Random atmospheric noise
if(BEG & BE_UP)

UP | | down

0000 or 1111

Button_states(); 4-bits

static int Hist = 0;

Hist = Hist << 4;

Hist 1 = Button_states();
\[ \begin{align*}
1234 & \quad \text{down} \\
1111 & \\
\frac{\text{Hist}^+}{-\text{Hist}^+} & = 1 \\
\frac{-\text{Hist}^+}{-\text{Hist}^+} & = 12 \\
\frac{-\text{Hist}^+}{-\text{Hist}^+} & = 123 \\
\frac{-\text{Hist}^+}{-\text{Hist}^+} & = 1234 \\
= 2345 & \\
\text{if} (\text{Hist}^+ \in \text{BTN1M}) & \quad \text{down} \\
\text{if} (\text{Hist}^+ \in \text{BTN1M}) & \quad \text{up} \\
\end{align*} \]
if (Down pattern exists)
  if (Last event != down event)
    Down event has occurred
Part 2  ADC

OLED Printing

986
99%

1 interrupt
int x = @ADC1 Buf X
Sum / 8;
Sum >= 3;

1023 → 1986

OLED_Clear();

0-7

2.2.2.2 = 8

XXX XXX
Recursion
Recursion

- Solving problems by breaking them into smaller parts
- "divide and conquer"
- Relies on the problem having self-similarity

Example

```c
int Factorial(int n)
{
    if (n <= 1) {
        return 1;
    }
    return n * Factorial(n - 1);
}
```
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack

Factorial term replaced with result

[4] \[ 5! = 5 \times 4! \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

[3] \(4! = 4 \times 3!\)
[4] \(5! = 5 \times 4!\)
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack

- Stack (top)
- Factorial(3)
- Factorial(4)
- Factorial(5)

Partial results pushed on stack
Factorial term replaced with result

\[ 3! = 3 \times 2! \]
\[ 4! = 4 \times 3! \]
\[ 5! = 5 \times 4! \]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

- Partial results pushed on stack
- Factorial term replaced with result

[1] \[2! = 2 \times 1!\]
[2] \[3! = 3 \times 2!\]
[3] \[4! = 4 \times 3!\]
[4] \[5! = 5 \times 4!\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(1)</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</td>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack

1! = 1
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
5! = 5 * 4!

Factorial term replaced with result
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
(based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack  (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0] $1! = 1$</td>
<td>$= 1$</td>
</tr>
<tr>
<td></td>
<td>[1] $2! = 2 \times 1!$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[2] $3! = 3 \times 2!$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3] $4! = 4 \times 3!$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[4] $5! = 5 \times 4!$</td>
<td></td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0] 1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>[1] 2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td></td>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>

Maxwell J. Dunne – Spring 2016
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack  | Factorial term replaced with result

[0] 1! = 1  | = 1
[1] 2! = 2 * 1!  | = 2 * 1 = 2
[2] 3! = 3 * 2!  | = 3 * 2 = 6
[3] 4! = 4 * 3!  |
[4] 5! = 5 * 4!  |
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>[0]</td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[1]</td>
<td>3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>[3]</td>
<td>5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>

Maxwell J Dunne – Spring 2016
CMPE-013/L: “C” Programming
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack

Factorial term replaced with result

[0] 1! = 1 = 1
[1] 2! = 2 * 1! = 2 * 1 = 2
[2] 3! = 3 * 2! = 3 * 2 = 6
[4] 5! = 5 * 4! = 5 * 24 = 120
5! = 5 × 4 × 3 × 2 × 1

Recursion

Summary

• Usable for solving problems that are divided into subproblems
  – Divide and conquer

• Initial conditions must be similar to conditions for any of the subproblems
  – No difference between solving the smaller computation stand-alone versus as part of a larger computation

• Requires well-defined termination condition
Recursion

Caveats

- Problem must have a well-defined termination condition/base case
- Must have enough memory
  - Memory use high from filling the function stack
Recursion

Limitations

- Limited stack space

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
<tr>
<td>Factorial(6)</td>
</tr>
<tr>
<td>Factorial(7)</td>
</tr>
<tr>
<td>Factorial(8)</td>
</tr>
<tr>
<td>Factorial(9)</td>
</tr>
<tr>
<td>Factorial(10)</td>
</tr>
</tbody>
</table>
Recursion

Multiple recursion

- Recursion is not limited to a single function call

Example

```c
int Fibonacci(int n)
{
    if (n <= 1) {
        return 1;
    }
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```
Recursion
Evaluation of Recursive Functions

- Evaluation of 5
  (based on code from previous slide)

\[ F_4 = F_3 + F_2 \]

**Stack** (top)

- Fibonacci(4)

Partial results pushed on stack
Function call replaced with result
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

```
Stack (top)

Fibonacci(3)
Fibonacci(4)
```

```
Partial results pushed on stack
Function call replaced with result

\[
F_3 = F_2 + F_1 \\
F_4 = F_3 + F_2
\]

Maxwell J Dunne – Spring 2016
CMPE-013/L: “C” Programming
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack

F₂ = F₁ + F₀
F₃ = F₂ + F₁
F₄ = F₃ + F₂

Function call replaced with result
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fibonacci(1)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack

Function call replaced with result

\[
\begin{align*}
F_1 &= 1 \\
F_2 &= F_1 + F_0 \\
F_3 &= F_2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Fibonacci(1)  
Fibonacci(2)  
Fibonacci(3)  
Fibonacci(4)  

Partial results pushed on stack
Function call replaced with result

\[
\begin{align*}
F_1 &= 1 \\
F_2 &= F_1 + F_0 \\
F_3 &= F_2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Fibonacci(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack

Function call replaced with result

\[
F_2 = 1 + F_0 \\
F_3 = F_2 + F_1 \\
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
F_0 = 1 \\
F_2 = 1 + F_0 \\
F_3 = F_2 + F_1 \\
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of $5!$
  (based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(0)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

$F_0 = 1$
$F_2 = 1 + F_0$
$F_3 = F_2 + F_1$
$F_4 = F_3 + F_2$
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Fibonacci(2)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
\begin{align*}
F_2 &= 1 + 1 \\
F_3 &= F_2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_3 = 2 + F_1 \]
\[ F_4 = F_3 + F_2 \]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Partial results pushed on stack
Function call replaced with result

F_1 = 1
F_3 = 2 + F_1
F_4 = F_3 + F_2

Fibonacci(1)
Fibonacci(3)
Fibonacci(4)
Recursion

Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Fibonacci(3)
Fibonacci(4)

F₃ = 2 + 1
F₄ = F₃ + F₂

Partial results pushed on stack
Function call replaced with result
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_4 = 3 + F_2 \]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
(based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Fibonacci(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[
F_2 = F_1 + F_0 \\
F_4 = 3 + F_2
\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
F_1 &= 1 \\
F_2 &= F_1 + F_0 \\
F_4 &= 3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_2 = 1 + F_0 \]
\[ F_4 = 3 + F_2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Fibonacci(0)
Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

F_0 = 1
F_2 = 1 + F_0
F_4 = 3 + F_2
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Fibonacci(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[ F_2 = 1 + 1 \]
\[ F_4 = 3 + F_2 \]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

\[ F_4 = 3 + 2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack
Function call replaced with result

\[ F_4 = 5 \]
Recursion
Self-similarity

- A structure that is similar to part of itself
  - Example: fractals
- Computation and data must be self-similar for recursion
- Previous examples only dealt with single integers
- But what about more complicated data?
Recursion

Complex data

• For example, operating on a string
  – How to do that in C?
• Passing a single string through functions is trivial.
• But what about splitting the string up?
Recursion
Complex data example

- Recursive word count
  - Count 1 word per function call

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

Example

```c
int WordCount(char *str) {
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

str

This is the end.
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
str   p
---   ---
This is the end.
1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
This is the end.
```

```
1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

str

This is the end.
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
str  p
This is the end.
    1  1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

---

This is the end.

1 1 1
Recursion

Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 1 1
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 1 1 1
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

![Diagram of recursion example with arrows pointing to 'This' and 'the' indicating the recursive calls. The output is 1 1 1 1.]}
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 1 2
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 3
Recursion
Complex data example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This | is | the | end.
---|---|---|---
4
Binary trees
Binary trees

- ADT where nodes:
  - Have 0, 1, or 2 children
  - Have a single parent
- Generally nodes only know their children
- Generally traversal is top-down
  - From parents to children
Binary trees

- We will only discuss full binary trees
- Size is $2^n - 1$
Binary trees

Traversal

- Binary trees store data at each node
- So the tree must be traversed to access the node that has the data we want
Binary trees
Self-similarity
Binary trees
Self-similarity
Binary trees
Self-similarity
Binary trees

Self-similarity
Binary trees

Serialization

- A linear representation of a tree
Binary trees
Serialization

<table>
<thead>
<tr>
<th>Root</th>
<th>Left subtree</th>
<th>Right subtree</th>
</tr>
</thead>
</table>

[Diagram of a binary tree with arrows pointing downwards to illustrate serialization.]
Binary trees

Serialization

<table>
<thead>
<tr>
<th>Root</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
</table>

Diagram of a binary tree with an arrow indicating the serialization process.
Binary trees
Serialization

- Root node at the 1\textsuperscript{st} element
- Left tree at the 2\textsuperscript{nd} element
- Right tree at 2^{n-1} element
Binary trees
Serialization

A  B  C  D  E  F  G
Binary trees
Serialization

A | B | C | D |
Binary trees

Serialization

A  B  C

A
  B
  C
Binary trees

Serialization

A   B   C   D   

A
B   D
C   

CMPE-013/L: “C” Programming
Binary trees
Serialization

```
A   B   C   D
```

Diagram of a binary tree with nodes A, B, C, and D.
Binary trees

Serialization

A  B  C  D  E  F  G
Binary trees

Serialization

A  B  C  D  E  F

A

B  E

C  D  F  G
Binary trees
Serialization

A  B  C  D  E  F  G
Binary trees
Serialization

| A | B | C | D | E | F | G |

A → B → C → D → E → F → G
Binary trees

Serialization

A  B  C  D  E  F  G
Binary trees
Serialization