Lecture 3: Computation Design Models and Concept for ES

2DT903, Embedded Systems – 5.0

Hemant Ghayvat

Department of Computer Science and Media Technology
hemant.ghayvat@lnu.se

Linnæus University

Non-Operational Quality Attributes

The important non-operational quality attributes are:

Testability & Debug-ability

Evolvability

Portability

Time-to-prototype and market

Per unit and total cost

Non-Operational Quality Attributes (continued)

Testability & Debug-ability

Testability deals with how easily one can test his/her design, application and by which means he/she can test it.

For an embedded product, testability is applicable to both the embedded hardware and firmware.

Embedded hardware testing ensures that the peripherals and the total hardware functions in the desired manner, whereas firmware testing ensures that the firmware is functioning in the expected way.

Debug-ability is a means of debugging the product as such for figuring out the probable sources that create unexpected behaviour in the total system.

Debug-ability has two aspects in the embedded system development context, namely, hardware debugging and firmware debugging.

Hardware debugging is used for figuring out the issues created by hardware problems whereas firmware debugging is employed to figure out the probable errors that appear as a result of flaws in the firmware.

Non-Operational Quality Attributes (continued)

Evolvability

For an embedded system, the quality attribute 'Evolvability' refers to the ease with which the embedded product (including firmware and hardware) can be modified to take advantage of new firmware or hardware technologies.

Assembly Lang
Embedded C

Non-Operational Quality Attributes (continued)

Portability

Portability is a measure of 'system independence'.

- An embedded product is said to be portable if the product is capable of functioning as such in various environments, target processors/controllers and embedded operating systems.
- A standard embedded product should always be flexible and portable.
- In embedded products, the term 'porting' represents the migration of the embedded software written for one target processor (e.g. Intel x86) to a different target processor (say Hitachi SH3 processor).
- Non-Operational Quality Attributes (continued)…………………..

Non-Operational Quality Attributes (continued)

Portability (continue)

If the firmware is written in a high-level language like ‘C’, it is very easy to port the firmware

- It has only few target processor-specific functions which can be replaced with the ones for the new target processor and re-compiling the program for the new target processor-specific settings.
- The program then needs to be re-compiled to generate the new target processor-specific machine code.
- If the firmware is written in Assembly Language for a particular family of processor (say x86 family), the portability is poor.
- It is very difficult to translate the assembly language instructions to the new target processor specific language.

Non-Operational Quality Attributes (continued)

Time-to-Prototype and Market

Time-to-market is the time elapsed between the conceptualisation of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial product).

Time to market in the embedded product market is highly competitive and reduction in time to market is a critical factor for the success of any commercial embedded product.

The more quickly a product product before you.

Rapid prototyping helps a lot in reducing time-to-market.

Prototyping is usually one phase of a product development in which the product ideas or concepts in a product under consideration are developed.

This phase proves to be faster than actual final product development since prototypes are not the scrutinised.

The time-to-prototype is also another critical factor.

If the prototypes are developed fast, the actual estimated development time can be brought down significantly.

In order to shorten the time to prototype, make use of all possible options like the use of off-the-shelf components, re-usable assets, etc.

Non-Operational Quality Attributes (continued)

Per Unit cost and Revenue

Computational Models in Embedded Design

The commonly used computational models in embedded system design are:

- Data Flow Graph Model
- Control Data Flow Graph Model
- State Machine Model
- Sequential Program Model
- Concurrent/Communicating Process Model

Data Flow Graph/Diagram (DFG) Model

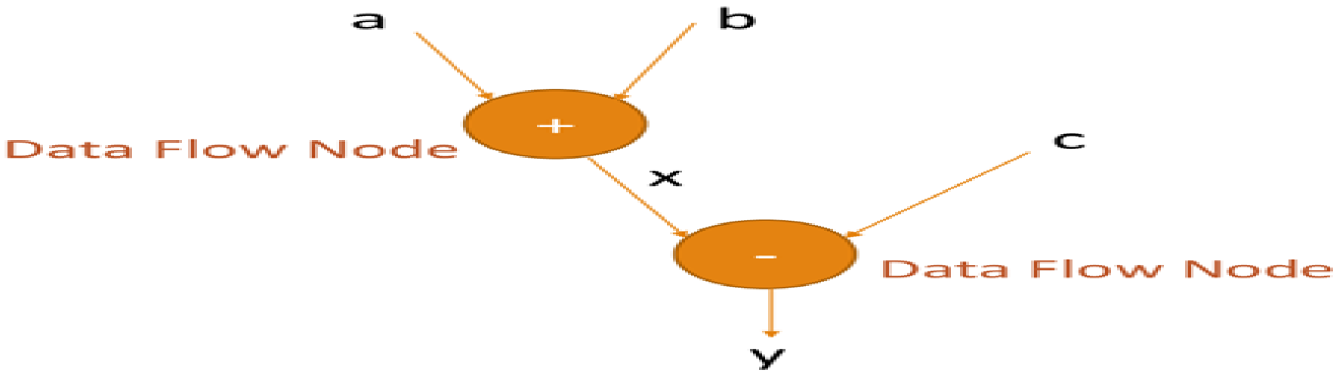
- The Data Flow Graph (DFG) model translates the data processing requirements into a data flow graph.
- It is a data driven model in which the program execution is determined by data.
- This model emphasizes on the data and operations on the data which transforms the input data to output data.
- Embedded applications which are computational intensive and data driven are modelled using the DFG model.
- **DSP** applications are typical examples for it

Data Flow Graph/Diagram (DFG) Model (continued)

Data Flow Graph (DFG) is a visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented by arrows.

An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation.

Suppose one of the functions in our application contains the computational requirement x = a * b and y = x + c.



Data Flow Graph/Diagram (DFG) Model (continued)

In a DFG model, a data path is the data flow path from input to output.

A DFG model is said to be acyclic DFG (ADFG) if it doesn't contain multiple values for the input variable and multiple output values for a given set of input(s).

Feedback inputs (Output is fed back to Input), events, etc. are examples for non-acyclic inputs.

A DFG model translates the program as a single sequential process execution.

Control Data Flow Graph/Diagram (CDFG) Model (continued)

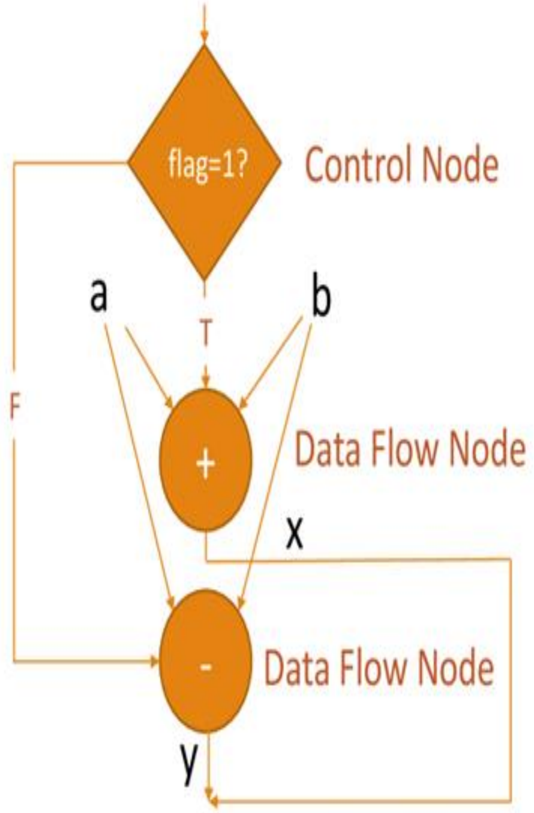
The Control Data Flow Graph/Diagram (CDFG) is a data-driven model in which the execution is controlled by conditional and non-conditional, the control operations (conditionals).

The Control DFG (CDFG) model is used for modelling applications involving conditionals or both data operations and control operations.

The CDFG uses Data Flow Graph (DFG) as element and conditional controls as decision flow nodes.

CDFG uses both data flow nodes and decision nodes, whereas DFG contains only data flow nodes.

Control Data Flow Graph/Diagram (CDFG) Model (continued)



Consider the implementation of the CDFG for the following requirement.

'If flag = 1, x = a + b; else y = a - b;'

The CDFG model for the same is given in the figure.

The control node is represented by a 'diamond' block which is the decision making element in a normal flow chart based design.

CDFG translates the requirement, which is modelled to a concurrent process model.

Control Data Flow Graph/Diagram (CDFG) Model (continued)

A real-world example of a CDFG in the embedded application using CDFG is capturing and saving of the image to a format set by the user in a digital still camera.

Here every action or data service required for generated analog signal to digital signal are controlled by the control nodes or buffer for the use of a media processor. The actions like focus, zoom, color auto correction, white balance adjusting, which perform various operations like image is stored (formats like JPEG, TIFF, BMP, etc.) is controlled by the camera settings, configured by the user.

State Machine Model

The State Machine Model is used for modelling **reactive or event-driven** embedded systems whose processing behaviour are dependent on state transitions.

Embedded systems used in the control and industrial applications are typical examples for event driven systems.

The State Machine model describes the system behaviour with **'States', 'Events', 'Actions' and 'Transitions**':

State is a representation of a current situation.

Event is an input to the state.

Action is the action that state system. **Action** is an activity to be performed by the state machine.

Transition is the movement from one state to another.

The **event** acts as stimuli for state transition.

Finite State Machine (FSM) Model

A Finite State Machine (FSM) model is one in which the number of states are finite.

The system is described using a finite number of possible states.

As an example, let us consider the design of an embedded system for driver/passenger car 'Seat Belt Warning' in an automotive using the FSM model.

The system requirements are captured as.

When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.

The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

Finite State Machine (FSM) Model (continued)

Here the States are

'Alarm Off'

'Waiting'

'Alarm On'

The events are

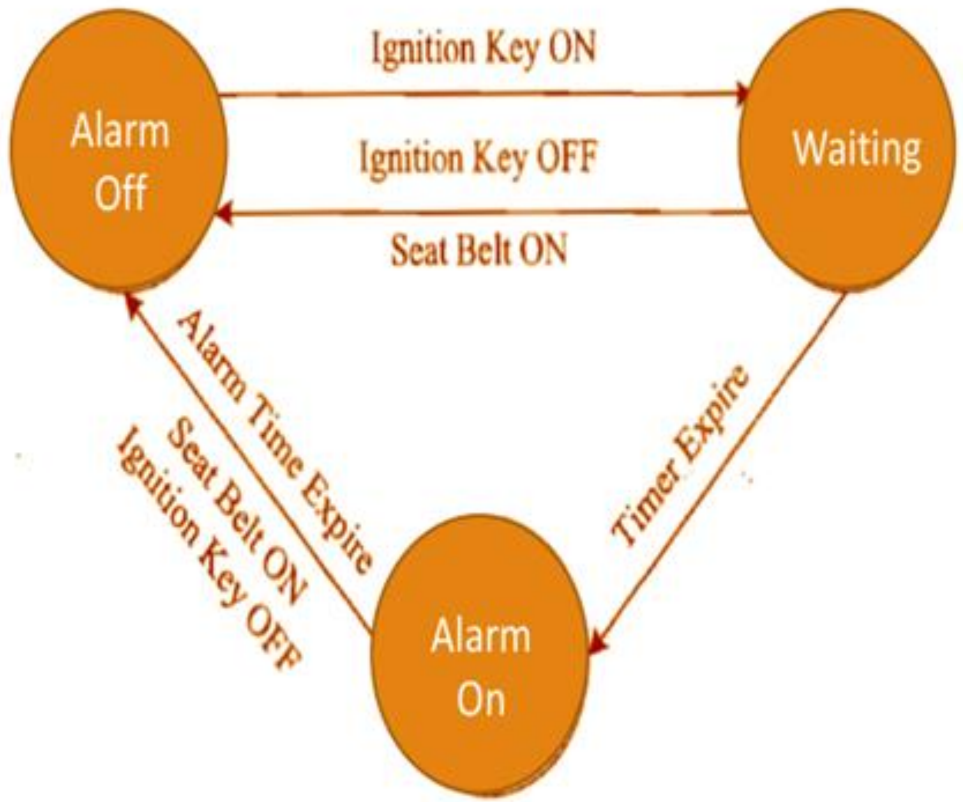
'Ignition Key ON'

'Ignition Key OFF'

'Timer Expire'

'Alarm Time Expire'

'Seat Belt ON'



FSM Model - Example 1

Design an automatic tea/coffee vending machine based on FSM model for the following requirement.

The tea/coffee vending is initiated by user inserting a 5 $ coin.

After inserting the coin, the user can either select 'Coffee' or 'Tea' or press 'Cancel' to cancel the order and take back the coin.

The FSM Model contains four states namely,

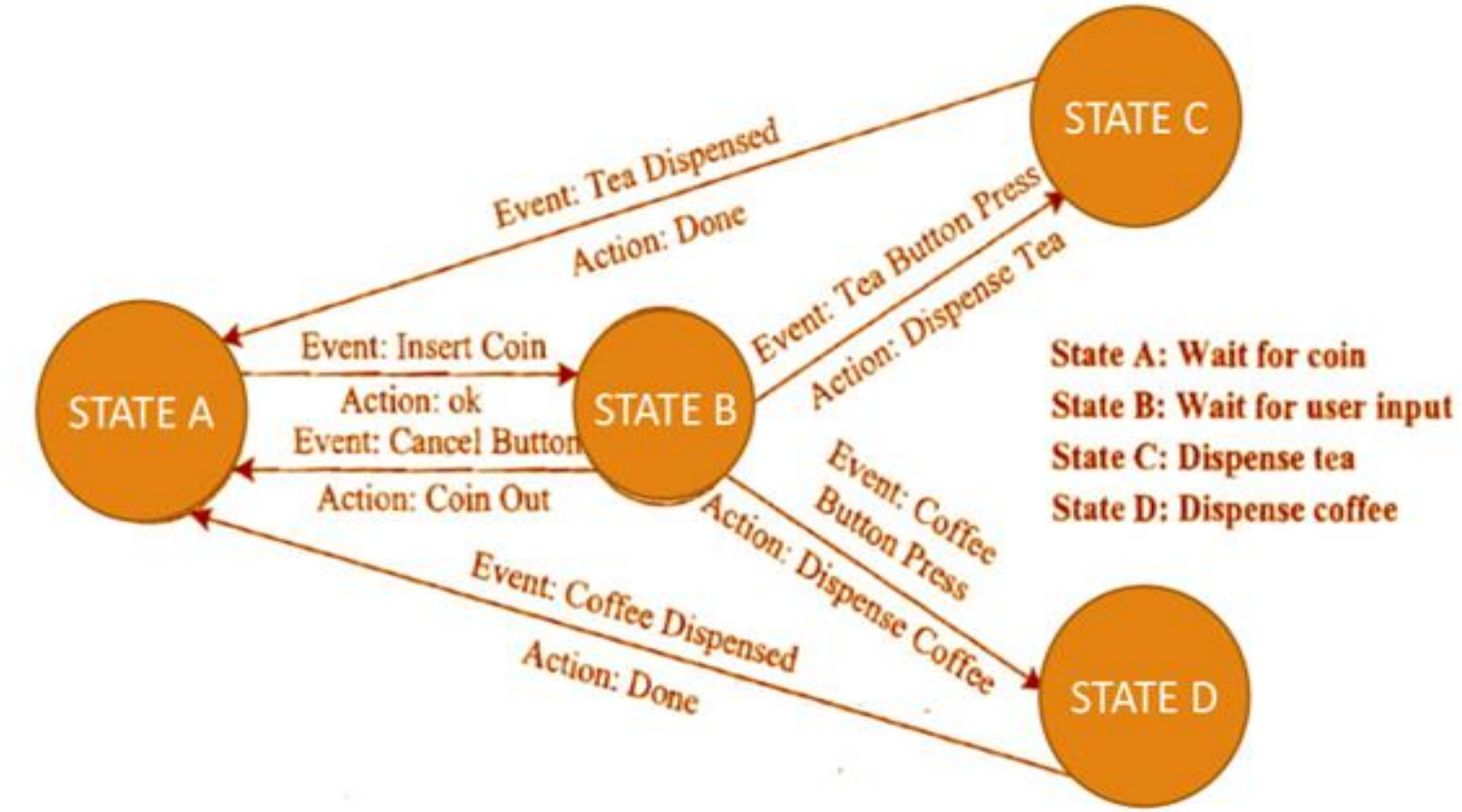
'Wait for coin'

'Wait for User Input'

'Dispense Tea'

'Dispense Coffee'

FSM Model - Example 1 (Coffee/Tea Vending Machine)



FSM Model - Example 1

FSM Model - Example 1 (continued)

The event 'Insert Coin' (5 rupee coin insertion), transitions the state to 'Wait for User Input'.

The system stays in this state until a further input is received from the buttons 'Cancel', 'Tea' or 'Coffee' (Tea and Coffee are the drink select button).

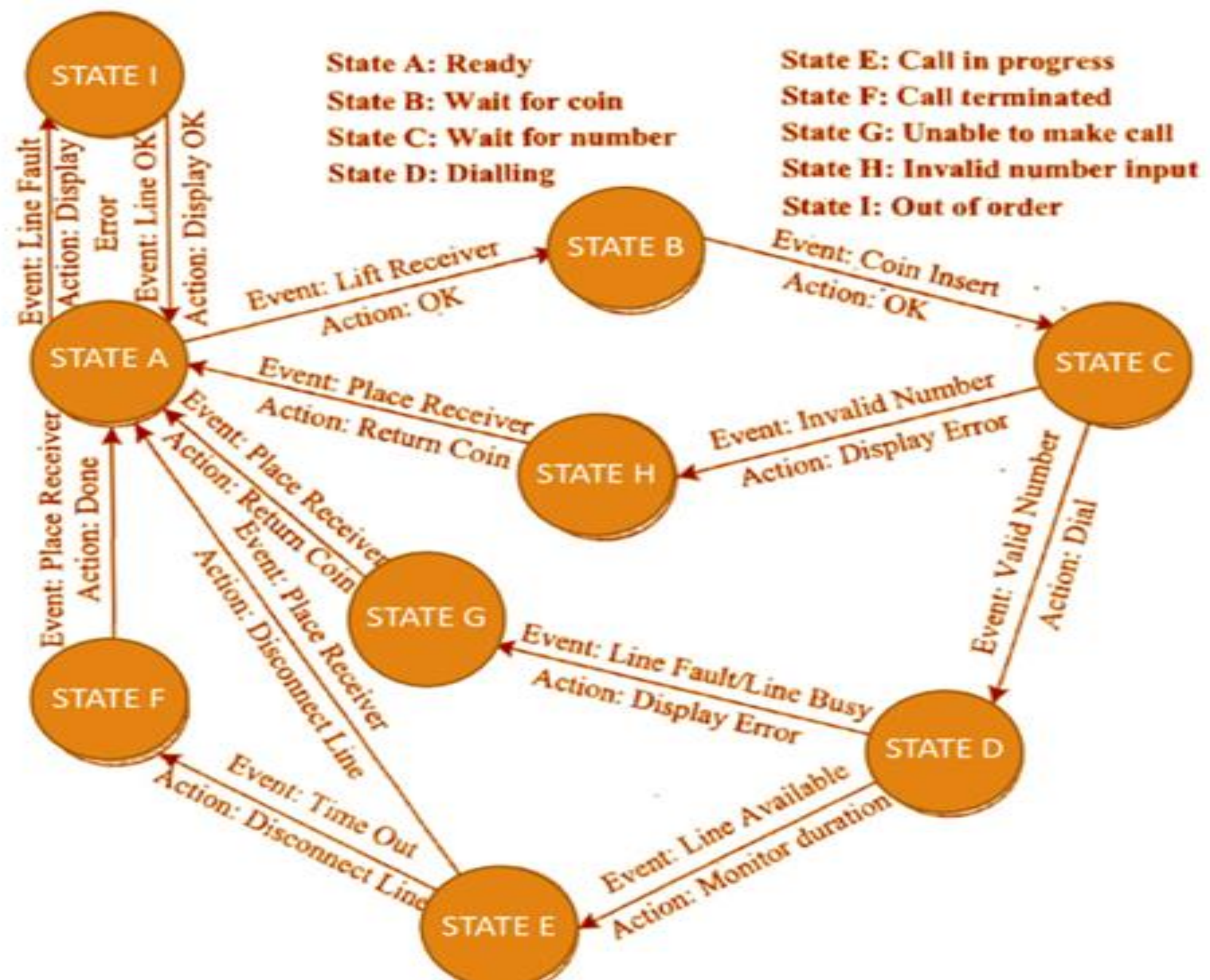
If the event triggered in 'Wait State' is 'Cancel' button press, the coin is pushed out and the state transitions to 'Wait for Coin'.

If the event received in the 'Wait State' is either 'Tea' button press, or 'Coffee' button press, the state changes to 'Dispense Tea' and 'Dispense Coffee' respectively.

Once the coffee/tea vending is over, the respective states transition back to the 'Wait for Coin' state.

FSM Model - Example 2 (Coin Operated Telephone Booth)

- Design a coin operated public telephone based on FSM model for the following requirements.
- The calling process is initiated by lifting the receiver (off-hook) of the telephone unit.
- After lifting the phone the user needs to insert a 1 Euro coin to make the call.
- If the line is busy, the coin is returned on placing the receiver back on the hook (on-hook).
- If the line is through, the user is allowed to talk till 60 seconds and at the end of 45th second, prompt for inserting another 1 euro coin for continuing the call is initiated.
- If the user doesn't insert another 1 euro coin, the call is terminated on completing the 60 seconds time slot.
- The system is ready to accept new call request when the receiver is placed back on the hook (on-hook).
- The system goes to the 'Out of Order' state when there is a line fault.


State A: Ready
State B: Wait for coin
State C: Wait for number
State D: Dialling
State E: Call in progress
State F: Call terminated
State G: Unable to make call
State H: Invalid number input
State I: Out of order
STATE I
STATE A
STATE B
STATE C
STATE D
STATE E
STATE F
STATE G
STATE H
Event: Line Fault
Action: Display Error
Event: Line OK
Action: Display OK
Event: Lift Receiver
Action: OK
Event: Coin Insert
Action: OK
Event: Place Receiver
Action: Return Coin
Event: Invalid Number
Action: Display Error
Event: Valid Number
Action: Dial
Event: Place Receiver
Action: Return Coin
Event: Line Fault/Line Busy
Action: Display Error
Event: Line Available
Action: Monitor duration
Event: Place Receiver
Action: Disconnect Line
Event: Time Out
Action: Disconnect Line
Event: Place Receiver
Action: Done

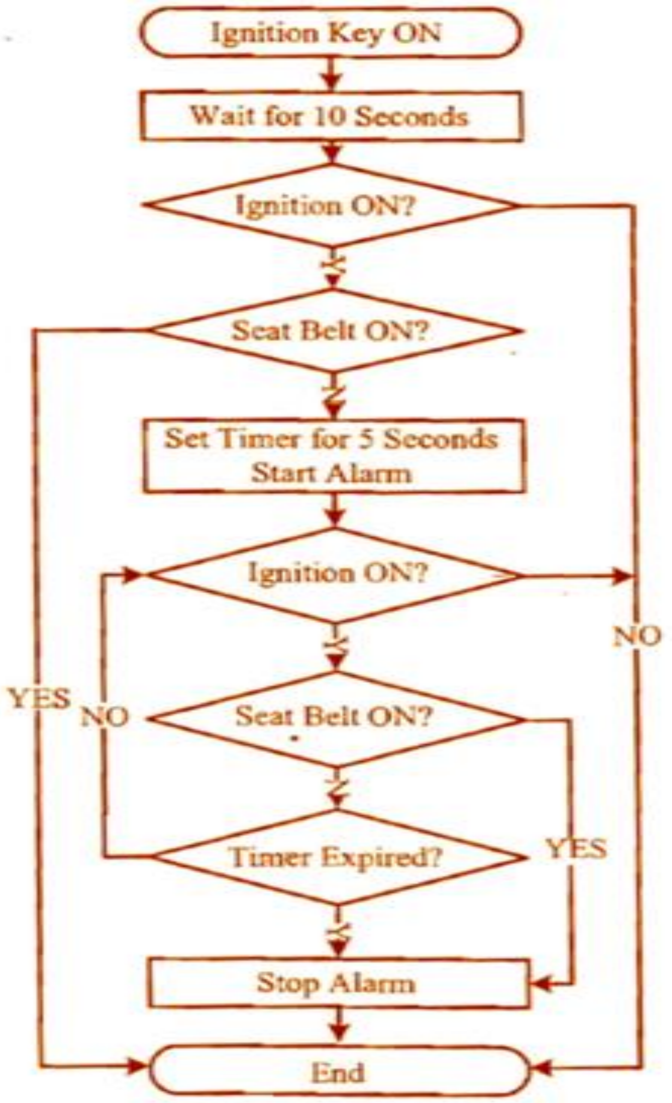
Sequential Program Model

- In the Sequential Program Model, the functions or processing requirements are executed in sequence.
- It is same as the conventional procedural programming.
- Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations.
- Finite State Machines (FSMs) and **Flow Charts** are used for modelling sequential programs.
- The FSM approach represents states, events, transitions and actions, whereas the Flow Chart models the execution flow.

Sequential Program Model (continued)

The execution of functions in a sequential program model for the 'Seat Belt Warning' system is illustrated below:

```
// Define ON
#define ON 1
// Define OFF
#define OFF 0
// ... [code snippet continues] ...
void main() {
  if ((ignition_key()==ON)){
    if ((check_seat_belt())==OFF)
      set_timer(5);
      wait_1_minute();
      if ((check_seat_belt())==OFF) && (ignition_key()==ON) && (time_expire()==NO)){
        // ... [code snippet continues] ...
      }
  }
}
```


Ignition Key ON
Wait for 10 Seconds
Ignition ON?
Y
Seat Belt ON?
N
Set Timer for 5 Seconds
Start Alarm
Ignition ON?
NO
Y
YES
NO
Seat Belt ON?
N
Timer Expired?
YES
Y
Stop Alarm
End

Concurrent/Communicating Process Model

The concurrent or communicating process model models concurrently executing tasks/processes.

It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution.

Sequential processing leads to a sequence task execution of task and thereby leads to poor performance since each sequential involves I/O waiting, sleeping for specified duration etc.

If the task is split into multiple subtasks, it is possible to tackle the CPU usage effectively by switching the task execution when the subtask under execution goes to a wait, sleep mode.

However, concurrent processing model requires additional overheads in task scheduling, synchronisation and communication.

Concurrent/Communicating Process Model (continued)

As an example, consider the implementation of the 'Seat Belt Warning' system using concurrent processing model.

Main task for the tasks initiation.

Timer task for waiting 10 seconds (wait timer task).

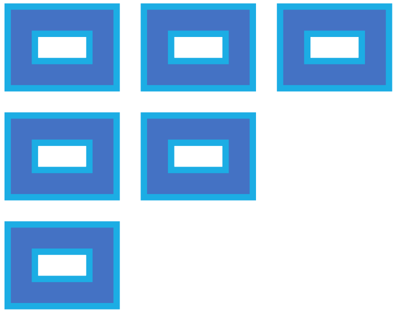
Task for checking the ignition key status (ignition key status monitoring task).

Task for checking the seat belt status (seat belt status monitoring task).

Task for starting and stopping the alarm (alarm control task).

Alarm timer task for waiting 5 seconds (alarm timer task).

Concurrent/Communicating Process Model (continued)



Definition of Petri Net

C = (P, T, I, O)

- Places
  $P = \{ p_1, p_2, p_3, \ldots, p_n\}$
- Transitions
  $T = \{ t_1, t_2, t_3, \ldots, t_n\}$
- Input
  $I : T \rightarrow P^r$ (r = number of places)
- Output
  $O : T \rightarrow P^q$ (q = number of places)

marking μ : assignment of tokens to the places of Petri net $\mu = \mu_1, \mu_2, \mu_3, \ldots \mu_n$

Applications of Petri Net

Petri net is primarily used for studying the dynamic concurrent behavior of network-based systems where there is a discrete flow.

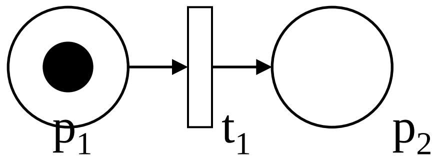
Petri Nets are applied in practice by industry, academia, and other places.

Basics of Petri Nets

Petri net consist two types of nodes: *places* and *transitions*. And arc exists only from a place to a transition or from a transition to a place.

A place may have zero or more *tokens*.

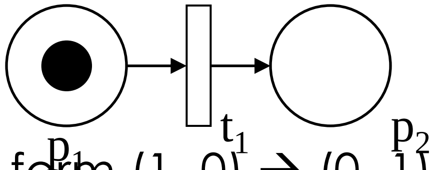
Graphically, places, transitions, arcs, and tokens are represented respectively by: circles, bars, arrows, and dots.



Basics of Petri Nets -continued

Below is an example Petri net with two places and one transaction.

Transition node is ready to ***fire*** if and only if there is at least one token at each of its input places



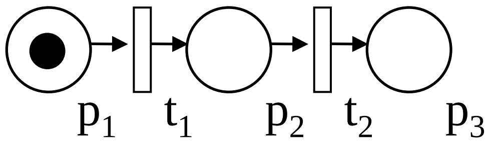
state transition of form (1, 0) → (0, 1)

p_1 : input place p_2: output place

Properties of Petri Nets

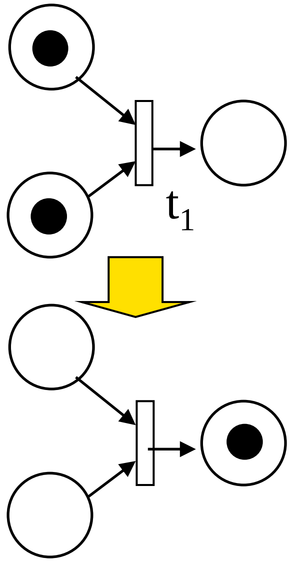
Sequential Execution

Transition t_2 can fire only after the firing of t_1. This impose the precedence of constraints "t_2 after t_1."



Synchronization

Transition t_1 will be enabled only when a token there are at least one token at each of its input places.



Merging

Happens when tokens from several places arrive for service at the same transition.

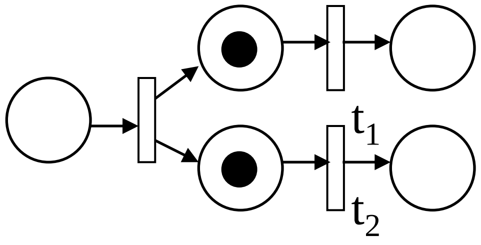
Linnæus University

Properties of Petri Nets –continued

Concurrency

t_1 and t_2 are concurrent.

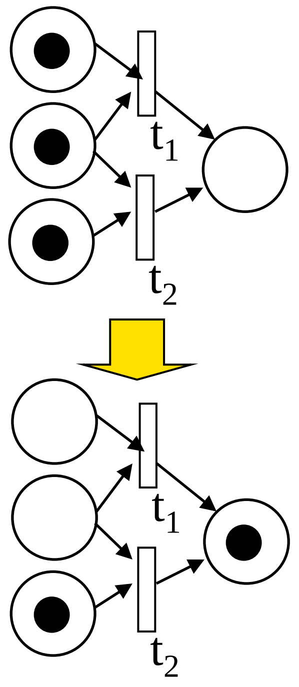
- with this property, Petri net is able to model systems of distributed control with multiple processes executing concurrently in time.



Properties of Petri Nets –continued

Conflict

t_1 and t_2 are both ready to fire but the firing of any leads to the disabling of the other transitions.

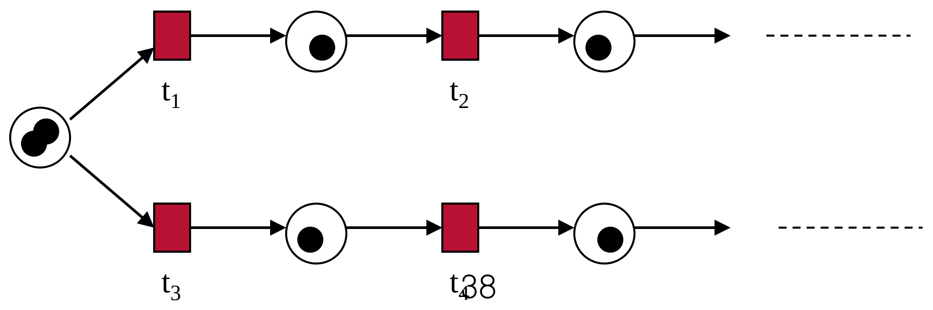


Properties of Petri Nets −continued

Conflict - continued

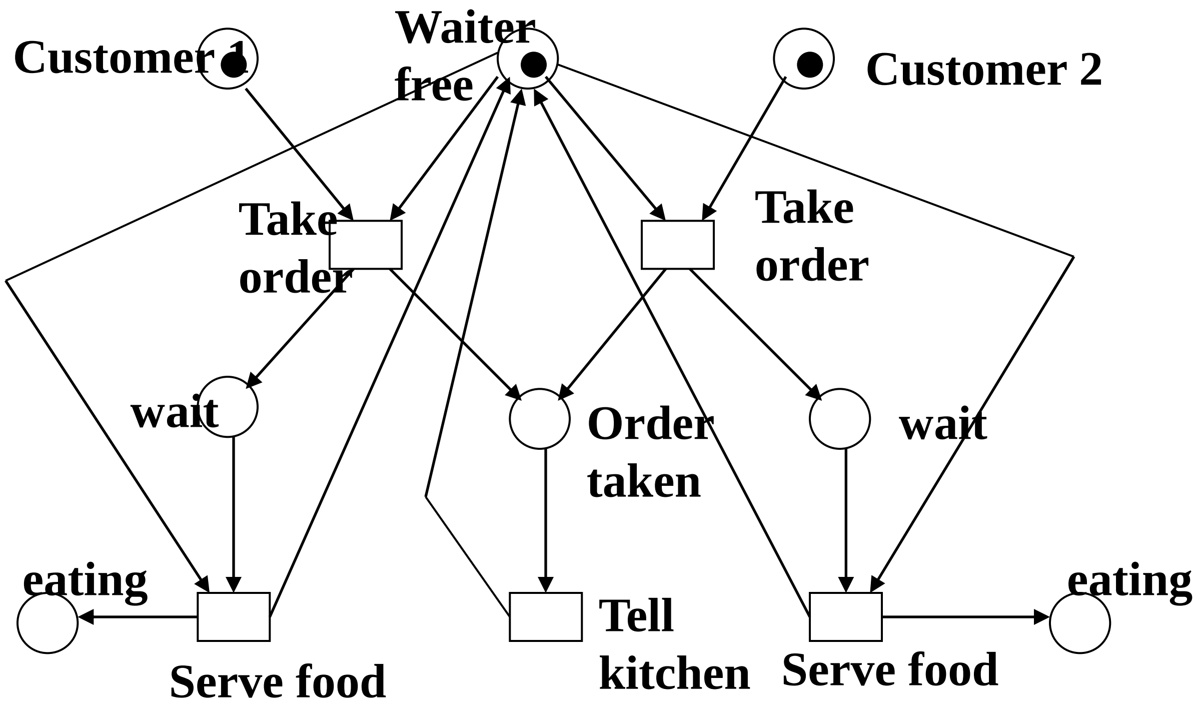
– the resulting conflict may be resolved in a purely non-deterministic way or in a probabilistic way, by assigning appropriate probabilities to the conflicting transitions.

there is a choice of either t_1 and t_2, or t_3 and t_4



Linnæus University

Example: In a Restaurant (A Petri Net)



Example: In a Restaurant (Two Scenarios)

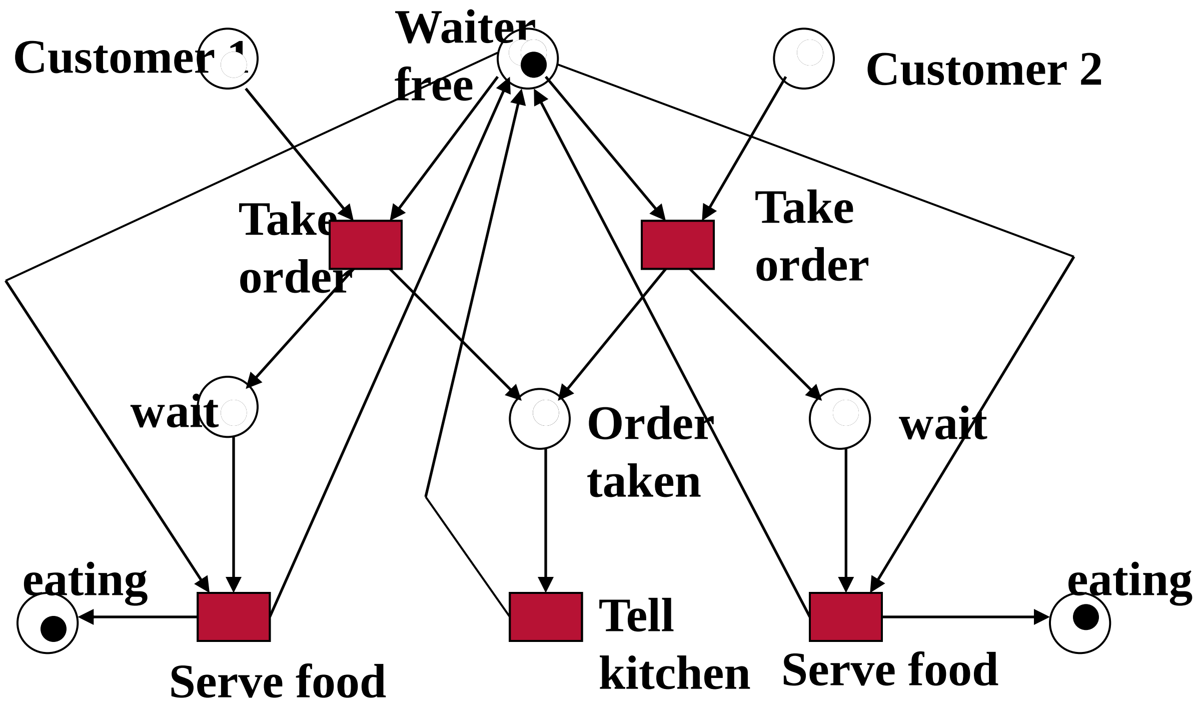
Scenario 1:

- Waiter takes order from customer 1; serves customer 1; takes order from customer 2; serves customer 2.

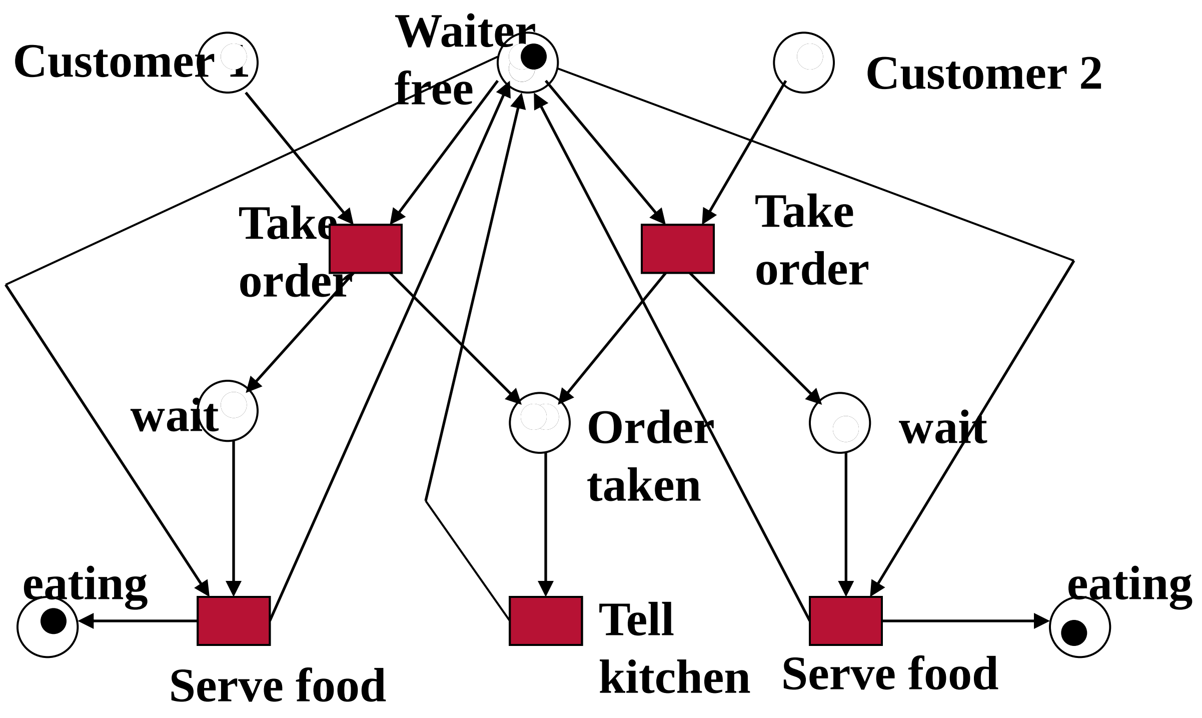
Scenario 2:

- Waiter takes order from customer 1; takes order from customer 2; serves customer 2; serves customer 1.

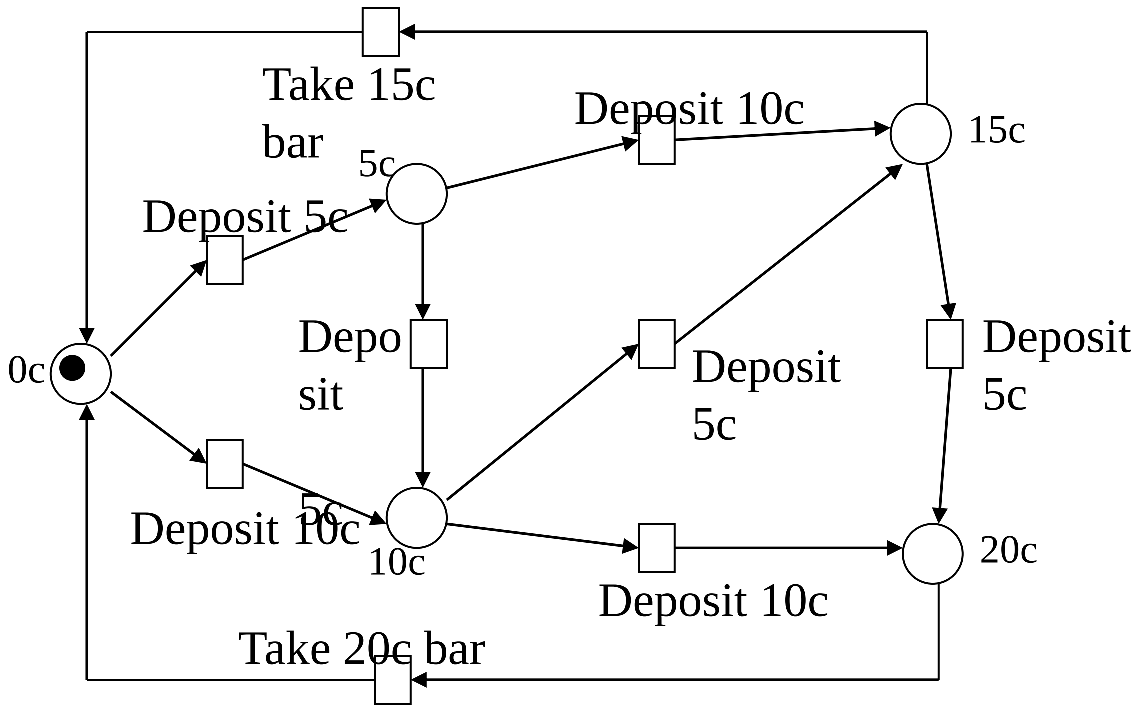
Example: In a Restaurant (Scenario 1)



Example: In a Restaurant (Scenario 2)



Example: Vending Machine (A Petri net)



Linnæus University

Example: Vending Machine (3 Scenarios)

Scenario 1:

– Deposit 5c, deposit 5c, deposit 5c, deposit 5c, take 20c snack bar.

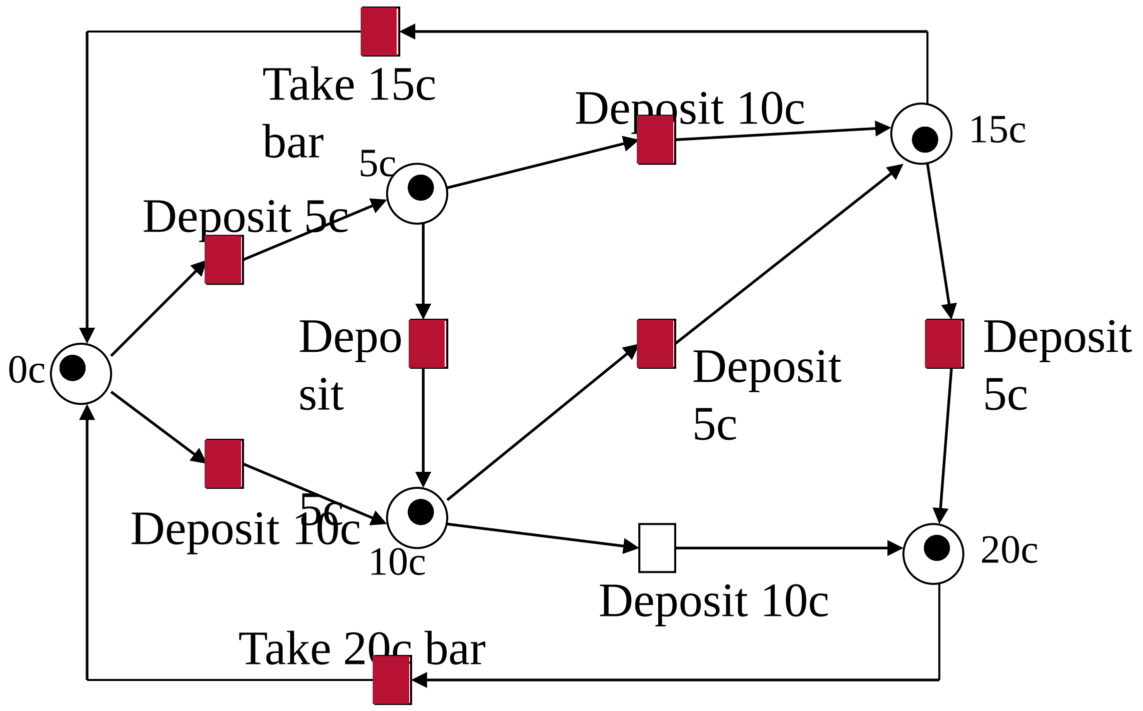
Scenario 2:

– Deposit 10c, deposit 5c, take 15c snack bar.

Scenario 3:

– Deposit 5c, deposit 10c, deposit 5c, take 20c snack bar.

Example: Vending Machine (Token Games)



Concurrent/Communicating Process Model (continued)

- The tasks cannot be executed randomly or sequentially.
- We need to synchronise their execution through some mechanism.
- For example, the alarm control task is executed only when the wait timer is expired and if the ignition key is in the ON state and seat belt is in the OFF state.
- We will use events to indicate these scenarios.
- The wait_timer_expire event is associated with the timer task event and will be in the rest state until it is initiated i.e when the timer expires.
- Similarly, events ignition_on and ignition_off are associated with the task ignition key status monitoring and the events seat_belt_on and seat_belt_off are associated with the seat belt status task.

Concurrent/Communicating Process Model (continued)

Create and initialize events

wait_timer_expire, ignition_on, ignition_off

seat_belt_on, seat_belt_off

alarm_timer_start, alarm_timer_expire

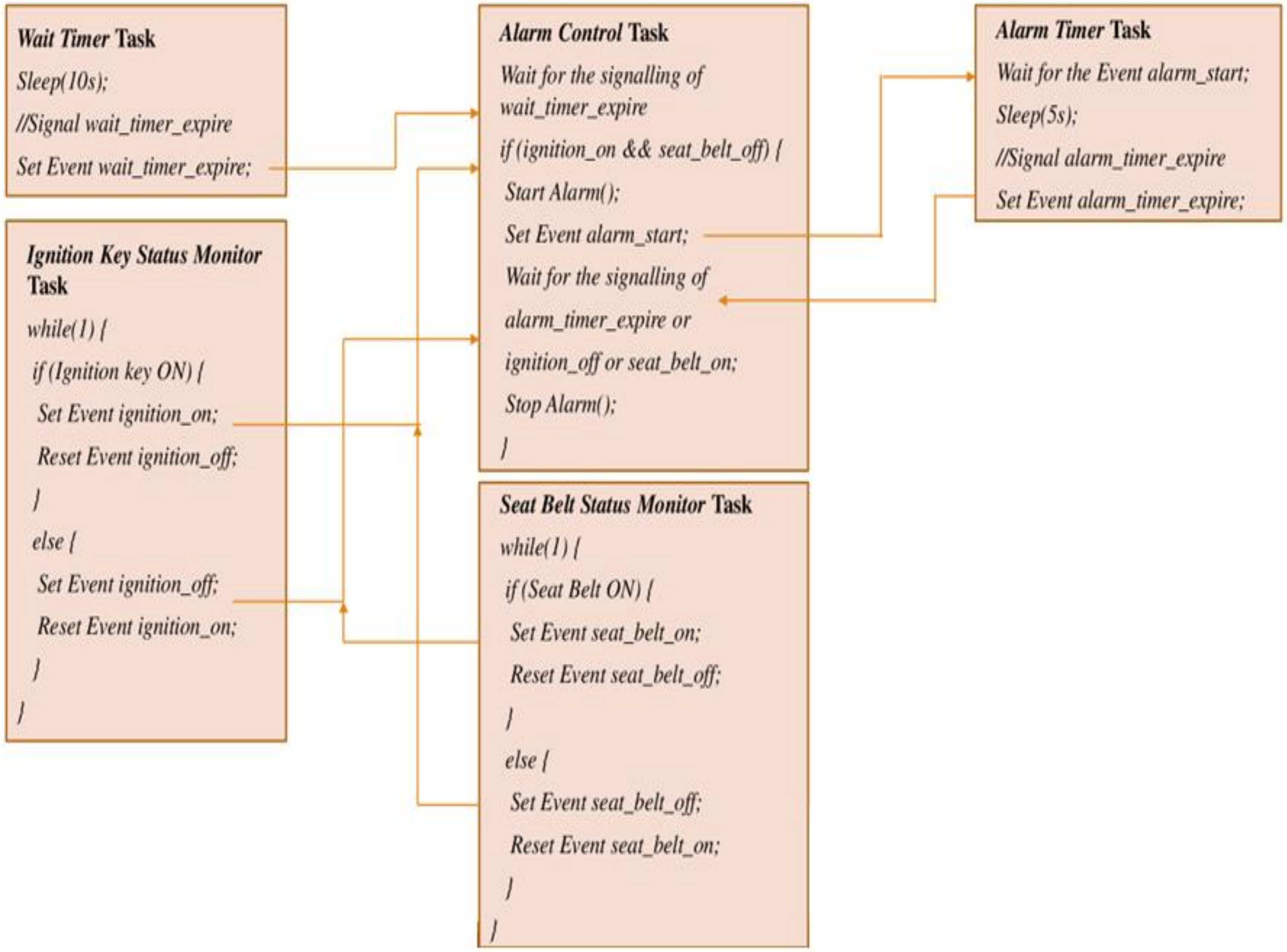
Create task Wait Timer

Create task Ignition Key Status Monitor

Create task Seat Belt Status Monitor

Create task Alarm Control

Create task Alarm Timer



```
Wait Timer Task
Sleep(10s);
//Signal wait_timer_expire
Set Event wait_timer_expire;
```

```
Ignition Key Status Monitor
Task
while(1) {
 if (Ignition key ON) {
  Set Event ignition_on;
  Reset Event ignition_off;
 }
 else {
  Set Event ignition_off;
  Reset Event ignition_on;
 }
}
```

```
Alarm Control Task
Wait for the signalling of
wait_timer_expire
if (ignition_on && seat_belt_off) {
 Start Alarm();
 Set Event alarm_start;
 Wait for the signalling of
 alarm_timer_expire or
 ignition_off or seat_belt_on;
 Stop Alarm();
}
```

```
Seat Belt Status Monitor Task
while(1) {
 if (Seat Belt ON) {
  Set Event seat_belt_on;
  Reset Event seat_belt_off;
 }
 else {
  Set Event seat_belt_off;
  Reset Event seat_belt_on;
 }
}
```

```
Alarm Timer Task
Wait for the Event alarm_start;
Sleep(5s);
//Signal alarm_timer_expire
Set Event alarm_timer_expire;
```

Introduction to Embedded Firmware Design

The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements.

Firmware is considered as the master **brain** of the embedded system.

Imparting intelligence to an Embedded system is a one time process and it can happen at any stage.

It can be immediately after the fabrication of the embedded hardware or at a later stage.

For most of the embedded products, the embedded firmware is stored at a permanent memory (ROM) and they are non-alterable by end users.

Some of the embedded products used in the Control and Instrumentation domain are adaptive.

Introduction to Embedded Firmware Design (continued)

Designing **embedded firmware requires understanding** of the particular embedded product hardware, like various component interfaces, memory map details, I/O port details, configuration and register details of various hardware chips used and also its programming language.

Embedded firmware development process starts with the conversion of the firmware requirements into a program model using modeling tools. Once the program model is created, the next step is the implementation of the tasks and actions by capturing the model using a language which is understandable by the target processor/controller.

Embedded Firmware Design Approaches

The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed, the speed of operation required, etc.

Two basic approaches are used for embedded firmware design:

Super Loop Based Approach (Conventional Procedural Based Design)

Embedded Operating System (OS) Based Approach

Super Loop Based Approach

The Super Loop based firmware development approach is adopted for applications that are not time critical and where the response time is not so important.

It is very similar to a conventional procedural programming where the code is executed task by task.

The task listed at the top of the program code is executed first and the tasks just below the top are executed after completing the first task.

In a multiple task based system, each task is executed in serial in this approach.

Super Loop Based Approach (continued)

- The firmware execution flow for this will be
- Configure the common parameters and perform initialization for various hardware components memory, registers, etc.
- Start the first task and execute it
- Execute the second task
- Execute the next task
- :
- :
- Execute the last defined task
- Jump back to the first task and follow the same flow

Super Loop Based Approach (continued)

The order in which the tasks to be executed are fixed and they are hard coded in the code itself.

Also the operation is an infinite loop-based approach.

We can visualize the operational sequence listed above in terms of a 'C' program code as

```
void main()
{
    Initialisations();
    while(1)
    {
        Task 1();
        Task 2();
        ...
        Task n();
    }
}
```

Super Loop Based Approach (continued)

Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation.

This repetition is achieved by using an infinite loop.

Hence the name 'Super loop based approach’.

The only way to come out of the loop is either a hardware **reset or an interrupt assertion**.

A **hardware reset** brings the program execution back to the **main loop**.

An interrupt request suspends the task execution temporarily and performs the corresponding interrupt routine and on completion of the interrupt routine it restarts the task execution from the point where it got interrupted.

Super Loop Based Approach (continued)

Advantage of Super Loop Based Approach:

It **doesn't** require an operating system

There is **no need for scheduling** which task is to be executed and assigning priority to each task.

The **priorities are fixed** and the order in which the tasks to be executed are also fixed.

Hence the code for performing these tasks will be residing in the code **memory without an operating system image**.

Super Loop Based Approach (continued)

Applications of Super Loop Based Approach:

This type of design is deployed in **low-cost embedded products** and products where **response time is not time critical**.

Some embedded products demands this type of approach if some tasks itself are **sequential**.

For example, reading/writing data to and from a card using a card reader requires a sequence of operations like checking the presence of card, authenticating the operation, reading/writing, etc.

It should strictly follow a specified sequence and the combination of these series of tasks constitutes a single task-namely data read/write.

Super Loop Based Approach (continued)

A typical example of a ‘Super Loop based’ product is an electronic video game toy containing a simple display unit.

The program runs continuously and doesn't pause unless designed in such a way that it reads the keys to interact with the user (e.g., every time it reads the keys to check whether the user has given any input and if any key press is detected the graphic display is updated).

The keyboard scanning and display updating happens at a reasonably high rate.

Even if the application misses a key press, it won't create any critical issues; therefore, it will be treated as bugs in the firmware.

It is not economical to embed an OS into low-cost products and it is an utter waste to do so if the response requirements are not crucial.

Super Loop Based Approach (continued)

Drawbacks of Super Loop Based Approach:

Any failure in any part of a single task will affect the total system.

If the program hangs up at some point while executing a task, it will remain there forever until watchdog timers (WDTs) step's function.

Watchdog Timers provide continuous coverage against this, but this, in turn, may cause additional hardware cost and firmware overheads.

Lack of real-time awareness.

If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases.

This brings the probability of missing out some events.

For example, in a system with keypads, in order to identify the key press, you may have to press the keys for a sufficient long time till the keypad status monitoring task is executed internally by the firmware.

Interrupts can be used for external events requiring real-time attention.

Embedded Operating System (OS) Based Approach

The Embedded Operating System (OS) based approach contains operating systems, which can be either a General Purpose Operating System (GPOS) or a Real Time Operating System (RTOS) to host the user written application firmware.

The General Purpose Operating System (GPOS) is very similar to a conventional PC based operating system (such as those running on Microsoft Windows XP, Linux/Unix etc.) for Desktop PCs) and you will be creating and running user applications on top of it.

Embedded Operating System (OS) Based Approach (continued)

Example of a GPOS in embedded product development is Microsoft Windows XP Embedded.

Example of Embedded products using Microsoft Windows XP OS are Personal Digital Assistants (PDAs), Hand held devices/Portable devices at the Point of Sale (POS) terminals.

Use of GPOS in embedded products merges the demarcation of Embedded Systems and general computing systems in terms of OS.

For developing applications on top of the GPOS, the OS supported APIs are used.

Similar to the 'device drivers' used to create drivers on the board to communicate with 'driver software' for different hardware present on the board application also require them.

Embedded Operating System (OS) Based Approach (continued)

- Real Time Operating System (RTOS) based design approach is employed in embedded products demanding Real-time response.
- RTOS responds in a timely and predictable manner to events.
- Real Time operating system contains a Real Time kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads, etc.
- A Real Time Operating System (RTOS) allows flexible scheduling of system resources like the ‘CPU and memory’ and offers more way to communicate between tasks.
- 'Windows CE', 'VxWorks', 'ThreadX', 'Linux/RTOS' (e.g., ‘Embedded Linux’, 'Symbian' etc. are examples of RTOS employed in embedded product development.
- Mobile phones, PDAs (based on Windows CE), VxWorks Mobile Platforms), handheld devices, etc. are examples of 'Embedded Products' based on RTOS.
- Most of the mobile phones are firmware on the popular RTOS 'Symbian’. (sic)

- Ingredients: Raw data and instructions.
- Recipe: High-level programming language.
- Cooking Utensils: Hardware components.
- Assembly Language: Simplified cooking instructions directly understandable by hardware.

Advantages of Assembly Language Based Development (continued)

Low Level Hardware Access

Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers, and low level interrupt routines, etc. are making use of direct assembly coding since low level device specific operation support is not commonly available with most of the high-level language cross compilers.

Code Reverse Engineering

Reverse engineering is the process of understanding the technology behind a product by extracting the information from a finished product.

'Reverse engineering is performed by 'hackers' to reveal the technology behind 'Proprietary Products'.

Through the use of the disassembler program and decompiler, if it may be possible to break mostly memory protection and read the code memory, it can easily be converted into assembly code or a dis-assembly code form for the target machine.

Drawbacks of Assembly Language Based Development

Embedded Firmware Development Languages

For embedded firmware development, we can use either assembly language or high level language (Generically known as 'target processor/controller specific language')

Assembly language for lower level (fundamental) tasks.

Contemporary / Conventional languages for higher level tasks (Like C, C++, JAVA, etc.

A good mix of assembly and high-level language or

a combination of Assembly and High level Language.

Assembly Language Based Development

'Assembly language' is the human readable notation of 'machine language'

'Machine language' is a processor dependent language.

Machine language is a binary representation and it consists of 1s and 0s.

Machine language is made readable by using specific symbols called 'mnemonics'.

Hence machine language can be considered as an interface between processor and programmer.

Assembly language and machine languages are processor/controller dependent and an **assembly program written for one processor/controller family will not work with others**.

Assembly language programming is the task of writing processor specific machine code in mnemonic form, converting these mnemonics into actual processor instructions (machine language) and associated data using an assembler.

Assembly Language Based Development (continued)

Assembly Language program was the most common type of programming adopted in the beginning of software revolution.

Even today also almost all low level, system related, programming is carried out using assembly language.

In particular, assembly language is often used in writing the low level interaction between the operating system and the hardware, for instance in device drivers.

Assembly Language Based Development (continued)

The general format of an assembly language instruction is an Opcode followed by Operands.

The Opcode tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the opcode.

For example

MOV A, #30

Here MOV A is the Opcode and #30 is the operand

The same instruction when written in machine language will look like

0111000 00011110

where the first 8-bit binary value 01110010 represents the opcode MOV A and the second 8-bit binary value 00011110 represents the operand 30.

Assembly Language Based Development (continued)

Each line of an assembly language program is split into four fields as given below

LABEL OPCODE OPERAND COMMENTS

A 'LABEL' is an optional identifier used extensively in programs to reduce the reliance on programmers for remembering where data or code is located.

For example

DELAY (Label) MOV R0 (Opcode), #255 ; Load Register R0 with 255

Assembly Language Based Development (continued)

The Assembly language program written in assembly code is saved as .asm (Assembly file) file or an .src (source file) (also .s file).

Any text editor like 'Notepad' or 'Wordpad' from Microsoft or the text editor provided by an Integrated Development (IDE) tool can be used for writing the assembly instructions.

Similar to 'C' and other high level language programming, we can have multiple source files called modules in assembly language programming.

Each module is represented by an ‘asm’ or ‘s’ file.

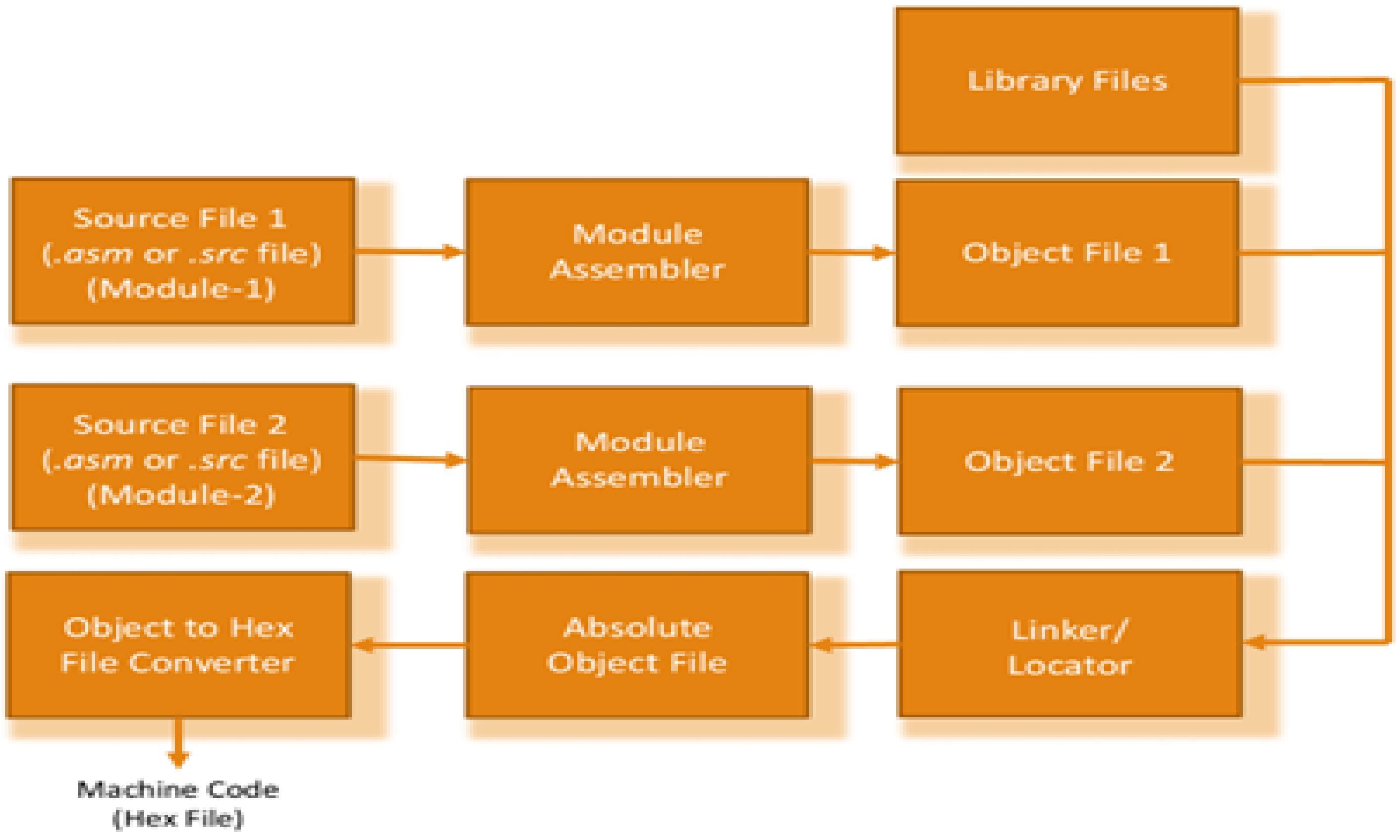
This approach is known as 'Modular Programming'.

Modular programming is employed when the program is too complex or too big.

In 'Modular Programming', the entire code is divided into submodules and each module is made re-usable.

Modular programs are usually easy to code, debug and alter.

Assembly to machine code conversion



Drawbacks of Assembly Language Based Development

High Development Time

Assembly language is much harder to program than high level languages.

The developer must pay attention to more details and must have thorough knowledge of the architecture, memory organisation and registered details of the target processor unit.

Learning the inner details of the processor and its assembly instructions is highly time consuming and it creates a delay impact in product development.

Also modules in an assembly code are required for performing an action which can be done with a single instruction in a high-level language like 'C'.

Drawbacks of Assembly Language Based Development (continued)

- Developer Dependency
- Unlike high level languages, there is no common written rule for developing assembly language based applications.
- Assembly language programmers, developers will have the freedom to choose the different methodology according to the development task depending on his/her own agreement.
- Also the memory space, register space and the resource platform can be achieved differently from each other.
- If the approach taken is assembly, then documented properly at the development stage, when the developer quits or is replaced, the approach is followed at a later stage or when a new developer needs to take over the code, he/she may not be able to understand the code or approach that was done by the previous programmer.
- If he/she needs it to be done differently modifying it on a later stage is very difficult.

References

Shibu K V, “Introduction to Embedded Systems”, Tata McGraw Hill, 2009.

Raj Kamal, “Embedded Systems: Architecture and Programming”, Tata McGraw Hill, 2008.
Shrishail Bhat, Dept. of ECE, AIITM Bhalki

Additional Reading Material:

Smith, J., & Bhat, S. (2015). "Secure Bootloader Design for Embedded Systems." IEEE Transactions on Dependable and Secure Computing, 12(4), 398-410.

Gupta, A., Kamal, R., & Patel, H. (2017). "Machine Learning Applications in Embedded Systems: A Survey." IEEE Transactions on Emerging Topics in Computing, 5(3), 398-410.

Kumar, S., & Shibu, K. V. (2019). "Energy-Efficient Task Scheduling Algorithms for Multi-Core Embedded Systems." IEEE Transactions on Computers, 68(8), 1189-1202.

Bhat, S., & Singh, A. (2016). "Wireless Sensor Networks for Environmental Monitoring: A Survey." ACM Transactions on Sensor Networks, 13(2), Article 15.

Patel, R., & Kamal, R. (2018). "Design Space Exploration Techniques for Embedded System Architectures." ACM Transactions on Embedded Computing Systems, 17(3), Article 78.

Jain, S., & Bhat, S. (2020). "Fault Diagnosis Techniques in Real-Time Embedded Systems: A Comprehensive Review." ACM Transactions on Design Automation of Electronic Systems, 25(4), Article 40.

Lnu.se