Code-Read and Write Cores

Core 64: Interactive Core Memory Badge, V0.1, Andrew Geppert, 2020-02-17

User Application

Use the cores to do something interesting

FreeRTOS

Facilitate running all task threads

**Core\_Mem\_HAL.h (with new API)**

IMPLEMENTED

Core\_Mem\_Bit\_Write (bit 0-63, 0/1)

Core\_Mem\_Bit\_Read (bit 0-63)

Core\_Mem\_Array\_Write()

NOT YET FULLY IMPLEMENTED

Core\_Mem\_Init (create an instance)

Core\_Mem\_Test (read all, clear all, verify clear, rewrite original state, verify rewrite, return 0-no test pass, 1st passed, 2nd passed)

Core\_Mem\_Wipe (set all to 0 or 1)

Core\_Mem\_Visual\_Write (x, y)

Core\_Mem\_Bit\_Toggle (bit 0-63)

Core\_Mem\_Byte\_Write (Byte 0-7, 0-255)

Core\_Mem\_Byte\_Read (Byte 0-7)

Core\_Mem\_Word\_Write (0x0-FFFFFFFF FFFFFFFF)

Core\_Mem\_Word\_Read ()

**Core\_Mem\_HAL.c**

Enables user application to interaction with the abstract core memory space (as a buffer) and request read/write to the real core memory.

Interaction is done in one of two modes: Visual (x,y)

logical (bits, bytes, word).

These functions ask the driver to do the detailed work.

HAL: Interact with the hardware in an abstract and conceptual way, using high level perspective. Like visual and logical arrangements.

**Core\_Mem\_Driver.h**

AllDriveIoSafe()

AllDriveIoRead()

AllDriveIoWrite()

AllDriveIoSetBit(bit)

AllDriveIoClearBit(bit)

AllDriveIoEnable()

AllDriveIoDisable()

HAL: Control the logical array to access the array of cores

**Core\_Mem\_Driver.c**

Translate bit # to row/col position.

Functions to control the array of wires.

Function to read sense signal.

Indirectly controlling hardware as active and Inactive.

**Core\_Mem\_Driver.h**

AllDriveIoSafe()

AllDriveIoRead()

AllDriveIoWrite()

AllDriveIoSetBit(bit)

AllDriveIoClearBit(bit)

AllDriveIoEnable()

AllDriveIoDisable()

**Sub-Functions**

MatrixEnableTransistorInactive()

MatrixEnableTransistorActive()

MatrixDriveTransistorsInactive()

SetRowAndCol (uint8\_t row, uint8\_t col)

ClearRowAndCol (uint8\_t row, uint8\_t col)

Translate to New discrete functions:

Core\_IO\_Enable\_Line (on, off)

Core\_IO\_XY\_Lines\_All\_Safe

Core\_IO\_XY\_Line\_Pair (north, south)

Core\_IO

DRIVER: Control the pins of the microcontroller

**Core\_Mem\_Driver\_Config.h**

Define the size, layout, matrix to drive transistor to wire mapping, maybe the MCU memory buffer.

**Core\_Driver.c**

Accesses the core memory bits by directly twiddling the IO lines in the core matrix. Controls the electronics (enable/drive transistors, read sense signal) through #include HardwareIOmap.h, Arduino.h. Translates active/inactive to low/high for the IO signals.

BSP: Abstraction of MCU pin reference number to the physical pin on the chip

**Arduino.c and .cpp in the Arduino IDE**

Abstraction of MCU pin to integer pin reference number

**Arduino.h**

2020-01-06 updated thinking on the layers above.

Top Levels as User Application and the FreeRTOS is OK.

Next level down is the DRIVER which has an API defined in it, and accessible from the Application.

Layer below that is the HAL and it reads and writes to the IO lines. But not directly.

BSP is the bottom layer of firmware, and it maps the friendly names of the IO lines in the HAL to the MCU specific pins. To keep things simple, this is effectively the Arduino layer.

Then there is the hardware.

EXAMPLE WITH LED ARRAY

DRIVER

LED\_ARRAY\_DRIVER.cpp (was LED\_Array.cpp) and is in the DRIVERS folder. Includes LED\_ARRAY\_DRIVER.h.

The application accesses the driver by calling API functions in the driver.

Set each LEDs on/off or color state in a memory buffers.

Read each LEDs state from the memory buffers.

Clear the memory buffers.

Request a memory buffer to be displayed on the LEDs.

Test functions: cycle through all LEDs in order of 1D and 2D arrangements to confirm expected order of the LEDs.

The driver contains the LED Array memory buffer.

Monochrome 2D 8x8 image as viewed by the user.

Monochrome 1D 64 bit word, with LSB on lower right, MSB on left. First row is low byte of 64 bit word, top row is highest byte.

Color 2D 8x8 image as viewed by the user.

HAL

Hardware Abstraction Layer, or in this case, Library.

The FastLED-3.3.2 library in the LIBRARIES folder.

This is implemented to talk to the hardware and has many of its own layers and BSPs.

BSP

Combination of files buried in the FastLED Library, and Arduino.

Set – top end of the column wire and left end of row wire are positive (VMEM). Other ends negative (GNDPWR).

Clear - top end of the column wire and left end of row wire are negative (GNDPWR). Other ends positive (VMEM).

Each end of a single row or column wire can be connected to either VMEM (3-4V) or GNDPWR (0V). That means that two transistors must be enabled for a row or column wire to be energized. Since a core is set or cleared by the current from TWO wires, a total of 4 transistors must be enabled .



4403 PNP – Enabled with base LOW, or negative.

High side (sourcing) odd numbered matrix drive pins (7) with P suffix (Q4P). The load is connected to negative and the PNP transistor switches the positive voltage.

4401 NPN – Enabled with base HIGH, positive.

Low side (sinking) even numbered matrix pins (8) with N suffix (Q4N). The load is connected to positive and the NPN transistor switches the negative (low) side.

Safe state, all transistors inactive:

Enable Pin = LOW

Matrix Drive 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 = HIGH

Matrix Drive 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 = LOW

This outline of how it works could be shown as three indented levels. Level one is DRIVER, level two is HAL, level three is HARDWARE.

1. Since the output lines used to address the matrix may have other functions (like 17 is shared with the onboard Teensy LC LED), need to check all of the lines to see if any are not in the safe state for addressing the matrix.
   1. If a line is set other than safe [future work]
   2. save the state for later reference [future work]
   3. set all matrix lines and the enable line to safe states.
2. Set a core to 1 (bit 0, upper left corner)
   1. Set column (X0)
      1. Connect top end of column (XT0) to VMEM
         1. Activate top transistor (Q3P), Pin\_Matrix\_Drive\_5, LOW
      2. Connect bottom end of column (XB0) to GNDPWR
         1. Activate bottom transistor (Q1N), Pin\_Matrix\_Drive\_2, HIGH
   2. Set row (Y0)
      1. Connect left end of row (YL0) to VMEM
         1. Activate left transistor (Q7P), Pin\_Matrix\_Drive\_13, LOW
      2. Connect right end of row (YR0=YR4=YL4=YL6) to GNDPWR
         1. Activate right transistor (Q9N), Pin\_Matrix\_Drive\_17, HIGH
   3. Set Enable
      1. HIGH
      2. Wait ?? ns to lock in the magnetized state.
      3. Clear enable
   4. Return 4 transistors to safe states
   5. Return any other shared lines that weren’t initially safe, to the state they were at.
   6. End
3. Clear a core to 0 (bit 0, upper left corner)

Decode bit position (0…63) to matrix drive transistors (1…20)

Each row of this array corresponds to the bit #. First row is bit 0, last row is bit 63.

Row contains four matrix drive #s, each followed by the corresponding low/high needed to activated the transistor.

Odd numbered drive line transistors are active LOW

Even numbered drive line transistors are active HIGH

uint8\_t CoreMemoryMatrixDriveSetBitTransistors[] = {

{ 5,0,2,1, 13,0,17,1},

…

{ }

};

Store core value as:

uint64\_t CoreMemorySixtyFourBit = 0b0000…1111;

-AND-

bool CoreMemoryArray[] = {

{ 0, 1, 2, 3, 4, 5, 6, 7, 8},

…

{55,56,57,58,59,60,61,62,63}

};

Core Library Basics

1. CoreWriteBit (position #, value)
   1. Position # is 0 to 63 for the 64 bits. Starting upper left as 0, left to right, top to bottom, ending with bit 63 in the lower right.
   2. Value is Boolean 0 or 1
   3. Returns nothing
2. CoreReadBit (position #)
   1. Returns value Boolean 0 or 1
3. CoreWriteByte (position #, value)
   1. Position # is 0 for the top row, 7 for bottom row. MSb is left.
   2. Value is uint8\_t 0-255
   3. Returns nothing
4. CoreReadByte (position#)
   1. See above
5. CoreWrite64bits (value)
   1. Value is uint64\_t
   2. Returns nothing
6. CoreRead64bits (void)
   1. Return uint64\_t