MCVU Application 2
NTP Server v1.2

Version	Date	Remark
1.2	Jun 12	Added Theory Tasks
1.1	Jun 1	Added clearifications and corrections (marked)
1.0	May 18	Initial release

General Remarks

Before you start with the application be sure you are familiar with TinyOS.

When you need to change files from the TinyOS system copy them into your application dir and change them there. This makes it a lot easier to apply future updates of TinyOS. It also makes submitting your application much easier as you only have to submit one folder.

No specification is complete! Design decisions have to be documented in the protocol.

1 High-Level Specification

You have to program a Network Time Protocol (NTP) server based on the TinyOS operating system. Upon an NTP request, your server should query the time of the on-board real-time clock (RTC) and reply according to the time fetched from it.

Additionally, there should be a serial GPS module[1] attached to your server to determine an “accurate” reference time. At every time the GLCD should display both, the current GPS time and the current RTC time. The GLCD should also hold two touchscreen buttons: one of which should set the RTC to the correct GPS time and the other one should set the RTC to the current GPS time but with an offset of $-42h$.

For simplifying the calculation of the NTP timestamp, you can assume a date and time greater or equal to the 01.01.2010 00:00. Therefore, you don’t have to calculate back to the 1.1.1900. You can also assume, that a 32bit timestamp is sufficient and we stay in the NTP Era 0. That is, we will not operate your server at or after the 8.2.2036. Please also note, the granularity of the time you are getting from the RTC is 1 second. Therefore, you can set the fractional part of the NTP timestamps to zero.

[1] In the case of our lab setup we will use a GPS emulator software as the GPS signal is not available at every workplace.

Your NTP server should respond correctly to NTP requests as well as to ICMP Echo requests and properly refuse all other UDP packets (ICMP destination unreachable).

The PC has the static IP address 10.60.0.1 and the NTP server should have the static IP address 10.60.0.10. The subnet address is 255.255.0.0.

⚠ **Note** *Note that you have to change the default destination address of the UDP demo in our lab setting.*

Overview

The external interfaces to the server are shown in Figure 1. These are the "connections to the real world" of the microcontroller application. They consist of the following elements:

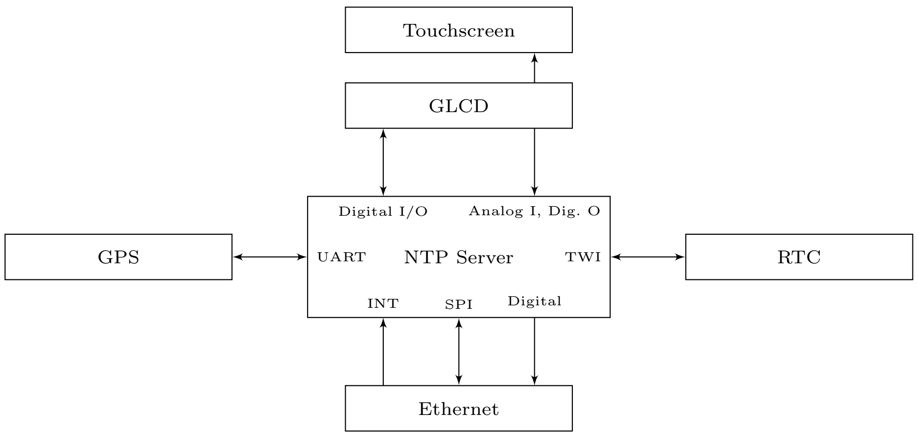


Figure 1: Pong – External Interfaces

Ethernet Module: We are using the *Mikroelektronika Serial Ethernet Board* [3] connected to the bigAVR6 development board. The extension board features the Microchip ENC28j60 Ethernet controller [2] completely implementing the physical as well as the MAC layer for a 10BASE-T Ethernet connection. The chip is connected to the ATmega via SPI. The wire connections between the extension board and the development board are shown in Table 1. You have to ensure that the MMC dip switches for the on-board SD Card reader on SW15 as well as the dip switches for the RS232-B on SW13 are turned off.

Serial Ethernet	bigAVR6
$\overline{\text{RST}}$	PB4
$\overline{\text{CS}}$	PB0
$\overline{\text{INT}}$	PD2
SCK	PB1
MISO	PB3
MOSI	PB2
VCC	VCC
GND	GND

Table 1: Connection plan of the Serial Ethernet extension

Note *There is a collision between the SCL pin of the TWI and the $\overline{INT}$ pin of the Ethernet module as it is set in the default configuration. However, one of the advantages of TinyOS is, that such a collision can be resolved extremely easy. Copy the component definition **Enc28j60C** from `tos/chips_ecs/enc28j60` into your application dir and wire* intETH *to* HplAtm128InterruptC.Int2. *You also have to wire the* intPin *to* IO.PortD2. *Use `git pull` to get the most recent version of the provided modules. Now the collision is resolved and the module should work with the above connection plan.* ~~*(In the initial release of TinyOS you also have to wire* intPin *to* IO.PortD2. *However, this need for double wiring if the pin is already removed and after `git pull` this is not necessary any more.*~~

Note *For debugging we have installed wireshark in the lab. You can use it by running `wireshark-eth1` and proceed with "run unprivileged". Debugging via `printf` is available even if networking is used as the interrupt pin only blocks the receive pin of the Uart1. Ensure that you have disconnected the PC by disabling the dip switch 7 on SW13.*

Real-Time Clock: The Maxim DS1307 real-time clock for the application is already included in the bigAVR6 development platform. It is connected to the ATmega by a serial TWI and is controlled by NV SRAM registers. To connect the RTC to the microcontroller you have to enable the dip switches 5 and 7 on SW14. Initially, the RTC should be set to *Mon 01.02.2010 15:51.* The RTC should be set on startup and on interactions of the user. During operation it should run by itself.

GPS: Since our lab does not allow the use of physical GPS modules as the received signal strength is to low (at least at most work places) we are using a GPS emulation software called gpsfeed+[2] to get time information in the NMEA format. gpsfeed+ can be configured such that it sends out the NMEA messages via the serial console to the development board. You have to parse the incoming stream and extract the time and date information from the $GPRMC command to provide it to your application. Even if you

[2]gpsfeed+ is available for download on the course homepage at `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/misc/task2-specific-stuff/gpsfeed-.tar.gz/view`.

can configure gpsfeed+ in a way that it sends only $GPRMC commands, your application also has to work with other commands enabled.

In principle your application should work with the standard UART setting of 4800bit/sec 8N1 specified by NMEA. However, faster baudrates are also allowed (document in protocol).

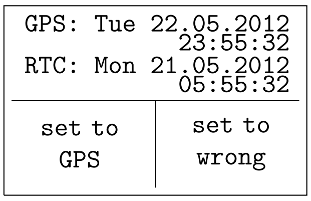
The GPS should communicate with the microcontroller using the Uart0 of the ATmega, connected to the *RS232-A* channel of the bigAVR6 board. Therefore, you have to enable dip switches 5 and 6 on SW13.

GLCD & Touchscreen: The *GLCD interface* gives feedback to the user via an LC–Display while the touchscreen allows the user to interact with your program.

The GLCD should hold (at least) the actual – with respect to a precision of 1 sec – weekday, time, and date from the RTC as well as the actual weekday, time, and date from the GPS. Additionally, there should be two buttons triggered by the touchscreen. One should set the RTC to the correct GPS time and date, the other button should set the RTC to a time different to the one from the GPS. The time offset should be

$$t_{RTC} = t_{GPS} - 42h$$

where t_{RTC} corresponds to the time and date the RTC will be set and t_{GPS} corresponds to the time and date received by the GPS. A sample display output is shown in Figure 1.



```
GPS: Tue 22.05.2012
           23:55:32
RTC: Mon 21.05.2012
           05:55:32
```

Figure 2: Sample Output on the LC–Display

The displayed weekday should be accurately calculated for every day between 1.1.2000 until 31.12.2050 (both included). Please note, that this bounds do not conflict with the bounds given before for the NTP timestamps as the server itself may not be working but the display of time and date should also work in this extended time range. For the weekday calculation please find a good tradeoff between time needed for calculation and needed memory.

Note *There are very easy and efficient algorithms for calculating the day of the week. Do not forget to take care of leap years!*

You have to ensure that the interaction of a user with the touchscreen is correctly recognized. As usually, the actual "design" of the interface is not specified (even if in this Application the possibilities are more limited).

To enable the GLCD and the touchscreen you have to enable the dip switches 1-4 on SW13 as well as dip switch 8 on SW15. The voltage reference jumper J18 should be set to VCC and jumper J15 should be disconnected from analog input channel 0 and 1.

2 Detailed Specification

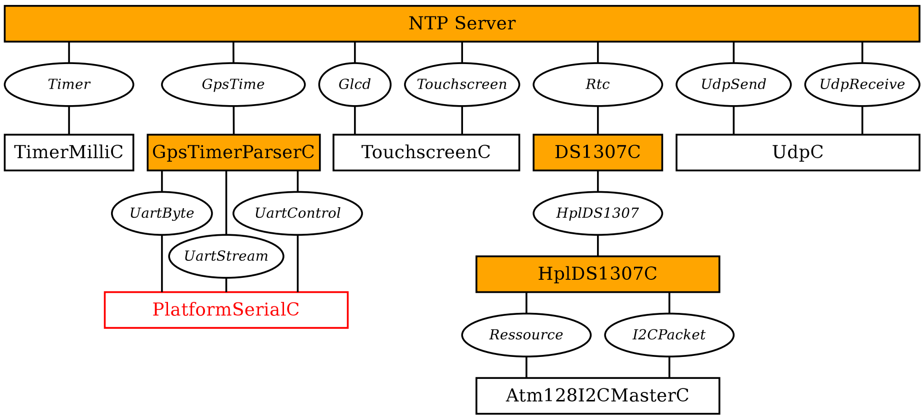


Figure 3: TinyOS Interface proposal

A proposal of TinyOS modules used in the application is shown in Figure 3. Square boxes mark modules while elliptic nodes denote interfaces. The modules filled in orange have to be implemented by you while the white modules are already provided by us. Note that the interface proposal is not complete as, depending on your implementation, you might need additional modules and there might be other modules below the ones that are shown. In fact you will have to alter some modules implementing the network functionality as requested by the specification.

The modules are explained in the following:

2.1 GpsTimerParserC (*GpsTime*)

The module **GpsTimerParserC** has to be programmed by you. It provides the interface *GpsTime* which is used by the application and shown in Figure 4. Please note that the variable type *timedate_t* used in the interface is not given by us and has to be defined by yourself.

The module uses up to three interfaces: *UartByte*, *UartStream*, and *Uart-Control* provided by the module **PlatformSerialC** which are used for serial communication with the (virtual) GPS module. A demonstration how this interfaces can be used is provided in the Uart demo contained in the apps_ecs section of the TinyOS release. You should use the interface *UartStream* for reception of serial data as it is interrupt driven and provides both, an event called upon the reception of a single byte, as well as the possibility to receive a bunch of bytes at once. Please note that you need to enable the receive interrupt when using the *UartStream* interface as otherwise the reception via the

```
interface GpsTime{
    /**
     * Starts the parsing service of the GPS output stream
     */
    command void startService( void );

    /**
     * Stops the parsing service of the GPS output stream
     */
    command void stopService( void );

    /**
     * Notification that a time and date was parsed
     */
    event void newTimeDate( timedate_t newTimeDate );
}
```

Figure 4: Interface GpsTime

UartByte interface was broken and we disabled them per default. The interrupt can be enabled by calling `UartStream.enableReceiveInterrupt`. For further details see the updated Uart demos.

2.2 TouchscreenC (*Glcd, TouchScreen*)

The module **TouchscreenC** is given by us and handles the output on the graphical LCD as well as the input via the resistive touchscreen. For this reason, it provides two interfaces: *Glcd* partly described in Figure 6 and *Touchscreen* described in Figure 5. The initialization of the module is directly wired to the *SoftwareInit* stage of TinyOS.

The command `getCoordinates` requires a pointer to a place where the module can put the result. Please note that the data is only valid after the event `coordinatesReady` was triggered. For type definition of the variable type *ts_coordinates_t* have a look at the corresponding header file. There is also a demo in apps_ecs that demonstrates the use of the touchscreen module.

2.3 DS1307C (*Rtc*)

The module **DS1307C** has to be implemented by you and provides the interface *Rtc*, a high-level interface to the on-board real-time clock (RTC). The interface is described in Figure 7. The module uses the interface *HplDS1307* described in Section 2.4 which provides a register read/write access to the RTC. For every command of the interface *Rtc* the module should do the following: (1) try to open the TWI, (2) if successfully opened, alter the register according to the function that has to be provided (be careful to use R-M-W for register access as you for example should not change the time when stopping the RTC). (3) close the TWI.

Please note that you have to implement a state machine as the commands need more than one action of the Hpl module, and that the events from the Hpl module are declared async and thus (a) special care has to be taken if the global state is accessed by async events as well as by synchronous commands

```
interface TouchScreen{
    /**
     * Sets calibration offsets for the touchscreen
     * \@return SUCCESS
     */
    command error_t calibrate( int8_t x_offset , int8_t y_offset );

    /**
     * Triggers request for touch coordinates
     * \@param pointer to buffer for coordinates
     * \@return SUCCESS if request was accepted
     *            EBUSY if another request is pending
     */
    command error_t getCoordinates( ts_coordinates_t *xy );

    /**
     * Notification that coordinates are ready
     */
    event void coordinatesReady( void );
}
```

Figure 5: Interface TouchScreen

and (b) they have to be synchronized inside the module by a task as the event `timeReady` of the provided interface *Rtc* is synchronous.

2.4 HplDS1307C (*HplDS1307*)

This module has to be programmed by you. It implements the hardware presentation layer to the DS1307 real-time clock [1] and therefore provides the interface *HplDS1307* described in the Figure 8.

The module has to properly access a shared resource, the TWI. To do so it uses the interface *Resource* and the interface *I2CPacket* provided by the module **Atm128I2CMasterC**. We recommend the following implementation strategy: The command `open` checks if the TWI resource is available (we recommend to use the `immediateRequest` command from the *Resource* interface as it directly returns, if the resource is available). After `open` was successful, register read and register write commands as well as bulk read and bulk write commands can be issued. Please note, that events from *I2CPacket* interface are async ones and therefore special care has to be taken if they access state variables used by other sync commands in the module. Clearly, all read and write commands have to check if the resource is assigned to the module (the open command was successful). The command `close` checks, if there is currently a read or write operation in progress and if not, it releases the resource.

Again, the variable type *ds1307_time_mem_t* used in the bulk read and bulk write commands is not defined by us. When defining it you should make use of stucts, bitfields, and unions to get easy access to all required places.

For detailed information about shared resources refer to TEP108.

2.5 UdpC (*UdpSend, UdpReceive*)

This module is given by us and implements the UDP communication. The module is a generic component and takes the UDP port as parameter on ini-

```
interface Glcd{
    /**
     * Set pixel
     * @param x-coordinate
     * @param y-coordinate
     * @return SUCCESS
     */
    command error_t setPixel(const uint8_t x, const uint8_t y);

    /**
     * Clear pixel
     * @param x-coordinate
     * @param y-coordinate
     * @return SUCCESS
     */
    command error_t clearPixel(const uint8_t x, const uint8_t y);

    /**
     * Draw line
     * @param first point x
     * @param first point y
     * @param second point x
     * @param second point y
     * @return SUCCESS
     */
    command error_t drawLine(const uint8_t x1, const uint8_t y1,
                             const uint8_t x2, const uint8_t y2);

    /**
     * Draw rectangle
     * @param upper left x
     * @param upper left y
     * @param lower right x
     * @param lower right y
     * @return SUCCESS
     */
    command error_t drawRect(const uint8_t x1,const uint8_t y1,
                             const uint8_t x2,const uint8_t y2);

    ...

    /**
     * drawText
     * @param text
     * @param x-coordinate of lower left edge
     * @param y-coordinate of lower left edge
     * @return SUCCESS
     */
    command void drawText(const char *text,
                          const uint8_t x, const uint8_t y);
}
```

Figure 6: Part of the interface Glcd

```
interface Rtc{
    /**
     * Starts the Realtime Clock
     * @param data  Starts the Clock, setting date/time
     *              If NULL is passed, the clock is
     *              started without changing anything.
     * @return SUCCESS if bus available and request accepted.
     */
    command error_t start( rtc_time_t *data );

    /**
     * Stops the Realtime Clock without changing the time
     * @return SUCCESS if bus available and request accepted.
     */
    command error_t stop( void );

    /**
     * Read time
     * @param data  A point to a data buffer
     *              where the results will be stored
     * @return SUCCESS if bus available and request accepted.
     */
    command error_t readTime( rtc_time_t *data );

    /**
     * Notification that the results are ready
     */
    event void timeReady( void );
}
```

Figure 7: The interface Rtc

```
interface HplDS1307{
  /**
   * Opend I2C device for communication with DS1307
   * @return SUCCESS if bus available and request accepted.
   */
  command error_t open( void );

  /**
   * Releases I2C device
   * @return SUCCESS if device was idle and is now closed.
   */
  command error_t close( void );

  /**
   * Reads the register of the DS1307 at the given address
   * @param address  Address of the register to be read.
   * @return SUCCESS if bus available and request accepted.
   */
  command error_t registerRead( uint8_t address );

  /**
   * Sets the DS1307 register at the given address to a given value
   * @return SUCCESS if bus available and request accepted.
   */
  command error_t registerWrite( uint8_t address,  uint8_t data );

  /**
   * Reads all time registers at once into the given structure
   * @param data A point to a data buffer where the results will be stored
   * @return SUCCESS if bus available and request accepted.
   */
  command error_t bulkRead( ds1307_time_mem_t *data );

  /**
   * Writes all time registers at once
   * @param data A point to a data buffer which values to set
   * @return SUCCESS if bus available and request accepted.
   */
  command error_t bulkWrite( ds1307_time_mem_t  *data );

  /**
   * Notification that the results are ready
   */
  async event void registerReadReady( uint8_t value );

  /**
   * Notification that the register is set
   */
  async event void registerWriteReady( void );

  /**
   * Notification that the results are ready
   */
  async event void bulkReadReady( void );

  /**
   * Notification that the register is set
   */
  async event void bulkWriteReady( void );
}
```

Figure 8: The interface HplDS1307

tialization. The module provides the interface *UdpReceive*, using the UDP port from the initialization as listen port, as well as the interface *UdpSend*, using the UDP port as sender port. It is possible to instance this component and only wire the receive interface and it is also possible to only wire the send interface. Furthermore it is possible to use multiple instances of the component in the same module (but with different port parameters).

There is a demo in the apps_ecs folder demonstrating the use of the network stack.

To implement the whole functionality of the application you have to dig into the network stack so you can implement the ICMP echo reply as well as the proper rejection of UDP packets received on ports on which no module listens. If you are changing something inside the network stack, please ensure that you copy the necessary file into your application and make the changes there and not directly inside the provided module as stated in the general remarks on the very beginning of the document.

⚠ **Note** *For implementing the additional functionality you have to dig a bit into the network stack (only a bit) and you have to know what to send when. However, the ratio between the points to get and the LOC to write is quite high. One hint: the interface* IpPacket *is wired for purpose.*

For getting an overview of the network stack you can use *make bigAVR6_1280 docs* or even *make bigAVR6_1280 appdoc*.

2.6 TimerMilliC

Provides the *Timer* interface you might need as system timer. For a detailed description of the timer system in TinyOS please refer to the TinyOS documentation (especially the TinyOS programming manual and for more in-depth knowledge also TEP 102).

3 Theory Tasks

Please work out the theory tasks very carefully, as there are very limited points to gain!

1. **[2 Points] GPS Fault Tolerance:** Assume that GPS modules may fail in an arbitrary way (i.e. produce arbitrary incorrect timing information) and you would like to tolerate up to $f > 0$ faulty GPS modules in your application. One technique to do so, is to use $n > 0$ replicas of the GPS module, let the microcontroller read timing values from all the GPS modules (for simplicity we assume that this can be done at once in 0 time), and let it calculate an f-fault-tolerant mean value $avg_f(t_1, \ldots, t_n)$ of the read values $t_1, \ldots, t_n$. Thereby $avg_f(t_1, \ldots, t_{2f+1})$ is defined as the average of its arguments after dismissing the f largest and the f smallest values, e.g., $avg_1(1, 0.5, 2, 2, 2) = (1 + 2 + 2)/3$. Again we assume for simplicity that calculating avg_f can be performed in 0 time. Further assume that there exists a $\pi \geqslant 0$ such that for all $t \geqslant 0$, a non-faulty GPS module read at Newtonian real-time t returns timing value $gps(t)$ with $|t - gps(t)| \leqslant \pi$.

   (a) Formally prove that, if $n \geqslant 2f+1$, the above technique yields a result as good as if only a single non-faulty GPS module would be read, by showing that for all $t \geqslant 0$, $|t - avg_f(gps_1(t), \ldots, gps_n(t))| \leqslant \pi$ holds.

   (b) Assume that two microcontroller share the same $n \geqslant 2f+1$ replicated GPS modules. Denote by $gps_i(t)$ the timing value returned by GPS module i, $1 \leqslant i \leqslant n$, if read by microcontroller 1 at real-time time t, and by $gps'_i(t)$ the timing value returned by GPS module i, $1 \leqslant i \leqslant n$, if read by microcontroller 2 at real-time time t. Formally prove that if both controllers read the GPS modules at the same time t, then they set their times to values at most 2π from each other, i.e., for all $t \geqslant 0$, $|avg_f(gps_1(t), \ldots, gps_n(t)) - avg_f(gps'_1(t), \ldots, gps'_n(t))| \leqslant 2\pi$ holds.

   Remember that faulty GPS modules can return arbitrary values and thus may respond differently to controller 1 and controller 2 even if read at the same time t, i.e., $gps_i(t)$ and $gps'_i(t)$ are not necessarily equal for a faulty GPS module i.

2. **[2 Points] Rounding of Values:** Consider our microcontroller platform. Assume that *signed* integer value a, with $-2^7 < a < 2^7$, is stored in a general purpose register r. Let n be a natural number. Provide assembler code (You can assume that n is a constant known at the time of programming. You do not need to write a function for different n.) that uses as less CPU cycles as you can think of such that:

   (a) After its execution, a holds the value $\lfloor a/2^n \rfloor$. State an upper bound on your solution's number of cycles in terms of n.

   (b) After its execution, a holds the value $\lceil a/2^n \rceil$. Again state an upper bound on your solution's number of cycles in terms of n.

   (c) After its execution, a holds the value "round to nearest integer" of $a/2^n$. Thereby "round to nearest integer" of a positive x is defined as $\lfloor x \rfloor$ if the fractional part of x is less than 0.5, and $\lceil x \rceil$ otherwise. For case (c) you can assume that a is positive ($0 \leqslant a < 2^7$). Again state an upper bound on your solution's number of cycles in terms of n.

Note, that solutions that use significantly more CPU cycles than our solutions will not be rated fully. Since n is given at the time of programming, you can use statements like, "n times execute the sequence: andi r, 2; inc r". The latter code would have $2n$ as an upper bound on the number of cycles.

3. **[3 Points] Rounding of Larger Values:** Consider the same setup as in the exercise before, however, this time a must fulfill $-2^{8R-1} < a < 2^{8R-1}$ where $R > 0$. The binary value of a is stored in $R > 0$ general purpose registers $r_R, \ldots, r_1$ in the following way: the 8 most significant bits of a are stored in r_R, down to the 8 least significant bits of a that are stored in r_1.

   Provide solutions for (a), (b) and (c), assuming that n and registers $r_R, \ldots, r_1$ are given at the time of programming. Again, for case (c) you can assume that a is positive ($0 \leqslant a < 2^{8R-1}$). Give upper bounds on the number of cycles required by your solutions in terms of n and R.

   Note, that solutions that use significantly more CPU cycles than our solutions will not be rated fully.

4 Demonstration and Protocol

If you want points for your application, you have to submit a **.tar.gz* archive to myTI until 28.06.2012, 23:59. Before you have to demonstrate *your* program to *your* tutor! Be prepared that the tutor asks you questions about your implementation you have to answer. Only submissions that were approved by the tutor are counted. You also have to write a LaTeX protocol explaining your implementation and the decisions you have made during implementation. It should also contain a break down of your working hours needed for the application and for which tasks you spent how much time (e.g., reading manuals, implementation, debugging, writing the protocol, ...). A LaTeX template will be provided by us.

5 Grading

Please note that the points below are upper bounds that are possible to reach. However, there is always the possibility of point reduction.

Therefore, if you aim for 16 points you should not gamble on dropping all theory tasks. Also remember the "Collaboration Policy" at the course web page which states 15 points detention for cheating for all involved. Discussion among students is welcome, but this is no group task. All programming and theory tasks have to be done by your own!

Points	Sub	Part
16		Application
	5	App
	5	Uart + Parser
	3	UI
	2	Port unreachable
	1	Ping
7		Theory Tasks (every task is evaluated separately).

References

[1] Maxim Integrated Products. *DS1307 – 64 x 8, Serial, I^2C Real-Time Clock*. `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/DS1307.pdf/view`.

[2] Microchip. *ENC28J60 Stand-Alone Ethernet Controller with SPI Interface.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/ENC28J60.pdf/view`

[3] MikroElektronika. *Serial Ethernet Additional Board Manual.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/serial_ethernet_manual_v100.pdf/view`