```
// i2c protocol sniffer
// written by Bill Grundmann, 2009
//
// this program only traces messages upto the limit
// of available memmory.  Once memory is full, all of the
// messages are output through the serial port

// default: SCL and SDA are both high
// high to low on SDA while SCL is high -> Start
// Low to high on SDA while SCL is high -> Stop
// low to high on SCL -> data bit
// high to low on SCL -> data changing
// all data is 8 bits starting with MSB
// acknowledge is 9th bit SDA = 0 (ack), SDA = 1 (noack) (driven by slave unit)
// for 400khz
// SCL low > 1.3 us, SCL high 0.6us, SDA setup before SCL rise > 100ns
//
#include <WProgram.h>

#define cbi(t, b) t &= ~_BV(b)

#define SCL_PIN 2  // digital pin #2
#define SDA_PIN 3  // digital pin #3
#define LED_PIN 5  // digital pin #13 (portb)

#define SCL (unsigned char) (PIND & _BV(SCL_PIN))
#define SDA (unsigned char) (PIND & _BV(SDA_PIN))
#define SDA_BIT (unsigned char) ((PIND >> SDA_PIN) & 1)
#define SCL_BIT (unsigned char) ((PIND >> SCL_PIN) & 1)

#define LED_ON() PORTB |= _BV(LED_PIN)
#define LED_OFF() PORTB &= ~_BV(LED_PIN)
#define SET_LED(v) v?LED_ON(): LED_OFF()
#define LED_T() PINB |= _BV(LED_PIN);

unsigned char *queue;          // the queue storage
unsigned char *queue_ptr;      // next location to write in queue
unsigned char *message_start;  // position containing message length for current message
unsigned char *queue_max;      // the location of the end of the queue + 1
unsigned char *ack_ptr;            // pointer to acknowledge bytes (packed bits)
unsigned char ack_bit;         // counter for keeping track of all bits
unsigned char ack_byte;

//*************************************************************************************↙
  ***
// determine largest area available

uint16_t availableMemory() {
  uint16_t size = 1024;  // start with the most
  unsigned char *buf;
  while((buf = (unsigned char *) malloc(--size)) == NULL) ;
  free(buf);
  return size;
}

//*************************************************************************************↙
  ***
// acknowledge bits are accumulated into a byte to save memory
// so, there are 8 data bytes before we have 1 acknowledge byte

inline void writeAck(unsigned char ack) {
  ack_byte = (ack_byte << 1) | ack;
  ack_bit++;
  if (ack_bit == 8) {  // byte is full
    *ack_ptr++ = ack_byte;
    ack_bit = 0;
  }
```

```
}

//*************************************************************************************↙
  ***
// write a data byte to the queue and add another acknowledge bit

inline void writeQueue(unsigned char data, unsigned char ack) {
  *queue_ptr++ = data;  // store the data
  writeAck(ack);
}

//*************************************************************************************↙
  ***
// a message is stored as a length byte followed by the data bytes
// this routine fills in the message length byte after all of the
// data bytes are written

inline void writeMessageSize() {
  // write the message size to the first byte of the message
  unsigned char size = queue_ptr - message_start - 1;
  *message_start = size;
  message_start = queue_ptr;
  queue_ptr++;  // reserve first location for the message size
}

//*************************************************************************************↙
  ***
// this is called to fixup the acknowledge byte due to the fact
// that we might not have done 8 bits yet.

void setupReadAck() {
  // adjust last ack byte and put into memory
  while(ack_bit != 8) {
    ack_byte = (ack_byte << 1);
    ack_bit++;
  }
  *ack_ptr = ack_byte;  // save last ack byte information
  ack_ptr = queue_max;  // restart from beginning
  ack_byte = *ack_ptr++;  // get first ack byte of information
  ack_bit = 0;
}

//*************************************************************************************↙
  ***
void setup() {
  // turn off timer0 interrupts
  cbi(TIMSK0, TOIE0);

  // enable debug LED output on portb<5>
  pinMode(13, OUTPUT);

  // allocate memory queue area
  // determine total size, but leave some room for the stack
  uint16_t pool_size = availableMemory() - 100;

  // the data queue is 8 data bytes per 1 acknowledge byte
  long queue_size = (long) pool_size * 1000 / 1125;

  // the data queue starts at the beggining of the memory pool
  queue = (unsigned char *) malloc(pool_size);

  // have the data queue pointer point to the next location
  queue_ptr = queue + 1;

  // pointer to the message byte location
  message_start = queue;
```

```
  // compute the limit of the data area, but is also the
  // beginning of the acknowledge bytes area
  queue_max = queue + queue_size;
  ack_ptr = queue_max;
  ack_bit = 0;

  Serial.begin(115200);

  // report various sizes
  Serial.print("Message memory size: ");
  Serial.println(pool_size);
  Serial.print("queue size: ");
  Serial.println(queue_size);
}

//*************************************************************************************↙
  ***
// cheaper ways of printing hexidecimal bytes
void printnibble(unsigned char d) {
  if (d < 10) Serial.print(d + '0', BYTE);
  else Serial.print(d - 10 + 'A', BYTE);
}
void printhex(unsigned char d) {
  printnibble(d >> 4);
  printnibble(d & 0xf);
}

//*************************************************************************************↙
  ***
void loop() {

  unsigned char data;     // byte to accumulate data information
  unsigned char bits;     // current bit being processed

  while(1) {
    // wait for both high
    while(SCL == 0 || SDA == 0);

    // wait for start condition
    while(SDA || SCL == 0);  // SDA falls and SCL is still high
repeated_start:

    // wait for SCL to fall
    while(SCL);
    bits = 0;

sample_bit:
    // now ready for next bit of the byte
    while(SCL == 0);  // look for rising

    // sample data on SDA
    // we might not use this data in the cases of either
    // stop or repeated start conditions
    unsigned char dt = SDA;

    // SCL is now high
    // wait for either of two conditions: SCL falls or SDA changes
    // split the cases to reduce the sub-loop overheads
    if (dt == 0) {
      // special case where stop condition might exist
      // because data was zero
look_for_bit_or_stop:
      if (SCL == 0) goto process_bit;
      if (SDA == 0) goto look_for_bit_or_stop;
      goto stop_condition;

    } else {
```

```
      // special case where we might have a repeated start
      // where SDA fails while SCL high
look_for_bit_or_repeated_start:
      if (SCL == 0) goto process_bit;
      if (SDA) goto look_for_bit_or_repeated_start;
      goto repeated_start;
    }

process_bit:
    // process another data bit
    dt = dt >> SDA_PIN;  // normalize the bit to the LSB position

    if (bits == 8) {
      // looking at the acknowledge bit
      if (queue_ptr == queue_max) goto queue_is_full;
      writeQueue(data, dt);
      bits = 0;  // next SCL rise starts a new byte of data
    }
    else {
      // accumulate bits of the byte's information (MSB to LSB)
      data = (data << 1) | dt;
      bits++;
    }
    goto sample_bit;

stop_condition:
    if (queue_ptr == queue_max) goto queue_is_full;
    writeMessageSize();
  }

  // the program comes here when the memory is full
queue_is_full:
  setupReadAck();  // clean up the acknowledge byte

  // dump all of the messages
  for (queue_ptr = queue; queue_ptr < queue_max;) {
    // start of a new message .. 1st byte is size of message
    uint16_t len = *queue_ptr++;  // length of message

    Serial.print("(");
    Serial.print(len);
    Serial.print("):");

    unsigned char *first = queue_ptr;  // remember the first location
    unsigned char len_start = len;
    unsigned char address_byte = *queue_ptr;

    // print the data for each message byte
    for (;len;len--) {
      unsigned char dt = *queue_ptr++;
      printhex(dt);

      // look at the acknowledge information for this byte
      if (ack_byte & 0x80) Serial.print("-");  // not acknowledged
      else Serial.print("+");  // acknowledged
      ack_bit++;
      ack_byte = ack_byte << 1;  // next bit
      if (ack_bit == 8) {
        ack_bit = 0;
        ack_byte = *ack_ptr++;  // next byte
      }
      Serial.print(" ");
    }
    Serial.print("\n\r");
    if (address_byte == 0xa5) {
      // repeat a response with data that needs to be decoded
      queue_ptr = first + 1;  // start over - skip the address byte
```

```
      Serial.print("        ");
      len_start--;
      for (;len_start;len_start--) {
        unsigned char dt = (*queue_ptr++ ^ 0x17) + 0x17;
        printhex(dt);
        Serial.print("  ");
      }
    }
    Serial.print("\n\r");
  }
  while(1);  // don't do anymore
}
```