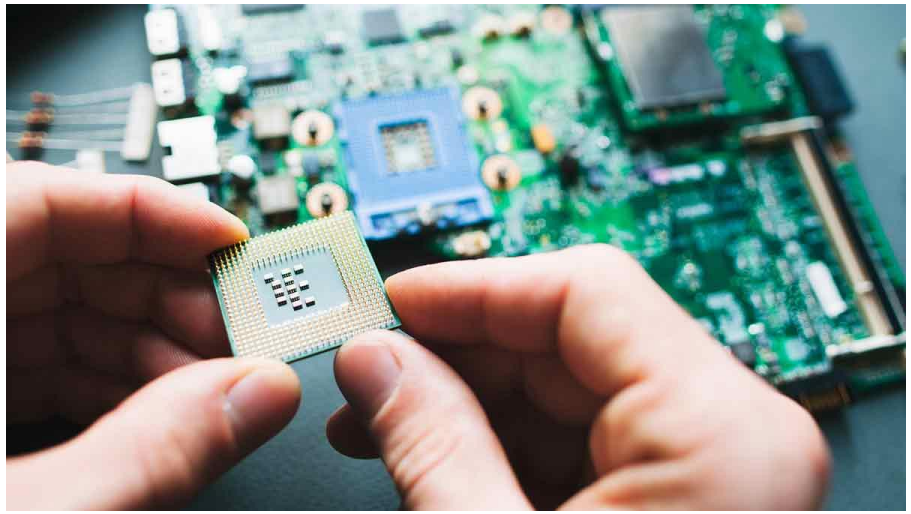

DTU

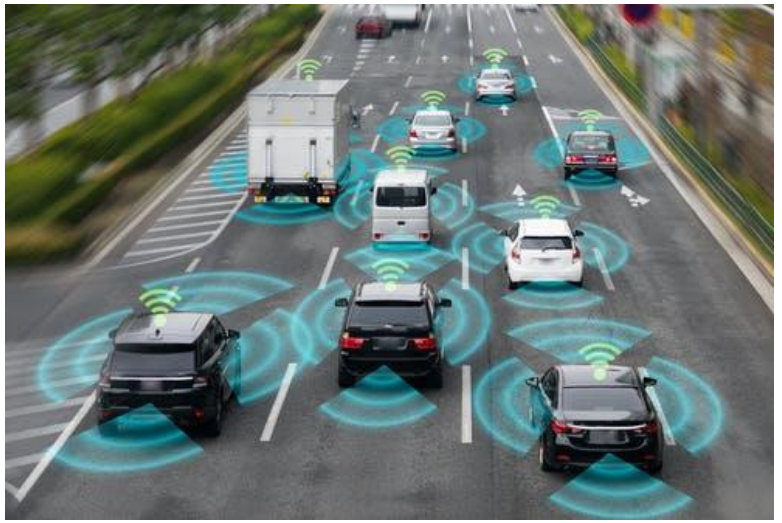
Exploiting the Stack for Key Disassembly and Extraction

You Can’t Hide

Embedded systems are everywhere

- An **embedded system** is a computer system, part of a larger **dedicated** system which it controls (e.g., in **IoT** we have smart TVs, drones, smart lightbulbs, etc.)





- Smart (IoT) devices outnumber humans on the planet
  - Today (2020), there are approximately **30 billion** (30,000,000,000)
  - By 2025 the number is expected to be **75 billion** (75,000,000,000)
  - By 2035 the number will likely transcend a **trillion** (1,000,000,000,000)
- Inherently resource-constrained and lack of protection mechanisms

Cryptographic keys: the foundation of security

- String of data that is used to lock or unlock **cryptographic** functions
- **Example:** to hide (encrypt **Enc**) a message *m* using function *alg* with key *k*:

k = 4E 46 5E 56 6E 66 7E 76 0E 06 1E 16 2E 26 3E 36

m = 54 68 65 43 61 6D 70 20 32 30 32 30 00 00 00 00 (TheCamp 2020 in HEX)

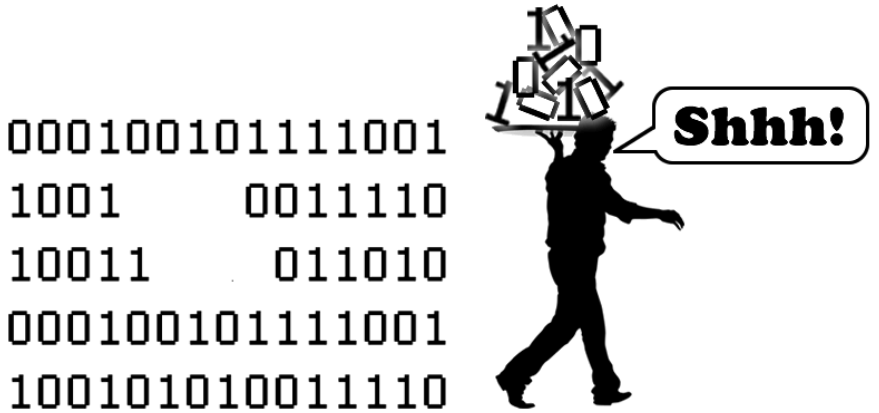
c = $\mathbf{Enc_{alg}}(m, k)$ = 01 96 FC 77 9E 55 74 3C 87 77 F3 8D A8 F0 AF 0F

p = $\mathbf{Dec_{alg}}(c, k)$ = 54 68 65 43 61 6d 70 20 32 30 32 30 00 00 00 00

(revealed using **knowledge** of *k*!)

- Brute-forcing *k* = 4e 46 5e 56 6e 66 7e 76 0e 06 1e 16 2e 26 3e 36 (128-bit) = 2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,455 **possibilities!**
  - **Infeasible** for large key-space
  - **Appealing** on a reduced key-space
- **Hypothesis:** keys used in programs must be stored somewhere in memory (also called the **key-exposure problem**)
- **Idea:** extract the memory and find the key (requires access to the memory)

Memory exfiltration

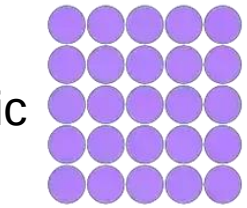


- Physical adversary (evil maid)
  - Cold-boot (remanence effect)
- Remote adversary
  - Heartbleed bug
  - Code injection through software vulnerabilities (e.g., buffer-overflow) and transmission of memory **(enables sophisticated exfiltration)**

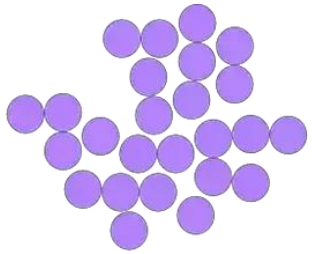
From “Halderman, J. Alex, et al. "Lest we remember: cold-boot attacks on encryption keys." *Communications of the ACM* 52.5 (2009): 91-98.”

Entropy scan

- **Hypothesis:** cryptographic keys are inherently random, thus entropic regions are good candidate keys
- **Approach:** visually inspect the memory to find entropic regions



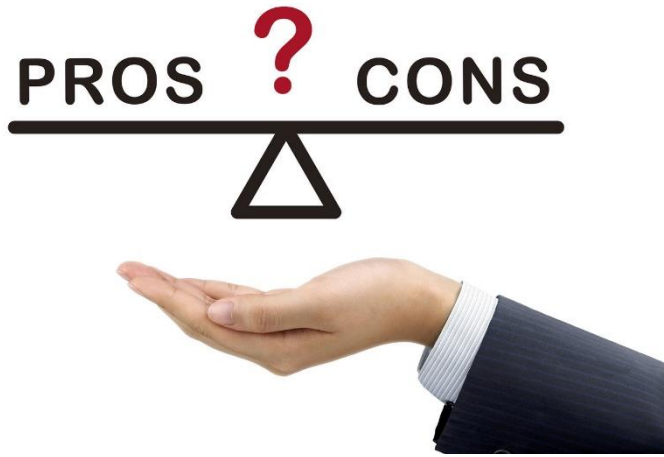
Low Entropy



High Entropy

Entropy scan

- **Pros**
  - Easy and fast
- **Cons**
  - Inaccurate
  - Works only for large keys
  - Only gives potential **area** (insufficient on its own)



Linear scan (sliding window)

- **Approach:**
  – Given a memory image whose size (*M*) is 72 bytes

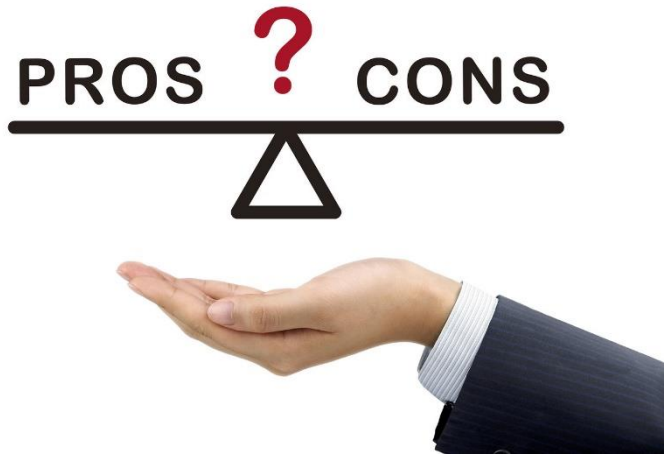
```
cd 62 6f 89 a0 83 86 35 3c c0 c6 a9 cc 88 00 f6 d7 9c e2 0b 05 a8 c3 57
6a 9e dc de 32 11 34 ba 15 19 d5 85 a6 04 66 47 12 32 a8 4e 05 af b1 be
```

  – Assuming the key length (*n*) is 4 bytes
  – Try each *n* chunk as a candidate key in a sliding fashion

- `cd 62 6f 89 a0 83 86 35 3c c0 c6 a9 cc 88 00 f6 d7 9c e2 0b 05 a8 c3 57`
  `6a 9e dc de 32 11 34 ba 15 19 d5 85 a6 04 66 47 12 32 a8 4e 05 af b1 be`
- `cd 62 6f 89 a0 83 86 35 3c c0 c6 a9 cc 88 00 f6 d7 9c e2 0b 05 a8 c3 57`
  `6a 9e dc de 32 11 34 ba 15 19 d5 85 a6 04 66 47 12 32 a8 4e 05 af b1 be`
- `...`
- `cd 62 6f 89 a0 83 86 35 3c c0 c6 a9 cc 88 00 f6 d7 9c e2 0b 05 a8 c3 57`
  `6a 9e dc de 32 11 34 ba 15 19 d5 85 a6 04 66 47 12 32 a8 4e 05 af b1 be`

Linear scan (sliding window)

- **Pros**
  - Exhaustiveness
- **Cons**
  - Linear complexity *M*-*n*+1 (M = size of memory, n = window size)
    - Example: For a small **512 MB** (536,870,912 bytes) memory image and using a windows size of 64 bytes there are approximately **536,870,849 candidate keys**
    - Even worse if we do not know the size of the key!

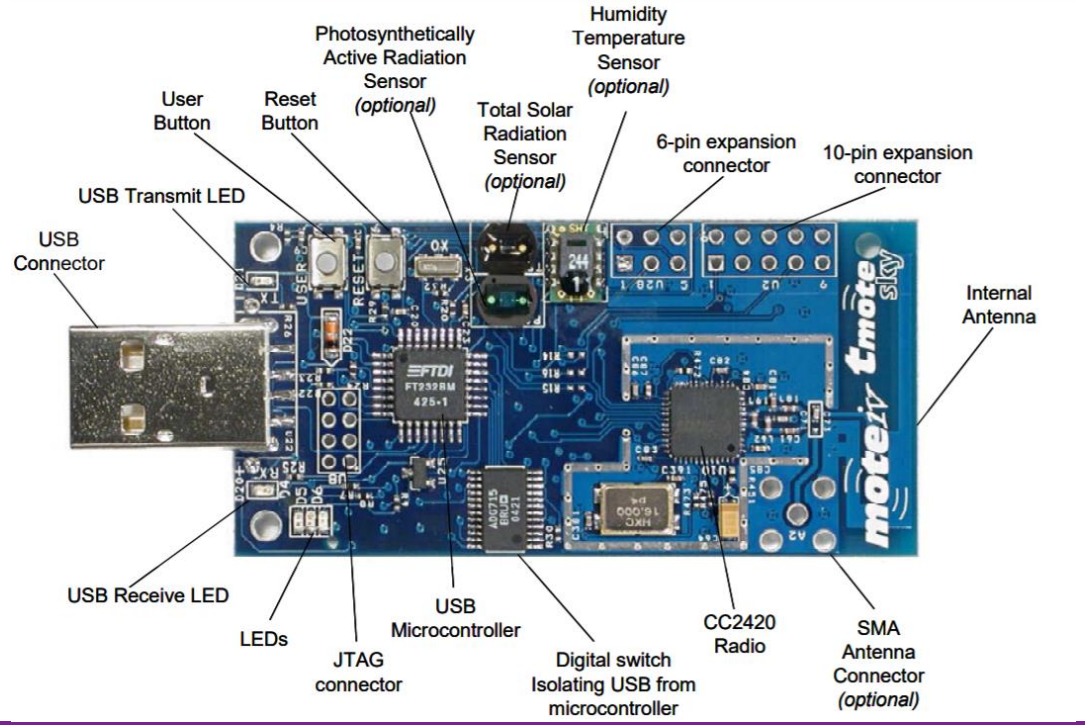


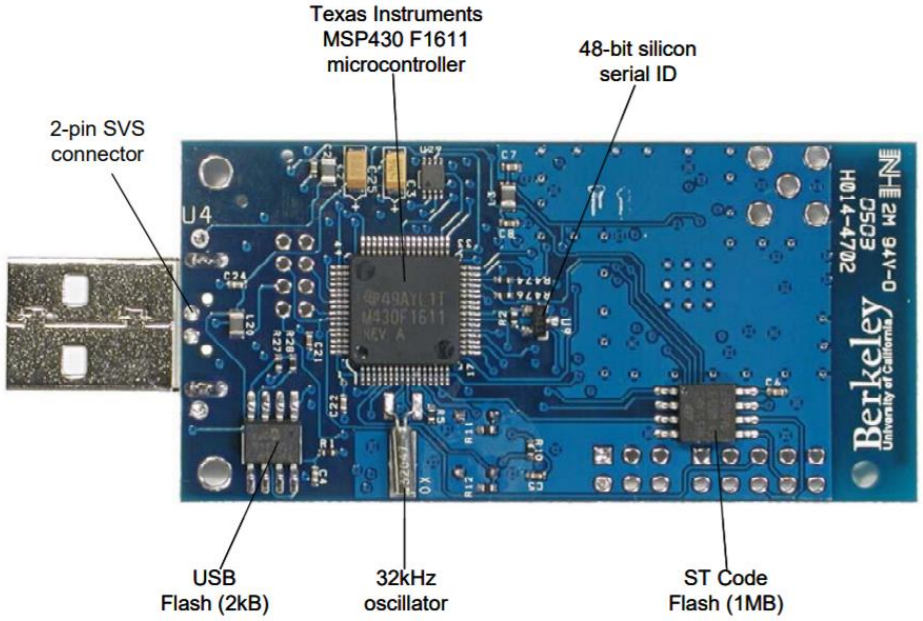
When to extract???

- Timing is an inherent difficulty in key extraction because:
  - Keys should **not** be stored in plaintext when they are not used
  - Keys appear in plaintext **only** when used
  - Plaintext keys **must** be deleted after use
- **Solution:** must approximate:
  - ***When*** the key is used
  - ***Where*** the key is stored while used
  - ***How long*** the key is used

Case Study: The Tmote Sky module

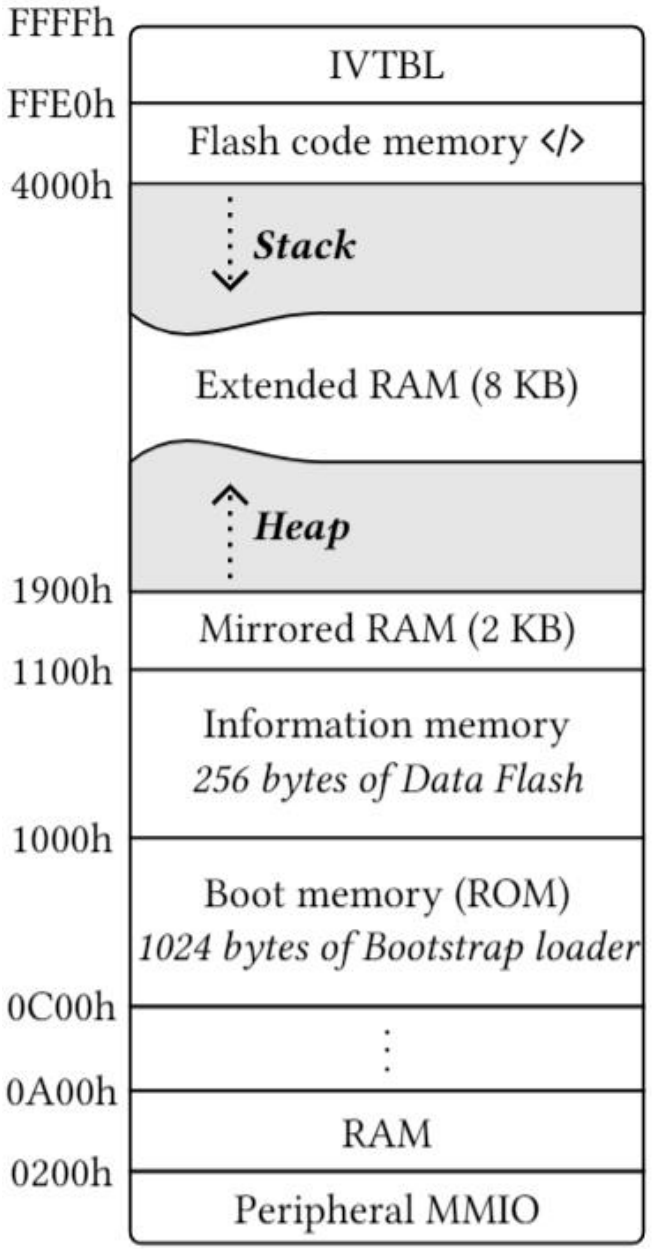
- 250kbps 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver (radio)
- 8MHz Texas Instruments MSP430 microcontroller
- Integrated Humidity, Temperature, and Light sensors
- TinyOS support (programmed using a dialect of C called nesC)





MSP430-F1611 memory architecture

- Von Neumann model (no separation of data and code)
- Event-driven (interrupts)
- Each hardware event triggers specific interrupts that through the IVTBL invoke designated ISRs
- Memory Mapped Input/Output

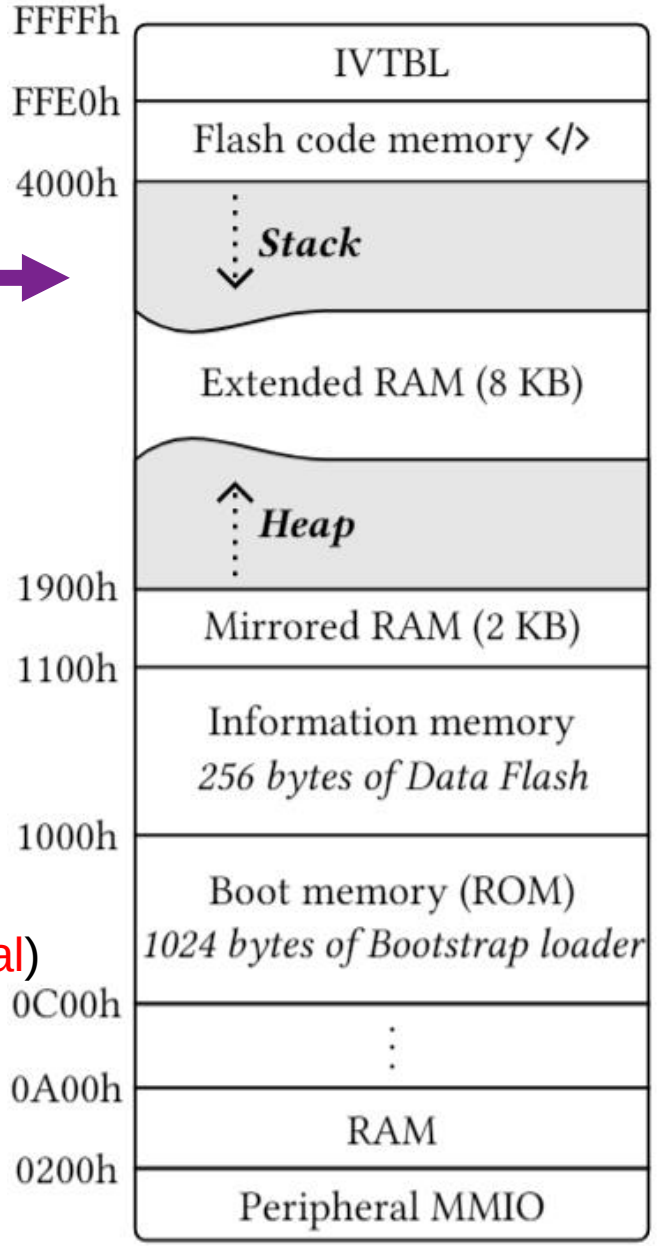


Q1: when to extract the memory?

1. Every *X* seconds
2. Every *X* milliseconds
3. The unit wakes up
4. The unit goes to sleep
5. Receiving a radio packet ⟵ **After**
6. Transmitting a radio packet ⟵ **Before**
7. Measuring humidity
8. Measuring the temperature
9. Measuring the light levels

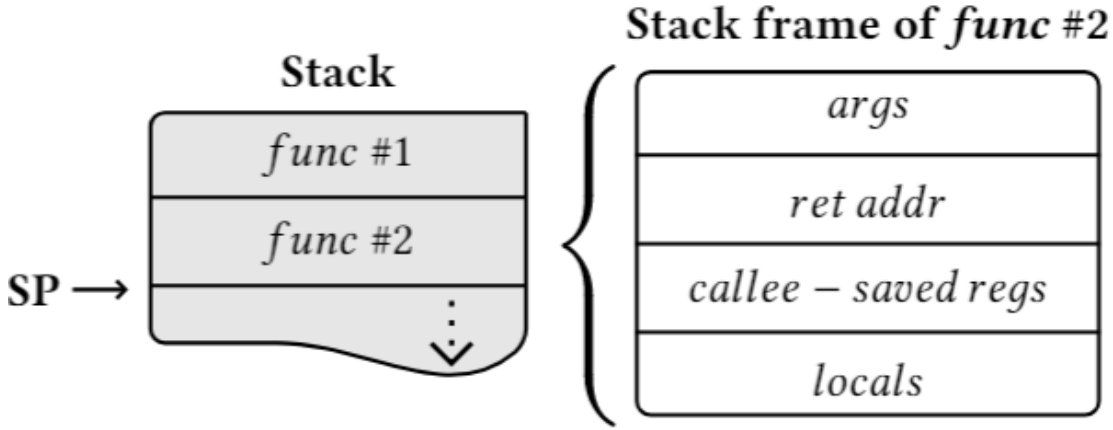
Q2: where is the key stored?

- The **stack** ← TARGET →
  - Function variables and program state
  - When a key is stored (even temporarily) in a variable it will be stored on the stack
- The **heap**
  - Dynamically allocated memory (data) during runtime
  - Unused in TinyOS (no dynamic memory)
- The CPU **registers**
  - Fast calculations and passing arguments (by value or reference)
  - Target CPU has only 12 16-bit general-purpose registers, thus it can **theoretically** fit a 192-bit key using all registers (impractical)
  - Spilled entirely on the stack if not enough registers
- Mirrored RAM
  - Above-function variables (global and static variables)
  - **Not advised** to use persistent memory to store plaintext keys

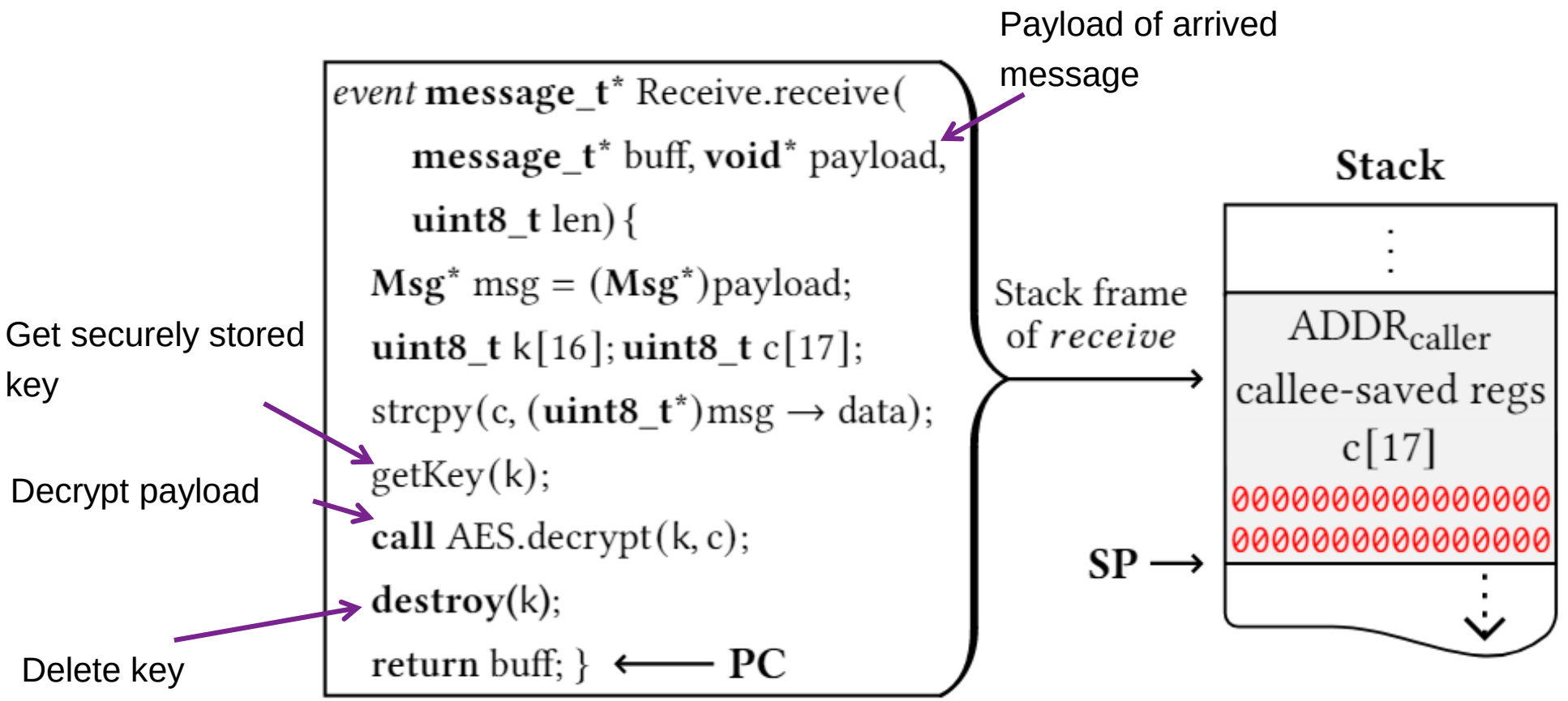


The stack nature is not random

- **Deterministic** and **sequential** data structure that grows and shrinks (up or down)
- Used to keep track of the runtime environment of a program (data and function calls)
  - Each program function is allocated **one stack frame** for its temporaries and exists on the stack exclusively while the function is executing
  - **Determinism:** stack allocation determined by the compiler but reflects the stack frame structure and adheres to system limitations (CPU architecture and memory model)
  - **Sequentialism:** frames are consecutively allocated in the order they are invoked and all bytes in all data objects are stored sequentially
- The CPU uses the Stack Pointer (**SP**) to keep track of the stack



The stack (demonstrative key-exposure at runtime)



- Program Counter (**PC**) points to the *next* machine instruction to be executed by the CPU

Q3: how long is the key stored?

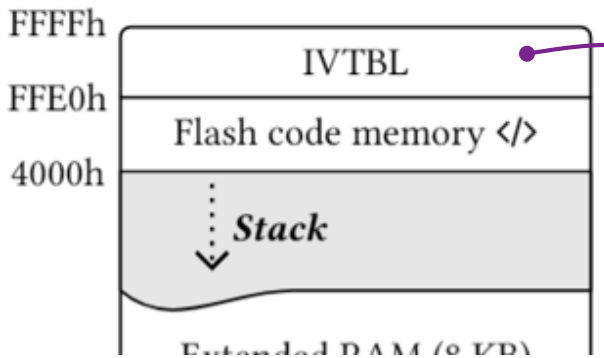
- Testing long **Enc** and **Dec** take to operate on a single block
  - Example: AES-128-ECB @4MHz (max CPU clock speed on the Tmote Sky)

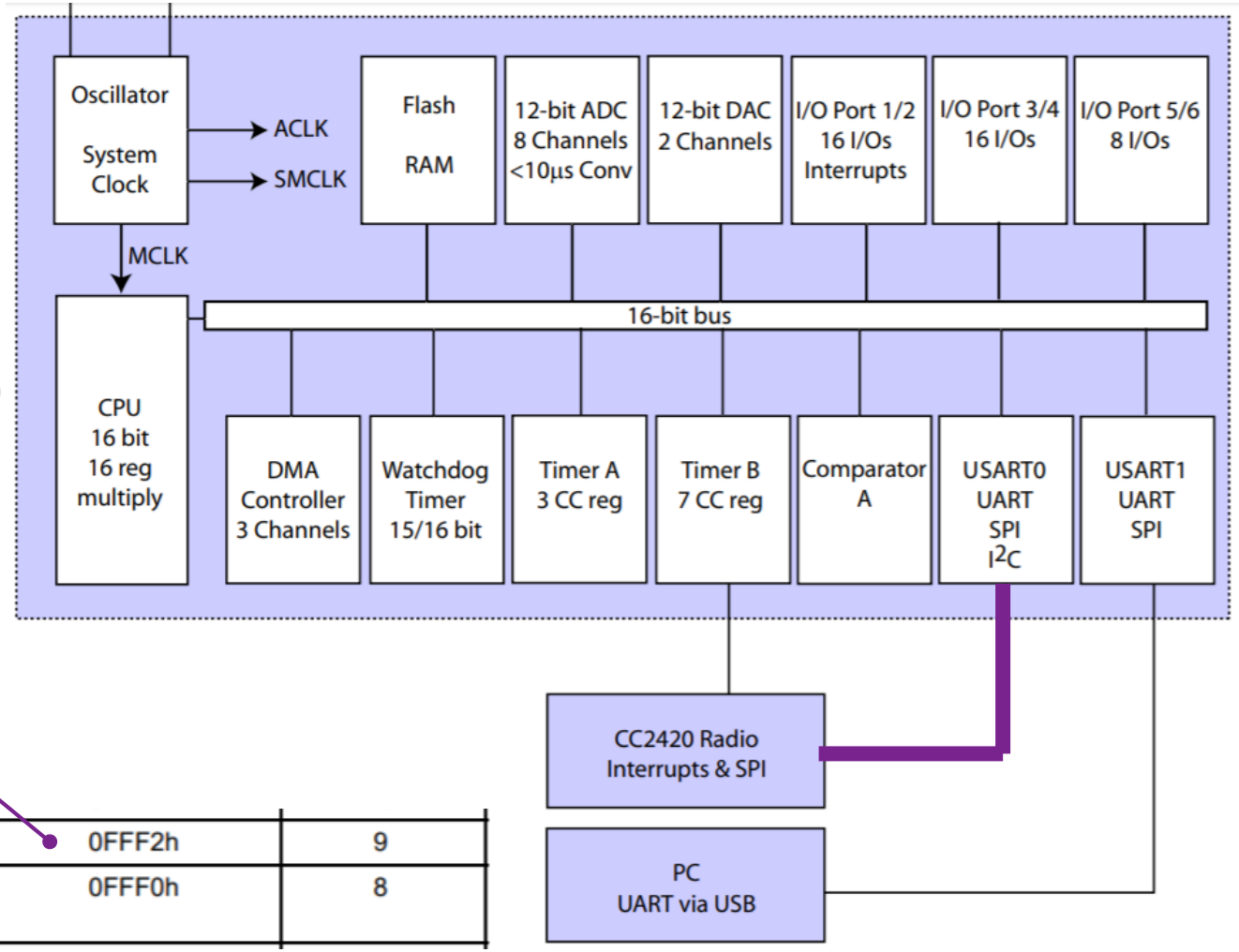
Imp.	Dir	O0	O1	O2	O3	Os
TinyAES	**Enc** (ms)	10.716	3.034	2.497	2.141	2.626
	Dec (ms)	69.045	5.737	4.540	3.940	5.511
TI AES	**Enc** (ms)	8.568	2.903	2.726	2.371	2.595
	Dec (ms)	13.013	4.283	3.503	3.297	4.241

- During **Dec** the key is visible for **at least** 3.5 ms (4.25 ms using default optimization level)

Practical timing (dissecting the reception event)

- Radio connected to USART0
- Will cause interrupts!
- **Reception** event triggers ISR whose address is stored at 0FFF2h
  - Reads packet from the radio then calls the TinyOS function **Receive.receive(..) -- WIN! (just latch onto)**



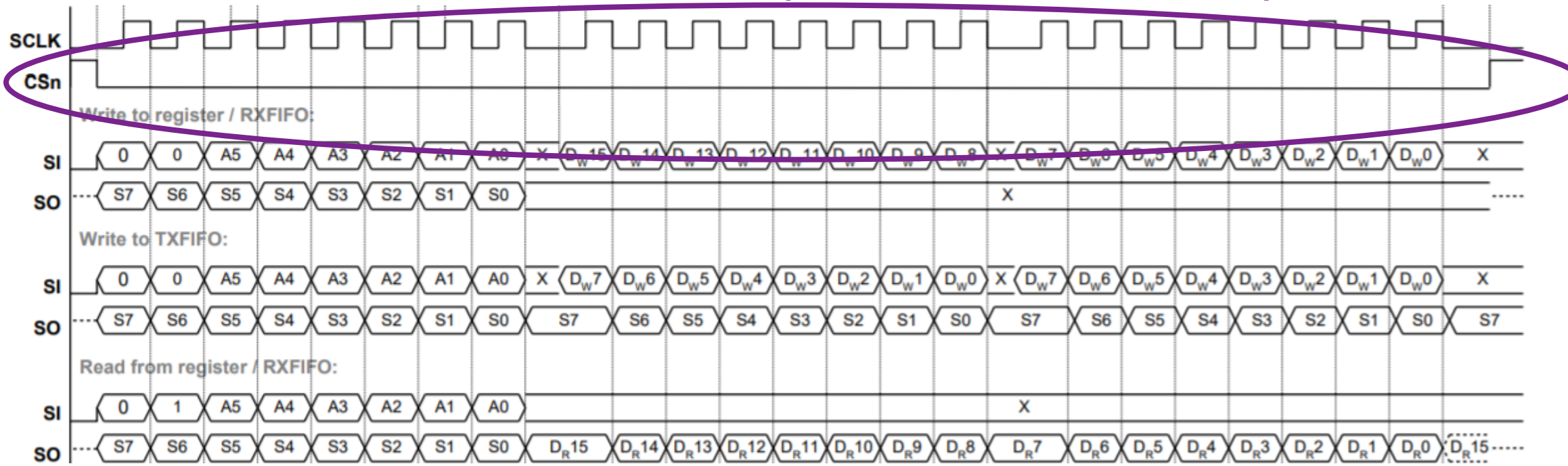


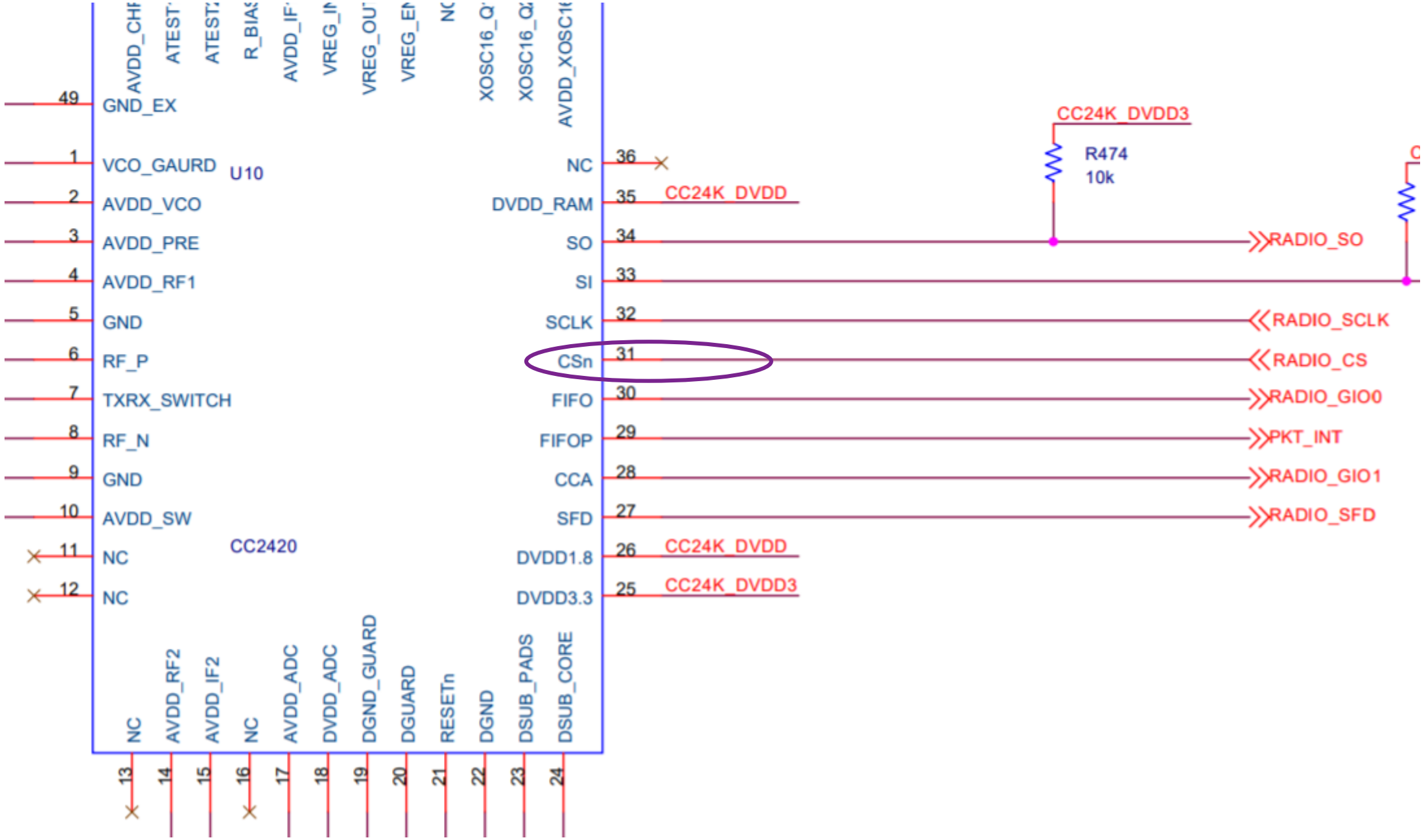
USART0 receive	URXIFG0	Maskable	0FFF2h	9
USART0 transmit I²C transmit/receive/others	UTXIFG0 I2CIFG (see Note 4)	Maskable	0FFF0h	8

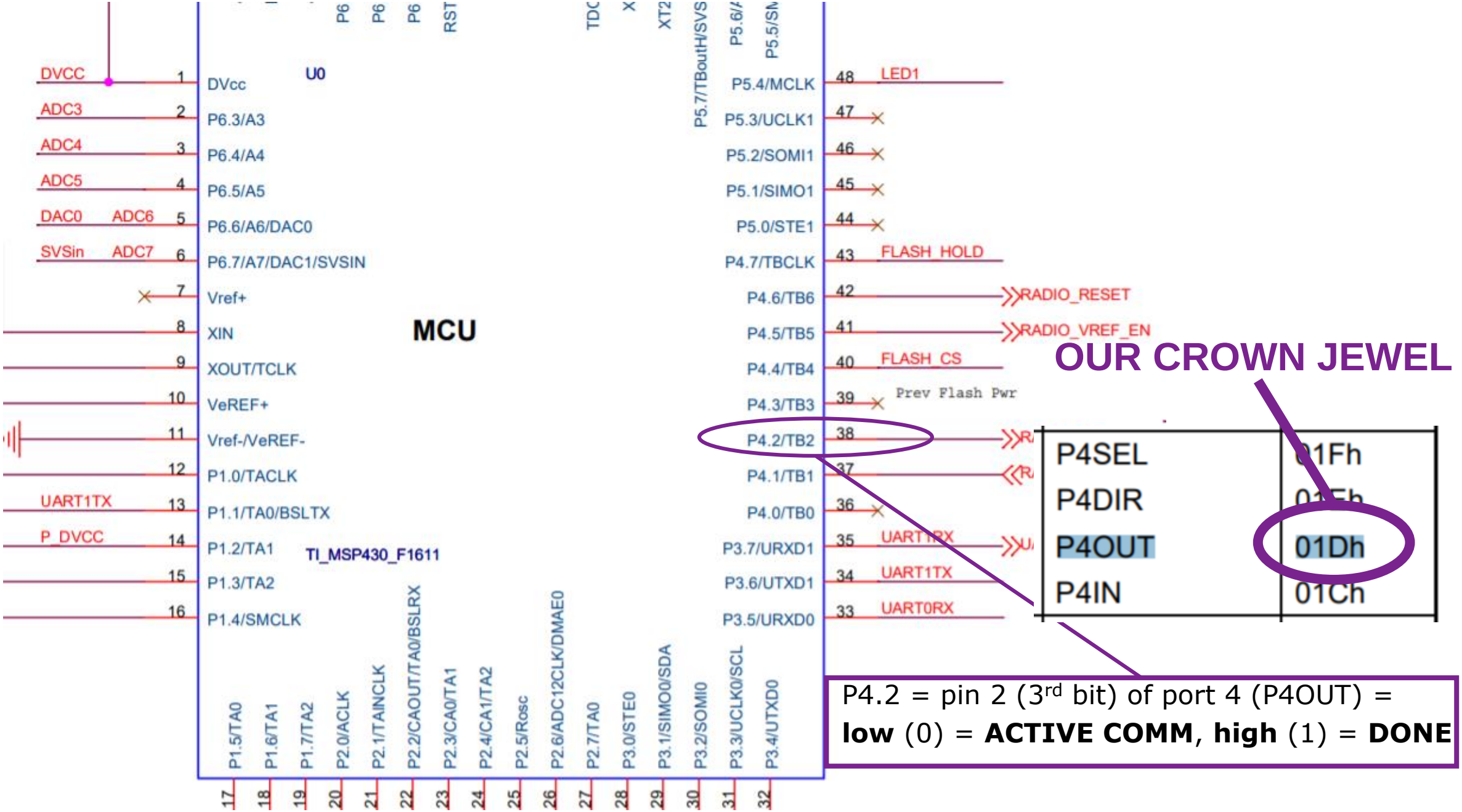
We can do better - narrowing even further

- But only one byte is read at a time from the Radio – when is the reception **actually** done?
- The Chip Select (CSn) **must remain** low (low voltage) while communicating with the Radio (read or write)
  - Thus, reading is done when it **gets high** and the invocation of the TinyOS reception handler is therefore imminent! **Lets use the power of MMIO to monitor this pin!**

Reading and writing to the Radio




U10
CC2420
GND_EX
VCO_GAURD
AVDD_VCO
AVDD_PRE
AVDD_RF1
GND
RF_P
TXRX_SWITCH
RF_N
GND
AVDD_SW
NC
NC
NC
DVDD_RAM
SO
SI
SCLK
CSn
FIFO
FIFOP
CCA
SFD
DVDD1.8
DVDD3.3
NC
AVDD_RF2
AVDD_IF2
NC
AVDD_ADC
DVDD_ADC
DGND_GUARD
DGUARD
RESETn
DGND
DSUB_PADS
DSUB_CORE
CC24K_DVDD3
R474
10k
CC24K_DVDD
CC24K_DVDD
CC24K_DVDD3
RADIO_SO
RADIO_SCLK
RADIO_CS
RADIO_GIO0
PKT_INT
RADIO_GIO1
RADIO_SFD



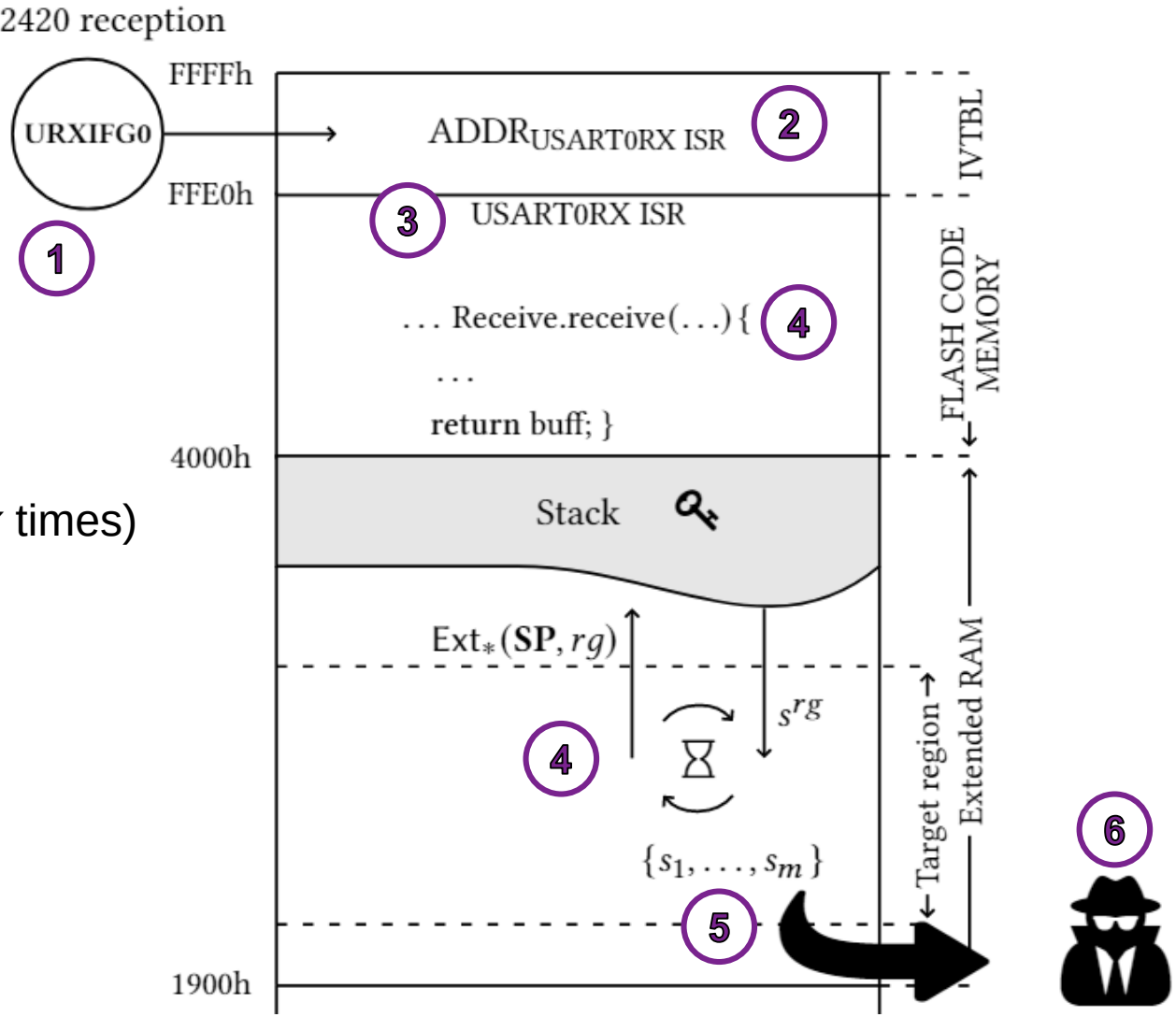
OUR CROWN JEWEL

P4SEL	01Fh
P4DIR	01Eh
P4OUT	01Dh
P4IN	01Ch

P4.2 = pin 2 (3rd bit) of port 4 (P4OUT) =
low (0) = **ACTIVE COMM**, **high** (1) = **DONE**

Putting it all together

1. Radio receives a packet
2. Triggers USART0 RX interrupt
3. USART0 RX ISR reads packet and we start observing 001Dh
4. When all bytes are read (&001Dh = #0100h) the reception handler is called and we start a timer to extract *rg* bytes above SP from the stack *s* every 1ms (*x* times) **(key visible for at least 3.5 ms)**
5. Transmit the accumulated data
6. Use the stack nature to derive a **reduced key-space** in which we search for the key



Key localization leveraging the stack nature

- **Hypothesis:** the key **must be sequentially stored on the stack** and will remain intact throughout the cryptographic process **but** values surrounding the key can fluctuate
- **Idea:** given multiple stack sequences of the same execution, then the key will be illuminated by its fluctuating surroundings, thus **revealing itself to us**

```
7e62 0000 0000 0000 0000 0000 0000 0000 7e4a 9649 9a12 944d 0000
1700 e063 4e46 5e56 6e66 7e76 0e06 1e16 2e26 3e36 a812 9a12 034d
7664 4e46 5e56 6e66 7e76 0e06 1e16 2e26 3e36 a812 9a12 024d 1400
```

Common subsequence

```
4e46 5e56 6e66 7e76 0e06 1e16 2e26 3e36 a812 9a12
```

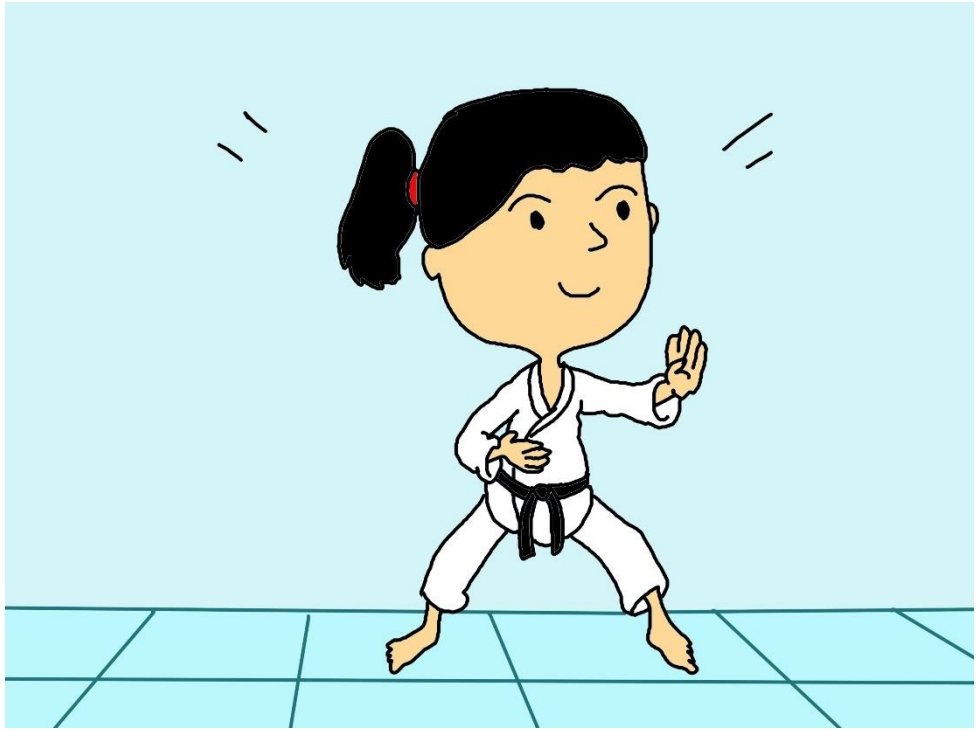
- The common subsequences are our **reduced search-space**
  - Search-space of 10 words (20 bytes) is easy to brute-force using sliding window
  - Even with **unknown** key size there are only $\sum_{k=1}^{10} 10 - k + 1 = 55$ possibilities!
  - **At least** less than 2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,455

Trying it out!

Questions?

Exercise (15 minutes)

- **Question:** how to prevent the attack?



Dependencies of Key Extraction and Identification

- Successful **extraction** depends on:
  - Timing
    - Key lifespan (but easy to reduce the timer intervals)
    - **Deterministic use** of the key
    - Key size (ability to capture it with our range *rg*)
- **Identification** depends on:
  - Stack representation
  - **Fluctuations**
  - That the key is indeed a **common** subsequence (must occur in several sequences)