Beat707.com

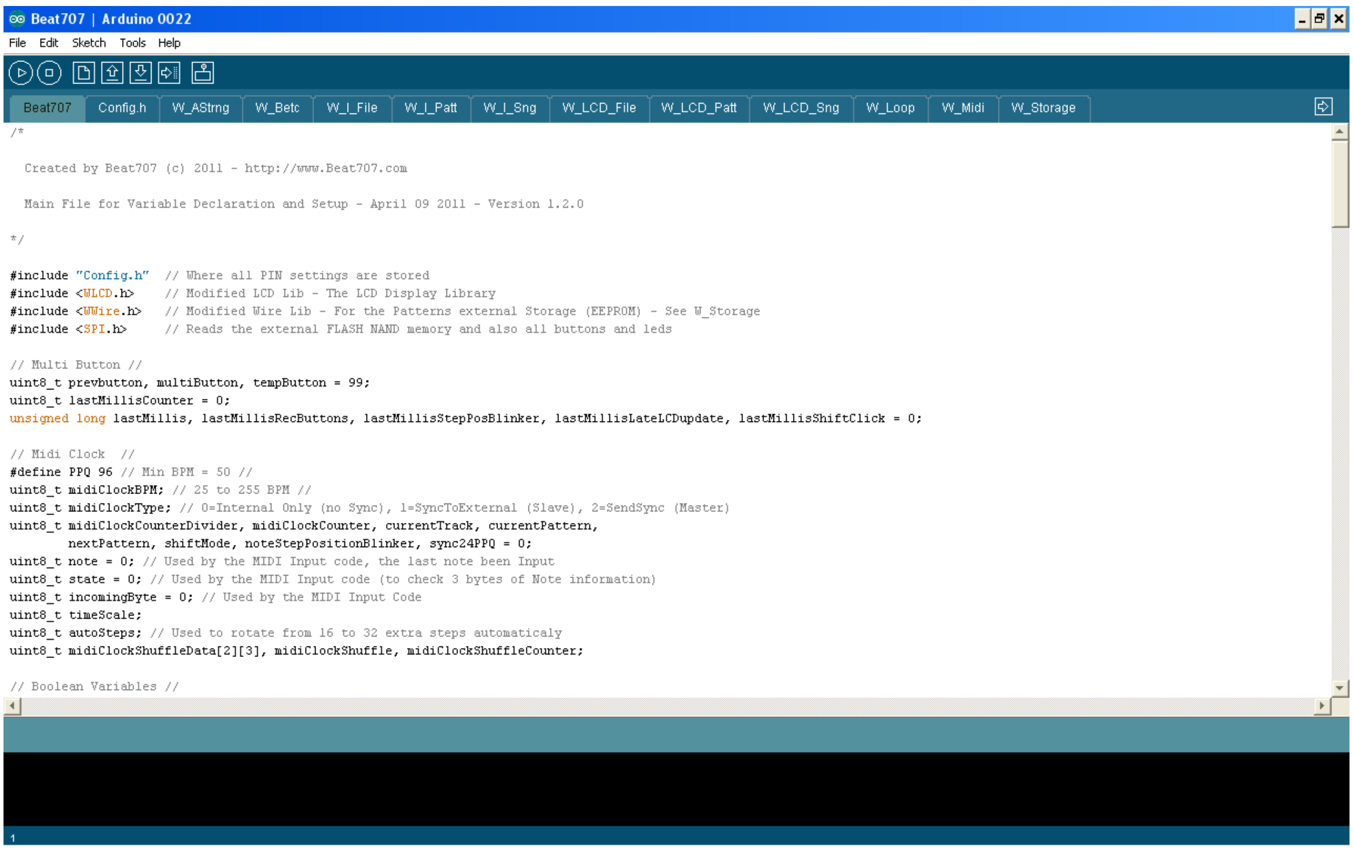
Drum Machine Software

(V1.2.0)

Welcome to Beat707 Drum Machine Software

Here we will talk about the Drum Machine & Groove Box software that comes with Beat707. The first step is to be sure you have the latest Arduino IDE and have updated the files according to the Beat707 DM Manual Instructions.

Below you can see the Arduino IDE with the Beat707 Software open. Notice the Tabs showing all the files used by the software. We have divided the files into names that helps understanding what each one does.



Below is the list of events the software will follow when you power up the device.

1. Execute setup() which is located in the very first tab named Beat707. This will use the Config.h settings, which has special configurations plus all pin information for the Arduino x Beat707 Hardware. This doesn't need changing, unless you created your own PCB from scratch and used a different pin structure. The following list explains a bit more about each option in the Config.h file. Keep in mind that 0 = Off and 1 = On.

   1.1 MAXSONGPOS & MAXSPATTERNS - can't be changed unless you know what you are doing, as this will change the whole way the system stores data into the EEPROM and Flash chips. So leave this alone, unless you already know how Storage works. The same goes for MAXSONGSFILE, which was calculated according to how much space each song takes. At the top of the W_Storage file (the last tab in the picture above) there's a short explanation on how to calculate the size of a song, and what are the special steps that one needs to take.

1.2 MIDIECHO - will add a special code that checks any incoming Midi data and echoes to the Midi Output channel.

1.3 CHECK_FOR_USB_MODE - This is a new option we added for V1.2.0 - Upon initiation of the system, it will hold for 1 second, waiting for an extra command to be sent serially by a computer, using the USB Cable. If this command is received, the unit will them acknowledge and enter USB Mode, instead of Midi Mode. At this time, any Note-On, Note-Off, ... Will be sent to the computer via the USB Cable. We have provided a special Windows VST Plugin that uses this special mode, allowing you to drive any computer based software that is compatible with VST plugins.

1.4 EXTENDED_DRUM_NAMES - This will add even more General Midi (GM) Drum Note Names, but also takes more Flash space. You can check the W_AStrng file to see the list of names, and even include your custom names. Mind you that there's a size limit, as the LCD has only so little space to show.

1.5 STORAGE_FORCE_INIT - Use this option only if somehow your machine is not working correctly and you can't reach the new File Mode -> System Initiation option.

1.6 All other options are used to Debug - check the software for problems or run tests - so don't mess with those options unless you know what you are doing.

2. Still in the setup() call, everything in the Hardware is setup and sysInit() is called, which will define default values.

3. The next step is to load information from the EEPROM chips from the last session and start up the Midi or USB device by clearing any notes that may be stuck in the target device. This is done with the sendMidiAllNotesOff() call, which is the last thing done by the setup() call.

4. After all this, the hardware will keep calling loop() which is located in the W_Loop Tab. The following shows the cycle of calls loop() produces, which is repeated forever.

   4.1 midiInputCheck() - will read any Midi Input information and process, checking for any Midi System Exclusive Data Dumps been received by the system. MidiInputCheck is located in the W_Midi Tab. Another thing this call does is check for the MIDIECHO option and echo any midi information if this option is set to 1. (On)

   4.2 doLCDupdate - if this is True, or if lastMillisLateLCDupdate was used, the LCD display will update according to the current selected mode. (Pattern, Song or File)

4.3 The next step is to read all the 16 Steps Buttons Input and also output all 16 Steps LEDs. Plus, check for the Interface Buttons. (Up / Down / Left / Right / Shift / Play / Stop / Rec) Each Mode has its own calls for each option. Below is some generic information.

4.3.1 For any mode, buttonsInputAndLEDsOutput() is used to read all buttons and output all LEDs. This call is located at the top of the W_Betc Tab. The extra 8 external button header is also read in this section, but currently no portion of the software uses the data, which is stored in the "extraExternal" 16-bit variable. (uint16_t)

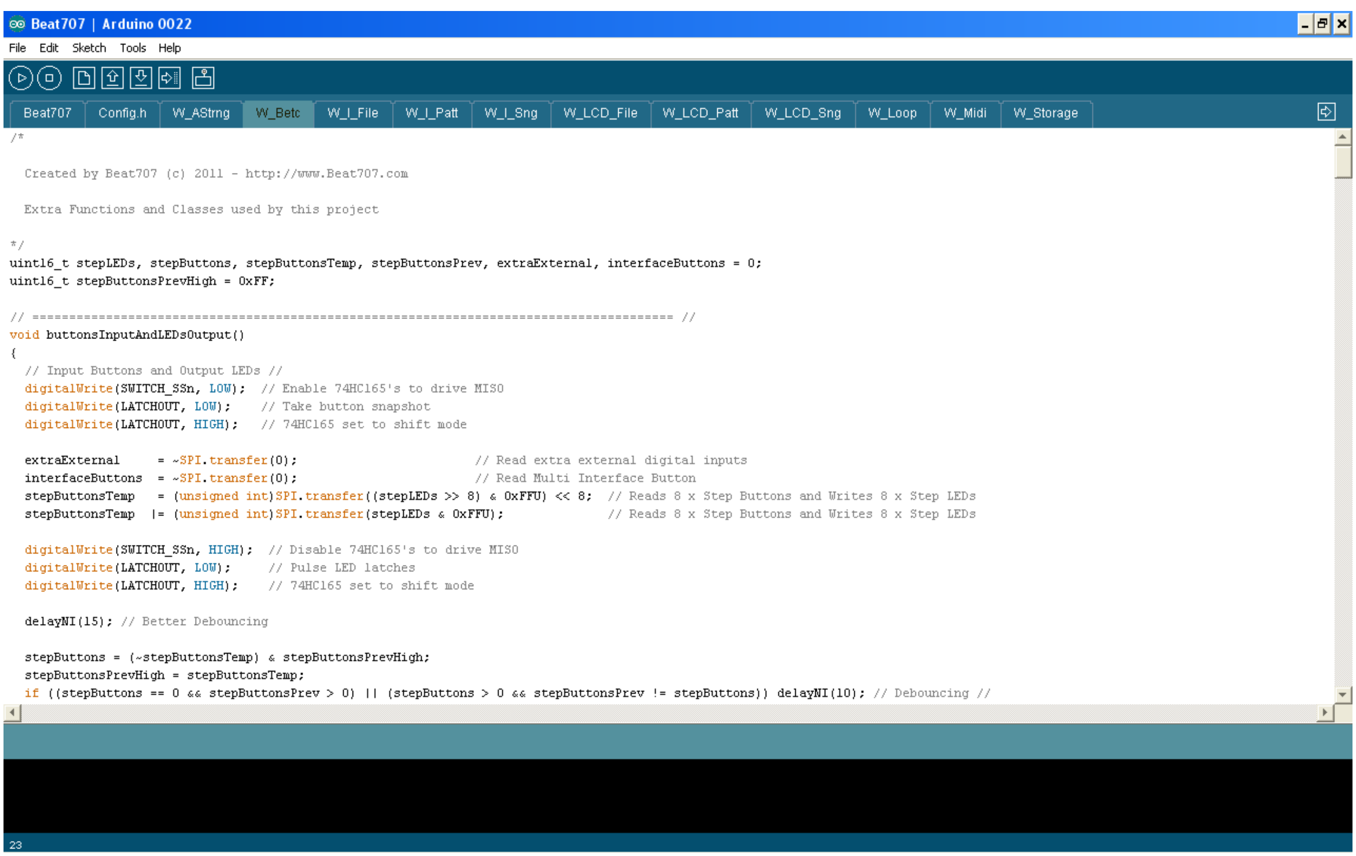
4.3.2 W_I_File / W_I_Patt / W_I_Sng - those are the Interface Tabs, where we define what will happen when you press Play, Stop, Rec, Up, Down, Left, Right and the Shift buttons. There's also a sector for the 16 Step Buttons and another for the 16 Step LEDs.

4.3.2.1 InterfaceTick**** - used by the Up / Down / Left / Right / Play / Stop & Rec buttons.

4.3.2.2 ShiftButton**** - called when you press the Shift button.

4.3.2.3 LEDs****Tick() - this has two functions: output the 16 Steps LEDs and input the 16 Steps Buttons.

4.3.3 W_LCD_File / W_LCD_Patt / W_LCD_Sng - those are the LCD Tabs, where we define how the LCD will be drawn according to the current selected mode.




```
/*

  Created by Beat707 (c) 2011 - http://www.Beat707.com

  Extra Functions and Classes used by this project

*/
uint16_t stepLEDs, stepButtons, stepButtonsTemp, stepButtonsPrev, extraExternal, interfaceButtons = 0;
uint16_t stepButtonsPrevHigh = 0xFF;

// ================================================================================== //
void buttonsInputAndLEDsOutput()
{
  // Input Buttons and Output LEDs //
  digitalWrite(SWITCH_SSn, LOW);  // Enable 74HC165's to drive MISO
  digitalWrite(LATCHOUT, LOW);    // Take button snapshot
  digitalWrite(LATCHOUT, HIGH);   // 74HC165 set to shift mode

  extraExternal     = ~SPI.transfer(0);                        // Read extra external digital inputs
  interfaceButtons  = ~SPI.transfer(0);                        // Read Multi Interface Button
  stepButtonsTemp   = (unsigned int)SPI.transfer((stepLEDs >> 8) & 0xFFU) << 8;  // Reads 8 x Step Buttons and Writes 8 x Step LEDs
  stepButtonsTemp  |= (unsigned int)SPI.transfer(stepLEDs & 0xFFU);             // Reads 8 x Step Buttons and Writes 8 x Step LEDs

  digitalWrite(SWITCH_SSn, HIGH);  // Disable 74HC165's to drive MISO
  digitalWrite(LATCHOUT, LOW);     // Pulse LED latches
  digitalWrite(LATCHOUT, HIGH);    // 74HC165 set to shift mode

  delayNI(15); // Better Debouncing

  stepButtons = (~stepButtonsTemp) & stepButtonsPrevHigh;
  stepButtonsPrevHigh = stepButtonsTemp;
  if ((stepButtons == 0 && stepButtonsPrev > 0) || (stepButtons > 0 && stepButtonsPrev != stepButtons)) delayNI(10); // Debouncing //
```

SPI Code to read Step Buttons and set Step LEDs

Midi Clock Information

The Midi Clock uses the Arduino's ATmega328 Timer1, which is a 16-Bit timer. (the Arduino system does not use this Timer) Messing up with the Timer Assembly code is a bit tricky, but we managed to make this clear by creating a few calls. All this is set in the W_Betc Tab: timerStart(), timerSetFrequency() and timerStop().

When the Timer is due, it will call an Interrupt named ISR(TIMER1_COMPA_vect), which is located in the top of the W_Midi Tab. This will call another thing named midiTimer() which is right below ISR(...). We ketp this separated so you can also use an external clock signal - eg: Midi Slave mode - 24 PPQ Midi Clock Signal.

We keep the code for midiTimer() as short and quick as possible, so it can be processed in time for the next clock call. Currently the system handles 96 PPQ without problems. PPQ = Pulses Per Quarter, meaning, 96 calls to midiTimer() per a Quarter Note.

Since we have 96 PPQ, we need to check when a note will happen, as the steps sequencer has only 12 PPQ internally. But we use 96 PPQ globally so we can do Midi Clock Shuffling/Swing and also do 24 PPQ External Sync without much extra code. Keep in mind that the Arduino code storage area is short, compared to larger systems.
We use the midiClockCounterDivider variable to count to 12, depending on the time scale used. (1/16 or 1/32) And we do two checks, if midiClockCounterDivider is over 12, or if its equal (==) 6. In this last case, it means is one of the extra hidden 1/32 (or 1/64) steps. Refer to the Beat707_DM_Manual or YouTube videos for some extra information about those extra steps.

Finally, midiTimer() also handles both Pattern and Song mode, by checking if a new pattern was selected during playback, and pre-loads to a temporary variable, so midi is not interrupted by this event. This is done by the patternBufferN variable, check W_Storage -> loadPattern to see what happens. All Pattern information uses a double matrix: one for current pattern, the other for the next pattern.

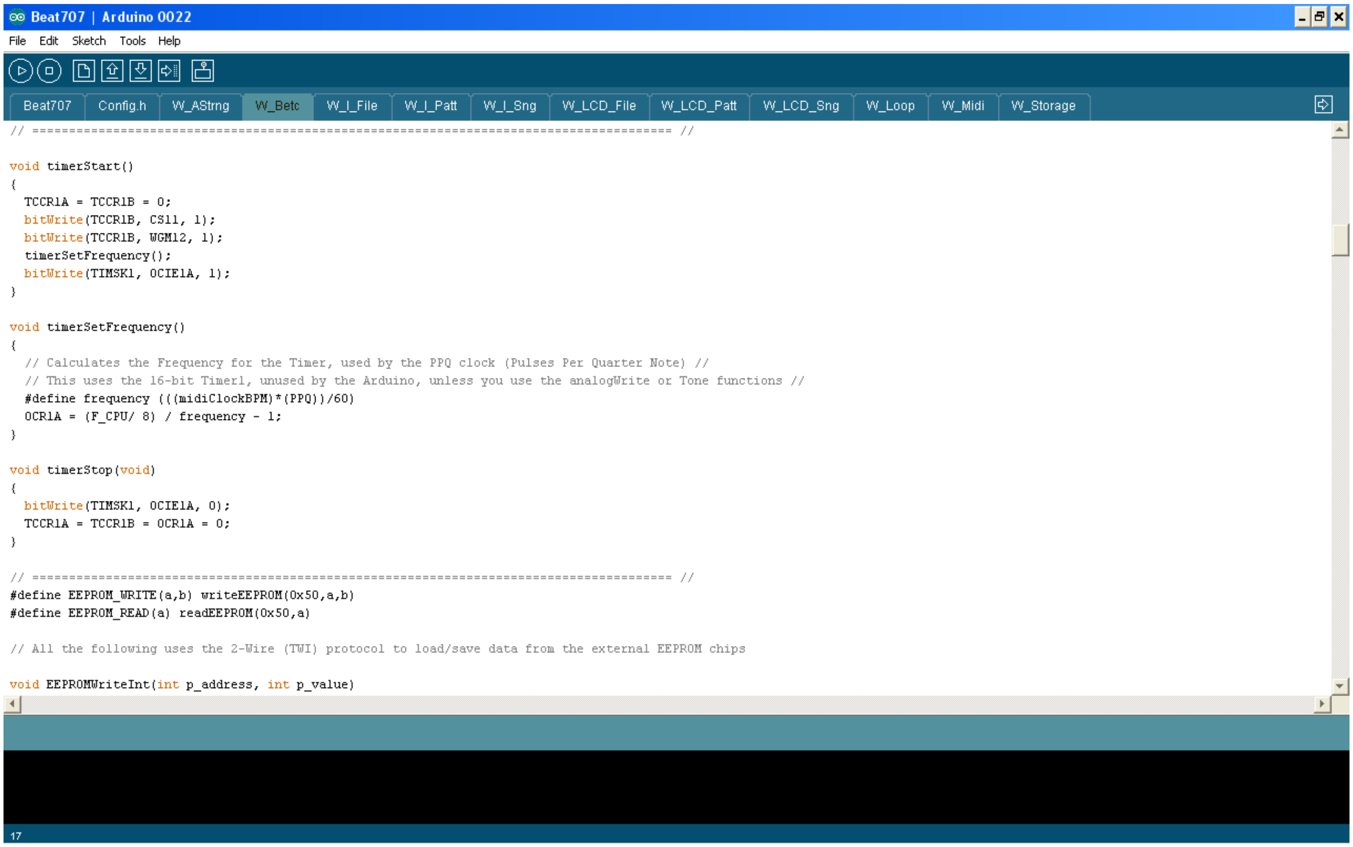
In case you want to check some extra Timer advanced functions, here's a quick list for Timer1 PWM modes. (we don't use PWM, only Interrupts)

```
Timer 1 Registers - ATmega328 Datasheet Page 113 (16 Bit Timer)

Bit          7       6       5       4       3       2       1       0
TCCR1A = COM1A1  COM1A0  COM1B1  COM1B0    R       R     WGM11   WGM10
TCCR1B = ICNC1   ICES1     R     WGM13   WGM12   CS12    CS11    CS10

COM1A = Out Pin 9
COM1B = Out Pin 10
WGM12 WGM10 = Mode 5 Fast PWM 8 Bits
WGM12 WGM11 WGM10 = Mode 7 Fast PWM 10 Bits
WGM13 WGM12 WGM11 = Mode 14 Fast PWM Top=ICR1
CS10 = No PreScaler

TIMSK1 = _BV(TOIE1); // timer overflow interrupt
sei(); // enable global interrupts
```



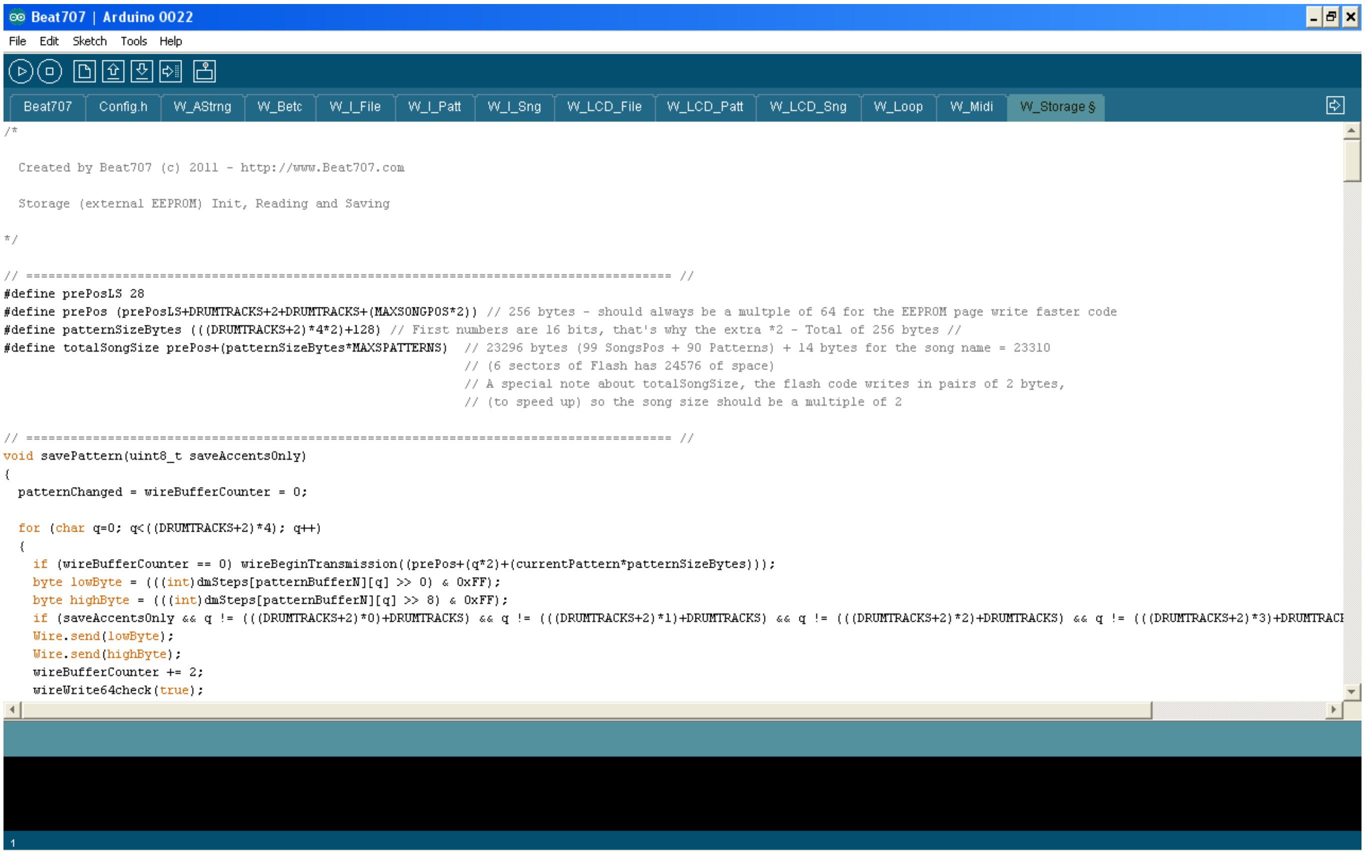
Timer1 Functions used by the Midi Clock Code

LCD Custom Characters

The Beat707 LCD interface allows the code to create up to 8 custom characters, this is done by the LcdCursors variable located in the W_Betc Tab. Each line of the code below is one custom character in binary format. Below we list the following Custom Characters:

- 0 = > Cursor
- 1 = X Mirror Editing
- 2 = > Recording Cursor
- 3 = A Pattern
- 4 = B Pattern
- 5 = A'' Pattern (extra hidden steps)
- 6 = B'' Pattern (extra hidden steps)
- 7 = X Mirror Recording Cursor

```
uint8_t LcdCursors[64] = {
      B00000,   B01000,   B01100,   B01110,   B01100,   B01000,   B00000,   B00000,
      B00000,   B01010,   B01110,   B01110,   B01110,   B01010,   B00000,   B00000,
      B11111,   B10111,   B10011,   B10001,   B10011,   B10111,   B11111,   B00000,
      B00000,   B00000,   B11110,   B10010,   B11110,   B10010,   B10010,   B00000,
      B00000,   B00000,   B11100,   B10010,   B11100,   B10010,   B11100,   B00000,
      B00011,   B00011,   B11110,   B10010,   B11110,   B10010,   B10010,   B00000,
      B00011,   B00011,   B11100,   B10010,   B11100,   B10010,   B11100,   B00000,
      B11111,   B10101,   B10001,   B10001,   B10001,   B10101,   B11111,   B00000   };
```



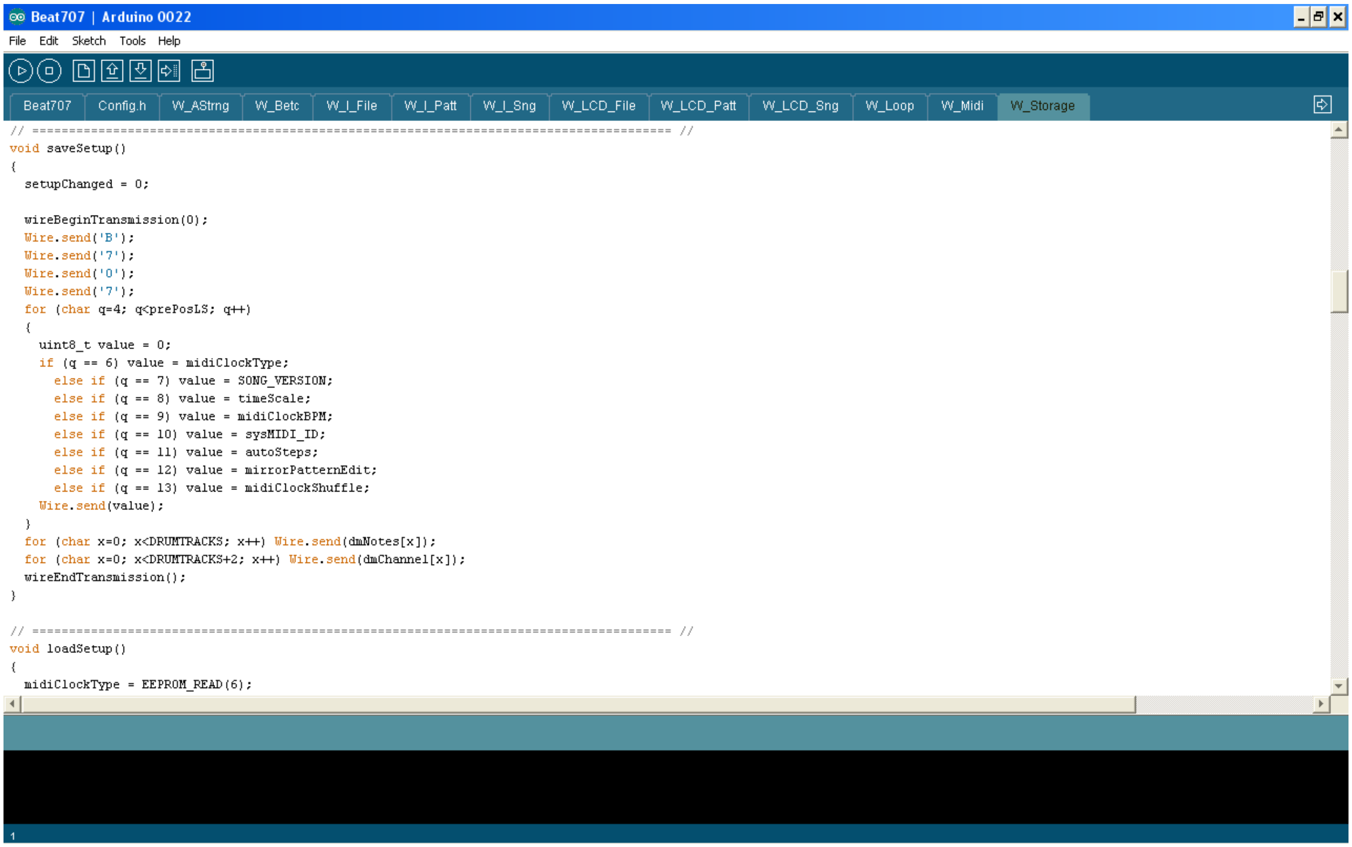
Patterns, Song and Setup Storage

Beat707 has 2 storage areas: EEPROM and Flash.

1. EEPROM = where everything is stored, nothing is just left in memory, so when you power down the unit all data is retained. By default Beat707 comes with a single EEPROM 32Kbytes chip, (256Kbits) but can be expanded to a total of 4 x 32Kbytes chips. Every time you change something, a variable is used to tell the code something has changed in the pattern, song or setup. When you hit Play or Stop it checks if something was changed and saves to the external EEPROM chip.

2. Nand Flash = where you can store complete songs. It will just copy the data from the EEPROM chip to specific sectors of the flash memory. The Flash chip has 512Kbytes of space. (4Mbits) The Flash chip has some drawbacks, one is that you can't just write to it as you wish. It will only allow you to write on empty locations, and you can only erase in blocks of 4096 bytes in the predetermined sectors. That's why we added the EEPROM chips instead of just using the Flash chip for everything. The EEPROM chip let you write any byte at any time, making it great for storing patterns as we need.

In order to speed up writing and loading processes, we use special Page Writing on both chips, which requires extra RAM but makes the whole process much faster. Keep in mind that the EEPROM chip Page Write is limited to a 64 byte boundaries and the Flash chip writes data in pairs of 2 bytes.

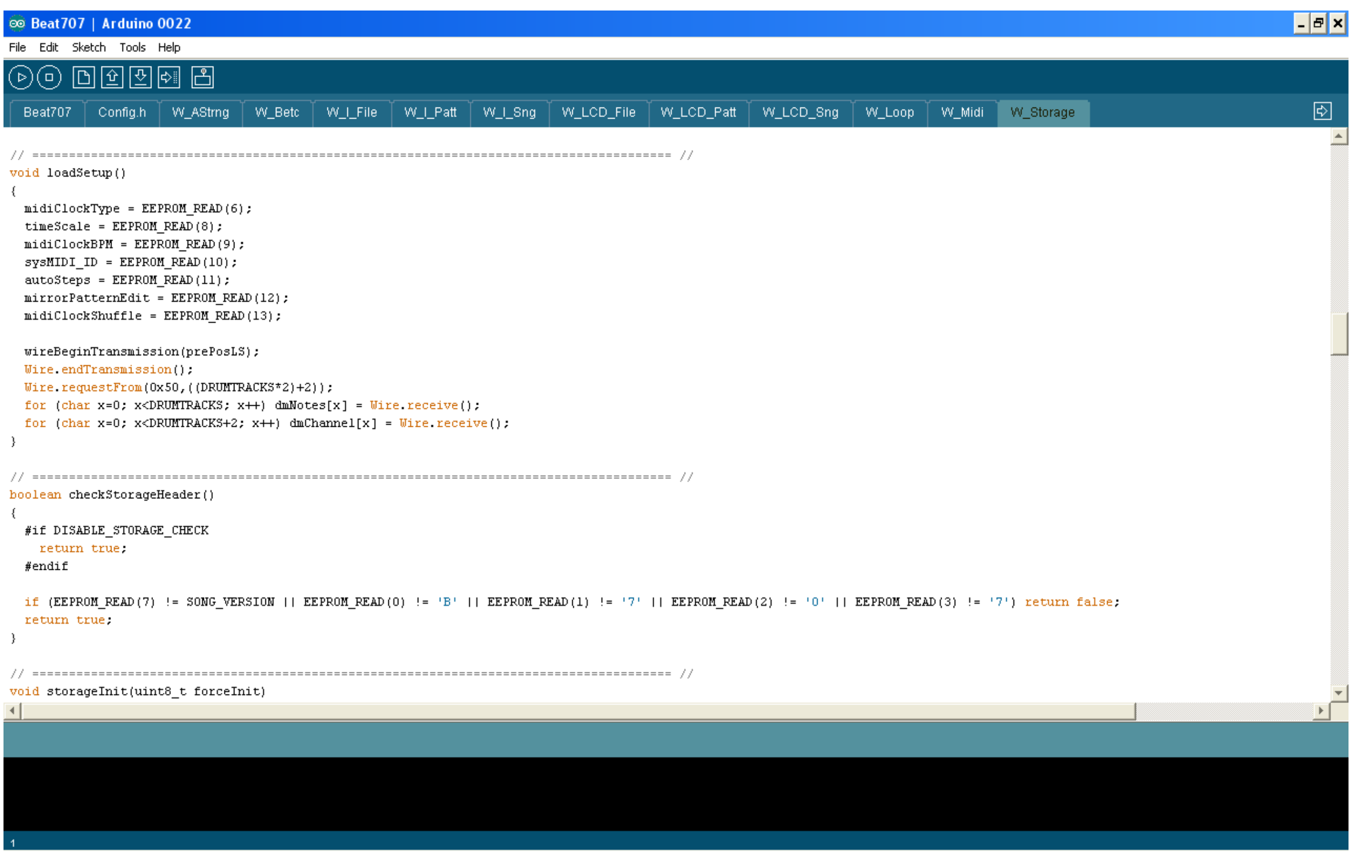


```
// ===================================================================================== //
void saveSetup()
{
  setupChanged = 0;

  wireBeginTransmission(0);
  Wire.send('B');
  Wire.send('7');
  Wire.send('0');
  Wire.send('7');
  for (char q=4; q<prePosLS; q++)
  {
    uint8_t value = 0;
    if (q == 6) value = midiClockType;
      else if (q == 7) value = SONG_VERSION;
      else if (q == 8) value = timeScale;
      else if (q == 9) value = midiClockBPM;
      else if (q == 10) value = sysMIDI_ID;
      else if (q == 11) value = autoSteps;
      else if (q == 12) value = mirrorPatternEdit;
      else if (q == 13) value = midiClockShuffle;
    Wire.send(value);
  }
  for (char x=0; x<DRUMTRACKS; x++) Wire.send(dmNotes[x]);
  for (char x=0; x<DRUMTRACKS+2; x++) Wire.send(dmChannel[x]);
  wireEndTransmission();
}

// ===================================================================================== //
void loadSetup()
{
  midiClockType = EEPROM_READ(6);
```

Setup Storage Code - Saving and Loading



```
// ===================================================================================== //
void loadSetup()
{
  midiClockType = EEPROM_READ(6);
  timeScale = EEPROM_READ(8);
  midiClockBPM = EEPROM_READ(9);
  sysMIDI_ID = EEPROM_READ(10);
  autoSteps = EEPROM_READ(11);
  mirrorPatternEdit = EEPROM_READ(12);
  midiClockShuffle = EEPROM_READ(13);

  wireBeginTransmission(prePosLS);
  Wire.endTransmission();
  Wire.requestFrom(0x50,((DRUMTRACKS*2)+2));
  for (char x=0; x<DRUMTRACKS; x++) dmNotes[x] = Wire.receive();
  for (char x=0; x<DRUMTRACKS+2; x++) dmChannel[x] = Wire.receive();
}

// ===================================================================================== //
boolean checkStorageHeader()
{
  #if DISABLE_STORAGE_CHECK
    return true;
  #endif

  if (EEPROM_READ(7) != SONG_VERSION || EEPROM_READ(0) != 'B' || EEPROM_READ(1) != '7' || EEPROM_READ(2) != '0' || EEPROM_READ(3) != '7') return false;
  return true;
}

// ===================================================================================== //
void storageInit(uint8_t forceInit)
```

```
/*

  Created by Beat707 (c) 2011 - http://www.Beat707.com

  Song User Interface

*/

// ======================================================================================= //
void InterfaceTickSong()
{
  switch (multiButton)
  {
    // ------------------------------ STOP ------------------------------ //
    case 0:
      if (songChanged) saveSongPosition();
      if (!midiClockRunning) curSongPosition = 0;
      if (midiClockRunning) MidiClockStop();
      if (setupChanged) saveSetup();
      checkPatternLoader();
      recordEnabled = 0;
      curZone = 0;
      loadSongPosition();
      updateLCDSong();
      break;

    // ------------------------------ PLAY ------------------------------ //
    case 1:
      startSong();
      break;

    // ----------------------------- RECORD ----------------------------- //
```

Interface Multi-Button Code

```
    // ------------------------------ PLAY ------------------------------ //
    case 1:
      lateAutoSave = 1;
      if (midiClockRunning) MidiClockStop();
      MidiClockStart();
      break;

    // ------------------------------ LEFT ------------------------------ //
    case 2:
      if (holdingShift) { curZone = 0; holdingShiftUsed = 1; } else { curZone--; if (curZone == 255) curZone = 11; }
      updateLCDPattern();
      break;

    // ------------------------------ RIGHT ----------------------------- //
    case 5:
      if (holdingShift) { curZone = 11; holdingShiftUsed = 1; } else { curZone++; if (curZone > 11) curZone = 0; }
      updateLCDPattern();
      break;

    // ------------------------------- UP ------------------------------- //
    case 4:
      if (holdingShift && !recordEnabled)
      {
        shiftMode++;
        if (shiftMode > 7) shiftMode = 0;
        holdingShiftUsed = 1;
        showOnlyOnce = 1;
      }
      else if (curZone == 0)
      {
        if ((!autoSteps || !midiClockRunning) && !editStepsPos && !mirrorPatternEdit)
        {
```

Special Midi USB Mode

If the CHECK_FOR_USB_MODE definition is 1, (set in the Config.h file) upon initiation of the unit, the system will send a special code serially to check if a computer is listening and will enter Midi USB Mode instead, if required. The process is simple, it follows the following steps:

1. Init Serial at 57600 Bps.
2. Sends SysEx "B707" to the serial output.
3. Keep the Midi Interface Off, don't enable it just yet.
4. Init everything else in the device.
5. Wait 1 Second.
6. Check if data was sent to the Serial Input.
7. If Data is equal to SysEx "USB" it will flag keepInUSBmode as True, otherwise it will leave it alone. (False)
8. If keepInUSBmode is True the device will keep the serial at 57600 Bps, it won't enable the Midi Interface, and will also send to the Serial Output the current Midi SysEx ID, so the computer can work with multiple devices, if needed.
9. If keepInUSBmode is False, the device will re-initiate the Serial port at 31250 Bps and will Initiate the Midi Interface by setting MIDI_Enn to Low. This will make the hardware use the external Midi Input and Output connectors instead.

```
void setup()
{
  pinMode(MIDI_ENn,OUTPUT);
  #if CHECK_FOR_USB_MODE
    MSerial.begin(57600); // Startup in USB Mode //
    digitalWrite(MIDI_ENn,HIGH);
    MSerial.write(240); MSerial.write('B'); MSerial.write('7');
    MSerial.write('0'); MSerial.write('7'); MSerial.write(247);
  #endif
```

```
  #if CHECK_FOR_USB_MODE
    unsigned long endtime = timer0_millis + 1000;
    while (((long)endtime - (long)timer0_millis) > 0) { ; } // Don't use delayNI here as it clears up MIDI Data

    uint8_t keepInUSBmode = false;
    if (MSerial.available() > 0)
    {
      keepInUSBmode = true;

      if (MSerial.read() != 240) keepInUSBmode = false;
        else if (MSerial.read() != 'U') keepInUSBmode = false;
        else if (MSerial.read() != 'S') keepInUSBmode = false;
        else if (MSerial.read() != 'B') keepInUSBmode = false;
        else if (MSerial.read() != 247) keepInUSBmode = false;
    }

    if (!keepInUSBmode)
    {
      MSerial.begin(31250); // Regular MIDI Interface
      digitalWrite(MIDI_ENn,LOW);  // Write to MIDI OUT, MIDI IN enabled
    }
    else
    {
      MSerial.write(240); MSerial.write('I'); MSerial.write('D'); MSerial.write(sysMIDI_ID); MSerial.write(247);
      #if SHOW_USB_MODE
        lcd.clear();
        lcdPrintString("USB Mode Ready");
        delayNI(1000);
      #endif
    }
  #endif
```

Delay() Timer Interrupts and the Midi Clock

One problem we had with the delay() function was that the Midi Clock was having problems, as delay() uses millis() which disables and re-enables Interrupts, messing up with the Midi Clock Interrupt. In order to fix this problem, we had to create our own delay and millis functions, which we called delayNI() and millisNI(), as in NI = Non-Interrupt. Those functions are at the bottom of the first Tab named Beat707.

Another thing we had to check was the Midi Input if MIDIECHO was set to 1. (On) Since we use delayNI() in some portions of the loop() code, we had to also check if no midi input was been feed so we could output it as quick as possible, so it wouldn't add any latency. Therefore, when MIDIECHO is used, we also check for midi data inside the delayNI() call as seem below.

```
unsigned long millisNI(void) { return timer0_millis; }
void delayNI(unsigned long ms)
{
  unsigned long endtime;
  endtime = timer0_millis + ms;
  while (((long)endtime - (long)timer0_millis) > 0)
  {
    #if MIDIECHO
      midiInputCheck();
    #else
    ;
    #endif
  }
}
```

Displaying Strings in the LCD (saving RAM)

Since we use a lot of strings to display in the LCD, we had to use PROGMEM in order to store strings in flash instead of RAM. One way to do this was to create a structure of strings, which has proved to work perfectly.

Another problem was to get rid of the Lcd.Print() function which uses too much Flash Space. Instead, we wrote our own functions for writing strings and numbers to the Lcd display.

Beat707 | Arduino 0022

File Edit Sketch Tools Help

Beat707 | Config.h | W_AStrng | W_Betc | W_I_File | W_I_Patt | W_I_Sng | W_LCD_File | W_LCD_Patt | W_LCD_Sng | W_Loop | W_Midi | W_Storage

```
/*

  Created by Beat707 (c) 2011 - http://www.Beat707.com

  Strings to be stored as Program (not in RAM, but in Flash)

*/

// GM Drum Set //
#if EXTENDED_DRUM_NAMES
  prog_char string_1[] PROGMEM   = "AcBass";  // 35
  prog_char string_2[] PROGMEM   = "Bass";    // 36
  prog_char string_3[] PROGMEM   = "Stick";   // 37
  prog_char string_4[] PROGMEM   = "Snare";   // 38
  prog_char string_5[] PROGMEM   = "Clap";    // 39
  prog_char string_6[] PROGMEM   = "Snare2";  // 40
  prog_char string_7[] PROGMEM   = "LFlTom";  // 41
  prog_char string_8[] PROGMEM   = "ClosHat"; // 42
  prog_char string_9[] PROGMEM   = "HFlTom";  // 43
  prog_char string_10[] PROGMEM  = "PedlHat"; // 44
  prog_char string_11[] PROGMEM  = "LowTom";  // 45
  prog_char string_12[] PROGMEM  = "OpenHat"; // 46
  prog_char string_13[] PROGMEM  = "LMTom";   // 47
  prog_char string_14[] PROGMEM  = "HMTom";   // 48
  prog_char string_15[] PROGMEM  = "Crash";   // 49
  prog_char string_16[] PROGMEM  = "HiTom";   // 50
  prog_char string_17[] PROGMEM  = "RideCmb"; // 51
  prog_char string_18[] PROGMEM  = "Chinese"; // 52
  prog_char string_19[] PROGMEM  = "RdeBell"; // 53
  prog_char string_20[] PROGMEM  = "Tmbrine"; // 54
  prog_char string_21[] PROGMEM  = "Splash";  // 55
  prog_char string_22[] PROGMEM  = "Cowbell"; // 56
```

Done Saving.

1

String PROGMEM Storage functions

Beat707 | Arduino 0022

File Edit Sketch Tools Help

Beat707 | Config.h | W_AStrng | W_Betc | W_I_File | W_I_Patt | W_I_Sng | W_LCD_File | W_LCD_Patt | W_LCD_Sng | W_Loop | W_Midi | W_Storage

```
prog_char myStrings_058[] PROGMEM  = "Processing";
#define PROCESSING 97
prog_char myStrings_059[] PROGMEM  = "! Empty Song !";
#define E_EMPTY_SONG_E 98
prog_char myStrings_060[] PROGMEM  = "C C#D D#E F F#G G#A A#B ";
#define NOTENAMESLIST 99
prog_char myStrings_061[] PROGMEM  = "On";
#define ON_ 100
prog_char myStrings_062[] PROGMEM  = "Off";
#define OFF_ 101
prog_char myStrings_063[] PROGMEM  = "Sld";
#define SLD 102
prog_char myStrings_064[] PROGMEM  = "MirrorEdMode";
#define MIRROREDMODE 103
prog_char myStrings_065[] PROGMEM  = "ClockShuffle";
#define CLOCKSHUFFLE 104


PROGMEM const char *stringlist[] = { empty_Str,
#if EXTENDED_DRUM_NAMES
  string_1, string_1, string_1, string_1, string_2, string_3, string_4, string_5, string_6, string_7, string_8, string_9, string_10, string_11, string_12, string_13, string_14, str
  string_16, string_17, string_18, string_19, string_20, string_21, string_22, string_23, string_24, string_25, string_26, string_27, string_28, string_29, string_30, string_31, st
#else
  empty_Str, empty_Str, empty_Str, empty_Str, string_2, string_3, string_4, string_5, empty_Str, empty_Str, string_8, empty_Str, empty_Str, empty_Str, string_12, empty_Str, empty_S
  empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, empty_Str, em
#endif
  myStrings_001, myStrings_002, myStrings_003, myStrings_004, myStrings_005, myStrings_006, myStrings_007, myStrings_008, myStrings_009, myStrings_010, myStrings_011,
  myStrings_012, myStrings_013, myStrings_014, myStrings_015, myStrings_016, myStrings_017, myStrings_018, myStrings_019, myStrings_020, myStrings_021, myStrings_022,
  myStrings_023, myStrings_024, myStrings_025, myStrings_026, myStrings_027, myStrings_028, myStrings_029, myStrings_030, myStrings_031, myStrings_032, myStrings_033,
  myStrings_034, myStrings_035, myStrings_036, myStrings_037, myStrings_038, myStrings_039, myStrings_040, myStrings_041, myStrings_042, myStrings_043, myStrings_044,
  myStrings_045, myStrings_046, myStrings_047, myStrings_048, myStrings_049, myStrings_050, myStrings_051, myStrings_052, myStrings_053, myStrings_054, myStrings_055,
  myStrings_056, myStrings_057, myStrings_058, myStrings_059, myStrings_060, myStrings_061, myStrings_062, myStrings_063, myStrings_064, myStrings_065};
```

Done Saving.

205

Reducing Flash Size

For V1.2.0 we went and tested everything we could do in order to reduce the size of the whole program, as the ATmega328 has only so little flash storage space - 32Kbytes.

The first thing we did was to work on the Lcd Library, as it has a lot of code we don't need, since the Beat707 Hardware is set in stone to work only at 4 bit-mode and 16 x 2 Displays. Therefore, we created a copy of the Liquid Crystal library and started hacking it until we had only what we were using. This reduced code-size by nearly 2K, which is a bit hit for the Arduino code.

Initially we were using the 8-bit Timer2 for the Midi Clock, but it required a lot of extra code to check for all Processor Scalars possible, so the resulting Timer would be as near as possible to the requested HZ speed for each PPQ. By switching to a 16-bit Timer1, the code was reduced by 90%.

For both Flash and EEPROM writing, we also created small functions to start and end each process, therefore reducing the number of redundant code laying around.

Another big thing we needed to get rid of was any math using floats, as they take a lot of extra flash space in order to run. The only place we were still using floats was in the Timer Frequency code. After some tests and a lot of help from the Arduino Community, we figured out we didn't really need to use floats. There was some differences on using integers and floats, but it was marginal. We even created a small sketch to test the theory.