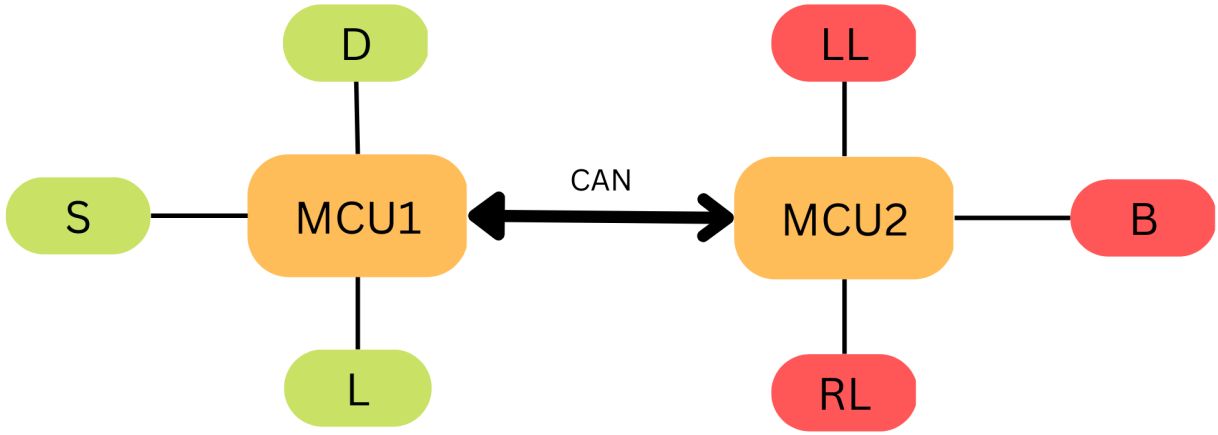
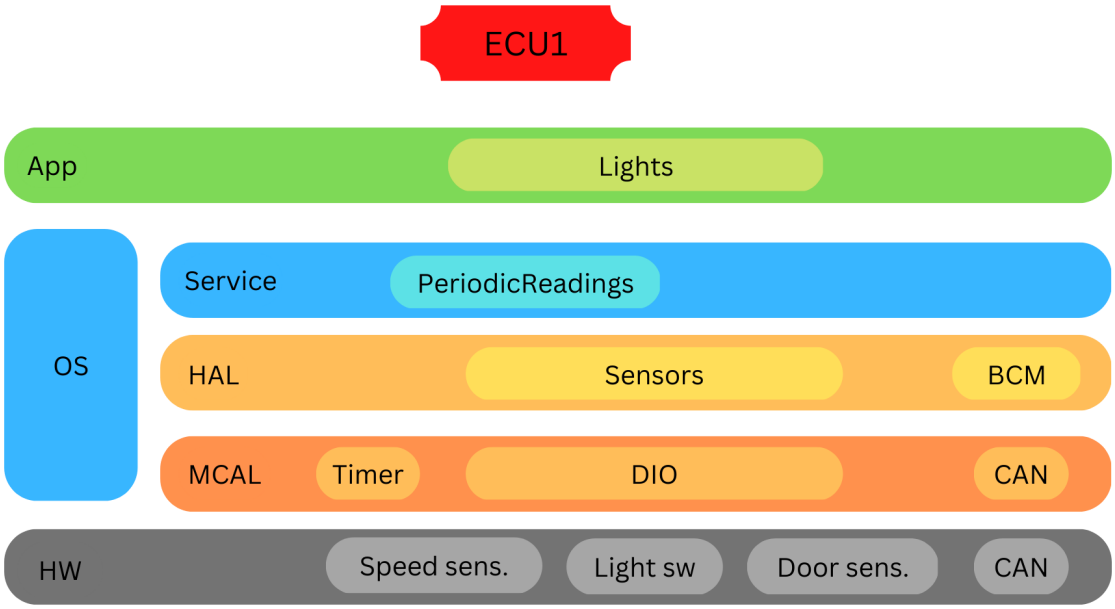
Static design

System schematic (Block Diagram):



ECU1:

1) Layered architecture



2) Components and modules

- Timer
- DIO
- CAN

3) APIs for each module as well as a detailed description for the used typedefs
we don't want the implementation of the function we need (description of each function and types of returns and parameters)

//CAN.h:

```
typedef struct{
        uint32_t id;
        uint8_t buffer[8];
        uint32_t buffer_length;
} can_t;
enum status {Disabled, Enabled};
#define CANCTL *(volatile (void *)0x40040000)
```

//CAN.c:

```
/*
*Description: Initialize the CAN module, by assigning related values to CAN registers.
*Parameters: CAN type
*Return: none
*/
void CanInit(can_t can){
        RCGC(can.id)= 0x12345678;
```

```
	CANCTL(can.id)= 0x12345678;
}
/*
*Description: Send message through CAN module by passing message byte by byte.
*Parameters: CAN type pointer
*Return: status of the operation.
*/
enum status CanSend(can_t * frame){
	for(int i=0; i<frame->buffer_length; i++){
		CANDATA= frame->buffer[i];
	}
	if(CANSTAT== 0x12345678)
		return 1;
	else
		return 0;
}
/*
*Description: Receive a CAN message if there is any.
*Parameters: CAN type pointer
*Return: receive status.
*/
bool CanReceive(can_t* frame){
	if(CANSTATE== 0x12345678){
		uint32_t i;
		for(i=0; CANSTATE== 0x12345678; i++){
			frame->buffer[i]=CANDATA;
		}
		frame->buffer_length=i+1;
		return 1;
	}
	else
		return 0;
}
```

//DIO.h:

```
enum Port{portA, portB, portC, portD};
enum Pin{pin0, pin1, pin2, pin3, pin4, pin5, pin6, pin7};
enum value{False, True};
enum state{ouput, input};
typedef struct {
	enum Port port;
	enum Pin pin;
	enum state s;
	enum value v;
```

```
} DIO_t;
```

//DIO.c:

```
/*
*Description: Initialize a pin in a port by assigning suitable values in different registers.
*Parameters: DIO type.
*Return: none
*/
void DioInit(DIO_t dio){
	if(dio.s==True)
		GPIODIR(dio.port)|=0x01<<dio.pin;
	else
		GPIODIR(dio.port)&=~0x01<<dio.pin;
}
/*
*Description: Pin configuration, use to change pin form output to input and vise versa.
*Parameters: DIO type
*Return: none.
*/
void DioConfig(DIO_t dio){
	if(dio.s==True)
		GPIODIR(dio.port)|=0x01<<dio.pin;
	else
		GPIODIR(dio.port)&=~0x01<<dio.pin;
}
/*
*Description: put a value on a pin.
*Parameters: DIO type
*Return: none
*/
void DioSet(DIO_t dio){
	if(dio.v==True)
		GPIODATA(dio.port)|=0x01<<dio.pin;
	else
		GPIODATA(dio.port)&=~0x01<<dio.pin;
}
/*
*Description: get a value from a pin.
*Parameters: DIO type
*Return: pin value.
*/
enum value Dioget(DIO_t dio){
	return (GPIODATA(dio.port)>>dio.pin)&1;
}
```

//Timer.h

```
enum ID {Timer1, Timer2};
enum Direction {CountUp, CountDwn};
enum State{Stop=0, Start}
typedef struct {
        enum ID id;
        enum Direction dir;
        uint32_t Count;
}Timer_t;
```

//Timer.c

```
/*
*Description: Initialize the timer, using given parameters
*Parameters: Timer type
*Return: none
*/
void TimerInit(Timer_t timer){
        TCUNT(timer.id)= timer.Count;
        TCONFIG(timer.id)= Start;
}
/*
*Description: Set timer value to start counting from.
*Parameters: Timer type
*Return: none
*/
void TimerSet(Timer_t timer){
        TCUNT(timer.id)= timer.Count;
}
/*
*Description: get timer current state counting or stopped.
*Parameters: Timer type
*Return: timer state
*/
enum State TimerStatus(Timer_t timer){
        if(TCURRNT(timer.id)=0)
                return Stop;
        else
                return Start;
}
/*
*Description: get timer current count.
*Parameters: Timer type
*Return: timer count.
```

```
*/
uint32_t TimerGet(Timer_t timer){
	return TCURRENT(timer.id);
}
```

//BCM.h:

```
enum BcmType {CAN, Other};
typedef struct{
	enum BcmType type;
	uint32_t id;
	uint8_t buffer[8];
	uint32_t buffer_length;
} Bcm_t;
```

//BCM.c:

```
/*
*Description: Initialize specified communication module - only CAN in this case.
*Parameters: BCM type
*Return: none
*/
void BcmInit(Bcm_t bcm){
	if(bcm.type== CAN)
		can_t can;
		can.id= bcm.id;
		can.buffer= bcm.buffer;
		can.buffer_length= bcm.buffer_length;
		CanInit(can);
}
/*
*Description: send data through the defined communication module - CAN in this case
*Parameters: BCM type
*Return: sending status
*/
enum status BcmSend(Bcm_t * bcm){
	if(bcm.id==CAN)
		return CanSend(bcm)
}
/*
*Description: receive a message through given communication module - only CAN in this case
*Parameters: BCM type
*Return: receive status
*/
bool BcmReceive(Bcm_t* bcm){
	if(bcm.id== CAN)
```

```
        return CanReceive(bcm);
}
```

//Sensors.h

```
typedef struct {
        uint32_t spd;
        uint32_t lit;
        uint32_t dor;
} Sens_t;
Dio_t speed;
Dio_t light;
Dio_t door;
```

//Sensors.c

```
/*
*Description: Initialize sensors through DIO module
*Parameters: none
*Return: none
*/
void SensInit(){
        DioInit(speed);
        DioInit(light);
        DioInit(door);
}
/*
*Description: read sensors values through DIO module
*Parameters: sensor type
*Return: none
*/
void SensRead(Sens_t sens){
        sens.spd= DioGet(speed);
        sens.lit= DioGet(light);
        sens.dor= Dioget(door);
}
```

//PeriodicReadings.h:

//PeriodicReadings.c:

```
/*
*Description: Initialize periodic reading module by initializing timer, and sensors, and set timer
value.
*Parameters: timer type
*Return: none
```

```
*/
void PeriodicInit(Timer_t timer){
	TimerInit(timer);
	SenseInit();
	TimerSet(timer);
}
/*
*Description: Periodic read return new readings from sensors if required time passed.
*Parameters: Timer type, and sensors type
*Return: none.
*/
void PeriodicRead(Timer_t timer, Sens_t sens){
	if(TimerStatus(timer)==Stop)
		SenseRead(sens);
}
```

//Lights.h:

//Lights.c:

```
/*
*Description: Initialize the lights module by initializing periodic readings and BCM
*Parameters: Timer type, and BCM type
*Return: none
*/
void LightInit(Timer_t timer, Bcm_t bcm){
	PeriodicInit(timer)
	BcmInit(bcm)
}
/*
*Description: Get periodic readings in time and send them through BCM.
*Parameters: Timer, sensors, and BCM types
*Return: none
*/
void LightRead(Timer_t timer, Sens_t sens, Bcm_t *bcm){
	PeriodicRead(Timer, sens);
	BcmSend(&bcm);
}
```

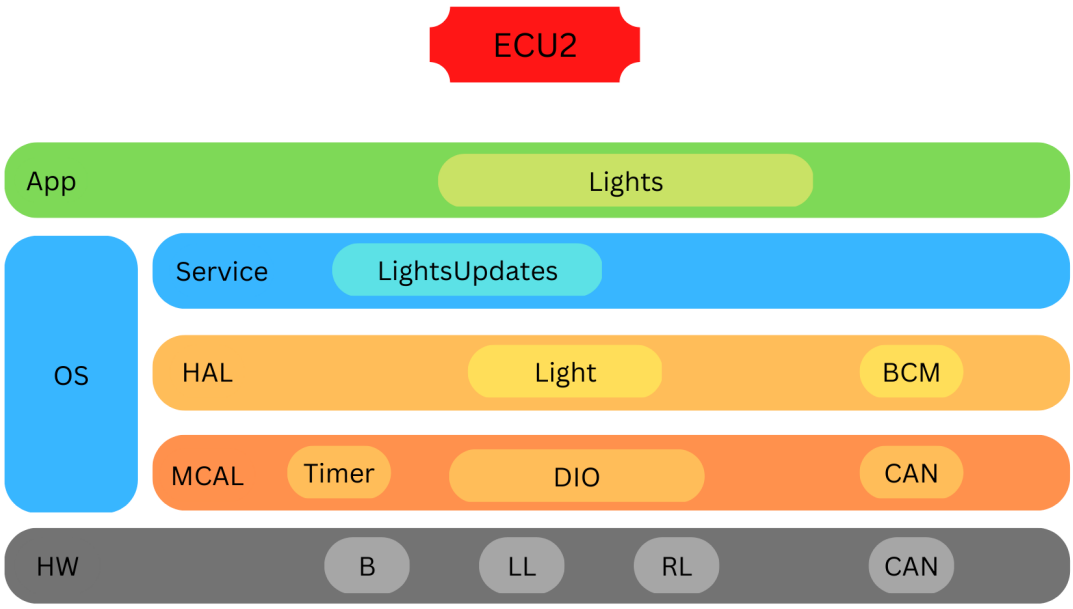
4) Folder structure

- ECU1
  - App
    - LightUpdates.h
    - LightUpdates.c
  - Hal

        - BCM.h
        - BCM.c
    - Mcal
        - CAN.h
        - CAN.c
        - DIO.h
        - DIO.c
        - Timer.h
        - Timer.c
    - Service
        - Sensors.h
        - Sensors.c

ECU2:

1) Layered architecture



2) Components and modules

- Timer
- DIO
- CAN

3) APIs for each module as well as a detailed description for the used typedefs

//Light.h

```
typedef struct {
        bool ll;
```

```
	bool rl;
	bool b;
} Light_t;
Dio_t llight;
Dio_t rlight;
Dio_t buz;
```

//Light.c

```
/*
*Description: Light initialization function, Initialize each light, and buzzer through DIO module.
*Parameters: none
*Return: none
*/
void LightInit(){
	DioInit(llight);
	DioInit(rlight);
	DioInit(buz);
}
/*
*Description: put a new output state through the DIO module
*Parameters: light type
*Return: none
*/
void LightWrite(Light_t light){
	DioSet(llight);
	DioSet(rlight);
	DioSet(buz);
}
```

//LightsUpdates.h:

```
typedef struct{
	Sens_t sens;
}Update_t;
```

//LightsUpdates.c:

```
/*
*Description: Lights updates initialization, initialize light and timer modules.
*Parameters: timer type
*Return: none
*/
void LUInit(Timer_t timer){
	TimerInit(timer);
	LightInit();
}
```

```
/*
*Description: receive updated data and generate new output state
*Parameters: update type
*Return: none
*/
void LUWrite(Update_t update){
        if(UpdateCheck(update)==True)
                LightWrite(light);
}
```

//Lights.h:

//Lights.c:

```
/*
*Description: Initialize the lights module by initializing lights updates and BCM modules.
*Parameters: timer and BCM types.
*Return: none
*/
void LightsInit(Timer_t timer, Bcm_t bcm){
        LUInit(timer)
        BcmInit(bcm)
}
/*
*Description: Receive sensors state from BCM and pass new updates to Lights Updates module
if there is any.
*Parameters: update and BCM type
*Return: none
*/
void LightsWrite(Update_t update, Bcm_t *bcm){
        BcmReceive(&bcm);
        LUWrite(update)
}
```

4) Folder structure

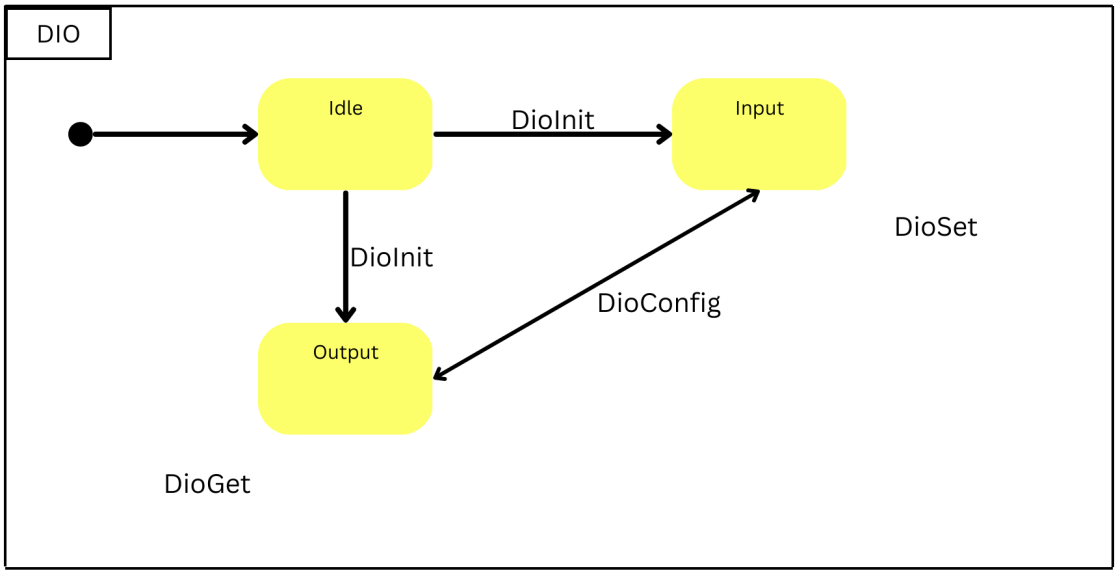
- ECU2
    - App
        - Lights.h
        - Lights.c
    - Service
        - LightsUpdates.h
        - LightsUpdates.c
    - Hal
        - Light.h

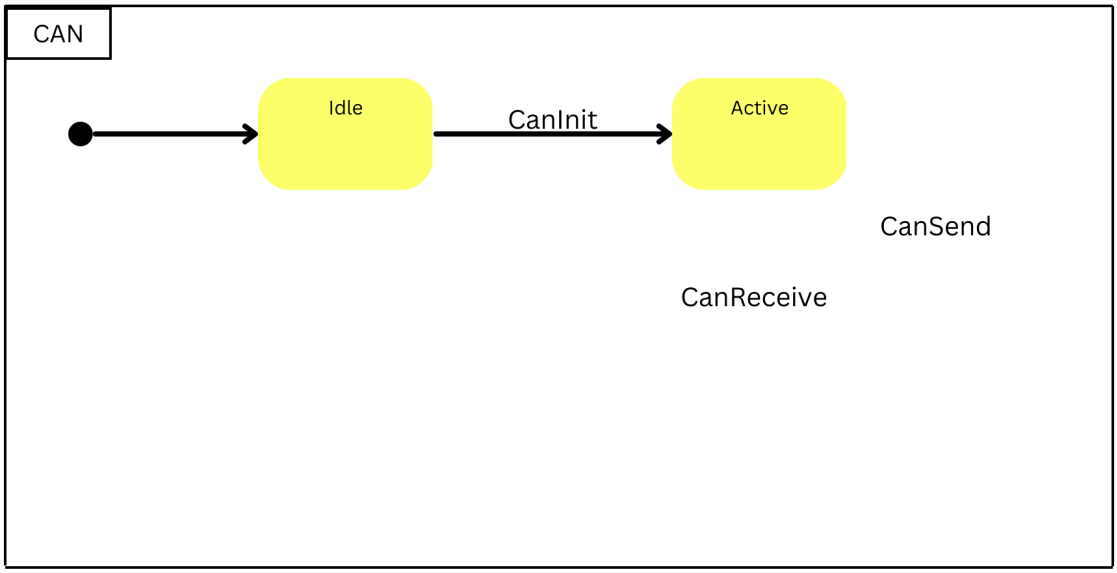
        - Light.c
        - BCM.h
        - BCM.c
    - Mcal
        - CAN.h
        - CAN.c
        - DIO.h
        - DIO.c
        - Timer.h
        - Timer.c

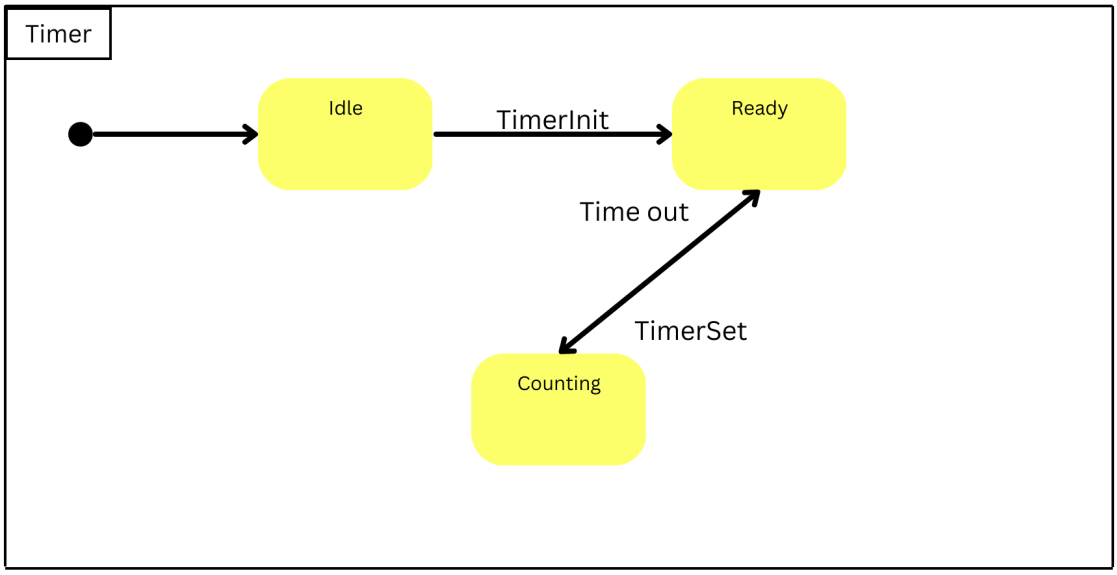
Dynamic design

ECU 1:

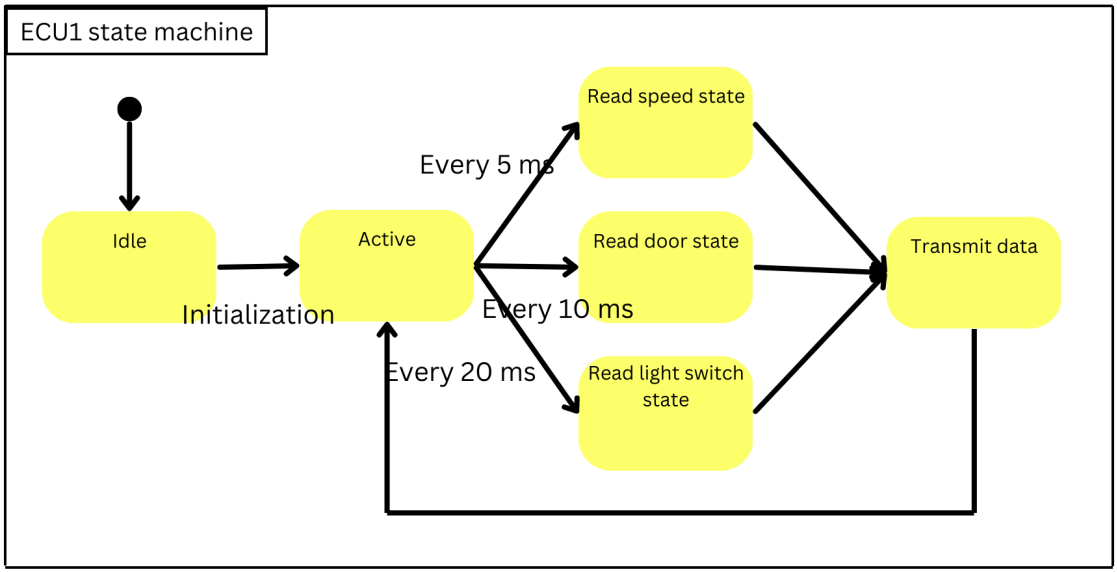
1) Draw a state machine diagram for each ECU component



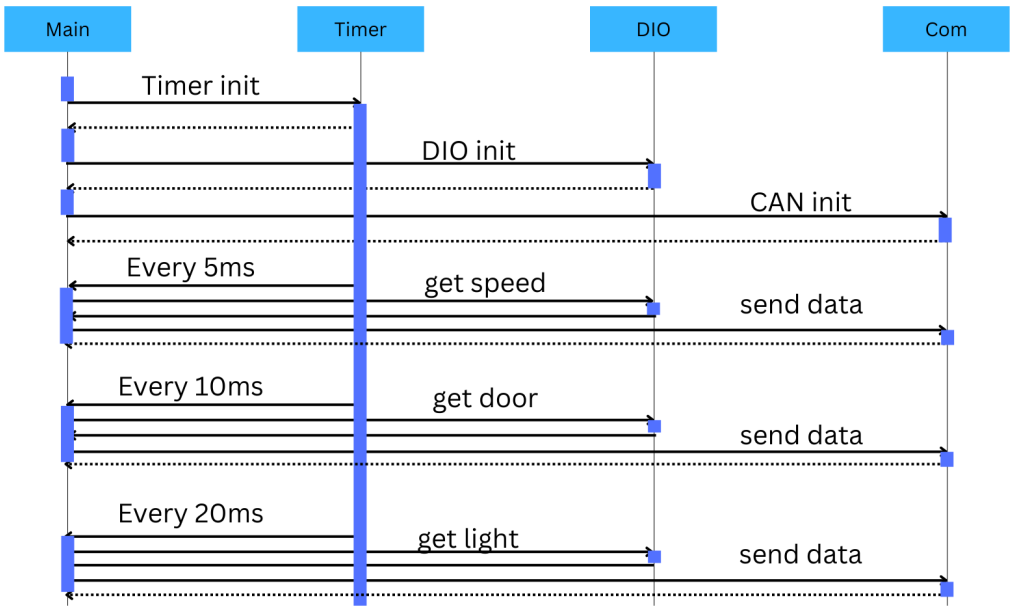




2) Draw a state machine diagram for the ECU operation



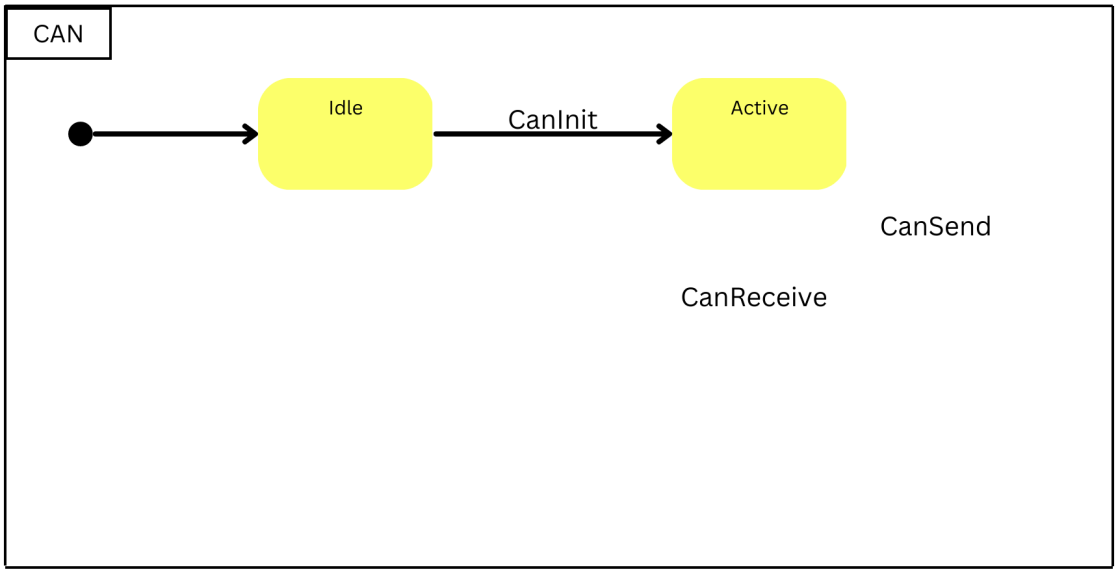
3) Draw the sequence diagram for the ECU

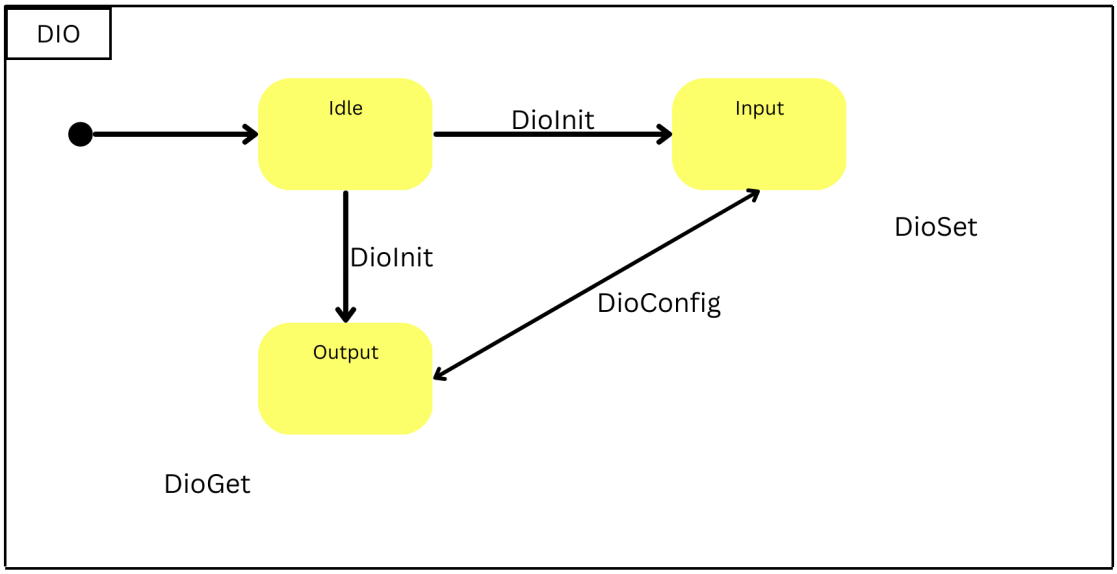


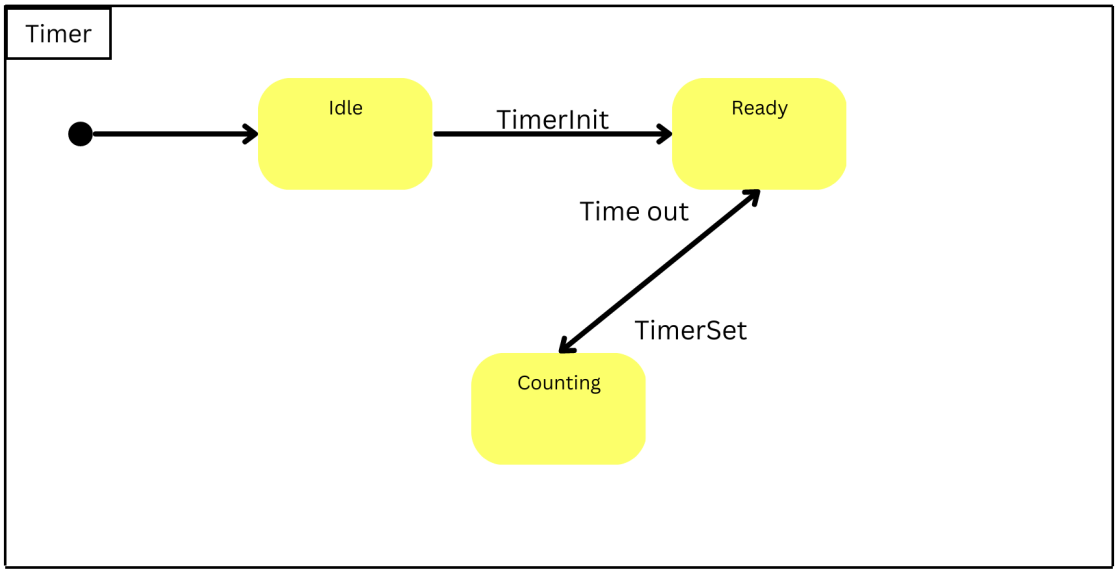
4) Calculate CPU load for the ECU
Hyper period= 20 ms.
CPU load= (4*speed task + 2*door task+ 1*Light switch task)/20

ECU 2:

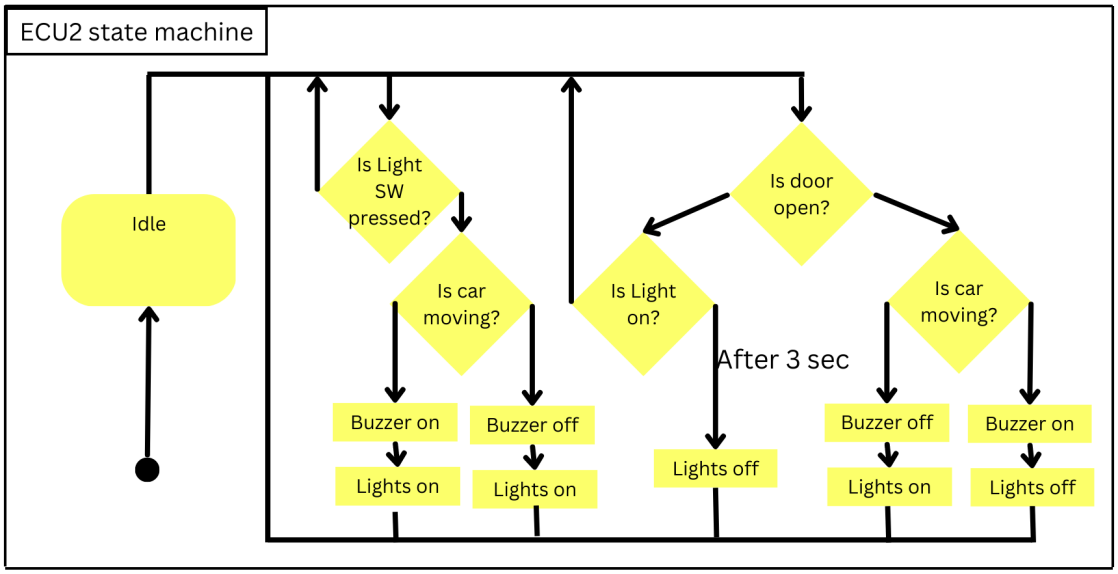
1) Draw a state machine diagram for each ECU component


CAN
Idle
CanInit
Active
CanSend
CanReceive

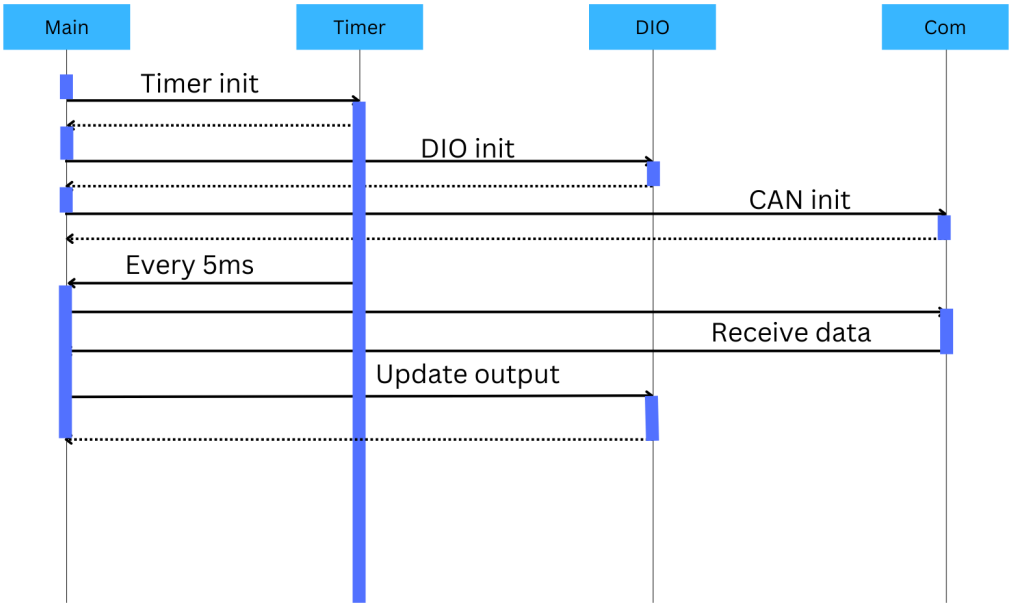

DIO
Idle
DioInit
Input
DioSet
DioInit
DioConfig
Output
DioGet



2) Draw a state machine diagram for the ECU operation



3) Draw the sequence diagram for the ECU



4) Calculate CPU load for the ECU
Hyper period = 5ms.
CPU load= Can receive task+ update output task / 5