University of York

Embedded Systems Design & Implementation

Open Individual Assessment

Open Assessment 1

Examination number:

Y3606797

Contents

1 Part 1 - Theory

1.1 Question 1

We can determine the rate X of actor H by producing a set of simultaneous equations from Table 1 and the provided Synchronous Dataflow model.

The topology matrix for the SDF model is as follows:

$$\Gamma = \begin{bmatrix} 2 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

This gives us the following simultaneous equations:

$$\begin{array}{lll} 2A - 2E = 0 & 2B - 2E = 0 & 2C - 2E = 0 \\ 2D - 2E = 0 & 2E - 6F = 0 & F - I = 0 \\ 3G - 2H = 0 & XH - 3I = 0 & I - G = 0 \end{array}$$

Using these equations I determined that $X = 2$.
Similarly, I determined the firing frequencies of the remaining actors:

$$q = \begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{array} \begin{pmatrix} 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 2 \\ 2 \\ 3 \\ 2 \end{pmatrix}$$

1.2 Question 2

Using the firing frequencies determined in Question 1, I was able to identify the following PASS schedule:

a.fire(3); b.fire(3); c.fire(3); d.fire(3); e.fire(3); f.fire(1); a.fire(3); b.fire(3); c.fire(3); d.fire(3); e.fire(3); f.fire(1); g.fire(2); h.fire(3); i.fire(2);

The maximum required FIFO buffer size is 6 as required and the number of firings of the actors match up with their frequencies in the vector q (Question 1).

1.3 Question 3

For my chosen PASS schedule the number of tokens that must be initially stored in the buffer of the feedback channel c9 is 2.

2 Part 2 - WSN MAC layer protocol

2.1 Question 1

For this task I opted to dedicate an entire class (SourceNodeActor) to perform the PtolemyII actor functions needed in the simulation. The class SourceNodeActor relies on another class (SourceNode) which provides the implementation of the protocol features, this in turn relies on another class SinkNodeModel which is responsible for modelling the sink nodes parameters and ultimately synchronising with their reception phases.

An instance of SourceNode is created with a number of channels provided to the constructor (SourceNodeActor, initialise, 62). This design was taken so that I could reuse the code during the second part of this question.

When the actor is initialised it begins reading beacons from a sink node channel for a specified amount of time, given by the constant DEFAULT_LISTENING_TIME (SourceNode, 12), when we read a beacon we extend the time to try get another and thus calculate the sink parameters. If the time expires we switch nodes and

do the same again (SourceNode, registerNextFire, 103-104). This is how I sync to multiple nodes effectively.

When n=1 we have to wait an entire protocol length to read another beacon, so we can switch away and come back later (SourceNode, readBeacon, 65). We then use the length between the iterations to calculate the sink parameters (SinkNodeModel, calcTForNEqual1, 116-120) and (SinkNodeModel, calcN, 103-110).

2.2 Question 2

For this task I created a class called MySourceNode, this class represents the mote functionality and is to be loaded onto a physical mote to implement the WSN MAC layer protocol as described.

This class reuses the original SourceNode and SinkNodeModel classes to again determine sink parameters and firing times for correct functionality. Thanks to our previous design choices it is simple to utilise our previous code on this new problem with 3 sinks instead of 5. I change the SinkNodeModel t and n constraints to match the problem description (SinkNodeModel, 13-16). With this exercise n != 1 and therefore the special case to handle that event becomes obsolete and the synchronisations are that much easier to compute.

The overall control flow is very similar to that of the Ptolemy exercise. I created a method that handles the reading of beacons on different channels as per SourceNode's previously described logic (Question 1). It uses timer callbacks to determine when to fire, when to change channel and when to read beacons (MySouceNode, handleNextAction, 142-169). It is almost equivalent to the fire method in the Ptolemy exercise (SourceNodeActor, fire, 69-90) except it is built around a physical radio which transmits and receives on wireless channels as opposed to a simulation. This meant that I could not change channel as easily as in Ptolemy, in mote I firstly switch off the radio (MySourceNode, switchChannel, 176-185), then switch channels before attempting to either transmit or receive.

In terms of energy efficiency, my implementation only has a very slight opti-

misation for energy saving. It is clear that if we know both the parameters for a given sink node, then we no longer need to receive any beacons from it, we have all the values needed to calculate each subsequent reception phase and therefore do not need any more beacons. We can therefore stop receiving when this is the case and save energy by having the radio off at this time, which my solution does (MySourceNode, handleNextAction, 157-167).

Something I could have adopted for energy efficiency would be to disable all interactions with mote LEDs, currently LEDS are used to indicate the mote is initialised, which stays lit for the entire duration of the motes powered-on lifetime with this assembly loaded onto it (MySourceNode, 94). They are also used when messages are being received (MySourceNode, 127) and when we are transmitting (MySourceNode, 155). Disabling LEDs would undoubtedly improve mote battery life, but my solution uses them anyway as they are very useful for debugging purposes.

3 Part 3 - Embedded platform modelling

3.1 Question 1

	without bus-invert	with bus-invert
address sub-bus	26236	22602
data sub-bus	25341	21916

Table 1: Number of transitions

Approximately a 14% decrease in bus transitions for both the address sub-bus and data sub-bus when using bus-invert.

Bus-invert coding for low power effectively reduces the number of transitions on a bus, as seen in Table 1 above. The following relationship shows how the number of transitions is linearly related to the dynamic power dissipation of the chip (P_{chip}):

$P_{chip} \propto C_{average} \cdot V_{dd}^2 \cdot f \cdot N(transitions)$ [**?**]

This decrease in the total number of transitions translates into a decrease in the activity factor of the bus, where $N(transitions)$ is the activity factor. This in turn means a linear decrease in the dynamic power dissipation of the chip.

We can therefore estimate that our implementation of bus-invert coding on the given multi-processor platform would reduce the dynamic power dissipation by approximately 14% just as the transitions were.

As for static power dissipation, in order to implement bus-invert coding, we require extra circuitry or in our simulated case we have extra actors. These extra circuits/actors result in a greater static power dissipation, however, static power dissipation is typically insignificant being very small in magnitude [**?**].

So in conclusion, the bus-invert coding is an effective method of reducing the dynamic power dissipation of a low-power embedded system and typically compensates for any increase in static power dissipation.

References

[1] Mircea R. Stan, Member, IEEE, and Wayne P. Burleson, Member, IEEE (1995). Bus-Invert Coding for Low-Power I/O. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 3, NO. I, MARCH 1995*, 49-58.