# LAB MODULE #2: ADVANCED ASSEMBLY PROGRAMMING of EMBEDDED SYSTEMS using SUBROUTINES, STACKS and EXPANDED MEMORY

## 2.1 OBJECTIVE

The purpose of the second laboratory module is the development of a more complex embedded system with advanced assembly programming and use of data structures like stacks. The following four features will be added to the embedded system developed in Module 1:

i. A simple user interface to start/stop system operation, and get read-out of the previously logged data;
ii. improved messaging-based data communication protocol to remove the pin count limits on the total size of communicated information, and hence addition of battery life to the list of logged parameters,
iii. data integrity checks through CRC error detection, and
iv. expanded memory capacity with external RAM.

You will plan and develop your control code in modules, taking advantage of subroutines for organization. You will then debug and simulate the code first on Microchip Studio, followed by embedded system on Proteus, paying attention to different operation modes and corner cases. Design and simulation results should be well documented in your submitted report before your scheduled laboratory session. You will then be prepared to quickly and effectively demonstrate your development and verification process to the lab instructor by sharing the Microchip Studio and Proteus sessions directly from your computer, and answering any questions. You need to be ready to create various scenarios by entering different operational system modes upon request.

## 2.2 BACKGROUND

### 2.2.1 Error Detection and Correction Techniques for Reliable Data Transmission & Storage

Error detection and correction techniques are commonly used in reliable computer and telecommunication systems for preserving the quality of data storage or transmission over wired or wireless channels. Communication over wired or wireless connections, for example, is subject to attenuation and noise. Most commonly used error detection schemes are repetition codes, parity bits, checksums, cyclic redundancy checks (CRC). CRC is particularly effective, and is common in wireless sensor networks subject to "burst" noise events that may corrupt multiple bits in a transmitted message. CRC error detection capability will be added to the data transmissions in your embedded system. Instead of error correction, the system will request retransmission of corrupted messages.

***Cyclic Redundancy Check (CRC)***

Cyclic Redundancy Check (CRC) is a verification method that is used to ensure that data being sent is not corrupted during transfer over communication channels. CRC codes are polynomial functions that can be represented in binary e.g. $x^5+x^3+x^2+x^0= (101101)_2$. The CRC algorithm, first proposed by W. Wesley Peterson (Brown, 1961), computes a CRC code for each set of transmitted data. Systematic cyclic codes are used to encode the message by adding a fixed-length check value, which is calculated with the use of a polynomial key. An *n*-bit CRC applied to a data block of arbitrary length will detect any single error burst not longer than *n* bits and will detect a fraction $1 - 2^{-n}$ of all longer error bursts. CRC can detect more errors than simple parity bit and checksum. The basic description of CRC procedure can be summarized as follows:

1. Transmitter and receiver both agree on a generator polynomial (key), ***G*** for CRC calculations.
2. Transmitter:
   - Performs modulo-2 division of the original message to be transferred by a generator polynomial ***G*** represented as a binary number (modulo 2 division means subtraction in the regular division is replaced by bitwise XOR operation), and
   - attaches the remainder of the division to the original message as the CRC code.
3. Receiver:
   - Performs the same division on the received message using the same generator polynomial ***G***, and compares the resulting remainder with the received remainder.

If a transmitter wants to send a message ***D*** with ***d*** data bits, a CRC code ***R*** with ***r*** bits of length is generated through modulo-2 division. Thus, the remainder is calculated through:

$$\boldsymbol{R = remainder\frac{(D*2^r)}{G}}. \quad (1)$$

The remainder bits are then appended to the original data for the receiver to check, as outlined above. The multiplication in Equation 1 can be achieved by padding ***r*** zeros to the message before doing the modulo-2 division by ***G***. An example for the CRC code calculation is provided below.

***Example:***

Consider the following 2-Byte data string (message) ***D***, and generator polynomial ***G***:

***D*** = 1101001110110000 ***G*** = $x^3 + x + 1$ (in binary this corresponds to 1011)

Since the maximum number of bits in the remainder after division by a 4-bit divisor polynomial ***G*** is ***r***=3, three 0's are first appended to ***D*** before the modulo-2 division. Detailed steps of the modulo-2 division are shown below:

```
1101001110110000 000   | 1011
1011                   111100011111001
01100
 1011
 01110
  1011
  01011
   1011
   00001101
       1011
       01101
        1011
        01100
         1011
         01110
          1011
          01010
           1011
           0001000
              1011
              011
```

← 3-bit remainder to be appended to 2 bytes of data as **CRC code**

### 2.2.2 External Memory (Memory Expansion)

General purpose computing often requires vast storage resources that start with on-chip SRAM memories of few Megabytes, and extend to Terabyte scale hard disk drives. Although embedded computers have much less storage requirement in most applications, more storage than what is available within the MCU is sometimes desirable. In such cases, external memory is added to the system bus as extension to internal memory. It is important to have a basic understanding of fundamental logic blocks and signaling requirements of MCU and Memory Chips to properly integrate external memory to an embedded system.

#### 2.2.2.1 Three State (Tri-state) Outputs as Requirement for a Common System Bus

MCU, external memory, and other devices can be connected to the common system bus, as long as the output pins can be tristated for each device. The high-impedance state of such pins is activated when the device is not enabled for output (e.g. when Chip-Select is OFF or when the memory is in Write mode). In high impedance state output buffers function as if they are not electrically connected to the bus. When several tri-state outputs are connected to the same physical wire or board trace, all but one must be disabled; only the enabled output determines the logic level of the signal. **If the outputs of two tri-state devices are enabled simultaneously the devices are subject to damage due to short-circuit currents.** The logic symbols for four possible variations of the tri-state output buffer are shown in Figure 2.1. Many devices such as stand-alone memory chips are designed with tri-state outputs so that they can be

multiplexed easily to a bus. For devices, which do not have tri-state outputs, separate tristate buffer ICs between the output of the device and the bus carry out the same function.

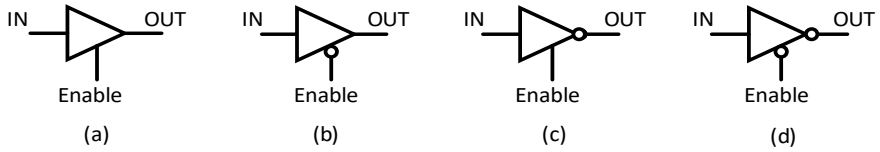


Figure 2.1. Different variations of tri-state buffers: (a) Non-inverting buffer with active high enable, (b) Non-inverting buffer with active low enable, (c) Inverting buffer with active high enable, (d) Inverting buffer with active low enable.

#### 2.2.2.2 Semiconductor Memories

A semiconductor memory device has to meet all the functional requirements of a memory system. You will be using 6116 Static RAM in this exercise to obtain an additional 2048 Bytes (2 kB) data memory space as memory expansion in your system. As any other semiconductor device 6116 contains:

i. An array of memory cells, each of which can store a single bit (top right block in Fig. 2.2);
ii. Address Decoder to select any location in the memory based on input Address bus (top left in Fig. 2.2),
iii. Read logic to allow outputting the contents of any memory location to 8-bit I/O Data bus (bottom right Output Control and tri-state buffers in Fig. 2.2),
v. For writeable memories, Write circuitry to allow any memory location to be written by the value on the I/O Data bus (Input Data Circuit and input buffers at bottom left of Fig. 2.2),
vi. Control Circuit to ensure Read/Write/Standby modes work consistently (bottom of Fig. 2.2).

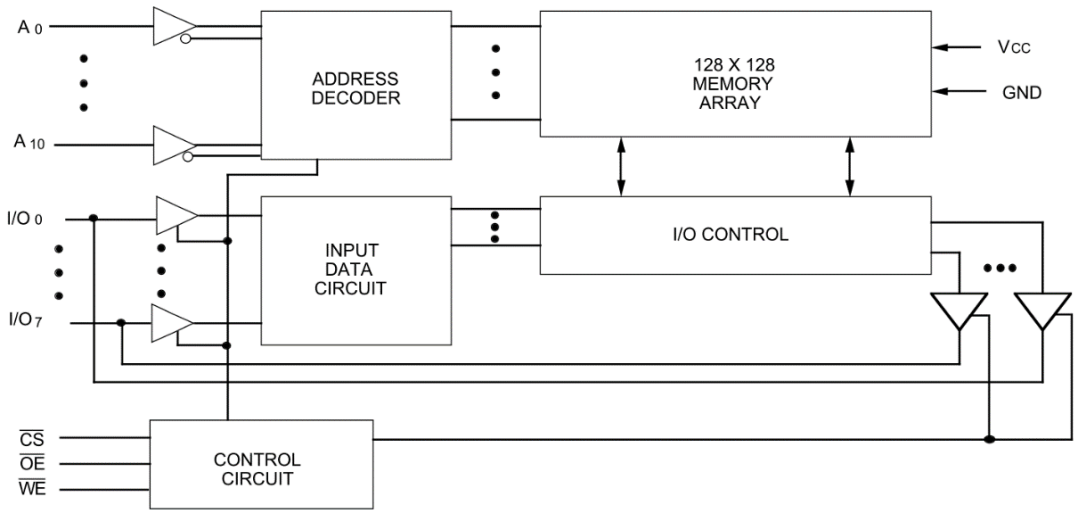


Figure 2.2. Functional block diagram of 6116 Static RAM to be used for memory expansion.

To ensure proper operation there are timing constraints on the sequencing of address, data, and control signals of a memory. Read and Write diagrams of the 6116 Static RAM are provided Figure 2.3, and are very similar to other RAM components. The basic sequence of operations for writing to or reading a random-access type memory is as follows:

1. An address is applied to the address inputs of the memory.
2. The chip is selected by application of proper logic level(s) at its chip select input(s).
3. Data to be written into the memory is applied at the data inputs for the write cycle. Then the R/W line is pulsed low.
4. For the read cycle, the R/W line is pulsed high and then the contents of the selected memory location appear at the data outputs of the memory after a period of time equal to its access time.
5. The address and chip enable signals can then be changed to select another memory location.

Detailed specifications and timing requirements of 6116 Static RAM can be obtained from the datasheet.

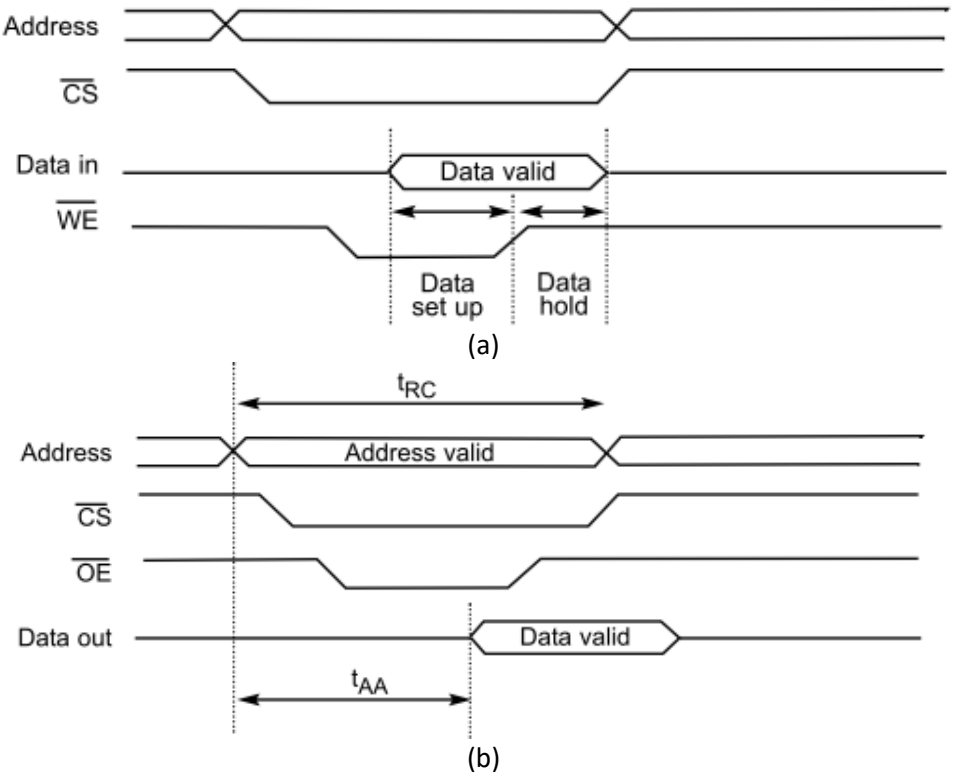


Figure 2.3. (a) WRITE timing diagram for a R/W memory and (b) READ timing diagram for a R/W memory.

#### 2.2.2.3 ATMega128 Address Space Organization and External Data Memory Interface

ATMega128 external data memory space starts at address (0x1100) following the last address (0x10FF) of the internal SRAM. Register file, I/O, Extended I/O and Internal SRAM occupies the lowest 4352 Bytes in normal mode so when using 64 kByte (65536 Bytes) of External Memory, 61184 Bytes of this external memory is actually available. Accessing external data memory takes one additional clock cycle per byte compared to access of the internal SRAM. This means that the commands LD, ST, LDS, STS, LDD, STD, PUSH, and POP take one additional clock cycle. One should be aware that this will increase the number of clock cycles required to execute the program that needs access to data memory. Consider the problem of physically assigning a 6116 SRAM to a 2K block in the MCU address space. The proper chip select signal for the 6116 needs to be generated so that the MCU can correctly access the chosen 2K block. In order to use external SRAM, one should activate the XMEM interface of the AVR processor and configure it as in Figure 2.4 according to the SRAM's specifications. When XMEM mode is activated, XMEM interface will override the setting in the data direction registers for the concerned MCU pins which are dedicated to the XMEM interface. In other words, XMEM pins will enter their alternate mode of operation.

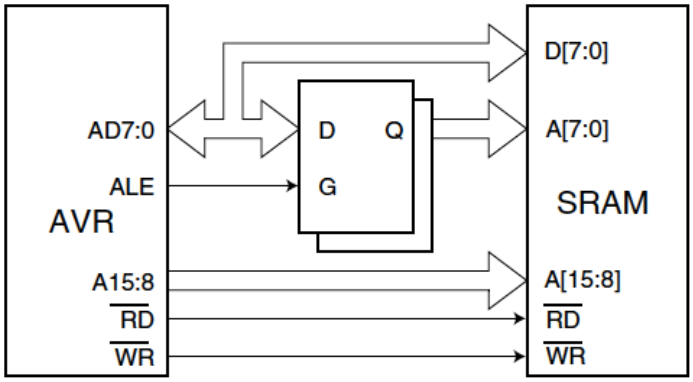


Figure 2.4. AVR data memory expansion through external SRAM connection (ATMega128 datasheet).

## 2.3 PROBLEM DESCRIPTION

### 2.3.1 *Revised System Functions and Interfaces*

As the *smart data logger system* is revised in this phase of the project, a simple user interface, a messaging-based data communication protocol, and 2 kB of external memory have been added. As depicted in Figure 2.5, the data communication interface is modeled using parallel ports in this revision, and is based on received and transmitted 1-Byte message packets. Similarly, the user interface uses separate parallel ports for entering controls and getting readouts. In a future phase of the project, fully serial links will most likely replace the parallel interfaces to favor further cost optimization. Although details of the external SRAM interface are not shown in Figure 2.5, these details can be found in Figure 2.4 and AVR XMEM specifications.

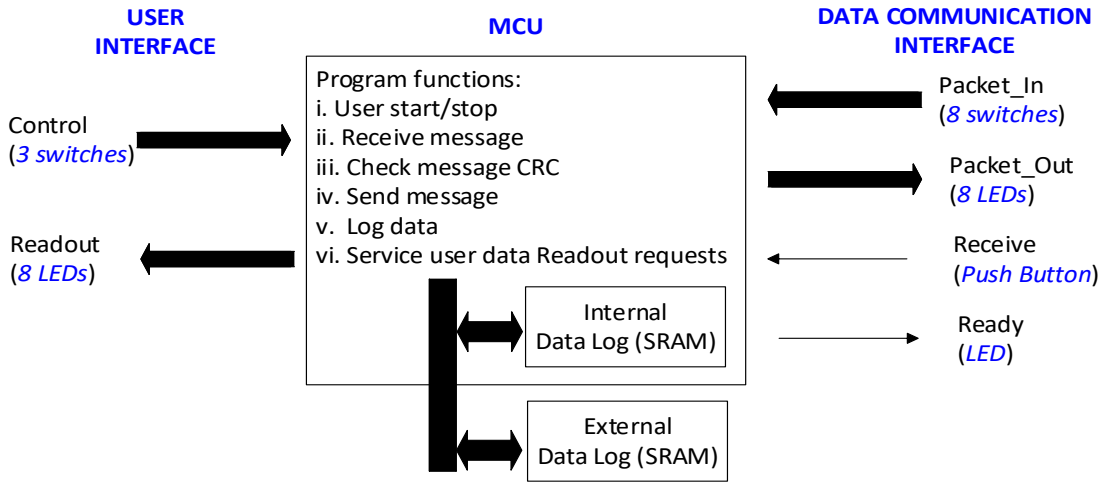


Figure 2.5. Main functions and interfaces of the improved smart data logger.

User and data communication interfaces, shown on the left and right side of Figure 2.5 respectively, are further defined in Table 2.1.

Table 2.1. User and data communication interface description of the revised embedded system

| Signal Name | Interface Type | Description |
|---|---|---|
| Start/Stop | User Control Input | Starts (1) and stops (0) the system (rest of the program functions) |
| Memory Dump | User Control Input | Command to Readout all internal/external SRAM contents one Byte at a time (Receive requests are not serviced during memory dump) |
| Last Entry | User Control Input | Command to Readout last data Byte logged in memory (if Memory Dump input is active, Last Entry input is ignored) |
| Readout [7:0] | User Data Output | Readout data that is output as a result of Memory Dump or Last Entry commands |
| Receive | Data Comm. Input | Control signal to trigger Input Packet (Packet_In) capture. For now, this signal will be treated as a push button that has to be pressed and released before the smart datalogger captures any data. |
| Ready | Data Comm. Output | Status signal that is turned OFF while Packet_In is processed by the MCU; turns back ON when the Packet_In is processed. |
| Packet_In [7:0] | Data Comm. Input | Arriving message packet to be captured when Receive is ON |
| Packet_Out [7:0] | Data Comm. Output | Message packet to be sent to the remote sensor as part of the packet-based communication protocol |

### 2.3.2 *Messaging Based Data Communication Protocol*

The message packets exchanged with remote sensors through the data communication interface have one of the following two formats:

Bit 7 — Message Packet Type 1 (***Command*** Packet): — Bit 0

| **0** | 2-bit Command | CRC-5-ITU Code with Generator polynomial $x^5+x^4+x^2+1$ |
|---|---|---|

Bit 7 — Message Packet Type 2 (***Data*** Packet): — Bit 0

| **1** | 2-bit Parameter ID | 5-bit Parameter Value |
|---|---|---|

Figure 2.6. Two alternative 1-Byte message packet formats for data communications.

Whenever a ***Command*** packet is received (Packet_In) or transmitted (Packet_Out), 2-bit command field (bits 6:5 in the ***Command*** packet) takes on one of the values described in Table 2.2.

Table 2.2. Decoding of Command field in ***Command*** Packet

| Command Packet [6:5] | Description |
|---|---|
| 00 | **Reset Request:** Can only be configured in a Packet_Out message by the smart data logger to request remote sensor to reset its operation and restart. Reset Request is also configured in the first Packet_Out after the system is started by the user. |
| 01 | **Log Request:** Can only be configured in a Packet_In message by remote sensor node to tell the smart data logger that previous Packet_In was a ***Data*** Packet. |
| 10 | **Acknowledge:** Can be configured in a Packet_In message (by the remote sensor) in response to a Reset Request or Packet_Out message (by the smart data logger) in response to a Log Request. Every Request ***Command*** packet should be responded by an Acknowledge ***Command*** packet as part of the communication handshake protocol between smart data logger and remote sensor. Otherwise a new request cannot be sent by either agent. |
| 11 | **Error/Repeat:** Can be configured in a Packet_In message (by the remote sensor) in response to the last Packet_Out or Packet_Out message (by the smart data logger) in response to the last Packet_In. A **Repeat** *Command* packet sent by either agent on the communication line tells the other agent the last packet failed the CRC check and should be sent again. |

Whenever a ***Data*** packet is received (Packet_In), 2-bit Parameter ID (bits 6:5 in the ***Data*** packet) takes on one of the values described in Table 2.3.

Table 2.3. Decoding of Parameter ID field in ***Data*** Packet

| Data Packet [6:5] | Description |
|---|---|
| 00 | Temperature |
| 01 | Moisture |
| 10 | Water Level |
| 11 | Battery Level |

When a ***Command*** packet is stand-alone (either a Packet_Out or a Packet_In which is not accompanied by a data packet), the 5-bit CRC code in the ***Command*** packet only applies to the rest of the three bits in the *Command* packet. When a ***Log Request Command*** packet arrives to the smart logger **after** a ***Data*** packet (two consecutive Packet_In messages), then the 5-bit CRC code in the ***Command*** packet applies to the 11 bits in the last ***Command*** packet and the preceding ***Data*** packet together. Therefore, if a CRC failure happens for a Log Request, both ***Data*** and ***Command*** packets need to be resent by remote sensor.

### 2.3.3 *Putting It All Together*

As you have probably already concluded, the smart logger in lab Module 2 is significantly more complex than the simple logger modeled in Module 1. Planning/organization and structure in design are good approaches to deal with complexity. Figure 2.6 depicts a sample flowchart you may use as a guide in implementing your system.

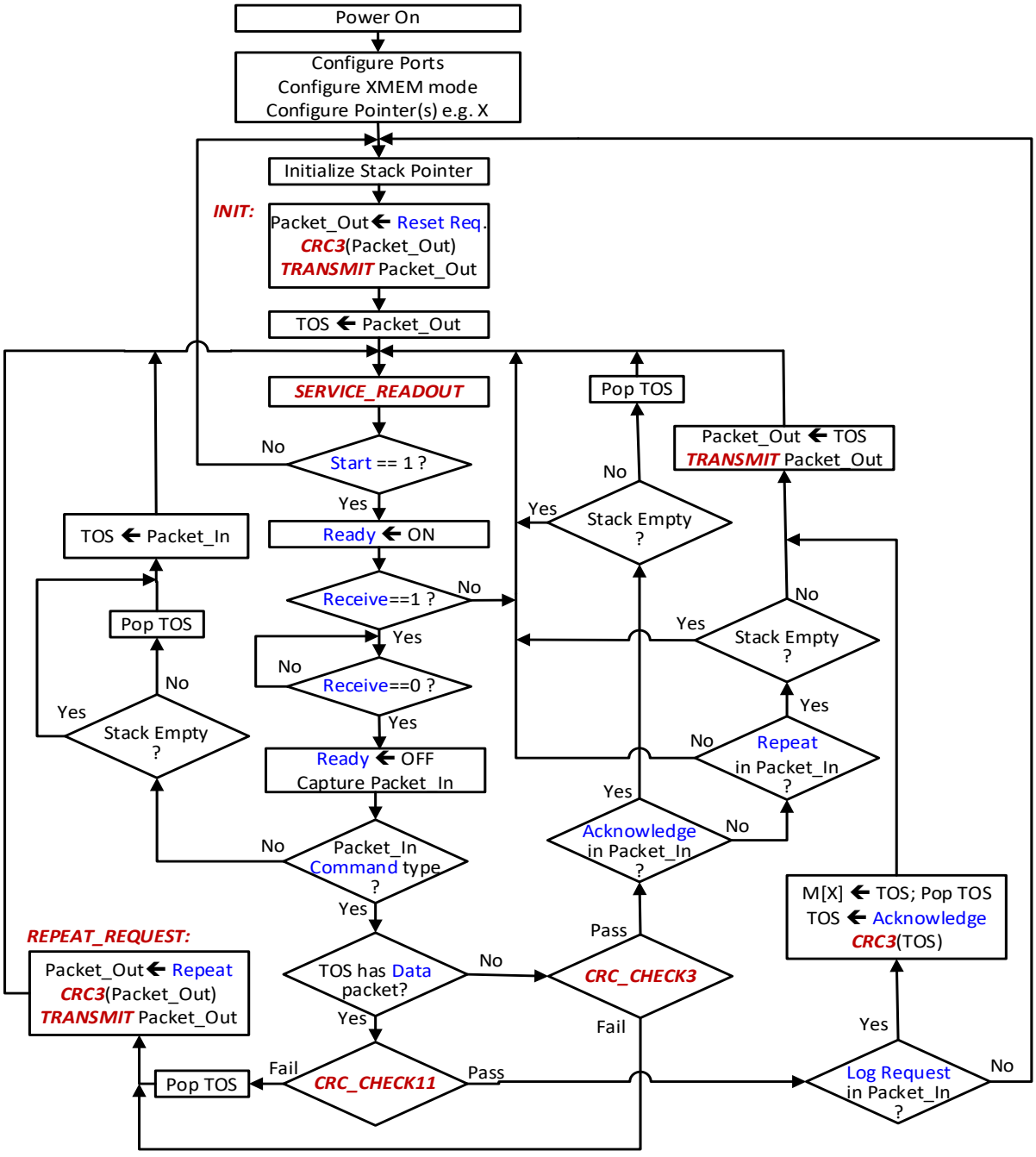


Figure 2.7. Sample flowchart for the implementation of the smart logger embedded system (subroutine calls have been highlighted in ***THIS FONT***.)

A stack data structure is used to record important arriving messages or messages getting transmitted. TOS in the figure denotes Top-of-Stack. Note the user Readout requests (either Memory Dump or Last Entry only if Memory Dump is OFF) can be serviced even if the system has not started yet. Last Entry Readout would output the last record stored at address *X*. If data communication has not happened since system power-on, then the Readout will not be meaningful. Memory logger works in round-robin fashion,

similar to lab Module 1, i.e. when both internal and external RAMs are filled, the next record continues at internal address 0x0000. Keep in mind the stack is stored in the data memory. Therefore, it is important not to add any of the logged data to the stack space. Reserve 20 Bytes at the end of the internal data RAM to be used by the stack and make sure data logger skips this area of the memory when logging data. You may write subroutines for correcting your memory data pointer (e.g. X register) whenever your data storage conflicts with the stack space.

## 2.4 DESIGN AND REPORTING

### 2.4.1 Preliminary Questions

Answer the following questions to enhance your understanding of the described system.

a) If these are the messages exchanged through the data communication interface of the described system in Section 2.3, determine if there is a CRC-5-ITU error for each case i - v, showing the details of your work. Use the generator polynomial in Figure 2.6. If there is no CRC error, explain the information contained in the provided message (or messages) using one sentence for each.
   i. Packet = 0x00
   ii. Packet = 0x5F
   iii. Packet = 0x6B
   iv. Packet = 0xA6 followed by Packet = 0x25
   v. Packet = 0xF2 followed by Packet = 0x26

b) Find information about following types of random-access memories, define them in 1 or 2 sentences, and comment on their application area. Stress distinctive features.
   i. ROM
   ii. SRAM
   iii. DRAM
   iv. SDRAM
   v. DDR3 SDRAM
   vi. FLASH

   Review the 6116 RAM specifications provided with this lab module. What type of RAM (pick from i - vi) is 6116? Explain.

c) Carefully explain why the 8-bit latch-based register in Figure 2.4 is needed in ATMega128 external memory interface? What would happen if this latch is excluded from the design?

### 2.4.2 Design

Here are the steps you should follow in the design process:

a) Complete the design of the external memory interface. 6116 datasheet will be posted to ODTUCLASS. Required latches for the 6116 A[7:0] inputs can be implemented using 74LS373 off-the shelf 8-bit latch component in your system. Any memory address decoding for 6116 can be implemented using 74LS00 / 74LS10 / 74LS30 NAND components with different number of inputs, as well as 74LS04 hex inverter component. First you need to consider if you need address decoding in this system for correct functionality.

b) Review ATmega128 I/O to identify how the required embedded system interfaces will map to the MCU pins.

c) Sketch the system layout based on (a) and (b) to illustrate all signal and component connectivity. This should provide a good idea about how you will draw the schematic in Proteus, and which ports will be used by your assembly program for different I/Os.

d) Study the provided system specifications in the previous sections, as well as the flowchart in Figure 2.7. Do not feel limited by this flowchart. There may be some ambiguities in the specifications, as in

any embedded system problem description. If the detail you need to complete the design is not clear in the descriptions, make an intuitive assumption on desired behavior, write it down, and complete your AVR assembly code. Implement common set of repeated functions as subroutines in your program, following a modular approach. For example, the following should be done by calling subroutines:

i. ***CRC3***: Calculates CRC-5-ITU code ***R*** for three bits in a one-byte Packet_Out message and places it into the proper CRC bits of the same packet.
ii. ***CRC_CHECK3***: Calculates CRC-5-ITU code ***R*** for three bits in a one-byte Packet_In message and compares this code against the code in the packet to return a Pass or Fail.
iii. ***CRC_CHECK11***: Calculates CRC-5-ITU code ***R*** for eleven bits in a two-byte (both the command packet that just arrived and the data packet on TOS) to return a Pass or Fail.
iv. ***INIT***: Transmits a Reset request in a Packet_Out to reset the remote sensor state.
v. ***SERVICE_READOUT***: Checks user Readout request inputs for Memory Dump (higher priority) or Last Entry (lower priority) and services the request if the corresponding input is ON. Since memory dump goes out on LED outputs, you may add a DELAY subroutine call after outputting the content of each memory address to make it easier for the human eye to follow the changes on LEDs. You may target about 1 second. Some delay examples have been covered in lectures.
vi. ***TRANSMIT***: Properly drives the required system outputs to transmit one byte of Packet_Out.
vii. ***REPEAT_REQUEST***: Transmits a CRC encoded Repeat request in a Packet_Out.

There could be other subroutines you may find useful to improve the organization and readability of your assembly code. In addition, note how some of the subroutines which are listed above, and which are highlighted with the same color and font in Figure 2.7, may be called by other subroutines (in a nested manner). Explain which subroutines in your code call others, and which are "**leaves**".

### 2.4.3 Verification

a) Use Microchip Studio to create Module2_MicrochipStudio project. Debug your code, entering input (port) pin values through the I/O window, and making sure that registers, memory locations, and output (port) pin values are correct. It is a good practice to cover different scenarios and corner cases during debug and verification. For example, take screen captures (PrtScr) to illustrate in your report four different command modes, logging data to internal RAM, and doing user readout of internal RAM.

b) Use Proteus to create Module2_Proteus project. Instantiate ATMEGA128 microcontroller, 6116 memory, any latch, NAND, NOT IC components you used in your design for external memory interface, and as many DIPSW_8 (8-bit dip switch), LED-BIRG (green led), 4k7 (4.7 kΩ resistor), and RX8 (pack of 8 resistors), push-button (e.g. SPST push-button in the library) components as you need in order to design the full system. Wire the system carefully. Load your debugged object code, and run different cases, especially demonstrating correct use (read/write) of external memory with different operation modes, without violating stack space in internal RAM. Take screen captures (PrtScr) again for the critical cases to include in your report.

### 2.4.4 Report

Follow the strict guidelines and format described in detail in the first lab manual to complete a concise and comprehensive report. Your report should represent your team's work only, and should clearly document your solutions to preliminary exercises (2.4.1), your complete design layouts (2.4.2), MCU and full embedded system critical verification results along with Proteus schematics. Each simulation screen you include should be carefully annotated and explained in a paragraph. If there are any problematic cases that do not function correctly, these should be discussed.