1 Code

```
using System.Diagnostics;

const int PM_SIZE = 50000;
const int RAM_SIZE = 65535;

Dictionary<string, OpCode> OPCodes = new Dictionary<string, OpCode>()
{
    // special operations
    { "NOP",  new OpCode{ byteCode=0b00000000, param=Param.none }},
    { "END",  new OpCode{ byteCode=0b11111111, param=Param.none }},

    // data transfer
    { "MOV",  new OpCode{ byteCode=0b00100000, param=Param.reg_reg  }},
    { "LD",   new OpCode{ byteCode=0b00100010, param=Param.reg_ram  }},
    { "LDI",  new OpCode{ byteCode=0b00100100, param=Param.reg_imdt }},
    { "ILD",  new OpCode{ byteCode=0b00100110, param=Param.reg      }},
    { "ST",   new OpCode{ byteCode=0b00101000, param=Param.reg_ram  }},
    { "IST",  new OpCode{ byteCode=0b00101010, param=Param.reg      }},
    { "ISTI", new OpCode{ byteCode=0b00101100, param=Param.imdt     }},
    { "PUSH", new OpCode{ byteCode=0b00101110, param=Param.reg      }},
    { "PUSHI",new OpCode{ byteCode=0b00110000, param=Param.imdt     }},
    { "POP",  new OpCode{ byteCode=0b00110010, param=Param.reg      }},
    { "LPM",  new OpCode{ byteCode=0b00110100, param=Param.reg_pm   }},
    { "ILPM", new OpCode{ byteCode=0b00110110, param=Param.reg      }},
    { "STM",  new OpCode{ byteCode=0b00111000, param=Param.reg_pm   }},
    { "ISTM", new OpCode{ byteCode=0b00111010, param=Param.reg      }},
    { "ISTMI",new OpCode{ byteCode=0b00111100, param=Param.imdt     }},

    // branch
    { "JMP",  new OpCode{ byteCode=0b01000000, param=Param.pm }},
    { "IJMP", new OpCode{ byteCode=0b01000010, param=Param.pm }},
    { "JHI",  new OpCode{ byteCode=0b01000100, param=Param.pm }},
    { "JSLO", new OpCode{ byteCode=0b01000110, param=Param.pm }},
    { "JGE",  new OpCode{ byteCode=0b01001000, param=Param.pm }},
    { "JLT",  new OpCode{ byteCode=0b01001010, param=Param.pm }},

    { "JEQ",  new OpCode{ byteCode=0b01100000, param=Param.pm }},
    { "JNEQ", new OpCode{ byteCode=0b01100010, param=Param.pm }},
    { "JSHI", new OpCode{ byteCode=0b01100110, param=Param.pm }},
    { "JLO",  new OpCode{ byteCode=0b01100100, param=Param.pm }},
    { "JMI",  new OpCode{ byteCode=0b01101000, param=Param.pm }},
    { "JPL",  new OpCode{ byteCode=0b01101010, param=Param.pm }},

    { "RJMP",  new OpCode{ byteCode=0b01000001, param=Param.pm }},
    { "RIJMP", new OpCode{ byteCode=0b01000011, param=Param.pm }},
    { "RJHI",  new OpCode{ byteCode=0b01000101, param=Param.pm }},
    { "RJSLO", new OpCode{ byteCode=0b01000111, param=Param.pm }},
```

```
{ "RJGE",  new OpCode{ byteCode=0b01001001, param=Param.pm }},
{ "RJLT",  new OpCode{ byteCode=0b01001011, param=Param.pm }},

{ "RJEQ",  new OpCode{ byteCode=0b01100001, param=Param.pm }},
{ "RJNEQ", new OpCode{ byteCode=0b01100011, param=Param.pm }},
{ "RJSHI", new OpCode{ byteCode=0b01100111, param=Param.pm }},
{ "RJLO",  new OpCode{ byteCode=0b01100101, param=Param.pm }},
{ "RJMI",  new OpCode{ byteCode=0b01101001, param=Param.pm }},
{ "RJPL",  new OpCode{ byteCode=0b01101011, param=Param.pm }},

// branch by flag
{ "JFZS", new OpCode{ byteCode=0b01100000, param=Param.pm }},
{ "JFCS", new OpCode{ byteCode=0b01100100, param=Param.pm }},
{ "JFNS", new OpCode{ byteCode=0b01101000, param=Param.pm }},
{ "JFVS", new OpCode{ byteCode=0b01101100, param=Param.pm }},
{ "JFTS", new OpCode{ byteCode=0b01110000, param=Param.pm }},
{ "JFIS", new OpCode{ byteCode=0b01110100, param=Param.pm }},
{ "JFRS", new OpCode{ byteCode=0b01111000, param=Param.pm }},
{ "JFPS", new OpCode{ byteCode=0b01111100, param=Param.pm }},

{ "JFZC", new OpCode{ byteCode=0b01100010, param=Param.pm }},
{ "JFCC", new OpCode{ byteCode=0b01100110, param=Param.pm }},
{ "JFNC", new OpCode{ byteCode=0b01101010, param=Param.pm }},
{ "JFVC", new OpCode{ byteCode=0b01101110, param=Param.pm }},
{ "JFTC", new OpCode{ byteCode=0b01110010, param=Param.pm }},
{ "JFIC", new OpCode{ byteCode=0b01110110, param=Param.pm }},
{ "JFRC", new OpCode{ byteCode=0b01111010, param=Param.pm }},
{ "JFPC", new OpCode{ byteCode=0b01111110, param=Param.pm }},

{ "RJFZS", new OpCode{ byteCode=0b01100001, param=Param.pm }},
{ "RJFCS", new OpCode{ byteCode=0b01100101, param=Param.pm }},
{ "RJFNS", new OpCode{ byteCode=0b01101001, param=Param.pm }},
{ "RJFVS", new OpCode{ byteCode=0b01101101, param=Param.pm }},
{ "RJFTS", new OpCode{ byteCode=0b01110001, param=Param.pm }},
{ "RJFIS", new OpCode{ byteCode=0b01110101, param=Param.pm }},
{ "RJFRS", new OpCode{ byteCode=0b01111001, param=Param.pm }},
{ "RJFPS", new OpCode{ byteCode=0b01111101, param=Param.pm }},

{ "RJFZC", new OpCode{ byteCode=0b01100011, param=Param.pm }},
{ "RJFCC", new OpCode{ byteCode=0b01100111, param=Param.pm }},
{ "RJFNC", new OpCode{ byteCode=0b01101011, param=Param.pm }},
{ "RJFVC", new OpCode{ byteCode=0b01101111, param=Param.pm }},
{ "RJFTC", new OpCode{ byteCode=0b01110011, param=Param.pm }},
{ "RJFIC", new OpCode{ byteCode=0b01110111, param=Param.pm }},
{ "RJFRC", new OpCode{ byteCode=0b01111011, param=Param.pm }},
{ "RJFPC", new OpCode{ byteCode=0b01111111, param=Param.pm }},

// double arithmetics
{ "ADD",  new OpCode{ byteCode=0b10000000, param=Param.reg_reg }},
{ "ADDC", new OpCode{ byteCode=0b10000010, param=Param.reg_reg }},
```

```
    { "SUB",  new OpCode{ byteCode=0b10000100, param=Param.reg_reg }},
    { "SUBC", new OpCode{ byteCode=0b10000110, param=Param.reg_reg }},
    { "MUL",  new OpCode{ byteCode=0b10001000, param=Param.reg_reg }},
    { "DIV",  new OpCode{ byteCode=0b10001010, param=Param.reg_reg }},
    { "MULS", new OpCode{ byteCode=0b10001100, param=Param.reg_reg }},
    { "DIVS", new OpCode{ byteCode=0b10001110, param=Param.reg_reg }},
    { "AND",  new OpCode{ byteCode=0b10010000, param=Param.reg_reg }},
    { "OR",   new OpCode{ byteCode=0b10010010, param=Param.reg_reg }},
    { "XOR",  new OpCode{ byteCode=0b10010100, param=Param.reg_reg }},
    { "CP",   new OpCode{ byteCode=0b10010110, param=Param.reg_reg }},
    { "CPC",  new OpCode{ byteCode=0b10011000, param=Param.reg_reg }},

    { "ADDI", new OpCode{ byteCode=0b10000001, param=Param.reg_imdt }},
    { "ADDCI",new OpCode{ byteCode=0b10000011, param=Param.reg_imdt }},
    { "SUBI", new OpCode{ byteCode=0b10000101, param=Param.reg_imdt }},
    { "SUBCI",new OpCode{ byteCode=0b10000111, param=Param.reg_imdt }},
    { "MULI", new OpCode{ byteCode=0b10001001, param=Param.reg_imdt }},
    { "DIVI", new OpCode{ byteCode=0b10001011, param=Param.reg_imdt }},
    { "MULSI",new OpCode{ byteCode=0b10001101, param=Param.reg_imdt }},
    { "DIVSI",new OpCode{ byteCode=0b10001111, param=Param.reg_imdt }},
    { "ANDI", new OpCode{ byteCode=0b10010001, param=Param.reg_imdt }},
    { "ORI",  new OpCode{ byteCode=0b10010011, param=Param.reg_imdt }},
    { "XORI", new OpCode{ byteCode=0b10010101, param=Param.reg_imdt }},
    { "CPI",  new OpCode{ byteCode=0b10010111, param=Param.reg_imdt }},
    { "CPCI", new OpCode{ byteCode=0b10011001, param=Param.reg_imdt }},

    // single arithmetics
    { "IS",  new OpCode{ byteCode=0b10100000, param=Param.reg }},
    { "COM", new OpCode{ byteCode=0b10100010, param=Param.reg }},
    { "NEG", new OpCode{ byteCode=0b10100100, param=Param.reg }},
    { "INC", new OpCode{ byteCode=0b10100110, param=Param.reg }},
    { "DEC", new OpCode{ byteCode=0b10101000, param=Param.reg }},
    { "SWAP",new OpCode{ byteCode=0b10101010, param=Param.reg }},
    { "LSL", new OpCode{ byteCode=0b10101100, param=Param.reg }},
    { "LSR", new OpCode{ byteCode=0b10101110, param=Param.reg }},
    { "ROL", new OpCode{ byteCode=0b10110000, param=Param.reg }},
    { "ROR", new OpCode{ byteCode=0b10110010, param=Param.reg }},
    { "ASR", new OpCode{ byteCode=0b10110100, param=Param.reg }},
    { "TST", new OpCode{ byteCode=0b10110110, param=Param.reg }}
};

Dictionary<string, string> RegSymbols = new Dictionary<string, string>()
{
    // GPIO Registers
    {"PA_IN",  "0xE2" },
    {"PB_IN",  "0xE3" },
    {"PC_IN",  "0xE4" },
    {"PA_OUT", "0xE5" },
    {"PB_OUT", "0xE6" },
    {"PC_OUT", "0xE7" },
```

```
    {"PA_CONF","0xE8" },
    {"PB_CONF","0xE9" },
    {"PC_CONF","0xEA" },
    {"PD_IN",  "0xEB" },
    {"PD_OUT", "0xEC" },
    {"PD_CONF","0xED" },

    // UART Registers
    {"UART_CONF","0xEE" },
    {"UART_TD",  "0xEF" },
    {"UART_RD",  "0xF0" },

    // Timer Registers
    {"TIMA_VAL", "0xF1" },
    {"TIMB_VAL", "0xF2" },
    {"TIMA_TOP", "0xF3" },
    {"TIMB_TOP", "0xF4" },
    {"TIMA_COMP","0xF5" },
    {"TIMB_COMP","0xF6" },
    {"TIMA_CONF","0xF7" },
    {"TIMB_CONF","0xF8" },
    {"MICR_L",   "0xF9" },
    {"MICR_H",   "0xFA" },

    // Special Registers
    {"ALU_B", "0xFB" },
    {"SP",    "0xFC" },
    {"IADDR", "0xFD" },
    {"PC",    "0xFE" },
    {"FLAGS", "0xFF" }
};

string[] input_extensions =
{
    ".hasm",
    ".asm"
};

string[] output_extensions =
{
    ".hex",
    ".bin"
};

string std_out_extension = ".hex";


main();
```

```
void main()
{
    Console.WriteLine("----------- HTMega Assembler ------------");
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    // expected arguments: input_file_path (optional)output_file_path
    string[] arguments = Environment.GetCommandLineArgs();

    if (arguments.Length < 2) // not enough arguments
        throw new ArgumentError("no file given");

    string input_path = parsePath(arguments[1],input_extensions,true);

    string output_path;
    if (arguments.Length < 3) // no output file path
    {
        output_path =
    parsePath(input_path.Replace(Path.GetExtension(input_path),
    std_out_extension),
            output_extensions,false); // input file with extension .bin
    as output path

        console_warning($"no output path given, output is written to
    {output_path}");
    }

    else if (arguments.Length > 3) // too many arguments
    {
        throw new ArgumentError($"to many arguments:
    {arguments.Length}");
    }

    else
    {
        output_path = parsePath(arguments[2], output_extensions,false);
    }

    Console.WriteLine(output_path);

    string[] lines = File.ReadAllLines(input_path);

    List<byte> bytecode = new List<byte>();
    for (int i = 1; i <= lines.Length; i++)
    {

        string line = lines[i-1].Trim();

        if (line.Length == 0) // skip if empty
            continue;
```

```
    if (line[..2] == "//") // skip if comment
        continue;

    line = line.Split("//")[0]; // split off comments
    string[] tokens = tokenize(line); // tokenize (split by space
and comma)
    string mnemonic = tokens[0].ToUpper();

    if (OPCodes.ContainsKey(mnemonic))
    {
        // get opcode of mnemonic and start building bytecode
        OpCode op = OPCodes[mnemonic];

        bytecode.Add(op.byteCode);

        // parse parameters and check for right amount of parameters
        string[] parameters = tokens[1..];
        switch (op.param)
        {
            case Param.none:
                if (parameters.Length > 0)
                    throw new SyntaxError($"no parameters allowed
for: {mnemonic}", i);
                bytecode.AddRange(new byte[] { 0x00, 0x00, 0x00});
                break;

            case Param.reg:
                testParameters(parameters, 2, i);
                bytecode.Add(parseRegAddr(parameters[0], i));
                bytecode.Add(0x00);
                bytecode.Add(0x00);
                break;

            case Param.reg_reg:
                testParameters(parameters, 2, i);
                bytecode.Add(parseRegAddr(parameters[0], i));
                bytecode.Add(parseRegAddr(parameters[1], i));
                bytecode.Add(0x00);
                break;

            case Param.reg_imdt:
                testParameters(parameters, 2, i);
                bytecode.Add(parseRegAddr(parameters[0], i));
                bytecode.AddRange(parseImdt(parameters[1], i));
                break;

            case Param.reg_pm:
                testParameters(parameters, 2, i);
                bytecode.Add(parseRegAddr(parameters[0], i));
```

```
                    bytecode.AddRange(parsePMAddr(parameters[1], i));
                    break;

                case Param.reg_ram:
                    testParameters(parameters, 2, i);
                    bytecode.Add(parseRegAddr(parameters[0], i));
                    bytecode.AddRange(parseRAMAddr(parameters[1], i));
                    break;

                case Param.pm:
                    testParameters(parameters, 1, i);
                    bytecode.Add(0x00);
                    bytecode.AddRange(parsePMAddr(parameters[0], i));
                    break;

                case Param.imdt:
                    testParameters(parameters, 1, i);
                    bytecode.Add(0x00);
                    bytecode.AddRange(parseImdt(parameters[0], i));
                    break;

                default: break;
            }
        }
        else
            throw new SyntaxError($"invalid operator: {mnemonic}", i);
    }
    File.WriteAllBytes(output_path, bytecode.ToArray());

    stopwatch.Stop();
    Console.WriteLine($"-------- finished, elapsed time:
    {stopwatch.ElapsedMilliseconds}ms ---------");
}

string[] tokenize(string line)
{
    List<string> lines = line.Split(',').ToList();

    for (int i = 0; i < lines.Count; i++)
    {
        lines[i] = lines[i].Trim();
        string[] parts = lines[i].Split(' ');
        if (parts.Length > 1)
        {
            parts.Select(part => part.Trim());
            lines.RemoveAt(i);
            lines.InsertRange(i, parts);
        }
    }
    return lines.ToArray();
```

```
}

string parsePath(string path, string[] allowedExtensions, bool isInput)
{
    // make relative path absolute
    if (!Path.IsPathRooted(path))
        path = Path.Combine(Environment.CurrentDirectory, path);

    if (!Path.HasExtension(path))
        throw new ArgumentError("expected file path, got: " + path);


    if (isInput && !File.Exists(path)) // test if input file exists
        throw new ArgumentError("input file does not exist: " + path);
    else
        Directory.CreateDirectory(Directory.GetParent(path).FullName);


    // check if file extension is allowed
    bool isAllowed = false;
    string formattedExtensions = "";
    foreach(string ext in allowedExtensions)
    {
        formattedExtensions += ext + ", ";
        if (Path.GetExtension(path) == ext)
            isAllowed = true;
    }
    formattedExtensions.Remove(formattedExtensions.Length - 1,1);
    if (!isAllowed)
        throw new ArgumentError($"wrong file type of {path}, expected:
    {formattedExtensions}");

    return path;
}

byte parseRegAddr(string line, int i)
{
    line = line.ToUpper();
    if (RegSymbols.ContainsKey(line))
        line = RegSymbols[line];

    if (StrParser.ToByte(line, out byte addr))
        return addr;
    else
        throw new ValueError($"expected register address of type byte,
    got {line}", i );
}

byte[] parsePMAddr(string line, int i)
{
```

```
    if (StrParser.ToUInt16(line, out ushort addr))
    {
        if (addr < PM_SIZE)
            return BitConverter.GetBytes(addr);
        else
            throw new ValueError($"program memory address {line} exceeds
    program memory size of {PM_SIZE}", i);
    }
    else
        throw new ValueError($"expected program memory address of type
    uint16, got {line}", i);
}

byte[] parseRAMAddr(string line, int i)
{
    if (StrParser.ToUInt16(line, out ushort addr))
        if (addr < PM_SIZE)
            return BitConverter.GetBytes(addr);
        else
            throw new ValueError($"RAM address {line} exceeds RAM size
    of {RAM_SIZE}", i);
    else
        throw new ValueError($"expected RAM address of type uint16, got
    {line}", i);
}

byte[] parseImdt(string line, int i)
{
    if (StrParser.ToInt16(line, out short Imdt))
        return BitConverter.GetBytes(Imdt);
    else
        throw new ValueError($"expected Value of type int16, got
    {line}", i);
}

void testParameters(string[] parameters, int expectedParams, int i)
{
    if (parameters.Length != 2)
        throw new SyntaxError($"wrong amount of parameters:
    {parameters.Length}; expected amount of parameters:
    {expectedParams}", i);

}

void console_warning(string message) => Console.WriteLine($"WARNING:
    {message}");
void console_info(string message) => Console.WriteLine($"INFO:
    {message}");
```

```
public class StrParser
{
    static public bool ToByte(string s, out byte value)
    {
        s = s.Trim();
        try
        {

            if (s.Contains("0x"))
                value = Convert.ToByte(s.Replace("0x", ""), 16);
            else if (s.Contains("0b"))
                value = Convert.ToByte(s.Replace("0b", ""), 2);
            else
                value = Convert.ToByte(s, 10);
            return true;
        }
        catch (Exception)
        {
            value = 0;
            return false;
        }
    }

    static public bool ToUInt16(string s, out ushort value)
    {
        s = s.Trim();
        try
        {

            if (s.Contains("0x"))
                value = Convert.ToUInt16(s.Replace("0x", ""), 16);
            else if (s.Contains("0b"))
                value = Convert.ToUInt16(s.Replace("0b", ""), 2);
            else
                value = Convert.ToUInt16(s, 10);
            return true;
        }
        catch (Exception)
        {
            value = 0;
            return false;
        }
    }

    static public bool ToInt16(string s, out short value)
    {
        s = s.Trim();
        try
        {
```

```
            if (s.Contains("0x"))
                value = Convert.ToInt16(s.Replace("0x", ""), 16);
            else if (s.Contains("0b"))
                value = Convert.ToInt16(s.Replace("0b", ""), 2);
            else
                value = Convert.ToInt16(s, 10);
            return true;
        }
        catch (Exception)
        {
            value = 0;
            return false;
        }
    }
}

[Serializable]
public class ValueError : Exception
{
    public ValueError() { }
    public ValueError(string message) : base(message) { }
    public ValueError(string message, Exception inner) : base(message,
    inner) { }
    public ValueError(string message, int line) : base($"at line {line}:
    {message}") { }
    protected ValueError(
      System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context) :
    base(info, context) { }
}

[Serializable]
public class SyntaxError : Exception
{
    public SyntaxError() { }
    public SyntaxError(string message) : base(message) { }
    public SyntaxError(string message, Exception inner) : base(message,
    inner) { }
    public SyntaxError(string message, int line) : base($"at line
    {line}: {message}") { }
    protected SyntaxError(
      System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context) :
    base(info, context) { }
}

public class ArgumentError : Exception
{
    public ArgumentError() { }
    public ArgumentError(string message) : base(message) { }
```

```
    public ArgumentError(string message, Exception inner) :
    base(message, inner) { }
    public ArgumentError(string message, int line) : base($"at line
    {line}: {message}") { }
    protected ArgumentError(
      System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context) :
    base(info, context) { }
}



class OpCode
{
    public Param param;
    public byte byteCode;
}

enum Param
{
    none,
    reg,
    pm,
    imdt,
    reg_reg,
    reg_imdt,
    reg_ram,
    reg_pm
}

--
    ------------------------------------------------------------------------------


--
    ------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity Timer_16bit is
  Port (
    clk_12 : in std_logic;
    value_out : out std_logic_vector(15 downto 0) ;
    compare_value, top : in std_logic_vector (15 downto 0);
    enabled, en_comparison, en_pwm, reset : in std_logic;
    out_comparison, out_overflow, out_port : out std_logic := '0';
    port_config : in std_logic_vector(1 downto 0);
    prescaler : in std_logic_vector(2 downto 0)
);
end Timer_16bit;

architecture Behavioral of Timer_16bit is

signal pre_value : integer range 0 to 16383 := 0;
```

```
signal value : integer range 0 to 65536 := 0;
signal port_value : std_logic := '0';
signal is_match, is_value_up, port_enabled : boolean := false;
signal prescaler_value : integer range 0 to 16383;
type prescaler_type is array(7 downto 0) of integer range 0 to 16383;

constant prescalers : prescaler_type := (12000,1200,120,12,6,3,2,1);

begin
  value_out <= std_logic_vector(to_unsigned(value, 16));
  out_comparison <= '1' when is_match else '0';
  out_port <= port_value when port_enabled and enabled = '1' else 'Z';
  prescaler_value <= prescalers(to_integer(unsigned(prescaler)))-1;

  process(clk_12) begin
    if rising_edge(clk_12) and enabled = '1' then

      out_overflow <= '0';
      if pre_value >= prescaler_value then
        pre_value <= 0;
        is_value_up <= true;
        if value >= (unsigned(top)-1) then
          out_overflow <= '1';
          value <= 0;
        else
          value <= value + 1;
        end if;
      else
        is_value_up <= false;
        pre_value <= pre_value + 1;
      end if;

      if reset = '1' then
        value <= 0;
        pre_value <= 0;
      end if;

      port_enabled <= port_config /= "00";

      if en_comparison = '1' then
        is_match <= value = (unsigned(compare_value)-1) and is_value_up ;

        if en_pwm = '1' then
          case port_config is
            when "01" =>
              if is_match then
                case port_value is
                  when '1' => port_value <= '0';
                  when '0' => port_value <= '1';
                  when others =>
                end case;
              end if;

            when "10" =>
              if value = 0 then
                port_value <= '1';
              elsif is_match then
                port_value <= '0';
              end if;

            when "11" =>
              if value = 0 then
                port_value <= '0';
              elsif is_match then
                port_value <= '1';
              end if;
            when others =>
```

```
        end case;

      elsif is_match then
        case port_config is
          when "01" =>
            case port_value is
              when '1' => port_value <= '0';
              when '0' => port_value <= '1';
              when others =>
            end case;
          when "10" =>
            port_value <= '0';
          when "11" =>
            port_value <= '1';
          when others =>
        end case;
      end if;
    end if;
  end if;
 end process;
end Behavioral;
```

Listing 1: Timer Module