

Data Structures (8A)

Copyright (c) 2014 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

Computer Organization and Design ARM Edition:
The Hardware Software Interface
By David A. Patterson, John L. Hennessy

<https://thinkingeek.com/arm-assembler-raspberry-pi/>

Heap

1. **dynamically allocated** by the program
when it creates the data structure
2. used by the program to store information
3. **dynamically released** by the program
when the structure is no longer needed

Heap manager

```
pt = malloc(size);    // returns a pointer to a block of size bytes  
free(pt);            // deallocates the block at pt
```

simple version to be considered

```
pt = Heap_Allocate(); // returns a pointer to a block of fixed size bytes  
Heap_Release(pt);    // deallocates the block at pt
```

Heap

```
#define SIZE 4
#define NUM 5
#define NULL 0 // empty pointer
```

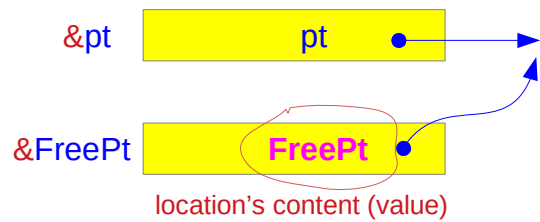
```
int32_t *FreePt;
int32_t Heap[SIZE*NUM];
```

SIZE	EQU	4
NUM	EQU	5
NULL	EQU	0
FreePT	SPACE	4
Heap	SPACE	SIZE*NUM*4

Pointer and array declarations – C and ARM assembly

C code

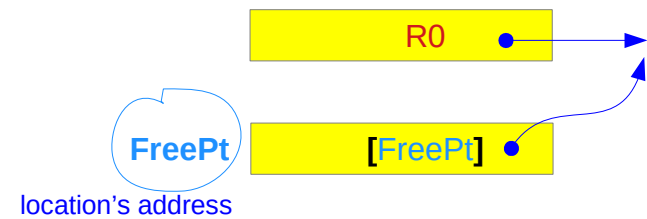
```
int32_t * pt;  
int32_t * FreePt;  
int32_t Heap[SIZE*NUM];
```



Heap	Heap[0]
Heap+1	Heap[1]
Heap+2	Heap[2]
Heap+3	Heap[3]
Heap+4	Heap[4]
Heap+5	Heap[5]

ARM assembly code

```
FreePt SPACE 4  
Heap SPACE SIZE*NUM*4
```



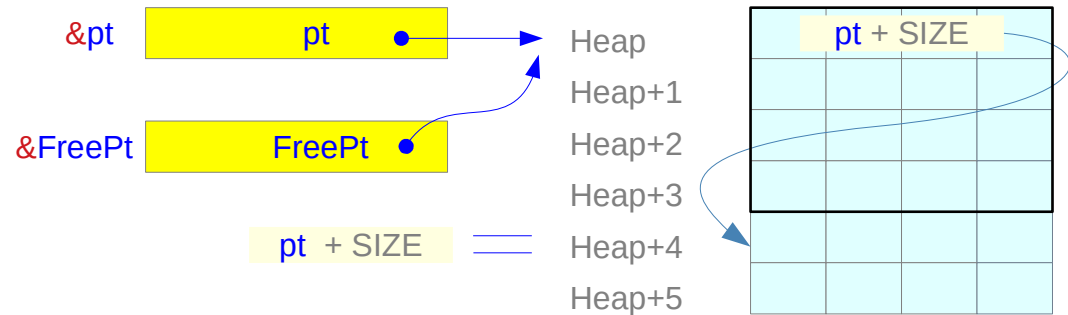
Heap	
Heap+1	
Heap+2	
Heap+3	
Heap+4	
Heap+5	

Heap_Init (1)

```
void Heap_Init(void) {  
    int    i  
    int32_t * pt;  
  
    pt = FreePt = &Heap[0];  
  
    for (i=1; i<NUM; i++) {  
        *pt = (int32_t) (pt + SIZE);  
        pt = pt + SIZE;  
    }  
  
    * (int32_t *) pt = NULL;  
}
```

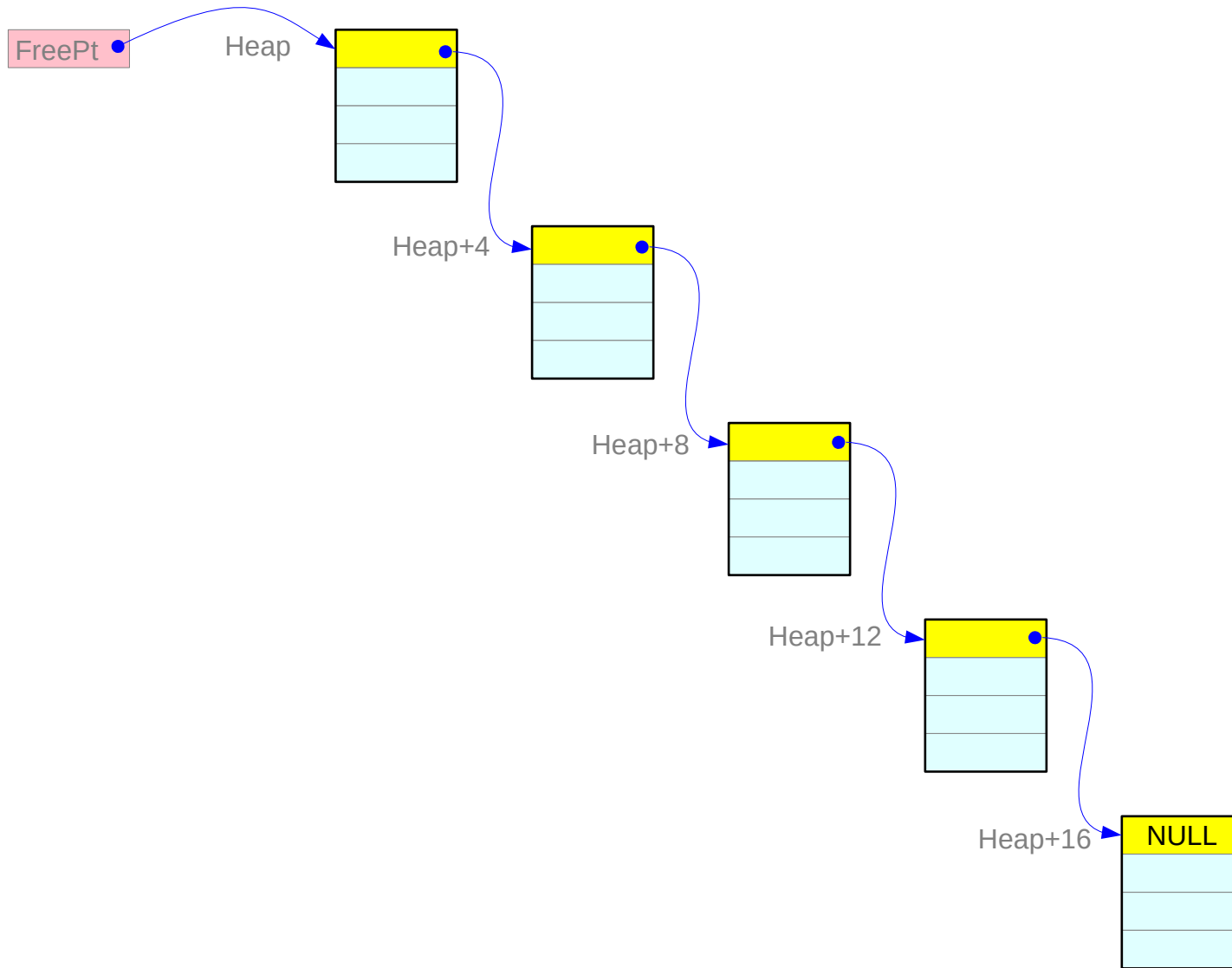
```
#define    SIZE 4  
#define    NUM 5  
#define    NULL 0    // empty pointer
```

```
int32_t    *FreePt;  
int32_t    Heap[SIZE*NUM];
```



```
int32_t * pt;  
int32_t * FreePt;
```


Heap_Init (3)



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Heap_Init (1)

```
Heap_Init    LDR    R0, =Heap           ; pt           ; R0 ← &Heap
             LDR    R1, =FreePt        ;             ; R1 ← FreePt
             STR    R0, [R1]          ; *FreePT = Heap ; [FreePt] ← Heap
             MOV    R2, #SIZE         ;             ; R2 ← #4
             MOV    R3, #NUM-1        ;             ; R3 ← #3
imLoop       ADD    R1, R0, R2, LSL #2 ; pt + SIZE    ; R1 ← R0 + R2*4
             STR    R1, [R0]          ; *pt = pt + SIZE ; [R0] ← R0 + R2*4
             MOV    R0, R1            ; pt = pt + SIZE ; R0 ← R1
             SUBS   R3, R3, #1        ;             ; R3 ← R3 - 1
             BNE    imLoop
             MOV    R1, #NULL         ; R1 ← #0
             STR    R1, [R0]          ; last ptr is NULL ; [R0] ← NULL
             BX     LR
```

Heap_Init (2)

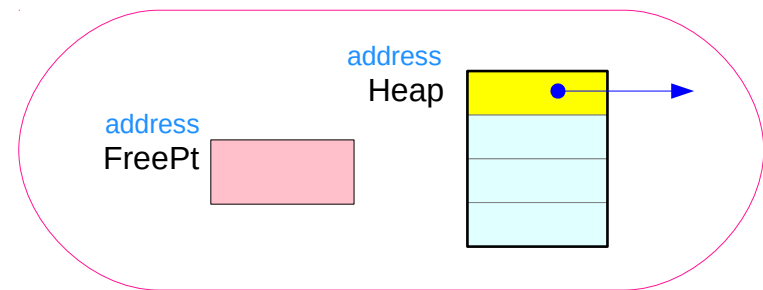
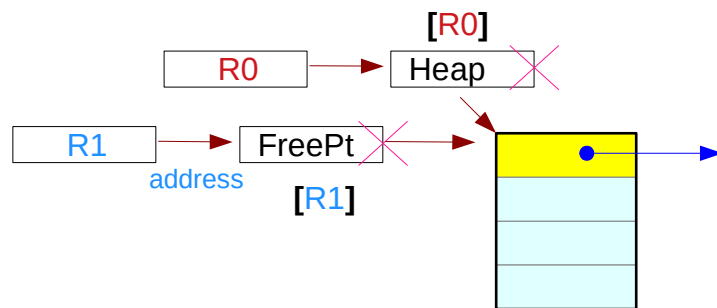
```

Heap_Init    LDR    R0, =Heap          ; pt          ; R0 ← Heap
             LDR    R1, =FreePt       ;             ; R1 ← FreePt
             STR    R0, [R1]          ; *FreePt = Heap ; [FreePt] ← Heap
    
```

~~LDR R0, =Heap ; R0 ← &Heap~~
~~LDR R1, =FreePt ; R1 ← &FreePt~~

~~[R0] ≡ Heap~~
~~[R1] ≡ FreePt~~

▼ Heap, FreePt : all address labels of memory locations

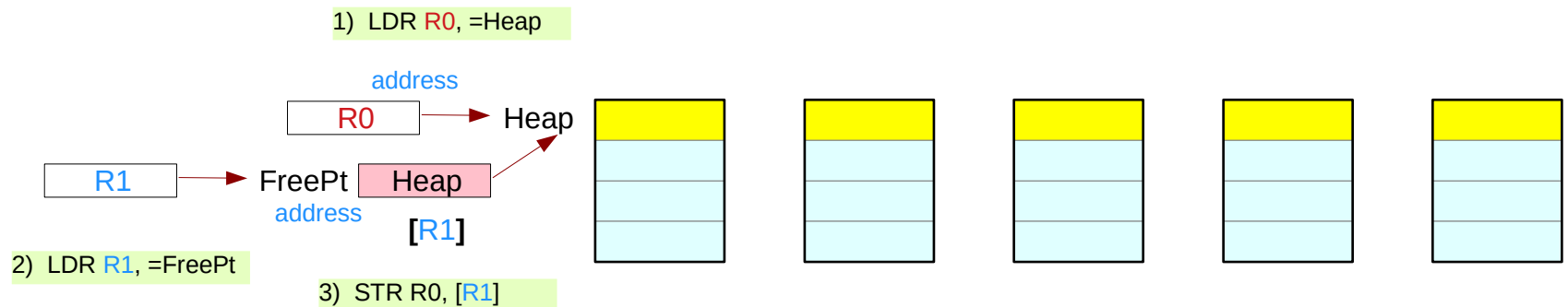


Heap_Init (3)

```
Heap_Init    LDR    R0, =Heap      ; pt          ; R0 ← Heap
             LDR    R1, =FreePt   ;             ; R1 ← FreePt
             STR    R0, [R1]      ; *FreePt = Heap ; [FreePt] ← Heap
```

```
LDR R0, =Heap      ; R0 ← Heap
LDR R1, =FreePt    ; R1 ← FreePt
```

```
R0 ≡ Heap
R1 ≡ FreePt
address
```

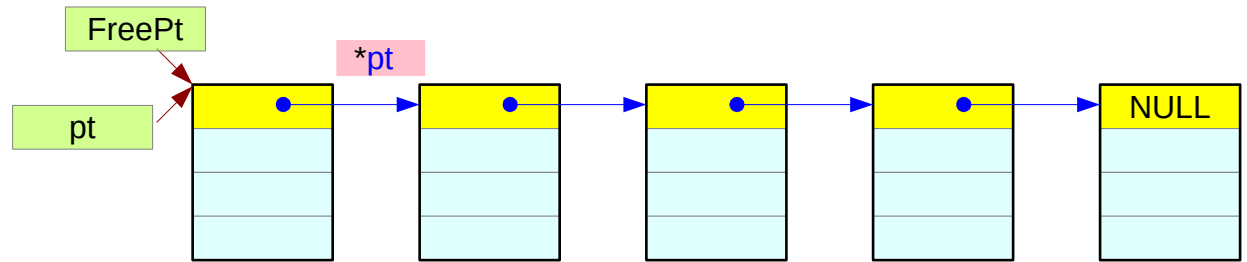


Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

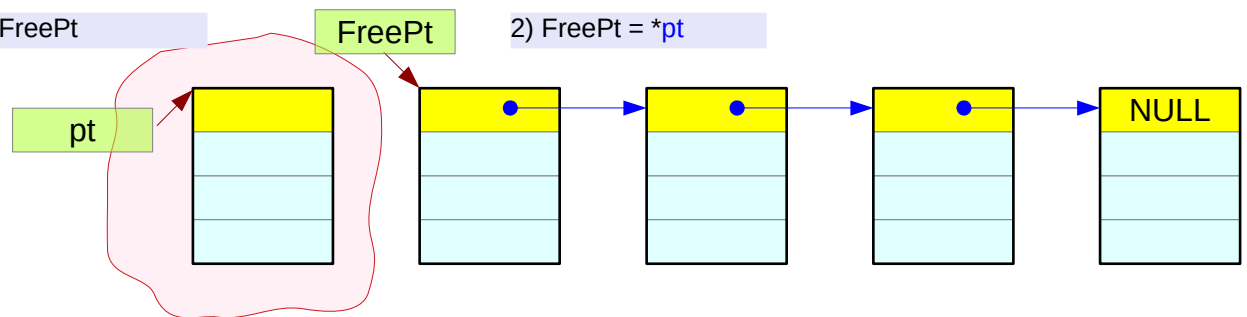
Heap_Allocate

```
int32_t *Heap_Allocate(void) {  
    int32_t * pt;
```

```
    pt = FreePt;  
    if (pt != NULL) {  
        FreePt = (int32_t *) *pt;  
    }  
    return(pt);  
}
```



1) pt = FreePt



2) FreePt = *pt

```
int32_t *FreePt;  
int32_t Heap[SIZE*NUM];
```

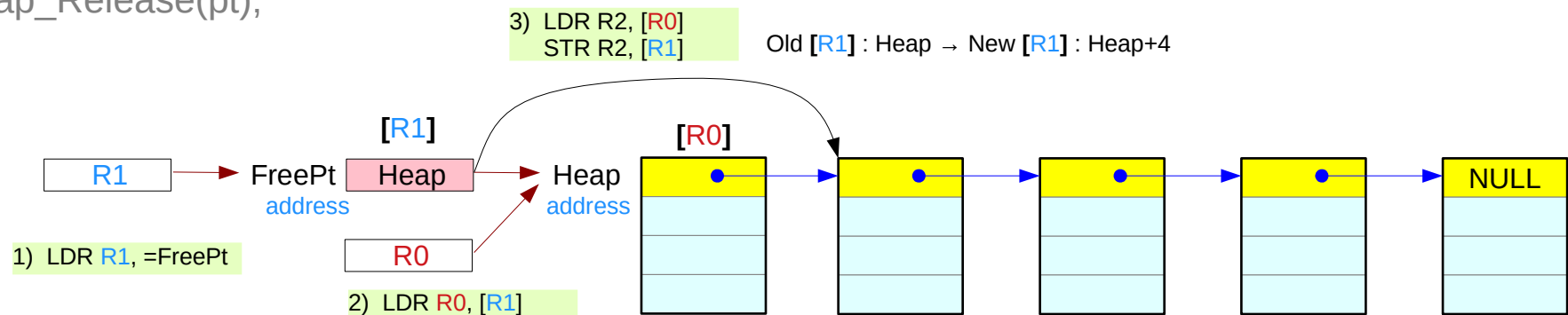
```
void Heap_Init(void) {  
    ...  
    int32_t * pt;  
    ...  
    pt = FreePt = &Heap[0];  
    ...  
}
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Heap_Allocate

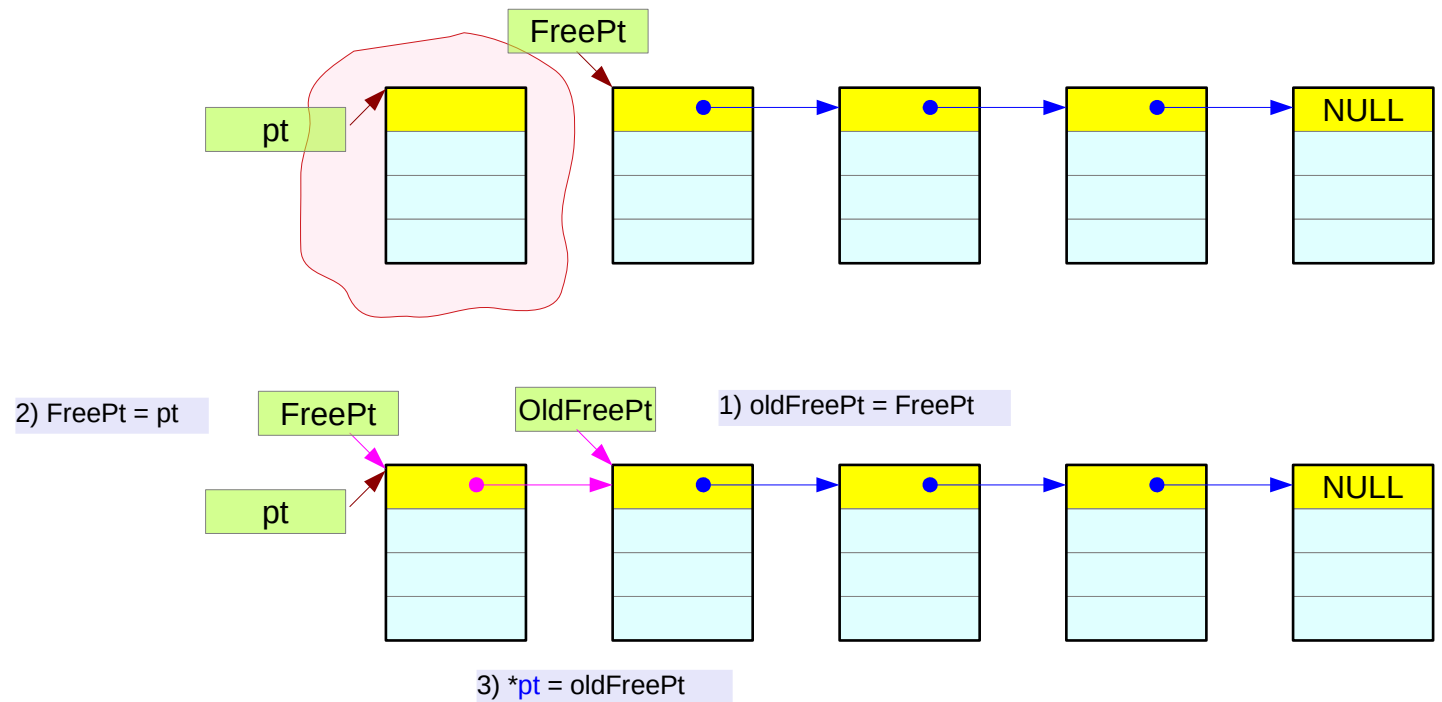
Heap_Allocate			; R0 points to new block
LDR	R1, =FreePt		; R1 = FreePt ; R1 ← &FreePt
LDR	R0, [R1]		; R0 = [FreePt] ; R0 ← [R1]
CMP	R0, #NULL		; if (pt != NULL)
LDR	R2, [R0]		; link next ; R2 ← [R0]
STR	R2, [R1]		; FreePt = *pt ; [R1] ← R2
Adone	BX	LR	

```
pt = Heap_Allocate();
Heap_Release(pt);
```



Heap_Release

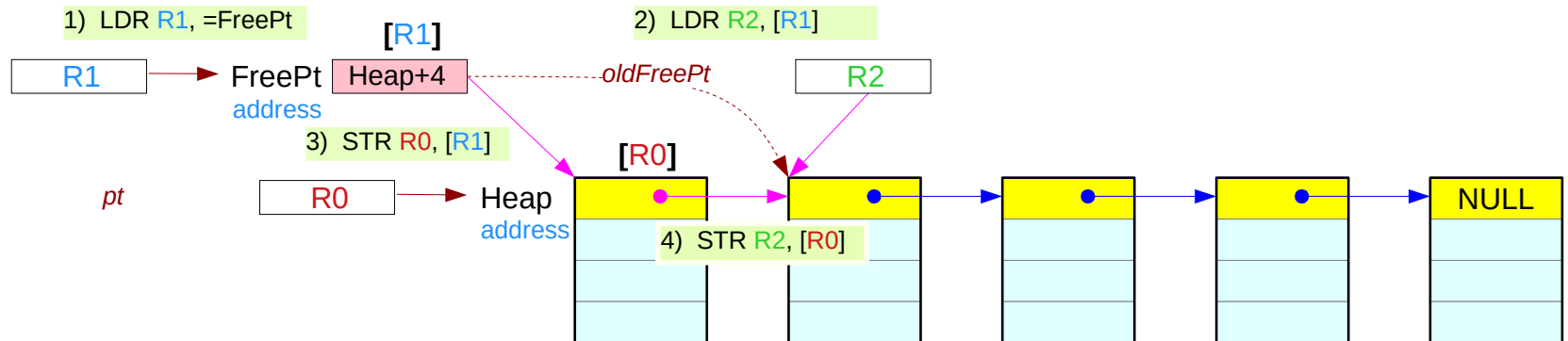
```
int32_t *Heap_Release(int32_t *pt) {  
    int32_t * oldFreePt;  
  
    oldFreePt = FreePt;  
    FreePt = pt;  
    *pt = (int32_t) oldFreePt;  
}
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Heap_Release

```
Heap_Release:                                ; R0 : block being released pt
    LDR    R1, =FreePt
    LDR    R2, [R1]                          ; R2 : oldFreePt
    STR    R0, [R1]                           ; FreePt = pt
    STR    R2, [R0]                           ; *pt=oldFreePt
    BX    LR
```



Fifo

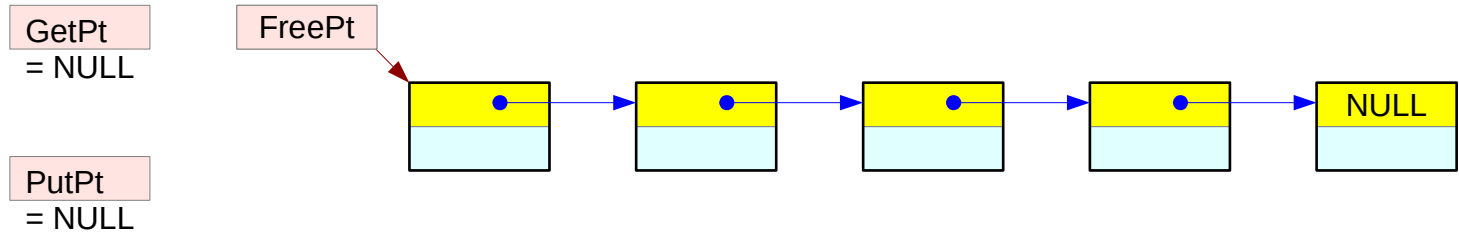
```
struct Node {  
    struct Node *Next;  
    int32_t Data;  
};  
typedef struct Node NodeType;
```

```
NodeType *PutPt; // place to put  
NodeType *GetPt; // place to get
```

```
; put in RAM  
NEXT EQU 0 ; next  
Data EQU 4 ; 32-bit data for node  
  
GetPt SPACE 4 ; pointer to oldest node  
PutPt SPACE 4 ; pointer to newest node
```

```
void Fifo_Init(void) {
    GetPt = NULL;
    PutPt = NULL;
    Heap_Init();
}
```

// Empty when null



In Heap_Init

```
#define SIZE 2
#define NUM 5
#define NULL 0 // empty pointer

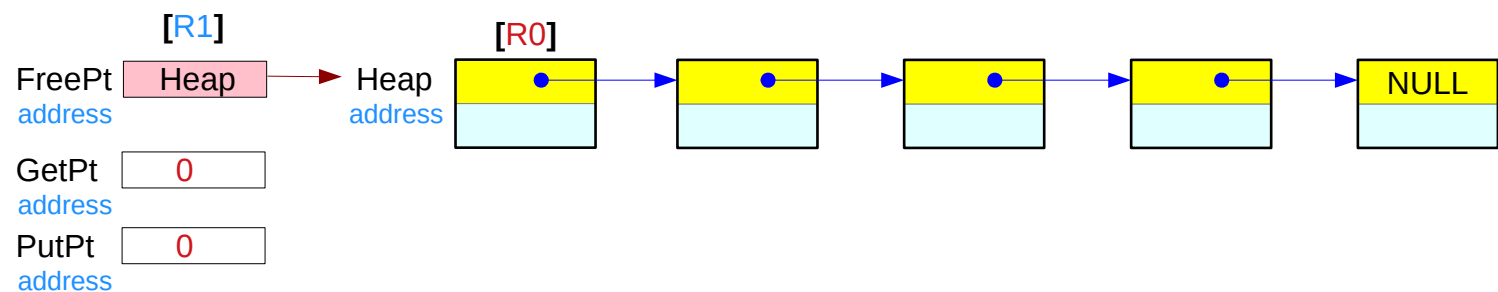
int32_t *FreePt;
int32_t Heap[SIZE*NUM];

SIZE EQU 2
NUM EQU 5
NULL EQU 0
FreePT SPACE 4
Heap SPACE SIZE*NUM*4
```

```

Fifo_Init:    PUSH    {R4, LR}
              MOV     R1, #NULL;
              LDR     R0, =GetPt          ; R0 ← GetPt
              STR     R1, [R0]           ; GetPt = NULL ; [GetPt] ← NULL
              LDR     R0, =PutPt         ; R0 ← PutPt
              STR     R1, [R0]           ; PutPt = NULL ; [PutPt] ← NULL
              BL     Heap_Init
              POP     {R4, PC}
    
```

3) LDR R2, [R0]
STR R2, [R1] Old [R1] : Heap → New [R1] : Heap+4



```
int Fifo_Put(int32_t theData) {
    NodeType *pt;           // Empty when null

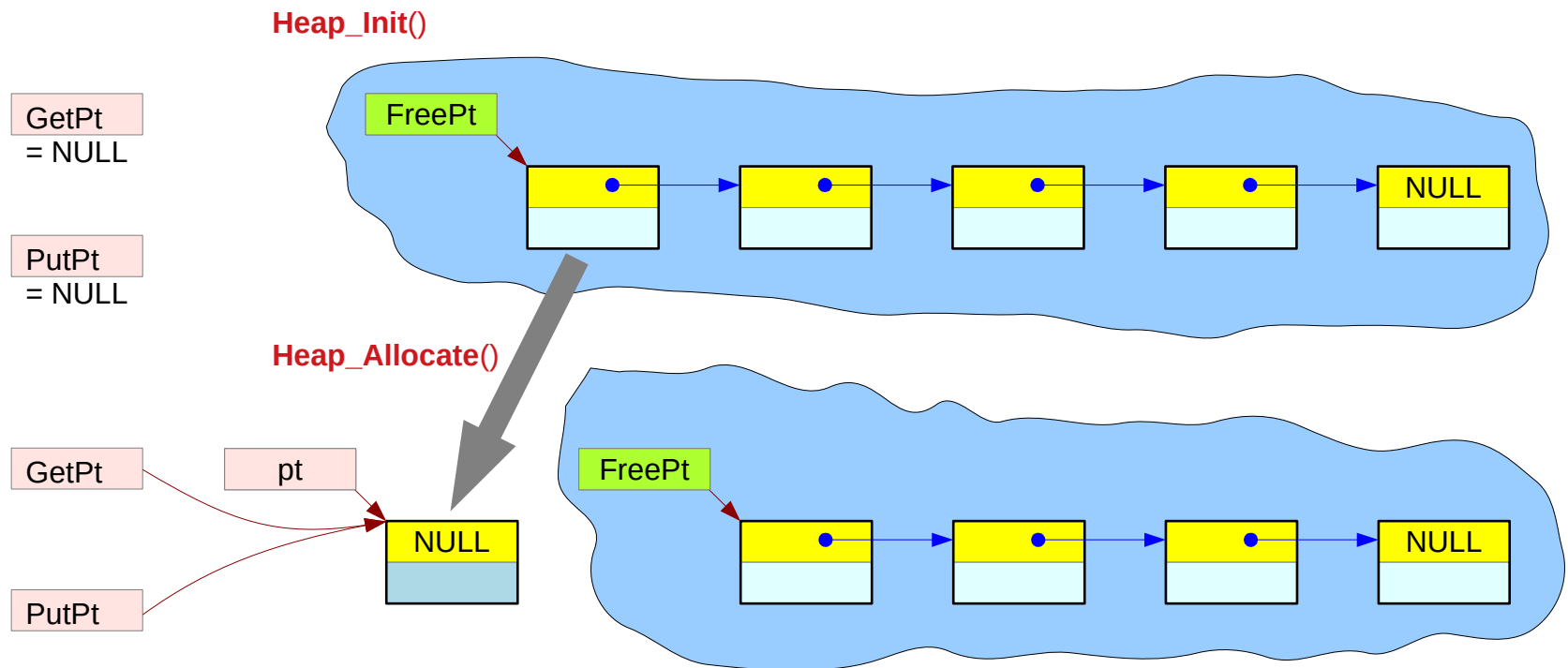
    pt = (nodeType*) Heap_Allocate();
    if (!pt) {
        return(0);         // full
    }

    pt->Data = theData;    // store
    pt->Next = NULL;

    if (PutPt) {
        PutPt->Next = pt; // Link
    } else {
        GetPt = pt;      // first one
    }

    PutPt = pt;
    return(1);           // successful
}
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano



```

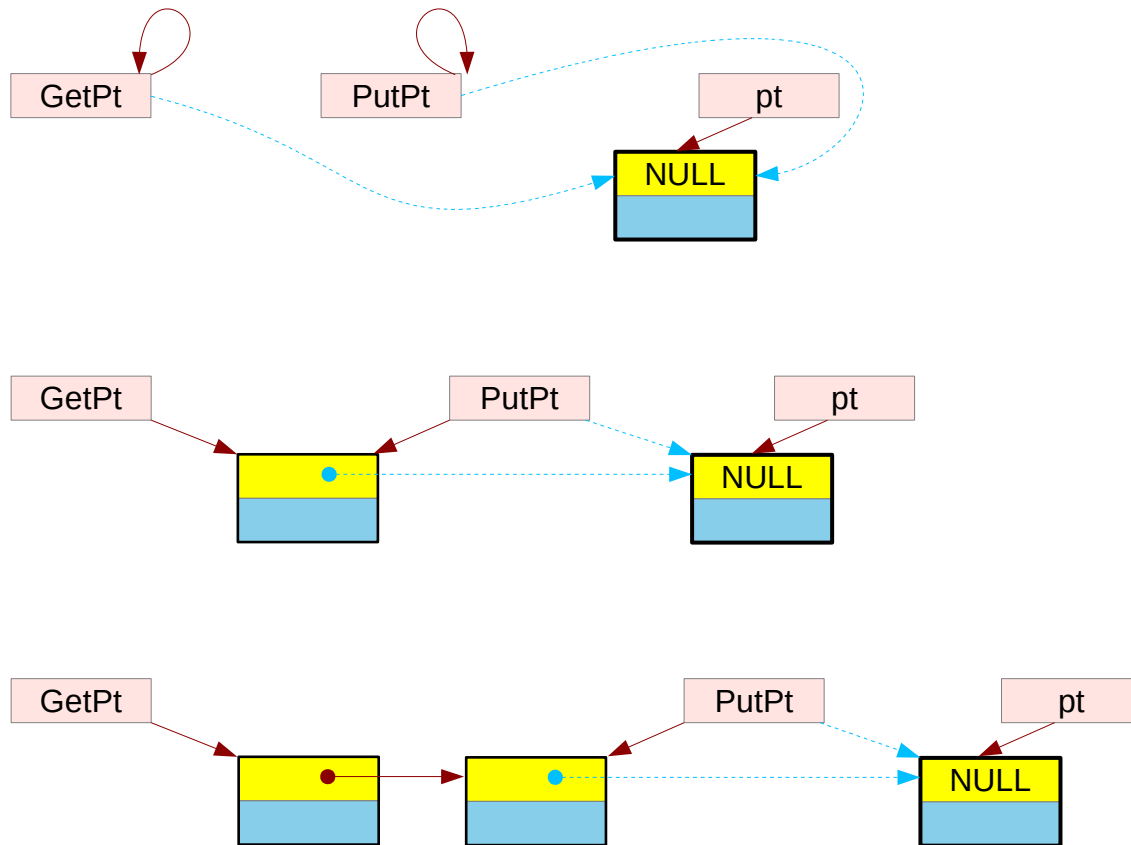
int Fifo_Put(int32_t theData) {
    NodeType *pt;
    pt = (nodeType*) Heap_Allocate();
    if (!pt) return(0);
    pt->Data = theData;
    pt->Next = NULL;
}

```

```

if (PutPt) PutPt->Next = pt;
else GetPt = pt;
PutPt = pt;
return(1);
}

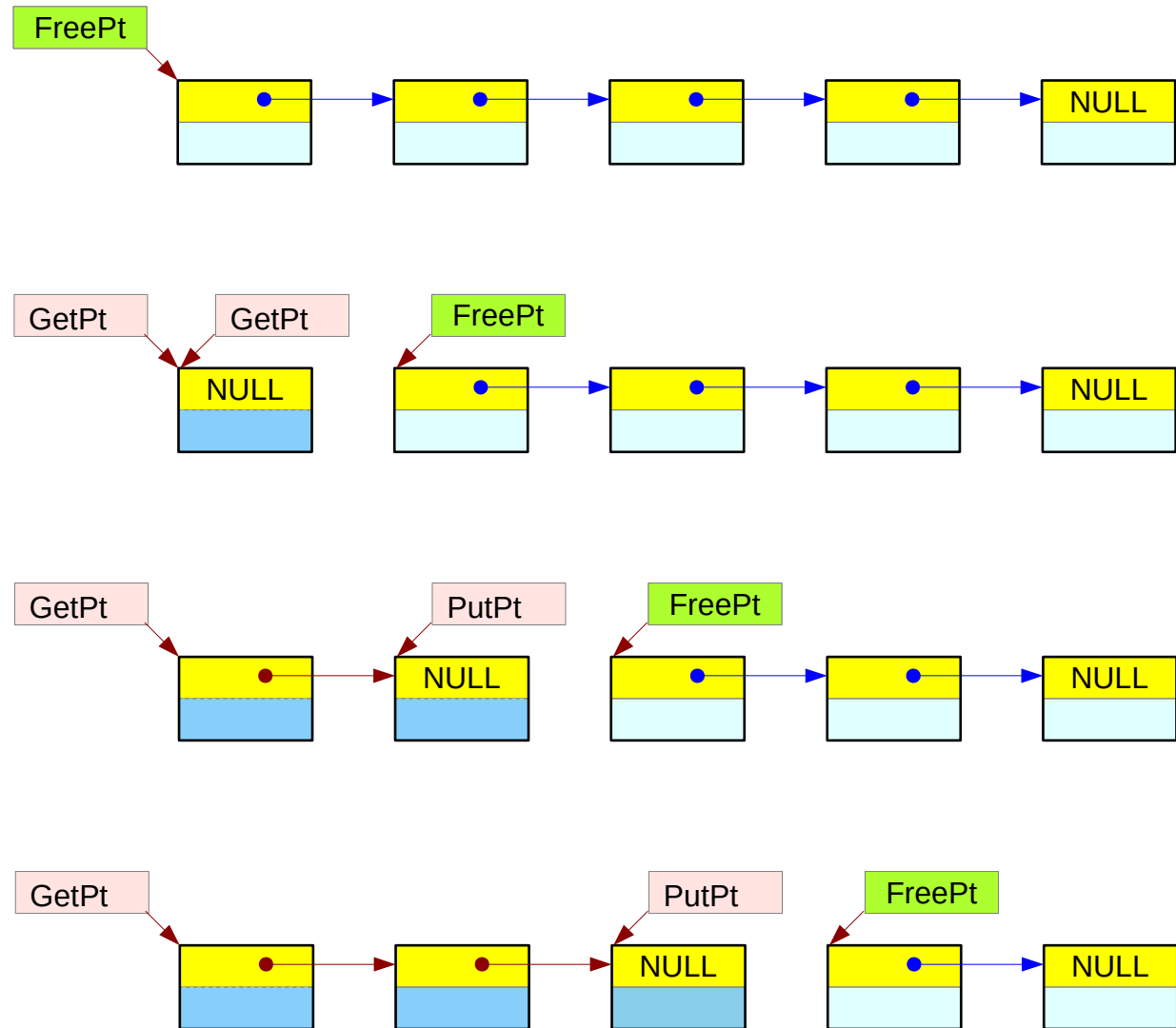
```

```
int Fifo_Put(int32_t theData) {
    NodeType *pt;
    pt = (nodeType*) Heap_Allocate();
    if (!pt) return(0);
    pt->Data = theData;
    pt->Next = NULL;
    if (PutPt) PutPt->Next = pt;
    else GetPt = pt;
    PutPt = pt;
    return(1);
}
```

GetPt
= NULL

PutPt
= NULL



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

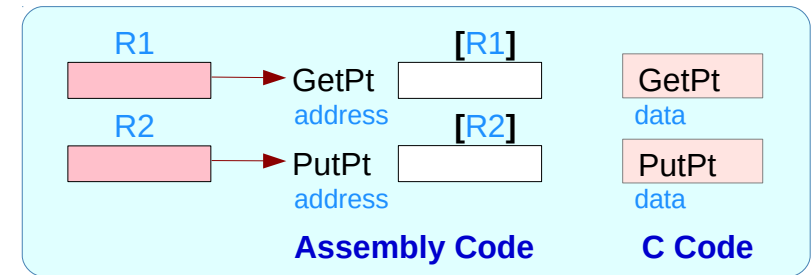
Fifo_Put – overview

; Inputs: R0 value, data to put
 ; Outputs: R0=1 if successful
 ; R0=0 in unsuccessful

Fifo_Put

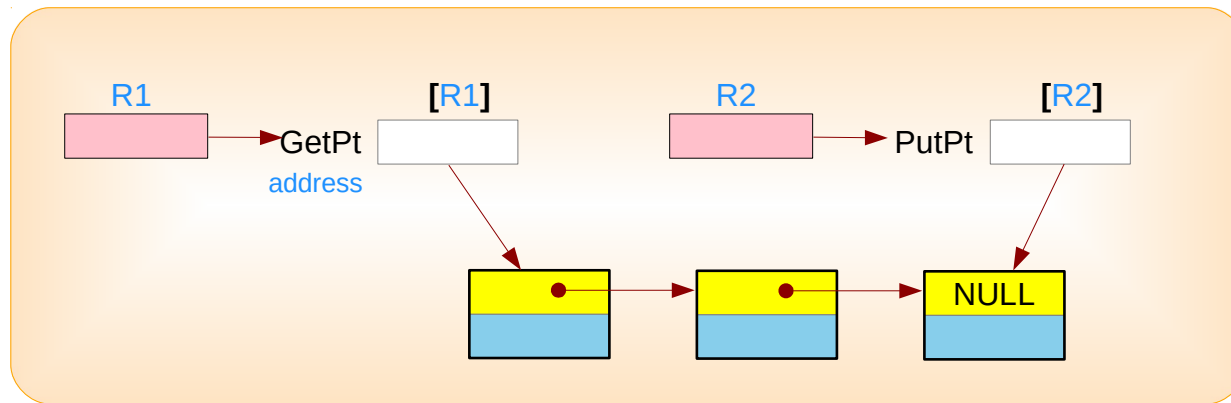
```

PUSH {R4, LR}
MOV R4, R0 ; copy the data to put into R4
BL Heap_Allocate ; returns the free node address in R0
CMP R0, #NULL ; R0 return value ; pt node address
BEQ Pful ; skip if full
STR R4, [R0, #Data] ; store 'data' ; R4 → pt's data
MOV R1, #NULL ; last node ; R1 ← #NULL
STR R1, [R0, #Next] ; store 'next' ; R1 → pt's next
LDR R2, =PutPt ; address PutPt ; R2 ← PutPt (address)
LDR R3, [R2] ; Put node address ; R3 ← [PutPt] (content)
CMP R3, #NULL ; previously empty? ; R3 == #NULL
BEQ PMT ; Put eMpTy routine – no link
STR R0, [R3, #Next] ; link previous ; R3 → [PutPt]'s next
B Pcon
PMT LDR R1, =GetPt ; address GetPt ; R1 ← GetPt (address)
STR R0, [R1] ; Get node address ; R0 → [GetPt] (content)
PCon STR R0, [R2] ; Points to newst ; R0 → [PutPt]
; Now one entry
MOV R0, #1 ; success
B Pdon
PFul MOV R0, #0 ; failure, full
PDon POP {R4, PC}
    
```

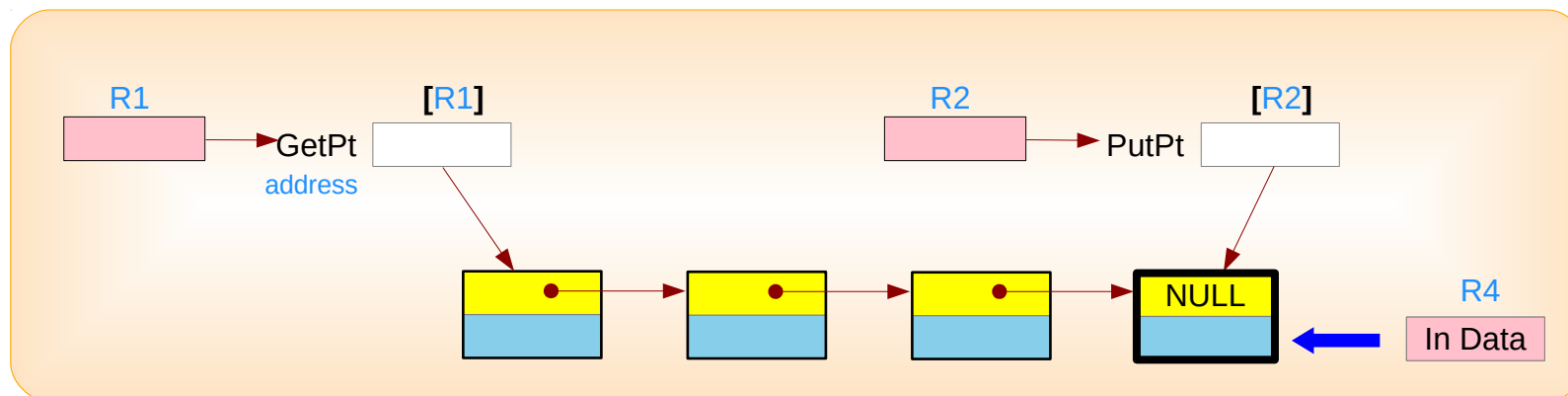
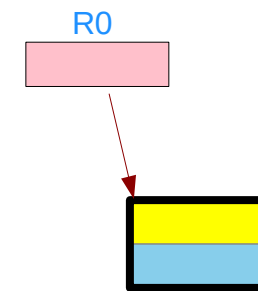


```

int Fifo_Put(int32_t theData) {
    NodeType *pt;
    pt = (NodeType*) Heap_Allocate();
    if (!pt) return(0);
    pt->Data = theData;
    pt->Next = NULL;
    if (PutPt) PutPt->Next = pt;
    else GetPt = pt;
    PutPt = pt;
    return(1);
}
    
```



returns a free node address



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Fifo_Put

PUSH {R4, LR}

MOV R4, R0

BL Heap_Allocate

CMP R0, #NULL

BEQ Pful

STR R4, [R0, #Data]

MOV R1, #NULL

STR R1, [R0, #Next]

LDR R2, =PutPt

LDR R3, [R2]

CMP R3, #NULL

BEQ PMT

STR R0, [R3, #Next]

B PCon

PMT LDR R1, =GetPt

STR R0, [R1]

PCon STR R0, [R2]

MOV R0, #1

B Pdon

PFul MOV R0, #0

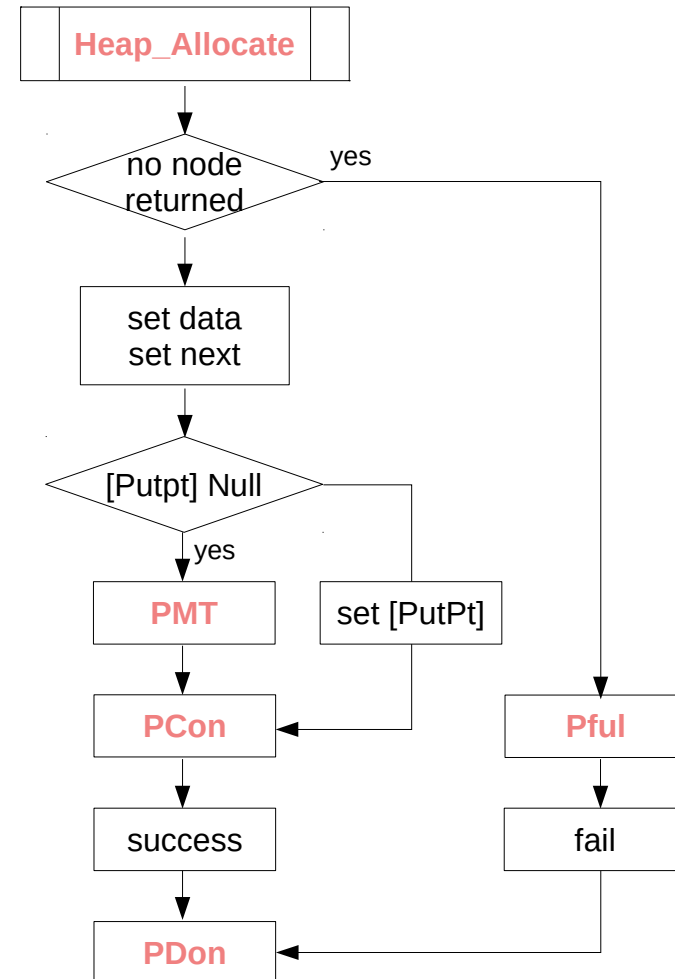
PDon POP {R4, PC}

Put_Full

Put_Empty

Put_Conclude

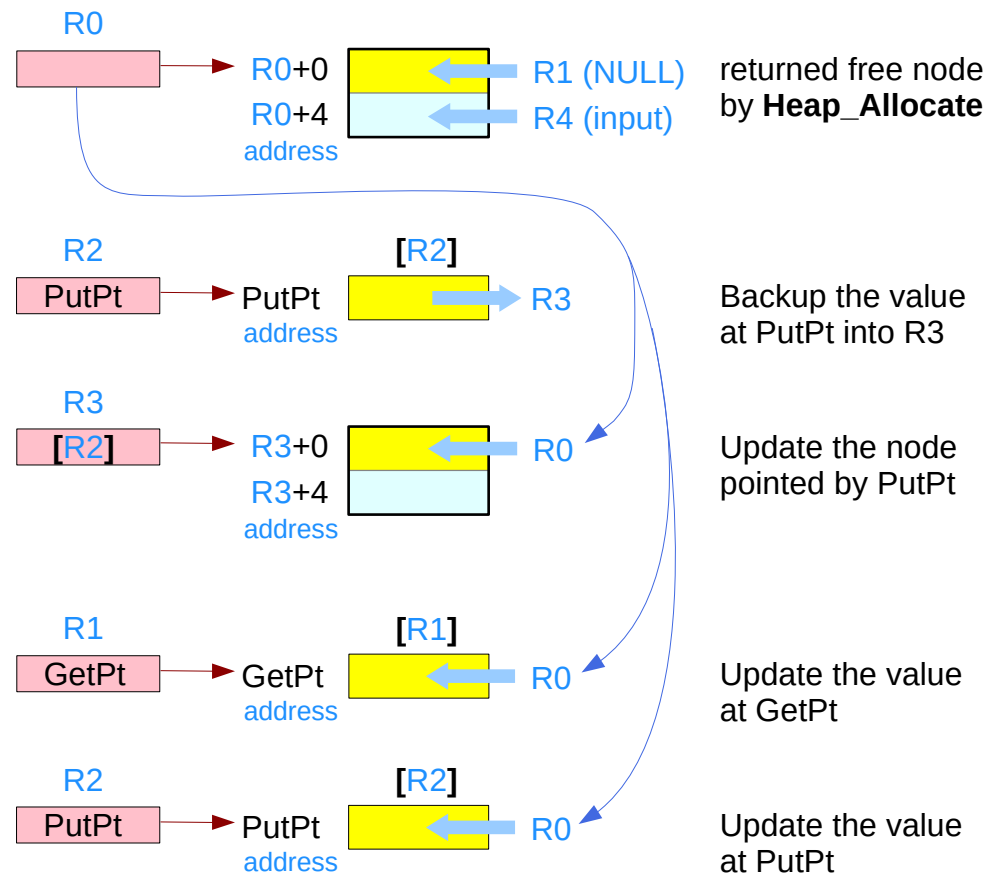
Put_Done



Fifo_Put

```

PUSH {R4, LR}
MOV R4, R0
BL Heap_Allocate
CMP R0, #NULL
BEQ Pful
STR R4, [R0, #Data]
MOV R1, #NULL
STR R1, [R0, #Next]
LDR R2, =PutPt
LDR R3, [R2]
CMP R3, #NULL
BEQ PMT
STR R0, [R3, #Next]
B Pcon
PMT LDR R1, =GetPt
STR R0, [R1]
PCon STR R0, [R2]
MOV R0, #1
B Pdon
PFull MOV R0, #0
PDon POP {R4, PC}
    
```



Fifo_Put

```
PUSH {R4, LR}
MOV R4, R0
```

BL Heap_Allocate

```
CMP R0, #NULL
BEQ Pful
```

```
STR R4, [R0, #Data]
```

```
MOV R1, #NULL
```

```
STR R1, [R0, #Next]
```

```
LDR R2, =PutPt
```

```
LDR R3, [R2]
```

```
CMP R3, #NULL
```

```
BEQ PMT
```

```
STR R0, [R3, #Next]
```

```
B Pcon
```

PMT LDR R1, =GetPt

```
STR R0, [R1]
```

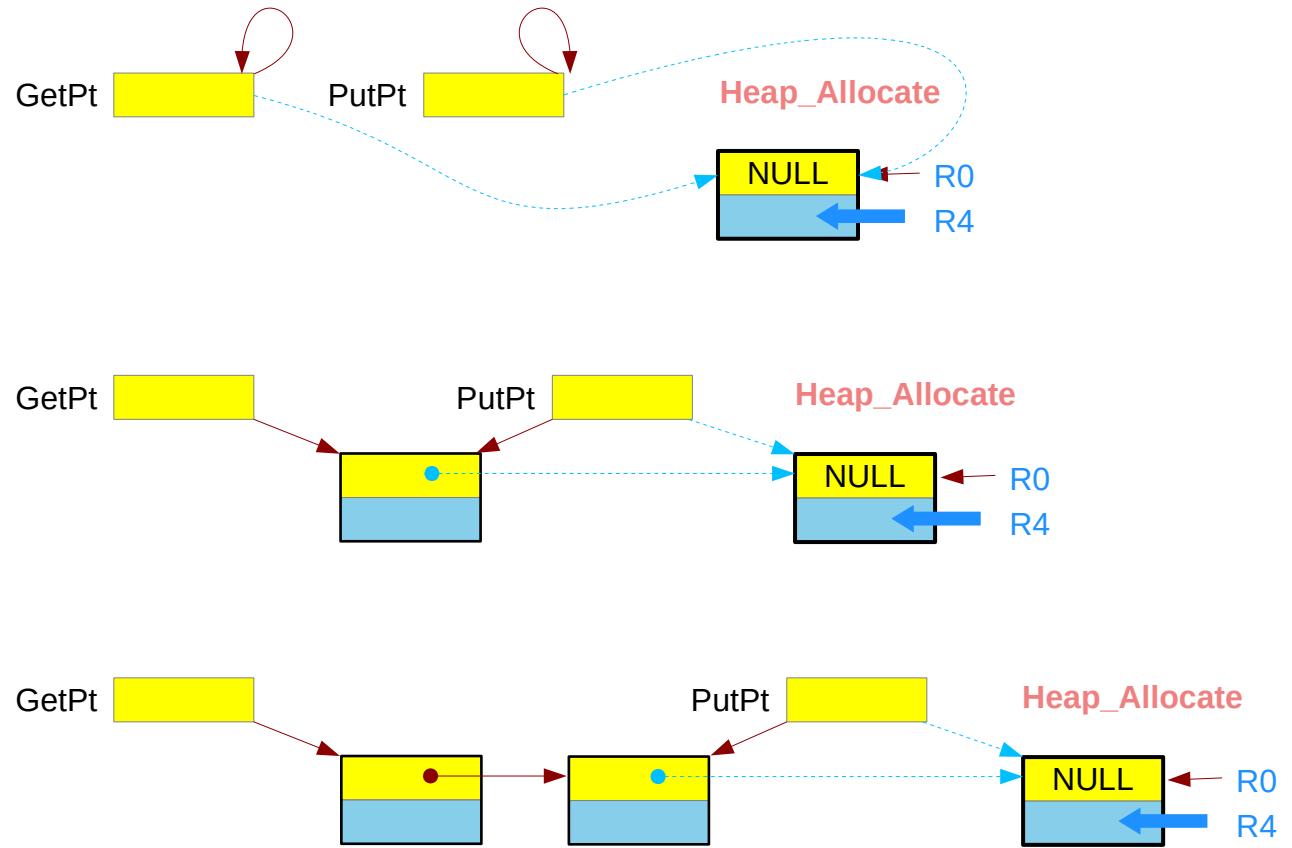
PCon STR R0, [R2]

```
MOV R0, #1
```

```
B Pdon
```

PFul MOV R0, #0

PDon POP {R4, PC}



```

; Inputs:    R0 value, data to put
; Outputs:  R0=1 if successful
;           R0=0 in unsuccessful
    
```

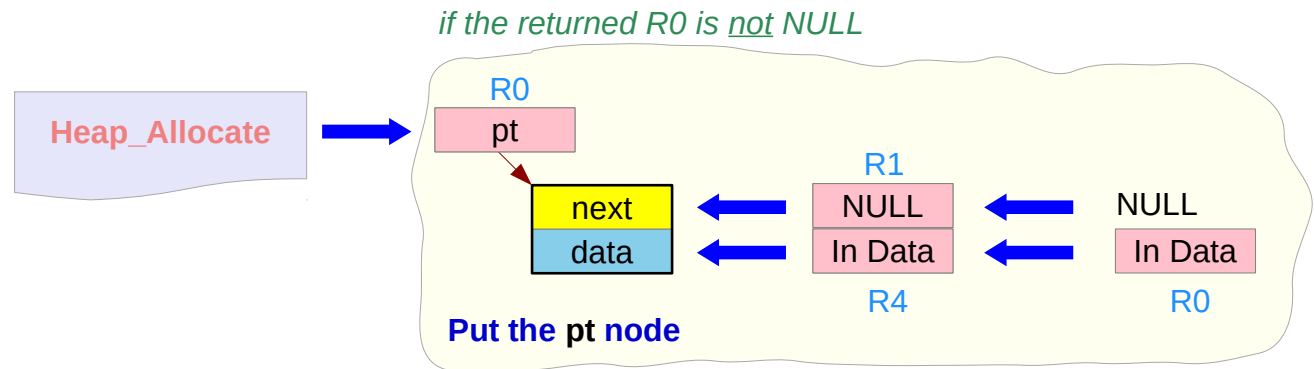
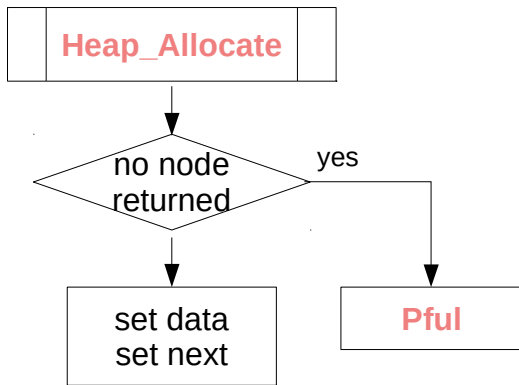
Fifo_Put

```

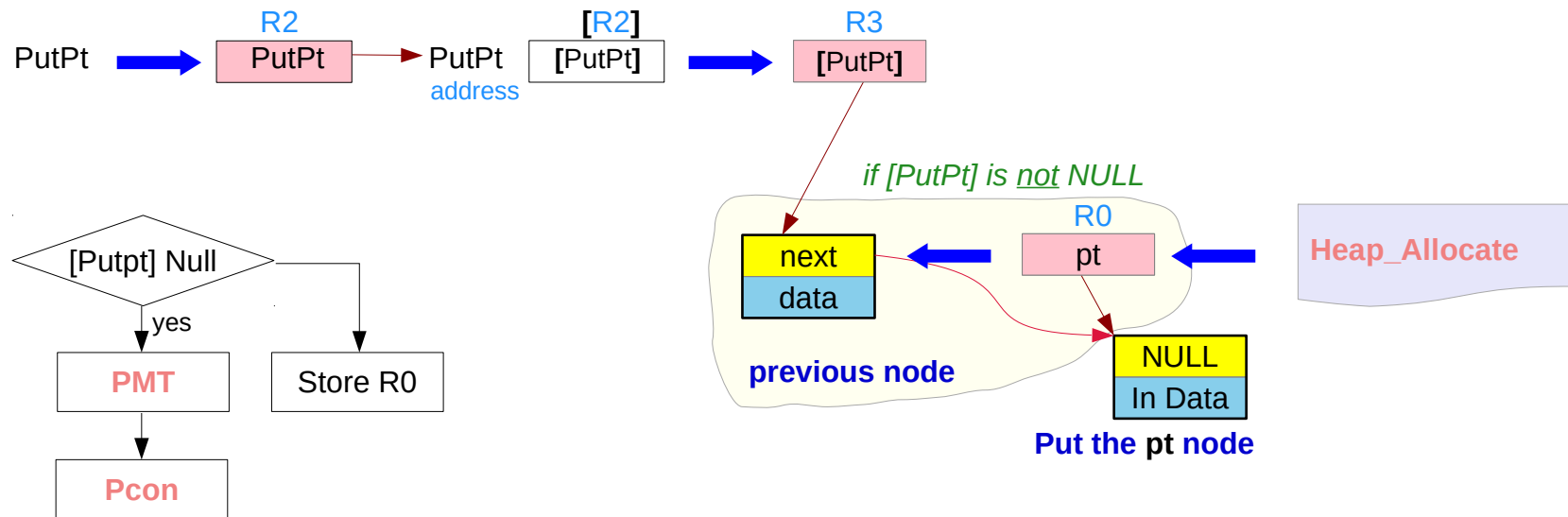
PUSH    {R4, LR}
MOV     R4, R0
BL      Heap_Allocate
CMP     R0, #NULL
BEQ     Pful
STR     R4, [R0, #Data]
MOV     R1, #NULL
STR     R1, [R0, #Next]
    
```

```

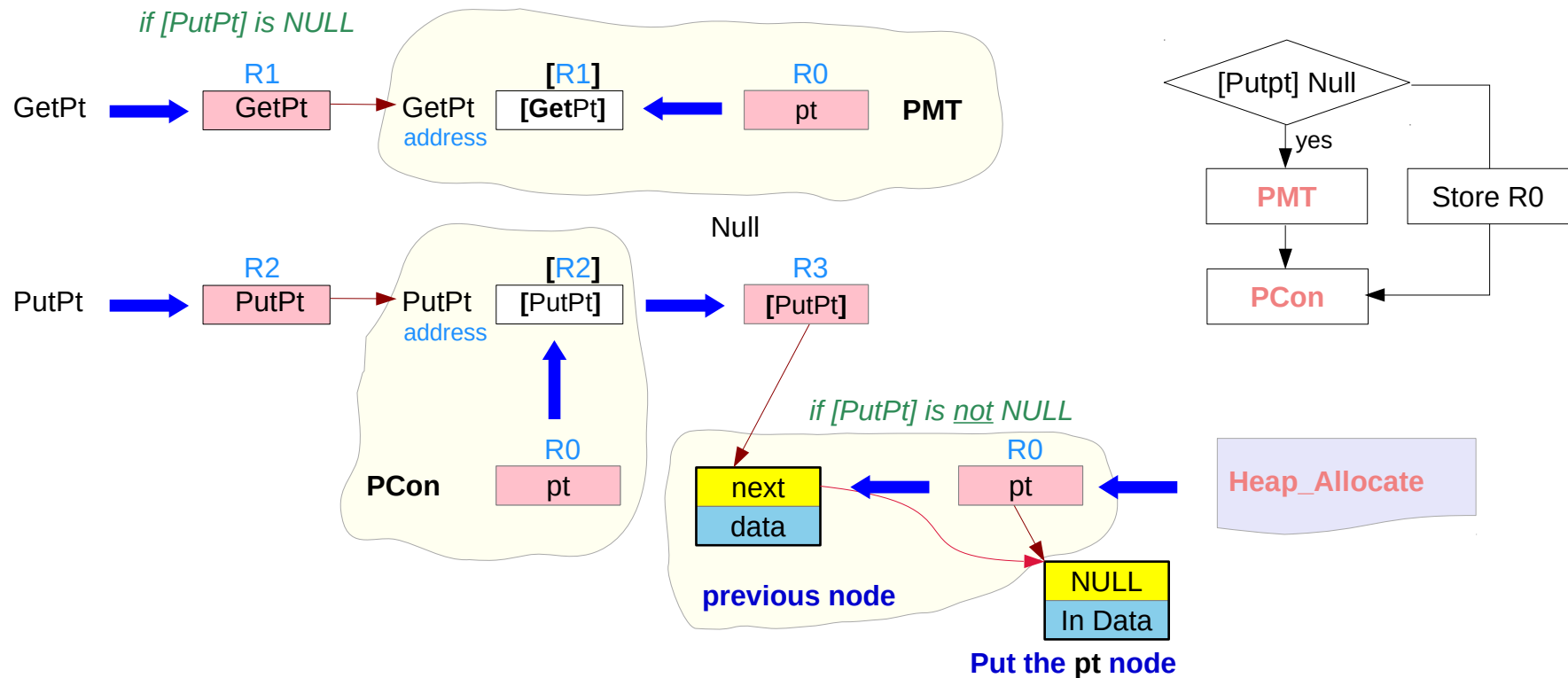
R0 In Data → R4 In Data
; copy the data to put into R4
; returns the free node address in R0
; R0 return value ; pt node address
; skip if full
; store 'data' ; R4 → pt's data
; last node ; R1 ← #NULL
; store 'next' ; R1 → pt's next
    
```



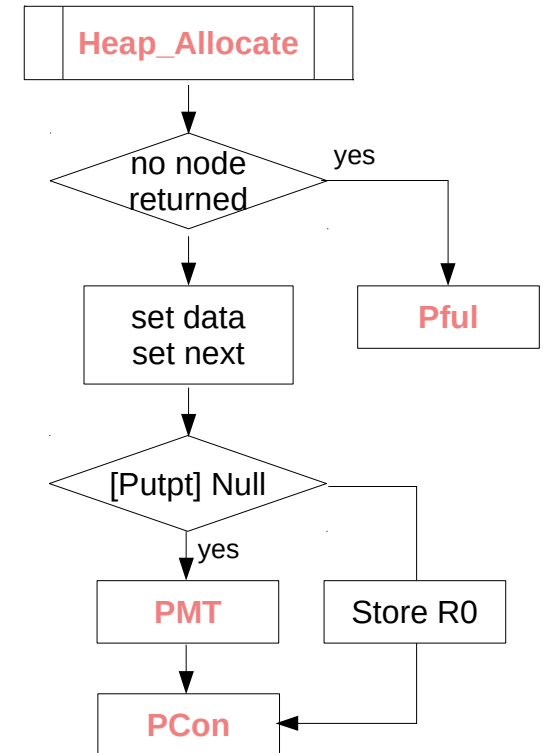
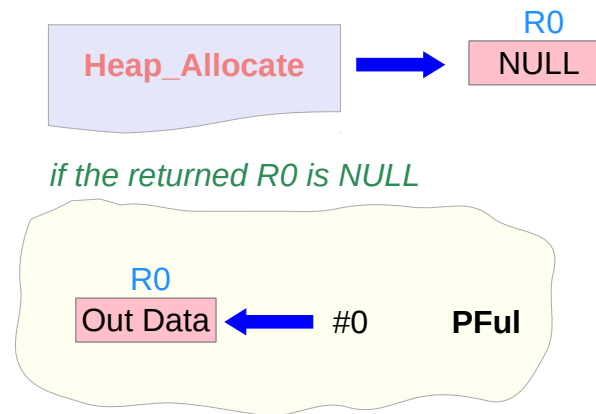
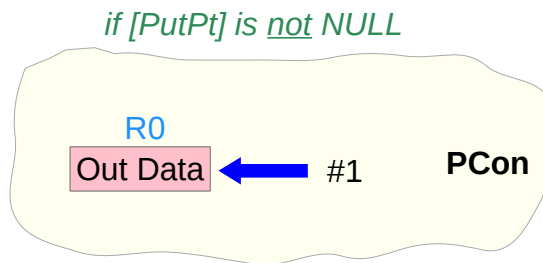
LDR	R2, =PutPt	; address PutPt	; R2 ← PutPt (address)
LDR	R3, [R2]	; Put node address	; R3 ← [PutPt] (content)
CMP	R3, #NULL	; previously empty?	; R3 == #NULL
BEQ	PMT	; Put eMpTy routine – no link	
STR	R0, [R3, #Next]	; link previous	; R3 → [PutPt]'s next
B	Pcon		



PMT	LDR R1, =GetPt	; address GetPt	; R1 ← GetPt (address)
	STR R0, [R1]	; Get node address	; R0 → [GetPt] (content)
PCon	STR R0, [R2]	; Points to newst	; R0 → [PutPt]
		; Now one entry	



	MOV	R0, #1	; success
	B	Pdon	
PFul	MOV	R0, #0	; failure, full
PDon	POP	{R4, PC}	



```
Int Fifo_Get(int32_t *datap) {
    NodeType *pt;

    if (!GetPt) {
        return (0);           // empty
    }

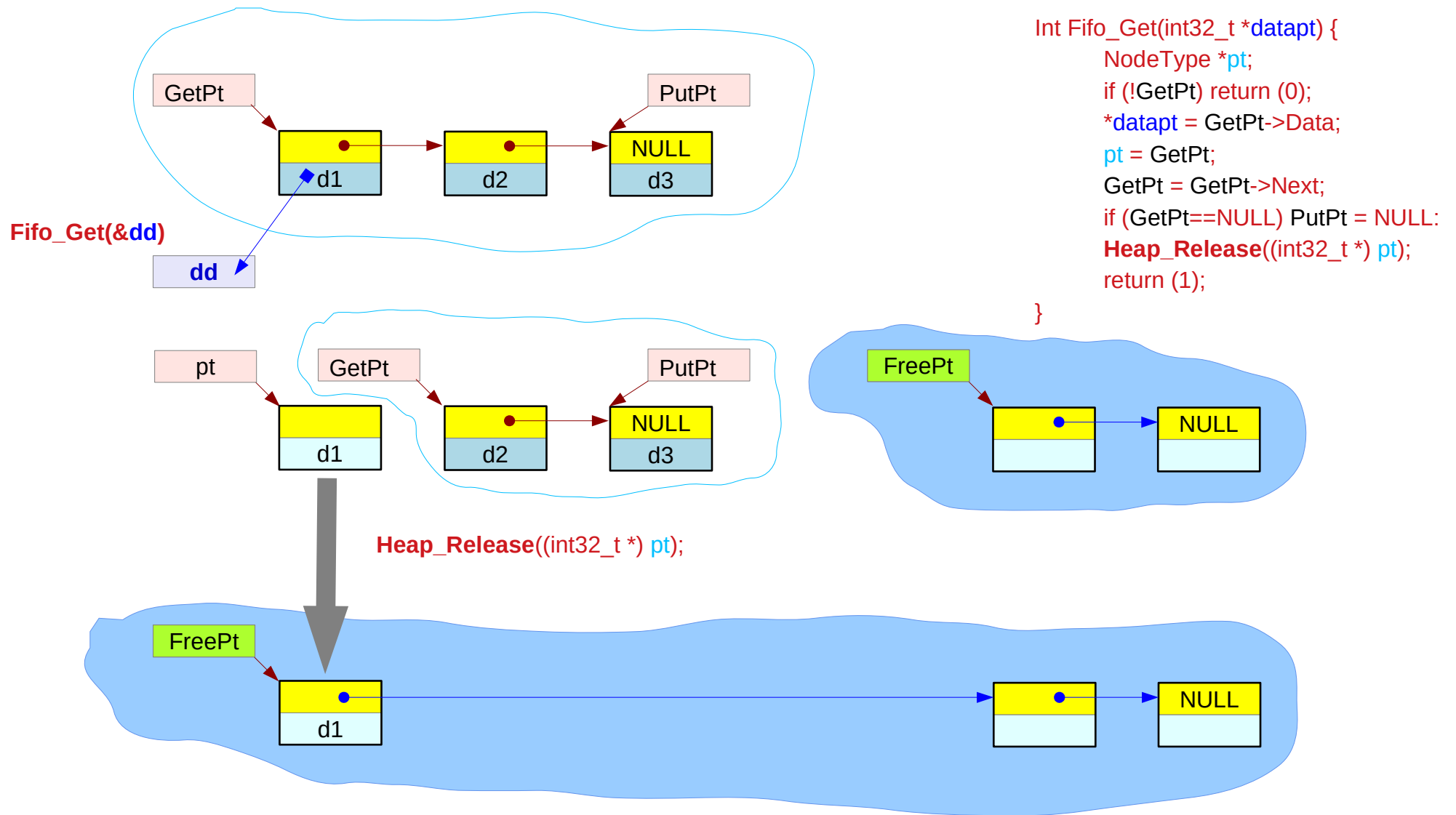
    *datap = GetPt->Data;
    pt = GetPt;
    GetPt = GetPt->Next;

    if (GetPt==NULL) {       // one entry
        PutPt = NULL;
    }

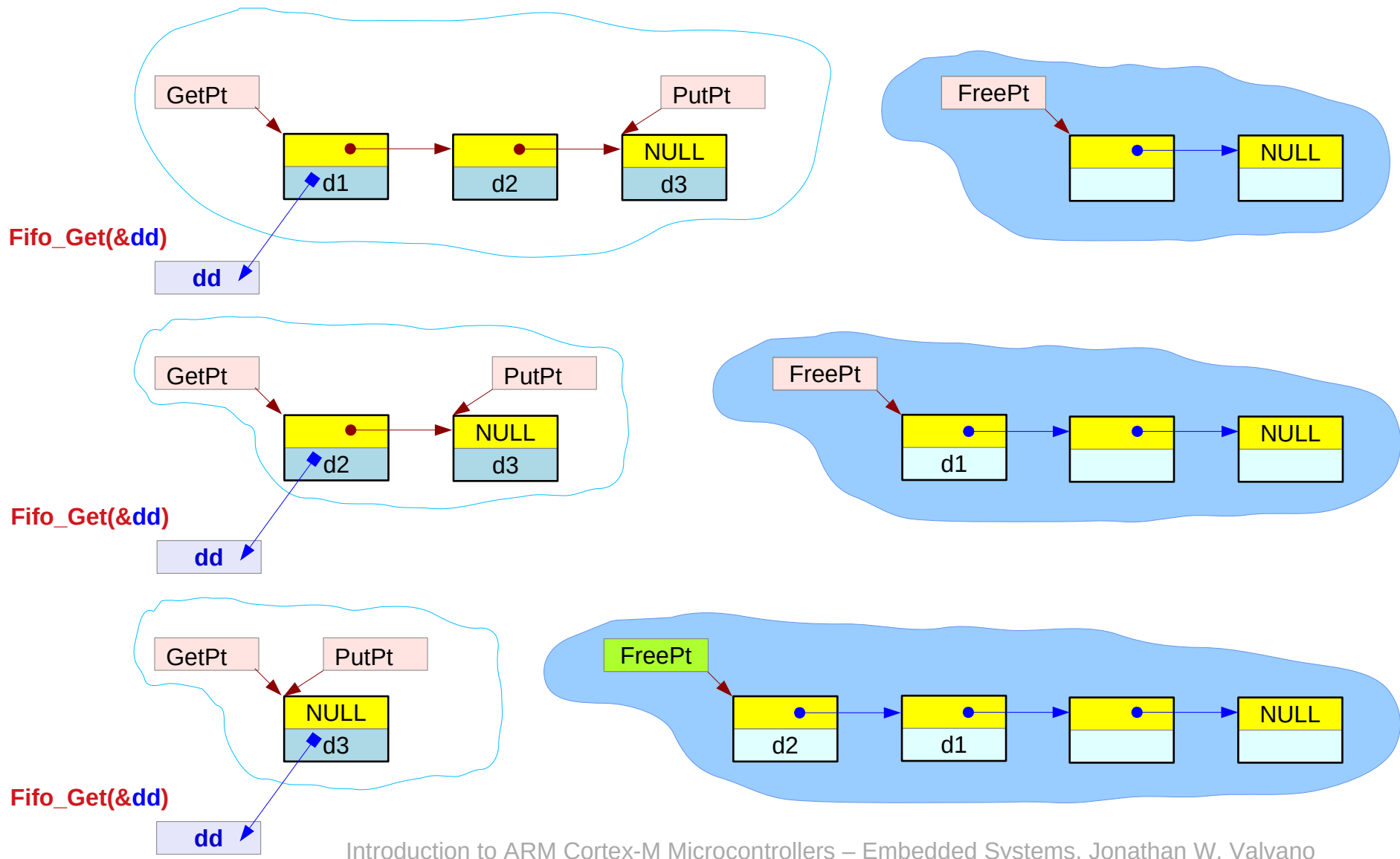
    Heap_Release((int32_t *) pt);

    return (1);             // success
}
```

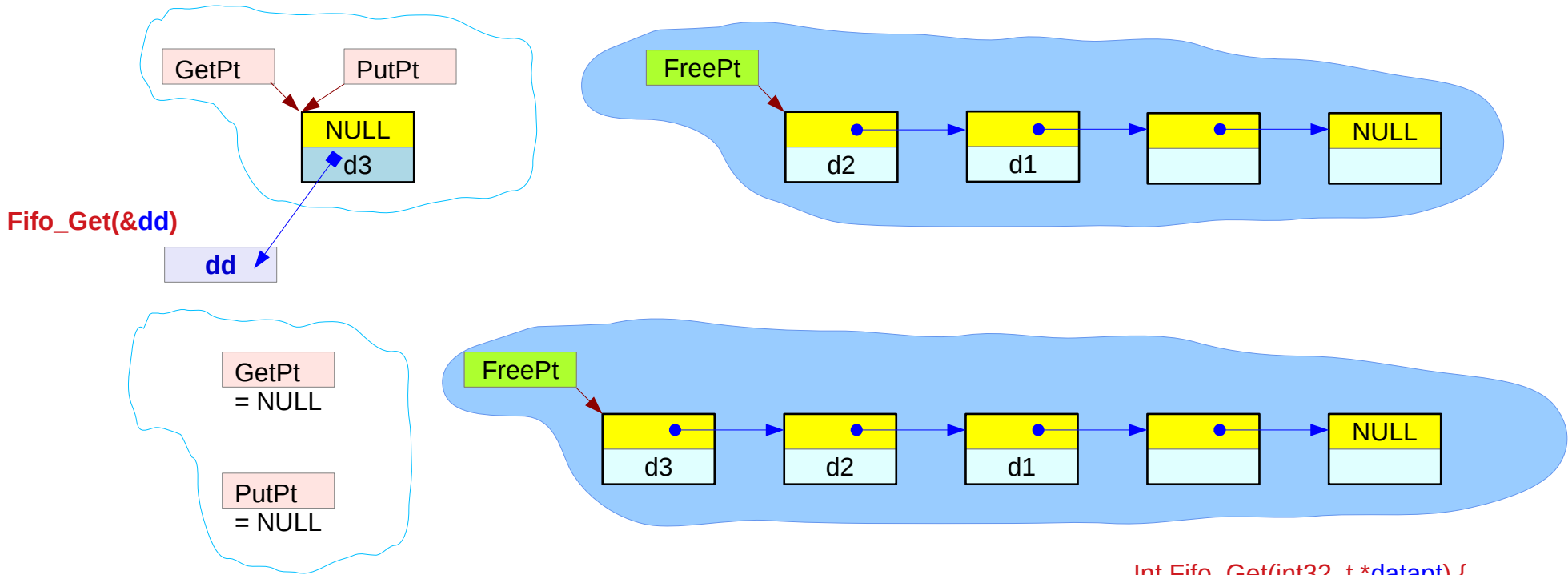
Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

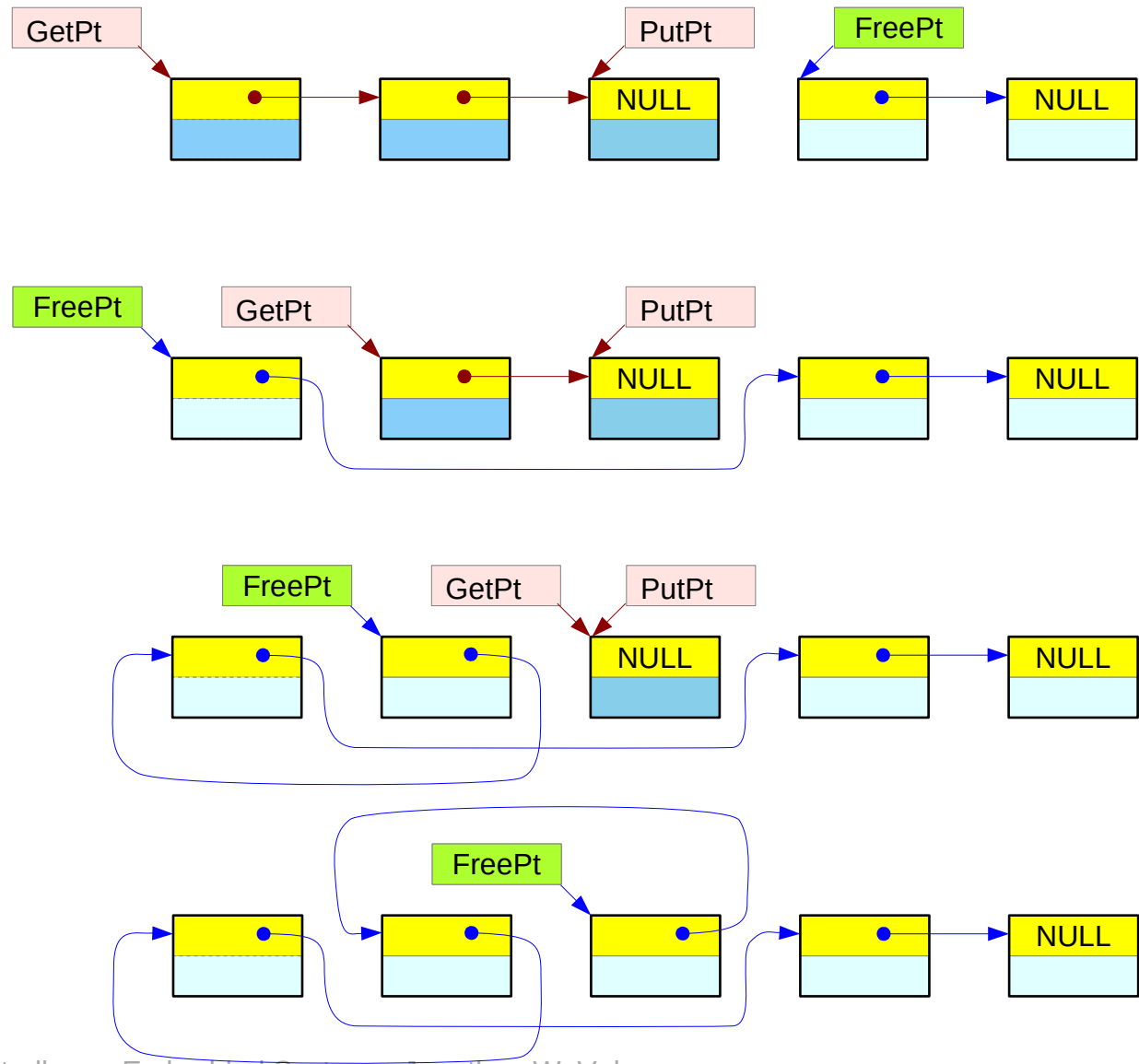


Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano



```

Int Fifo_Get(int32_t *datap) {
    NodeType *pt;
    if (!GetPt) return (0);
    *datap = GetPt->Data;
    pt = GetPt;
    GetPt = GetPt->Next;
    if (GetPt==NULL) PutPt = NULL;
    Heap_Release((int32_t *) pt);
    return (1);
}
    
```



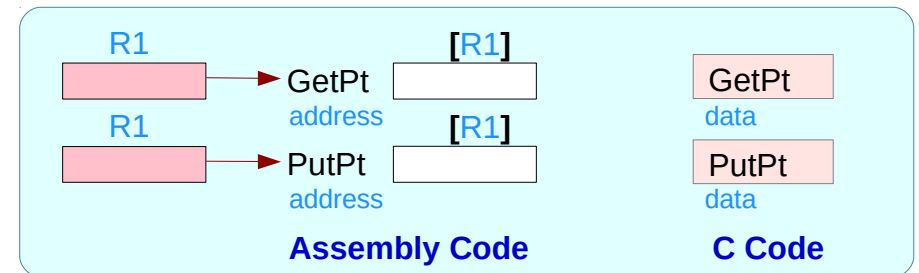
Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

; Inputs: R0 points to an empty place
 ; Outputs: data removed to place
 R0=0 if successful
 R0=1 if empty

Fifo_Get

```

PUSH {R4, LR}
LDR R1, =GetPt      ; address GetPt      ; R1 ← GetPt (address)
LDR R2, [R1]        ; Get node address ; R2 ← [GetPt] (content)
CMP R2, #NULL
BEQ GMT            ; empty if NULL
LDR R3, [R2, #Data] ; read 'data'      ; R3 ← pt's data
STR R3, [R0]        ; store 'data'     ; int Fifo_Get(int32_t *datap)
LDR R3, [R2, #Next] ; read 'next'     ; R3 ← pt's next
STR R3, [R1]        ; store 'next'     ; R3 → [GetPt]
CMP R3, #NULL      ; the last node   ; pt node
BNE GCon           ; if not empty     ; pt's next <> NULL
LDR R1, =PutPt     ; address PutPt   ; R1 ← PutPt (address)
STR R3, [R1]        ; Put node address ; NULL → [PutPt] (content)
MOV R0, R2          ; old data       ; R0 ← [GetPt] ; input arg
BL Heap_Release
MOV R0, #1          ; success
B GDon
GMT                 ; failure, empty
GDon               POP {R4, PC}
    
```

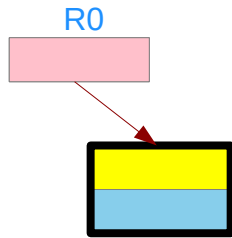
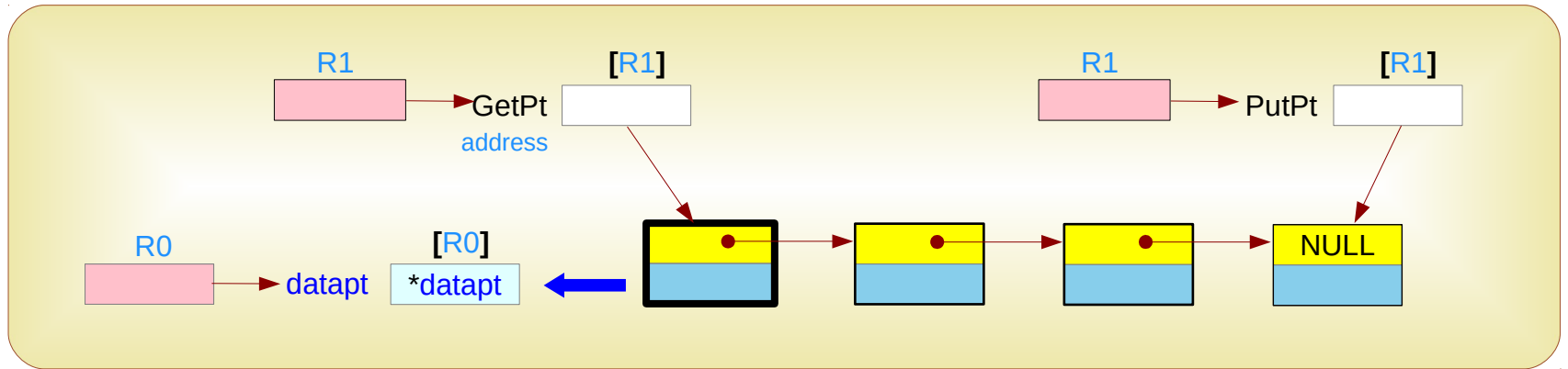


```

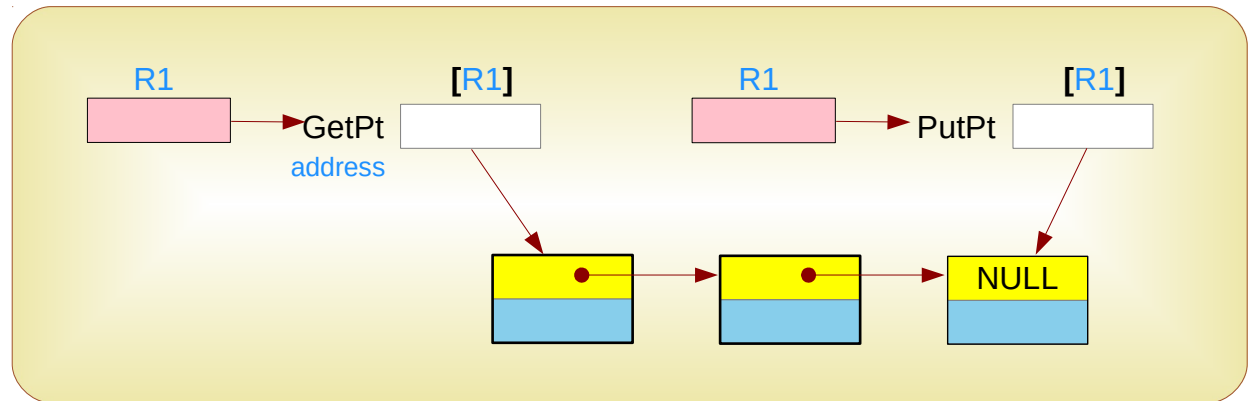
int Fifo_Get(int32_t *datap) {
    NodeType *pt;

    if (!GetPt) return (0);
    *datap = GetPt->Data;
    pt = GetPt;
    GetPt = GetPt->Next;
    if (GetPt==NULL) PutPt = NULL;
    Heap_Release((int32_t *) pt);
    return (1);
}
    
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano



argument : a release node address



Fifo_Get

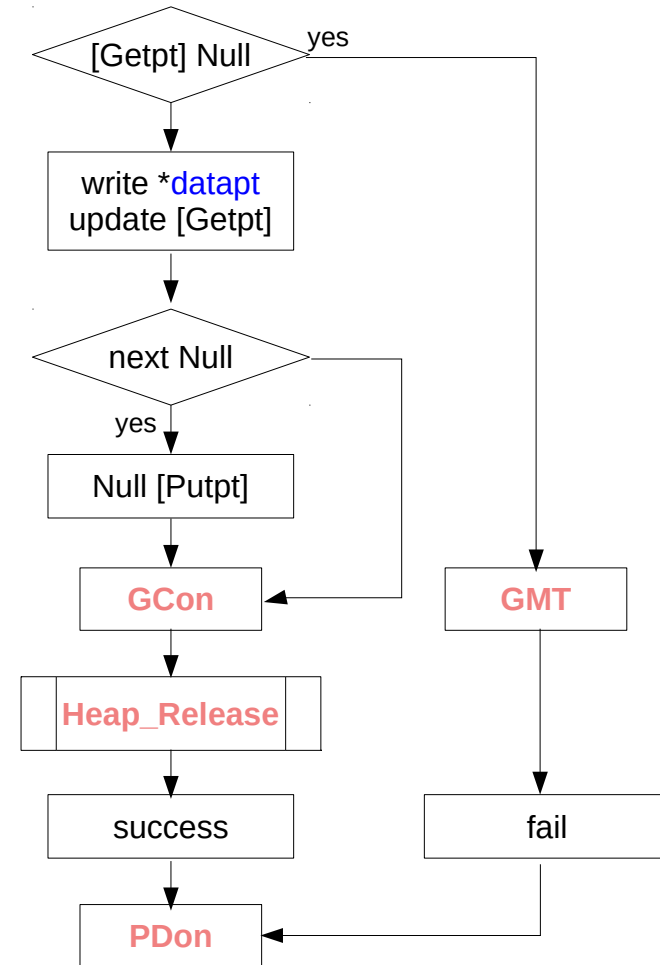
```

PUSH {R4, LR}
LDR R1, =GetPt
LDR R2, [R1]
CMP R2, #NULL
BEQ GMT
LDR R3, [R2, #Data]
STR R3, [R0]
LDR R3, [R2, #Next]
STR R3, [R1]
CMP R3, #NULL
BNE GCon
LDR R1, =PutPt
STR R3, [R1]
Gcon MOV R0, R2
BL Heap_Release
MOV R0, #1
B GDon
GMT MOV R0, #0
GDon POP {R4, PC}
    
```

Get_Empty

Get_Conclude

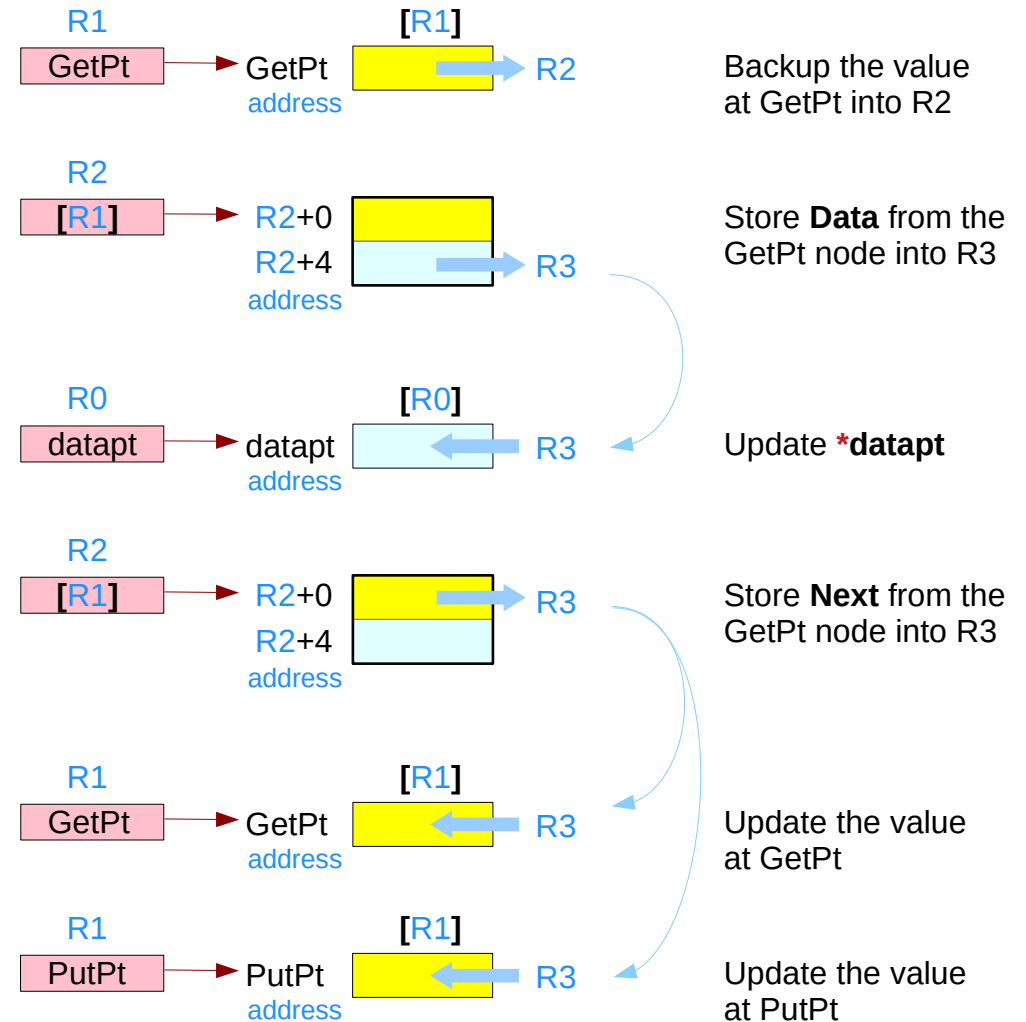
Get_Done



Fifo_Get

```

PUSH {R4, LR}
LDR R1, =GetPt
LDR R2, [R1]
CMP R2, #NULL
BEQ GMT
LDR R3, [R2, #Data]
STR R3, [R0]
LDR R3, [R2, #Next]
STR R3, [R1]
CMP R3, #NULL
BNE GCon
LDR R1, =PutPt
STR R3, [R1]
Gcon MOV R0, R2
BL Heap_Release
MOV R0, #1
B GDon
GMT MOV R0, #0
GDon POP {R4, PC}
    
```



Fifo_Get

```
PUSH {R4, LR}
```

```
LDR R1, =GetPt
```

```
LDR R2, [R1]
```

```
CMP R2, #NULL
```

```
BEQ GMT
```

```
LDR R3, [R2, #Data]
```

```
STR R3, [R0]
```

```
LDR R3, [R2, #Next]
```

```
STR R3, [R1]
```

```
CMP R3, #NULL
```

```
BNE GCon
```

```
LDR R1, =PutPt
```

```
STR R3, [R1]
```

```
GCon MOV R0, R2
```

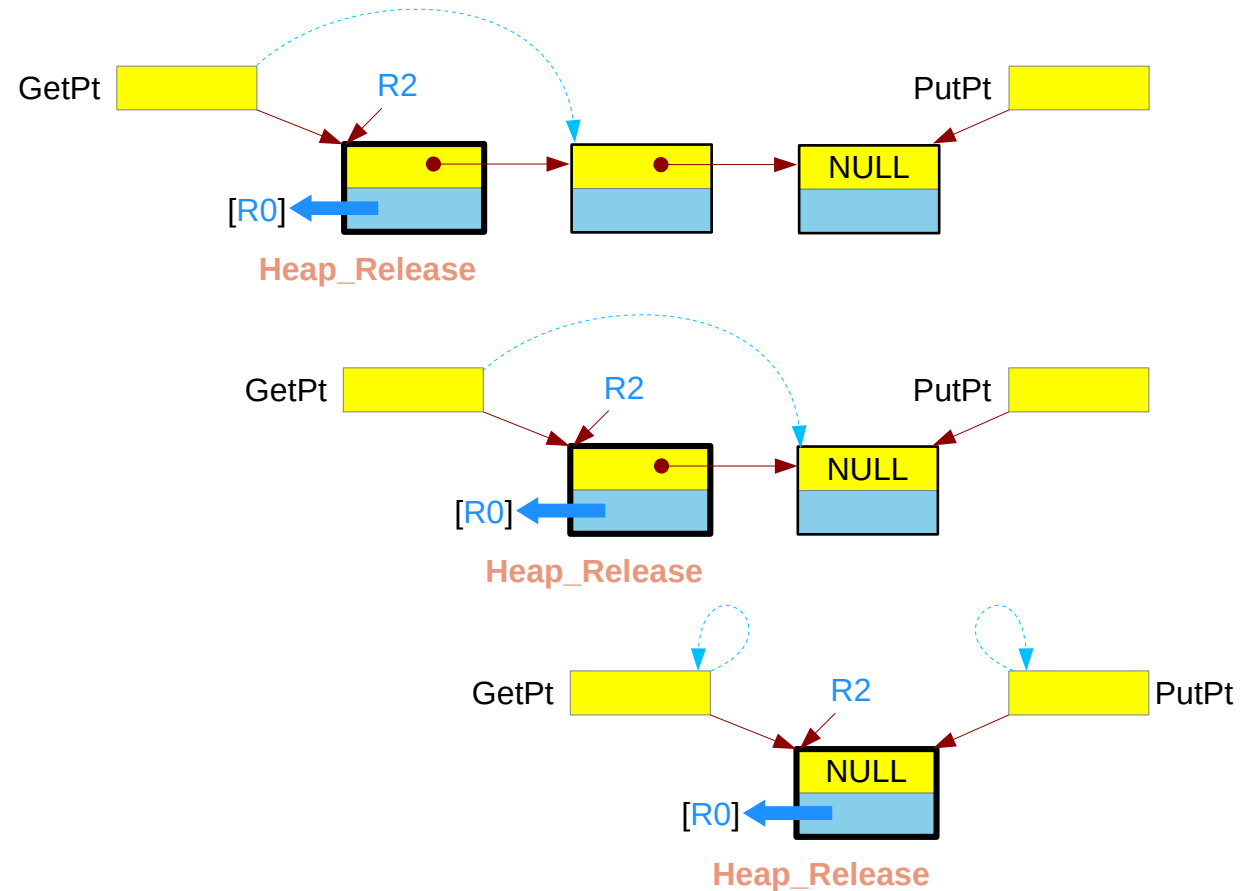
```
BL Heap_Release
```

```
MOV R0, #1
```

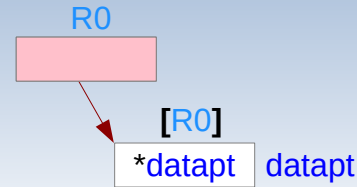
```
B GDon
```

```
GMT MOV R0, #0
```

```
GDon POP {R4, PC}
```



; Inputs: R0 points to an empty place
 ; Outputs: data removed to place
 R0=0 if successful
 R0=1 if empty



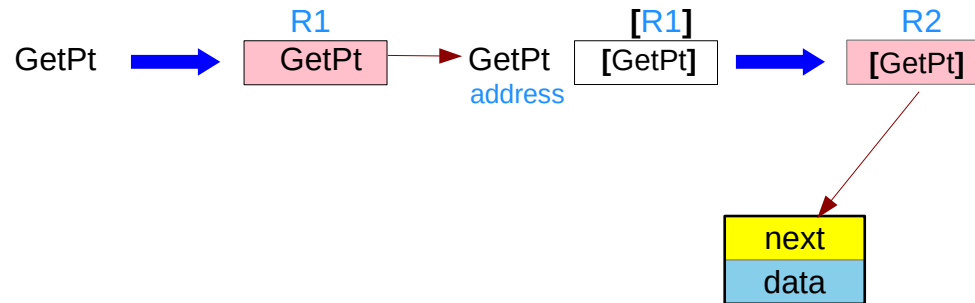
Fifo_Get

PUSH {R4, LR}

LDR R1, =GetPt

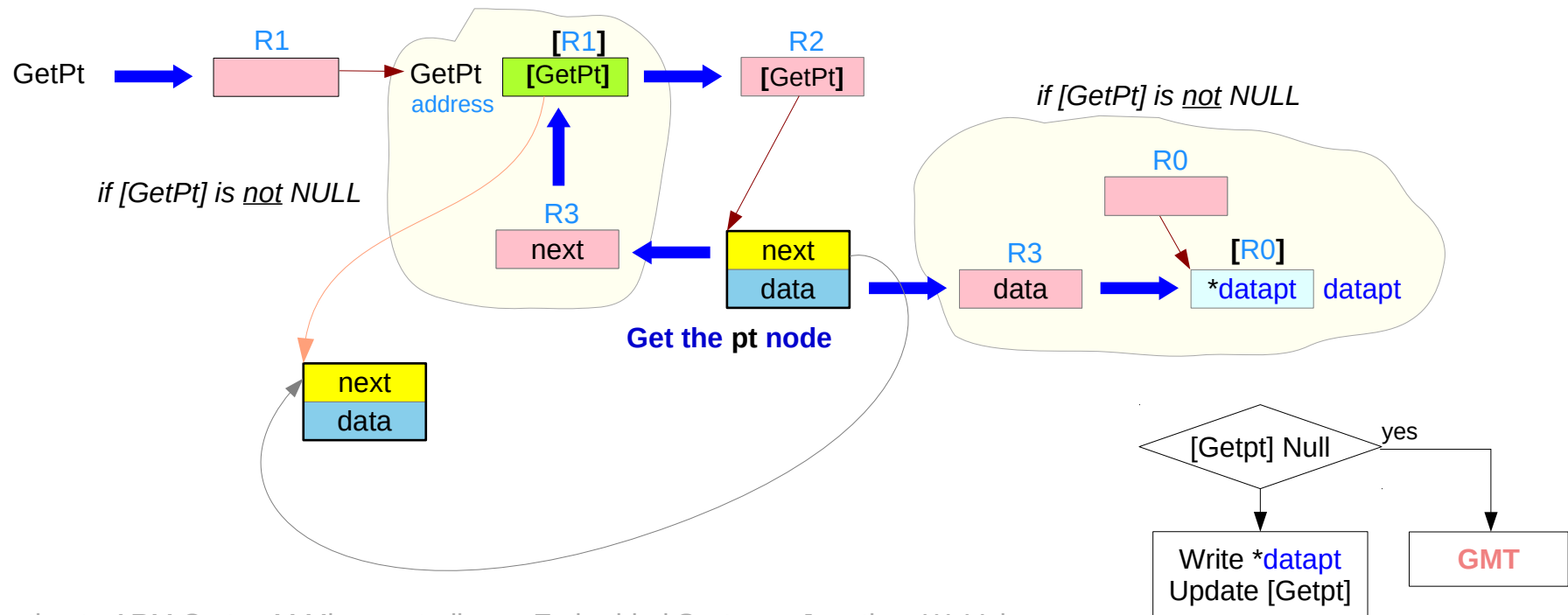
LDR R2, [R1]

; address GetPt ; R1 ← GetPt (address)
 ; Get node address ; R2 ← [GetPt] (content)



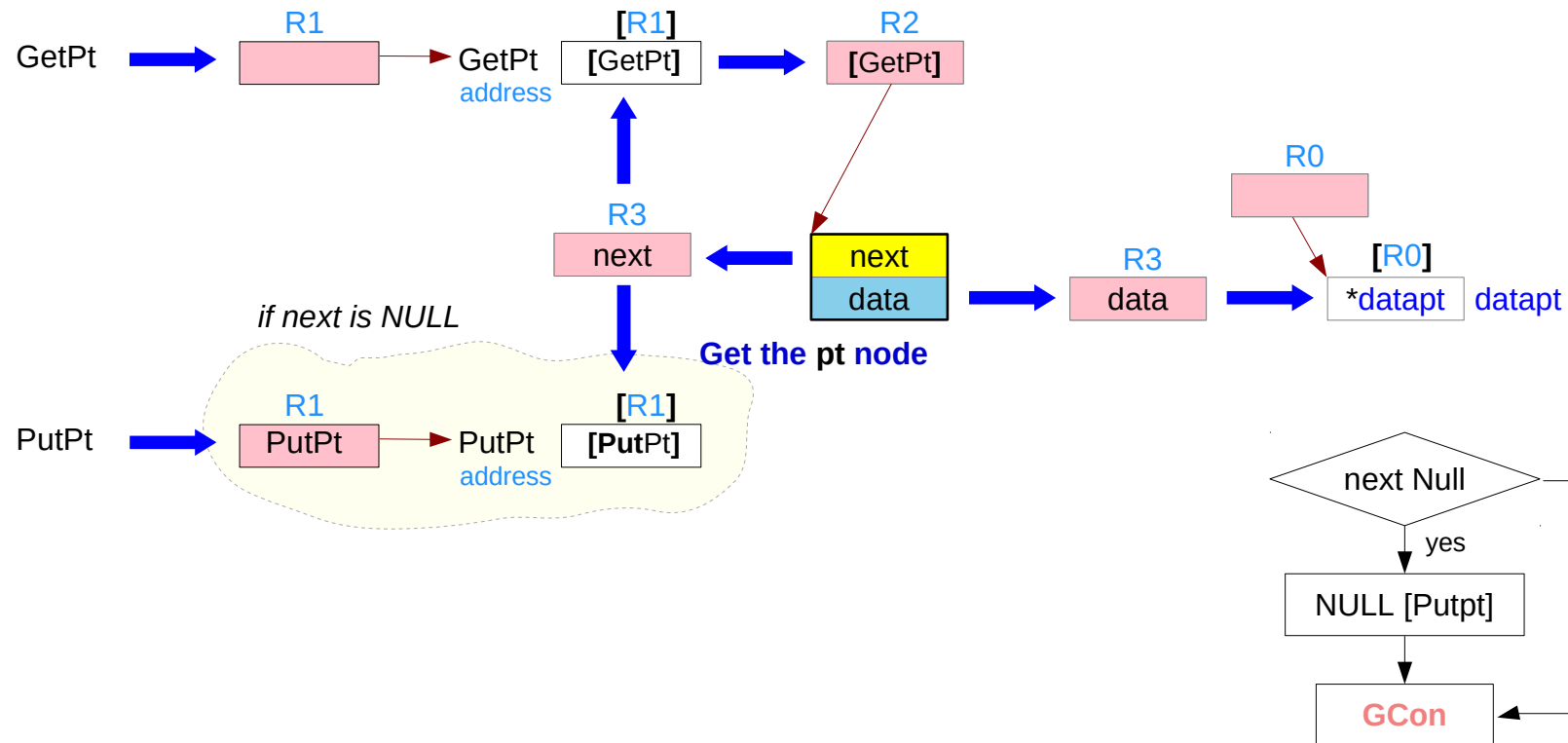
Get the pt node

CMP	R2, #NULL		
BEQ	GMT		; empty if NULL
LDR	R3, [R2, #Data]		; read 'data' ; R3 ← pt's data
STR	R3, [R0]		; store 'data' ; int Fifo_Get(int32_t *datapt)
LDR	R3, [R2, #Next]		; read 'next' ; R3 ← pt's next
STR	R3, [R1]		; store 'next' ; R3 → [GetPt]



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

CMP	R3, #NULL	; the last node	; pt node
BNE	GCon	; if not empty	; pt's next <> NULL
LDR	R1, =PutPt	; address PutPt	; R1 ← PutPt (address)
STR	R3, [R1]	; Put node address	; NULL → [PutPt] (content)

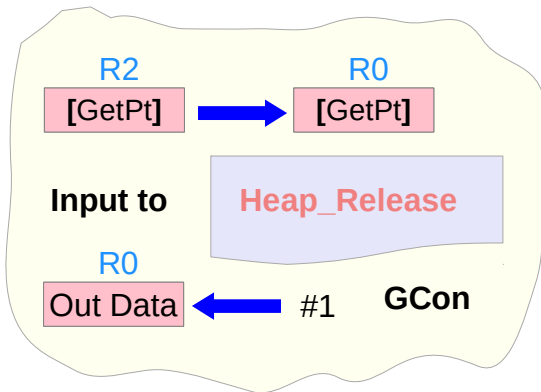



```

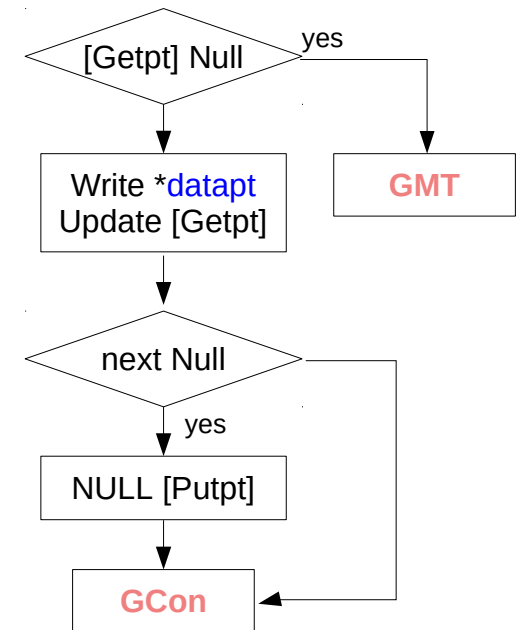
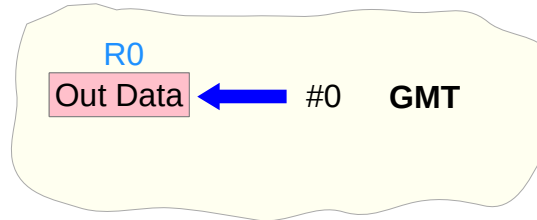
GCon      MOV    R0, R2      ; old data      ; R0 ← [GetPt] ; input arg
             BL     Heap_Release
             MOV    R0, #1    ; success
             B     GDon
GMT      MOV    R0, #0    ; failure, empty
GDon     POP    {R4, PC}
    
```



if [GetPt] is not NULL



if [GetPt] is NULL



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Matrix read

```
uint8_t  M[2][3];      // byte matrix with 2 rows and 3 columns

// Base + n*i + j

// Read an 8-bit value from (i, j)
// row or column major by compiler
// Input:  i is the row index
           j is the column index
// Output: retrieved value
// Assume: (0 <= i <= 1) and
//         (0 <= j <= 2)

uint8_t Matrix_read(uint8_t base[], uint8_t i, uint8_t j) {
    return base[i][j];
}
```

Accessing and storing matrices

Store in <u>row</u> major order	Store in <u>column</u> major order
<p>byte matrix uint8_t M[2][3]; Base + (3*i + j)</p>	<p>byte matrix uint8_t M[2][3]; Base + (2*j + i)</p>
<p>halfword matrix uint16_t M[2][3]; Base + 2*(3*i + j)</p>	<p>halfword matrix uint16_t M[2][3]; Base + 2*(2*j + i)</p>
<p>word matrix uint32_t M[2][3]; Base + 4*(3*i + j)</p>	<p>word matrix uint32_t M[2][3]; Base + 4*(2*j + i)</p>

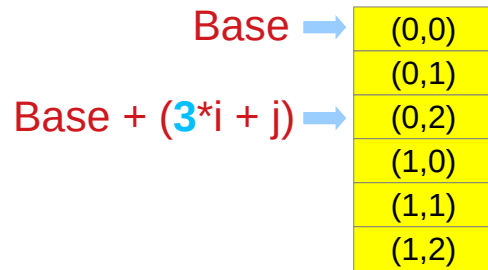
Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Matrix M[m][n] in row major order

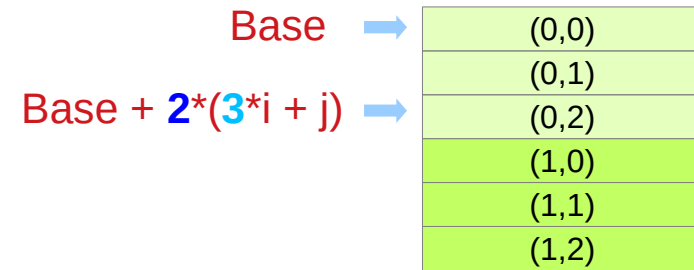
(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

byte matrix `uint8_t M[2][3];`

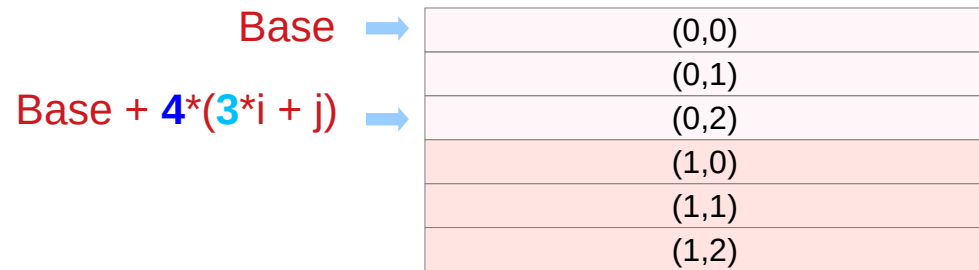


halfword matrix `uint16_t M[2][3];`



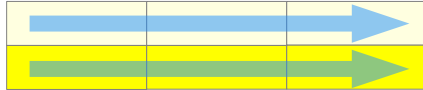
(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

word matrix `uint32_t M[2][3];`



row major order vs column major order

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

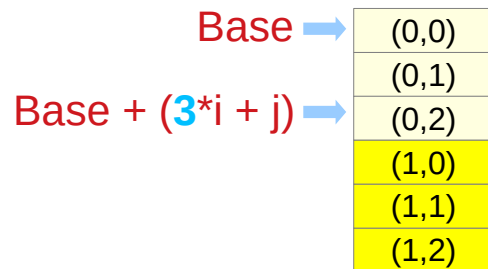


(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)



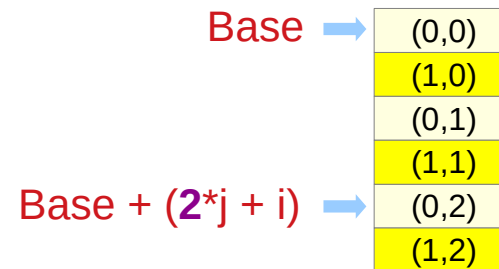
```
uint8_t M[2][3];
```

byte matrix – row major order



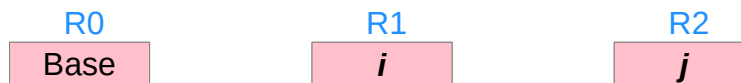
```
uint8_t M[2][3];
```

byte matrix – column major order



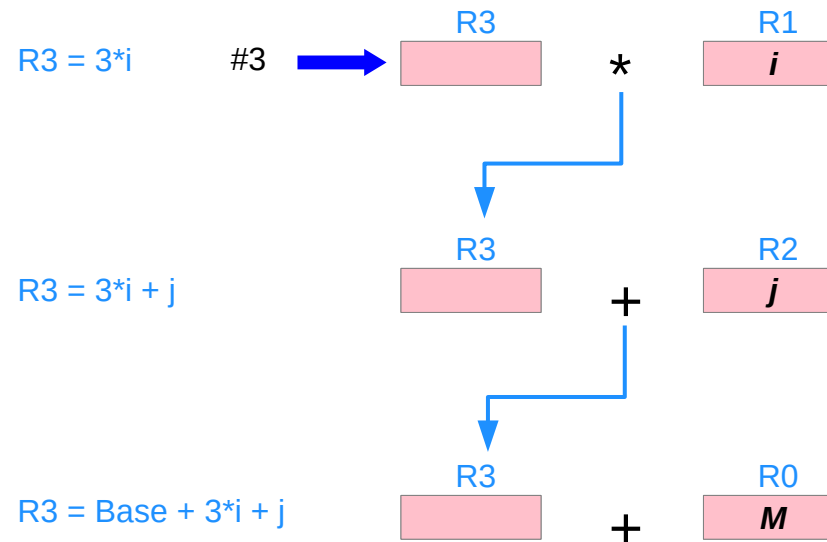
Matrix

```
; Read an 8-bit value from (i, j)                ;; Base + n*i + j
; Input:  Base    (R0)    pointer to matrix      ;; R0 + n*R1 + R2
;         i      (R1)    is the row index
;         j      (R2)    is the column index
; Assume: (0 <= R1 <= 1)
;         (0 <= R2 <= 2)
Matrix_read
    MOV     R3, #3                ; R3 = 3 columns M[2][3]
    MUL     R3, R3, R1           ; R3 = 3*i
    ADD     R3, R3, R2           ; R3 = 3*i + j
    ADD     R3, R3, R0           ; R3 = Base + 3*i + j
    LDRB   R0, [R3]             ; R0 = M[i, j]
    BX     LR                    ; return
```



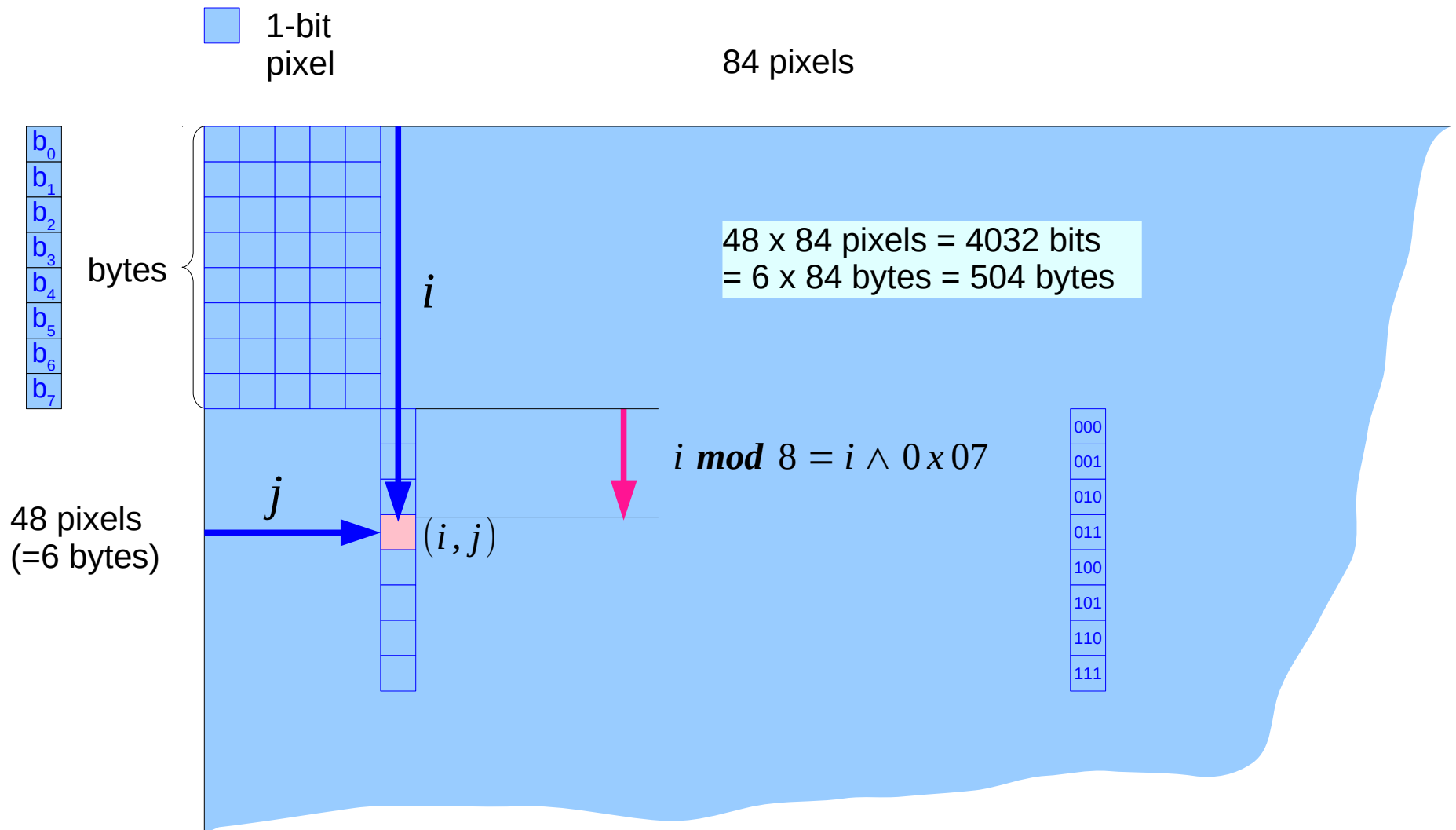
Matrix

```
Matrix_read
MOV R3, #3          ; R3 = 3 columns M[2][3]
MUL R3, R3, R1      ; R3 = 3*i
ADD R3, R3, R2      ; R3 = 3*i + j
ADD R3, R3, R0      ; R3 = Base + 3*i + j
LDRB R0, [R3]      ; R0 = M[i, j]
BX LR              ; return
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

48 x 84 B/W Image



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

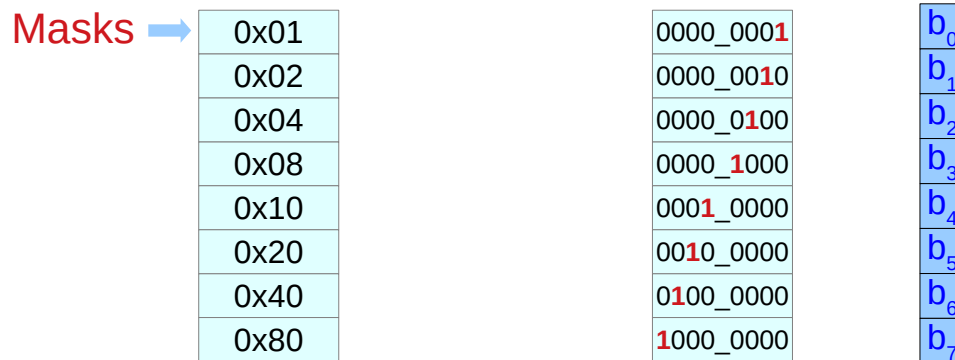
Matrix

```
uint8_t Screen[504]           // stores the next image to be printed on the screen

Screen + 84*(i>>3)           // Screen + 84*(i div 8)
Screen + 84*(i>>3) + j;      // Screen + 84*(i div 8) + j

k = i & 0x07;                // i mod 8

Masks FCB 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80
```



FCB Directive

fcf Form Constant Byte

```
(<label>) fcb <expr>(,<expr>,....,<expr>) (<comment>)  
(<label>) dc.b <expr>(,<expr>,....,<expr>) (<comment>)  
(<label>) db <expr>(,<expr>,....,<expr>) (<comment>)  
(<label>) .byte <expr>(,<expr>,....,<expr>) (<comment>)
```

The FCB directive may have one or more operands separated by commas. The value of each operand is truncated to eight bits, and is stored in a single byte of the object program. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case a single byte of zero will be assigned for that operand. An error will occur if the upper eight bits of the evaluated operands' values are not all ones or all zeros.

A string can be included, which is stored as a sequence of ASCII characters. The delimiters supported by TExaS are " ' and \. The string is not terminated, so the programmer must explicitly terminate it. For example:

```
str1 fcb "Hello World",0
```

<http://users.ece.utexas.edu/~valvano/assmbly/syntax.htm#fcb>

Matrix

; Input : R0 the row index i (0 to 47 in this case), Y-coordinate
; R1 the column index j (0 to 83 in this case), X-coordinate
; Output: R0 points to the byte of interest
R1 the Mask to access that 1-bit pixel

R0

i

R1

j

R2

$i \bmod 8$

Access

```
LDR    R3, =Masks
AND    R2, R0, #0x07
LDRB   R2, [R3, R2]
LDR    R3, =Screen
LSR    R0, R0, #3
MOV    R2, #84
MUL    R2, R0, R2
ADD    R0, R2, R3
ADD    R0, R0, R1
MOV    R1, R2
BX     LR
```

; Access the Image pixel at (i, j)

; R3 = Masks (pointer)

; R2 = $k = i \& 0x07$

; R2 = Mask[k]

; R3 = Screen (pointer)

; R0 = $R0 \gg 3$ ($i \text{ div } 8$)

; R2 = $84 * (i \text{ div } 8)$

; R0 = Screen + $84 * (i \text{ div } 8)$ (pointer)

; R0 = Screen + $84 * (i \text{ div } 8) + j$ (pointer)

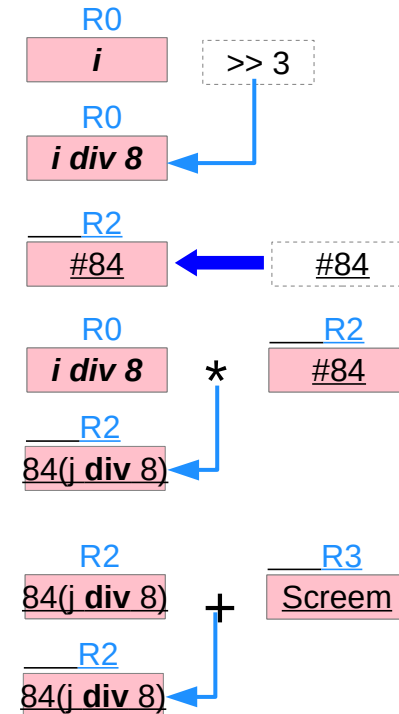
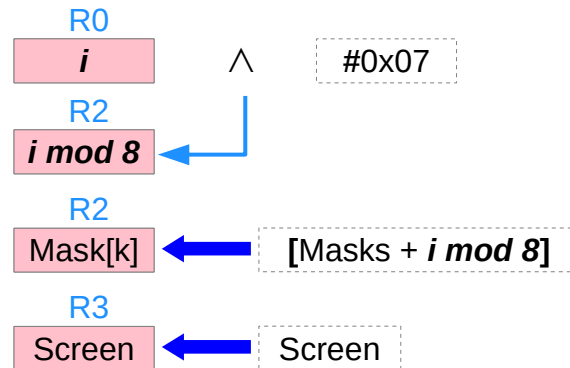
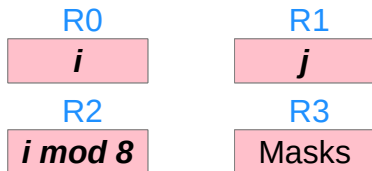
; R1 is Mask

Matrix

```

Access                                ; Access the Image pixel at (i, j)
LDR  R3, =Masks                       ; R3 = Masks (pointer)
AND  R2, R0, #0x07                    ; R2 = k = i & 0x07
LDRB R2, [R3, R2]                     ; R2 = Mask[k]
LDR  R3, =Screen                      ; R3 = Screen (pointer)
LSR  R0, R0, #3                       ; R0 = R0 >> 3 (i div 8)
MOV  R2, #84                          ; R2 = 84 * (i >> 3)
MUL  R2, R0, R2                        ; R2 = 84 * (i >> 3)
ADD  R0, R2, R3                        ; R0 = Screen + 84 * (i >> 3) (pointer)
ADD  R0, R0, R1                        ; R0 = Screen + 84 * (i >> 3) + j (pointer)
MOV  R1, R2                            ; R1 is Mask
BX   LR

```



Matrix

```
; Clear the Image pixel at (I, j), turning it dark
; Input:  R0 the row index I (0 to 47 in this case)      Y-coordinate
          R1 the column index j (0 to 83 in this case)  X-coordinate
; Output: none      modifies R0, R1, R2, R3
```

ClrPxl

```
    PUSH    {LR}
    BL      Access      ; get pointer to pixel to change
    LDRB   R3, [R0]     ; R3 = [R0] = read 8 pixels
    BIC    R3, R3, R1   ; R3 = R3 & ~R1    clear proper pixel
    STRB   R3, [R0]     ; [R0] = R3 = write 8 pixels
    POP    {PC}
```

Matrix

```
; Set the Image pixel at (I, j) to the given value
; Input:  R0 the row index I (0 to 47 in this case)      Y-coordinate
          R1 the column index j (0 to 83 in this case)  X-coordinate
; Output: none      modifies R0, R1, R2, R3
```

SetPxl

```
    PUSH    {LR}
    BL      Access      ; get pointer to pixel to change
    LDRB   R3, [R0]     ; R3 = [R0] = read 8 pixels
    ORR    R3, R3, R1   ; R3 = R3 | R1      set proper pixel
    STRB   R3, [R0]     ; [R0] = R3 = write 8 pixels
    POP    {PC}
```

Table

```
const int32_t lxTabl[22] = {
    0,13,26,38,51,64,77,90,102,
    115,128,141,154,166,179,192,205,
    218,230,243,255, 256};
```

```
const int32_t lyTabl[22] = {
    0,39,75,103,121,127,121,103,
    75,39,0,-39,-75,-103,-121,-127,
    -121,-103,-75,-39,0,0};
```

lxTabl	DCD	0,13,26,38,51,64,77,90,102
	DCD	115,128,141,154,166,179,192,205
	DCD	218,230,243,255, 256

lyTabl	DCD	0,39,75,103,121,127,121,103
	DCD	75,39,0,-39,-75,-103,-121,-127
	DCD	-121,-103,-75,-39,0,0

Table

```
// lx is 0 to 255 (pi/128)
// ly is -127 to +127 (1/256)

int32_t Sin(int32_t lx) {
    int32_t x1, x2, y1, y2;
    int i=0;

    while (lx >= lxTab(i+1)) {
        i++;
    }
    x1 = lxTab[i];
    x2 = lxTab[i+1];
    y1 = lyTab[i];
    y2 = lyTab[i+1];

    return ((y2-y1)*(lx-x1))/(x2-x1) + y1;
}
```


Table

; Input : R0 is 0 to 255, lx
; Output: R1 is -127 to +127, ly

Sin	PUSH	{R4-R6, LR}	
	LDR	R1, =lxTab	; find x1 <= lx < x2
	LDR	R2, =lyTab	
Lookx1	LDR	R6, [R1, #4]	; x2
	CMP	R0, R6	; check lx < x2
	BLO	found	; R1 >= x1
	ADD	R1, #4	
	ADD	R2, #4	
	B	lookx1	

Table

Found	LDR	R4, [R1]	; x1
	SUB	R4, R0, R4	; x-x1
	LDR	R5, [R2, #4]	; y2

Pointer access to an array

References

- [1] <ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf>
- [2] <https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf>