Lab 1 – Tower Serial Communications

Port access. UART initialization. Polling. Circular buffer. Packet decoding.

Introduction

The Universal Asynchronous Receiver / Transmitter (UART) is a simple yet extremely important peripheral of the K70 microcontroller. On the Tower board it is connected to a serial-to-USB chip that enables the board to communicate to a PC running Windows 10. The Tower board is to implement the serial communication protocol as outlined in the separate document entitled *Tower Serial Communication Protocol*.

Objectives

1. To set up a UART on a microcontroller.
2. To implement a circular buffer in the C language.
3. To decode and respond to packets according to a defined protocol.

Equipment

- 1 TWR-K70F120M-KIT
- 1 USB cable
- NXP Kinetis Design Studio

Safety

This is a Category A laboratory experiment. Please adhere to the Category A safety guidelines (issued separately).

Cat. A lab

L1.2

Software Overview

The serial communication between the Tower board and the PC uses 5-byte packets. The task of the Tower software is to receive and send packets of information asynchronously.

This is a classic producer-consumer problem. One solution to the asynchronous nature of the communication is to implement a first-in first-out (FIFO) buffer between the producer and consumer. This is shown diagrammatically below:

Data flow graph showing two FIFOs that buffer data between producers and consumers

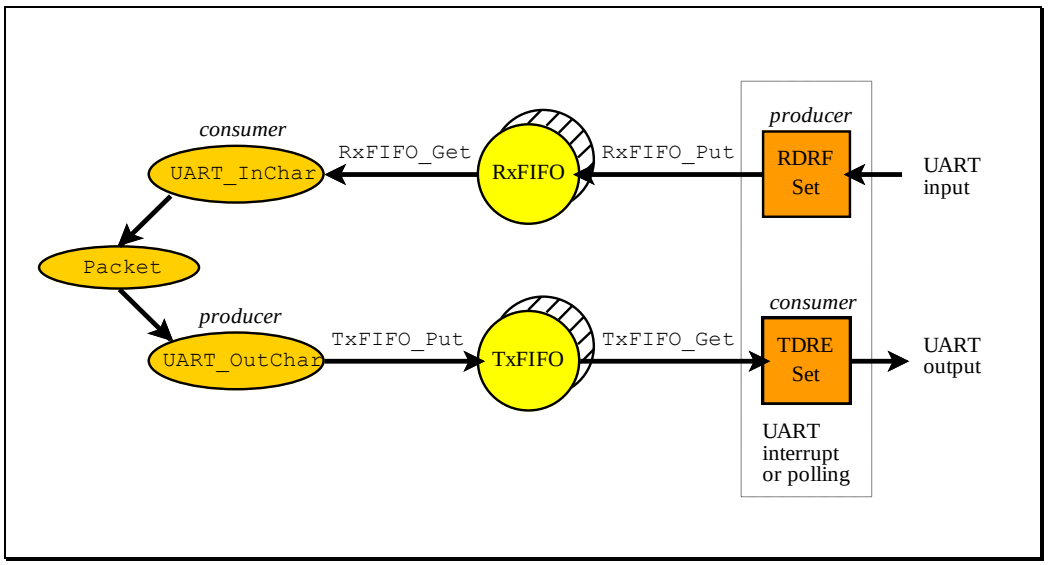


Figure L1.1

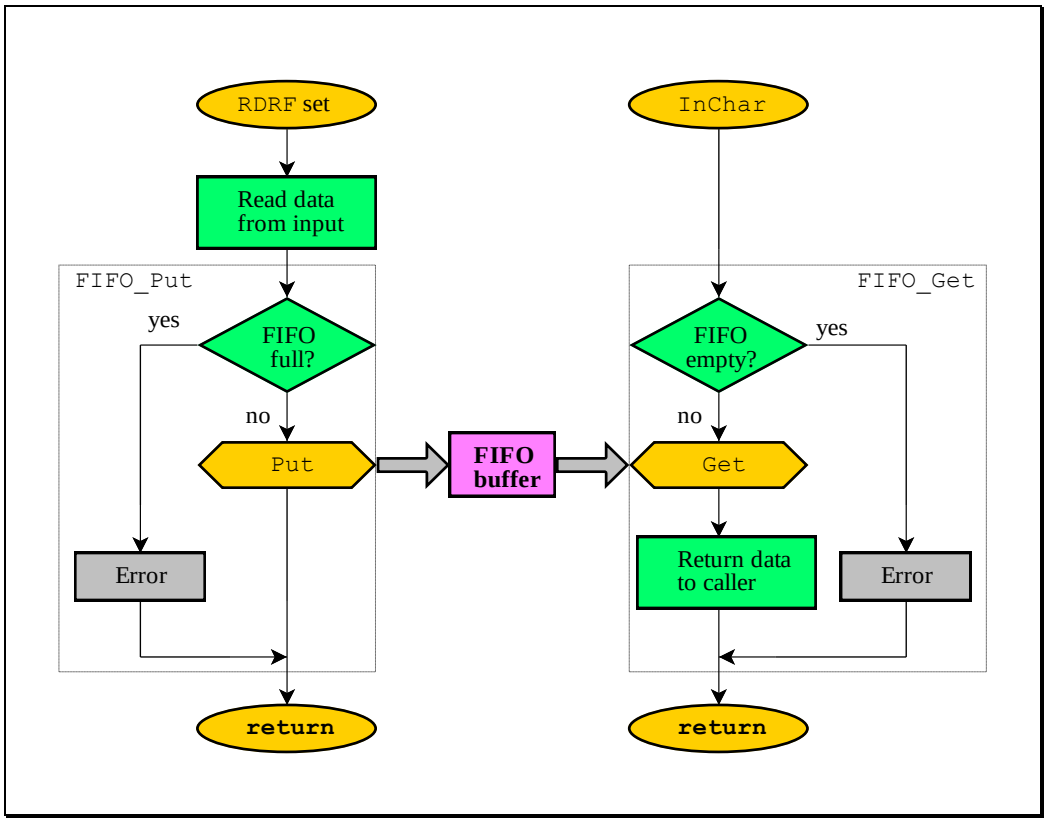
A simple (but inefficient) way of handling the asynchronous nature of the UART communication is to check the status of the UART hardware repeatedly by calling a function in the main loop of the program that *polls* the status of the UART, and calls `RxFIFO_Put` or `TxFIFO_Get` as required.

Receiving Data

When the packet module wishes to receive input, it calls `UART_InChar`, which will attempt to get data from the `RxFIFO` (it will fail if the FIFO buffer is empty). How does data get in the `RxFIFO`?

The incoming serial data will set the Receive Data Register Full (`RDRF`) flag in the UART Status Register 1 (`UART_S1`) register, indicating that the receiver hardware has just received a byte of data. In the main loop, a poll of the `RDRF` flag is performed. If it is set, then the program tries to accept the data and put it in the `RxFIFO`. The `RxFIFO` buffers data between the input hardware and the main program that processes the data. If the `RxFIFO` becomes full, then data will be lost. This is illustrated below:

In this lab, the receive interrupt service routine is replaced by a polling operation



A FIFO queue can be used to pass data between an input device and a main thread

Figure L1.2

FIFO full errors will always occur if the average input rate (number of bytes arriving per second from the input hardware) exceeds the average processing rate (number of bytes processed per second by the main program). In this situation, either the output rate must be increased (by using a faster computer

or by writing a better software processing algorithm), or the input rate must be decreased (by slowing down the arrival rate of data). The second way the `RxFIFO` could become full is if there is a temporary increase in the arrival rate or a temporary decrease in the processing rate. For this situation, the full errors could be eliminated by increasing the size of the `RxFIFO`.

It is inefficient, but not catastrophic, for the main program to wait on an empty `RxFIFO` in some cases. Efficiency can be improved for the buffered input problem by performing other tasks while waiting for data.

Sending Data

When the packet module wishes to output, it calls `UART_OutChar`, which will put the data in the `TxFIFO` and arm the output device.

The setting of the Transmit Data Register Empty (`TDRE`) flag by the UART hardware signals that the output shift register is idle and ready to output more data. In the main loop, a poll of the `TDRE` flag is performed. If it is set, then the program tries to retrieve the data in the `TxFIFO`., and send it out the serial port. If the `TxFIFO` becomes empty, then no data will be sent out the serial port. This is illustrated below:

In this lab, the transmit interrupt service routine is replaced by a polling operation

A FIFO queue can be used to pass data between a main thread and an output device

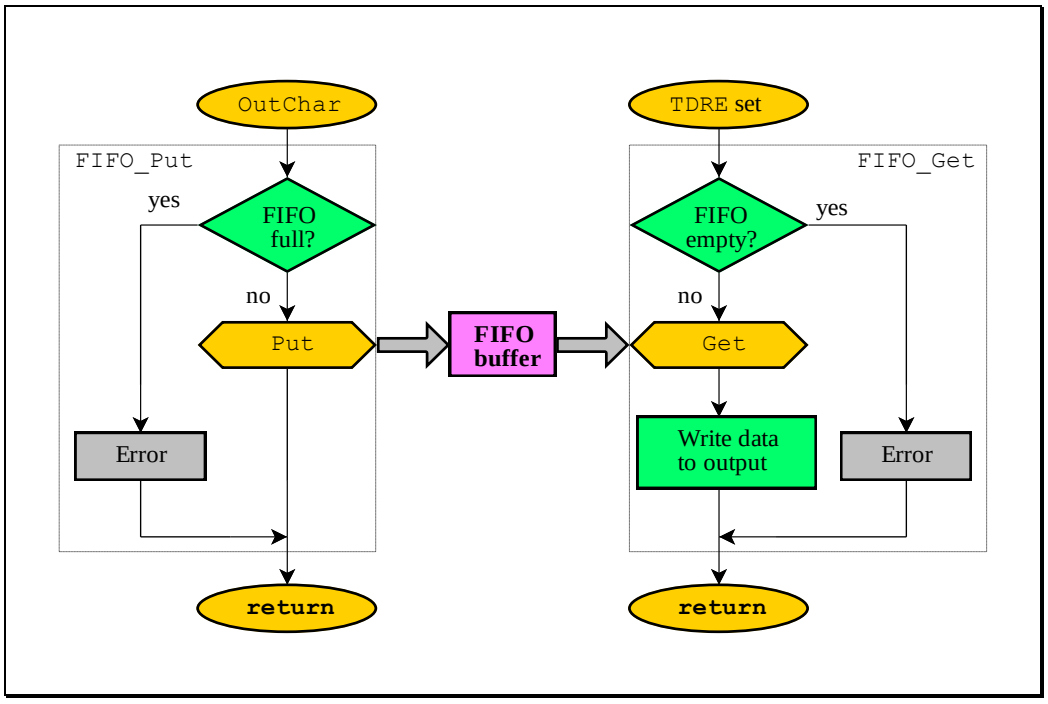


Figure L1.3

It is inefficient, but not catastrophic, for the main program to wait on a full `TxFIFO`. Efficiency can be improved for the buffered output problem by increasing the `TxFIFO` size.

Software Modules

Various modules should be written to support the serial communication protocol, in such a way that modules build upon one another in a hierarchy. The graph below shows the software module dependency – each layer abstracts away the "nuts and bolts" so that at the top of the hierarchy (in `main.c`, which is your application) you can simply call high-level `Packet` module routines without knowing any details of how packets are received and transmitted.

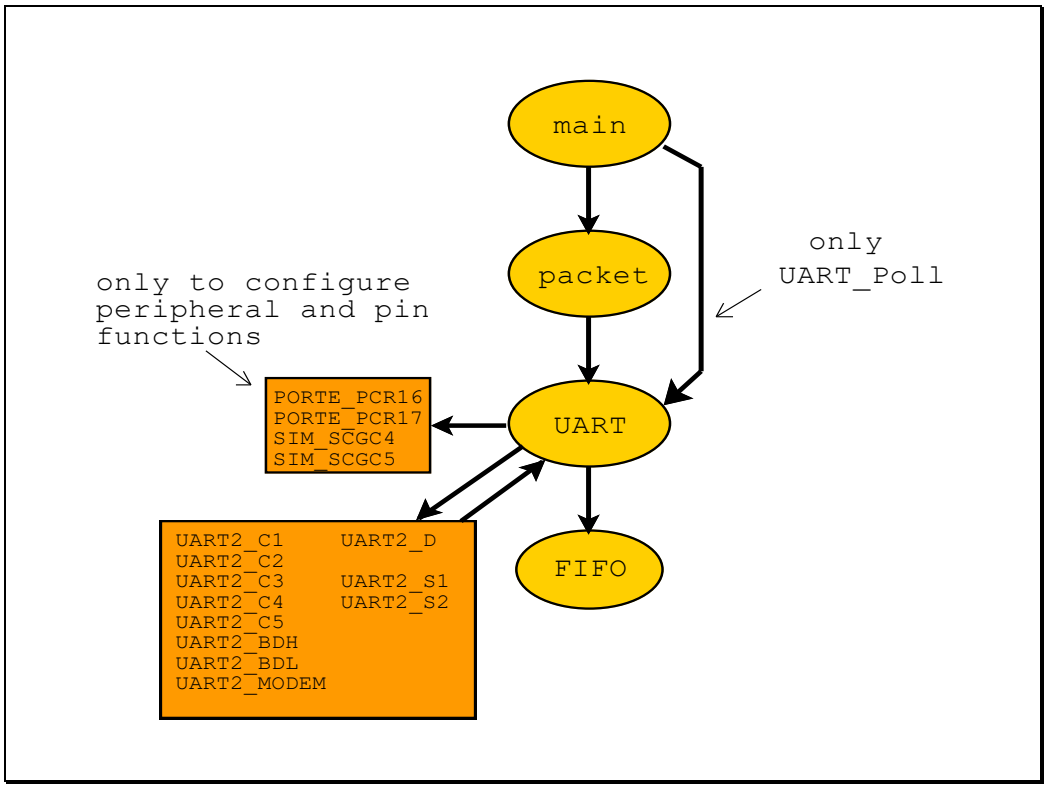


Figure L1.4

The concept of a software module "abstracting away" the hardware is known as a hardware abstraction layer (HAL). As you work up the software module hierarchy (from the bottom hardware layer to the top application layer) the software modules present more abstract (and powerful) functions to the layer above, until you arrive at the penultimate top layer which is, in effect, an application programming interface (API).

L1.6

Labs/Project Workflow

The labs have been designed so that you will progressively build up various low-level modules in preparation to undertake the final project. The following diagram shows a typical timeline of Git commits heading towards this goal:

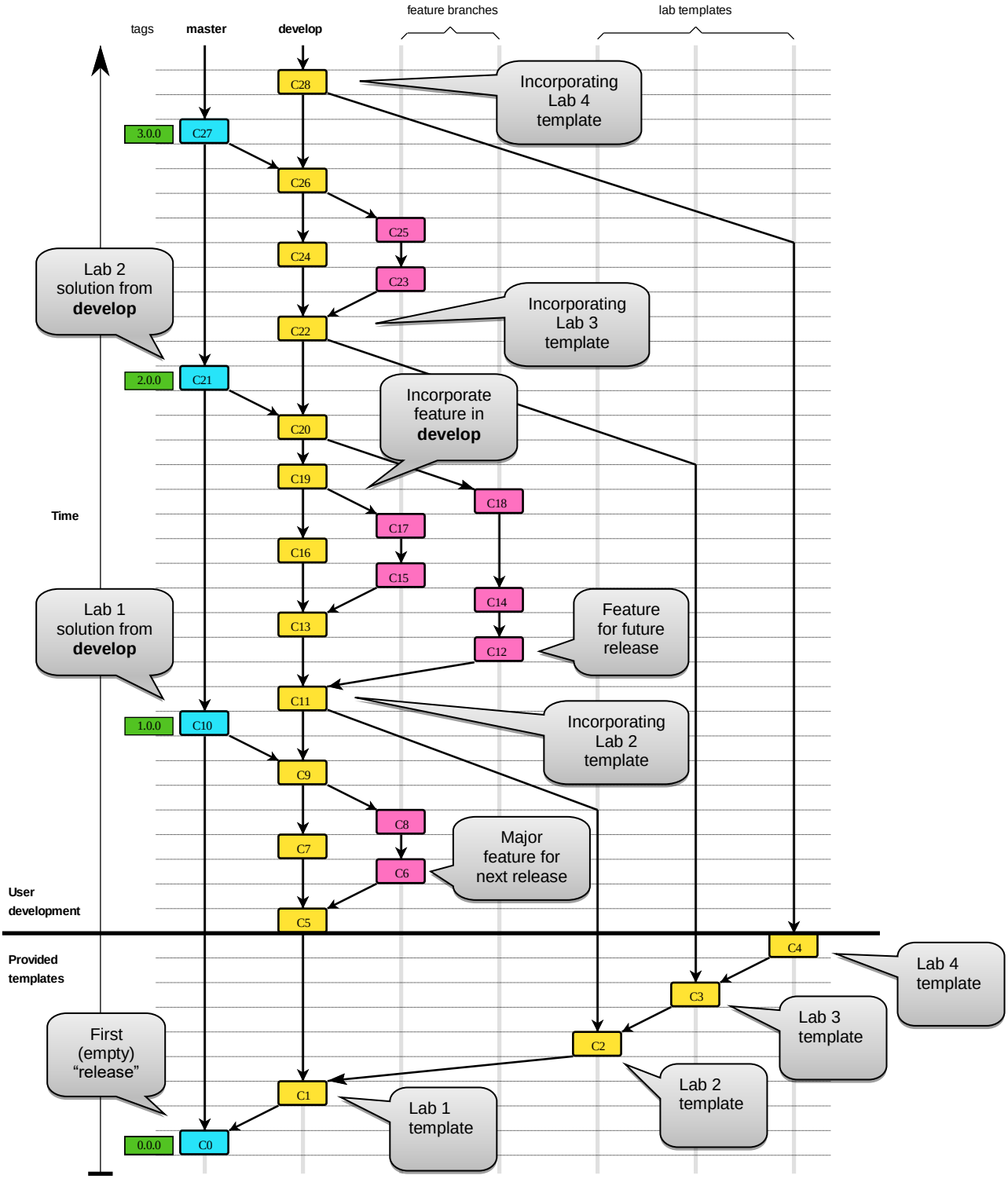


Figure L1.5

Software Requirements

1. You should implement the public functions and variables, as "advertised" in the supplied header files.
2. The serial-to-USB bridge uses UART2. The baud rate is to be 38400 baud, with an 8N1 frame. You are required to set/clear individual bits in the UART's control registers, based on this software requirements specification (SRS).
3. The CPU bus clock frequency can be found in "CPU.h". The CPU bus clock is supplied to UART2 as its "module clock" (see Table 5-2 in K70P256M150SF3RM.pdf). The UART2 module clock is needed for baud rate divisor calculations.
4. A frequent polling operation in the main loop should be used to check the status of the serial port.
5. The commands of the Tower serial protocol to be implemented (with packet acknowledgement) are:

Tower to PC	PC to Tower
0x04 Tower startup	0x04 Special – Get startup values
0x09 Special – Tower version	0x09 Special – Get version
0x0B Tower number	0x0B Tower number (get & set)

6. In response to reception of a "0x04 Special – Get startup values" packet from the PC, the Tower should transmit three packets:
    - a "0x04 Tower startup" packet
    - a "0x09 Special – Tower version" packet
    - a "0x0B Tower number" packet
7. Upon power up, the Tower should send the same 3 packets as above.
8. Use the last 4 digits of your student number to initialise the Tower number. Note that it may be changed via the "0x0B Tower number" packet at a later time.
9. The "0x09 Special – Tower Version" should be V1.0.

10. Git must be used for version control. Note that version control will be assessed in Lab 5 based on the development of the software from Lab 1 through to Lab 5.

    The Project template (Labs 1 to 5 build towards a Project, so the template is called “Project”) can be cloned from your private GitLab central repository at:

    http://git.pmcl.net.au/48434_YYYY_SSS/esXX.git where:

    - **YYYY** is the year, e.g. 2019
    - **SSS** is the first 3 letters of the session, e.g. AUT, SPR, SUM
    - **es** stands for Embedded Software
    - **XX** is the group number, always expressed as two digits

    For example, the project template for Group 3, undertaking the subject in Autumn 2019, can be retrieved from:

    http://git.pmcl.net.au/48434_2019_AUT/es03.git

Hints

1. Together, a `.h` and `.c` file form a "module". Recall that the header file (`.h`) "advertises" the capabilities of the module by "exposing" or "making available" public functions and variables. The implementation of the public functions and variables occurs in the C file (`.c`), along with any other private "helper" functions (which are **not** made public in the header file).

2. You need to enable the UART module in `SIM_SCGC4`.

3. To enable pin routing for Port E, you need to enable Port E in `SIM_SCGC5`.

4. For PIN multiplexing, see p. 280 of K70P256M150SF3RM.pdf. UART2 shares pins with PortE bits 16 and 17.

5. For the UART control registers, you only need to write code to change individual bits from their reset state, if required by the SRS (in other words, your software may assume that a valid reset has taken place before it is executed). For every bit that is set or cleared, there should be one line of code, with appropriate comments.

6. The Baud Rate Divisor Register and Fine Adjust are discussed on p. 1974 of K70P256M150SF3RM.pdf.

7. The project has automatically included MK70F12.h which defines all the registers, such as `UART2_S1`, as well as the bit masks, such as `UART_S1_RDRF_MASK`.

8. Valid commands also require valid parameters when handling a packet.

L1.10

Marking

1. The software to be assessed must reside in the *remote* GitLab repository by the time and date specified in the Timetable in the Learning Guide. Create a merged commit from the `develop` branch to the `master` branch (remember to use the `--no-ff` flag) and tag the commit with `1.0.0`.
2. During marking, you may create a `hotfix-1.0.1` branch in order to develop a TODO list.
3. Software marking will be carried out in the laboratory, in the format of a code review.
4. Refer to the document “Software Style Guide” for more details of some of the assessment criteria.

Assessment Criteria

Your lab will be assessed according to the following criteria:

Item	Detail	Evaluation	Mark
Opening comments	File headers are correct.	**E G A P N**	**/0.5**
Function descriptions	Function descriptions are appropriate and correct.	**E G A P N**	**/0.5**
Naming conventions	Names conform to the Software Style Guide.	**E G A P N**	**/0.5**
Code structure	Code structure conforms to the Software Style Guide.	**E G A P N**	**/0.5**
UART functions	All control register bits are set individually. Baud rate set correctly. Correct use of FIFOs. Polling is correct.	**E G A P N**	**/1**
Packet functions	Packet functionality is correct. Errors are handled correctly.	**E G A P N**	**/1**
FIFO implementation	Functionally correct, easy to modify (read) and efficient.	**E G A P N**	**/2**
Protocol implementation	Protocol response is correct, including ACK/NAK.	**E G A P N**	**/2**
		TOTAL	**/8**

L1.12

When we evaluate an assessment item, we will use the following criteria:

Evaluation	Mark (%)	Description
Excellent	100	**All relevant material is presented in a logical manner showing clear understanding, and sound reasoning.** For software – correct coding style, correct software architecture including: modularity; functions; parameters; and types, very efficient implementation (code and time) and/or novel (and correct) code.
Good	75	**Nearly all relevant material is presented with good organisation and understanding.** For software – mostly correct coding style, mostly correct software architecture including: modularity; functions; parameters; and types, reasonably efficient implementation (code and time).
Acceptable	50	**Most relevant material is presented with acceptable organisation and understanding.** For software – inconsistent coding style, reasonable software architecture (but could show improvement in modularity, use of functions, parameters, or types), some code may be prone to errors under certain operating conditions (e.g. input parameters) or usage, occasional inefficient or incorrect code.
Poor	25	**Little relevant material is presented and/or poor organisation or understanding.** For software – Conceptual difficulty of the underlying concepts, numerous coding style errors, functionality missing, poor software architecture, inappropriate or incorrect use of functions, parameters or types. Very inefficient and / or incorrect code.
No attempt	0	**No attempt.** For software – missing modules and/or functionality.

Oral Defence

During the assessment of your work you will be asked questions based on material which you have learnt in the subject and then used to implement the assessment task. You are expected to know exactly how your implementation works and be able to justify the design choices which you have made. If you fail to answer the questions with appropriate substance then you will be awarded **zero** for that component.