in education," *Harvard Educational Review*, vol. 84, pp. 495–504, dec 2014. doi:10.17763/haer.84.4.34j1g68140382063.

[3] *FabLab: Of Machines, Makers, and Inventors (Cultural and Media Studies)*. Transcript-Verlag, 2014. isbn:978-3-8376-2382-6.

[4] M. Authors, "Arduino website." urlhttps://www.arduino.cc/, Mar. 2018. accessed in 03/05/2018.

[5] M. Banzi and M. Shiloh, *Getting Started with Arduino: The Open Source Electronics Prototyping Platform (Make)*. Springer, 2015. isbn:978-981-10-4417-5.

[6] Y. A. Badamasi, "The working principle of an arduino," in *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*, IEEE, sep 2014. doi:10.1109/ICECCO.2014.6997578.

[7] M. Authors, "Raspberry pi website." urlhttps://www.raspberrypi.org/, Mar. 2018. accessed in 03/05/2018.

[8] S. McManus and M. Cook, *Raspberry Pi For Dummies*. For Dummies, 2017. isbn:1119412005.

[9] M. Authors, "Renato archer information technology center website," Mar. 2018. accessed in 03/07/2018 (in portuguese).

[10] M. R. McGee, P. Gray, and S. Brewster, "Haptic perception of virtual roughness," in *CHI 01 extended abstracts on Human factors in computing systems - CHI 01*, ACM Press, 2001. doi:10.1145/634067.634162.

[11] *Haptic Human-Computer Interaction*. Springer, 2001. isbn:978-3-540-42356-0.

[12] M. Authors, "The modbus organization." http://www.modbus.org, Mar. 2018. accessed in 03/05/2018.

[13] "ModBus," in *The Internet of Things*, John Wiley & Sons, Ltd, dec 2011. doi:10.1002/9781119958352.ch5.

[14] I. Modicon, "Modicon modbus protocol reference guide," *North Andover, Massachusetts*, pp. 112–115, 1996.

[15] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection," *IEEE Transactions on communications*, vol. 28, no. 4, pp. 425–432, 1980. doi:10.1109/TCOM.1980.1094702.
[16] I. Modbus, "Modbus application protocol specification v1. 1a," *North Grafton, Massachusetts (www.modbus.org/specs.php)*, 2004.
[17] S. Automation, "Modbus over serial line–specification and implementation guide," *V2002*, 2002.
[18] M. Authors, "Beaglebone website," Apr. 2018. accessed in 04/05/2018.
[19] T. C. Maxino and P. J. Koopman, "The effectiveness of checksums for embedded control networks," *IEEE Transactions on dependable and secure computing*, vol. 6, no. 1, pp. 59–72, 2009. doi:10.1109/TDSC.2007.70216.
[20] A. Atmel, "Atmel atmega640/v-1280/v-1281/v-2560/v-2561/v datasheet," *2014*, 2014.

**Andre Luiz Delai** was born in Londrina, Paraná, Brazil, in 1978. He is a computer engineer (University of Northern Paraná - 2004) holding a master's degree in electrical engineering from the University of Campinas (2008 - São Paulo - Brazil). He Acted as a college professor in algorithms and data structures, programming, computer architecture and operating systems. Being an embedded system engineer and system analyst in banking automation. Areas of activity are embedded systems, evolutionary computing, artificial intelligence, assistive technologies, evolvable hardware and programming. Currently works as an assistive technology scholarship researcher at the Renato Archer Information Technology Center (Campinas - Brazil).

**Alberto Noboru Miyadaira** is an automation and control engineer from the Faculty Assis Gurgacz (2007 – FAG - Brazil), received a master degree in electrical engineering from University of Campinas (2011 – UNICAMP - Brazil) and a doctor degree in electrical engineering from UNICAMP (2017) and University of Alcalá (UAH - 2017 - Spain). His specialties are control systems, automation, robotics, electronics and embedded systems. He is currently full professor of electrical engineering at the Federal University of Technology - Paraná (UTFPR – Medianeira – Brazil).

**Tania Cristina Lima** received her B.S. in social sciences (1975), M.Sc in social anthropology (1983) and Ph.D. in social sciences (2012) from the University of Campinas (UNICAMP). Has experience acting in social sciences with urban anthropology emphasis, information technology, social computing, social enactive systems, assistive technologies, ethnography and environmental protection. Currently works as an associated researcher at the Renato Archer Information Technology Center (Campinas - Brazil) where is the institutional coordinator of scientific initiation scholarships.