9: Memory

Thursday 9th of November, 2017

Intro

In Princeton/Von Neumann Architecture, the program and data are in the same memory.

In Harvard Architecture, where the program and data are in separate memories.

Microcontrollers typically use Harvard architecture, because the program needs to persist while the controller is powered down as much as possible (to conserve energy).

Different Memories

There's flash memory, SRAM, and EEPROM in the Arduino we're using. Typically flash is for the program, SRAM is for data, and the EEPROM is for configuration data. The SRAM is also known as the heap.

The flash and EEPROM both persist if the power is cut, so when the system reboots it can read configuration data from EEPROM to restore state.

Typically the flash memory can't be modified from executing code.

String literals are all stored in flash, and are copied into the SRAM when the program starts. This can take up a lot of working memory. There's a system macro `F()` which can be used to keep strings in flash and not copy them to SRAM – you have to invoke special functions to read the string from flash when you need it. This saves memory but costs time.

The flash memory also contains a bootloader, which loads your program into flash. If you need to reclaim this space, it's possible

Heap / SRAM

This stores dynamically allocated data, and grows upwards from the top of the static data section (towards the stack, which grows downwards), consuming free memory.

There are some system defined values:

- `__heap_start` – the bottom of the heap
- `__brkval` – the top of the heap

Using macros for functions means you don't create stack frames.

If the heap and stack collide, behaviour is undefined.

There're available functions like `freeRAM()` which determine how much space is available between the heap and the stack. It doesn't take heap fragmentation into account.

EEPROM

Non-volatile memory (persists) – can be read/written from an executing program, but is done byte-by-byte and is much slower than SRAM.

Programmer needs to keep track of memory usage here.

How to Save Space

Flash

You can remove redundant code, e.g. using conditional compilation.

You can also remove redundant variables – global variables take up space in flash and in SRAM. (?)

SRAM

- Remove unused variables.
- Literal strings are copied to SRAM as static data at program startup. This is the same for literal numbers.
- Place constant data into program memory – do this using the keyword `PROGMEM` in the declaration:

  "'c #include <avr/pgmspace.h>

  const byte table[] PROGMEM = {0, 1, 4, 9, 16, 25, 36, 49};

  pgm_read_byte(&table[i]); "'
- There's also e.g. `pgm_read_int()` for integers.
- Reduce buffer sizes – check libraries for use of large buffers, and reduce them.
- Reduce oversized variables – use `char` instead of `int`, etc.

- Consider global variables vs. local variables.
  - Thought required here for appropriate tradeoff
- Avoid dynamic memory allocation to prevent heap fragmentation.
- Consider conditional compilation to insert/remove debug code:

  “‘c #define DEBUG

  … #ifdef DEBUG … #endif “‘

  - The statements in the if are only executed if DEBUG is defined. You can define DEBUG in the source code, or using an argument to the compiler.

Reading/Writing to the EEPROM

```
#include […]

EEPROM.write(address, byte);
```

Can only write one byte at a time, so need to write bytes of an integer separately, and need to either use big-endian or little-endian representation. Microcontrollers typically use little-endian, but don’t always.

There are predefined functions:

```
EEPROM.write(addr, lowByte(number));
EEPROM.write(addr + 1, highByte(number));

byte loByte = EEPROM.read(addr);
byte hiByte = EEPROM.read(addr + 1);

int i = word(hiByte, loByte);
```