CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

What is wrong with this call to malloc?

```
//prototype of malloc() (for reference)
void* malloc (size_t size);


void main (void){

    int * myBigArray;

    //allocate space for an array of a hundred ints:
    malloc(100 * sizeof(int));

    if (myBigArray = NULL){
        //do something
        .
        .
        .
}
```

myBigArr = ???

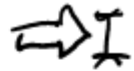
Roadmap

- Announcements, more lab4 notes
- Dynamic memory review
- Linked Lists
- Break
- Test Harness design
- Other Lab5 stuff
- Intro to Embedded Code (?)
  - Hardware
  - xc.h

Announcements

- Revote on push-based reports
- Grading mechanism tweak

Lab 4 notes



- LF / CR + LF
- Git diff
- Weak test harnesses
- Use your debugger

1) find the problem line(s)
2)

Dynamic memory review

Dynamic/static/stack

Dynamic memory	Static Memory	Stack memory
Size determined at runtime	Size determined at compile time	Size determined at compile time
Can leak	Cannot leak	Cannot leak
Program must keep track of a pointer to each block		
Coder must arrange data in block	Compiler arranges data	Compiler arranges data
Data persists until program discards it	Data persists forever	Data persists while program is in scope of data
Code can overwrite other data if you aren't careful	Compiler protects against overwriting other code	Compiler protects against overwriting other code

array [10] = 5;

Why use dynamic memory

- Use dynamic memory when:
  - You don't know at compile time how much memory you will need
  - On an OS, when you need LOTS of memory (stacks are often size-limited)

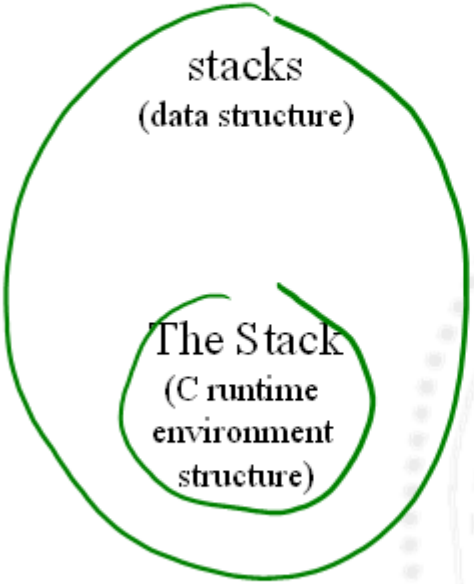
- Don't use dynamic memory when:
  - You don't want the overhead
  - You want speed
  - You want safety

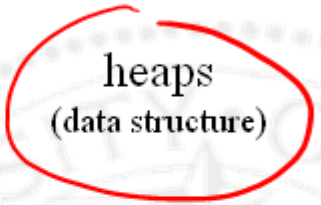
Not generally good for embedded

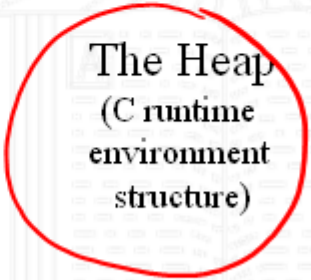
heaps

Max Lichtenstein

The Stack ∈ {stacks},
The Heap ∉ {heaps}

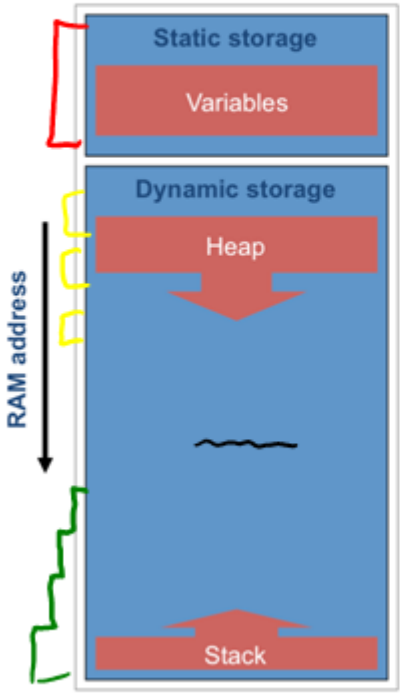




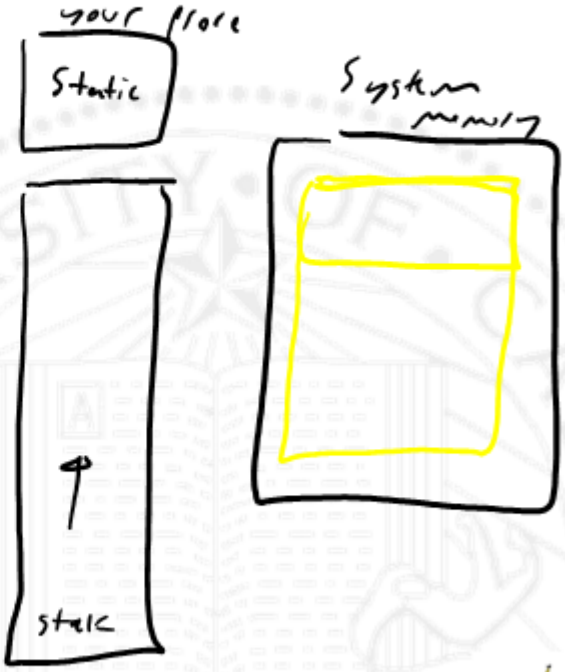


Data memory

Embedded

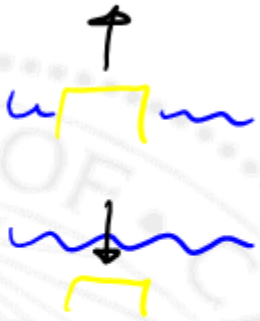


OS



Basic allocator calls

- malloc()
  - allocate space on heap (or try, anyway)
- free()
  - Deallocate space on heap
- calloc()
  - (attempt to) allocate space AND zero it out
- realloc()
  - (attempt to) increase the size of an existing block



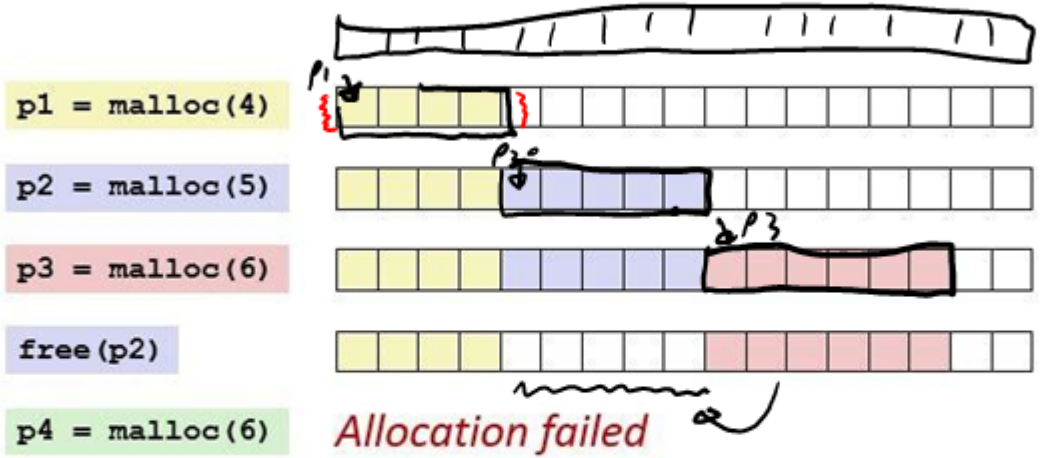
Allocator


Lionel Hutz
FOR SALE


FOR SALE
COBY
REAL ESTATE
Sheila & Joel
HAMMOND
661 555 0196


I need 1000 bytes
here's your new land!
123 Fake St
0x A000 24
sure, here's your address
0

malloc() and free() in action



p4 = NULL

Using malloc()

- ALWAYS ALWAYS ALWAYS:
  - Save returned pointer
  - Check returned pointer
  - free() before destroying
    or losing returned pointer

free(p1);

- Owning real estate is a responsibility!

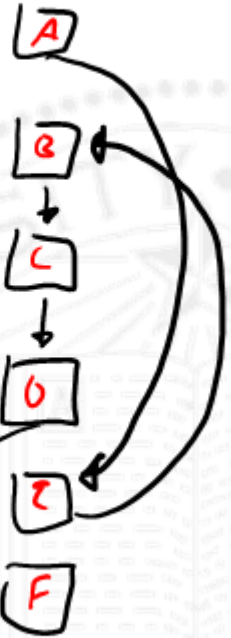
Linked Lists

And other Lab05 stuff

Lab 5

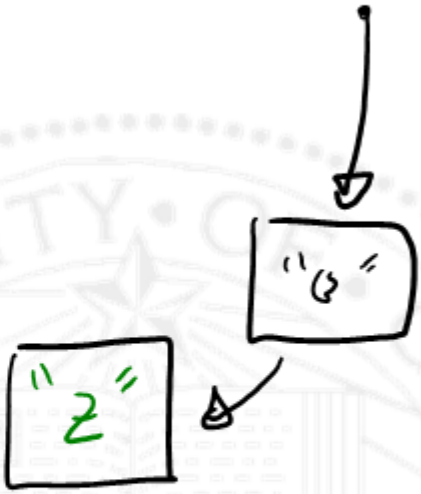
- Doubly Linked Lists
- Allocation
- String Comparison
- Sorting
- Algorithm Analysis

Arrays vs Dynamic Lists

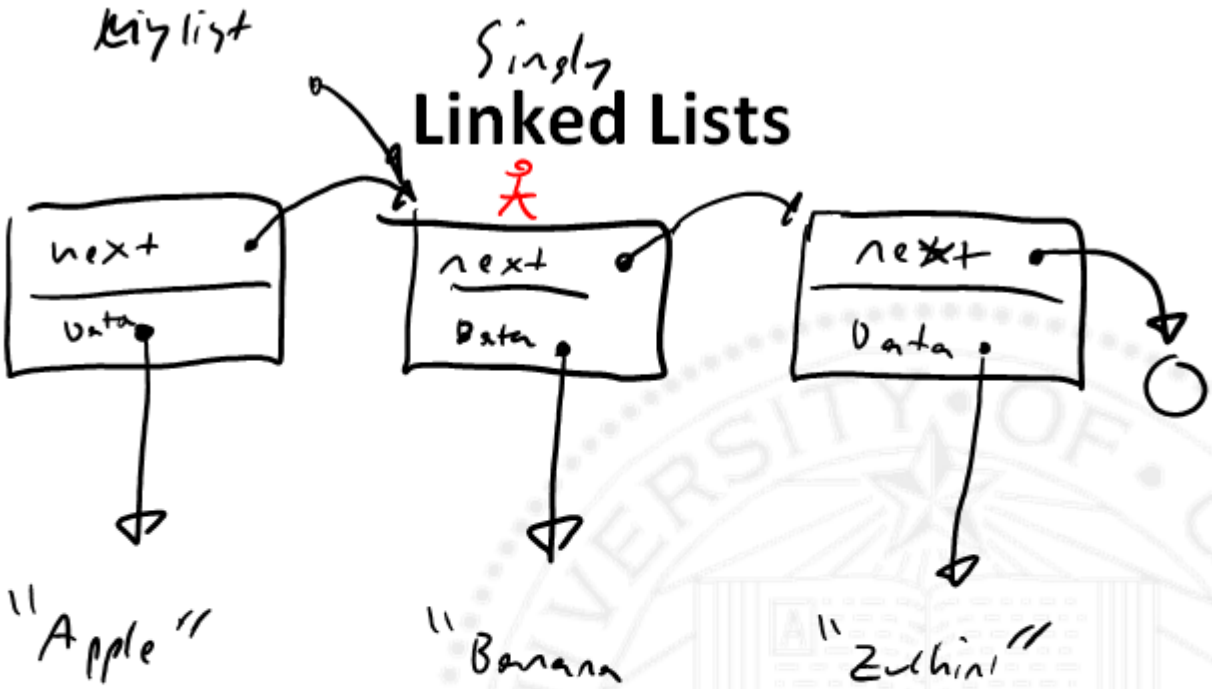


Max Lichtenstein

Example: Smart fridge

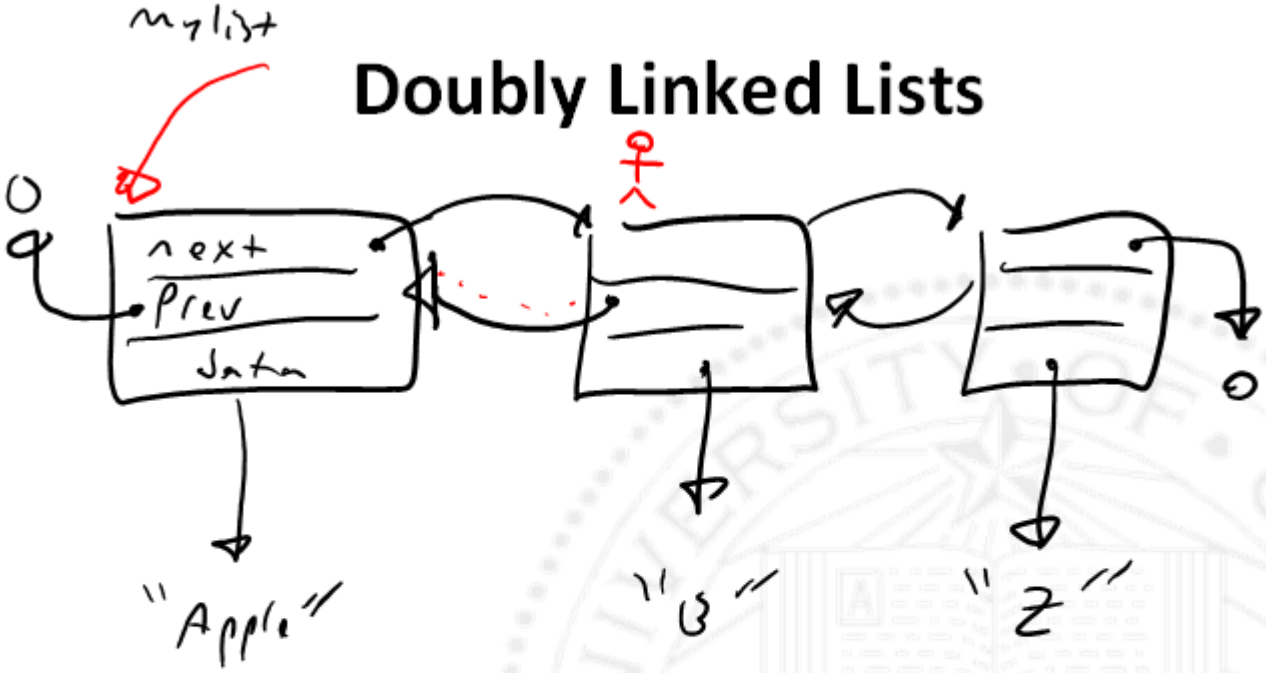


Singly Linked Lists



Max Lichtenstein

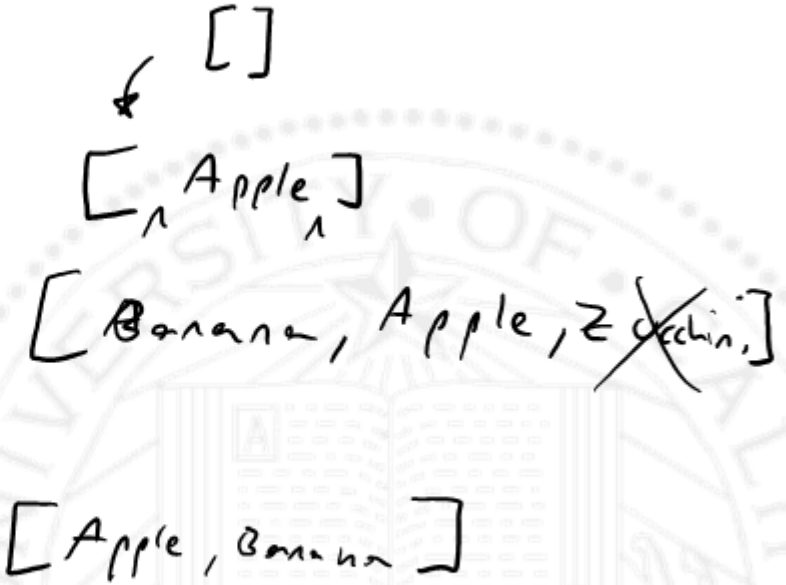
Doubly Linked Lists



Why use a dynamic list?

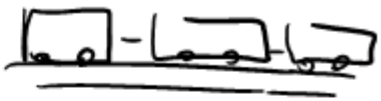
List Operations

- Create
- Insert
- Remove
- Swap



Max Lichtenstein

Lab 5 in C



```
typedef struct ListItem {
    struct ListItem *previousItem;
    struct ListItem *nextItem;
    char *data;
} ListItem;
```

Piazza Poll:

- sizeof(ListItem) = 12 (4 bytes for previous address, 4 bytes for next address, 4 bytes for data address).

- If our heap is initially 36 bytes, and we do the following:

```
ListItem * animal_list = LinkedListNew("Elephant");
animal_list = LinkedListCreateAfter(animal_list, "Moose");
animal_list = LinkedListCreateAfter(animal_list, "Cuttlefish");
```

How much memory is left in our heap?

A) 0 bytes
B) More than 0 bytes
C) Less than 0 bytes

very efficient malloc()

Normal malloc)

still free!

BREAK

Example usage of LL functions

```
ListItem *LinkedListNew(char *data);
```

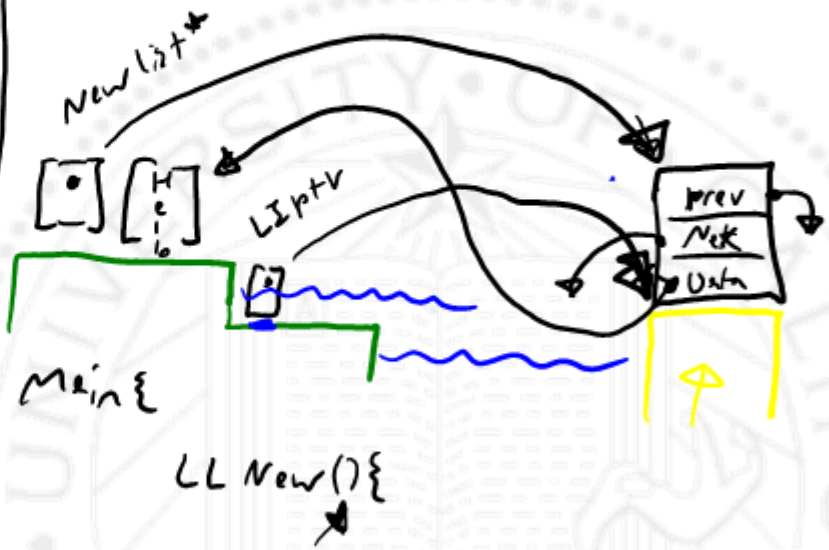
1) Make a ptr

2) Call malloc
   - Capture return value

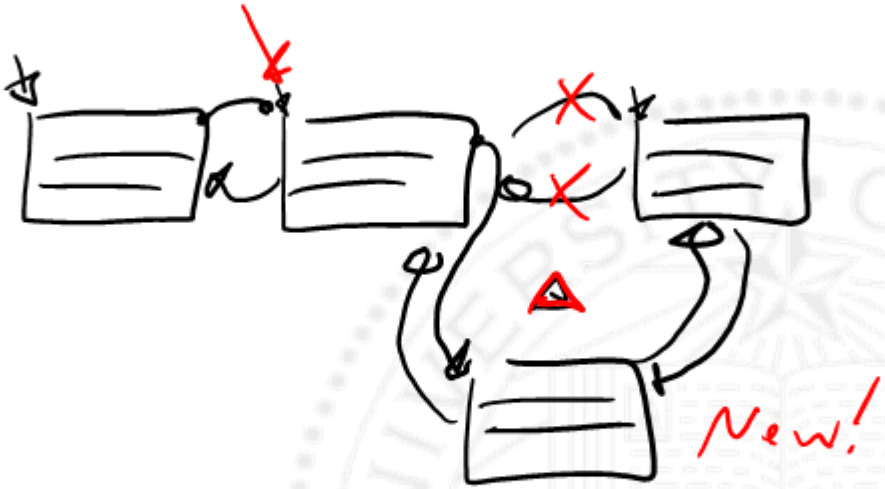
3) Check result of malloc

4?) Modify data in over allocated space

5) return ptr

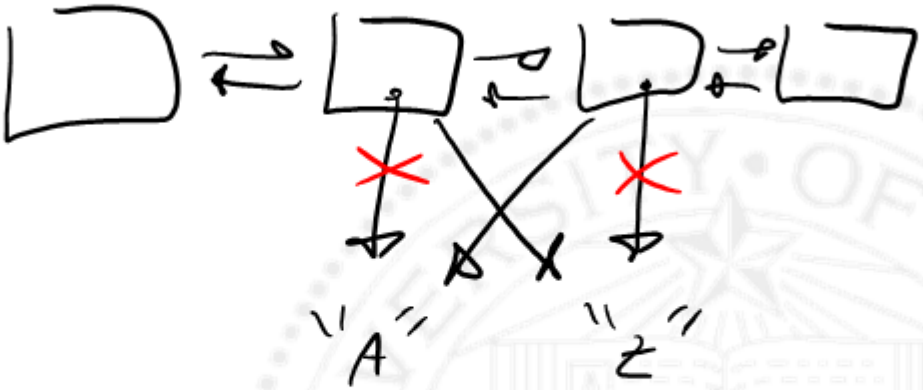


```
ListItem *LinkedListCreateAfter(
        ListItem *item, char *data);
```



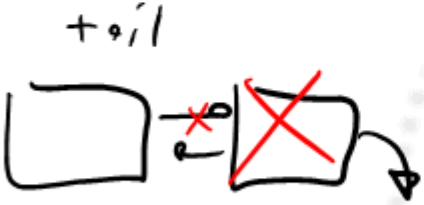
Max Lichtenstein

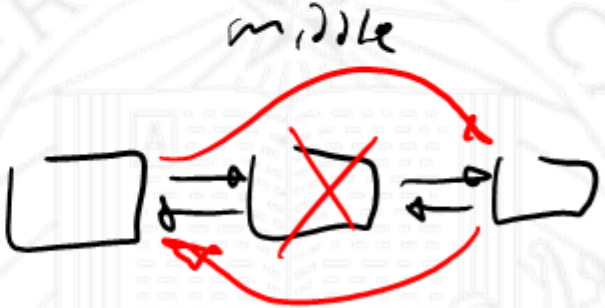
```
int LinkedListSwapData(
        ListItem *firstItem, ListItem *secondItem);
```



Max Lichtenstein

```
char *LinkedListRemove(ListItem *item);
```





li->prev = new_prev;

Max Lichtenstein

```
ListItem *LinkedListGetFirst(ListItem *list);
```

Test Harness Design

Test harness design principles

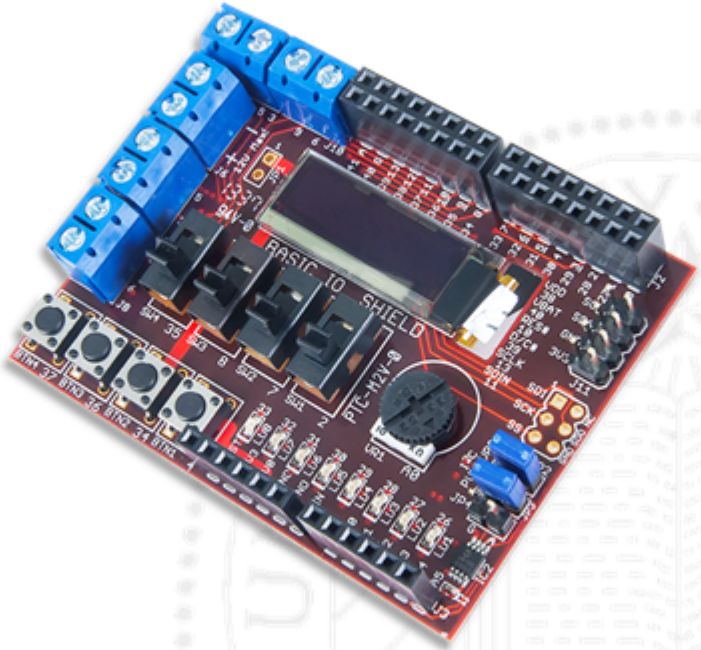
- Tests grow in complexity
- But test *harness* should be simple(ish)
- Add observability
- Make it easy to localize problems
- Test harness should be modular too

Let's do it!

Max Lichtenstein

Embedded code intro

How does code interact with the world?

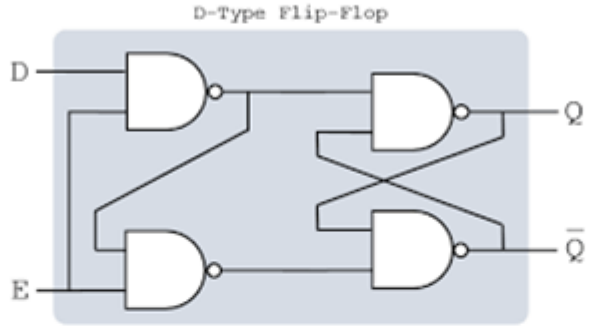


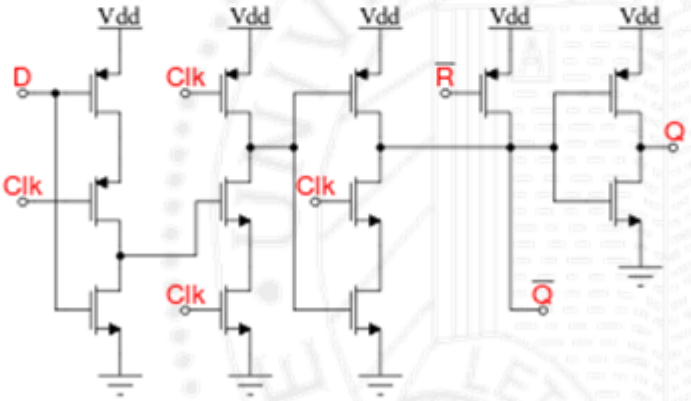
Max Lichtenstein

Example:

```
//If button 1 is on, turn on LED 1:
if((PORTF & 0b0001) != 0){
    LATE = 0b0001;
}
```

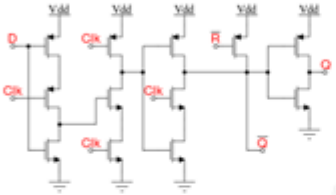
1s and 0s are voltages

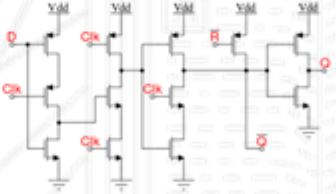




Max Lichtenstein

1s and 0s to the outside world

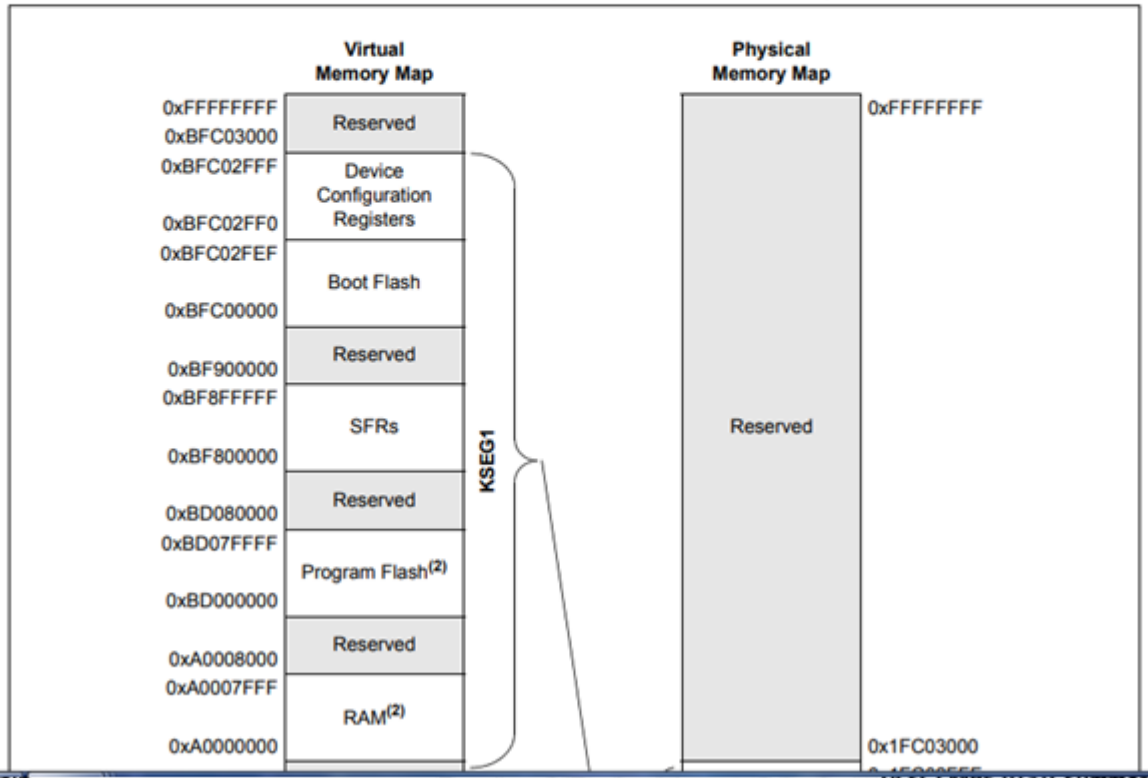




Max Lichtenstein

SFRs:

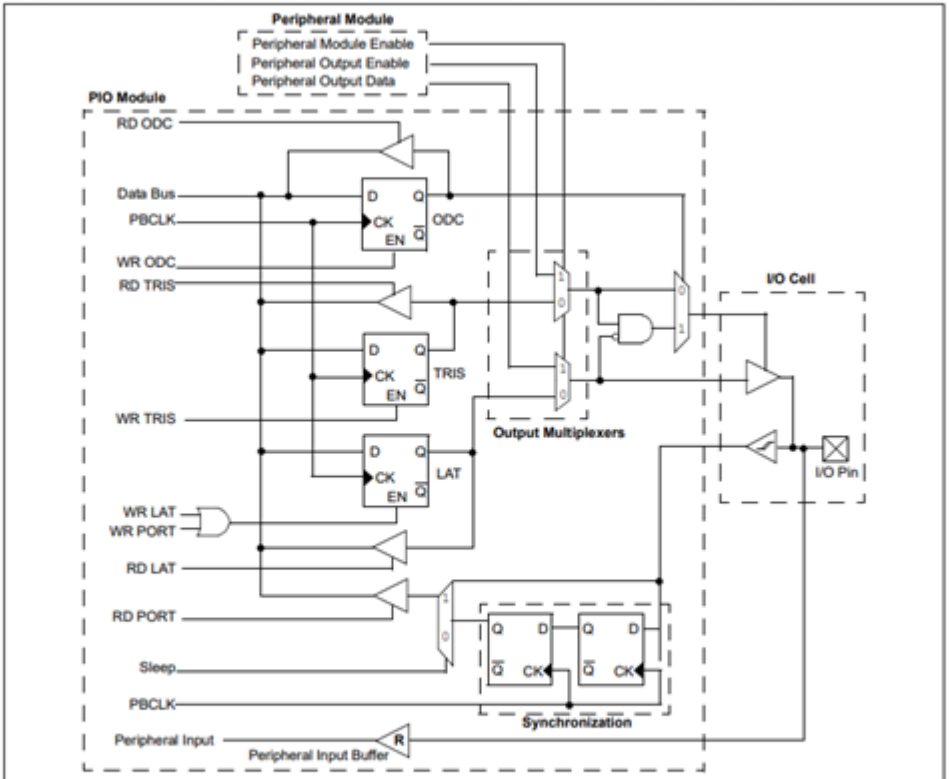
FIGURE 4-6: MEMORY MAP ON RESET FOR PIC32MX340F512H, PIC32MX360F512L, PIC32MX440F512H AND PIC32MX460F512L DEVICES(1)



SFRs:

- Special locations of memory that can interact with the outside world
- Can be accessed via variables in <xc.h>

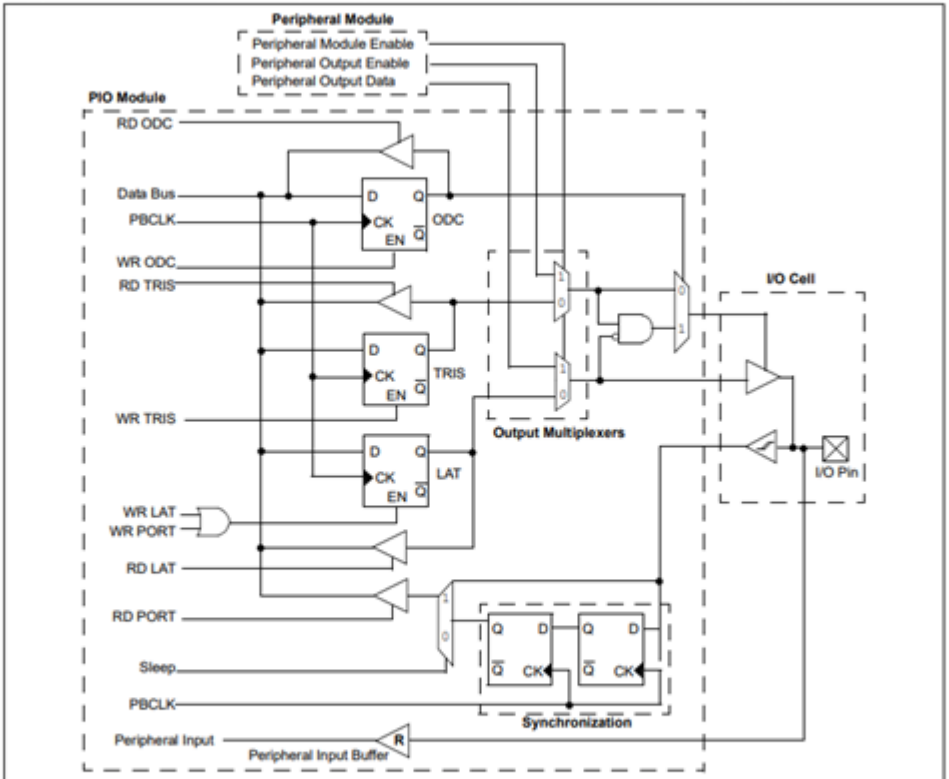
FIGURE 12-1: BLOCK DIAGRAM OF A TYPICAL MULTIPLEXED PORT STRUCTURE



Legend: R = Peripheral input buffer types may vary. Refer to Table 1-1 for peripheral details.

Note: This block diagram is a general representation of a shared port/peripheral structure is only for illustration purposes. The actual structure for any specific port/peripheral combination may be different than it is shown here.

FIGURE 12-1: BLOCK DIAGRAM OF A TYPICAL MULTIPLEXED PORT STRUCTURE



Legend: R = Peripheral input buffer types may vary. Refer to Table 1-1 for peripheral details.

Note: This block diagram is a general representation of a shared port/peripheral structure is only for illustration purposes. The actual structure for any specific port/peripheral combination may be different than it is shown here.