Design and Integration of Embedded Systems (BMEVIMIMA11)

Homework Assignment

The description of the problem

In a supervisory control and data acquisition (SCADA) system, there are *field objects* (e.g., controlled machines) and *control objects* (e.g., modules of the plant level control system). These objects have to exchange their data regularly: field objects send their status while the control objects send commands. This data exchange is provided by a protocol that includes the following phases:

- Establishing the connection: a link is built between a field and a control object.
- Object data transfer: each object sends its data to the other object in a regular manner.

The protocol uses the following messages:

- OBJ1 and OBJ2: Messages used in the phase of establishing the connection.
- OBJUP: Data transfer from the field object to the control object.
- OBJDOWN: Data transfer from the control object to the field object.

The connection is established in a two-way handshake: the first message is initiating the connection, and the response to it serves as an acknowledgment. In particular, in the field object, the connection is initiated by sending OBJ1 and is acknowledged by receiving OBJ2. In the control object, sending OBJ2 initiates and receiving OBJUP acknowledges the connection. Object data are transferred regularly via messages OBJUP and OBJDOWN. The operation of the protocol is time-dependent, i.e., objects have to react to specific timeouts (e.g., by resetting the connection or sending a message).

The diagrams in Figure 1 and Figure 2 represent the connection handling and data transfer of the field and control objects, respectively.

- Receiving a message is denoted by *receive(…)?*, while sending is denoted by *send(…)!*.
- The objects have separate local timers, *clkReset* and *clkSync*, and two time parameters *tRTMax* and *tSync*. Timer *clkReset* is used to measure the time between receiving relevant messages to keep the connection alive (the related time limit is *tRTMax*), while *clkSync* is used to measure time between sending messages (the related time limit is *tSync*). Reset of a local timer is denoted by *reset* while checking a timeout is denoted by a guard in square brackets, e.g., *[clkReset≥tRTMax]*. Transitions between states are triggered by received messages or by timeouts; the related actions (if any) are sending messages and/or resetting timers. Note that if a transition is labeled by more than one trigger, then these represent alternatives. Guards written inside of a state symbol represent state invariants: the state has to be left before any of the invariants is violated (this is assured by the outgoing transitions).
- In the *field object* (Figure 1), the protocol is initially in state *Reset*. When the timer *clkSync* reaches *tSync*, the field object sends OBJ1 to the control object and traverses to state *Connecting* (with resetting the timers). In the *Connecting* state, it waits for maximum *tRTMax* time units for the OBJ2 message that acknowledges the connection; in the meantime it keeps sending OBJ1 periodically. If OBJS2 is received in time, the protocol state is set to *Connected*, and the data transfer phase starts. Otherwise, as timer *clkReset* reaches *tRTMax*, the protocol resets. In the Connected state, when *clkSync≥tSync* is reached, the field object sends an OBJUP message. If the time since the last OBJDOWN message received reaches *tRTMax*, the connection resets. The field object is also prepared to receive messages in wrong order, these messages are dropped.
- The protocol of the *control object* is defined in a similar way (Figure 2).

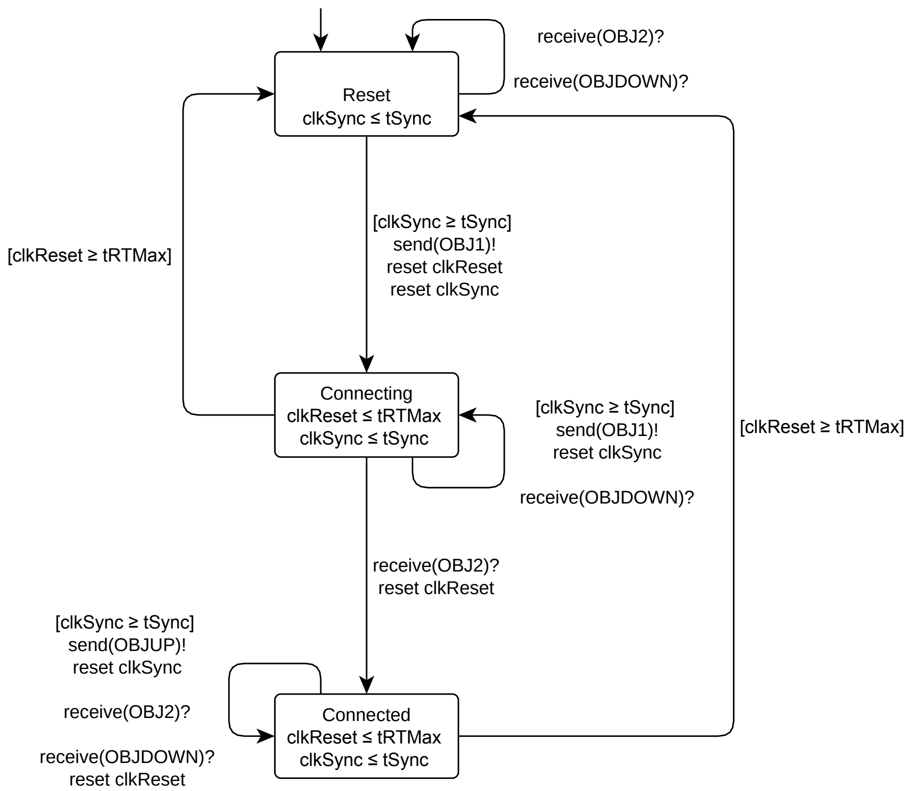

Figure 1: Operation of the field object

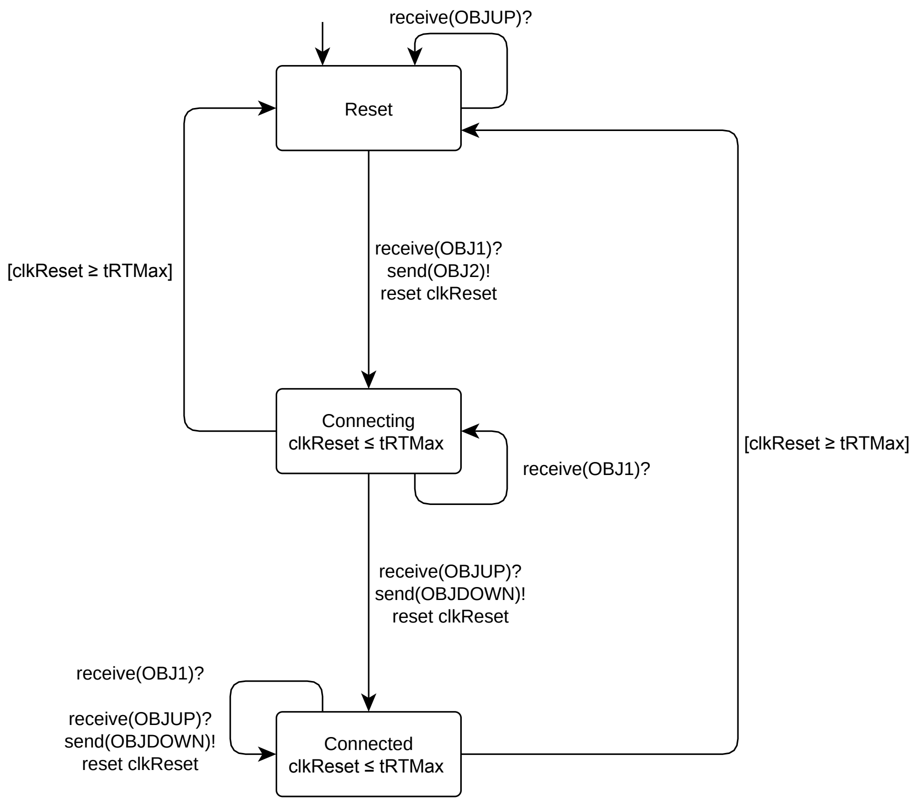

Figure 2: Operation of the control object

Homework assignment

As your homework, select *either Option 1 or Option 2 below* and solve the related tasks. (If you are not familiar with C programming then Option 2 is advised, otherwise Option 1.) Prepare your homework individually *by yourself* (it is not group work); submitted artifacts will be checked w.r.t. plagiarism.

Option 1: Prototype implementation and testing of the protocol

Design and implement in C an *emulation environment* in which the operation of the protocol can be demonstrated and tested on a regular PC.

1. First, implement the protocol of the *field object* as a function in C. When called, this function shall check the incoming message (reading a message buffer) and the occurrence of a timeout (reading the value of timers) and react according to the protocol (sending a message by writing to another message buffer and/or resetting timers). Processing an incoming message shall have priority over processing a timeout if they occur at the same time. Implement the message buffers and the timers as shared (global) data structures that are updated by the environment. There is no need to implement message queues.

   Similarly, implement the *control object* as another function in C.
2. Design and implement the environment for your protocol implementation: Implement a *scheduler* that is able to emulate without OS-specific functionality the transfer of messages (by reading and updating message buffers) and elapse of time (by updating timers). Besides this, the scheduler has to call regularly the above-mentioned functions that implement the field and control objects.
3. Set the time parameters of the objects to values that allow normal operation (i.e., without resetting the connection) in this emulated environment. Demonstrate by executing the emulation and logging the transferred messages that the protocol is able to establish connection and transfer data between the objects.
4. On the basis of Figure 1 (as the protocol specification of the field object) specify a *test suite* that covers *all state transitions of the protocol*. Implement a corresponding *test driver* that executes this test suite. Note that *only the field object* has to be tested; the test driver replaces the scheduler and the control object. Execute the test suite and measure the *branch coverage* on the C implementation of the field object (i.e., on the corresponding function).
5. Submit the following artifacts in a ZIP file (note that incomplete results can be submitted but may result in a lower grade):
   a. The C source code of your implementation. It has to be successfully compiled using up-to-date standard gcc (if you use Linux) or MinGW or Cygwin (if you use Windows 10).
   b. The demonstration (in form of a message log) of the execution of the emulation according to task 3 above (it is sufficient to show the transfer of 4 data messages).
   c. The source code of the test driver.
   d. The branch coverage summary and the detailed branch coverage report provided by gcov for the function implementing the field object.
   e. A short Word document that includes (1) your name, (2) your Neptun code, (3) a declaration that the submitted homework has been prepared by yourself, (4) the description of the design decisions related to the implementation of the scheduler and the test driver (esp. how are message buffers and timers implemented), and (5) explanation in case the branch coverage of the test suite is not 100% (i.e., what are the branches in the implementation that are not covered by the test suite, and why).

Option 2: Modeling and model-based verification of the protocol

Construct the formal model of the protocol and verify that the operation of the protocol satisfies formalized requirements.

1. Study the UPPAAL modeling, simulation and verification tool available at http://uppaal.org/. The tool can be downloaded and installed on Windows or Linux; it is free for academic use. A tutorial is available at the above-mentioned web page or directly at http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf. (Formal modeling and verification using the UPPAAL tool is also presented in the last lecture; the related slides are available at https://www.mit.bme.hu/eng/oktatas/targyak/vimima11/materials)
2. Prepare the timed automata models of the field object and control object in UPPAAL according to the protocol specifications in Figure 1 and Figure 2. Note that the UPPAAL automata models use *synchronous communication* on channels between two partners. Shared variables can be used to represent the content of the message that is relevant for a given synchronization.

   Modeling tip 1: When sending message, first the shared variable representing the message content shall be set on a separate transition, then the synchronization shall follow on the next transition of the automaton. The intermediate state between these transitions shall be a *committed* state.

   Modeling tip 2: Do not forget to set the state invariants according to the protocol specification.
3. Set the time parameters of the objects to values that allow normal operation (i.e., without resetting the connection). Demonstrate by simulation that the protocol is able to establish a connection and transfer data between the objects.
4. Formalize the following properties and verify them. In case of unsuccessful verification, generate the shortest diagnostic trace that is able to demonstrate the violation of the required property.
   a. There is no deadlock in the protocol.
   b. In case of all executions, the control object will eventually reach the *Connecting* state.
   c. It is possible that the field and control objects eventually reach their *Connected* states.
5. Submit the following artifacts in a ZIP file (note that incomplete results can be submitted but may result in a lower grade):
   a. The UPPAAL model that you prepared. The models shall be free from syntax errors using the latest stable version of UPPAAL.
   b. The demonstration of the operation of the protocol (simulation trace saved from UPPAAL) according to task 3 above (it is sufficient to show the transfer of 4 data messages).
   c. The formalized requirements, according to task 4 above.
   d. The results of the verification of the formalized requirements (screenshot if the verification was successful, or the corresponding shortest diagnostic trace if the verification of a formalized property was unsuccessful).
   e. A short Word document that includes (1) your name, (2) your Neptun code, (3) a declaration that the submitted homework has been prepared by yourself, (4) the description of the design decisions related to the modeling of the protocol (esp. about the variables that are used in the models), and (5) explanation in case of unsuccessful verification of a property (i.e., how the diagnostic trace demonstrates the violation of the property).