your fish scale if you have a big motor.

That is the math you would use to choose your motors. This will get you close to what you need for success. Gordon McComb (of *Robot Builders Bonanza* fame) shared his secret for success: the heft method. He would look at his robot frame, pick up a motor, and hold it to gauge its *heft* to determine its suitability. I myself do not have Gordon's calibrated arm, so I tend to do a little math.

Q. I want to store way-points and sensor data during a robot's run-time. This really adds up and is way more than most EEPROMs I've seen can do. I've been trying to get an SD card to work using an SPI interface, but it just isn't working. I can't initialize the card. How does this work?

— Thomas Q.
Boise, ID

A. I have been fascinated with SD cards for a variety of reasons, and this gives me a great reason to work with them. I am currently having fun with Microchip PIC24 devices and so used one of their demo boards with an SD PICtail card for experimenting. The target hardware isn't all that important, so my code should work with anyone's microcontroller; just change how you deal with the SPI hardware to get your required clock rates. An SD card can take SPI clocks of 20 to 50 MHz, so there is no issue about going too fast. As it happens, my 32 MHz PIC24FJ64 part can only go 8 MHz, but that gave me pretty good transfer rates (compared to serial ports, anyway) and I was happy. The secret to success is to start out at a clock rate of 400 kHz until the card is up and listening, then move to the high speed clock. To fully understand the interface — which is pretty simple — and the protocol — which is slightly less simple, check out this site and get the SD Association *Simplified* Specs: **www.sdcard.org/dev.../pls/simplified_specs**.

This site took me a long way towards getting my SD cards to work: **http://elm-chan.org/docs/mmc/mmc_e.html**.

There is a raft of information out on the net for handling the reading and writing of data blocks to the SD card, so that part was easy. The big thing to know is that the SD card wants the block address in bytes, on 512 byte boundaries; the SDHC card has a much higher capacity and wants its block addressing done where each block is 512 bytes. My initialization code shown here takes this into account by storing the attributes of the SD or SDHC card being used so that the read/write routines know which type of block addressing to use. Many thanks to the pioneers of SD card usage that helped me with this work!

I have given my various routines here, but did not bother with the actual block read and writes. I'll leave that as an exercise for the reader. If anyone is interested, you can send me an email (see end of article) and I'll do a more detailed write-up.

There are a lot of constants in **Listing 1**. They refer to various values for commands, which are pretty obvious when you look at the specifications for SD commands. I

Listing 1: SD Card Initialization.

```
uint8_t SD_Write(uint8_t b)
 /**
 *  Read and write a single 8-bit word to
 *  the SD/MMC card. Using standard, non-buffered
 *  mode in 8 bit words.
 *  **Always check SPI1RBF bit before reading
 *  the SPI2BUF register
 *  **SPI1BUF is read and/or written to
 *  receive/send data
 *
 *  PRECONDITION: SPI bus configured, SD card
 *  selected and ready.
 *  INPUTS: b = byte to transmit (or dummy
 *  byte if only a read done)
 *  OUTPUTS: none
 *  RETURNS:
 */
 {
  SPI1BUF = b;
         // write to buffer for TX
  while( !SPI1STATbits.SPIRBF);
         // wait for transfer to complete
  SPI2STATbits.SPIROV = 0;
         // clear any overflow.

  return SPI1BUF;
         // read the received value
 }

 // Not worth code defining these since
 // they are all the same.
 #define SD_Read()    SD_Write( 0xFF)
 #define SD_Clock()    SD_Write( 0xFF)
 #define SD_Disable() nMEM_CS = 1; SD_Clock()
 #define SD_Enable()  nMEM_CS = 0

 uint8_t SD_SendCmd(uint8_t cmd, LBA addr)
 /**
 * Send an SPI mode command to the SD card.
 *
 * PRECONDITION: SD card powered up, CRC7
 * table initialized.
 * INPUTS: cmd = SPI mode command to send
 *          addr= 32bit address
 * OUTPUTS: none
 * RETURNS: status read back from SD card (0xFF
 * is fault)
 * *** NOTE nMEM_CS is still low when this
 * function exits.
 *
 *  expected return responses:
 * FF - timeout
 * 00 - command accepted
 * 01 - command received, card in idle state
 * after RESET
 *
 *  R1 response codes:
 * bit 0 = Idle state
 * bit 1 = Erase Reset
 * bit 2 = Illegal command
 * bit 3 = Communication CRC error
 * bit 4 = Erase sequence error
 * bit 5 = Address error
 * bit 6 = Parameter error
 * bit 7 = Always 0
 */
 {
   uint16_t     n;
   uint8_t        res;
   uint8_t        byte;
   uint8_t        CRC7 = 0x95;
         // Generic CRC7 byte

   SD_Enable();
         // enable SD card
```

```
  byte = cmd | 0x40;
  SD_Write(byte);
        // send command packet (6 bytes)
byte = addr>>24;
  SD_Write(byte);
        // msb of the address
  byte = addr>>16;
  SD_Write(byte);
  byte = addr>>8;
  SD_Write(byte);
  SD_Write( addr);
        // lsb

  SD_Write(CRC7);
        // Not used unless CRC mode on

  // now wait for a response (allow for up
  // to 8 bytes delay)
  n = 9;
  do {
    res = SD_Read();
        // check if ready
    if ( res != 0xFF)
        break;
  } while ( —n > 0);

  return (res);
        // return the result
}

void SD_InitSPI( void)
/**
*   Configure the SD card SPI bus hardware
*   settings and software. *
*** Using the SD SPI mode spec settings
*   instead of the MCHP example.
*
*   PRECONDITION: none
*   INPUTS: none - The hardware is explicitly
*   set up
*   OUTPUTS: none
*   RETURNS: none.
*/
{
  nMEM_CS = 1;
        // De-select the SD card

  if (sdcard.cardInit == 1) {
      return;
  }
    // init spi module for a slow (init) clock
    // speed, 8 bit byte mode
    // Master, CKE=0; CKP=1, sample end,
    // prescale 1:64 (250KHz)
  SPI1STATbits.SPIEN = 0;
        // disable SPI for configuration
    SPI1CON1 = 0x027c;
    SPI1CON2 = 0x0000;
        // No buffer, no frame mode
    SPI1STAT = 0x8000;
        // enable
}

uint8_t SD_InitMedia( void)
/**
*  Discover the type and version of the
*  installed SD card.  This routine will find
*  any SD or SDHC card and properly set it up.
*
*  PRECONDITION: none
*  INPUTS: none
*  OUTPUTS: none
*  RETURNS: 0 if successful, some other
*  error if not.
*/
{
  uint16_t     n;
  uint8_t     res = 0;
  // // If we get that far...
  uint32_t    timer;
  uint8_t        cmd;
  uint8_t        db[16];
    // when we get data back to look at

  if (sdcard.cardInit == 1) {
      return(0);
    // done, don't do it again.
  }

  // 1. start with the card not selected
  SD_Disable();
  // 2. send 80 clock cycles so card can
  // init registers
  for ( n=0; n<10; n++)
      SD_Clock();
  // 3. now select the card
  SD_Enable();

  // 4. send a reset command and look for
  // "IDLE"
  res = SD_SendCmd( RESET, 0); SD_Disable();
  if ( res != 1) {
     SD_Disable();
     return(LOG_FAIL);
        // card did not respond with "idle"
    }

  // 5. Check card voltage (type) for SD 1.0
  // or SD 2.0
  res = SD_SendCmd(SEND_IF_COND, 0x000001AA);
  // didn't respond or responded with an
  // "illegal cmd"
  if ( (res == 0xFF) || (res == 0x05)) {
      sdcard.cardVer = 1;
        // means it's an SD 1.0 or MMC card

    // 6. send INIT until receive a 0 or
   // 300ms passes
    timer = t_1ms + 300;
     while(timer > t_1ms) {
        res = SD_SendCmd(INIT,0);
        SD_Disable();
        // SendSDCmd() enables SD card
        if (!res) {
            break;
        // The card is ready
        }
    }
    if (res != 0) {
        return(LOG_FAIL);
        // failed to reset.
    }
    SD_Disable();
        // remember to disable the card
 }
else {  // need to pick up 4 bytes for v2
    card voltage description
        sdcard.cardVer = 2;
        // SD version 2.0 card
        for (n=0; n<4; n++) {
            db[n] = SD_Read();
        }
        // but we'll ignore these bytes
        SD_Disable();

       // 6. send INIT or SEND_APP_OP
       // repeatedly until receive a 0
       cmd = SEND_APP_OP;
       timer = t_1ms + 300;
        // wait up to .3 seconds for life
        // will still be in idle mode (0x01)
        // after this
       res = SD_SendCmd(APP_CMD, 0);
        SD_Disable();
       while (timer > t_1ms) {
        res = SD_SendCmd(cmd, 0x40000000);
       SD_Disable();
       // ACMD41 not recognized, use CMD1
```

Cont.

```
        if ( (res &0x0F) ==
       0x05 ) {
              cmd = INIT;
        }
        else {
            cmd = SEND_APP_OP;
        }
       if (!res) {
            break;
        }
     }
     if (res != 0) {
         return(LOG_FAIL);
        // failed to reset.
     }

     // 7. Check for
     // capacity of the card
     res = SD_SendCmd
    (READ_OCR,0);
     if (res != 0) {
       return(LOG_FAIL);
        // error, bad thing.
     }
     for (n=0; n<4; n++) {
       db[n] = SD_Read();
     }
     SD_Disable();
    // check CCS bit (bit 30),
    // PoweredUp (bit 31) set
    // if ready.
     if ( ((db[0] & 0x40)
    == 0x40) && (db[0] !=
    0xFF) ) {
         sdcard.cardCap = 1;
      // card is high capacity
      }
      else{
         sdcard.cardCap = 0;
      // card is low capacity
      }
}

sdcard.cardInit = 1;
   // successfully initialized

// Get the CSD register to
// find the size of the card
res = SD_SendCmd(SEND_CSD,0);
if (res != 0) {
    return(LOG_FAIL);
}
timer = t_1ms + 300;
       // wait for a response
while(timer > t_1ms) {
    res = SD_Read();
    if (res == DATA_START) {
        break;
  }
}
if (res == DATA_START) {
    // no timeout, read data
     for (n=0; n< 16; n++) {
         db[n] = SD_Read();
    // read the received value
     }
     // ignore CRC (for now)
        SD_Read();
        SD_Read();
        SD_Disable();
    }
    else {
        return(LOG_FAIL);
    }
    if (sdcard.cardCap == 1) {
         // Uses SDHC capacity
         // calculation
         sdcard.cardSize =
         db[9] + 1;
         sdcard.cardSize +=
        (uint32_t)(db[8] << 8);
         sdcard.cardSize +=
        (uint32_t)(db[7] &
         0x0F)<<12;
         sdcard.cardSize *=
         524288;
         // multiply by 512KB
         // (C_SIZE + 1)
         // * 512 * 1024
         sdcard.cardNumBlocks =
         sdcard.cardSize/sdcard
         .cardBlock;
    }
    else {
         // Uses SD capacity
         // calculation
      sdcard.cardSize =
      (uint16_t)((db[6] &
      0x03)<<10) | (uint16_t)
      (db[7]<<2) | (uint16_t)
      ((db[8] & 0xC0)>>6)) + 1;
       sdcard.cardSize =
      sdcard.cardSize <<((
      (uint16_t)((db[9] &
      0x03)<<1) | (uint16_t)
      ((db[10] & 0x80)>>7)) +2);
       sdcard.cardSize =
      sdcard.cardSize <<(
      (uint16_t)(db[5] & 0x0F));
      // (C_SIZE +1) <<(C_SIZE_
      // MULT + 2) <<(READ_BL_LEN)
      sdcard.cardNumBlocks =
      sdcard.cardSize/
      sdcard.cardBlock;
// Set block size to 512 bytes
      res = SD_SendCmd(
       SET_WBLEN,0x00000200);
      SD_Disable();
    }

    // Now kick to full speed
    // 8MHz mode. disable SPI for
    // configuration
    SPI1STATbits.SPIEN = 0;
    // Master, CKE=0; CKP=1,
    // sample end, prescale 1:2
    // (8MHz) all works
    SPI1CON1 = 0x027b;
    // re-enable SPI after
    // configuration
    SPI1STATbits.SPIEN = 1;
    return(res);
}
```

don't like "magic numbers" in my code, so I use defined constants. This procedure will identify both SD and SDHC cards. I have not tested it with SDXC cards, however, since I don't have any of them.

Whew! Another Mr. Roboto has come to an end. There were many obstacles that got in my way to get here, but "where there is a will ..." If you have any questions for Mr. Roboto, please send them to roboto@servomagazine.com. I love to hear from you and will do my best to answer your questions. Until then, keep on working on those robots! **SV**