Gingl Zoltán, 2020, Szeged

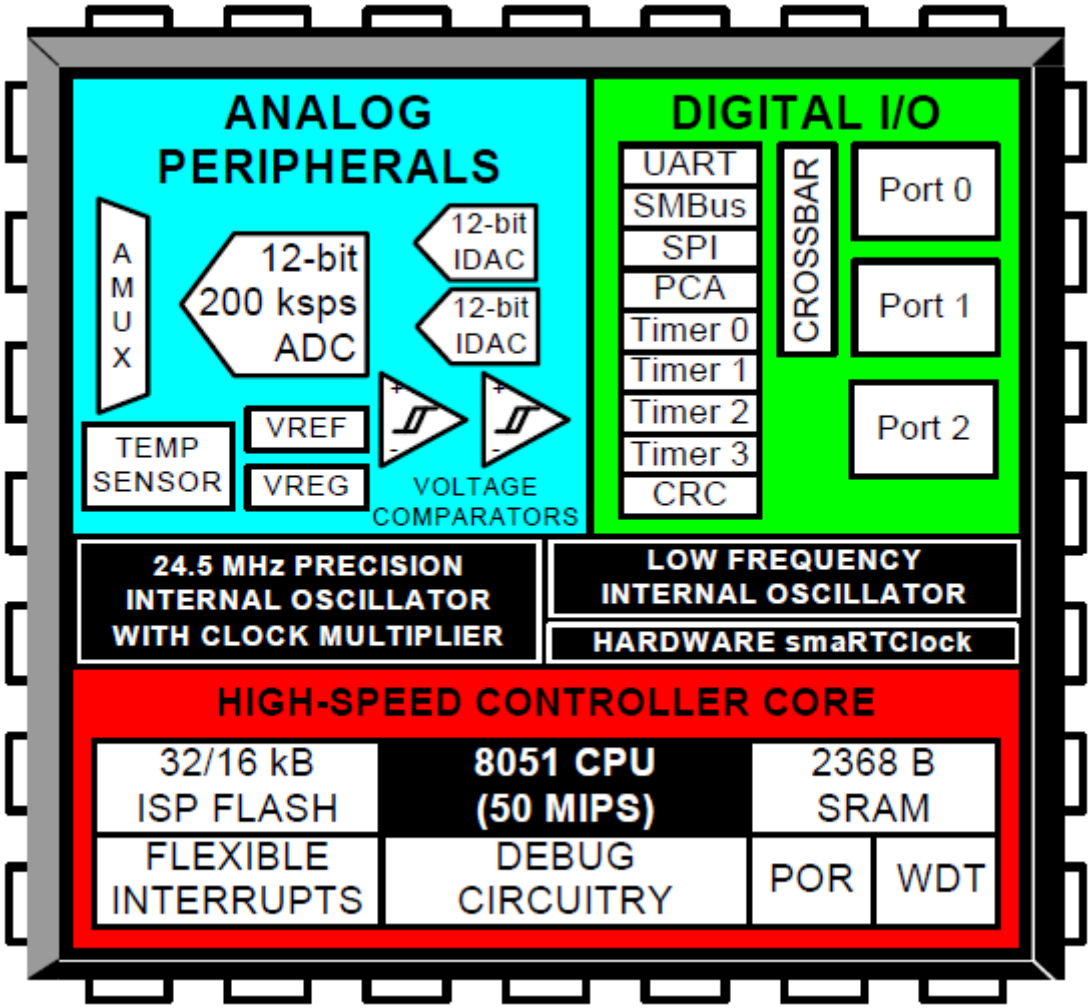
Mikrovezérlők Alkalmazástechnikája

A C8051Fxxx mikrovezérlők felépítése és programozása

Silicon Laboratories C8051Fxxx mikrovezérlők

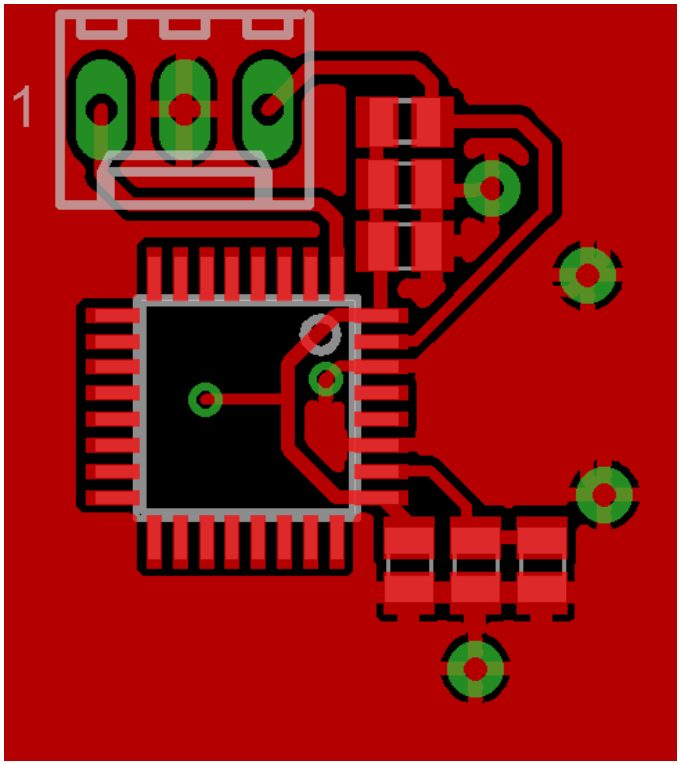
- Intel MCS-51 utasításkészlettel kompatibilis
- Módosított 8051 mag: CIP-51
  - 12 órajel/gépi ciklus → 1 órajel/gépi ciklus
  - 12MHz → akár 100MHz
  - on-chip debug
  - alacsony fogyasztású módok
  - sokkal több megszakítás
  - sokkal több periféria
  - sokkal jobb perifériák
- valódi önálló számítógép

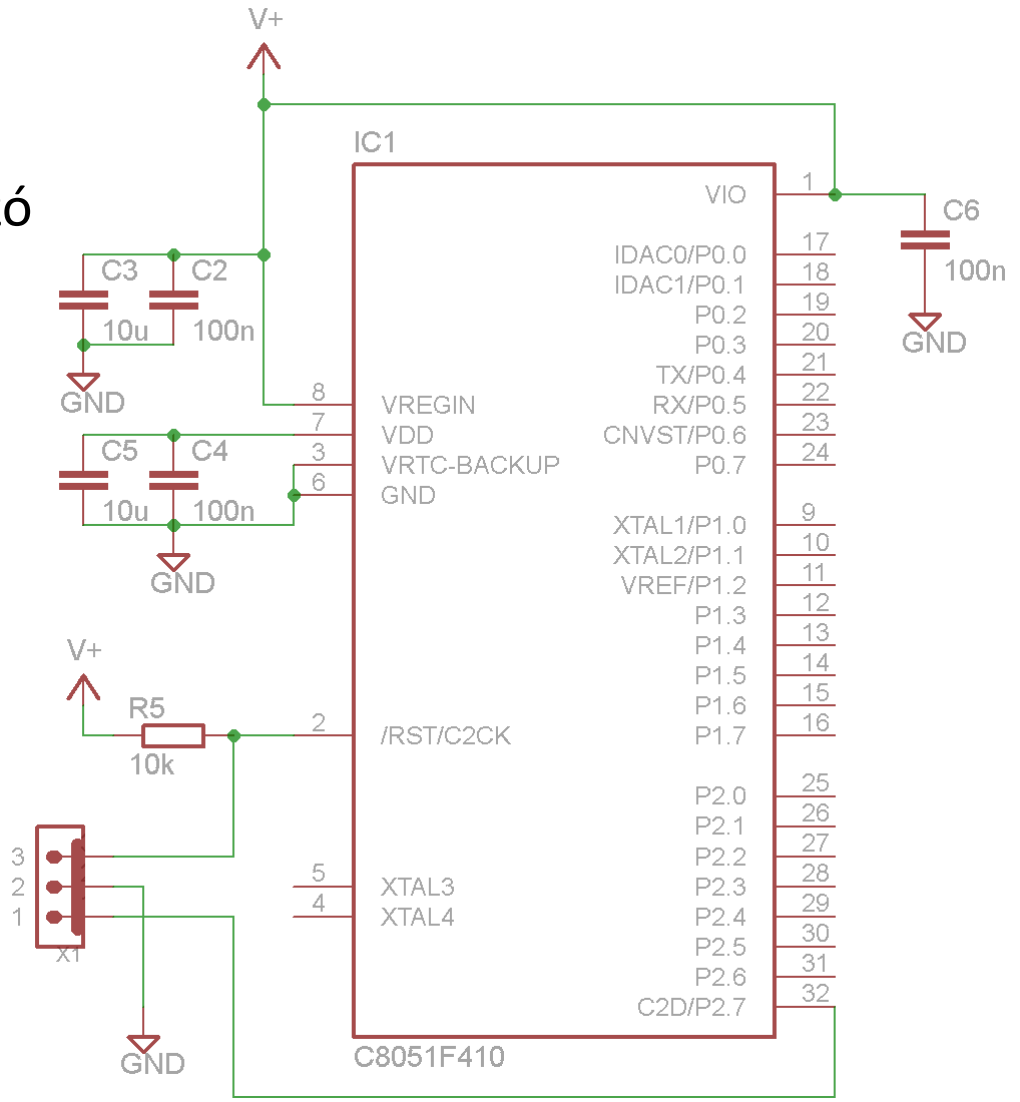
A C8051F410 felépítése



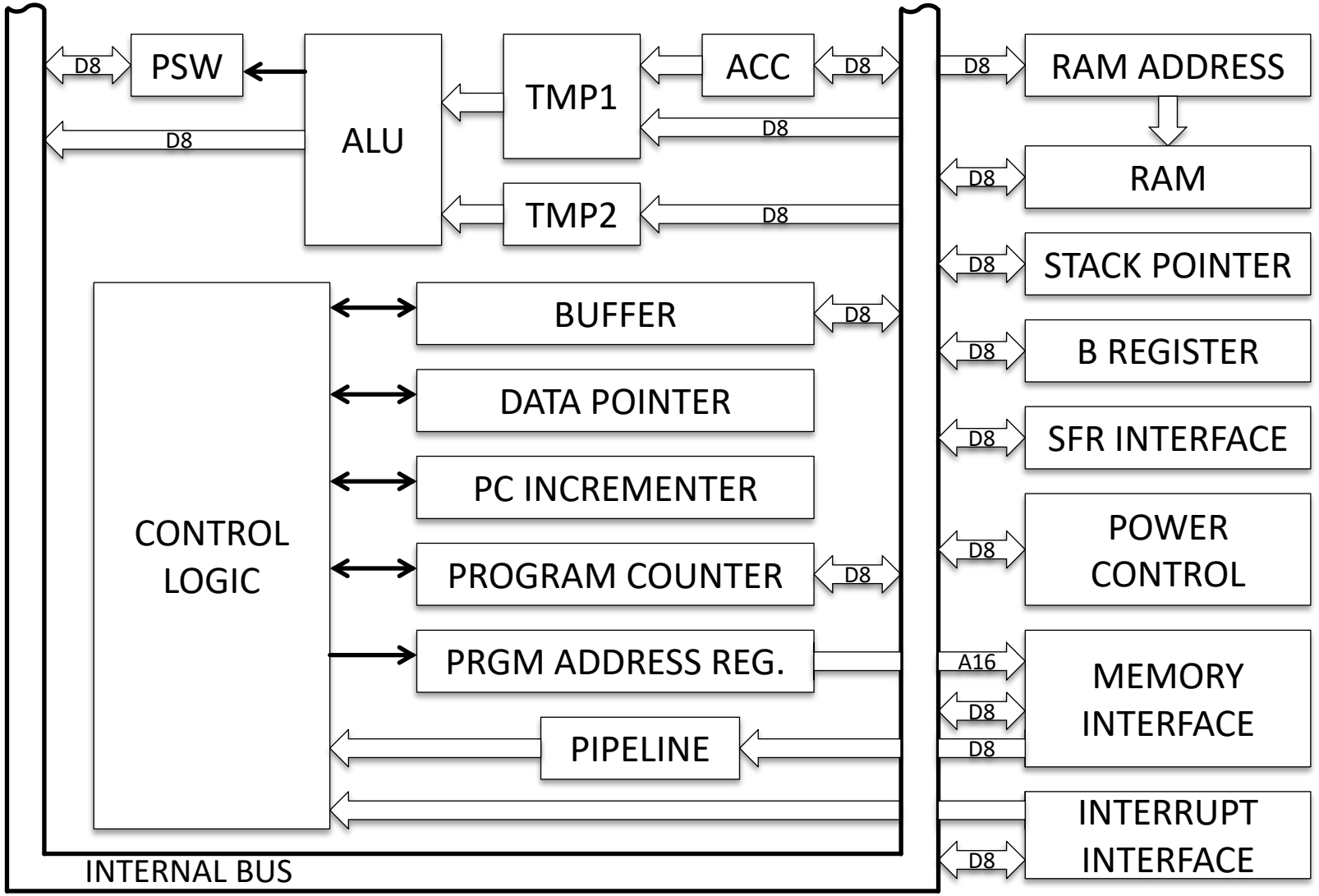
Silabs C8051F410 minimumkonfiguráció

- Tápfeszültség: 2V..5V
- Tápszűrés: kondenzátorok
- Programozó/debug csatlakozó
- A RESET vonalon R





A CIP-51 felépítése ▸ vezérlés



Regiszterek

Registers						Reset value
A, ACC	accumulator, ALU results					0
B	general purpose register and register for multiplication, division					0
R0..R7	general purpose registers, R0, R1 also used for indirect addressing					0
PSW	Bit 7: **CY**	carry bit (*ADDC, SUBB*)				0
	Bit 6: **AC**	aux carry (at 3rd bit, 4-bit arithmetics)				0
	Bit 5: **F0**	user flag				0
	Bit 4: **RS1**	R0..R7 at 00: 0x00	R0..R7 at 01: 0x08	R0..R7 at 10: 0x10	R0..R7 at 11: 0x18	0
	Bit 3: **RS0**					0
	Bit 2: **OV**	overflow (*MUL, DIV, ADD, SUBB*)				0
	Bit 1: **F1**	user flag				0
	Bit 0: **PAR**	parity bit: 1, if sum of bits in A is 1				0
DPH, DPL	**DPTR**, data pointer, 16-bit indirect addressing					0
SP	stack pointer					7

Memória

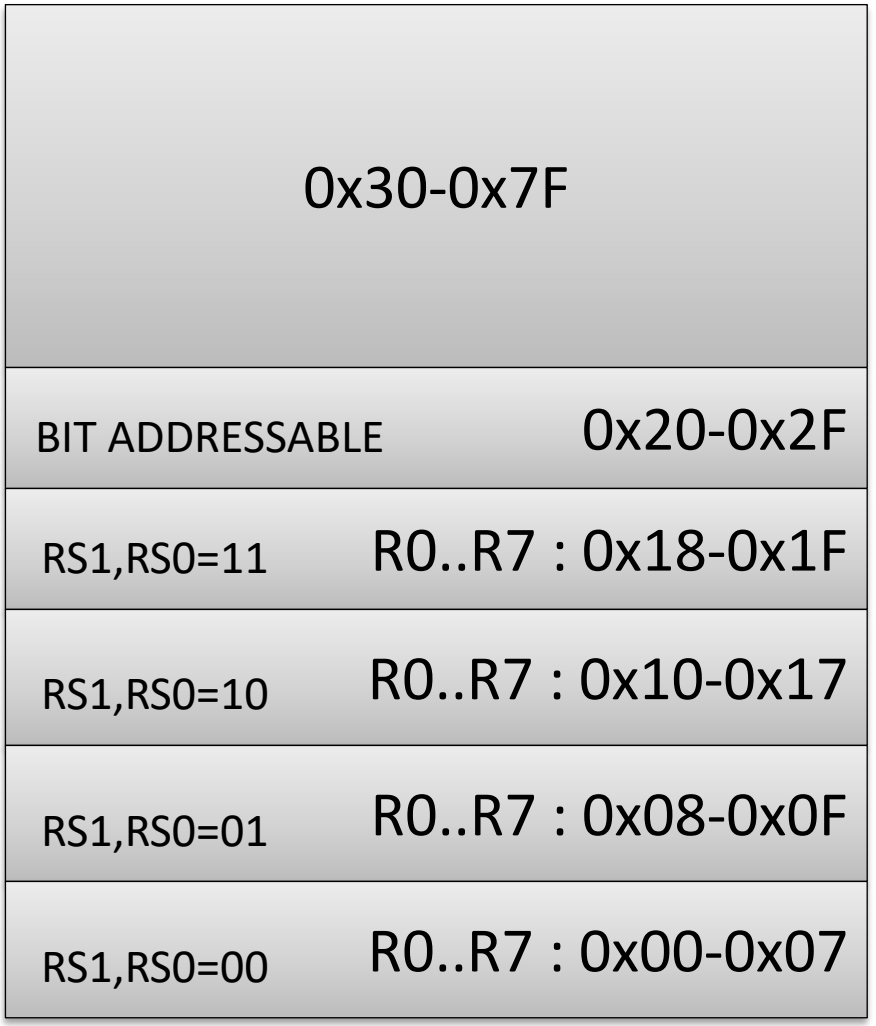
- Harvard architektúra
  - külön adat- és programmemória
  - a programmemória csak olvasható
  - a programmemória adatokat is tárolhat (csak olvasás)
- Adatamemória: RAM, XRAM
- Programmemória: flash
- Silicon Laboratories módosítás
  - a programmemória (flash) speciális módban írható

Memória ▸ Belső RAM

- Felső 128 byte
- Csak indirekt címzéssel
  - **MOV R0, #0B0h**
    MOV A, @R0
- Alsó 128 byte
- Direkt/indirekt címzéssel
  - **MOV A, 10**
    MOV A, 71h
    MOV 0B0h, a

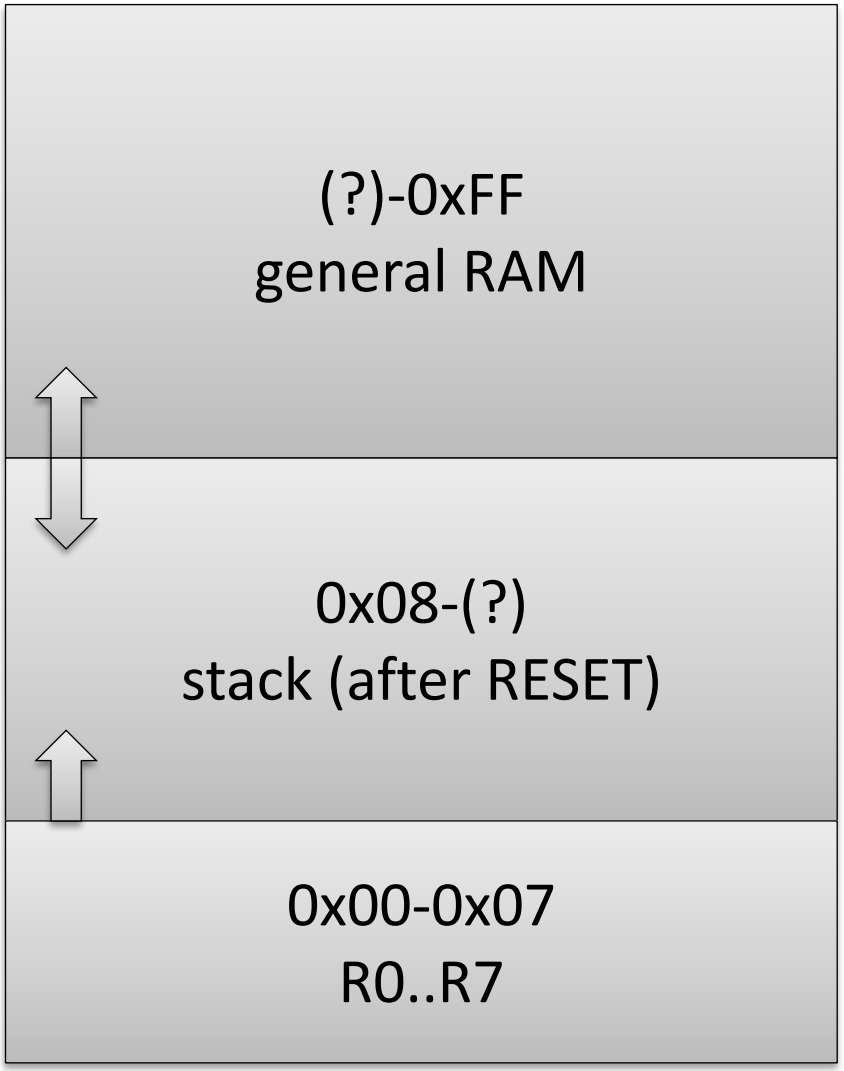
Memória ▸ Belső RAM ▸ alsó 128 byte

- 0x20-0x2F
  - 128 bit egyedileg elérhető
  - Bitváltozók számára
  - **MOV C,22.3h**
    carry ← a 22h című byte 3. bitje
- 0x00-0x1F
  - Az R0..R7 regiszterbank helye változtatható (PSW: RS1,RS0)
  - Figyelembe kell vennünk a programozásnál
  - Hasznos a regiszterek tartalmának őrzéséhez
  - Tipikusan interrupt rutinokban



Memória ▸ Belső RAM ▸ Verem

- **RESET** után:
  - a veremmutató értéke 0x07
  - R0..R7: 0x00-0x07
- A veremmutató növekszik, ha használjuk
- A veremmutató átmehet a felső 128 byte-ra is!
- Szokásos stílus: boot után SP=adatok utáni cím
- A verem méretét a programozó dönti el!



Memória ▸ SFR (special function registers)

- Az integrált perifériák, regiszterek elérése:
  - „memory mapping”
  - memória írás/olvasás műveletekkel érhetők el
  - **direkt címzés a 0x80-0xFF belső RAM területen**
    - **ADD A,240** ; B regiszter SFR elérése
    - **MOV A,PSW** ; PSW elérése
- Az **ACC, B, DPH, DPL, SP** regiszterek is elérhetők SFR regiszterként

Néhány SFR – C8051F410

Address	0	1	2	3	4	5	6	7
0xB8	IP							
0xB0								
0xA8	IE							
0xA0	P2							
0x98	SCON	SBUF						
0x90	P1							
0x88	TCON	TMOD	TL0	TH0	TL1	TH1		
0x80	P0	SP	DPL	DPH				

Column 0 is bit addressable

P0, P1, P2: általános célú 8 bites port input/output (GPIO)

SCON, SBUF: soros kommunikációs kontrol és adatregiszterek

TCON, TMOD, TL0, TH0,TL1,TH1: számláló időzítő regiszterek

Néhány SFR

Cím	0	1	2	3	4	5	6	7
0xF8								
0xF0	**B**							
0xE8								
0xE0	**ACC**							
0xD8								
0xD0	**PSW**							
0xC8								
0xC0								

Az 0. oszlop bitcímezhető

Memória ▸ Külső RAM (XRAM)

- 16-bit címtartomány
- maximum 64kbyte
- eredeti 8051:
  XRAM külön chipen
- mai 8051
  architektúrákon:
  - **integrált** (512 byte..8kbyte)
  - **integrált+külső**

- Elérés: **movx** (indirekt)
- 16-bites cím
  - **MOV DPTR,#0A000H**
    MOVX A,@DPTR
- 8-bites cím
  - **MOV R0,#10H**
    MOVX A,@R0
  - felső 8-bit egy SFR-ben
    (F410: EMI0CN)

Memória ▸ Külső RAM (XRAM) ▸ off-chip

- Külön chip-en van a RAM
- SRAM áramkörök csatlakoztathatók
- Lehet sebességkorlát (pl. 100ns elérési idő)
- Lehet akár külső periféria is, ami SRAM-szerű:
  - ADC, DAC, UART, …
- Később részletezzük a hardver működését

Memória ▸ Programmemória (flash)

- Program tárolására
- Régebben külső chip (UV-EPROM, EEPROM)
- Ma: flash, áramkörben is programozható
- Programozó adapterrel írható
- Flash: tápfeszültség nélkül megőrzi tartalmát
- Endurance: hányszor írható (≈10k-100k)
- Data retention: az adatokat meddig őrzi (≈20-100 év)
- Biztonsági bitek: titkosítás lehetséges
  a flash olvasása tiltható (törölhetőség)

Memória ▸ Programmemória ▸ adatok

- Programot és konstansokat is tárol
- Például:
  - **ADD A,#10** ; 10-et ad a-hot, a 10 konstans
  - **MOV A,PSW** ; a PSW egy SFR cím, 0D0h, konstans
- Közvetlenül is elérhető:
  - **CLR A**
    MOV DPTR,#LOOKUPTABLE
    MOVC A,@A+DPTR ; lookuptable+a címről olvas
- Hasznos lookup táblázatok, egyéb paraméterek tárolására

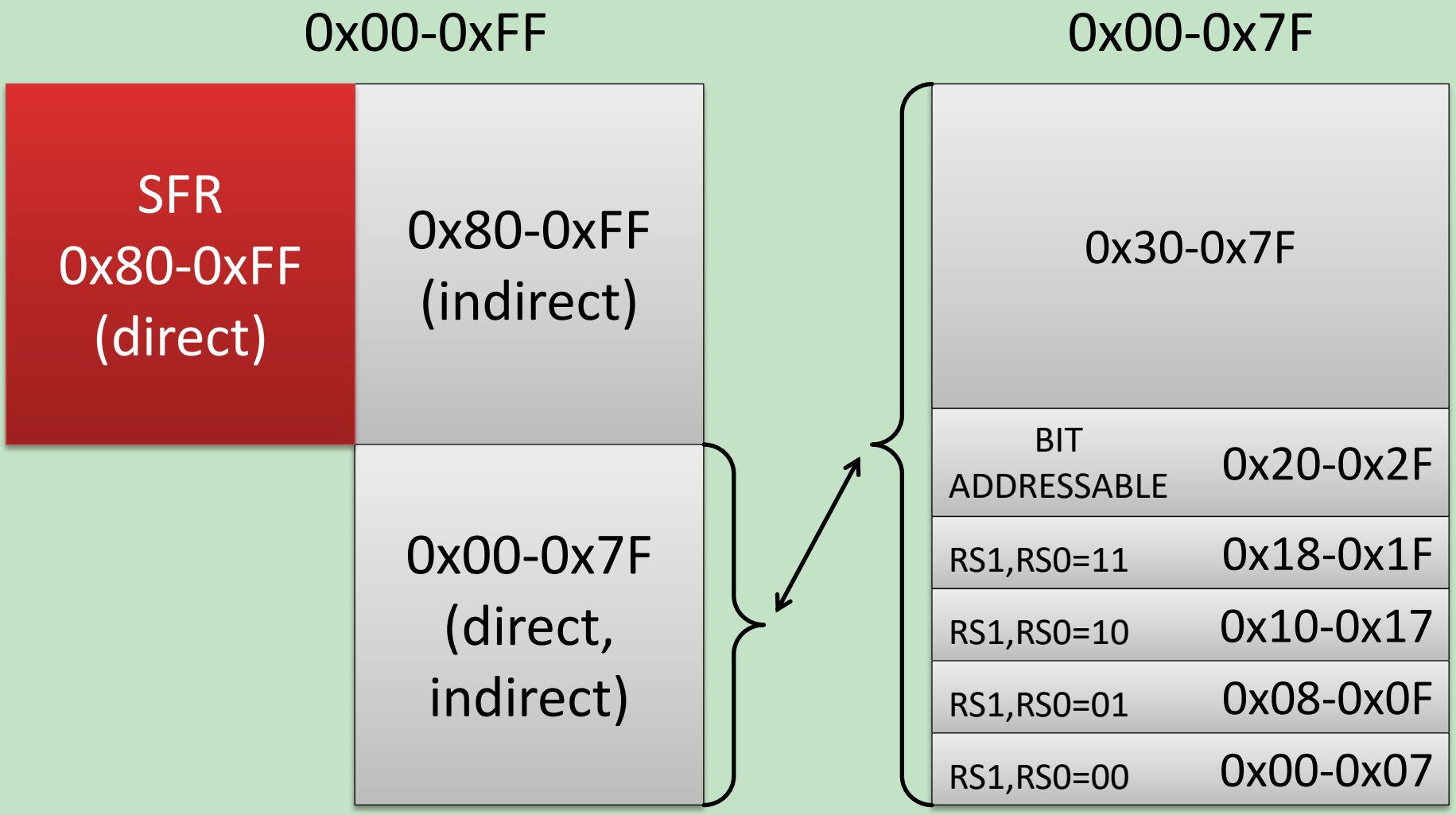
Memória ▸ Programmemória ▸ kód

- A CIP-51 mag olvassa (PC – program counter)
- Kiosztás:
  - 0000h: RESET-kor ide kerül a vezérlés
  - 0003h: 0. megszakítás címe
  - 000Bh: 1. megszakítás címe
  - …
- A tényleges program az interrupt tábla után van
- A fordító és a linker osztja el
  - assemblernél különösen figyelnünk kell!

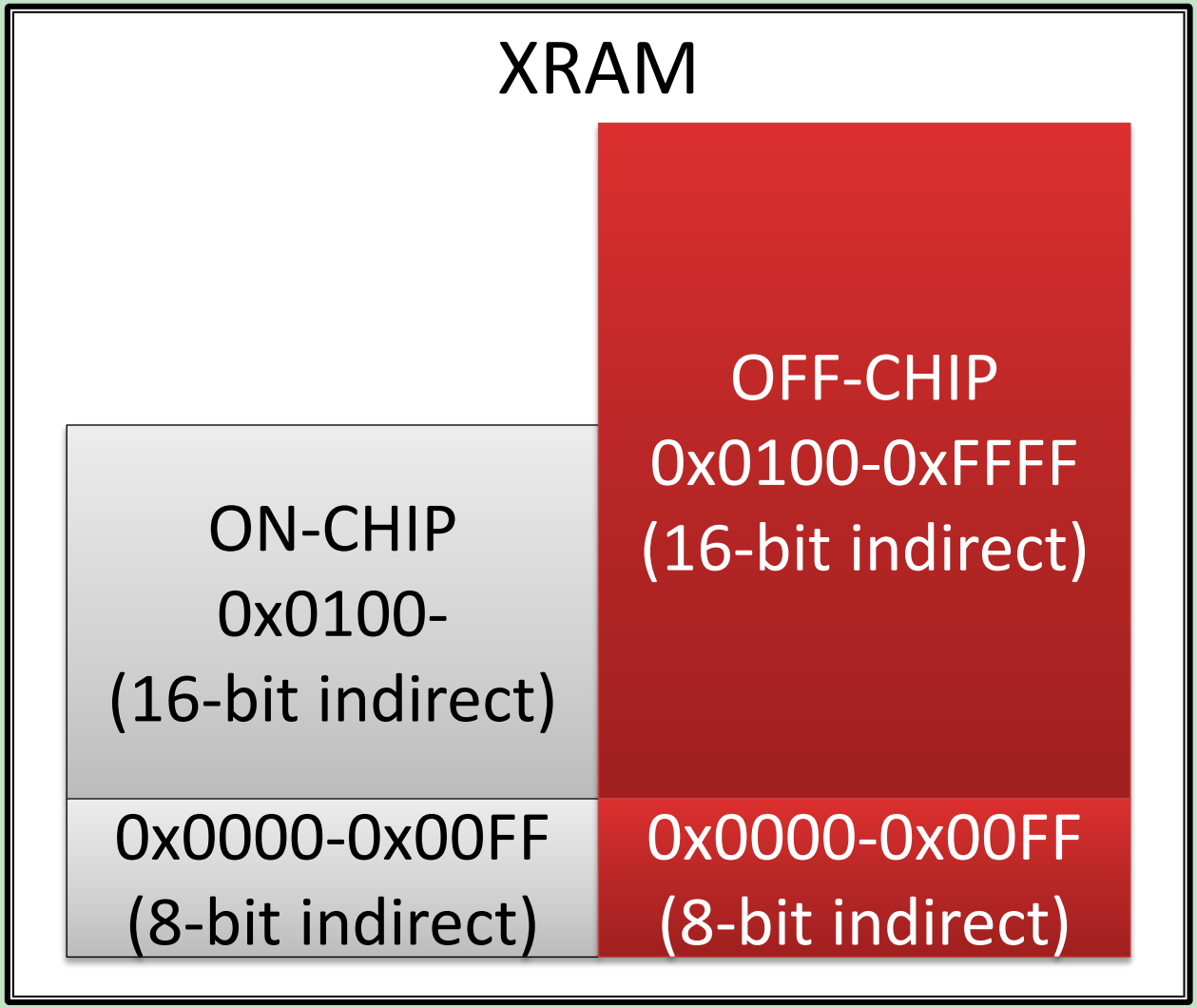
Memória ▸ Címzési módok

Addressing mode	MNEMONIC	Description
register	**MOV A,B**	A ← B
immediate constant	**MOV A, #10**	A ← 10 (value)
direct	**MOV A, 10** **MOV A, P0**	A ← byte at address 10 A ← bits at port P0 (SFR)
indirect	**MOV A, @R0**	A ← byte at address pointed by R0

Memória ▸ Összefoglaló ▸ RAM

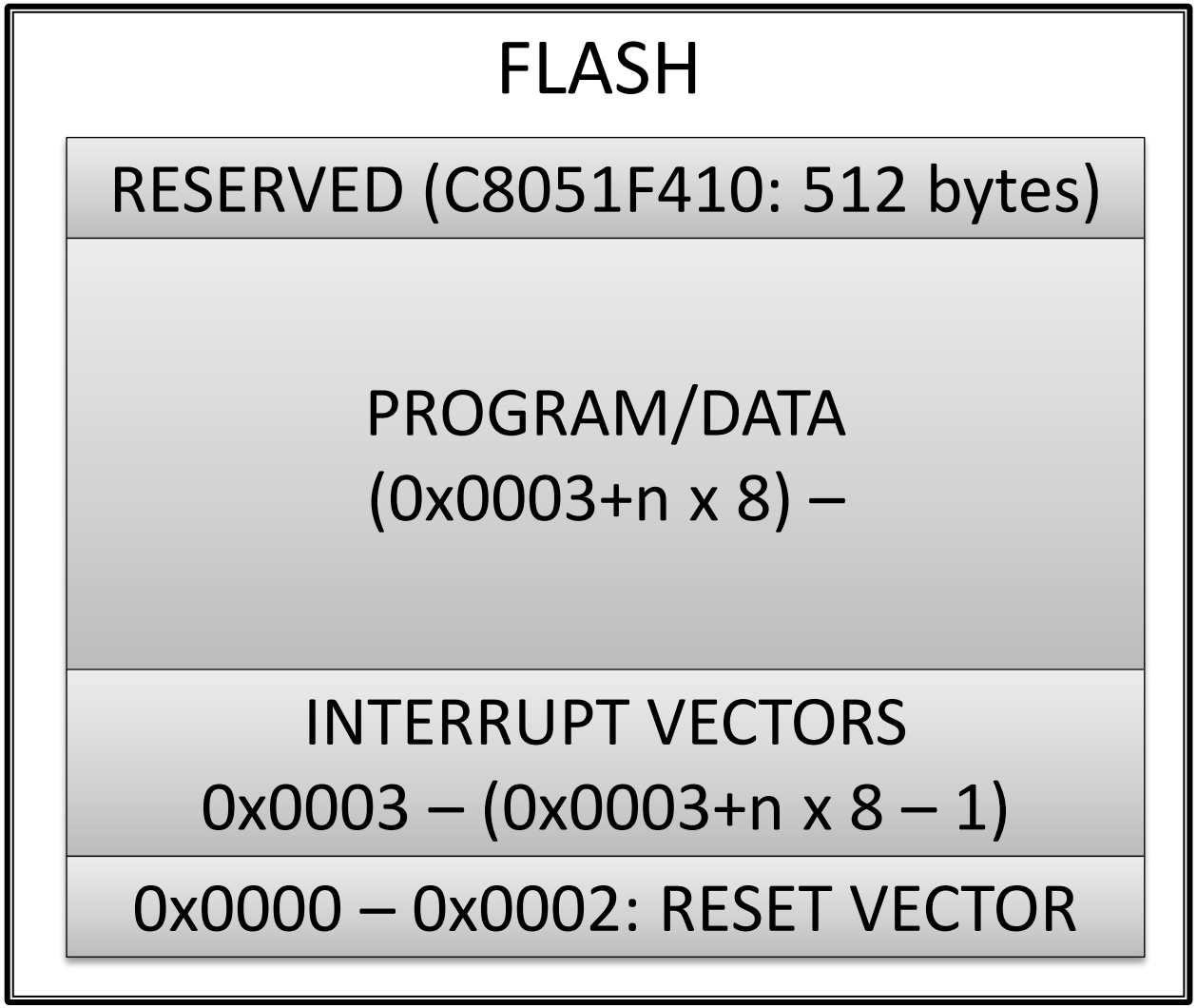


Memória ▸ Összefoglaló ▸ XRAM



Memória ▸ Összefoglaló ▸ ROM/FLASH

n: a legnagyobb használt megszakítás sorszáma



Utasításkészlet ▸ Aritmetikai utasítások

MNEMONIC	OPERATION	ADDRESSING				FLAG			
		DIR	IND	REG	IMM	CY	AC	OV	P
ADD A,<byte>	A = A + <byte>	√	√	√	√	√	√	√	
ADDC A,<byte>	A = A + <byte> + C	√	√	√	√	√	√	√	
SUBB A,<byte>	A = A – <byte> – C	√	√	√	√	√	√	√	
INC A	A = A + 1								
INC <byte>	<byte> = <byte> + 1	√	√	√					
INC DPTR	DPTR = DPTR + 1	only DPTR							
DEC A	A = A – 1	only A							
DEC <byte>	<byte> = <byte> – 1	√	√	√					
MUL AB	B:A = B x A	only A és B				0		√	
DIV AB	A = Int[A/B] B = Mod[A/B]	only A és B				0		√	
DA A	Decimal Adjust	only A				√			

Utasításkészlet ▸ Logikai utasítások

MNEMONIC	OPERATION	ADDRESSING				FLAG
		DIR	IND	REG	IMM	P
ANL A,<byte>	A = A & <byte>	√	√	√	√	√
ANL <byte>,A	<byte> = <byte> & A	√				
ANL <byte>,#data	<byte> = <byte> & #data	√				
ORL A,<byte>	A = A | <byte>	√	√	√	√	√
ORL <byte>,A	<byte> = <byte> | A	√				
ORL <byte>,#data	<byte> = <byte> | #data	√				
XRL A,<byte>	A = A ^ <byte>	√	√	√	√	√
XRL <byte>,A	<byte> = <byte> ^ A	√				
XRL <byte>,#data	<byte> = <byte> ^ #data	√				

Utasításkészlet ▸ Logikai utasítások

MNEMONIC	OPERATION	ADDRESSING	FLAG			
			CY	AC	OV	P
CRL A	A = 00H	only A				√
CPL A	A = ~A	only A				√
RL A	Rotate ACC Left 1 bit	only A				√
RLC A	Rotate Left through Carry	only A				√
RR A	Rotate ACC Right 1 bit	only A				√
RRC A	Rotate Right through Carry	only A				√
SWAP A	Swap Nibbles in A	only A				√

Utasításkészlet ▸ Bit logikai utasítások

MNEMONIC	OPERATION
ANL C,bit	C = C & bit
ANL C,/bit	C = C & !bit
ORL C,bit	C = C | bit
ORL C,/bit	C = C | !bit
MOV C,bit	C = bit
MOV bit,C	bit = C
CLR C	C = 0
CLR bit	bit = 0
SETB C	C = 1
SETB bit	bit = 1
CPL C	C = !C
CPL bit	bit = !bit

Utasításkészlet ▸ Adatmozgató utasítások

MNEMONIC	OPERATION	ADDRESSING			
		DIR	IND	REG	IMM
MOV A,<src>	A = <src>	√	√	√	√
MOV <dest>,A	<dest> = A	√	√	√	
MOV <dest>,<src>	<dest> = <src>	√	√	√	√
MOV DPTR,#data16	DPTR = 16-bit immediate constant				√
PUSH <src>	INC SP:MOV“@SP”,<src>	√			
POP <dest>	MOV <dest>,“@SP”:DEC SP	√			
XCH A,<byte>	ACC and <byte> exchange data	√	√	√	
XCHD A,@Ri	ACC and @Ri exchange low nibbles		√		

Utasításkészlet ▸ Adatmozgató utasítások

MNEMONIC	OPERATION
MOVX A,@Ri	A ← XRAM @Ri
MOVX @Ri,A	XRAM @Ri ← A
MOVX A,@DPTR	A ← XRAM @DPTR
MOVX @DPTR,A	XRAM @DPTR ← A
MOVC A,@A+DPTR	A ← code @(A + DPTR)
MOVC A,@A+PC	A ← code @(A + PC)

Utasításkészlet ▸ Feltétel nélküli ugrás

MNEMONIC	OPERATION
JMP <addr>	Jump to <addr> PC ← <addr>
JMP @A+DPTR	Jump to A + DPTR PC ← A + DPTR
ACALL <addr>	Call subroutine at 11-bit <addr> @(SP, SP-1) ← PC+2 SP ← SP+2 PC ← <addr>
LCALL <addr>	Call subroutine at 16-bit <addr> @(SP, SP-1) ← PC+3 SP ← SP+2 PC ← <addr>

Utasításkészlet ▸ Feltétel nélküli ugrás

MNEMONIC	OPERATION
RET	Return from subroutine PC ← @(SP, SP-1) SP ← SP-2
RETI	Return from interrupt PC ← @(SP, SP-1) SP ← SP-2 az interrupt logika visszaállítása
NOP	No operation

Utasításkészlet ▸ Feltételes ugrás

MNEMONIC	OPERATION	ADDRESSING			
		DIR	IND	REG	IMM
JZ rel	Jump if A = 0	only A			
JNZ rel	Jump if A !=0	only A			
DJNZ <byte>,rel	Decrement and jump if not zero	√		√	
CJNE A,<byte>,rel	Jump if A p <byte>	√			√
CJNE <byte>,#data,rel	Jump if <byte> p #data		√	√	
JC rel	Jump if C = 1				
JNC rel	Jump if C = 0				
JB bit,rel	Jump if bit = 1				
JNB bit,rel	Jump if bit = 0				
JBC bit,rel	Jump if bit = 1; CLR bit				

Utasításkészlet ▸ Utasítások kódolása

- Utasítások: 1-3 byte hossz, 1-8 ciklus

cycles	1	2	2/4	3	3/5	4	5	4/6	6	8
instructions	26	50	5	10	7	5	2	1	2	1

Példák

instruction	1. byte	2. byte	3. byte	cycles
ADD A, Rn	0010 1nnn			1
ADD A, #10	0010 0100	0000 1010		2
ANL 15,#10	0101 0011	0000 1111	0000 1010	3
DIV AB	1000 0100			8
JZ <rel address>	0110 0000	rel address		2/4

Megszakítások

Megszakítások (interrupt)

- Előre nem ismert időben bekövetkező esemény beavatkozást igényel
- Belső áramkörök vagy külső események
  - gombnyomás
  - számláló túlcsordul
  - adat érkezett egy külső eszköztől
  - …
- Esemény kezelése: alprogram (szubrutin)
- A hardver hívja meg (interrupt controller)
- Többféle megszakítás lehet, mindhez más rutin

Megszakítások ▸ Mechanizmus

- A főprogram fut, bárhol előfordulhat megszakítás
- Megszakításkor:
  - a főprogram éppen folyamatban levő művelete lefut
  - a vezérlés átkerül a megfelelő megszakítási rutinra
  - a megszakítást kiszolgáló rutin lefut
  - a vezérlés visszakerül a főprogram következő utasítására

Megszakítások ▸ Mechanizmus

- A megszakítási rutin is a regisztereket, RAM-ot használja
  - Szükség esetén el kell menteni a rutin elején
  - vissza kell állítani a rutin végén
- A megszakítások tilthatók/engedélyezhetők
  - EA: globális engedélyezés/tiltás bit
  - Minden megszakítási forráshoz egyedi engedélyező bit

Megszakítások ▸ Flag és kezelése

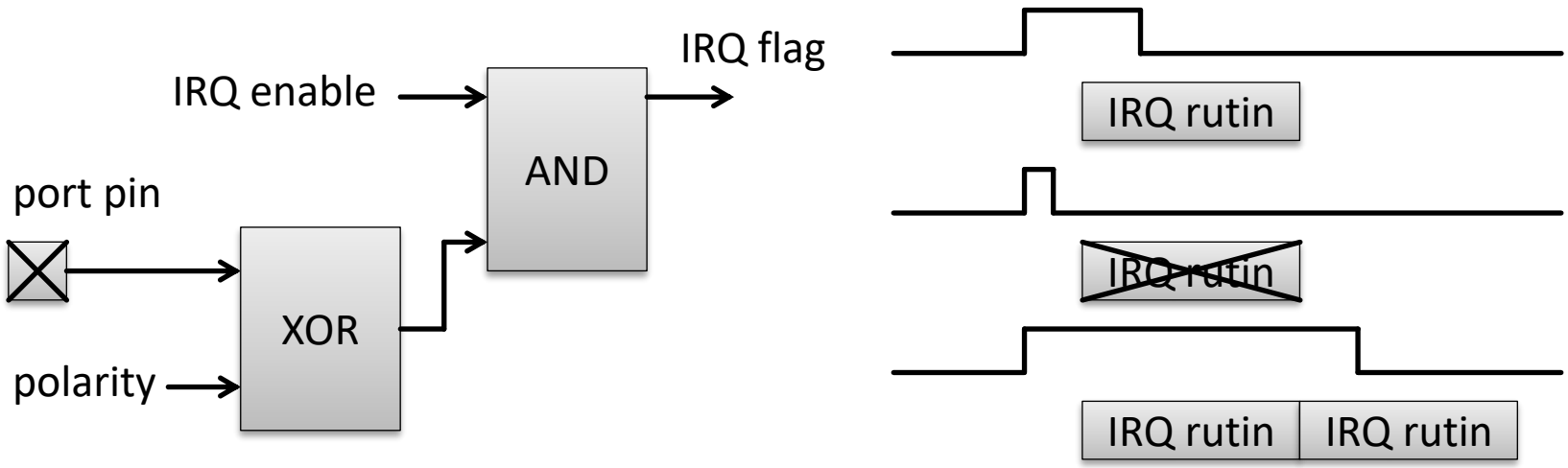
- **Megszakítási kérelemkor az eseményhez tartozó flag 1-re vált**
- A megszakítási rutin elején a programnak törölnie kell (néhány automatikusan törlődik)
- **Ha nem töröljük, a rutin befejezésekor újra megszakítás történik – tipikus hiba!**
- A flag bármikor lekérdezhető szoftveresen (polling)
  - akkor is, ha nincs engedélyezve a megszakítás
- A flag akár beállítható is, ez eseményt szimulál!
  - hasznos a program tesztelésekor

Megszakítások ▸ Külső megszakítások

- /INT0 és /INT1, külső logikai jelek
- Állítható: 0 vagy 1 aktív
- Állítható:
  - állapotvezérelt: a flag maga a bemenő jel
  - élvezérelt: le/felfutó él a flag-et 1-re állítja

Megszakítások ▸ Állapotvezérelt

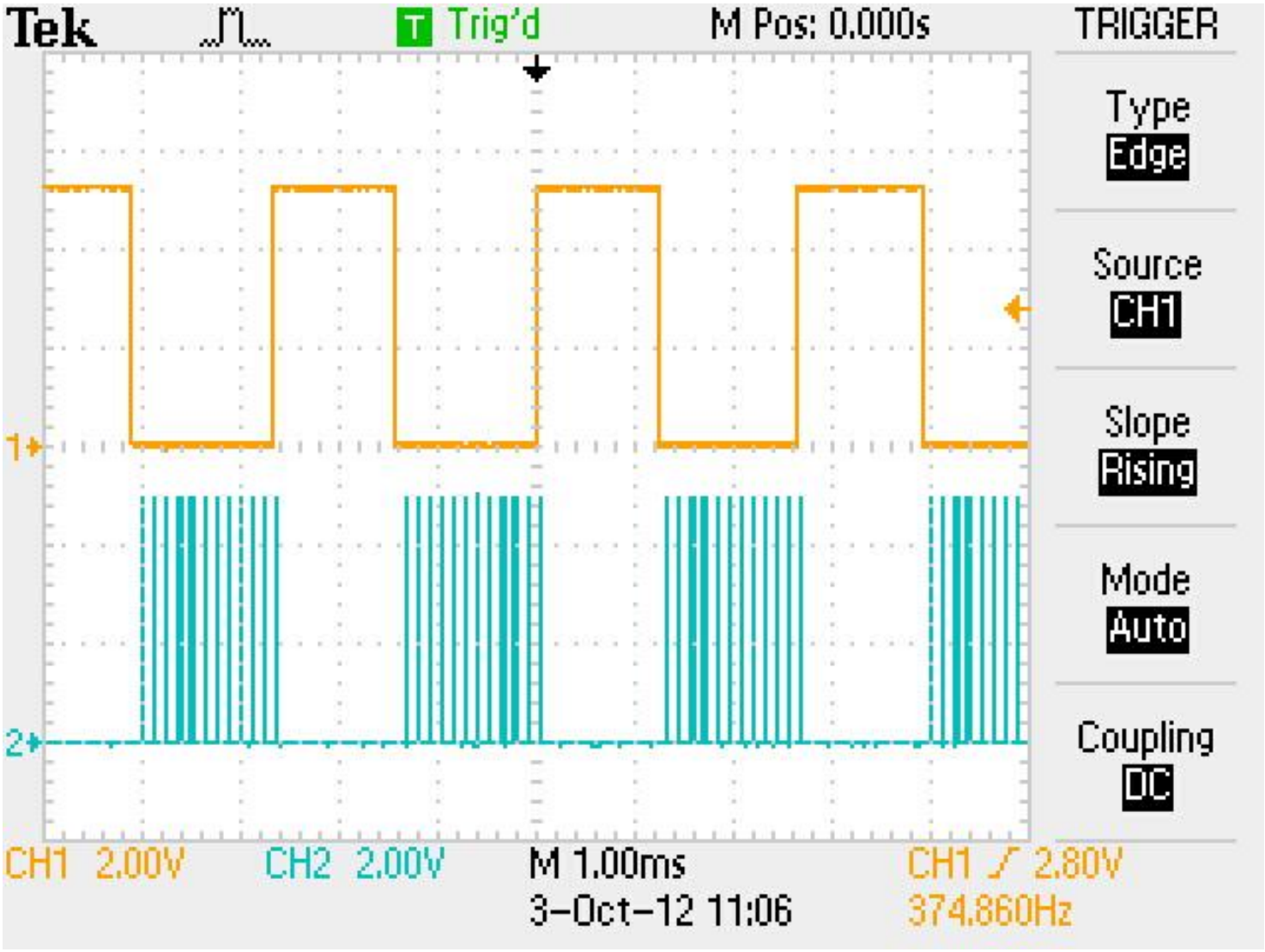
- ha egy állapotra kell reagálni
- folyamatosan megszakítás generálódik, ha aktív
- **a rutin feladata megszüntetnie a kiváltó okot**
- bizonyos külső perifériák ilyen viselkedésűek
- ha túl rövid ideig aktív, elmaradhat a megszakítás



Példa

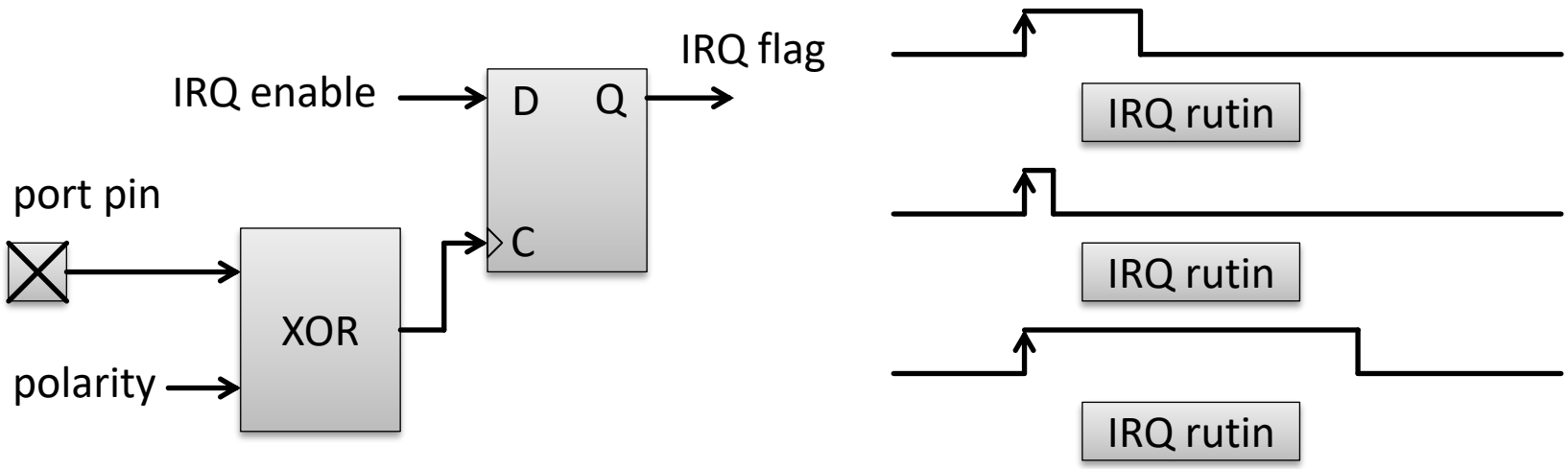
- A megszakítási alprogram egy kimenő jelet rövid ideig logikai magas értékre állít
- Ez oszcilloszkópon jól megfigyelhető (kék színű jel)
- A külső megszakítási jel egy négyszögjel (sárga színű)

Megszakítások ▸ Állapotvezérelt, aktív alacsony

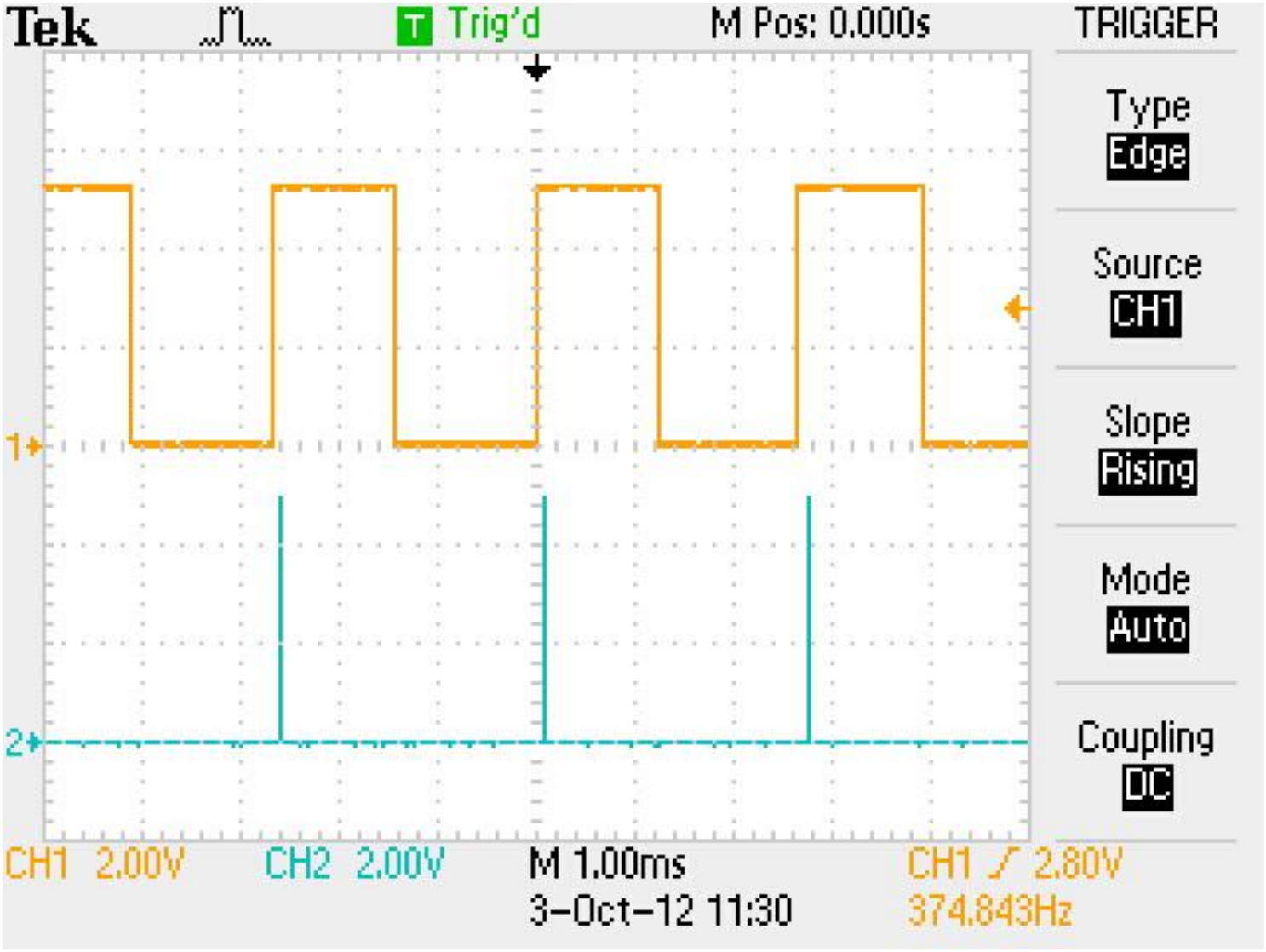


Megszakítások ▸ Élvezérelt

- ha egyes események bekövetkeztére kell reagálni
- csak az aktív váltás generál megszakítást
- egyébként nem számít a jelszint
- akkor is megszakítás, ha igen rövid az impulzus (2 óraciklus a garantált detektálási minimum)



Megszakítások ▸ Állapotvezérelt, felfutó él



Megszakítások ▸ Prioritás

- **Ha fut egy megszakítás és érkezik egy másik?**
  - ha a másik prioritása nagyobb, akkor újabb megszakítás történik, de további már nem
  - egyébként előbb befejeződik az előző, aztán indul a másik
  - ha a megszakítási alprogram futása közben befutott ugyanolyan kérelmek elvesznek – a flag bit nem tud több kérelmet tárolni.

Megszakítások ▸ Interakció a főprogrammal

- A rutin elvileg beállíthat perifériákat, portbiteket, amit a főprogram is
- Figyelni kell a következetes használatra!
- **Kerüljük a főprogram és megszakítási rutin által végzett feladatok keverését!**
- Amit tipikusan el kell menteni/vissza kell állítani:
  - **PSW**
  - **ACC**
  - illetve minden használt regisztert, amit a főprogram is használ
- Hova mentsük el? Hogyan állítsuk vissza?

Megszakítások ▸ Állapotmentés/visszaállítás ▸ A verem használata

- A mentés helye a verem
- Példa – esemény: soros porton adat érkezett
  - **RI**: flag, ami 1-re vált
  - **RI**-t a hardver nem törli, ha a megszakítási rutinra kerül a vezérlés
  - **SBUF**: a beérkezett adat SFR-je

- Példa – esemény: soros porton adat érkezett:

```
push ACC      ; ACC (SFR of A) to stack
push PSW      ; flags to stack
clr  RI       ; clear flag
mov  A,SBUF   ; move data into A
anl  A, #1    ; A and PSW change here
mov  P0,A     ; move data to port PO
pop  PSW      ; restore PSW
              ; reverse order!
pop  ACC      ; restore ACC (A)
reti          ; return and reset interrupt
              ; logic to default state
```

Megszakítások ▸ Állapotmentés/visszaállítás ▸ Regiszterbank váltása

- Regiszterbank váltása a **PSW** RS1,RS0 bitjeivel
- A regiszterbankot a **POP PSW** visszaállítja
- Vigyázat: ***a verem RESET alapértékét módosítani kell!***

```
push ACC      ; ACC (SFR of A) to stack
push PSW      ; flags to stack
mov  PSW,#8   ; use registerbank #1, PSW changes
clr  RI       ; clear flag
mov  R0,#1    ; R0 changes
anl  A, R0    ; A and PSW change
mov  P0,A     ; move data to port PO
pop  PSW      ; restore PSW and registerbank
              ; selection, reverse order!
pop  ACC      ; restore ACC (A)
reti          ; return and reset interrupt
              ; logic to default state
```

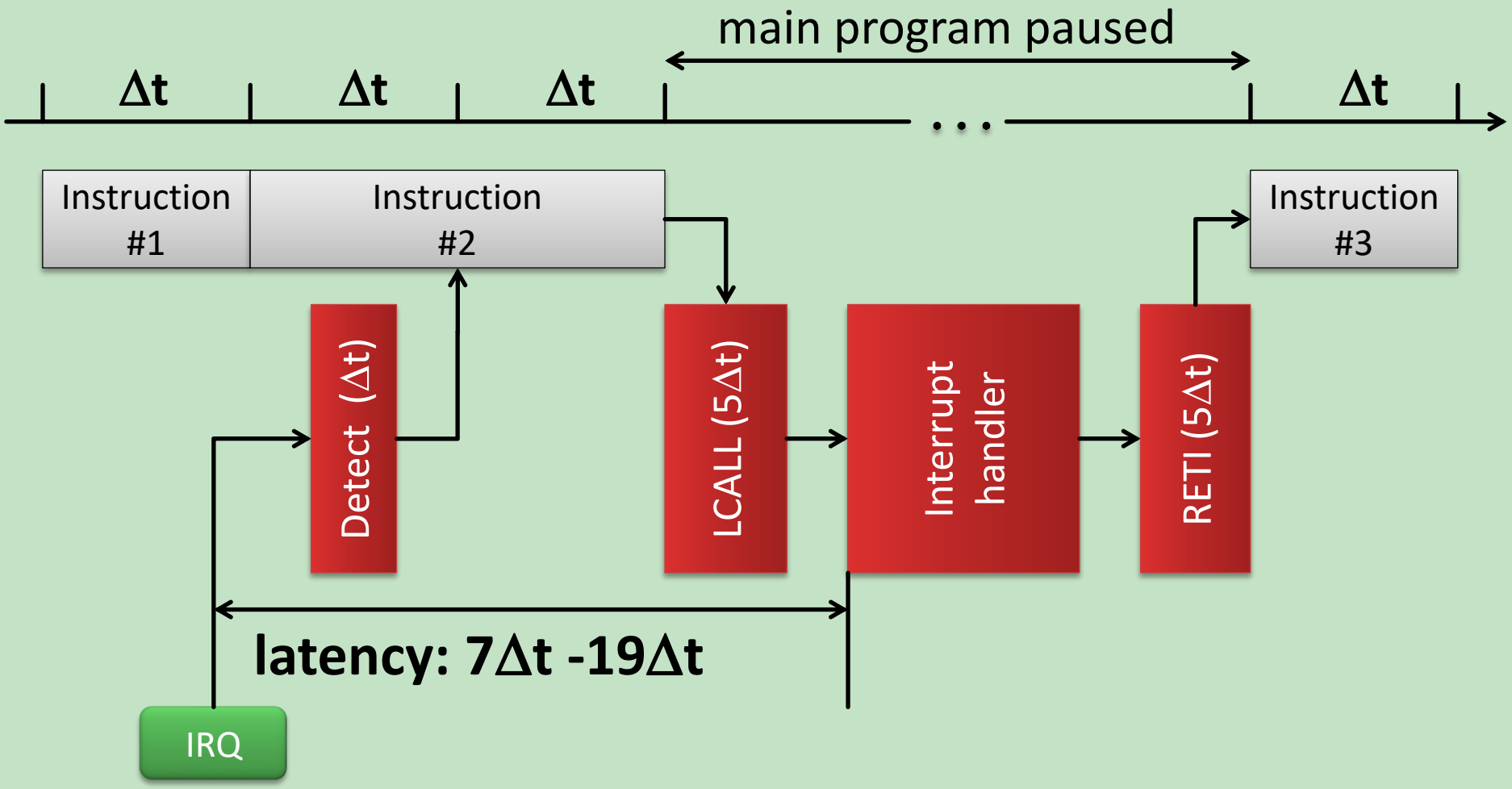
Megszakítások ▸ Késleltetés (latency)

- Mikrovezérlők: valós idejű rendszerekben
- Fontos a precíz időzítés
- **Mennyire pontos a megszakítások indulása?**
  - Az esemény detektálása 1 ciklus
  - Az utasítások hossza 1-8 ciklus lehet
  - A megszakítás meghívása 5 ciklus (**LCALL**)
  - Ha **RETI** közben kérés jön, 1 művelet még végrehajtódik az újabb meghívásig

Megszakítások ▸ Késleltetés

- **A leggyorsabb reagálás: 7 ciklus**
  - 1: detektálás
  - 1: főprogram utasítása lefut
  - 5: a rutin meghívása
- **A leglassabb: 19 ciklus**
  - 1: detektálás
  - 5: **RETI** közben aktíválódik az új kérelem
  - 8: **DIV AB** – a leghosszabb utasítás
  - 5: a rutin meghívása

Megszakítások ▸ Idődiagram



Megszakítások ▸ Késleltetés ▸ Gond ez? - példa

- Feladat: **100kHz-es logikai jel generálása**
  (10us periódusidő)
- A processzor órajele: 25MHz
  (40ns ciklusidő)
- Egy időzítő megszakítást generál 5us időnként
- A megszakítás késleltetési bizonytalansága
  19-7=12 ciklus, azaz 12x40ns=480ns
- Így a jelváltási idők bizonytalansága akár
  480ns/5us = 480/5000 = 0,096 = **9,6%!**

Megszakítások ▸ Késleltetés

- A késleltetés 7-19 ciklus között lehet
- Leggyakrabban 8-9 ciklus
- Elég véletlenszerű is lehet a reagálási idő
- A szoftveres időzítés így nem tökéletesen precíz
- A legtöbbször ez nem gond, de mérlegelni kell
- **Ha nagyon precíz időzítésű jelek szükségesek**:
  - például adott frekvenciájú jelek, PWM jelek
  - hardveres megoldás szükséges, nem szoftveres!
  - Ehhez periféria: PCA (programmable counter array)
- Több megszakítás esetén további késleltetés lehet

Megszakítások ▸ Alkalmazási megfontolások

- Csak rövid idejű feladatokra
- Felszabadítja a főprogramot az eseménykezeléstől
- Egyszerűbbé, megbízhatóbbá teszi a programot
- Gondosan kell tervezni:
  - ***engedélyezés előtt az eseményforrás konfigurálása***
  - ***a főprogrammal ne ütközzön (mentés, visszaállítás)***
  - ***nem atomi műveletek a főprogramban (pl. 16-bit)***
  - ***a verem igénybevételének megfontolása***
  - ***túl gyakori kérések/a rutin futási ideje fontos***
  - ***elveszhetnek kérések***
  - ***flag-ek kezelése, törlése, több flag – egy megszakítás***
  - ***több megszakítási forrás kezelése, extra késleltetés!***
  - ***a prioritás figyelembe vétele***

Megszakítások ▸ vektorok táblázata

- Cím: 0x0003+8*megszakítási sorszám
- A RESET csak 3 byte → LJMP a főprogramhoz
- A többi 8 byte, így a rutinhoz is LJMP kell

Megszakítások ▸ vektorok táblázata

Source	Address	Number	Flag	Cleared by hardware
Reset	0x0000	-	-	-
/INT0 external	0x0003	0	IE0	yes
Timer 0 overflow	0x000B	1	TF0	yes
/INT1 external	0x0013	2	IE1	yes
Timer 1 overflow	0x001B	3	TF1	yes
UART0	0x0023	4	RI0, TI0	no
Timer 2 overflow	0x002B	5	TF2H, TF2L	no
SPI0	0x0033	6	SPIF, WCOL,MODF, RXOVRN	no
SMB0	0x003B	7	SI	no

Megszakítások ▸ vektorok táblázata

Source	Address	Number	Flag	Cleared by hardware
smaRTClock	0x0043	8	ALRM, OSCFAIL	no
ADC0 Window Comparator	0x004B	9	AD0WINT	no
ADC0 End of Conversion	0x0053	10	AD0INT	no
Programmable Counter Array	0x005B	11	CF, CCFn	no
Comparator 0	0x0063	12	CP0FIF, CP0RIF	no
Comparator 1	0x006B	13	CP1FIF, CP1RIF	no
Timer 3 overflow	0x0073	14	TF3H, TF3L	no
Voltage regulator dropout	0x007B	15	-	no
PORT match	0x0083	16	-	no

Assembler és C programozás

Aktuális ma az assembler?

- a processzor, hardver alapos ismerete
- erőforrások pontos ismerete
- C kód ellenőrzése
  - a fordító optimalizálása gondot okozhat a perifériáknál
  - jobban kell figyelni, mint általános C programozásnál
  - a C fordító által generált assembler megértése
  - profiling, optimalizálás
  - hatékonyabb hibakeresés a hardverismeret alapján
- assembler library rutinok megértése, felhasználása

Assembler kód beszúrása C kódba

- teljes programozói szabadság, hatékonyság
- a hardverhez legközelebbi programozás
- C fordító limitációinak kikerülése
- optimalizálás, gyorsítás
  - regisztervváltozók
  - megszakítás-kezelő rutinok
  - ciklusok
  - C-ből hiányzó matematikai műveletek (példa: 24-bites aritmetika)
  - inline-szerű C függvények írása (prolog, epilog optimalizálás)

Assembler programozás
Keil 51

Assembler ▸ Direktivák

- Nem a nyelv elemei
- A fordítónak adnak információkat
- Neveket adhatunk konstansoknak, memóriarészeknek
- Helyet foglalhatunk adatoknak
- Kezdőértéket adhatunk adatoknak
- A programrészek helyét megadhatjuk

Assembler ▸ Szegmensek

- Adatok és kód elhelyezésének definiálása
- Abszolút – a programozó rögzíti a helyet
- Áthelyezhető – a compiler/linker rögzíti
- Öt típus
  - **CODE** – programmemória
  - **DATA** – direkt címzésű adatmemória (RAM: 0-127, SFR 128-255)
  - **IDATA** – indirekt címezhető adatmemória (0-255)
  - **XDATA** – külső RAM (XRAM, lehet on-chip is)
  - **BDATA** vagy **BIT** – bitmemória (20h-2Fh)

Assembler ▸ Abszolút című szegmensek

```
DSEG at 030h
…
…
CSEG at 0100h
…
…
XSEG at 0000h
…
…
```

Assembler ▸ ORG – kezdőcím definiálása

```
DSEG
ORG 030h
X:   DS   1 ; 030h
Y:   DS   1 ; 031h
Z:   DS   8 ; 032h-039h

CSEG at 0000h ; reset address
  LJMP  Main  ; jump to code
ORG 0003h     ; INT0 address
  RETI

Main:
  MOV  a,#10
  …
  …
END
```

Assembler ▸ Változók, konstansok

- Változók foglalása: **DS** (Define storage)
  - nincs értékadás, csak helyfoglalás
  - <változó neve>: **DS** <foglalandó byte-ok száma>

```
x: DS 2
y: DS 1
```

- Konstansok definiálása (értékadás):
  - **DB** (Define byte), **DW**, **DD** (Define word, double word)
  - csak **CSEG** lehet a helye, mivel konstans!

Assembler ▸ Konstansok definiálása

```
CSEG at 0000h

   mov  dptr,#LOOKUP_TABLE ; address of table
   mov  a,#2               ; index
   movc a,@(a+dptr)        ; move to A
   jmp  $                  ; infinite loop!

MESSAGE:   ; no more program code
           ; can't put at the beginning!
   DB 'Number of samples', 0

LOOKUP_TABLE:              ; square of numbers
   DB 0,1,4,9,16
   DB 25,36,49,64

END
```

Assembler ▸ Nevek definiálása

- Hasonló a C nyelv #define direktívájához

```
$include (C8051F410.INC)

LED   EQU  P0.5
COUNT EQU  5

CSEG at 0000h
   anl  PCA0MD,#0BFh  ; watchdog off
   mov  PCA0MD,#000h
   mov  XBR1,#040h    ; crossbar on

   mov  R7,#COUNT
L1:
   cpl  LED    ; complemet LED
   djnz R7,L1  ; repeat COUNT times
   jmp  $      ; infinite loop!
END
```

Assembler ▸ Feltételes végrehajtás

```
    jz    L1
    mov   a,#10
L1:
    add   a,#2
    …
```

Mennyi **a** értéke?

```
    jz    L1
    mov   a,#10
    jmp   L2
L1:
    add   a,#2
L2:
    …
```

Mennyi **a** értéke?

Assembler ▸ Feltételes végrehajtás

```
L1:
    jnb  P1.3,L1
    …
```

Mi történik?

```
    cjne a,#3,L1
    …      ; a=3
    jmp L3
L1:
    jc   L2
    …      ; a>3
    jmp L3
L2:
    …      ; a<3
L3:
    …
```

Assembler ▸ Alprogramok használata

- Alprogramok meghívása:
  - **CALL** <alprogram neve>
  - Az alprogram neve egy memóriacímet jelent
- Bemeneti változók? Visszatérési érték?
  - a programozóra van bízva, ő döntheti el
  - lehet **DPL, DPH, B, ACC**
    - az SDCC compiler is ezt követi
    - baj lehet, ha a szubrutin hív egy másikat is
  - lehet a verem is (tipikus C megoldás)

Példa: LED villogtatása, a Wait bemenő paramétere az akkumulátor

```
$include (C8051F410.INC)

LED EQU  P0.2

CSEG at 0000h
    jmp Main
Main:
    anl   PCA0MD, #0BFh  ; watchdog off
    mov   PCA0MD, #000h
    mov   XBR1,   #040h  ; crossbar on
Main1:
    cpl   LED            ; complement LED
    mov   a,#5           ; wait a*10 ms
    call Wait
    jmp   Main1          ; repeat forever
```

Példa: a Wait alprogram

```
Wait:                   ; SYSCLK=191406Hz
    push acc            ; 2 cycles
Wait1:
    push acc            ; 2 cycles
    mov  a,#127         ; 2 cycles
    djnz acc,$          ; 127*5 cycles
    djnz acc,$          ; 256*5 cycles
    pop  acc            ; 2 cycles
    djnz acc,Wait1      ; 5 cycles
    pop  acc            ; 2 cycles
    ret                 ; 6 cycles
END
```

Assembler ▸ 16-bites összeadás

```
; z=x+y
; 16-bit addition
; little endian data

$include (C8051F410.inc)

DSEG at 020h
   x: DS 2
   y: DS 2
   z: DS 2

CSEG at 0000h
   jmp main
```

```
org 0100h

Main:
   mov  a,x
   add  a,y
   mov  z,a
   mov  a,x+1
   addc a,y+1
   mov  z+1,a
   jmp  $
end
```

Assembler ▸ Tömb elemeinek feltöltése

```
; fill an array with
; ascending numbers

$include
  (C8051F410.inc)

DSEG at 020h
   x: DS 10

CSEG at 0000h
   jmp main
```

```
org 0100h

Main:
   mov  R7,#10
   mov  R0,#x
   clr  a
Loop:
   mov  @R0,a
   inc  a
   inc  R0
   djnz R7,Loop
end
```

Assembler ▸ Tömb elemeinek összegzése

```
DSEG at 020h
  x:   DS 10
  sum: DS 2

CSEG at 0000h
   jmp main

org 0100h

Main:
   mov  R7,#10
   mov  R0,#x
   clr  a
   mov  sum,a
   mov  sum+1,a
```

```
Loop:
   mov  a,@R0
   add  a,sum
   mov  sum,a
   clr  a
   addc a,sum+1
   mov  sum+1,a
   inc  R0
   djnz R7,Loop
end
```

C programozás, SDCC

Programozás C nyelven, SDCC

- Speciális esetek
  - Bitműveletek
  - SFR elérés
  - memóriatípusok
  - assembler programrészek
- Az erőforrások korlátosak!
  - változók pontosság
  - tömbök mérete
- C és assembly keverése

C ▸ Numerikus adattípusok

type	width bit	default	signed range	unsigned range
__bit	1	unsigned	-	0,1
char	8	signed	-128..127	0..255
short	16	signed	-32768..32767	0..65535
int	16	signed	-32768..32767	0..65535
long	32	signed	-2147483648 +2147483647	0.. 4294967296
float IEEE754	32	signed		1.175494351E-38, 3.402823466E+38
pointer	8-24	generic		

C ▸ Numerikus adattípusok ▸ Float

Előjel **1 bit** b31	**Kitevő** **8 bit** b30..b23	**mantissza** **23 bit** b22..b0

- Előjel: 0:pozitív, 1:negatív
- Kitevő: -127..128
- Mantissza: előjel nélküli fixpontos bináris szám, bináris ponttal kezdve
- Pontosság: 23 bit, azaz ≈7 digit

$$\left(1-2\cdot b_{31}\right)\cdot\left(1+b_{22}2^{-1}+\ldots+b_0 2^{-23}\right)\cdot 2^{b_{30}2^7+\ldots+b_{23}2^0-127}$$

C ▸ Numerikus egészek kezelése – shift

- Példa: ADC adat
  - 12-bites előjel nélküli szám
  - unsigned short x;
  - x = (ADC0H << 8) + ADC0L;
  - x = ADC0H*256+ADC0L;
- Aritmetikai shift: signed esetben
  - csak jobbra toláskor különbözik: a legfelső (előjel-) bitet másolja jobbra
  - Ezért is fontos a megfelelő deklarálás!

C ▸ Numerikus egészek kezelése – shift

- Aritmetikai shift: signed esetben
  - csak jobbra toláskor különbözik: a legfelső (előjel-) bitet másolja jobbra
  - Ezért is fontos a megfelelő deklarálás!
  - Hibás kód:

```
int data; // declared as signed (default type in C)
data=(ADC0H << 12)+(ADC0L << 4); // left justified
…
data = data >> 1;  // intent: division by 2
```

C ▸ Numerikus egészek kezelése

```
unsigned short x;
x = -100; // 65536-100, i.e. 65436

char temperature;
// float arithmetic
temperature = (adcData*(1500.0/4096.0)-900)/2.95;
// long arithmetic
temperature = (adcData*150000/4096-90000)/295;

// wrong! adcData*1500 will oveflow!
temperature = (adcData*1500/4096-900)*100/295;
// fixed version, force to use long arithmetic:
temperature = (adcData*1500L/4096-900)*100/295;
```

C ▸ bitek kezelése

- Bit törlése
  - x = x & ~(1 << 3); // 1111 0111
  - x &= ~(1 << 3);
- Bit beállítása
  - x = x | (1 << 3); // 0000 1000
  - x |= (1 << 3);

C ▸ Tárolási osztály ▸ data,xdata

BELSŐ RAM (0-127, DIREKT)

```
__data unsigned char x;

   x=3;

assembler:
   mov  _x,#3
```

KÜLSŐ RAM (16-BITES CÍMZÉS)

```
__xdata unsigned char x;

   x=3;

assembler:
   mov  dptr,#_x
   mov  a,#3
   movx @dptr,a
```

C ▸ Tárolási osztály ▸ idata,pdata

BELSŐ RAM (0..255, INDIREKT)

```
__idata unsigned char x;

    x=3;

assembler:
    mov   r0,#_x
    mov   @r0,#3
```

KÜLSŐ RAM (8-BITES CÍMZÉS)

```
__pdata unsigned char x;

    x=3;

assembler:
    mov   r0,#_x
    mov   a,#3
    movx  @r0,a
```

C ▸ Tárolási osztály ▸ code, bit

CODE

```
__code unsigned char x=3;

   y=x;

assembler (read only):
   mov  dptr,#_x
   clr  a
   movc a,@a+dptr
   mov  _y,a
```

BIT

```
__bit b;

   b=1;

assembler:
   setb _b
```

C ▸ Tárolási osztály ▸ sfr

SFR

```
__sfr __at 0x80 P0;

    P0=3;

assembler:
    mov  0x80,#3
```

SFR16

```
__sfr16 __at 0x8C8A TMR0;

    TMR0=100;

assembler:
    mov  0x8A,#100
    mov  0x8C,#0
vagy:
    mov  0x8C,#0
    mov  0x8A,#100
```

A sorrend számíthat!
(pl. PCA0L olvasás,
PCA0CPLn írás)
Ekkor inkább két 8-bites!

C ▸ Tárolási osztály ▸ sbit

```
__sbit __at 0xD7 CARRY;

    CARRY=1;

assembler:
    setb 0xD7
```

C ▸ Abszolút címzés ▸ XRAM

- A programozó megszabhatja, hová, melyik memóriatartományba kerüljenek az adatok
- Mire jó ez?
  - Egyszerűbb címzési aritmetika
  - Gyorsabb elérés
  - Pl: gerjesztéshez és méréshez használt tömb címe így különbözhet egyetlen bitben

C ▸ Abszolút címzés ▸ XRAM

```
__xdata __at (0x4000) unsigned char x[16];

    x[2]=7;

assembler:
    mov  a,#7
    mov  dptr,#0x4002
    movx @dptr,a
```

C ▸ Abszolút címzés ▸ code

```
__code __at (0x7f00) char Msg[] = "Message";

   putchar(Msg[2]);

assembler:
   mov   dptr, #0x7F02
   clr   a
   movc  a, @a+dptr
   mov   dpl,a             ; paraméterátadás
   lcall _putchar          ; _ a név elé
```

C ▸ Abszolút címzés ▸ bit

```
__bit __at (0x80) GPIO_0;

   GPIO_0=1;

assembler:
   setb 0x080
```

C ▸ Memóriamodellek

- **A változók alapértelmezett helyét szabja meg**
- Célszerű választás: small
  - leggyorsabb kód
  - a programozó dönt
- A tárolási osztály választható
- Nem szabad keverni:
  - más modellű object fájlok
  - más modellű library fájlok
- Verem:
  - idata
  - opcionálisan: pdata

Modell	Variables
small	data
medium	pdata
large	xdata
huge	xdata

C és assembler kapcsolat

C & ASM ▸ Inline assembler

- C kódba assemblert illeszthetünk:

```
__asm
    clr   a             /* C stílusú komment */
    mov   R0,#0         // P0, C++ stílusú komment
    mov   R1,#0x80      // C stílusú hexadecimális
    mov   a,R2
    jz    L1            // címke használata
    mov   R0,#0
L1:
    mov   R1,#1
__endasm;
```

C & ASM ▸ Változók és elérésük

- A nevek elé a C fordító _-t generál:

```
unsigned char c;
c=5;

__asm
    mov _c,#5
__endasm;
```

- lokális változók lehetnek regiszterekben:

```
void f(void)
{
    char c;           /* regiszterbe kerül */
    static char d;    /* memóriába kerül */
```

C & ASM ▸ Paraméterátadás

- **A függvények paraméterei:**
  - 1 byte: DPL
  - 2 byte: DPL, DPH
  - 3 byte: DPL, DPH, B
  - 4 byte: DPL, DPH, B, ACC
  - második paraméter: adatmemória
- **Visszatérési értékek:**
  - 1 byte: DPL
  - 2 byte: DPL, DPH
  - 3 byte: DPL, DPH, B
  - 4 byte: DPL, DPH, B, ACC
- **__reentrant függvények: verem**

C & ASM ▸ Példa

```
char add(char a, char b)
{
   return a+b;
}

_add:
   mov  r2,dpl
   mov  a,_add_PARM_2 ; direkt címzés
   add  a,r2
   mov  dpl,a
   ret
```

C & ASM ▸ Példa: lokális változó, paraméter

```
char MaskP0(char a)
{
   char c;

   c=P0;
   c=c & a;
   return c;

}

_MaskP0:
   mov  r2,dpl   ; ez a függvény bemenő paramétere
   mov  r3, P0   ; a c változó az r3 regiszter
   mov  a,r2     ; a művelet előkészítése
   anl  ar3,a    ; r3 direkt címzése (erre van anl)
   mov  dpl,r3   ; a függvény visszatérési értéke
   ret
```

C & ASM ▸ __reentrant függvények

- A függvény egyszerre több példányban futhat (például megszakításban és a főprogramban)
- Nem használhat olyan változókat, amik statikusok
  - pl. R0-R7
- A verem a helye a paramétereknek, lokális változóknak
- Ritkán használjuk, de fontos:
  - Számos művelet ebben a környezetben függvényhívás!
  - Például: valós aritmetika, long aritmetika
  - Kerüljük ezek használatát megszakításban!

C & ASM ▸ Megszakítások kezelése

```
volatile unsigned char counter;

/* Timer 2: interrupt number: 5 */
void IntHandler(void) __interrupt 5
{
  TMR2CN&=~0x80; // clear flag, bit 7 of TMR2CN
  counter++;
}

_IntHandler:
   anl  _TMR2CN,#0x7F ; clear flag
   inc  _counter
   reti
```

C & ASM ▸ Megszakítások ▸ using – R0..R7

```
volatile unsigned char counter;

/* Timer 2: interrupt number: 5 */
void IntHandler(void) __interrupt 5 __using 1
{
  TMR2CN&=~0x80;      /* clear flag*/
  counter++;
}

_IntHandler:
   push psw
   mov  psw,#0x08   ; R0..R7 at 0x08-0x0F
   anl  _TMR2CN,#0x7F
   inc  _counter
   pop  psw
   reti
```

C & ASM ▸ Megszakítások ▸ __critical

```
/**************************
EA (globális megszakítás engedélyezés)
kikapcsolása az elején, visszaállítás a végén
**************************/

void f(void) __critical
{
    P0=0;
    P1=0xFF;
}

__critical {
    P0=0;
    P1=0xFF;
}
```

C & ASM ▸ Megszakítások ▸ Mire figyeljünk?

- *engedélyezés előtt az eseményforrás konfigurálása*
- *a főprogrammal ne ütközzön (mentés, visszaállítás)*
  - *C: az Rn regisztereket nem menti*
- *a verem igénybevételének megfontolása*
- *túl gyakori kérések/a rutin futási ideje fontos*
- *elveszhetnek kérések*
- *flag-ek kezelése, törlése*
- *több flag – egy megszakítás*
- *több megszakítási forrás kezelése, extra késleltetés!*
- *a prioritás figyelembe vétele*

C & ASM ▸ Megszakítások ▸ Mire figyeljünk?

- **Volatile** változók
  - értékük bármikor változhat a megszakítás rutinban
  - az optimalizálásból kimarad a változó
- **Nem atomi műveletek**
  - atomi művelet: egyetlen utasítás
  - 16 bites változó módosítása nem atomi
- **__critical** használata
  - *Megvédi a programrészt a megszakítások által okozott késleltetéstől, adatmódosítástól, műveletektől*
  - *a megszakítás késhet emiatt*

```
unsigned char d;
volatile unsigned char vd;

   d=5;
       mov   _d,#0x05
   P0=d+3;
       mov   _P0,#0x08
   vd=5;
       mov   _vd,#0x05
   P0=vd+3;
       mov   a,#0x03
       add   a,_vd
       mov   _P0,a
```

- **volatile int x;**
- Megszakításban x változhat
- Főprogram:

C	Assembler

`if (x>10) …`	`clr c`
	`mov a,#0x0A`
	`subb a,_x`
	`clr a`
	`subb a,(_x + 1)`
	`jnc …`

- **volatile int x;**
- Kimaradhat egy megszakításból x frissítése

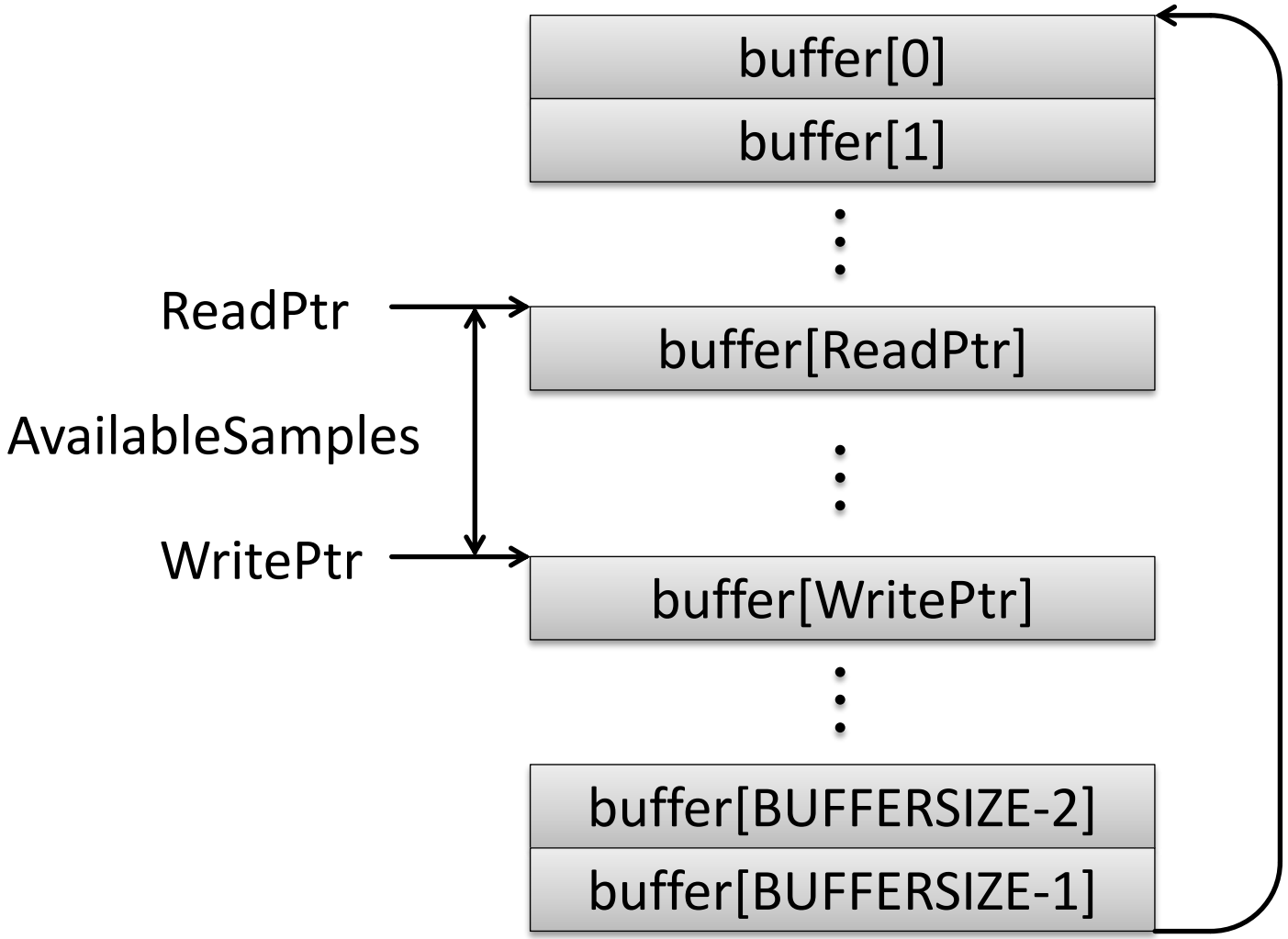
Main program	Interrupt handler

`protect_x=1;`
`if (x>10) …`
`protect_x=0;`
	`if (!protect_x) {`
	`  x = (ADC0H << 8) | ADC0L;`
	`}`

Példa megszakításkezelésre, adatcserére

- Az A/D konverter mintavételez adott rátával
- Az adatok egy tömbbe kerülnek folyamatosan
- A tömb végére érve az írás/olvasás a tömb elejéről kezdődik (ring buffer)
- Az A/D megszakítás írja a tömböt, a főprogram kiolvassa az adatokat és továbbítja
- Csak annyi olvasható, amennyi éppen van
- Miért szükséges?
  - Több adat is beérkezhet kiolvasás nélkül, a főprogramnak nem kell egyenként gyorsan reagálnia

C & ASM ▸ Megszakítások ▸ Ring buffer példa



C & ASM ▸ Megszakítások ▸ Ring buffer példa

```
volatile int buffer[BUFFERSIZE];
volatile unsigned char WritePtr;
volatile unsigned char AvailableSamples;
unsigned char ReadPtr;

void IRQ(void) __interrupt ADC_VECTOR
{
   AD0INT=0;
   buffer[WritePtr] = (ADC0H << 8) | ADC0L;
   WritePtr = (WritePtr + 1) % BUFFERSIZE;
   AvailableSamples++;  // error, if > BUFFERSIZE
}
…
if (AvailableSamples) {
   Send(buffer[ReadPtr]);
   ReadPtr = (ReadPtr + 1) % BUFFERSIZE;
   AvailableSamples--;
}
```

*Feladat: módosítsuk adatcsomagok esetére (**PacketSize** adja meg)*

C & ASM ▸ Megszakítások ▸ Alternatív megoldás

```
volatile bit UpdateAvailableSamples;
volatile unsigned int AvailableSamples;   // 16-bit!

void IRQ(void) __interrupt ADC_VECTOR
{
…
    if (UpdateAvailableSamples)
    {
      AvailableSamples = (WritePtr-ReadPtr ) % BUFFERSIZE;
      UpdateAvailableSamples=0;
    }
}
…
UpdateAvailableSamples=1;
while (UpdateAvailableSamples);

for (i=0; i<AvailableSamples; i++)
{
   Send(buffer[ReadPtr]);
   ReadPtr = (ReadPtr + 1) % BUFFERSIZE;
}
```

C & ASM ▸ Start-up kód

- **A main() előtt a fordító generál egy kódot**
- **Változók kezdőértékének beállítása**
- Tömböket is inicializálhat (0 értékkel)
  - **A watchdog miatt ez gond lehet! Hosszú ideig tarthat**
- Saját kézbe vétel: átírjuk a kódot

```
unsigned char _sdcc_external_startup ()
{
  PCA0MD &= ~0x40; // watchdog off
  PCA0MD = 0x00;
  return 0;        // 0: initialise variables
}
```