EECE 276
Embedded Systems

Procedural design techniques
Partitioning - Modularization

- To achieve high cohesion/low coupling
  1. Difficult design decisions and things likely to change are captured
  2. Modules are designed to *hide* the design decision/changing feature from the rest of the system.

- Module: information hiding device
  » Has an “abstract interface” that abstracts details
  » Implementation is hidden inside the module.

- In real-time systems:
  » Modules as abstractions for hardware components

- Top-down and bottom-up approaches
Structured Design

- From SA to SD: Dataflow Diagrams (DFD) capture essential data processing functions.
- DFD-s are hierarchical – DFD0 is derived from the context diagram
- DFD-s have:
  - Data- and control flows
  - Sources and sinks
  - Data stores
  - Processes (with PSPECs)
- Data dictionary:
  Description of all relevant data elements
  (Type, name, alias, description, location on DFD)
DFD Example
Context Diagram for IMU
DFDO for IMU

Diagram of Data Flow Diagram (DFD) for IMU (Inertial Measurement Unit).

1. Temperature, Other Physical Data
   - 10-ms Interrupt
   - Δx, Δy, Δz

2. Process Accelerometer Data
   - Attitude Data
   - True Acceleration, Velocity, and Position Information

3. Store and Transmit Data
   - Display Information
   - True Acceleration, Velocity, and Position Information
   - Temperature, Other Physical Data

4. Diagnostics
   - 1000-ms Interrupt
   - Torquing Pulses (gx, gy, gz)
   - Attitude Data
   - 40a-ms Interrupt

5. Diagnostics
   - 40b-ms Interrupt

6. Diagnostics
   - Display Information
   - True Acceleration, Velocity, and Position Information
   - Temperature, Other Physical Data

7. Diagnostics

8. Diagnostics

9. Diagnostics

10. Diagnostics

11. Diagnostics

12. Diagnostics

13. Diagnostics

14. Diagnostics

15. Diagnostics

16. Diagnostics

17. Diagnostics

18. Diagnostics

19. Diagnostics

20. Diagnostics

21. Diagnostics

22. Diagnostics

23. Diagnostics

24. Diagnostics

25. Diagnostics

26. Diagnostics

27. Diagnostics

28. Diagnostics

29. Diagnostics

30. Diagnostics

31. Diagnostics

32. Diagnostics

33. Diagnostics

34. Diagnostics

35. Diagnostics

36. Diagnostics

37. Diagnostics

38. Diagnostics

39. Diagnostics

40. Diagnostics

41. Diagnostics

42. Diagnostics

43. Diagnostics

44. Diagnostics

45. Diagnostics

46. Diagnostics

47. Diagnostics

48. Diagnostics

49. Diagnostics

50. Diagnostics

51. Diagnostics

52. Diagnostics

53. Diagnostics

54. Diagnostics

55. Diagnostics

56. Diagnostics

57. Diagnostics

58. Diagnostics

59. Diagnostics

60. Diagnostics

61. Diagnostics

62. Diagnostics

63. Diagnostics

64. Diagnostics

65. Diagnostics

66. Diagnostics
DFD2 for IMU

DFD 2: Process Accelerometer Data

Temperature, Other Physical Data

Compensate Accelerometer Data

True Acceleration, Velocity, and Position Information

True Acceleration, Velocity, and Position Information

Errors

Diagnostic Information

Errors

Errors
SD in real-time systems

- Concurrency? Events? Time?
- SASD extension with control model
- Explicit process activation
- CFD = FSM
- CSPEC: logic table
Design using FSM-s

FSM tables lend themselves to procedural implementation:

typedef states: (state 1,...,state n); {n is# of states}
    alphabet: (input 1,...,input n);
    table_row: array [1..n] of states;
procedure move_forward; {advances FSM one state}
    var
        state: states;
        input: alphabet;
        table: array [1..m] of table_row; {m is the size of the alphabet}
    begin
        repeat
            get(input); {read one token from input stream}
            state:=table[ord(input)] [state]; {next state}
            execute_process (state);
            until input = EOF;
    end;

<table>
<thead>
<tr>
<th></th>
<th>MA</th>
<th>LO</th>
<th>TD</th>
<th>MC</th>
<th>EE</th>
<th>ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAK</td>
<td>NAV</td>
<td>TAK</td>
<td>TAK</td>
<td>TAK</td>
<td>TAK</td>
<td>TAK</td>
</tr>
<tr>
<td>NAV</td>
<td>NAV</td>
<td>NAE</td>
<td>NAA</td>
<td>LAN</td>
<td>NAV</td>
<td>NAV</td>
</tr>
<tr>
<td>NAE</td>
<td>NAE</td>
<td>NAE</td>
<td>NAE</td>
<td>NAE</td>
<td>NAA</td>
<td>NAE</td>
</tr>
<tr>
<td>NAA</td>
<td>NAA</td>
<td>NAE</td>
<td>NAA</td>
<td>NAA</td>
<td>NAA</td>
<td>NAV</td>
</tr>
<tr>
<td>LAN</td>
<td>LAN</td>
<td>LAN</td>
<td>LAN</td>
<td>LAN</td>
<td>LAN</td>
<td>LAN</td>
</tr>
</tbody>
</table>

Procedure execute_process (state: states);
begin
    case state of
        state 1: process 1; {execute process 1}
        state 2: process 2; {execute process 2}
        ...
        state n: process n; {execute process n}
    end
Example: Problem

- Design a process controller for an environment monitoring system that collects data from a set of air-quality sensors situated around a city. There are 100 sensors located in 5 neighborhoods. Each sensor must be polled four times per second. When more than 40% of the sensors in a particular neighborhood indicate that the air quality is below an acceptable level, local warning lights are activated. All sensor readings are used to generate a report every 15 mins that is sent to a report file.

![Diagram of Environmental Monitoring System]

Note: N = 1..5, M = 1..20
Example: DFD
Example: Explanation

- Each neighborhood (1..5) is monitored by a **ScanSensors** process. 100 sensors over 5 neighborhoods means 20 sensors/area, each of which must be scanned 4 times per second, thus the **ScanSensors** processes must be triggered at every 1/80 seconds (i.e. 80 Hz). The **ScanSensors** processes generate a **Rdy** event when they read all the sensors in the neighborhood, and they deposit all sensor data in the **ScanData** stores (one for each area). The **Rdy** events are used to release the corresponding **Ck_Area** process that checks if more than 40% of the sensors read “high” values, and if so they turn the corresponding light ON. If not, and the light was on in the previous run, they turn the light OFF. The **Log** process is triggered every 15 mins, and it reads the contents of the **ScanData** stores, and produces a report.