

OHIO NORTHERN UNIVERSITY
T.J. SMULL COLLEGE OF ENGINEERING
ECCS DEPARTMENT

SMOKE DETECTOR NETWORKS

BLUETOOTH DEVICE BASED PICONETS

Isaac Hirt
Lucas Hissong
Elliott Metzger

May 2, 2003

EXECUTIVE SUMMARY

This report describes a wireless smoke detector system aimed at the residential market. Current wireless smoke detector systems are typically high priced and focused towards commercial uses. Many also require a central control station that adds to the price and complexity of the system. This new system uses the Bluetooth v1.1 wireless medium. Bluetooth was chosen because of its low power consumption and relatively low cost. The development kits utilized are from Stonestreet One and were chosen for the system because of their software support and documentation, multiple serial interfaces, standalone power supply and Ericsson Bluetooth module v1.1.

System setup is as simple as popping a battery in each smoke detector and then placing them throughout a building or home; keeping in mind the 10 meter range of the Bluetooth modules. By integrating smoke detectors with Bluetooth modules, each smoke detector is capable of communicating directly with other Bluetooth-enabled smoke detectors; no central control station is needed. This functionality allows for all building occupants to be notified of a fire immediately, regardless of its location, and thereby reduces the number of fire-related injuries caused by people not hearing an alarm.

The architecture of each smoke detector in the system includes: one standard Kidde model 0916 smoke detector; three Xilinx XC9536 CPLDs, one for smoke detector control logic and two for Bluetooth interface logic; one MAX3100 UART controller chip from Maxim ICs; one Ericsson Bluetooth module v1.1 (currently using the Stonestreet One development kit); one crystal oscillator (currently using a wave generator); one 2N2222 transistor. Combinational logic was developed on the three XC9536s to provide an interface between the smoke detector, the UART and the Bluetooth module. The transistor was controlled by the combinational logic to connect the test leads of the smoke detector, therefore activating its alarm. The wave generator was used in place of a crystal, due to time constraints, to setup the clock frequency with which to synchronize the operation of the UART and combinational logic.

TABLE OF CONTENTS

EXECUTIVE SUMMARY	2
TABLE OF CONTENTS.....	3
INTRODUCTION.....	4
PROBLEM DEFINITION	4
ALTERNATE PRODUCTS	4
CONSTRAINT ANALYSIS	5
PREPARATION	6
PROJECT DEVELOPMENT.....	7
SMOKE DETECTOR.....	8
BLUETOOTH DEVELOPMENT BOARD.....	9
UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER	10
BLUETOOTH COMMANDS	11
COMBINATIONAL LOGIC.....	11
CONCLUSION.....	15
FUTURE CONSIDERATIONS.....	15
REFERENCES	17
APPENDIX.....	18

INTRODUCTION

Problem Definition

There is a need for a cost-efficient wireless smoke detector system to alert people in buildings that are not within the audible range of the current smoke detectors. Wireless smoke detectors would be able to alert people in the general vicinity of the fire and also send a signal to other smoke detectors in the network, activating their alarms as well.

These smoke detectors could use a variety of wireless communication methods. All of the wireless communication methods eliminate the cost of running a transfer medium, such as wire, through buildings. Wireless mediums remove the mobility restraints of building walls and make devices user-friendlier by removing the need to figure out where to plug each wire.

Bluetooth is our medium of choice to implement a wireless smoke detector network. Its security features, reliability, low cost, and very low power consumption make it the ideal platform for our development needs. Upon activation, the Bluetooth devices connect to each other to form a Bluetooth network, also known as a piconet, and communicate with each other about the status of the smoke detectors. The wireless smoke detector system we plan to develop will communicate on the piconet to determine if any detectors are in an alarm state and then turn on all other alarms in the piconet. This is necessary to alert all building occupants of the fire, not just those in the vicinity of the fire. The audibility of the horn in conventional smoke detectors is not sufficient over long distances. If smoke was detected in a room using wireless smoke detectors, an alarm signal would travel throughout the network setting off every alarm and alerting all of the building's occupants.

Alternate Products

The idea for a wireless smoke detector network is not a new idea. Numerous applications for this idea started to emerge not long after wireless technology became commonplace. The thinking behind this is that many smoke detectors/alarms could be networked to a central location that could act as a main control system for an entire building, or even several buildings. The control system could be located in the main security office of the building, the central building of a group of buildings, or even the local fire department. The cost saved in manpower and wiring, makes wireless smoke detectors/alarms an attractive alternative to conventional smoke detectors/alarms. A discussion of two examples of currently available wireless smoke detectors/alarms follows.

The first product is produced by Digital Security Controls Ltd. Security Products. DSC Security Products is a manufacturer of intrusion alarm systems and detection devices in the world. DSC's product, the WLS906-433 Meridian smoke detector, is a 433 MHz wireless smoke detector. Among its features is an 85-decibel piezoelectric alarm horn, a high immunity to radio frequency interference to help eliminate false alarms, and a patented smoke chamber with an advanced 90 degree photoelectric detection pattern, which provides accurate and early detection of smoke. One of the drawbacks of this product is its high price. Another drawback to this product is that it works optimally with a central system, which would have to be either bought from DSC or another system would have to be procured that had similar specifications [1].

The second product is produced by ITI, a company that designs and manufactures wireless security systems as well as other wireless products. ITI's area of design interest is mostly radio frequency (RF) design and manufacturing. The ESL Residential Wireless Smoke Alarm by ITI is described as the industry's first wireless smoke alarm with analog features such as remote maintenance/trouble reporting and drift compensation. Among the features of the ESL Residential Wireless smoke alarm are: a Photoelectric smoke sensor, a Built-in Sounder and fixed temperature/Rate-of-rise heat sensor, ITI's patented Learn mode technology, a Field replaceable optical sensor, immunity to dust, and Clean Me™ self-diagnostics feature performs a daily full diagnostic test. However, this product also has drawbacks. One of which is that one must be a registered user of their site to place an order. (A registered user does not imply they cannot sell few of their products: Rephrase this sentence) which means that they cannot buy their product if only a few are required (for a new home or some other small-scale project). Another disadvantage is that the product is not targeted for residential use, but rather for commercial and large-scale use [2].

CONSTRAINT ANALYSIS

There are many different constraints that affect how the wireless smoke detector system is developed. These shape the method in which the product is designed and the way that it is used. The economic constraints placed on the smoke detector network focuses on keeping them at a marketable price in relation to its competition. This means concern needs to be put into manufacturability and developmental costs so the product will sell. There are no environmental

PROJECT DEVELOPMENT

Developing the smoke detector network consisted of breaking the design into smaller, workable parts. The current implementation of the wireless smoke detector is constructed using a standard smoke detector, a universal asynchronous receiver transmitter (UART), a Bluetooth development board, and combinational logic that contains both control logic and interface logic. A block diagram showing these blocks and how they link together can be seen in Figure 2 below.

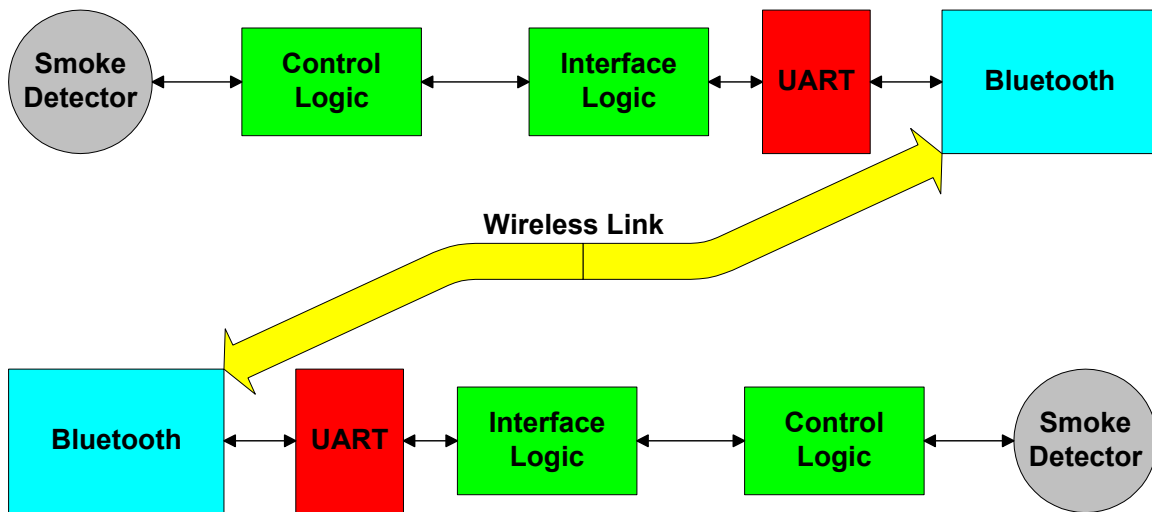


Figure 2: Problem Block Diagram

Each block has a specific function that needs to be integrated within the system. The smoke detector handles the inputs and outputs. It contains a 'test' button, the smoke sensor input, and the piezoelectric horn output. It is linked to the combinational logic by lines that send the smoke detector into alert mode and a line that tells the logic that smoke is detected. The combinational logic handles the operation of the system. It determines whether the devices need to be in alert mode, as well as setting up the piconet to which the wireless smoke detectors belong. A UART is used to connect the combinational logic and the Bluetooth development board. This is put in place to help handle the handshaking between the two devices. Following this is the Bluetooth development board. This is our wireless medium that maintains the piconet, handles security, reliability, and data transfer amongst devices. The combination of these blocks creates a system complete with security, reliability, and power consumption considerations.

SMOKE DETECTOR

The smoke detector chosen for this application, Kidde model 0916, is considered to be a standard off-the-shelf model that has the common features of test and reset buttons, as well as a LED to show it has battery power. For this application, it is modified to send a signal to the combinational logic that informs the system that the smoke detector is in alert mode, as well as an input that initiates the smoke detector to enter alert mode. This is shown in Figure 3. Alterations to the smoke detector became a problem because of the manufacturer's refusal to provide any details on its operation. As a result, reverse engineering was performed to analyze the voltage levels on the printed circuit board pins. To add this functionality, the smoke detector was disassembled and analyzed to determine how it could be best optimized for this application. Pins integrated in the smoke chamber were probed, and one was found that changes logic levels when the smoke detector enters alert mode. This pin was selected to be the input to the combinational logic. Probing the smoke detector in this manner was considered the best option because the smoke chamber used is commonly implemented in many other smoke detection devices. The test button was used to tell the smoke detector to enter the alert mode. Two leads were soldered over the two contacts for the test button and were then connected to a 2N2222 transistor that was controlled by the combinational logic. These modifications prepared the smoke detector to link to combinational logic.

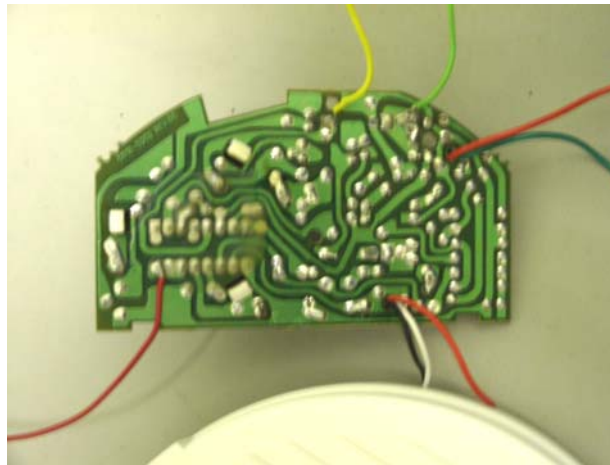


Figure 3: Smoke Detector Modifications

BLUETOOTH DEVELOPMENT BOARD

The Bluetooth development board chosen for this design is the Stonestreet One DP series that features an Ericsson Bluetooth module. Many considerations were given when selecting which development board to use. These considerations include: price, power consumption, software package, and which Bluetooth version was used. The highest weight was given to the price, due to the budget constraints of the project. The next highest weight was given to what software package was included with the development board because of the need for good software and documentation so that the Bluetooth protocol could be understood and implemented. Power consumption is given a minor role, as all the Bluetooth development boards have low power consumption, but still the lower the power consumption, the better. The lowest priority was given to the version of Bluetooth that was used. Although there are some incompatibility issues between Bluetooth version 1.0B and version 1.1, this system is merely a proof of concept and either version would be acceptable.

Table 1: Development Board Decision Matrix

		Bluetooth Development Boards							
		Teleca Comtec		Stonestreet One		GCT		Atmel	
Selection Criteria	Weight	Rating	Weighted Score	Rating	Weighted Score	Rating	Weighted Score	Rating	Weighted Score
Price	40	4	1.6	3	1.2	1	0.4	1	0.4
Power	15	4	0.6	4	0.6	4	0.6	1	0.15
Software	35	2	0.7	4	1.4	3	1.05	2	0.7
Version	10	1	0.1	4	0.4	4	0.4	4	0.4
Total Score		3		3.6		2.45		1.65	
Rank		2		1		3		4	

UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER

The Stonestreet One Bluetooth development board that we chose to use has three different forms of serial communication: RS232, UART and USB (Universal Serial Bus). In order to choose an input type, a decision matrix was used based on communication speed, communication protocol and ease of connecting to the Bluetooth development board. The lowest weight was given to the communication speed because if the connection type can handle communication at a 57600 baud rate, then it is fast enough for this application. The next lowest score was given to the ease of connectivity due to the fact that the connection only needs to be done once. With the communication protocol being the most important due to it being what gets dealt with when communication with the Bluetooth development board.

Table 2: Decision matrix for which type of communication interface to use

Communication Interface							
		RS232		UART		USB	
Selection Criteria	Weight	Rating	Weighted Score	Rating	Weighted Score	Rating	Weighted Score
Communication speed	10	2	0.2	1	0.1	3	0.3
Communication Protocol	60	2	1.2	3	1.8	1	0.6
Ease of Connection	30	2	0.6	3	0.9	1	0.3
Total Score		2		2.8		1.2	
Rank		2		1		3	

The UART was chosen to be the communication interface between the Bluetooth device and the combinational logic. To send information on the UART a communication protocol must be followed, the appropriate RTS (request to send) and CTS (clear to send) signals need to be high, and the combinational logic and the UART need to be synchronized; this happens via a clock signal produced by the Bluetooth development board. To establish this communication between the combinational logic and the UART interface on the Bluetooth development board, another integrated circuit was added, the UART controller. This UART controller takes information that is given to it by the interface logic in a serial manner, and converts it into the appropriate signal sets for the UART interface with the

Bluetooth device. This is done by waiting for an interrupt to be received at the interface logic from the UART controller, which signals that either data has arrived or more data can be sent. A mask is then used to determine which interrupt has actually been sent, and the appropriate response is taken.

BLUETOOTH COMMANDS

In order to communicate with the Bluetooth module, data must be sent in the form of HCI (Host Controller Interface) command packets. The HCI is an interface which allows data to be sent to the Bluetooth module via RS232, I2C, USB or UART. By compiling and running the provided Bluetooth test software provided by Stonestreet One, it was possible to determine which codes were being sent to the Bluetooth module during each phase of communication. First off, a serial port listener was used to interpret exactly which opcodes were being sent to the Bluetooth module. After identifying each function of Bluetooth communication (e.g. reset, scan, inquiry, connect, data transfer) with a set of opcodes, a program called RS232 Hex Com Tool v3.0 was used. Hex Com Tool merely sends data in hexadecimal form across the RS232 connection of a PC. By sending these opcodes to the Bluetooth module via the PC's RS232 connection, it was possible to emulate the combinational logic that would be sending the same opcodes via a serial communication link (UART). Upon verification that Bluetooth communication was operational without the use of the development software, the process of developing combinational logic to handle the smoke detector/Bluetooth interface began.

COMBINATIONAL LOGIC

Combinational logic is used between the smoke detector and the Bluetooth development board to control the system. The available programmable logic devices, XC9536 CPLDs, are used to implement the logic. The combinational logic is broken into two parts, the control logic and the interface logic. These two parts work together to communicate with the Bluetooth development board through the UART.

Control Logic

The control logic is designed to set up a Bluetooth device upon power up, and communicate with the network on Bluetooth enabled smoke detectors about the current status of the piconet and if there is a fire detected to alert the network to sound the fire alarms. The control logic was designed using VHDL to create a finite state machine that uses clock signal to synchronize the control logic with the other circuitry in the smoke detector. The control logic state diagram is displayed in Figure 4. To send commands to the Bluetooth device which is being controlled, a control signal needs to be sent to the interface logic which will interpret the control signal and translate it into a series of UART commands.

In the final product the control logic would only have one direct input, a reset button. But in its development phase there are 9 inputs, which are used to simulate response signals from the interface logic. These inputs control when the control logic will change states and send control signals to the interface logic.

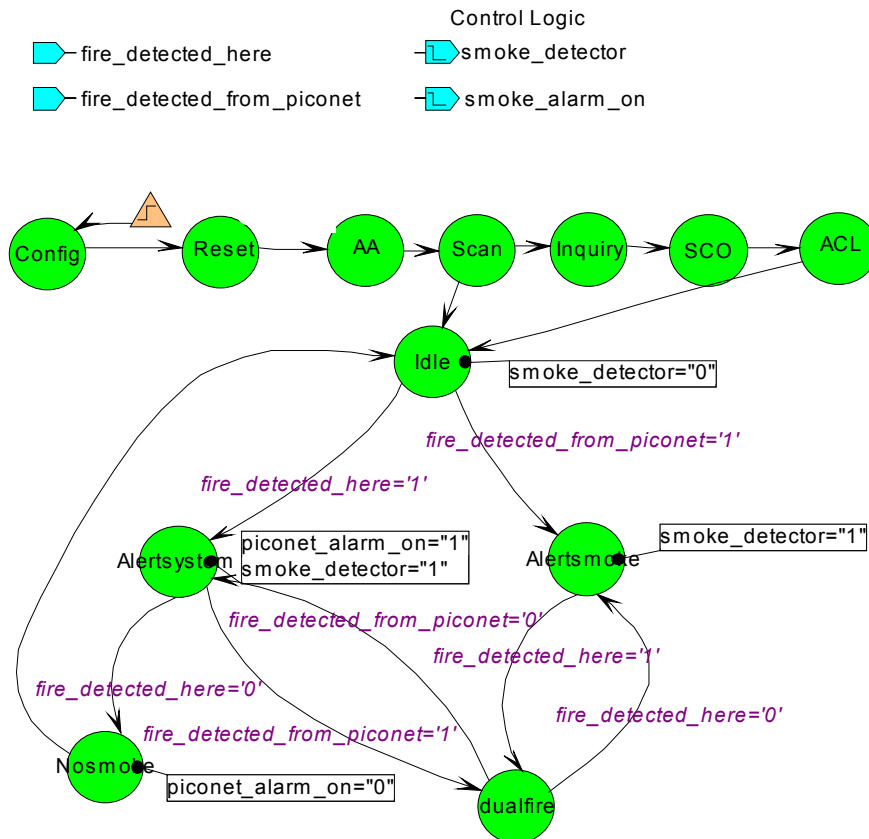


Figure 4: Control Logic State Diagram

When the smoke detector is powered on or reset, the control logic needs to initialize communication of the Bluetooth device. To accomplish this, a series of commands must be sent to the Bluetooth module to instruct it to form or join a piconet. Each device must first enter auto accept mode, which allows a device to connect to other devices without user input. Once auto accept mode has been turned on, the devices are free to begin scouring the radio waves to find another device with which to connect. Upon finding another device, the commands to setup the piconet will be executed.

Once communication has been established, the control logic places itself in a state where it will wait for fires to be detected on the network or from its smoke detector. If a fire is detected at the smoke detector the control logic enters a state where the piconet has been alerted of a fire on the network, but the control logic does not need to sound the alarm at its smoke detector because that has already been triggered. If a fire is detected by another smoke detector on the network, the control logic will enter another fire detected state, and sound the alarm so the occupants of the building are alerted of a fire in the building. If fire is detected both at the smoke detector and by another smoke detector on the network the control logic will place itself in a state of multiple fires detected, so when one of these fires is no longer being detected the alarm will not briefly shut off as there is still a fire in the building.

When connected on a breadboard with the 2N222 transistor and the smoke detector, the control logic functioned correctly.

Interface Logic

The interface logic is designed to translate the control signals sent to it by the control logic into the operation codes that the Bluetooth development board requires. To do this, the codes must be encapsulated for the UART that lies between the two blocks. The UART requires certain bits to be set so that data writing is possible. The 8-bit Bluetooth operation codes are put in a 16-bit packet that the UART can read. They are then sent in parallel to a shift register that outputs them in a serial manner for the UART. On every rise of the interrupt, the interface block reads the control lines and sends the correct operation code to the shift register, which loads them in the data_in lines and starts shifting through them at the output with the most significant bit.

The interface block of the interface logic is written in VHDL as a state machine. The schematic entry for the interface logic can be seen in Figure 5 below.

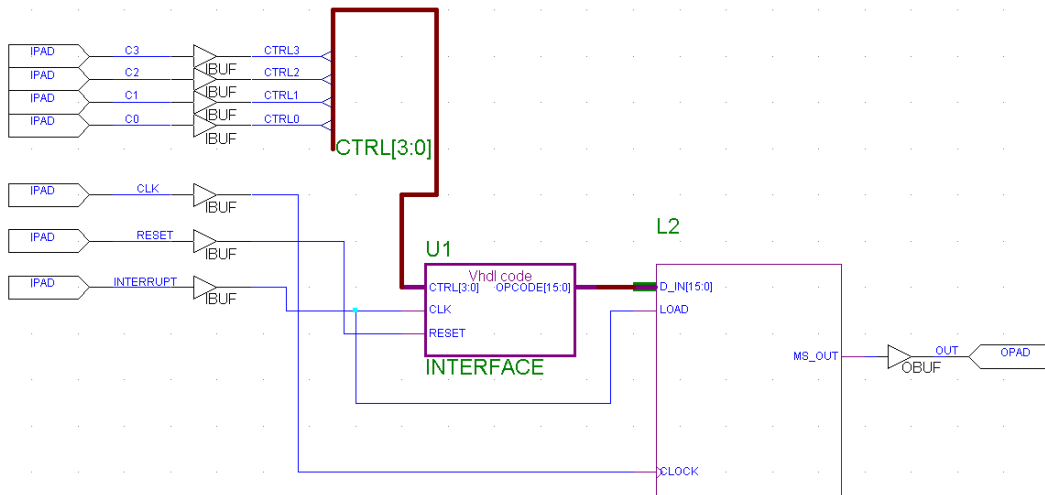


Figure 5: Interface Logic

When given a ready to send interrupt from the UART, the interface logic reads the next intended control operation from the control lines, and then steps through the different 16-bit packets of information for each control operation. Depending on the operation, there can be up to seventeen control packet states that need to be stepped through. The logic keeps track of this using the different states of the FSM.

The shift register part of the interface logic is generated using the LogiBlox macro-generator. Using this, the bit size of the shift register, shift direction, and output are selected to fit this application. The macro is then linked to the interface in a schematic entry.

After entering both blocks in schematic entry, simulations are run to ensure the blocks work in the way that they are designed. A clock rate is assigned for the clock input and the other inputs are raised using assigned keys. Its function is then verified and implementation is begun.

Special consideration is given to the implementation of the interface logic. It is too large to fit on one XC9536 CPLD, so different solutions to the problem are developed. The first is to split the design into two parts; one XC9536 CPLD holds the interface, and the other holds the shift register. Another solution is to

purchase a larger CPLD or FPGA to fit the existing design. The last solution is to redesign the shift register LogiBlox macro, because of the inefficiencies present when using LogiBlox. The first solution is determined to be the best due to the design budgetary issues as well as the time constraints given. When split apart, the blocks fit easily on two XC9536 CPLDs. When connected on a breadboard and tested, they functioned correctly.

CONCLUSION

During the design process, several goals were accomplished. First off, smoke detector circuit properties were determined and appropriate connections were created. HCI command packets were broken down into opcodes to determine how to setup the piconet through which the devices could communicate. The opcodes were then used to verify data transfer across the piconet. Combinational logic for the system was developed to control the smoke detector and to provide an interface between the smoke detector and the Bluetooth module. Finally, the combinational logic was tested and verified to work correctly.

FUTURE CONSIDERATIONS

There were a few goals that were not achieved during the design process. One such unfulfilled goal was to verify the correct operation of all the blocks integrated into one system. Currently the system is being tested using a wave generator in place of a crystal to verify that the system will indeed work, but has not yet been verified. Another goal that was not achieved was the integration of all the combinational logic onto a single programmable chip. In the future, it would be ideal to place all the logic onto a single FPGA or PROM of some type.

There were also a few new ideas that may enhance the functionality and operation of the system. First off, it would be beneficial to be able to differentiate between alarm signals coming from the system to identify where in the building the fire originated. Along those same lines, the development of a central monitoring station would provide a graphical display of where the fires are being detected in the system. Finally, by gaining better understanding of the UART interface, it will be possible to develop combinational logic on the same programmable chip to

deal with flow control and eliminate the need for the external UART controller. If all the logic functionality is integrated on a single integrated circuit, the cost of the smoke detector will be significantly reduced. If these devices are to be produced in high volume, then the control logic as well as Bluetooth module can be integrated onto a single chip, Bluetooth-enabled, smoke detection system.

REFERENCES

1. DSC; <http://www.dsc.com>. [Accessed 20 October 2002]
2. GE Interlogix; <http://www.itii.com/index.jsp>. [Accessed 20 October 2002]

Appendix B – Control Logic State Diagram

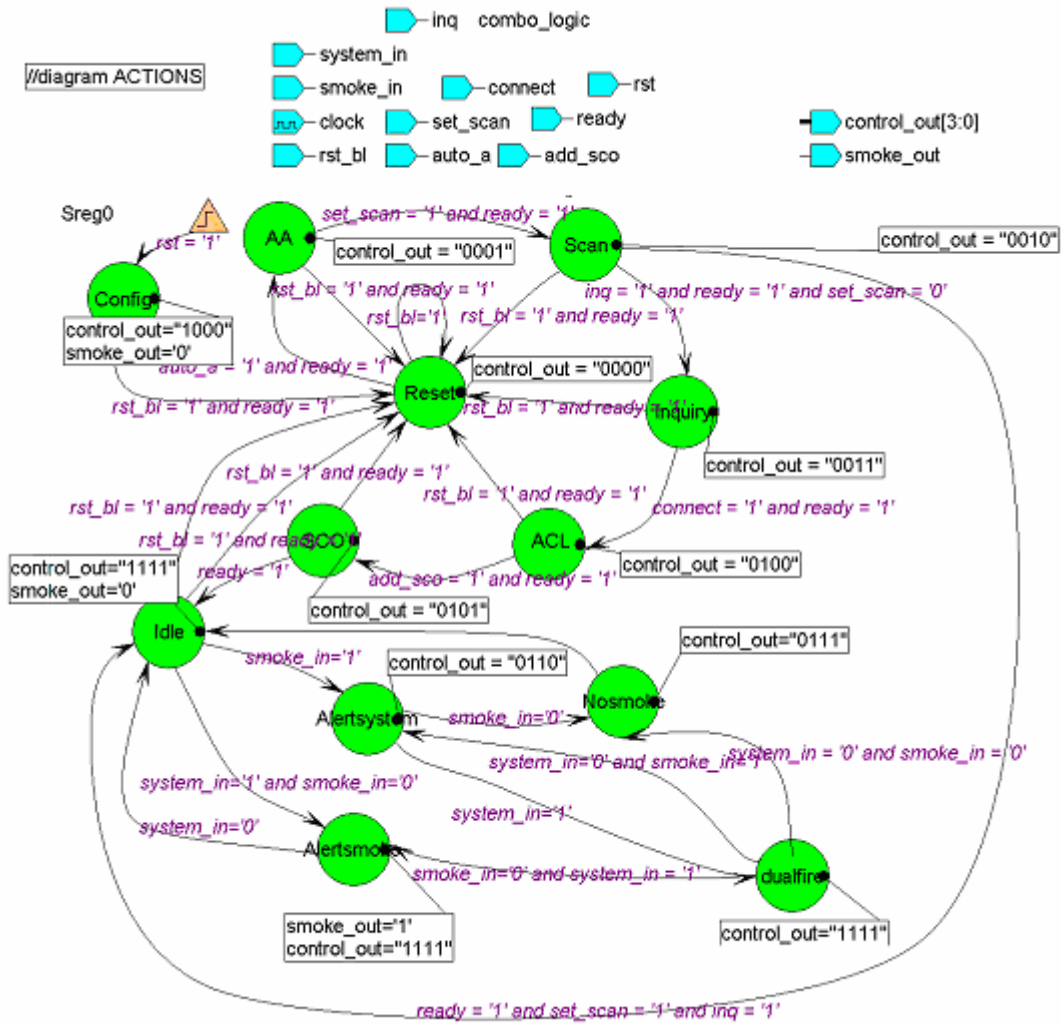


Figure 7: Control Logic State Diagram

Appendix C – VHDL code for Control Logic

```
--
-- File: D:\combo\combo_logic.vhd
-- created: 04/15/03 09:32:25
-- from: 'D:\combo\combo_logic.ASF'
-- by fsm2hdl - version: 2.0.1.60
--
library IEEE;
use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity combo_logic is
  port (add_sco: in STD_LOGIC;
        auto_a: in STD_LOGIC;
        clock: in STD_LOGIC;
        connect: in STD_LOGIC;
        inq: in STD_LOGIC;
        ready: in STD_LOGIC;
        rst: in STD_LOGIC;
        rst_bl: in STD_LOGIC;
        set_scan: in STD_LOGIC;
        smoke_in: in STD_LOGIC;
        system_in: in STD_LOGIC;
        control_out: out STD_LOGIC_VECTOR (3 downto 0);
        smoke_out: out STD_LOGIC);
end;

architecture combo_logic_arch of combo_logic is

  -- SYMBOLIC ENCODED state machine: Sreg0
  type Sreg0_type is (AA, ACL, Alertsmoke, Alertsystem, Config, dualfire, Idle, Inquiry,
  Nosmoke, Reset, Scan, SCO);
  signal Sreg0: Sreg0_type;

  begin
  -- concurrent signals assignments

  Sreg0_machine: process (clock)
```

```

begin

if clock'event and clock = '1' then
  if rst = '1' then
    Sreg0 <= Config;
  else
    case Sreg0 is
      when AA =>
        if rst_bl = '1' and ready = '1' then
          Sreg0 <= Reset;
        elsif set_scan = '1' and ready = '1' then
          Sreg0 <= Scan;
        end if;
      when ACL =>
        if add_sco = '1' and ready = '1' then
          Sreg0 <= SCO;
        elsif rst_bl = '1' and ready = '1' then
          Sreg0 <= Reset;
        end if;
      when Alertsmoke =>
        if system_in='0' then
          Sreg0 <= Idle;
        end if;
      when Alertsystem =>
        if system_in='1' then
          Sreg0 <= dualfire;
        elsif smoke_in='0' then
          Sreg0 <= Nosmoke;
        end if;
      when Config =>
        if rst_bl = '1' and ready = '1' then
          Sreg0 <= Reset;
        end if;
      when dualfire =>
        if smoke_in='0' and system_in = '1' then
          Sreg0 <= Alertsmoke;
        elsif system_in = '0' and smoke_in = '0' then
          Sreg0 <= Nosmoke;
        elsif system_in='0' and smoke_in='1' then
          Sreg0 <= Alertsystem;
        end if;
      when Idle =>
        if smoke_in='1' then
          Sreg0 <= Alertsystem;
        elsif system_in='1' and smoke_in='0' then

```

```

        Sreg0 <= Alertsmoke;
    elsif rst_bl = '1' and ready = '1' then
        Sreg0 <= Reset;
    elsif rst_bl = '1' and ready = '1' then
        Sreg0 <= Reset;
    end if;
when Inquiry =>
    if connect = '1' and ready = '1' then
        Sreg0 <= ACL;
    elsif rst_bl = '1' and ready = '1' then
        Sreg0 <= Reset;
    end if;
when Nosmoke =>
    Sreg0 <= Idle;
when Reset =>
    if rst_bl='1' then
        Sreg0 <= Reset;
    elsif auto_a = '1' and ready = '1' then
        Sreg0 <= AA;
    end if;
when Scan =>
    if rst_bl = '1' and ready = '1' then
        Sreg0 <= Reset;
    elsif ready = '1' and set_scan = '1' and inq = '1' then
        Sreg0 <= Idle;
    elsif inq = '1' and ready = '1' and set_scan = '0' then
        Sreg0 <= Inquiry;
    end if;
when SCO =>
    if ready = '1' then
        Sreg0 <= Idle;
    elsif rst_bl = '1' and ready = '1' then
        Sreg0 <= Reset;
    end if;
when others =>
    null;
end case;
end if;
end process;

```

-- signal assignment statements for combinatorial outputs

control_out_assignment:

```

control_out <= "0001" when (Sreg0 = AA) else
    "0100" when (Sreg0 = ACL) else
    "1111" when (Sreg0 = Alertsmoke) else
    "0110" when (Sreg0 = Alertsystem) else

```

```
"1111" when (Sreg0 = dualfire) else  
"1111" when (Sreg0 = Idle) else  
"0011" when (Sreg0 = Inquiry) else  
"0111" when (Sreg0 = Nosmoke) else  
"0000" when (Sreg0 = Reset) else  
"0010" when (Sreg0 = Scan) else  
"0101" when (Sreg0 = SCO) else  
"1000";
```

```
smoke_out_assignment:  
smoke_out <= '1' when (Sreg0 = Alertsmoke) else  
    '0' when (Sreg0 = Idle) else  
    '0';
```

```
end combo_logic_arch;
```

Appendix D – VHDL Code for Interface Logic

```
library IEEE;
use IEEE.std_logic_1164.all;
entity interface is
port( reset, clk : in std_logic;
      ctrl : in std_logic_vector(3 downto 0);
      opcode : out std_logic_vector(15 downto 0) );
end entity interface;

architecture behavioral of interface is
type statetype is (state0, state2, state3, state4, state5, state6,
                  state7, state8, state9, state10, state11, state12, state13,
                  state14, state15, state16, state17);
signal state : statetype;
begin
process(clk)
begin

if clk'event and clk='1' then
  if reset = '1' then
    state <= state0;
  else
case state is
when state0 =>
case ctrl is
  when "0000" => --RESET
    opcode <= "1000000000000001";
  when "0001" => --AA
    opcode <= "1000000000000001";
  when "0010" => --SCAN
    opcode <= "1000000000000001";
  when "0011" => --INQUIRY
    opcode <= "1000000000000001";
  when "0100" => --CONNECT ACL
    opcode <= "1000000000000001";
  when "0101" => --ADD SCO
    opcode <= "1000000000000001";
  when "0110" => --ALERT
    opcode <= "1000000000000010";
  when "0111" => --CLEAR
    opcode <= "1000000000000010";
  when others => --CONFIG
    opcode <= "1000000000000000";
end case;
state <= state2;
```

```

when state2 =>
case ctrl is
  when "0000" => --RESET
    opcode <= "100000000000011";
  when "0001" => --AA
    opcode <= "100000000000101";
  when "0010" => --SCAN
    opcode <= "100000000011010";
  when "0011" => --INQUIRY
    opcode <= "100000000000001";
  when "0100" => --CONNECT ACL
    opcode <= "100000000000101";
  when "0101" => --ADD SCO
    opcode <= "100000000000111";
  when "0110" => --ALERT
    opcode <= "100000000000001";
  when "0111" => --CLEAR
    opcode <= "100000000000001";
  when others => --CONFIG
    opcode <= "100000000000000";
end case;
state <= state3;
when state3 =>
case ctrl is
  when "0000" => --RESET
    opcode <= "100000000001100";
  when "0001" => --AA
    opcode <= "100000000001100";
  when "0010" => --SCAN
    opcode <= "100000000001100";
  when "0011" => --INQUIRY
    opcode <= "100000000000100";
  when "0100" => --CONNECT ACL
    opcode <= "100000000000100";
  when "0101" => --ADD SCO
    opcode <= "100000000000100";
  when "0110" => --ALERT
    opcode <= "100000000100000";
  when "0111" => --CLEAR
    opcode <= "100000000100000";
  when others => --CONFIG
    opcode <= "100000000000000";
end case;
state <= state4;
when state4 =>

```

```

case ctrl is
  when "0000" => --RESET
    opcode <= "1000000000000000";
    state <= state0;
  when "0001" => --AA
    opcode <= "1000000000000011";
    state <= state5;
  when "0010" => --SCAN
    opcode <= "1000000000000001";
    state <= state5;
  when "0011" => --INQUIRY
    opcode <= "1000000000000101";
    state <= state5;
  when "0100" => --CONNECT ACL
    opcode <= "100000000001101";
    state <= state5;
  when "0101" => --ADD SCO
    opcode <= "100000000000100";
    state <= state5;
  when "0110" => --ALERT
    opcode <= "100000000000101";
    state <= state5;
  when "0111" => --CLEAR
    opcode <= "100000000000101";
    state <= state5;
  when others => --CONFIG
    opcode <= "1000000000000000";
    state <= state5;
end case;
when state5 =>
case ctrl is
  when "0001" => --AA
    opcode <= "1000000000000010";
    state <= state6;
  when "0010" => --SCAN
    opcode <= "1000000000000011";
    state <= state0;
  when "0011" => --INQUIRY
    opcode <= "1000000000110011";
    state <= state6;
  when "0100" => --CONNECT ACL
    opcode <= "1000000000000101";
    state <= state6;
  when "0101" => --ADD SCO
    opcode <= "1000000000000001";
    state <= state6;
  when "0110" => --ALERT

```

```

        opcode <= "1000000000000000";
        state <= state6;
when "0111" => --CLEAR
        opcode <= "1000000000000000";
        state <= state6;
when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state6;
end case;
when state6 =>
case ctrl is
when "0001" => --AA
        opcode <= "1000000000000000";
        state <= state7;
when "0011" => --INQUIRY
        opcode <= "1000000010001011";
        state <= state7;
when "0100" => --CONNECT ACL
        opcode <= "1000000001111000";
        state <= state7;
when "0101" => --ADD SCO
        opcode <= "1000000000000000";
        state <= state7;
when "0110" => --ALERT
        opcode <= "1000000000000001";
        state <= state7;
when "0111" => --CLEAR
        opcode <= "1000000000000001";
        state <= state7;
when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state7;
end case;
when state7 =>
case ctrl is
when "0001" => --AA
        opcode <= "1000000000000010";
        state <= state0;
when "0011" => --INQUIRY
        opcode <= "1000000010011110";
        state <= state8;
when "0100" => --CONNECT ACL
        opcode <= "1000000000010100";
        state <= state8;
when "0101" => --ADD SCO

```

```

        opcode <= "1000000011100000";
        state <= state8;
when "0110" => --ALERT
        opcode <= "1000000000000000";
        state <= state8;
when "0111" => --CLEAR
        opcode <= "1000000000000000";
        state <= state8;
when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state8;
end case;
when state8 =>
case ctrl is
when "0011" => --INQUIRY
        opcode <= "1000000000001000";
        state <= state9;
when "0100" => --CONNECT ACL
        opcode <= "1000000000110111";
        state <= state9;
when "0101" => --ADD SCO
        opcode <= "1000000000000000";
        state <= state0;
when "0110" => --ALERT
        opcode <= "1000000001000000";
        state <= state9;
when "0111" => --CLEAR
        opcode <= "1000000001000000";
        state <= state9;
when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state9;
end case;
when state9 =>
case ctrl is
when "0011" => --INQUIRY
        opcode <= "1000000000000001";
        state <= state0;
when "0100" => --CONNECT ACL
        opcode <= "1000000010000000";
        state <= state10;
when "0110" => --ALERT
        opcode <= "1000000000000000";
        state <= state10;
when "0111" => --CLEAR
        opcode <= "1000000000000000";
        state <= state10;

```

```

when others =>          --CONFIG
    opcode <= "1000000000000000";
    state <= state10;
end case;
when state10 =>
case ctrl is
when "0100" => --CONNECT ACL
    opcode <= "1000000000000000";
    state <= state11;
when "0110" => --ALERT
    opcode <= "1000000000110001";
    state <= state0;
when "0111" => --CLEAR
    opcode <= "1000000000110010";
    state <= state0;
when others =>          --CONFIG
    opcode <= "1000000000000000";
    state <= state11;
end case;
when state11 =>
case ctrl is
when "0100" => --CONNECT ACL
    opcode <= "1000000000011000"; --18
    state <= state12;
when others =>          --CONFIG
    opcode <= "1000000000000000";
    state <= state12;
end case;
when state12 =>
case ctrl is
when "0100" => --CONNECT ACL
    opcode <= "1000000000000000";
    state <= state13;
when others =>          --CONFIG
    opcode <= "1000000000000000";
    state <= state13;
end case;
when state13 =>
case ctrl is
when "0100" => --CONNECT ACL
    opcode <= "1000000000000001";
    state <= state14;
when others =>          --CONFIG
    opcode <= "1000000000000000";
    state <= state14;

```

```

    end case;
when state14 =>
case ctrl is
    when "0100" => --CONNECT ACL
        opcode <= "1000000000000000";
        state <= state15;
    when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state15;
    end case;
when state15 =>
case ctrl is
    when "0100" => --CONNECT ACL
        opcode <= "1000000000000000";
        state <= state16;
    when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state16;
    end case;
when state16 =>
case ctrl is
    when "0100" => --CONNECT ACL
        opcode <= "1000000000000000";
        state <= state17;
    when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state17;
    end case;
when state17 =>
case ctrl is
    when "0100" => --CONNECT ACL
        opcode <= "1000000000000000";
        state <= state0;
    when others =>          --CONFIG
        opcode <= "1000000000000000";
        state <= state0;
    end case;
when others =>
end case;
end if;
end if;
end process;
end architecture behavioral;

```