```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Definitions
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _DEFS_H_
#define _DEFS_H_

#include "bitdefs.h" // BIT0HI, BIT0LO....definitions
#include <htc.h>

/* Variables */
#define RFID_MESSAGE_LENGTH 3
#define SC_MESSAGE_LENGTH 3
unsigned char rfidData[RFID_MESSAGE_LENGTH]; //needs a #define
unsigned char securityCode[SC_MESSAGE_LENGTH];


/* Port A bits: */
#define DEBUG_LED_BIT 0
#define DEBUG_LED_DDR TRISA
#define DEBUG_LED_PORT PORTA

/* Port B bits: */
#define SDI_BIT 4
#define SDI_DDR TRISB
#define SDI_PORT PORTB

#define RX_BIT 5
#define RX_DDR TRISB
#define RX_PORT PORTB

#define SCK_BIT 6
#define SCK_DDR TRISB
#define SCK_PORT PORTB

#define TX_BIT 7
#define TX_DDR TRISB
#define TX_PORT PORTB

/*Port C bits*/
#define WATERLOG_BIT 0
#define WATERLOG_DDR TRISC
#define WATERLOG_PORT PORTC

#define DATA_READY_BIT 2
#define DATA_READY_DDR TRISC
```

```c
#define DATA_READY_PORT PORTC

#define SS_BIT 6
#define SS_DDR TRISC
#define SS_PORT PORTC

#define SDO_BIT 7
#define SDO_DDR TRISC
#define SDO_PORT PORTC


/* Start and End */
#define SC_START 0x04
#define  SC_END 0x08

#define RFID_START 0x02
#define  RFID_END 0x03


/* Convenience */
#define FALSE 0
#define TRUE 1

#define LOW 0
#define HIGH 1

#define LEFT 0
#define RIGHT 1

#define SETBIT(ADDRESS,BIT) (ADDRESS |= (1<<BIT)) //set bit (high)      //output
#define CLEARBIT(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT)) //clear bit (low)  //input
#define CHECKBIT(ADDRESS,BIT) (ADDRESS & (1<<BIT)) //test single bit in I/O location
#define TOGGLEBIT(ADDRESS,BIT) (ADDRESS ^= (1<<BIT)) //toggle single bit

#endif
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Events Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include <htc.h>
#include "defs.h"
#include "events.h"
#include "security.h"
#include "sync.h"

/* Functions */
/* Checks and returns any events that have occurred. */
Event_t EVENTS_CheckForEvents(void) {
	Event_t event = NO_EVENT;

	//Check if sync or async event
	event = SYNC_CheckEvents();
	if (event != NO_EVENT)
	    return event;

	//Check if waterlogged, sync or async event
	event = SECURITY_CheckEvents();
	if (event != NO_EVENT)
	    return event;

	return event;
}
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Events Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _EVENTS_H_
#define _EVENTS_H_

typedef enum {
	WATER_LOGGED,
	RECEIVED_SECURITY,
	RECEIVED_RFID,
	NO_EVENT
} Event_t;

Event_t EVENTS_CheckForEvents(void);

#endif
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Main Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include <htc.h>
#include "bitdefs.h" // BIT0HI, BIT0LO....definitions
#include "defs.h"
#include "events.h"
#include "security.h"
#include "sync.h"
#include "state.h"

#define STEP_TIME_HIGH_BYTE 0xFF //for timer 1 setup

__CONFIG(HS & WDTDIS & BORDIS); //only call once

/* Function Prototypes */
void MAIN_Init(void);
void MAIN_SetupTimer1(void);
static void MAIN_Debug(void);

/* Functions */
void main (void) {
        MAIN_Init();
    MAIN_SetupTimer1();
        SECURITY_Init();
        SYNC_Init();

        //after all inits, enable interrupts
        ei(); //enable global interrupts
        PEIE = 1;  //all peripheral interrupts are enabled
        SSPIE = 1; // enable interrupts from the SPI

    //Uncomment the following line to debug and not run the state machine.
    //MAIN_Debug();

    //Initialize the state machine and loop forever to run it.
        STATE_InitStateMachine();
        for(;;) {
            STATE_RunStateMachine(EVENTS_CheckForEvents());
        }
}

void interrupt int_res (void) {
        if(RCIF != 0) { //we've had a receive interrupt ASYNC
            SECURITY_ReadMessage();
```

```
            RCIF = 0; //clear the interrupt flag
        }

        //Test TXIE and TXIF, only then respond
        if(TXIF != 0) { //we've had a TX interrupt meaning the output buffer is empty and we can transmit
now
            //check to see if there actually is a message to send.
            if(SECURITY_IsTransferInProgress() == 0) {
                        //we should never be getting here, but if we do, this will help to keep the system running
                        TXIE = 0; //disable this interrupt
            } else {
                        SECURITY_SendNextMessageByte();
            }
        }
        if(SSPIF != 0) { //SPI serial buffer is full, we need to read it
            SYNC_ReadAndSendMessage();
        }
}

void MAIN_Init(void){
        //set inputs and clear outputs

        PORTA = 0x00; //clear all of port a
        PORTB = 0x00; //clear all of port b
        PORTC = 0x00; //clear all of port c

        //for debugging
        CLEARBIT(DEBUG_LED_DDR, DEBUG_LED_BIT); //output

        ANSEL = 0;  //NO ADC, all digital pins
        ANSELH = 0;  //NO ADC, all digital pins
}

void MAIN_SetupTimer1(void) {
        //Using timer 1 for the step control
        //Turn on timer 1
        //use internal instruction clock 20mhz/4 = 5mhz
        //use a /8 prescaler = .625Mhz
        T1CON = 0b10110101;         //setup the inital timer control bits
        di();

        //now set it up for output compare, internally, software, no external pin
        CCP1CON = 0b00001010; //set the CCP1CON compare reg to output compare, software
        CCPR1H  = STEP_TIME_HIGH_BYTE; //move the inital step time value into the upper byte of the
compare reg
        CCPR1L  = 0x00; //clear the lower byte of the compare time reg (we don't need the precision!)

        TMR1H = 0x00; //clear timer high byte
        TMR1L = 0x00; //clear timer low byte
        PIR1 = 0x00; //clear interrupt flags
        ei();
}

/* Will always loop as it runs debugging tests for each module. These debugging
 * tests are ONLY for debugging and have blocking code. */
static void MAIN_Debug(void) {
```

```
    //(void)printf("\nDebugging...\r\n");
    //(void)printf("\nNote that debugging is case sensitive.\r\n");
    for(;;) {
        SYNC_Debug();
        SECURITY_Debug();
    }
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Security Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include "security.h"
#include "events.h"

/* Variables */
/* TX */
unsigned char txMessage[TX_FRAME_LENGTH + RFID_MESSAGE_LENGTH];
unsigned char txMessageBuffer[TX_FRAME_LENGTH + RFID_MESSAGE_LENGTH];
unsigned char txMessageBufferIndex = 0;
unsigned char transferInProgressFlag = 0;
unsigned char messageLength = RFID_MESSAGE_LENGTH;
unsigned char rfidDataDummy[RFID_MESSAGE_LENGTH];

/* RX */
unsigned char rxByte = 0;
unsigned char rxMessageBuffer[RX_FRAME_LENGTH + SC_MESSAGE_LENGTH];
unsigned char rxMessageBufferIndex = 0;
unsigned char rxCompleteFlag = 1;
unsigned char entered = 0;
unsigned char receivedSecurityCode = 0;

unsigned char waterLoggedFlag = 0;
static Event_t lastEvent = NO_EVENT;


/* Prototypes */
static void SECURITY_InitPins(void);
static void SECURITY_SendByte(unsigned char);
static void SECURITY_FinishedSendingMessage(void);
static void blockingDelay(unsigned int delayTimeH);
static void SECURITY_ParseMessage(void);


/* Functions */
/* Initialize Security communications */
void SECURITY_Init(void) {
	SECURITY_InitPins();

	di();
	/*Baud rate setup*/
	SCKP = 0; //transmit non-inverted data
	ABDEN = 0; //autobaud functionality turned off
	WUE = 0; //normal reciver operation
```

```
	BRG16 = 0; //8 bit baud generator used
	SPBRG = 126; //Set USART speed  9.6 kbps, .16% error, FOSC 20MHz

	/*TRANSMITTER*/
	//TXSTA REGISTER
	TX9 = 0; //8 bit transmission
	SYNC = 0; //async mode
	BRGH = 1; //High Speed moded
	TRMT = 1; //Transmit shift reg
	TXEN = 1; //enable transmission
	TXIF = 0; //lower the transmit flag

	/*RECEIVER*/
	//RCSTA REGISTER
	RX9 = 0; //8 bit reception
	ADDEN = 0; //disable address detection wakup
	FERR = 0; //framing error flag lowered
	OERR = 0; //overrun error flag lowered
	SPEN = 1; //enable serial port pins
	CREN = 1; //Enable continious reception
	RCIF = 0; //clear the flag

	/*INTERRUPTS*/
	RCIE = 1;  //Enable RX interrupt
	ei();
}

static void SECURITY_InitPins(void) {
	//for waterlog
	SETBIT(WATERLOG_DDR, WATERLOG_BIT); //input

	//for asyncronous communications
	CLEARBIT(TX_DDR, TX_BIT); //output
	SETBIT(RX_DDR, RX_BIT); //input
}

/* TX */
void SECURITY_PrepareMessage(unsigned char dataPacket[]) {
	txMessage[0] = SC_START;
	txMessage[1] = dataPacket[0];
	txMessage[2] = dataPacket[1];
	txMessage[3] = dataPacket[2];
	txMessage[4] = SC_END;
}

/* Checks to see if a message is in the process of being transferred. */
unsigned char SECURITY_IsTransferInProgress(void) {
	return transferInProgressFlag;
}

/* Begins sending of message process by setting flag, resetting the transfer buffer, and enabling tx
interrupts. */
void SECURITY_SendMessage(void) {
	unsigned char i;
	transferInProgressFlag = 1; //raise the flag that indicates that we are sending a message
	//transfer USART message to TX message buffer
```

```
        for (i = 0; i < (sizeof(txMessage)); i++) {
            txMessageBuffer[i] = txMessage[i];
        }
        txMessageBufferIndex = 0;  //reset the index
        TXIE = 1; //enable TX interrupts
}

/* Transfers the next byte in the message. When end of message has been reached, will stop sending. */
void SECURITY_SendNextMessageByte(void) {
        //this will send the data byte in the message buffer that is being pointed to with the index value
        SECURITY_SendByte(txMessageBuffer[txMessageBufferIndex]);
        //increase the index count
        if(txMessageBufferIndex >= (sizeof(txMessageBuffer) - 1)) { //we are at the last byte of the message
            //we have finished sending the message, so cleanup
            SECURITY_FinishedSendingMessage();
        } else {
            txMessageBufferIndex++;
        }
}

/* Actually transfers byte to the transfer register. */
static void SECURITY_SendByte(unsigned char dataByteToSend) {
        //do some sort of MAC or line checking here...if we need to
        TXREG = dataByteToSend;
}

/* Called at the end of a message transfer to stop sending by disabling tx interrupts and clearing flag. */
static void SECURITY_FinishedSendingMessage(void) {
        TXIE = 0; //disable TX interrupts
        transferInProgressFlag = 0; //lower the flag that indicates that we are sending a message
}

/* RX */
/* Reads in message byte by byte on the receive register and stores them in the rxMessageBuffer array. */
void SECURITY_ReadMessage(void) {
        rxByte = RCREG;  //read in the data
        rxMessageBuffer[rxMessageBufferIndex] = rxByte;
        if(rxMessageBufferIndex == 0){
            //SETBIT(DATA_READY_PORT, DATA_READY_BIT); //data ready set when receiving data
        }
        rxMessageBufferIndex++; //increment the rxMessageBuffer index
        if(rxMessageBufferIndex > (sizeof(rxMessageBuffer) - 1)) {
            rxMessageBufferIndex = 0;
            //CLEARBIT(DATA_READY_PORT, DATA_READY_BIT); //data ready cleared when not
receiving data
            SECURITY_ParseMessage();
        }
}

static void SECURITY_ParseMessage(void) {
        SETBIT(DATA_READY_PORT, DATA_READY_BIT);
        receivedSecurityCode = 1;
        securityCode[0] = rxMessageBuffer[0];
        securityCode[1] = rxMessageBuffer[2];
        securityCode[2] = rxMessageBuffer[4];
}
```

```
Event_t SECURITY_CheckEvents(void) {
	Event_t event = NO_EVENT;

	if (CHECKBIT(WATERLOG_PORT, WATERLOG_BIT)) {
	    waterLoggedFlag = 1;
	    event = WATER_LOGGED;
	}

	if (receivedSecurityCode) {
	    receivedSecurityCode = 0; //reset flag
	    CLEARBIT(DATA_READY_PORT, DATA_READY_BIT);
	    event = RECEIVED_SECURITY;
	}

	if (event != lastEvent) {
	    lastEvent = event;
	    return event;
	}

	return NO_EVENT;
}

/* Blocking function to create a delay. Used only for debugging. */
static void blockingDelay(unsigned int delayTimeH) {
	unsigned char currentTimeH;
	unsigned char lastTimeH;
	unsigned char deltaTime;
	unsigned int timerTime = 0;

	lastTimeH = TMR1H;

	while (timerTime <= delayTimeH) { //loop until we've reached the current time
	    currentTimeH = TMR1H;
	    deltaTime = (currentTimeH - lastTimeH);
	    timerTime = timerTime + deltaTime;
	    lastTimeH = currentTimeH;
	}
}

/* Debugging */
void SECURITY_Debug(void) {
	/*if (entered == 0) {
	    entered = 1;
	    rfidDataDummy[0] = 0x84;
	    rfidDataDummy[1] = 0x08;
	    rfidDataDummy[2] = 0x5E;
	}*/

	//for(;;) {
	    //SECURITY_PrepareMessage(rfidData);
	    //blockingDelay(500);
	//  if(SECURITY_IsTransferInProgress() == 0) { //check to see if we are already sending a message
	//		SECURITY_SendMessage();
	//  }
	//}
```

```
}
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Security Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _SECURITY_H_
#define _SECURITY_H_

#include <htc.h>
#include "defs.h"
#include "events.h"

#define TX_FRAME_LENGTH 2
#define RX_FRAME_LENGTH 3

/* Functions */
void SECURITY_Init(void);
void SECURITY_PrepareMessage(unsigned char[]);
unsigned char SECURITY_IsTransferInProgress(void);
void SECURITY_SendMessage(void);
void SECURITY_SendNextMessageByte(void);
void SECURITY_ReadMessage(void);
Event_t SECURITY_CheckEvents(void);
void SECURITY_Debug(void);

#endif
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * State Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include "defs.h"
#include "state.h"
#include "sync.h"
#include "security.h"

/* Variables */
static State_t state;


/* State machines */
/* Prototypes */
static void WaitingState(Event_t event);

/* Initialize state machine by setting initial state and timer. */
void STATE_InitStateMachine(void) {
	SYNC_DummySDOSecurityCodeArray();
	state = WAITING;
}

/* Figure out which state we are in, and call the corresponding function to handle the event. */
void STATE_RunStateMachine(Event_t event) {
	switch (state) {
	    case WAITING:
			WaitingState(event);
			break;
	    default:
			break;
	}
}

/* State functions */
static void WaitingState(Event_t event) {
	switch (event) {
	    case RECEIVED_RFID:
			//read in interrupt
			//and send
			SECURITY_PrepareMessage(rfidData);
			if(SECURITY_IsTransferInProgress() == 0) { //check to see if we are already sending a
message
				SECURITY_SendMessage();
			}
			break;
```

```
		case RECEIVED_SECURITY:
			SYNC_PrepareMessage();
			SETBIT(DATA_READY_PORT, DATA_READY_BIT); //make sure it is ON
			break;
		case WATER_LOGGED:
			break;
		default:
			break;
	}
}
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * State Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _STATE_H_
#define _STATE_H_

#include <htc.h>
#include "defs.h"
#include "events.h"

typedef enum {
	WAITING,
}State_t;

void STATE_InitStateMachine(void);
void STATE_RunStateMachine(Event_t event);

#endif
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Sync Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include "sync.h"

#define SIZE_OF_SYNC_SDO_ARRAY 5
#define SIZE_OF_SYNC_SDI_ARRAY 5


/* Variables */
/* SDO */
//Send out the Security Code Data (which is 6 bytes)
unsigned char SYNC_SDO_SecurityCode = 0;
unsigned char SYNC_SDO_SecurityCodeArray2[SIZE_OF_SYNC_SDO_ARRAY];
unsigned char SYNC_SDO_SecurityCodeArrayIndex = 1;
unsigned char SYNC_SendingByteFlag = 0;
unsigned char SYNC_SDO_FirstMessageOut = 0;
unsigned char SYNC_SDO_SecurityCodeArray[SIZE_OF_SYNC_SDO_ARRAY];

/* SDI */
//Receive the RFID Data to the Security Control PIC (5 bytes + 1 dummy)
unsigned char SYNC_ReceivedSDI = 0;
unsigned char SYNC_RFIDIn = 0;
unsigned char SYNC_SDI_RFIDDataArray[SIZE_OF_SYNC_SDI_ARRAY];
unsigned char SYNC_SDI_RFIDDataArrayIndex = 0;

unsigned char receivedRFID = 0;
static Event_t lastEvent = NO_EVENT;
unsigned char readingMessage = 0;


/* Prototypes */
static void SYNC_InitPins(void);
static void SYNC_ClearSDOSecurityCodeArray(void);
static void SYNC_ClearSDIRFIDDataArray(void);
static void SYNC_InitSSPAsSlave(void);
static void SYNC_SendSSPMessage(unsigned char Message);
static void SYNC_ParseMessage(void);


/* Functions */
void SYNC_Init(void) {
	SYNC_InitPins();
	SYNC_InitSSPAsSlave();
	SYNC_ClearSDOSecurityCodeArray();
```

```
        SYNC_ClearSDIRFIDDataArray();
        SSPBUF = 0xC1;
}

static void SYNC_InitPins(void) {
        //for synchronous communications
        SETBIT(SDI_DDR, SDI_BIT);                 //SDI must have TRIS bit set
        CLEARBIT(SDO_DDR, SDO_BIT);               //SDO must have TRIS bit cleared
        SETBIT(SCK_DDR, SCK_BIT);                 //SCK(Slave Mode) must have the TRIS bit set

        SETBIT(SS_DDR, SS_BIT);                   //SS must have tris bit set (input) (for slave) can use
as general purpose output
        CLEARBIT(DATA_READY_DDR, DATA_READY_BIT); //output
        CLEARBIT(DATA_READY_PORT, DATA_READY_BIT); //make sure it is off
}

static void SYNC_ClearSDOSecurityCodeArray(void) {
        SYNC_SDO_SecurityCodeArray[0] = 0x00;//RFID_START;
        SYNC_SDO_SecurityCodeArray[1] = 0x00;//0xA6;
        SYNC_SDO_SecurityCodeArray[2] = 0x00;//0xA7;
        SYNC_SDO_SecurityCodeArray[3] = 0x00;//0xA8;
        SYNC_SDO_SecurityCodeArray[4] = 0x00;//RFID_END;
}

static void SYNC_ClearSDIRFIDDataArray(void) {
        SYNC_SDI_RFIDDataArray[0] = 0x00;
        SYNC_SDI_RFIDDataArray[1] = 0x00;
        SYNC_SDI_RFIDDataArray[2] = 0x00;
        SYNC_SDI_RFIDDataArray[3] = 0x00;
        SYNC_SDI_RFIDDataArray[4] = 0x00;
}

void SYNC_DummySDOSecurityCodeArray(void) {
        SYNC_SDO_SecurityCodeArray[0] = 0xC1;//RFID_START;
        SYNC_SDO_SecurityCodeArray[1] = 0xC2;//0xA6;
        SYNC_SDO_SecurityCodeArray[2] = 0xC3;//0xA7;
        SYNC_SDO_SecurityCodeArray[3] = 0xC4;//0xA8;
        SYNC_SDO_SecurityCodeArray[4] = 0xC5;//RFID_END;
}

/* Initalizes the Serial Data functionality. */
static void SYNC_InitSSPAsSlave(void) {
        SSPSTAT = 0x00; //Clear the Sync Serial Port Status Register, SMP = 0, CKE = 0
        //Enable Interrupts if we want, do that here
        //setup SPI port
        SSPCON = 0b00010100; //(5)Don't Enable SSP Yet, (4) Clock Polarity Select (idle high)
        // Clock SPI Slave mode, clock = Fosc/16
        SETBIT(SSPCON, 5); //enable SSP
}

static void SYNC_SendSSPMessage(unsigned char Message) {
        SSPBUF = Message; //this should start the transmission
        //for debugging
        if (SSPBUF == 0xC1) {
            SYNC_SDO_FirstMessageOut = 0;
        }
```

```
}

void SYNC_ReadAndSendMessage(void) {
	SYNC_RFIDIn = SSPBUF;		//Read in message that we got when we transmitted

	if(SYNC_RFIDIn == RFID_START) {
	    SYNC_SDI_RFIDDataArrayIndex = 0;
	} else {
	    SYNC_SDI_RFIDDataArrayIndex++; //increase the rx buffer index
	    if(SYNC_SDI_RFIDDataArrayIndex > (SIZE_OF_SYNC_SDI_ARRAY - 1)) {  //have we gone
to the end?
			SYNC_SDI_RFIDDataArrayIndex = 0;
	    }
	}
	SYNC_SDI_RFIDDataArray[SYNC_SDI_RFIDDataArrayIndex] = SYNC_RFIDIn;
	//Store the message in the RxArray

	//Send out to Boat to get back the Security Code
	SYNC_SendSSPMessage(SYNC_SDO_SecurityCodeArray[SYNC_SDO_SecurityCodeArrayIndex]
);
	SYNC_SDO_SecurityCodeArrayIndex++;

	if(SYNC_SDO_SecurityCodeArrayIndex > 1) {
	    CLEARBIT(DATA_READY_PORT, DATA_READY_BIT);  //lower the trigger port/bit
	}
	if(SYNC_RFIDIn == RFID_END) {
	    SYNC_ParseMessage();
	}
	if(SYNC_SDO_SecurityCodeArrayIndex > (SIZE_OF_SYNC_SDO_ARRAY - 1)) {  //have we
gone to the end?
	    SYNC_SDO_SecurityCodeArrayIndex = 0;
	}

	if(WCOL != 0){ //has there been a buffer write collision?
	    //indicate to us that this has happened
	    WCOL = 0;
	}
	SSPIF = 0; //Clear the SSP flag
}

void SYNC_PrepareMessage(void) {
	unsigned char i;
	SYNC_SDO_SecurityCodeArray[0] = SC_START;
	for (i = 0; i < (SIZE_OF_SYNC_SDO_ARRAY - 1); i++) {
	    SYNC_SDO_SecurityCodeArray[i + 1] = securityCode[i];
	}
	SYNC_SDO_SecurityCodeArray[4] = SC_END;
}

static void SYNC_ParseMessage(void) {
	unsigned char i;
	receivedRFID = 1;
	for (i = 0; i < (SIZE_OF_SYNC_SDI_ARRAY - 1); i++) {
	    rfidData[i] = SYNC_SDI_RFIDDataArray[i + 1];
	}
}
```

```
Event_t SYNC_CheckEvents(void) {
	Event_t event = NO_EVENT;

	if (receivedRFID) {
	    receivedRFID = 0; //reset flag
	    event = RECEIVED_RFID;
	}

	if (event != lastEvent) {
	    lastEvent = event;
	    return event;
	}

	return NO_EVENT;
}

void SYNC_Debug(void) {
	//blank for now since this is the slave
/*	rfidData[0] = 0x84;
	rfidData[1] = 0x1E;
	rfidData[2] = 0xFE;*/

	SYNC_DummySDOSecurityCodeArray();
}
```

```c
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Security PIC
 * Sync Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _SYNC_H_
#define _SYNC_H_

#include <htc.h>
#include "defs.h"
#include "events.h"


/* Functions */
void SYNC_Init(void);
void SYNC_ReadAndSendMessage(void);
void SYNC_PrepareMessage(void);
void SYNC_DummySDOSecurityCodeArray();
Event_t SYNC_CheckEvents(void);
void SYNC_Debug(void);

#endif
```