A sensor network for environmental monitoring using PSoCs

A Thesis Presented
by

Varun Subramanian

to

The Graduate School
in partial fulfillment of the
Requirements
for the degree of

Master of Science
in
Electrical and Computer Engineering

Stony Brook University
December 2008

Stony Brook University
The Graduate School
Varun Subramanian
We, the thesis committee for the above candidate for the
Master of Science degree,
hereby recommend acceptance of this thesis.

Dr. Alex Doboli, Advisor of Thesis
Associate Professor, Department of Electrical and Computer Engineering

Dr. Milutin Stanacevic, Chairman, Assistant Professor,
Department of Electrical and Computer Engineering

Dr. Sangjin Hong, Associate Professor,
Department of Electrical and Computer Engineering

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

A sensor network for environmental monitoring using PSoCs

by
Varun Subramanian
Master of Science
in
Electrical and Computer Engineering
Stony Brook University
2008

This report presents a detailed description of the implementation of a grid type sensor network and presents a case study based on an application for environmental monitoring. Programmable System on Chip (PSoCs) are used for the implementation as PSoC is a reconfigurable System on a Chip (SoC) which helps in designing adaptation policies for reconfigurable sensor nodes. The implementation is specific to regions defined within the network and various functionalities to be implemented are specified for each region.

Prior to execution, the user has to define parameters that update the data structure of the nodes. These parameters are defined by basic commands which are transmitted by the server (PC) to the PSoC network in the form of packets. The server communicates directly to one node (PSoC) called the *Entry Point* which is located at co-ordinates (0,0) in the network. The Entry Point receives these commands from the server, processes them and broadcasts them to all the nodes in the network using a broadcast scheme.

The commands include defining regions and various parameters associated to the regions. After defining these parameters, various events and goals are defined specific to regions using a different set of commands. The nodes produce actuation signals depending on the goals defined.

After all parameters are specified by the commands, the user gives out a command to start execution which involves sensing, processing and networking under tight hardware, bandwidth and energy constraints.

To My Parents and my beloved sister Divya

Table of Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to begin this acknowledgement by thanking my advisor Dr. Alex Doboli who has been my mentor throughout the process of working on this interesting project. He always found time to solve my queries in his busy schedule. I will always remember the interesting conversations I had with him, the topics ranging from research and academia to entertainment and sports. I really learnt a lot from him and I tried to be an ideal student as he is an ideal advisor.

I also thank my colleagues Cristian Ferent, Pengbo Sun and Meng Wang in the VLSI Systems Designs Laboratory (VSD Lab) for the help they have provided when needed. I would specially like to thank Cristian who has been a good friend to me and also helped me solve complex issues during the implementation. We have also been helpful to each other to ease the stress and tension we go through in the lab by periodically making jokes and having a good laugh. It is always good to have a friend like Cristian at your workplace.

I would like to end this acknowledgement by thanking my parents and my beloved sister in India and also my uncle Mr. C.V Kumar who lives here in Long Island. Even though my parents and my sister were not physically present here to help me, they have been very supportive in whatever decisions I made. My uncle was my mentor and guardian throughout my stay here in the United States. He helped me a lot in making rational decisions ever since I decided to pursue my higher studies in the United States.

Chapter 1

Introduction

Sensor networks are emerging as a key concept for many modern day applications like environmental monitoring, tracking, healthcare etc. This is possible because of the reason that sensing and electronic devices are small and cheap and thus, can be deployed in large numbers. It also enables superior decision making capabilities as related problems are tackled together instead of separately which also helps in improving the reliability and robustness of the system as failure of local nodes do no affect the overall functionality of the system. This report discusses an efficient implementation which involves sensing, processing and networking by each individual node in the network under tight hardware, bandwidth and energy constraints.

Programmable System on Chip (PSoCs) are the nodes in the grid network. PSoC is a reconfigurable System on Chip which helps in designing adapation policies for reconfigurable sensor nodes. The implementation involves the following two design steps: (i)defining regions & the associated parameters, events and goals specific to the regions with the help of command packets through the server and (ii)start execution which includes sensing, processing and networking.

The regions and the associated parameters are defined by the user through the server which sends the commands to the Entry Point which is located at co-ordinates (0,0) in the network. The functionality of the Entry Point is detailed in chapter 2. The parameters associated with the region include *Target Point*, *Path*, *Path Probability*, *Aggregation Function* and *Precision.* The Entry Point processes these commands by converting the ASCII characters into the hexadecimal equivalent before broadcasting these commands to all nodes in the network by using broadcast scheme 'A' which ensures that all the nodes receive these commands and they receive it only once. It is important that they receive it only once as there is no way the nodes will know if the same command packet was received earlier and it may update the data structure with redundant data. In scheme A, the command packets are first sent to all nodes in row 0 of the network (i.e. all the nodes with X co-ordinate equal to 0) and these nodes in row 0 send the commands to all the nodes in their respective columns. It is thus ensured that all the nodes in the network receive the commands only once. The nodes after receiving the command packets check if the data in the packet matches the co-ordinates of the node and accordingly update the data structure before sending the packet to the neighboring node(s). The command packet structure is detailed in chapter 3 and the data structure for the various parameters is detailed in chapter 4. After defining the regions and the parameters, the user defines the goals associated with the regions and also the events. The goals defined for the region determines the type of actuation signals the nodes have to produce.

After defining all the parameters and goals for the regions, the user gives the command to start execution. The routine 'execute' is called which is a loop and the node exits the loop only on network reset. Within the loop, the code first checks if the node is a Target Point. If the node is not the Target Point, it performs the functionality of sensing, processing and networking. The nodes sense data, perform the aggregation function defined for the region and select one of the paths defined for the region to transmit the aggregated data to the Target Point. The nodes send this data to the Target Point in the form of *Data Packets*. The nodes then check for events which occur when the aggregated data exceeds the threshold value. The node informs the Target Point about the occurrence of an event by sending the *Event Packet* to the Target Point using the transmit scheme B, which is defined in chapter 5. The node glows an LED when an event occurs and the LED is turned off only when the recomputed aggregated data does not produce an event. After checking for the occurrence of an event, the node generates an actuation signal depending on the goals defined for the region. Since the case study is based on an application for environmental monitoring, the goals defined for the regions is to keep the temperature within a range. Hence, the actuation signals generated by the nodes control the speed of a fan. The aggregated data is compared with the range of temperature values defined for the region and the speed of the fan is controlled. While executing the functionalities mentioned above, the nodes also periodically check for received data and the packets received by the nodes during execution are Information packets which are forwarded by the nodes to one of the neighbor nodes. The nodes repeat the same functionality for all the regions using a round robin policy. The node stops execution only on network reset which can either be a hard reset or a soft reset. Sending the command *Reset network* by the user through the server results in a soft reset.

If the node is the Target Point, it does not perform any sensing or processing and performs the functionality of collecting data from all other nodes in the region and form a data pool. After receiving data from all nodes in the region, the Target Point creates the *Data pool packet* which is transmitted to the Entry Point using the transmit scheme C. The Entry Point forwards this packet to the server. The *Data pool packet* contains information about all nodes in the region which includes the node co-ordinates, aggregated data and the aggregation function. The server hence periodically gets updates from the Target Point.

Chapter 2

Entry Point Interface with PSoC Network

Configuration and execution commands are setup in the PSoC Network by using a server (PC) as the user terminal and access point to the network. For this, the serial interface is used between the server and a special node in the PSoC network, the entry point. This node has a dedicated mode of operation. It receives commands from the server, decodes and translates them into an appropriate packet format for the PSoC nodes in the network, and then it broadcasts the commands to all the PSoC nodes.

The entry point also transmits back to the server information about the network's operation. This data is received by the entry point from the target points of the different regions in the network and then is sent to the server.

For the serial interface implemented between the server and the PSoC Network, each command packet is made up of multiple 8 bits (1 byte) long serial frames. For transmitting these commands to the PSoC network, the entry point translates the commands into a format that reduces the number of serial frames in a packet.

The individual commands are described in Section II. Section III shortly describes the format translation done by the entry point in order to adapt the command packet for the PSoC nodes. The mode of operation of the entry point is presented in Section IV and in Section V an analysis of the clock cycles needed for operation is given.

2.1 Entry Point Interface with PSoC Network

This section describes the packet format which is sent throughout the PSoC network, after the entry point (node 0,0) decodes the commands received from the server (PC).

The command packet sent in the network is formed of 8 bit wide serial UART frames. Each of these UART frame contains information about the specific command, according to the command structure described in the previous section. At the end of each packet, an 8 bit value of 0xFF is sent in the network in order to indicate the end of the packet. At this time, the command type and command parameters are sent as their equivalent ASCII codes (8 bits), except for numeric information. Numbers (ex. coordinates) are processed by the entry point and changed from their ASCII representation to an 8 bit value. The spaces from the server commands are eliminated, in this way reducing the size of a command packet and optimizing the communication channel usage.

Using this approach, the network command packets follow the same format as the server commands with the exclusion of white spaces. All UART frames have the same meaning.

The start network command is converted to a single frame with the fixed predefined value of 0xFD. The reset network command is processed in a similar way, with the single frame value being 0xFE.

All commands are sent to all the nodes in the PSoC network, implementing a broadcast communication scheme.

2.2 Entry Point Implementation Description

The entry point code is written in assembly in order to optimize it for speed, by trying to reduce the overhead a C program would generate. This approach has been taken because of the basic lower level at which these commands operate within the PSoC network. Regardless of the application implemented on the network, there is always the need to execute these commands in order to set up the network's data structure, configure the communication between the nodes and set the region's or individual node's operation.

Apart from speed, code size is also taken into consideration when implementing the commands on the entry point. Because all subsequent applications have their own program (code) added to this current structure, the command processing algorithm code size has to be kept to a minimum. This is done by defining general routines that can be reused often. The penalty paid with this approach is that some clock cycles are wasted with the *call* and *return* instructions. However, this loss in execution speed is compensated by the extended

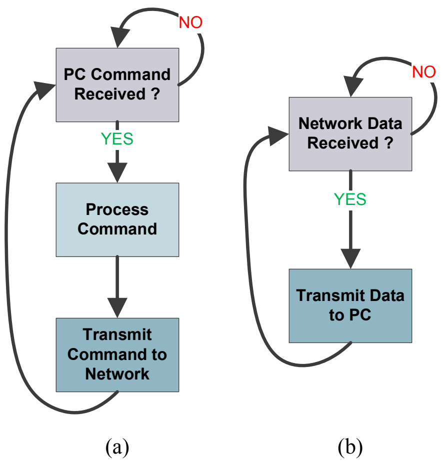

Figure 2.1: Entry point operation. (a) Server command execution. (b) Network data execution

program memory available for other applications.

As described in the previous sections, the commands are received from the server thorough the serial interface using ASCII characters. The implementation only supports one character sized names (region_id, path_id, etc) and only two digit numbers (coordinates, precision). The entry point has two dedicated operation modes. The flow charts for these two modes are shown in Fig.1.

One mode of operation deals with the commands received from the server and processes these commands and transmits them in the entire PSoC network (Fig.1 (a)). The second mode of operation focuses on receiving data from the network and transmitting it to the server (Fig.1 (b)).

2.2.1 Receiving Server Commands

Commands are received through the serial interface from the server, therefore on the PSoC node designated as an entry point an RX8 module is used with the receive interrupt for this module, minimizing the software resources needed and freeing up the CPU for other tasks.

Commands are received from the server one byte (one ASCII character) at a time and for each byte the receive interrupt is triggered. After checking if the byte received is correct (no parity errors), another check is performed to see if the byte is an end of command string byte (ASCII for new line). If the byte is not an end of command indicator, the value is saved in SRAM memory. When an end of command indicator is received, a variable is set in order to signal the main program loop that the entire server command has been received successfully.

The main program constantly monitors this variable showing a complete command received, and if so, processes the command and then clears this variable and resets the index at which bytes are to be received for the next command (a new command will overwrite the previous command bytes in memory only after the command has been processed and transmitted to the network).

2.2.2 Processing Server Commands

After a command is received, it undergoes several processing steps before it is sent forward in the network. Multiple routines are implemented in order to deal with this aspect. Using function calls for these general purpose operations (they are shared by all the server commands used) reduces the code size required.

The processing routines implemented in the design are:

- get_command_type
- get_region_name
- get_path_name
- get_number
- get_number_last
- save_path_coordinate

All of these functions listed above assume the X register is set up to point to the next character in the command string that needs to be processed. Since the command format is well determined, decoding or processing a command implies calling these functions in the correct sequence and adjusting the X register to point at the next character that is to be considered before calling a function. No check is done to determine the correctness of a command received and no error message is produced in case an unknown command is received or in case there are syntax issues in the command string. The first condition will not generate a functional issue (the command is ignored), but the former will determine the system to malfunction.

As discussed previously, the command string is saved byte by byte in SRAM (starting at a predefined location) until the ASCII new line character is received, at which point the RX8 interrupt service routine signals the main program to start processing. At this time, the X register is set up to point to the first byte received and the *get_command_type* function is called. This function places the first byte of the command string (command identifier) in a predefined SRAM location (*pkt_start*) which is the first byte of the packet that will be sent in the network once the entire command is decoded. The main program then checks the content of the *pkt_start* location and determines the command type (and executes the correct sequence of functions for that command) or ignores the entire command string in case of an unknown command.

The *get_region_name* and *get_path_name* functions operate in a similar way with the *get_command_type* function, saving the region name and path name, respectively, from the command string to the A register (returned value). This value is afterwards saved to the next position available in the packet that is being assembled for transmitting in the network.

The *get_number* function (also *get_number_last*) is used to determine the numbers present in the command string. Before calling this function, the X register is set up to point to the next unprocessed byte in the string. When called, this function counts the number of digits in the number (how many characters between two consecutive spaces) and transforms the ASCII representation to the actual number: 0x30 is subtracted from each byte and if the number is composed of two digits, the first digit is multiplied by 10 (using the hardware multiplier) and then the result added to the second digit (only maximum 2 digit support). The actual number represented on 8 bits is returned in the A register and later saved in the next available space of the packet that is being assembled for transmitting in the network. The *get_number_last* function operates in the same way, except it counts the number of digits between a space and the end of command indicator (ASCII new line character).

Because the path defining command does not have a fixed number of bytes, the function *save_path_coordinate* was implemented to overcome this issue and keep track where each coordinate needs to be placed in the packet that is going to be sent in the network. The combination of this function and the *get_number* or *get_number_last* is used in a loop until all the coordinates from the command string have been added to the final network packet.

Once all the ASCII characters from the command string have been decoded (processed) and the network packet is assembled, the main program places the packet end indicator (0xFF). This indicator is used by the receiving PSoC nodes in a similar way the entry point uses the ASCII new line character when communicating with the server to determine the end of a command.

In the case of a start or reset command being received from the server, the main program does not do any function calls other than *get_command_type*. After determining that the command is a start or reset command, the start or reset indicator is sent in the PSoC network (0xFD or 0xFE, respectively). The reset command is automatically executed and sent in the network on power-up or hard reset. This is a safety measure taken to prevent power-up noise from generating false commands in the network.

2.2.3 Transmitting Server Commands

A processed command is assembled into a packet terminated by an end of packet indicator (0xFF) as described in the previous sections. This packet is then sent to all the nodes in the PSoC network using the TX8 modules. Transmitting is not implemented using interrupts and a broadcast technique is used: all the nodes in the network will receive the command, but only the ones that are affected by it will interpret it, all the others will just ignore and forward it.

The entry point is considered to be of coordinates 0,0 in this grid implementation of the PSoC network. For this reason, it uses two TX8 modules, one labeled 'up' (for transmitting to upper row in the network) and the other label 'right' (for transmitting to next column in the network). Both of these interfaces are used to send command packets. Because an interrupt approach is not available and simultaneous transmission is not possible, first the 'up' direction and then the 'right' direction is activated.

Two different functions are implemented to deal with these two transmit directions: *send_up_UART* and *send_right_UART*. They operate in a similar way, except they use different physical TX8 modules, hence they call different API routines (specific to each module).

The implementation of the *send_up_UART* function uses a loop to send each byte of the command packet. Starting from SRAM location *pkt_start* bytes are sent one at a time on the TX8 module by calling the *send_data* TX8 API routine. The loop executes until the last byte transmitted is equal to

0xFF. Except for path command packet, all other packets have a known size (total number of bytes) and therefore the number of iterations through the loop can be determined.

The *send_right_UART* function also includes the usage of a third TX8 module, this one connected back to the server. The role of this module is to enable an echo function back to the server (all the bytes sent in the network can be viewed on the server terminal). It serves debugging purposes only and has no application functionality.

2.2.4 Network Data

The entry point also serves the function of sending network data and information back to the server so that it can be analyzed on the terminal. This process is similar to the server commands process discussed previously. Using RX8 modules and the interrupts associated with them, data from the network is received in SRAM, starting from a predefined index. When the packet end indicator is received (0xFF) the interrupt service routine signals the main program loop that new data has been received from the network by setting a specific data received flag. At this point there is no processing of the data packet involved, it is just forwarded to the server through the same TX8 module used to echo the server commands. After the last byte is sent (0xFF), the data received flag is reset by the main loop. There are two possible directions from which the entry point can receive network data. Data overwrites from the two different locations are overcome by using two different SRAM locations to store the data packets and two different data received flags (one for each direction).

2.3 Entry Point Clock Cycles Analysis

This section presents a clock cycle analysis done on the entry point assembly language implementation. It is intended to give a measure of the complexity of each part of the application. Different scenarios (branches taken or not taken) are discussed and total number of clock cycles are given for each segment of the code: receiving a server command, processing a server command and transmitting a server command.

This analysis does not include the network data process since it is based on similar routines with the process involving a server command. For receiving network data, the clock cycle count is close to the count for receiving server commands. In the case of transmitting network data to the server, the same formula that is given for transmitting a path command can be used for reference.

2.3.1 Receiving Server Commands

For this section, the RX8 receive interrupt service routine is presented from the total number of clock cycles perspective, taking into account the different branch instruction outcomes. Ignoring the case of parity errors, there are two possible branch outcomes: the ASCII character is not new line (normal) or it is ASCII new line (complete command was received). This data is shown in

Byte type	**Clock Cycles**
normal	115
new line	112

Table 2.1: Receive Interrupt Clock Cycle Count (one byte)

Table I. Information is given for a single byte received. In case of a complete command, the total number of clock cycles is determined with the formula (1). API routines clock cycle count are not included, only the *call* and the *return* instructions.

$$Clks_{command} = (N + 1) \times Clks_{normal} + Clks_{newline} \tag{2.1}$$

In (1), N stands for the number of bytes the command string has (including spaces). Clks$_{normal}$ and Clks$_{newline}$ are the data from Table I for a normal byte received and a newline character, respectively. The N + 1 accounts for the ASCII carriage return character received before the new line character (considered as a normal character).

2.3.2 Processing Server Commands

All routines presented in section IV.B are shown and the total number of clock cycles are given (Table II). *Call* and *return* instructions for these functions are considered in this count. Furthermore, the total number of clock cycles data is also provided about processing each of the commands from section II (Table III). Best case scenarios consider that all the numbers in the command are single digit. Worst case scenarios assume that all the numbers in a command are made up of two digits. Since a path command has a variable total number

Function	Best Case	Worst Case
get_command_type	40	-
get_region_name	35	-
get_path_name	35	-
get_number	164	255
save_path_coordinate	64	-

Table 2.2: Processing Functions Clock Cycle Count

Command	Best Case	Worst Case
define region	877	1241
define target	518	700
define path	$489 \times N_c + 428$	$671 \times N_c + 519$
define path probability	423	514
define region precision	583	765
define aggregation function	585	771
define region event	423	514
define node event	782	1146
define region range	795	1159
define node range	1157	1612
network start	272	-
network reset	207	-

Table 2.3: Processing Complete Commands Clock Cycle Count

of bytes, a formula depending on this number is given (N_c stands for the number of nodes in the server path command string).

2.3.3 Transmitting Server Commands

In this section, the total number of clock cycles for transmitting the server command packet for all commands discussed in section II is presented (Table IV). Since a path command has a variable total number of bytes, a formula

Command Transmitted	**Clock Cycles**
define region	2×706
define target	2×500
define path	$2 \times (103 \times \mathrm{N}_p + 88)$
define path probability	2×500
define region precision	2×500
define aggregation function	2×603
define region event	2×397
define node event	2×603
define region range	2×603
define node range	2×809
network start	2×55
network reset	2×55

Table 2.4: Transmitting Complete Commands Clock Cycle Count

depending on this number is given (N_p stands for the number of bytes in the path packet - coordinates are 1 byte and there are no spaces in packet at this point, 0xFF not included in N_p). The TX8 module API routine clock cycle count is not included, only the *call* and the *return* instructions are considered. The total number of clock cycles for both 'up' and 'right' transmitting are shown in Table IV (2 * one direction count).

Chapter 3

The Packet Structure

In the PSoC network, packets are the means by which data is communicated between nodes and also between the server and the PSoC network. The size of a packet is equal to the number of bytes in the packet. The size is fixed for a type of packet with a few exceptions. The value FFh denotes the *End of Packet* and is included for all packets. The *End of Packet* is included while computing the size of packet. *Reset Network* and *Start Execution* are the only packets with do not include an *End of Packet.*

The packets are of two types namely (i)Command Packets and (ii)Information Packets. The command packets are further subdivided into (a)Define regions & the associated parameters, (b)Define events & actuation procedures and (c)Reset network & Start execution.

3.1 Command Packets

Command packets are sent from the server to the PSoC network to define various parameters that update the data structure of the nodes in the network.

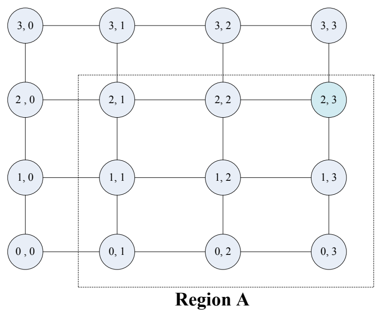

Figure 3.1: Defining a Region

The nodes perform different functionalities relative to these parameters. The command packets are further divided into the following subtypes:

3.1.1 Define Regions & the associated parameters

These packets are command packets which are used to define a region and the parameters associated with it which include the *Target point*, *Region's path*, *Path Probability*, *Region's precision* and *Region's aggregation function*.

3.1.1.1 Define Region

This command is used to define a set of nodes within the PSoC grid to be a part of the same region. The size of this packet is 7.

Command string format:

$$r \ldots region\_id \ldots x_1 \ldots y_1 \ldots x_2 \ldots y_2 \ldots FFh$$

- r - command id (define region)
- region_id - region name
- x_1, y_1 - coordinates for bottom-left corner of region
- x_2, y_2 - coordinates for top-right corner of region
- FFh - End of Packet

ex. r A 0 1 2 3 *// defines a region named "A", node (0,1) and (2,3) set the boundaries of region "A", node (0,1) being the bottom-left corner of the region and node (2,3) being the top-right corner of the region as shown in figure 3.1,*

3.1.1.2 Define Region's Target Point

This command establishes which of the nodes associated with a region is the target node. The target node is the node where all the paths within the region will end. The Target node collects data from all the nodes in the network and forms a data pool. The data from the region is fed back to the *Entry Point* and from the *Entry Point* to the server from the target node. The size of this packet is 5.

Command string format:

$$t \ldots region\_id \ldots x \ldots y \ldots FFh$$

- t - command id (define target point)

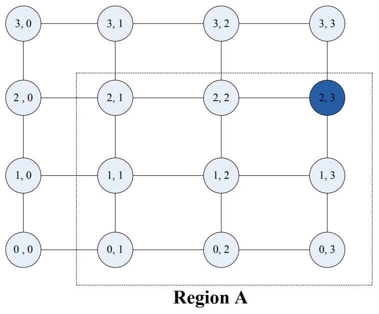

Figure 3.2: Defining a Target Point

- region_id - region name
- x, y - coordinates of target point
- FFh - End of Packet

ex. t A 2 3 *// defines the node (2,3) to be the Target Node for region "A" as shown in figure 3.2 where the Target Point is highlighted.*

3.1.1.3 Define Region's Path

This command sets a path inside a predefined region. Only one path can be set at a time but a region can have multiple paths. The size of this packet is not fixed as the length of a path is not fixed.

Command string format:

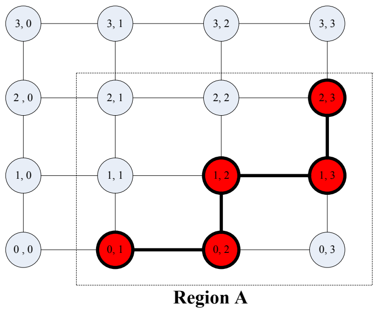

Figure 3.3: Defining a Path

$$p \ldots region\_id \ldots path\_id \ldots no\_of\_nodes \ldots list\_of\_nodes \ldots FFh$$

- p - command id (define path)
- region_id - region name
- path_id - path name
- no_of_nodes - total number of nodes on path
- list_of_nodes - x_i, y_i - coordinates of nodes on path, separated by spaces and last node is always the target point
- FFh - End of Packet

ex. p A P 5 0 1 0 2 1 2 1 3 2 3 *// defines a path named "P" in region named "A" having 5 nodes with the co-ordinates (0,1), (0,2), (1,2), (1,3) &*

(2,3). (2,3) is the target node and the last node in the path. The path is shown in figure 3.3 where the path from (0,1) to (2,3) is highlighted.

3.1.1.4 Define Path Probability

This command is used to set the probability with which a path is chosen within a predefined region. The size of this packet is 5.

Command string format:

$$q \ldots region\_id \ldots path\_id \ldots path\_probability \ldots FFh$$

- q - command id (define path probability)
- region_id - region name
- path_id - path name
- path_probability - the probability with which a path is chosen (given in %)
- FFh - End of Packet

ex. q A P 30 *// sets the probability of path named "P" in region named "A" to 30% = 0.3.*

3.1.1.5 Define Region's Precision

This command sets the precision of the data acquisition within a predefined region by specifying the resolution of the ADC (this changes the sampling time) and the time interval between two measurements. All nodes within the region will use these settings. The size of this packet is 5.

Command string format:

$$s \ldots region\_id \ldots no\_of\_bits \ldots no\_of\_seconds \ldots FFh$$

- s - command id (define space and time precision within a region)
- region_id - region name
- no_of_bits - ADC bit resolution
- no_of_seconds - time interval in which at least one measurement must be made
- FFh - End of Packet

ex. s A 8 5 *// precision in region named "A" is set to 8 bit ADC resolution and an interval of 5 seconds between 2 measurements.*

3.1.1.6 Define Region's Aggregation Function

This command sets the way in which data is processed by a node within a predefined region. Special keywords are used to set the different functions. This command is specific to a node in the region and separate commands have to be defined for each node in the region. The size of this packet is 6.

Command string format:

$$f \ldots region\_id \ldots keyword \ldots x \ldots y \ldots FFh$$

- f - command id (define aggregation function within a region)
- region_id - region name
- keyword - sets which function to use
- x, y - coordinates of the node for which the function applies
- FFh - End of Packet

Example of possible keywords: a - arithmetic mean for 5 sensed values; g - arithmetic mean for 4 sensed values

ex. f A g 2 3 *// in region named "A", arithmetic mean ("g") of 4 sensed values will be used as aggregation function for node (2,3).*

3.1.2 Define Events & Actuation Procedures

These packets are command packets which are used to define events and actuation procedures. An event occurs when the temperature value sensed exceeds a threshold value. A fan is used for the actuation and hence, the speed of the fan needs to be controlled depending on the range of the temperature value sensed which is specified by the commands.

3.1.2.1 Define Region's Event

This command is used to define an event for a region. The size of this packet is 4.

Command string format:

$$h \ldots region\_id \ldots threshold \ldots FFh$$

- h - command id (define a region's event)
- region_id - region name
- threshold - threshold temperature value that generates an event
- FFh - End of Packet

ex. h A 30 *// defines an event for region "A", 30º celcius being the threshold temperature value that generates an event.*

3.1.2.2 Define a Node's Event for a Region

This command is used to define an event for a node (x, y) specific to region "A". The default threshold value for the node for region "A" is the one defined by the command *Define a region's event.* The size of this packet is 6.

Command string format:

n. . . region_id. . . x. . . y. . . threshold. . . FFh

- n - command id (define an event for a node)
- region_id - region name
- x, y - coordinates of node
- threshold - threshold temperature value that generates an event
- FFh - End of Packet

ex. n A 3 12 30 *// defines an event for node (3,12) for region ”A”, 30º celcius being the threshold temperature value that generates an event. The node may be a part of other regions and they may have different threshold values specific to those regions.*

3.1.2.3 Define Region’s Range

This command defines a range of temperature values for controlling the speed of the fan for a specific region. The command packet specifies three temperature values T1, T2 and T3. If the temperature value is less than T1, the fan is very slow i.e. the PWM which drives the fan has 25% duty cycle. If the temperature value is between T1 & T2, the speed of the fan is slow (PWM with 50% duty cycle). If the temperature value is between T2 & T3, the speed of the fan is medium (PWM with 75% duty cycle) and the speed is fast (PWM with 100% duty cycle) if the temperature value is greater than T3. The size of this packet is 6.

Command string format:

g. . . region_id. . . T1. . . T2. . . T3. . . FFh

- g - command id (define region’s range)
- region_id - region name
- T1 - temperature value 1
- T2 - temperature value 2

- T3 - temperature value 3
- FFh - End of Packet

ex. g A 20 25 30 *// defines a range of temperature values for region "A" such that the fan is very slow if the temperature value is less than 20° C, speed of the fan is slow for the temperature range of 20° C - 25° C, speed of the fan is medium for the temperature range of 25° C - 30° C and the speed is fast if the temperature value exceeds 30° C.*

3.1.2.4 Define a Node's Range for a Region

This command defines a range of temperature values for controlling the speed of the fan for a node (x, y) specific to a region "A". The command packet specifies three temperature values T1, T2 and T3. If the temperature value is less than T1, the speed of the fan is very slow i.e. the PWM which drives the fan has 25% duty cycle. If the temperature value is between T1 & T2, the speed of the fan is slow (PWM with 50% duty cycle). If the temperature value is between T2 & T3, the speed of the fan is medium (PWM with 75% duty cycle) and the speed is fast (PWM with 100% duty cycle) if the temperature value is greater than T3. The default range for the node for region "A" is the one defined by the command *Define region's range*. The size of this packet is 8.

Command string format:

o... region_id... x... y... T1... T2... T3... FFh

- g - command id (define node's range for a region)
- region_id - region name
- x, y - co-ordinates of node
- T1 - temperature value 1
- T2 - temperature value 2
- T3 - temperature value 3
- FFh - End of Packet

ex. g A 3 12 20 25 30 *// defines a range of temperature values for node (3, 12) specific to region "A" such that the speed of the fan is very slow if the temperature value is less than 20^o C, speed of the fan is slow for the temperature range of 20^o C - 25^o C, speed of the fan is medium for the temperature range of 25^o C - 30^o C and the speed is fast if the temperature value exceeds 30^o C. The node may be a part of other regions and they may have different ranges specific to those regions.*

3.1.3 Reset network & start execution

These command packets are used to *Reset the network* & *Start execution.*

3.1.3.1 Reset Network

This command resets the entire PSoC Network. It is intended as a software version of a hard reset, enabling the user to remotely reset all the nodes in the network, hence deleting all previous information stored in the data structure (region, target, path, path probability, aggregation function, precision, event and range). This command also stops execution. The size of this packet is 1 and it does not include an *End of Packet*.

Command string format:

$$x$$

This command can be also be executed by pressing the reset button of node (0,0) which is the *Entry Point*.

The actual information is not deleted from memory, rather all the indexes in the data structure are reset to zero, this being equivalent to a total loss of information.

3.1.3.2 Start execution

This command enables execution for the network. All the parameters which include region and it's parameters, events and ranges are defined prior to enabling execution. After the execution is enabled, the nodes in the network start sensing temperature, execute various functions within the regions, check for events and also produce the actuation signals. The size of this packet is 1 and it does not include *End of Packet*.

Command string format:

$$y$$

After this command is defined, the nodes stop execution only on hardware reset or on software reset using the *Reset Network* command.

3.2 Information Packets

These type of packets are not command packets as they are not defined by the server and are also not used to define the data structure and parameters related to execution. They are just information packets which are used to communicate data between nodes in the network.

3.2.1 Define Data Packet

The *Data Packet* is defined by a node in the network. The node uses this packet format to transmit data associated to it to the Target Point of the region along one of the paths defined by the server. All other nodes in the path just forward this information packet to the subsequent node in the path. The size of this packet is 8.

Command string format:

$$D \ldots region\_id \ldots path\_id \ldots x \ldots y \ldots data \ldots funct \ldots FFh$$

- D - command id (Communicate data to Target Point)
- region_id - region name
- path_id - path used to send data to Target Point
- x, y - co-ordinates of the node
- data - aggregated data computed by the node
- funct - aggregation function used by the node

- FFh - End of Packet

ex. D A P 2 3 17 g *// An information packet defined by node (2,3) which sends data associated to region "A" to the Target Point using path "P".*

3.2.2 Define Data pool Packet

This is an information packet defined by the Target Point which collects data from all nodes in the region to form a data pool after which it transmits the collected data to the server. The size of this packet is not fixed as the number of nodes in a region is not fixed.

Command string format:

$$e \ldots region\_id \ldots x_1 \ldots y_1 \ldots data_1 \ldots funct_1 \ldots \ldots x_n \ldots y_n \ldots data_n \ldots funct_n \ldots FFh$$

- e - command id (Send collected data to the server)
- region_id - region name
- x_1, y_1 - co-ordinates of first node in region_id
- $data_1$ - aggregated data of node (x_1, y_1)
- $funct_1$ - aggregation function used by the node (x_1, y_1)
- x_n, y_n - co-ordinates of last node in region_id
- $data_n$ - aggregated data of node (x_n, y_n)

- funct$_n$ - aggregation function used by the node (x$_n$, y$_n$)
- FFh - End of Packet

ex. e A 1 0 17 g 1 1 16 a *// An information packet defined by Target Point of region A which consists of only two nodes (1,0) and (1,1). The aggregated data associated with node(1,0) is 17 & the function used is g and the aggregated data associated with node (1,1) is 16 & the function used is a.*

3.2.3 Define Event Packet

This information packet is defined by an individual node in case an event occurs for a defined region. An event occurs when the computed data is higher than the threshold value defined for the node and the event is specific for a region. The node defines this packet to inform the Target Point that an event has occurred. The size of this packet is 5.

Command string format:

v. . . region_id. . . x. . . y. . . FFh

- v - command id (Inform the Target Point about an event)
- region_id - region name
- x, y - co-ordinates of the node
- FFh - End of Packet

ex. v A 2 3 *// An information packet defined by an individual node which transmits the packet to the Target Point.*

Chapter 4

The Data Structure

The user has to update the data structure for the nodes before giving out the command to *start execution*. The data structure is updated by the user with the help of commands which define the various parameters. These commands are the command packets discussed previously in chapter 3. Once the data structure is updated, the nodes have all the necessary information to start execution.

4.1 Data structure for regions & the associated parameters

The data structure for region and it's associated parameters is shown in figure 4.1.

The command for defining a region has the following format:

$$r \ldots region\_id \ldots x_1 \ldots y_1 \ldots x_2 \ldots y_2 \ldots FFh$$

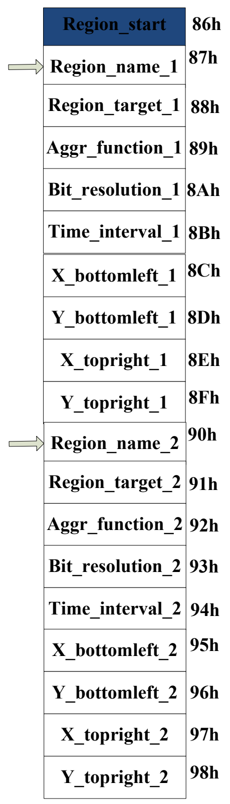

Figure 4.1: Data structure for region & the associated parameters

This command contains the region_id (i.e. the region name) and the co-ordinates of the endpoints of the region. The data structure is updated with these parameters.

The data structure for the region starts at memory address location 86h in SRAM page 1 as shown in figure 4.1. The variable *region_start* is the pointer to the start of the table and is located at memory address location 86h and the value stored in this location is equal to the number of regions defined for the node.

After receiving this command, the node checks if it exists in the region by comparing it's own co-ordinates with the endpoints of the region defined in the command. If the node exists in the region, it updates it's data structure by storing region_id and the co-ordinates of the endpoints of the region.

The node computes the index pointer that points to the location where it should store the region_id. This is done by using the formula:

X = 9 $\times n + 1 + 86h$.

where, n is the number of regions previously defined for the node.

Referring to figure 4.1, if n = 0, X = 87h, if n = 1, X = 90h, if n = 2, X = 99h and so on. These are the memory locations where the region_id is stored.

For example, if n = 1, X = 90h. After computing this index pointer, the node stores the values of the region_id, x_1, y_1, x_2 and y_2 at locations region 2(90h), X_bottomleft 2(95h), Y_bottomleft 2(96h), X_topright 2(97h) and Y_topright 2(98h) respectively. The other parameters are yet to be defined and so they are initialized to 0.

The user defines the *Target Point* for a region using the format:

$$t \ldots region\_id \ldots x \ldots y \ldots FFh$$

The node first checks if it lies in the region specified by the command by comparing region_id in the command with the region _names in the table. If the region exists, then the node compares it's co-ordinates with x & y in the command. If they are equal, then the node is the *Target Point* for that region and it stores a value of 1 in the *region_target* location which is located below the corresponding region_name. A value of 1 in that location indicates that the node is the *Target Point* of the region and the default value of 0 indicates that it is not the *Target Point*.

For example, if the node is a Target Point for region_name_2(90h), then it stores a value of 1 in region_target_2(91h).

The *Target Point* has it's own data structure which the node updates when this command is specified which is discussed in the next section.

The remaining parameters namely *Aggregation function*, *Bit Resolution* and *Time Interval* are also updated on receiving the corresponding commands. In each case, the node first checks if it lies in the region specified in the command by comparing the region_id in the command with the region_names in the table. If it exists, then it updates these parameters to the corresponding locations which are initialized to 0 when a region is defined.

4.2 Data structure for Target Point

The data structure for defining a *Target Point* for a region is shown in figure 4.2.

The command for defining the *Target Point* has the following format:

$$t \ldots region\_id \ldots x \ldots y \ldots FFh$$

This command contains the region_id (i.e. the region name) and the co-ordinates of the *Target Point* of the region. There are 2 different data structures for the *Target Point* as shown in figure 4.2 and they are updated depending on the co-ordinates of the node. The data structure shown in figure 4.2

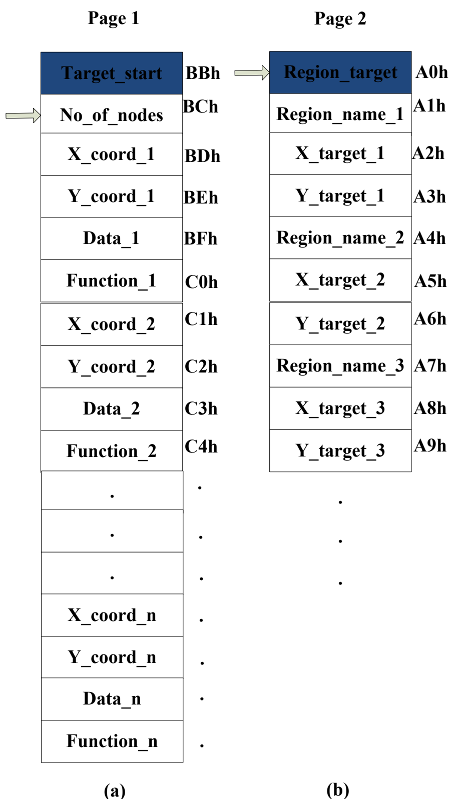

Figure 4.2: Data structure for the Target Point

(a) is updated if the node is the *Target Point* and the data structure shown in figure 4.2 (b) is updated if the node lies in the region but is not the *Target Point*.

The node first checks if it lies in the region specified by the command by comparing region_id in the command with the region _names in the table (refer to figure 4.1). If the region exists, then the node compares it's co-ordinates with x & y in the command. If they are equal, then the node is the *Target Point* for that region and it stores a value of 1 in the *region_target* location which is located below the corresponding region_name. As mentioned earlier, as the node is the *Target Point* for the region, it updates the data structure shown in figure 4.2 (a). The node stores a value of 1 in the *Target_start* location which shows that the node is a *Target Point*. Also *Target_start* acts as a pointer to the start of the table and is located at SRAM page 1 location BBh. After that, using the information of the co-ordinates of the endpoints stored in the table shown in figure 4.1, the node computes the co-ordinates of all the nodes which lie in the region and allocate 4 memory locations for each of them. The 4 memory locations correspond to the x and y co-ordinates of the nodes, the aggregated data of the nodes and the aggregation functions of the nodes. The data structure is now set up as shown in figure 4.2 (a). The *Target Point* collects data from all the nodes and updates this data structure with the information. The memory location *No_of_Nodes* located below *Target_start* indicates the number of nodes that lie in the region.

If the node lies in the region and is not the *Target Point*, the data structure shown in figure 4.2 (b) is updated. This data structure contains the co-ordinates of the *Target Point* of the region. This information is helpful when an event occurs and the node has to inform the *Target Point* of the occurrence of the event. This information packet does not follow the paths defined for the region as the paths are reserved for the data. So the node has to know the location of the *Target Point*.

4.3 Data structure for Path & Path Probability

The data structure for defining a *Region's Path* is shown in figure 4.3.

The command for defining the *Region's Path* has the following format:

$$p\ldots region\_id\ldots path\_id\ldots no\_of\_nodes\ldots list\_of\_nodes\ldots FFh$$

This command contains the region_id (i.e. the region name), the path_id, the number of nodes in the path and the co-ordinates of all the nodes in the path.

The node first checks if it lies in the region specified by the command by comparing region_id in the command with the region _names in the table (refer to figure 4.1). If the region exists, then the node checks if it lies in the path specified in the command by comparing it's own co-ordinates with all the co-ordinates in the path. If the node lies in the path, then it determines the path direction by comparing the co-ordinates of it's neighbors with those of the next node in the path. If the node is the *Target Point*, then it is the last node in the path. Now the node updates the data structure shown in figure 4.3 with the path name and path direction. The path probability is not yet defined and hence, it is initialized to 0. Figure 4.3 shows the data structure for the path. The variable *path_start* is located at location 30h in SRAM page 1 and it points to the start of the data structure for the paths. The paths specific to regions are then updated. The data structure is defined in such a

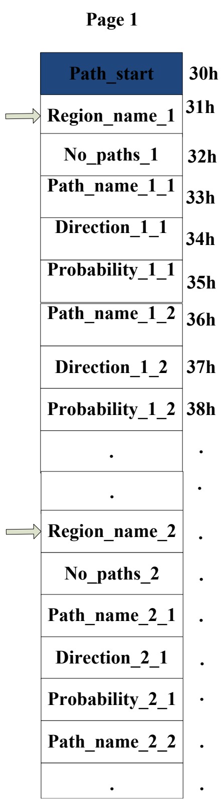

Figure 4.3: Data structure for Path & Path Probability

way that the user has to first define all the paths for a region before moving to the next region. It starts with the region name, the next location is the path strength, i.e the number of paths for that region and then the path names, path directions and the path probabilities. Once the user moves on to the next region, more paths for the previous regions cannot be defined but the path probabilities can be modified.

The command for defining the probability for a path has the format:

$$q\ldots region\_id\ldots path\_id\ldots path\_probability\ldots FFh$$

The node first checks if it lies in the region specified by the command by comparing region_id in the command with the region _names in the table (refer to figure 4.1). If the region exists, then the node checks if it lies in the path specified in the command by comparing path_id in the command with all the path names specific to region_id in the table shown in figure 4.3. If the path exists, the node updates the probability for the corresponding path.

During execution, the node selects the path according to the probabilities defined for them. The node uses these paths to send the aggregated data to the *Target Point* which collects data from all the nodes. Hence each and every path specified for a region ends into the *Target Point*.

4.4 Data structure for Events & Actuation

The data structure for defining an *Event* is shown in figure 4.4 (a).

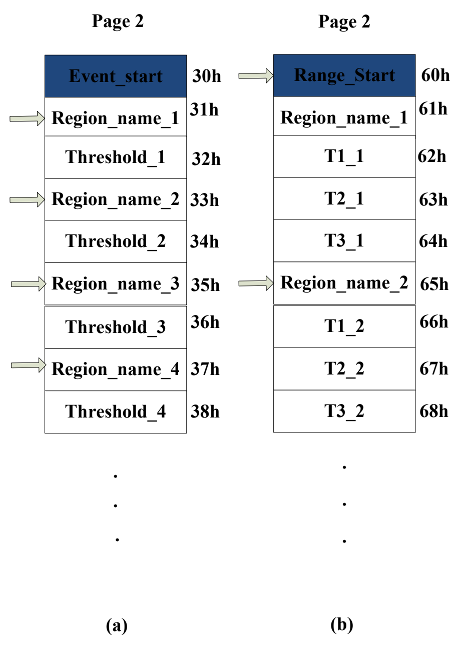

Figure 4.4: Data structure for Events & Actuation

The command for defining an *Event* has the following format:

$$h\ldots region\_id\ldots threshold\ldots FFh$$

This command contains the region_id (i.e. the region name) and the threshold value that generates an event. The event is specific to the region but the user also has the option to change the threshold value for individual nodes.

The node first checks if it lies in the region specified by the command by comparing region_id in the command with the region _names in the table (refer to figure 4.1). If the region exists, then the node updates the data structure shown in figure 4.4 (a). The variable *Event_start* is located at SRAM page 2 location 30h and the value stored is equal to the number of regions for which events are defined.

The command for defining the range (actuation) for a region has the format:

g. . . region_id. . . T1. . . T2. . . T3. . . FFh

This command contains the region_id (i.e. the region name) and the range of temperature values to specify the actuation procedure. The range is specific to the region but the user also has the option to change the range for individual nodes.

The node first checks if it lies in the region specified by the command by comparing region_id in the command with the region _names in the table (refer to figure 4.1). If the region exists, then the node updates the data structure shown in figure 4.4 (b). The variable *Range_start* is located at SRAM page 2 location 60h and the value stored is equal to the number of regions for which the range is defined.

4.5 Clock cycle and Power Consumption analysis

This section discusses the number of clock cycles the code takes to define the parameters and update the data structures. Assembly language was used for optimization. According to the scheme used by the nodes to broadcast data, the nodes transmit the packets to the neighboring node(s) after updating the

Function	**Number of clock cycles**
Define region	1974
Define Target Point	$1847 + 84 \times (r-1) + 112 \times no\_nodes$
Define Path	$453 + 84 \times (r-1) + 68 \times n + 234 \times p$
Define Path Probability	$1400 + 84 \times (r-1) + 86 \times q$
Define Function	$1626 + 84 \times (r-1)$
Define Precision	$1378 + 84 \times (r-1)$
Define Event_Region	$1284 + 84 \times (r-1)$
Define Event_Node	$1692 + 84 \times (r-1)$
Define Range_Region	$1843 + 84 \times (r-1)$
Define Range_Node	$2233 + 84 \times (r-1)$

Table 4.1: Clock cycle count to define parameters

data structure. Hence the clock cycle analysis also include the transmit part along with the part where the node updates the data structure. The number of clock cycles the node takes to transmit depends on the size of the packet 'n' and is equal to:

no_clock_cycles_tx = 12 + 117 * n.

During the analysis, we assume that the parameter defined is relevant to the node. For example, when we define a region, the node is assumed to exist in the region or when a path probability is defined, we assume the node lies in the corresponding path.

The code takes 1974 clock cycles to define a region. As mentioned earlier, it is assumed that the node exists in the region defined.

Before defining any other parameter, the node checks if it exists in the region by comparing the region names in the data structure with the region name in the command packet received. The number of clock cycles taken to execute this functionality is equal to 84 * (r - 1). The region associated with the parameter is the rth region defined in the data structure. Hence the code executes the loop which checks if the region exists (r - 1) times and it takes 84 clock cycles to execute the loop once.

While defining the *Target Point*, the node first checks if it exists in the region which takes 84 * (r - 1) clock cycles. If it exists in the region, it updates the data structure shown in figure 4.2 (a) which takes '1847 + 112 * no_nodes' clock cycles, where no_nodes is the total number of nodes in the region. The node allocates 4 memory locations for each node in the region and hence the number of clock cycles to define the *Target Point* depends on the number of nodes in the region.

While defining the path, the node first checks if it exists in the region which takes 84 * (r - 1) clock cycles. If it exists in the region, it updates the data structure shown in figure 4.3 which takes '453 + 68 * n + 234 * p' clock cycles. The parameter n denotes that the node is the nth node in the path and p denotes that the size of the packet to be transmitted to the neighbor nodes which depends on the number of nodes in the path.

While defining the probability for the path, the node first checks if it exists in the region which takes 84 * (r - 1) clock cycles. If it exists in the region, it updates the data structure shown in figure 4.3 with the probability which is initialized to 0 when the path is defined. This takes '1400 + 86 * q' clock cycles where q denotes that the region associated is the qth region in the data structure.

Function	Power in mW
Define region	259.42
Define Target Point	271.45
Define Path	251.65
Define Path Probability	252.93
Define Function	251.96
Define Precision	240.06
Define Event_Region	239.51
Define Event_Node	251.83
Define Range_Region	251.715
Define Range_Node	250.105

Table 4.2: Power consumed by the nodes while defining parameters

In case of the rest of the commands namely *Define Function*, *Define Precision*, *Define Event_Region*, *Define Event_Node*, *Define Range_Region* & *Define Range_Node*, the node checks if it exists in the region which takes 84 * (r - 1) clock cycles and if the node exists in the region, the commands update the respective data structures and the number of clock cycles taken is as shown in Table 4.1.

The average power consumed by the node to define all the above parameters was computed and is as shown in Table 4.2.

Chapter 5

Methodology of Execution

The user gives out the command to *start execution* after defining all the parameters through commands to update the data structures of the nodes. The parameters include regions and the associated parameters, events and goals (actuation). Once the nodes start execution, they only stop on network reset.

A bit 'start_execute' is set and the routine 'execute' is called which is a loop that performs the functionalities of the nodes.

The nodes sense data, perform aggregation function, check for events and generate actuation signals. The nodes which act as the *Target Point* for a region do not perform these functionalities. They just collect data from all other nodes in the region forming a data pool and send this data to the server. Hence the nodes perform two different kinds of functionalities depending on whether they are specified as a *Target Point*. The node exits the 'execute' loop either on hard reset or soft reset. Sending the command *Reset network* by the user through the server results in a soft reset. This results in resetting the bit 'start_execute' which gets the node out of the routine 'execute'. The data structure is also erased by the node after receiving this command. This is done by resetting all the variables that act as pointers to the start of the data structures. These variables also contain the number of parameters defined. Hence resetting these variables result in a complete loss of data as the code uses these variables to update or retrieve data from the data structure. The hard reset is the hardware reset of the individual nodes.

In the 'execute' routine, the node first checks if it is a *Target Point.* This is done by checking if the variable *target_start* is set. This variable is set or reset while defining the *Target Point* for a region depending on whether the node is the *Target Point.* If the node is the *Target Point*, the execution shifts to a function named *targetpoint_datacollect* that performs the functionality of data collection and if the node is not a *Target Point*, the execution enters a loop *exec_loop* which performs the functionality of sensing and processing. A detailed description of functionalities of *targetpoint_datacollect* and *exec_loop* is given in the sections below.

5.1 Sensing, Processing & Networking by the Nodes

If the node is not a Target Point, the function *exec_loop* is called which performs 3 functionalities for a region namely (i)Sense and aggregate data, (ii)Check for events and (iii) Generate actuation signals. The node performs the same functionalities for all the regions using a round robin policy. The node continues to execute this loop until the network is reset. Figure 5.1

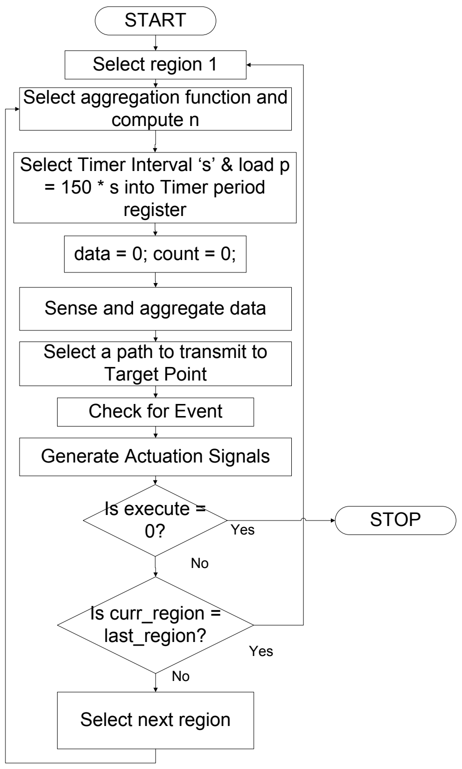
START
Select region 1
Select aggregation function and compute n
Select Timer Interval 's' & load p = 150 * s into Timer period register
data = 0; count = 0;
Sense and aggregate data
Select a path to transmit to Target Point
Check for Event
Generate Actuation Signals
Is execute = 0?
Yes
STOP
No
Is curr_region = last_region?
Yes
No
Select next region

Figure 5.1: Flow of execution

illustrates the flow of execution.

After entering the routine *exec_loop*, performs the following functionalities:

5.1.1 Sense & Aggregate Data

The node first sets the value of n depending on the *aggregation function* defined for the region. A 16-bit Timer circuit enables the sensing of data. The Timer is clocked with a frequency of 150 Hz. Hence, the value of 'p = 150 * s' is loaded into the period register so that the Timer circuit generates an interrupt every 's' seconds, where 's' is defined to be the *Timer Interval* between measurements for the region. The interrupt service routine for the Timer sets the value of the variable 'sense' to 1. Two variables namely 'data' and 'count' are reset to 0.

After setting the value of n, loading the period register of the Timer circuit and resetting the variables, the node enters a waiting loop to see if the variable sense has been set to 1 by the Timer interrupt. But this waiting loop is not idle and the node constantly checks if it has received data from other nodes by calling the function *check_received_data* and forwards the received data to the *Target Point* using the designated path. This forwarding of data by the node takes very few clock cycles to implement and hence it does not affect the *Time Interval* between measurements. When the sense variable is set to 1, the node moves out of the waiting loop and resets the variable 'sense' before calling the routine to sense data. The sensed value is stored in the variable 'data_sense' which is added to the variable 'data'. The variable 'count' is incremented by 1 and then compared with the value of n, which is the number of times the data is sensed before performing the aggregation function. If the value is less than n, the execution moves back to the busy waiting loop to sense data as soon as the variable 'sense' is set to 1 again. If the value of 'count' is equal to n, the aggregation function is performed. The variable 'data' which contains the summation of the data sensed n times is then divided by n to perform an average. The result is again stored into 'data'.

Figure 5.2 gives a detailed description of the *Sense & Aggregate functionality* of the node.

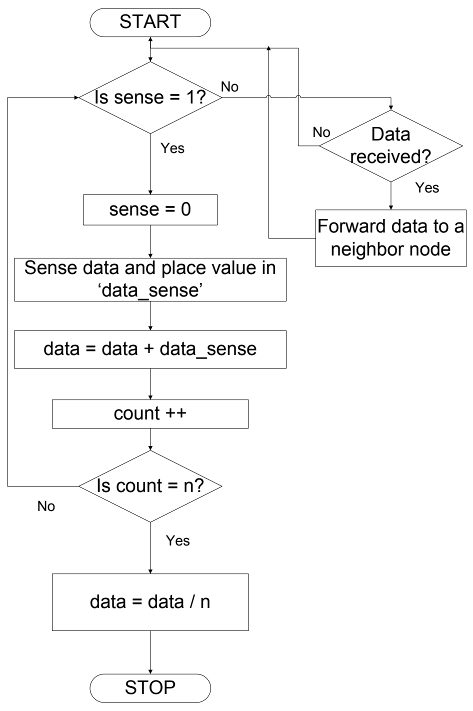

Figure 5.2: Sense & Aggregate data

Once the aggregated data is computed, the data needs to be transmitted to the *Target Point* of the region using one of the paths defined for the region. The routine *select_path* is called for this purpose. The path is selected with the help of the probabilities defined for the paths. The probabilities are in the form of percentage and so, a random number between 1 and 100 is generated. This number is compared to the probability of the path which has the highest probability. If the random number is less than the probability, that path is selected else the random number r is changed to 'r = sum of probabilities of non-eliminated paths - r' and then, the path with the highest probability is eliminated. The process is again repeated until a path is selected. In the case where there is only 1 path left and the rest are eliminated, the path that remains is selected without checking the probabilities thus ensuring that a path is definitely selected to transmit data to the *Target Point.* Once the path is selected, the *Data pool packet* is formed (refer to chapter 3 for the packet structure) and transmitted to the neighbor node which lies in the path. All other nodes in the network just forward this packet to their respective neighbor nodes which lie in the path and the *Data packet* is successfully transmitted to the *Target Point.* Figure 5.3 describes the functionality of selecting a path using the probabilities.

5.1.1.1 Check if data received

The function *check_received_data* is called when the node enters the busy waiting loop before sensing data. The routine checks if data is received from other nodes. Since the nodes have started execution, the nodes will receive only

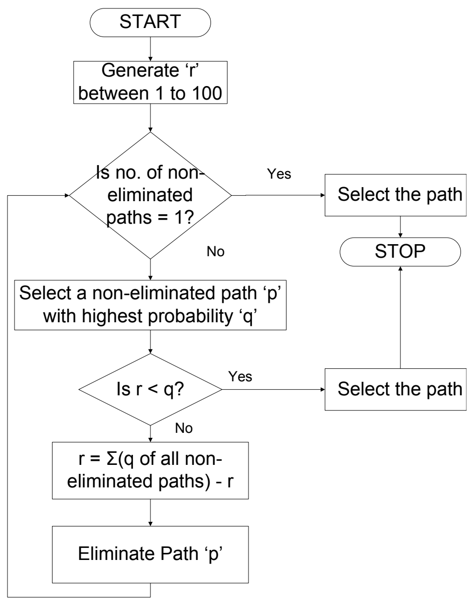
START
Generate ‘r’ between 1 to 100
Is no. of non-eliminated paths = 1?
Yes
Select the path
STOP
No
Select a non-eliminated path ‘p’ with highest probability ‘q’
Is r < q?
Yes
Select the path
No
r = Σ(q of all non-eliminated paths) - r
Eliminate Path ‘p’

Figure 5.3: Select a path

information packets. The receiving is done with the help of interrupts and the ISRs set a bit 'ff_entered' after receiving an end of packet. If the bit is set to 1, the data is retrieved by the node from the buffers, the bit 'ff_entered' is reset and the received data is forwarded to one of the neighboring nodes depending on the type of packet received.

The packet received is either a *Data packet*, a *Data pool packet* or an *Event packet.* If a *Data packet* is received, the node forwards the packet to the neighboring node which will follow the path defined in the packet. This path was selected by the node which transmitted the *Data packet.* If a *Data pool packet* is received, the packet was originally transmitted by the *Target Point* and is meant to be transmitted to the server. According to scheme C used to transmit this packet to the server, if the X co-ordinate of the node is 0, the node transmits the packet to the left neighbor else it transmits the packet to the bottom neighbor. The section on *Data collection by the Target Node* gives a detailed explanation of this scheme. If an *Event packet* is received, the node forwards data using scheme B which is used to transmit an event to the *Target Point.* A detailed explanation of this scheme is given in the next subsection viz. *Check for Events.*

5.1.2 Check for events

An event is defined for the region by the user prior to execution using the command *Define Region's Event.* The user can change the event for an individual node in the region using the command *Define a Node's Event for a Region.*

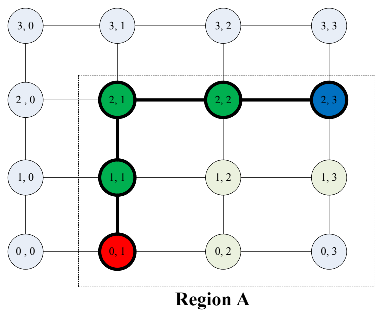

Figure 5.4: Transmitting scheme B for Event Packet

The user defines a threshold value for the event. If the aggregated data exceeds the threshold value, an event occurs. On the occurrence of an event, the node illuminates the LED and turns it off only when the recomputed aggregated data value does not lead to the occurrence of an event. When an event occurs, the node has to inform the *Target Point* about the occurrence of the event and it creates the *Event packet*. The node uses scheme B to transmit to the *Target Point*. In this scheme, the node first compares it's own X co-ordinate with the X co-ordinate of the *Target Point*. If it is not equal, the node transmits top or bottom depending on where the *Target Point* is located. This scheme is also followed by the forwarding nodes. The nodes transmit the *Event Packet* vertically in the network until the packet reaches the row where the *Target Point* is located and then by comparing the Y co-ordinate of the node to that of the *Target Point*, the nodes transmit horizontally till the packet reaches the *Target Point*. Figure 5.4 shows an example where node (0,1) in region A has to transmit an event to the *Target Point* which is located at node (2,3). The node which is shaded with red color is the node which generates the event, the *Target Node* is the one with the dark blue shade and the forwarding nodes are shaded with green color. Since the *Target Node* is located above the node which generates an event, the *Event Packet* is transmitted vertically to the top until it reaches the row at which the *Target Node* is located. After that, the nodes transmit to the right until the packet reaches node (2,3) which is the *Target Node*.

5.1.3 Generate actuation signals

After checking for events, the node generates actuation signals by comparing the aggregated data with the range of values defined by the user for actuation. The range is defined for the region by the user using the command *Define Region's Range*. The user can change the event for an individual node in the region using the command *Define a Node's Range for a Region*. Since the sensed data is a temperature value, the actuation signals generated will adjust the speed of the fan connected to the node. The range defined has 3 temperature values. The aggregated data is compared with these values and the speed of the fan is adjusted according to Table 5.1.

Range	**Speed of fan**
data $\leq T1$	very slow
T1 $\leq data \leq T2$	slow
T2 $\leq data \leq T3$	medium
data $\geq T3$	fast

Table 5.1: Speed control for the fan

5.2 Data collection by the Target Node

If the node is a *Target Point* for the region, it does not perform the functionalities of sensing & processing like the other nodes but collects data from all the nodes in the region and form a data pool. The function *targetpoint_datacollect* is called which performs this functionality. When the node is defined as a *Target Point* before start of execution, the node forms the data structure to store data from all the nodes in the region. The data structure is shown in figure 4.2(a) where 4 memory locations are reserved for each node which include the X co-ordinate, Y co-ordinate, data and aggregation function. Whenever the *Target Point* receives a *Data Packet*, it updates this data structure. It is also assumed that the *Target Point* may exist only in one region as it cannot perform any functionality other than data collection.

The function of the *Target Point* is to form a data pool. Once it receives *Data Packets* from all the nodes in the region, it sends the *Data Pool Packet* to the server. The function *targetpoint_datacollect* has a variable named 'nodes_rec' which keeps a record of the number of nodes from which data was received. Once the count is equal to the number of nodes in the region, the *Tar-*

get Point transmits the *Data pool packet* to the server using transmit scheme C. The variable 'nodes_rec' and the memory locations where the aggregation function is stored are reset to 0 after transmitting the *Data pool packet*.

Each time the *Target Node* receives a *Data Packet*, it first upates the memory location with the data and checks if the memory location where the aggregation function is stored is reset before updating the aggregation function. If that memory location was reset, it increments the variable 'nodes_rec' and if it was not reset, it means that the *Target Point* had received a *Data packet* from that node earlier and it need not increment 'nodes_rec'. Now the *Target Point* checks if 'nodes_rec' is equal to the number of nodes 'n' in the region. If it is equal, the *Target Point* creates the *Data pool packet* which is defined in section 3.2.2 in chapter 3. This packet contains information about all nodes in the network which includes the node co-ordinates, aggregated data and the aggregation function. The size of this packet is not fixed and it depends on the number of nodes defined for the region. The maximum size of the packet for this network is 20 because the maximum size of the receive buffer in all the nodes is 20. The packet structure of *Data pool packet* is defined such that the packet cannot hold information from more than 4 nodes if the maximum size is 20. Hence if the number of nodes in the region is greater than 4, more *Data pool packets* are created by the *Target Point* and the number of *Data pool packets* depend on the size of the region (i.e the number of nodes in the region). These *Data pool packets* are transmitted separately to the server one after the other.

The *Target Point* needs to introduce a delay of 1 second between consecutive transmissions. If the *Target Point* does not introduce the delay, the forwarding nodes will not be quick enough to process all the received *Data pool packets* from the *Target Point* which results in the loss of information as the receive buffer in the node gets overwritten. It was also observed that the forwarding node which is the neighbor node to the *Target Point* gets stuck in an infinite loop which results in a node failure if the packets are transmitted by the *Target Point* without any delay. If the *Target Point* is a neighbor to the *Entry Point*, it has to directly transmit the *Data pool packets* to the *Entry Point*. In this case, it need not introduce any delay between transmissions as the *Entry Point* only has the functionality of forwarding the data received to the server whereas the forwarding nodes also have their own individual functionalities.

The *Data pool packets* are transmitted from the Target Point to the server using scheme C by which the *Data pool packets* are transmitted down till they reach row 0 after which they are transmitted left to the *Entry Point* which forwards the packets to the server. Figure 5.5 shows an example where the *Target Point* which is highlighted with dark blue color is located at node (2,3). The forwarding nodes are highlighted with green color and the Entry Point is highlighted with red color.

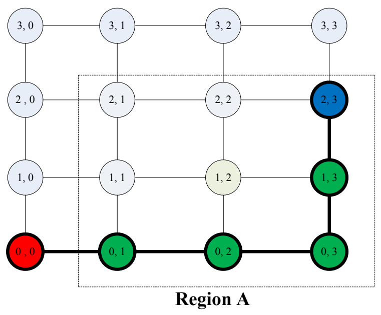

Figure 5.5: Transmitting scheme C for Data Pool Packet

Function	**Number of clock cycles**
Check_received_data	200 + transfer_from_buffer $\times 4$
Transfer_from_buffer	124 + 101 $\times n + \mathit{process}$
Process before execution	180 + command_execution
Process after execution	97 + forwarding_function
Sense	64720
Select_Path	$1233 + 75 \times (r-1) + 91 \times (q-1)$
Check_Event	$1038 + 84 \times (r-1) + 84 \times (t-1)$
Actuation	$268 + 84 \times (r-1)$
Forward_Data_Packet	$1087 + 75 \times (r-1) + 35 \times (p-1)$
Forward_Data_Pool_Packet	$46 + 117 \times n$
Forward_Event_Packet	$848 + 84 \times (r-1)$

Table 5.2: Clock cycle analysis during execution

5.3 Clock cycle and Power Consumption analysis

Table 5.2 shows the clock cycles analysis for the execution part. The code enters the execute loop after receiving the command to start execution. This loop performs the operation of sensing, performing the aggregation function, checking if an information packet is received, checking for events and generating actuation signals. The execute loop calls these functions to implement those functionalities and Table 5.2 gives the clock cycle analysis for each of the functions. The execute loop takes the number of clock cycles given below:

n_clkcycles(execute) = 535 + [(35 + check_received_data)n + 83 + sense] * f + divide + select_path + check_event + actuation.

The execute loop performs the functionality detailed in Figure 5.1. In the equation above, the code checks if data is received while in the waiting loop by calling the function 'check_received_data'. This function checks if data is received from the 4 neighbor nodes connected to the node. If data is received, this function calls the routine that extracts data from the receive buffer (i.e. the function Transfer_from_buffer in Table 5.2) which calls the function 'process' after transferring the data into memory. The routine 'Process' checks if the packet received is an Information Packet during execution after which it calls the function that forwards the packet to the neighboring nodes. Before the start of execution, 'Process' checks if the packet received is a Command Packet and calls the function that updates the data structure. Table 5.2 gives the number of clock cycles taken by the routine 'Process' for both situations.

The execute loop moves out of the waiting loop once the variable 'sense' is set by the Timer Interrupt. The equation assumes that the routine to check if data is received is called n times in the waiting loop. Since the variable to sense data is set, data is sensed by the node. The number of clock cycles to sense data is tabulated in Table 5.2. The aggregation function defines that data needs to be sensed f times before performing the aggregation function (average of the f sensed values), which is included in the equation. After aggregating the data, a path needs to be selected to transmit this data to the Target Point. The function that selects the path is called and it takes '1233 + 75 * (r - 1) + 91 * (q - 1)' clock cycles to execute. The parameter r denotes that the current region is the rth region in the data structure and q denotes the number of paths eliminated while selecting the path.

After selecting the path and transmitting the *Data packet*, the execute loop calls the routine 'check_event' that checks if the aggregated data exceeds the threshold value defined. This routine takes '1038 + 84 * (r - 1) + 84 * (t - 1)' clock cycles, where r denotes that the current region is the rth region in the data structure for events (figure 4.4(a)) and t denotes that the current region is the tth region in the data structure that stores the Target Points for all regions defined for the node (figure 4.2(b)). The data structure in figure 4.2(b) is used to retrieve the co-ordinates of the Target Point of the region so that the node can transmit the event to the Target Point using scheme B.

After checking for events, the execute loop calls the routine 'actuation' which compares the value of the aggregated data with the range to generate actuation signals. This routine takes '268 + 84 * (r - 1)' clock cycles, where r denotes that the current region is the rth region in the data structure for actuation (figur 4.4(b)).

The routine 'Forward_Data_Packet' forwards the data packet along the path designated to it. The routine takes '1087 + 75 * (r - 1) + 35 * (p - 1)' clock cycles, where r denotes that the region name in the data packet is the rth region in the data structure that defines the path (figure 4.3) and p denotes that the path name in the data packet is the pth path defined for that region in the data structure (figure 4.3).

The routine 'Forward_Data_Pool_Packet' forwards the data packet along the path designated to it. The routine takes '46 + 117 * n' clock cycles, where n is the size of the packet which can be at the most 20.

The routine 'Forward_Event_Packet' forwards the data packet along the path designated to it. The routine takes '848 + 84 * (r - 1)' clock cycles, where r denotes that the region in the event packet is the rth region in the data structure that stores the Target Points for all regions defined for the node (figure 4.2(b)). This function then compares its own co-ordinates with the co-ordinates of the Target Point extracted from that data structure and sends the Event Packet to a neighboring node by using the scheme discussed earlier.

The average power consumed by the node to perform these functionalities was computed and is as shown in Table 5.3.

Function	**Power in mW**
Execute loop	263.515
Check_received_data	232.75
Transfer_from_buffer	253.46
Process	230.9
Sense	571.86
Select_Path	243.385
Check_Event	242.83
Actuation	242.21
Forward_Data_Packet	227.215
Forward_Data_Pool_Packet	249.85
Forward_Event_Packet	236.725

Table 5.3: Power consumed by the nodes during execution

Chapter 6

Related work

A model has been proposed in [1] which provides reliable and efficient decision making capabilities to massively distributed embedded systems. The existing approaches focus on either centralized or local control, centralized control being well understood and reliable but not scalable for large systems while local control works well for large systems but it's overall performance is hard to capture. The proposed decision making model is flexible due to low-level reactive behavior, scalable because it uses more abstract (aggregated) data as application size increases, and less reactive and more predictable as global decisions are based on deterministic formalisms.

A systematic procedure for designing adaptation policies for reconfigurable sensor networks is proposed in [2] where the policies control online the characteristics of the sensor nodes and data routing such that the performance requirements are optimized. The procedure includes two steps namely Design Point (DP) generation which represent multiple performance-cost tradeoffs and calculating the switching rates between alternative DPs and possible communication paths so that performance is optimized. Multimode Graphs are used by the DP generation to capture multiple modes of operation of the sensor nodes. Data communication networks is a model which offers an aggregated description of data flow through the sensor network. The paper offers a technique to systematically design adaptation policies of the sensor nodes in the network while co-optimizing the sensing, processing and networking of the embedded nodes.

The concept of Visual Programming (VP) was arguably proposed in the 80s [7], however, it is only recently that its advantages for embedded applications became apparent. VP languages have been proposed for applications like managing smart oilfields, vehicle tracking, contour finding, environmental monitoring, etc.

Sensor data are generally collected to determine what is going on during specific dynamic processes [5]. These processes are quite often very complex in nature and cannot be run without spending large resources unless they are simulated in some way. A simulation framework has been used to demonstrate how a number of decision support tools (services), of which the query languages are the most powerful and general, can be used to monitor dynamic processes. A central part of this framework is a scenario engine that keeps track of a large number of events. The input data to the simulator framework comes from a set of sensor models. That means that for each sensor type there is a piece of software that simulates the real sensor and generates data corresponding to the ongoing situations. As a consequence, a scenario with a number of ongoing events can be run while applying the various decision support tools over time.

Region Streams [8] is a functional macro-programming language for sensor networks. The specification model is based on successive filtering and functional processing of data pools. The data model is based on continuous data streams sampled from the environment and groups of nodes defined by their specific interests in space and over time. Language constructs enable aggregation of the data streams from a region and application of a function to the streams in a region. Abstract Task Graphs [4] is also a functional specification in which tasks sample from and place data into data pools. There is no other type of interaction between tasks. Channels act as filters for associating to a task only specific data from the pool. Tasks are executed periodically or when input data is available.

Semantic Streams [11] implements a query-based programming paradigm, which fits well applications in which sensor networks operate as large distributed databases. Queries formulated as logic programs are converted by the compiler into a service graph for the network. Data sensing is modeled as streams. Other constructs include filtering by specifying properties of the streams, defining regions and sub-regions of the physical space, and performance requirements (e.g., quality of service). Kairos [6] proposes a set of language-independent extensions for describing global behavior of sensor networks controlled centrally. The extensions assume shared memory to allow any node to iterate through its neighbors and address arbitrary nodes.

An ontology based semantic mediation approach is used to enable spatial system interoperability which is essential for many applications including spatial decision support systems that require integration of traditional and spatial information systems [12]. Interoperability involves accessing, manipulating and sharing data across heterogeneous systems and requires semantic mediation among the sources. The semantic mediation framework consists of ontology based descriptions of content and contexts, information mediation components and query processors. The query processors reformulate and submit queries on local data repositories and combine the results of the submitted sub-queries. The mediation component searches for relevant data and reconciles the semantic differences among the data sources.

Future large scale Networked Sensor Systems(NSSs) will demand concurrent execution of protocols like positioning, topology maintenance, medium access control, time synchronization, calibration, error detection, routing and the application level functionality making it difficult to optimize the design and also ensuring correct operation resulting in the need for systematic methodologies for designing algorithms for NSS applications. A systematic algorithm design methodology is proposed [13] which enables domain experts to design, analyze and optimize algorithms based on an abstract network model without requiring the knowledge of lower level networking and the hardware aspects of the system. A simple case study using two models, namely Collision Free Model (CFM) and Collision Aware Model (CAM) is discussed.

A Model based Integrated Simulation framework (MILAN) [16] is introduced to facilitate embedded system design and optimization. MILAN facilitates seamless integration of a variety of simulators at multiple levels of granularity into the framework. A single GUI allows designers to specify different aspects of embedded system hardware, software and performance requirements. The results of the individual simulators are interpreted in the global context to provide system wide estimates of different performance metrics. More emphasis is given to power estimation and optimization. The architectures modeled in MILAN consist of tightly-coupled, heterogeneous, digital components or SoC architectures.

The ability to deploy unmanned surveillance missions by using wireless sensor networks is of great importance in military applications [19] and the focus is to acquire and verify information about enemy capabilities and positions of hostile targets. The article describes the design and implementation of a complete running system called VigilNet for energy-efficient surveillance. VigilNet contains a group of cooperating sensor devices to detect and track the positions of moving vehicles in an energy-efficient and stealthy manner. Other related work for suveillance include [20] which includes an application involving Intrusion detection and the related problems of classifying and tracking targets. [22] presents an ad-hoc wireless sensor network-based system which detects and accurately locates shooters in urban environments. The performance of the system is superior to that of centralized countersniper systems in dense urban terrains.

Sensor networks are also widely used in Habitat and environmental monitoring [21]. The focus is mainly on nodal and network performance, with an emphasis on lifetime, reliability, and the static and dynamic aspects of single and multi-hop netwoks. Another application on military surveillance and environmental monitoring [23] includes a wireless sensor network which deploys heterogeneous collections of sensors capable of observing and reporting on various dynamic properties of their surroundings. Such systems suffer bandwidth, energy and throughput constraints that limit the quantity of information transferred from end to end. Mechanisms o perform data-centric aggregation utilizing application-specific knowledge provide a means to augmenting throughput but have limitations due to their lack of adaptation and reliance on application-specific decisions.

Chapter 7

Conclusion

This dissertation focuses on the development of a reconfigurable grid type sensor network using reconfigurable System on Chip (SoC) as the reconfigurable nodes. The implementation is specific to regions within the network and the nodes operate with respect to the parameters defined for the regions. The design flow includes two main steps: (i)defining regions & the associated parameters, goals and events with the help of command packets through the server and (ii)start execution which includes sensing, processing and networking.

The regions and the associated parameters are defined by the user through the server which sends the commands to the Entry Point which broadcasts the commands to the network using scheme A. The associated paramters include *Target Point*, *Path*, *Path Probability*, *Aggregation Function* and *Precision.* The nodes after receiving the command packets check if the data in the packet matches the co-ordinates of the node and accordingly update the data structure before sending the packet to the neighboring node(s). After defining the regions and the parameters, the user defines the goals associated with the regions and also the events.

After defining all the parameters and goals for the regions, the user gives the command to start execution. If the node is not the Target Point, it performs the functionality of sensing, performing the aggregation function, selecting a path to transmit the aggregated data to the Target Point, checking for events and generating actuation signals depending on the goals set for the region. The execution flow is as shown in figure 5.1. The nodes stop execution only on network reset. If the node is the Target Point, it does not perform any sensing or processing and performs the functionality of collecting data from all other nodes in the region to form a data pool. The Target Point periodically sends this information of all the nodes in the region to the server.

Appendix A refers to a case study which illustrates the proposed methodology of execution. A major part of the code was written in Assembly language for optimization. A detailed analysis of the clock cycles taken by each routine to execute and also the average power consumed by each routine is given in chapter 4 and chapter 5. A trade-off analysis between bandwidth and accuracy was performed. Accuracy in this case relates to the loss of data and the ability to implement a wide range of aggregation functions and precision. Bandwidth relates to the amount of data communication that takes place in the network and the number of clock cycles required to implement individual routines. The data communication in the network during execution is in the form of Information packets. The Information packets are the three types of packets namely the *Data Packets* which send the aggregated data from the individual nodes to the Target Point, the *Data Pool Packets* which send information of all the nodes from the Target Point to the server and the *Event Packets* which are used by the nodes to inform the Target Point about the occurrence of an event. The trade-off analysis was done specific to the case study which resulted in an accurate, optimized and power efficient implementation with acceptable bandwidth.

Bibliography

[1] Varun Subramanian, M.Wang, A. Doboli, 2008, *Towards A Model and Specification for Visual Programming of Massively Distributed Embedded Systems*, to IEEE International Workshop on Robotic and Sensors Environments.

[2] Varun Subramanian and A. Doboli, 2008, *Online Adaptation Policy Design for Grid Sensor Networks with Reconfigurable Embedded Nodes*, Design, Automation and Test in Europe (DATE).

[3] Batini, C. et al. *Visual languages and quality evaluation in multichannel adaptive information systems*, Journal of Visual Languages Computing, 18 (2007), 513-522.

[4] Bakshi, A, Prasanna, V. et al. *The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems*, Proc. EESR (2005).

[5] Camara, K., Jungert, E. *A visual query language for dynamic processes applied to a scenario deiven environment*, Journal of Visual Languages Computing, 18 (2007), 315-338.

[6] Gummadi, K. et al. *Macro-programming Wireless Sensor Networks using Kairos*, Proc. Int'l. Conference on Distributed Computing in Sensor Systems (2005).

[7] Johnston W. et al. *Advances in Dataflow Programming Languages*, ACM Computing Surveys, Vol. 36, No. 1, (March 2004), 1-34.

[8] Newton, R., Welsh, M. *Region Streams: Functional Macroprogramming for Sensor Networks*, Proc. Workshop on Data Management for Sensor Networks (2004).

[9] Soma R. et al. *A Semantic Framework for Integrated Asset Management in Smart Oilfields*, IEEE Symposium on Cluster Computing and the Grid (2007), 119-126

[10] Wache, H et al. *Ontology-Based Integration of Information - A Survey of Existing Approaches*, Proc. IJCAI - 01 Workshop: Ontologies and Information Sharing (2001).

[11] Whitehouse, K., Zhao, F., Liu, J. *Semantic Streams: a Framework for Declarative Queries and Automatic Data Interpretation*, Technical Report, Microsoft Research, MSR-TR-2005-45 (2005).

[12] Yetagnon, K. et al. *A Web-centric semantic mediation approach for spatial information systems*, Journal of Visual Languages Computing, Elsevier, 17 (2006), 1-24.

[13] Yu, Y. et al. *On Communication Models for Algorithm Design in Networked Sensor Sysemts: A Case Study*, Pervasive and Mobile, 1, Issue 1 (2005), 95-121.

[14] Zhang, C., Bakshi, A., Bakshi, V. *ModelIML: a Markup Language for Automatic Model Synthesis*, IEEE Conf. Information Reuse and Integration (2007), 317-322.

[15] Bakshi, A., Prasanna, V., Ledeczi, A. *A Model Based Integrated Simulation Framework for Design of Embedded Systems*, ACM SIG-PLAN Notices (2001).

[16] Lange, C., Wijns, M., Chaudron, M. *Supporting task-oriented modeling using interactive UML views*, Journal of Visual Languages and Computing, 18 (2007), 399-419.

[17] Passino, K. *Biomimicry for Optimization, Control, and Automation*, Springer (2005).

[18] Eles, P. et al. *Scheduling with Bus Access Optimization for Distributed Embedded Systems*, IEEE Transactions on VLSI Systems, VOl. 8, No. 5 (Oct 2000), 472-491.

[19] Tian He, Sudha Krishnamurthy et al. *VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance.*

[20] A. Arora, P. Dutta, S. Bapat et al. *A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking.*

[21] Robert Szewczyk, Alan Mainwaring et al. *An Analysis of a Larget Scale Habitat Monitoring Application*,

[22] Gyula Simon, Miklos Maroti et al. *Sensor Network - Based Countersniper System*,

[23] Tian He, Brian M. Blum et al. *AIDA: Adaptive Application-Independent Data Aggregation in Wireless Sensor Networks*,

[24] Tian He, John A Stankovic et al. *SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks*,

[25] Varun Subramanian, Michael Gilberti, Alex Doboli, Daniel Curiac, Dan Pescaru. *A Goal-Oriented Programming Model and Middleware Execution Support for Grid Sensor Networks with Recon?gurable Embedded Nodes*, to IEEE Transactions on Industrial Informatics. Special Section on: "Real-Time and (Networked) Embedded Systems".

Appendix A

A Case Study

This section presents a case study on a PSoC Network. Experiments were performed on PSoC Networks with 9, 16 and 25 nodes respectively and the results are tabulated. Figures A.1, A.2 and A.3 show the regions and the parameters defined for the PSoC Networks with 9, 16 and 25 nodes.

This section discusses the execution flow and the number of clock cycles required to compute aggregated data and transmit it along the PSoC Network with 9 nodes. Experiments were also conducted on networks with 16 & 25 nodes and the results are tabulated. The implementation can also be extended to larger networks.

Figure A.1 shows the network with 9 nodes (3 rows and 3 columns) and two regions A and B are defined. The 2 nodes (1,1) and (2,1) highlighted in yellow color in the figure are common to both regions and hence, they will sense, compute data and check for events for both regions, one at a time. Also, the goals for these nodes which are common to regions A and B is the intersection of the goals for the 2 regions.

Table A.1 gives the number of clock cycles to define the commands. The best case is the case when the node that receives and processes the commands is a neighbor to the Entry Point. This will include the clock cycles taken for the command to reach node (0,1) and to process it. This node is the first node that receives the commands from the Entry Point. The worst case is the case where the node is at the top-right corner of the region. Node (2,2) is the top-right corner in figure A.1 and this node is the last node to receive the command from the Entry Point. The clock cycle analysis includes the number of clock cycles taken by each of the nodes highlighted in green color in figure A.1 to receive, process and transmit. This procedure can be mapped into networks with 16 and 25 nodes and Tables A.2 and A.3 gives the clock cycle analysis for these networks.

Table A.4 shows the number of clock cycles to sense data, compute the aggregated value and transmit the data to the Target Point. The critical path shown in figure A.1 is the longest path defined in the network. Node (0,1) is the Target Point which is highlighted in dark blue. The node which is neighbor to the Target Point and lies in the critical path is node (0,2) in figure A.1 and the best case section in Table A.4 shows the number of clock cycles taken by this node to sense data, compute aggregated data and transmit the data to the Target Point. The node (2,2) in the critical path is the farthest node from the Target Point. The worst case section in Table A.5 shows the number of clock cycles taken by node (2,2) to sense data, compute aggregated data & transmit data and also the number of clock cycles taken by the forwarding nodes in the path to forward this data to the Target Point. Table A.4 shows 2 cases where two different types of aggregation functions are used. In one case, data is sensed twice before aggregation and in the other case, data is sensed four times before aggregation. This procedure can also be mapped into networks with 16 and 25 nodes.

Table A.5 shows the number of clock cycles taken by the nodes to generate an event and also, transmit the occurrence of the event to the Target Point. The nodes use scheme B to transmit the event. The path following by scheme B in this case coincides with the critical path. Node (0,1) in the network is a neighbor to the Target Point and the best case section in Table A.5 shows the number of clock cycles taken by this node to generate an event and transmit the occurrence of the event to the Target Point. Node (2,2) is the farthest node in the region from the Target Point. The worst case section in Table A.5 shows the number of clock cycles taken by this node to generate an event and transmit the occurrence of the event to the Target Point and also the number of clock cycles taken by the forwarding nodes. This procedure can also be mapped into networks with 16 and 25 nodes.

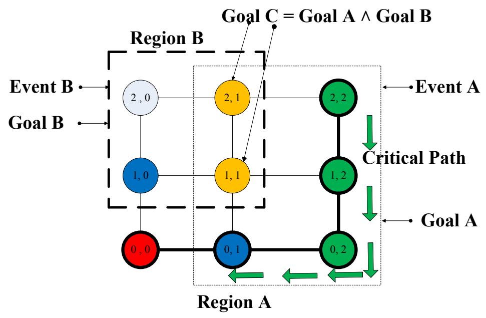

Figure A.1: PSoC Network with 9 nodes

Functionality	**Best Case**	**Worst Case**
Define Region	3200	12800
Define Target Point	3395	13580
Define Path	5838	20718
Define Path Probability	2361	9444
Define Aggregation Function	2676	10704
Define Precision	2253	9012
Define Region's Event	1984	7936
Define a Node's Event	2742	10968
Define Region's Range	2893	11572
Define a Node's Range	3633	14532

Table A.1: Clock Cycle Analysis for defining commands in a PSoC Network with 9 nodes

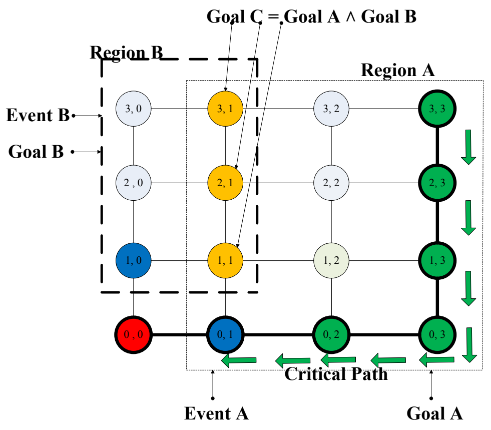

Figure A.2: PSoC Network with 16 nodes

Functionality	**Best Case**	**Worst Case**
Define Region	3200	19200
Define Target Point	4066	24396
Define Path	7474	39897
Define Path Probability	2361	14166
Define Aggregation Function	2676	16056
Define Precision	2253	13518
Define Region's Event	1984	11904
Define a Node's Event	2742	16452
Define Region's Range	2893	17358
Define a Node's Range	3633	21798

Table A.2: Clock Cycle Analysis for defining commands in a PSoC Network with 16 nodes

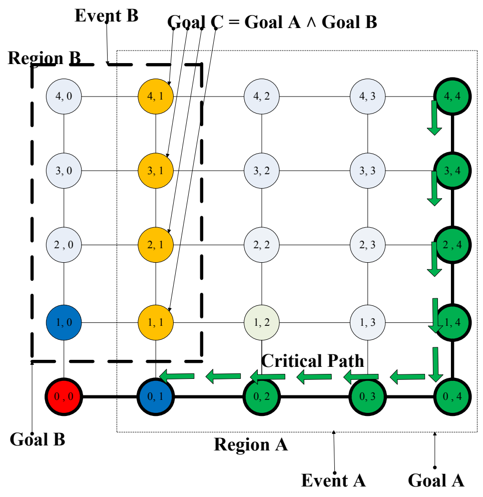

Figure A.3: PSoC Network with 25 nodes

Functionality	**Best Case**	**Worst Case**
Define Region	3200	25600
Define Target Point	4962	39696
Define Path	9110	64956
Define Path Probability	2361	18888
Define Aggregation Function	2676	21408
Define Precision	2253	18024
Define Region's Event	1984	15872
Define a Node's Event	2742	21936
Define Region's Range	2893	23144
Define a Node's Range	3633	29064

Table A.3: Clock Cycle Analysis for defining commands in a PSoC Network with 25 nodes

Number of nodes & Aggregation Function	**Best Case**	**Worst Case**
n = 9 & f = 2	131445	138906
n = 16 & f = 2	131445	143880
n = 25 & f = 2	131445	148854
n = 9 & f = 4	261039	268500
n = 16 & f = 4	261039	273474
n = 25 & f = 4	261039	278448

Table A.4: Clock Cycle Analysis for computing and transmitting data to Target Point in a PSoC Network

Number of nodes & Aggregation Function	**Best Case**	**Worst Case**
n = 9 & f = 2	131958	137127
n = 16 & f = 2	131958	140573
n = 25 & f = 2	131958	144019
n = 9 & f = 4	261552	266721
n = 16 & f = 4	261552	270167
n = 25 & f = 4	261552	273613

Table A.5: Clock Cycle Analysis for transmitting an event to Target Point in a PSoC Network