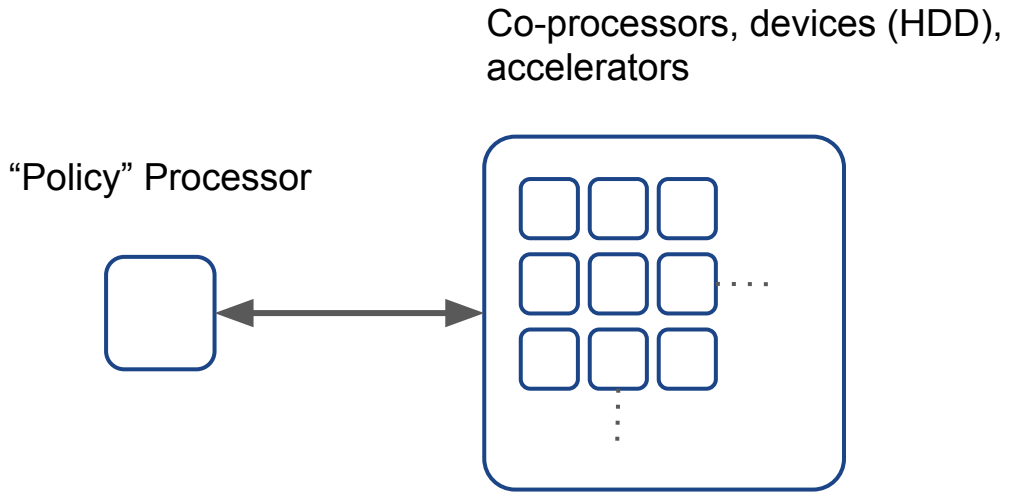
Tasklets

Paul Soulier

Asynchronous, Concurrent Processing

Systems with high “transaction” counts and/or low CPU usage

- Potentially many thousands of “in-flight” operations
- Computation is done on accelerators, co-processors, external devices
- High latency between operations (i.e., operations not CPU bound)



Threads vs Events

Threads

Pros

- Intuitive
- Easy to write, maintain, extend

Cons

- High resource requirements
- Much slower (context switching, cache locality, etc.)

Events

Pros

- Very fast
- Minimal resource requirements

Cons

- More difficult to write
- Non-trivial tasks can become extremely complicated to design/maintain

The Problem With Threads

- Resource constrained systems
  - Threads may not be available
  - Insufficient resources for threads
- Systems with many concurrent operations
  - Potentially many thousands of “in-flight” operations
  - Unreasonable to create threads for each operation

Existing Approaches

- Go (goroutines), Swift (GCD), etc.
- Automated tools (generate code from flow charts, etc.)
- Various C language extensions
  - Protothreads (“Tasklets” are a variant of this concept)
  - nesC and Tiny RTOS
  - Many others...

Illustrative Example

Consider a task composed of the following steps:

1. Allocate buffer resource (may need to wait)
2. Receive data (may only receive max of 256-bytes at a time)
3. Do some computation on data (need to wait)
4. Complete

Thread Version

```
void resume(sem) {
     sem_post(sem);
}

void task(ctx) {
     Semaphore sem;
     uint8_t   *buf;
     int       len;

     get_buffer(resume, &sem, &buf, ctx.data_len);
     sem_take(&sem);

     len = 0;
     while len < data_len {
          rx_data(resume, &sem, &buf[len], min(256, ctx->data_len-len));
          sem_take(&sem);
          len += min(256, data_len-len);
     }

     compute(resume, &sem, buf, ctx->data_len);
     sem_take(&sem);

     task_done();
}
```

Event-Based Version

```
void start_task(context) {
      get_buffer(have_buffer, context, &context->buf, context->data_len);
}

void have_buffer(ctx) {
      ctx->len = 0;
      rx_data(data_received, ctx, ctx->buf, min(256, ctx->data_len));
}

void data_received(ctx) {
      ctx->len += min(256, ctx->data_len - ctx->len));
      if ctx->len < ctx->data_len {
            rx_data(data_received, ctx, &ctx->buf[ctx->len], min(256, ctx->data_len - ctx->len));
      }
      else {
            compute(compute_done, ctx, ctx->buf, ctx->buf_len);
      }

}

void compute_done(ctx) {
      task_done(ctx);
}
```

- Non-reusable
- All state must be kept in some “context” structure
- Error handling, additional states → more difficult

Tasklets

- Uses indirect gotos
  - Hides what are really call/callbacks
  - Mimics thread-like semantics
- Each Tasklet has a stack (separate from the C runtime stack)
  - Stores addresses for indirect goto
  - Stores Tasklet-local variables
- A Tasklet can call other functions that yield/block
  - When a function blocks, the C runtime stack is “unwound”
  - When resumed, the stack is rewound
- Tasklets can allocate Tasklet-local variables with lifetimes that match the Tasklet
- Variables stored on C runtime stack are lost at each “yield” point

Anatomy of a Tasklet

```
void Tasklet::main() {
```

Prelude - always executed, declare `TaskletVars`

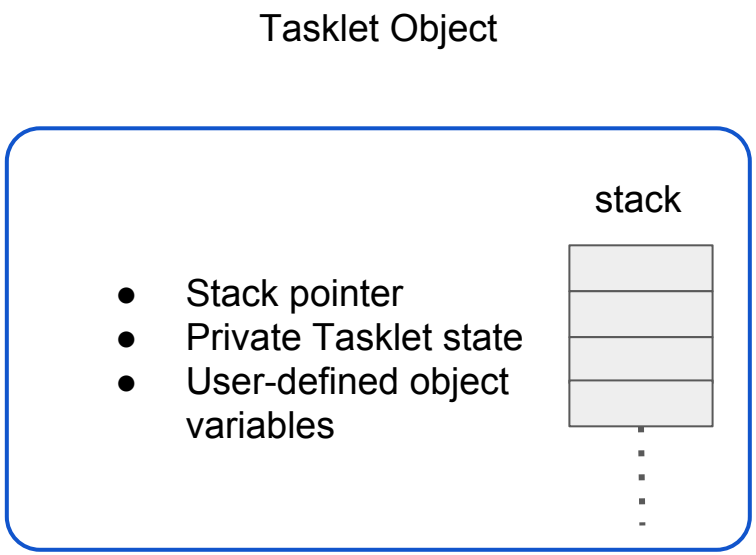
```
TASKLET_BEGIN();
```

Body - executed with thread-like semantics. Use TASKLET_YIELD() and TASKLET_WAIT()

```
TASKLET_END();
```

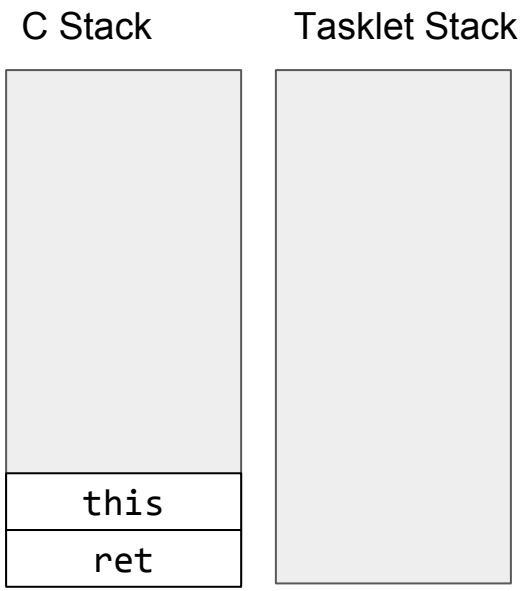
End - executed a single time after execution passes `TASKLET_END()`. Tasklet-local vars destructed.

```
}
```



Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```



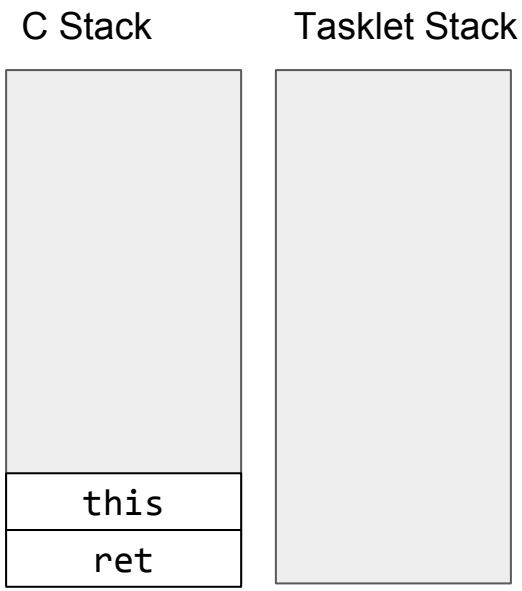
Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:                                  , this->buf, this->data_len);
8:
9:                                  his, this->buf, this->data_len);
10:
11:
12:
13:
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len
21:
22:                                                  &buf[len], min(256, data_len - *len));
23:
24:                                                  );
25:     }
26:
27:     TASKLET_END();
28: }
```

Predefined “resume” function

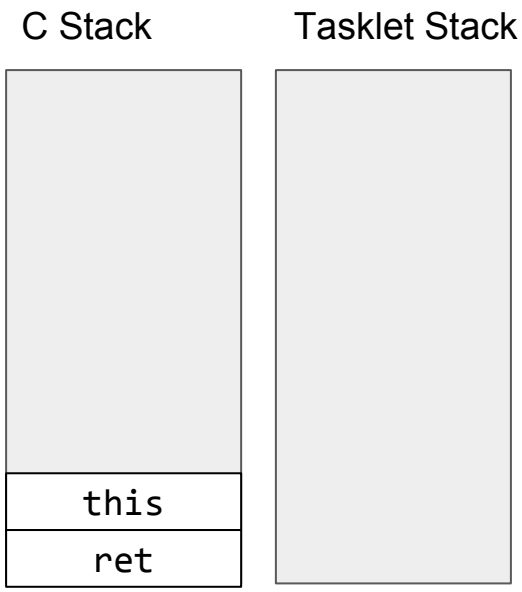
TASKLET_WAIT combines a blocking call and a yield. Necessary for a call that may not actually block.

Use TASKLET_BEGIN_FUNC since tskl_rx isn’t member function.



Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```

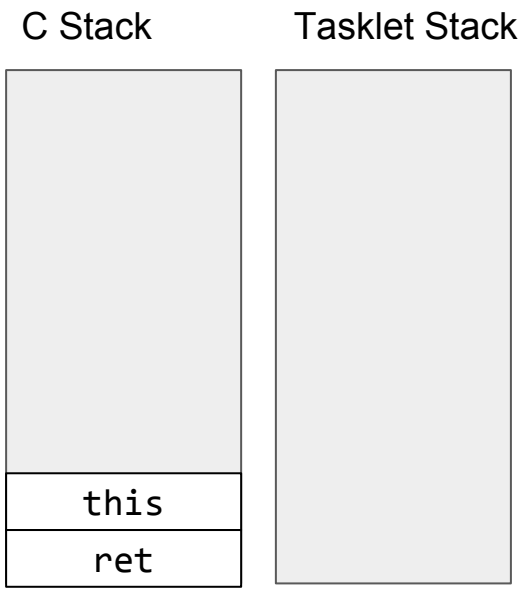


Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```

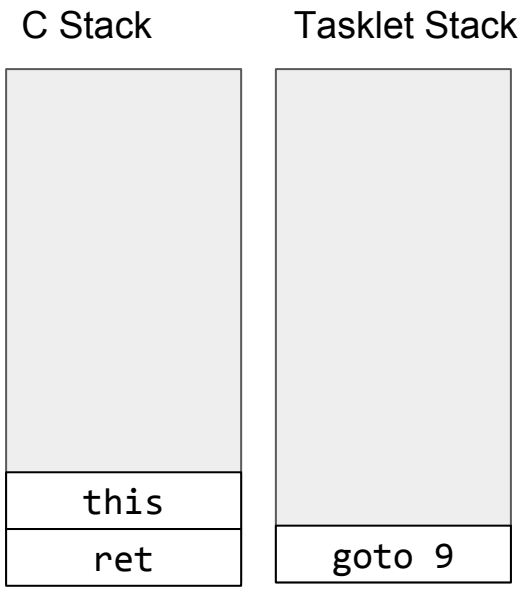
Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```



Tasklet Example

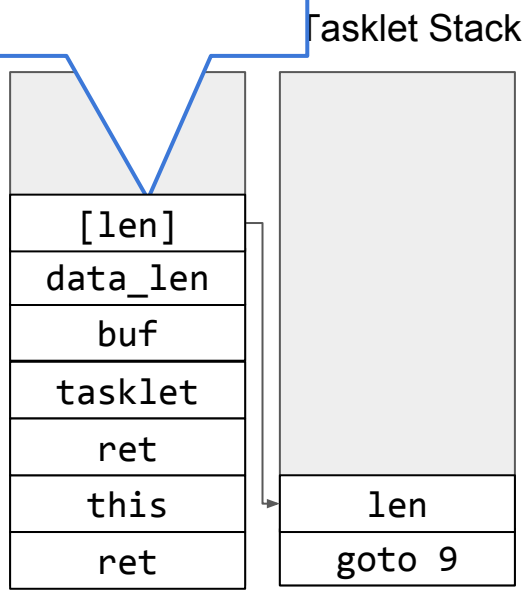
```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```



Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this,
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```

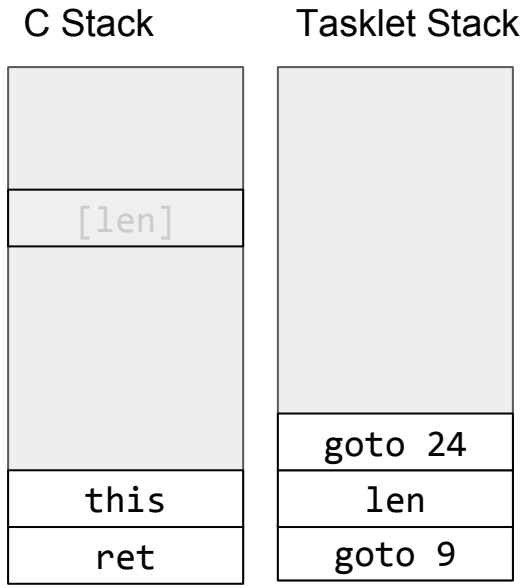
Constructs a “proxy” on the C stack and an `int` on Tasklet stack



Tasklet Example

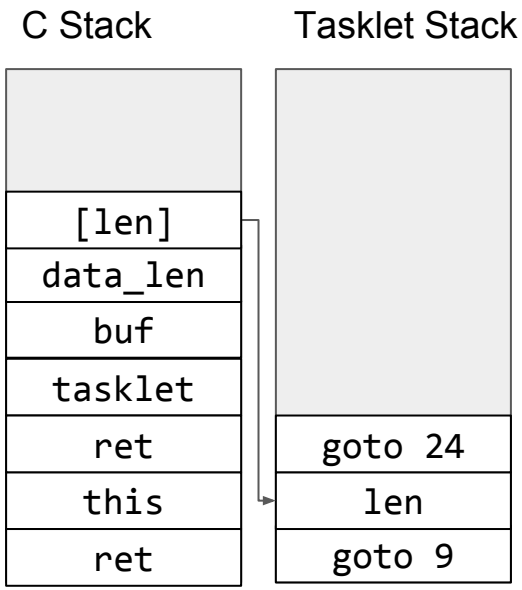
```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, t
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```

Proxy goes away, but actual data remains on Tasklet stack.



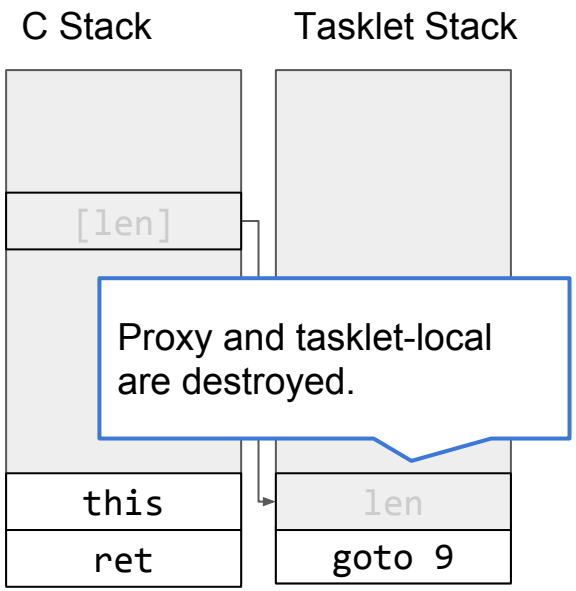
Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```



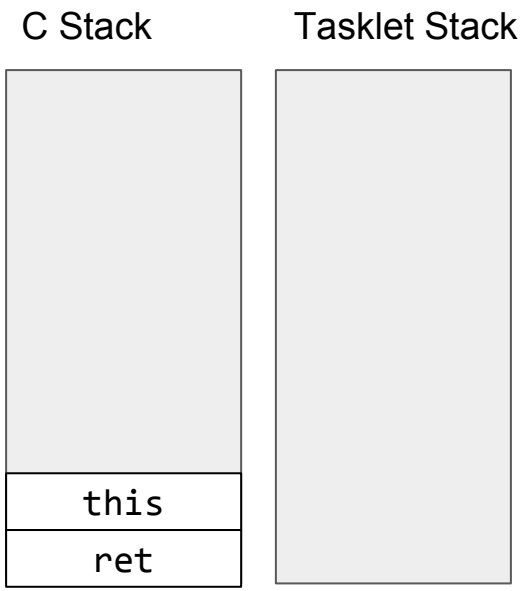
Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```



Tasklet Example

```
1:  Tasklet::main() {
2:      TASKLET_BEGIN();
3:
4:      TASKLET_WAIT(get_buffer, Tasklet_Resume, this, &this->buf, this->data_len);
5:
6:
7:      TASKLET_CALL_FUNC(tskl_rx, this->buf, this->data_len);
8:
9:      compute(Tasklet_Resume, this, this->buf, this->data_len);
10:     TASKLET_YIELD();
11:
12:     TASKLET_END();
13: }
14:
15: void tskl_rx(Tasklet *tasklet, uint8_t *buf, int data_len) {
16:     TaskletVar<int>          len(tasklet);
17:
18:     TASKLET_BEGIN_FUNC(tasklet);
19:
20:     *len = 0;
21:     while *len < data_len {
22:         rx_data(Tasklet_Resume, tasklet, &buf[len], min(256, data_len - *len));
23:         TASKLET_YIELD();
24:         *len += min(256, data_len - *len);
25:     }
26:
27:     TASKLET_END();
28: }
```



Tasklets Summary

- **Useful when:**
  - Underlying system is event-based (call/callback centric)
  - Resource constrained systems
  - Large number of concurrent tasks that aren't CPU-bound
- **Not perfect**
  - No perfect solution without language support
  - Probably ignores a dozen good coding practices - gotos, macro (mis)use, etc.
  - Some "gotchas" that can be tough to figure out (e.g., no local variables)
  - More difficult to debug
  - Not as fast as pure event-based code, not as nice as pure thread-based code