Salvatore Moscuzza (Shockwaver)

SEMPLICE MAPPATURA DELLA RAM NEI PIC

20 July 2013

Premessa introduttiva

A volte nell'utilizzo dei Microcontrollori PIC della Microchip può capitarci di voler/dover modificare la struttura di allocazione delle variabili nella memoria RAM, vuoi per esigenze di ottimizzazione vuoi per esigenze di progetto. Lo scopo di qusto articolo è di fornire una guida basilare alla comprensione dei meccanismi necessari per una semplice mappatura della RAM a disposizione. Effettueremo tali operazioni sia per il compilatore C18 con la modifica di un linker script, sia per la nuova generazione X degli strumenti di sviluppo Microchip la IDE MPLAB X con relativo compilatore XC8 intervenendo direttamente nella GUI di suddetta IDE. In questo articolo verrà preso in considerazione il PIC18F47J53 in quanto Core del PIERIN PIC18, scheda di progettazione filantropicamente messa a disposizione da TardoFreak.

Se nel browser le immagini risultassero tagliate, aperte separatamente dalla pagina dell'articolo le potrete visualizzare nella loro interezza.

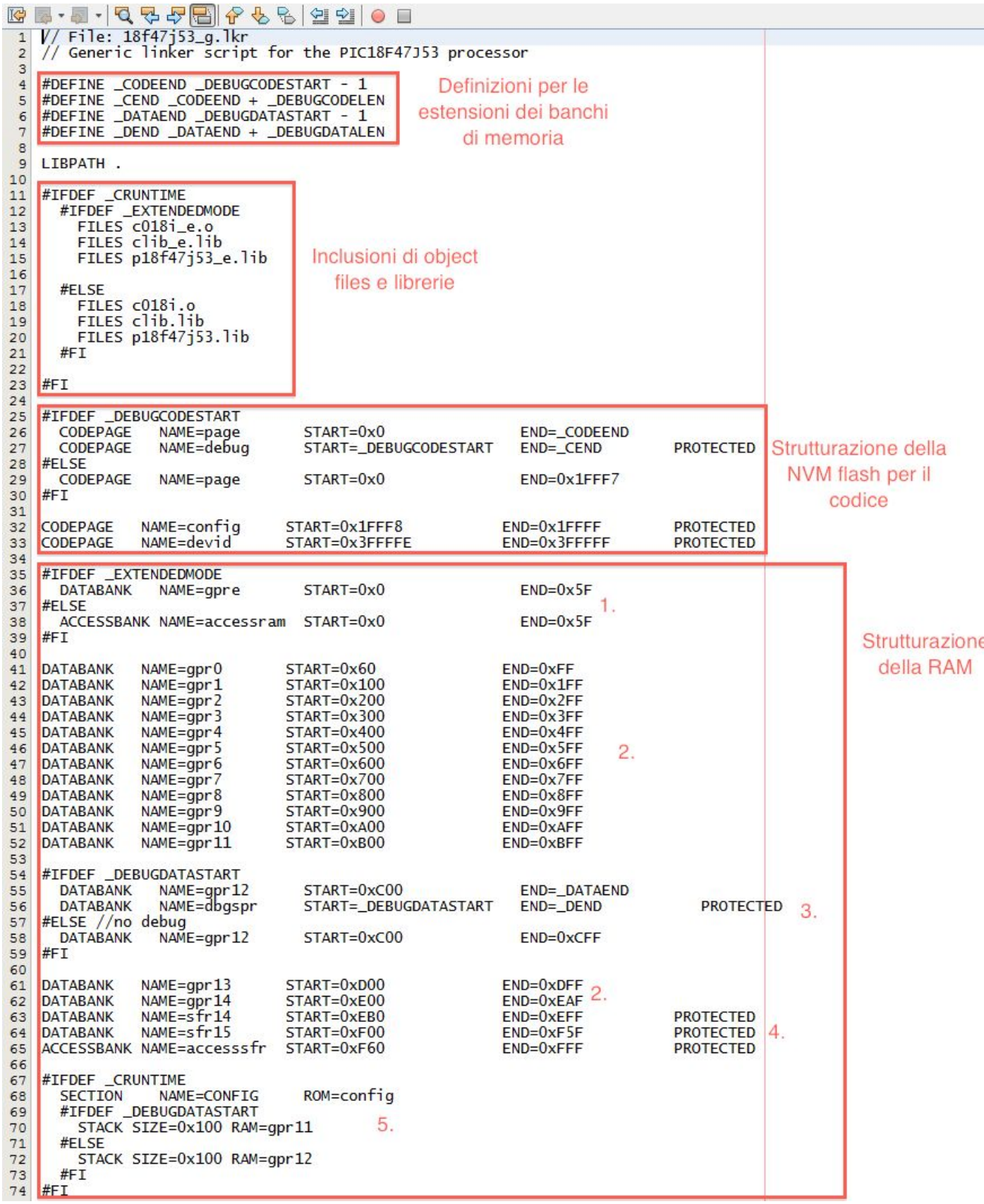
C18

Analizziamo la "vecchia guardia", perché si sa.. gallina vecchia fa buon brodo!

Il nostro Linker Script

Un Linker Script nell'ambito dei microcontrollori, come da nome, è un file di script che il Linker utilizza per diversi scopi, tra cui quello in analisi in questo articolo, ovvero la mappatura della memoria RAM del microcontrollore. Scelta la MCU, in una delle sottocartelle nella directory di installazione del compilatore possiamo trovare un linker script di esempio perfettamente utilizzabile per il nostro micorcontrollore.

Partendo dalla directory di installazione del compilatore mplabc18 e spostandoci in ./dat/lkr/ troviamo, tra gli altri, il nostro linker script da analizzare/modificare: 18f47j53_g.lkr. Aprendolo nella IDE vedremo questo:



```
1  // File: 18f47j53_g.lkr
2  // Generic linker script for the PIC18F47J53 processor
3
4  #DEFINE _CODEEND _DEBUGCODESTART - 1
5  #DEFINE _CEND _CODEEND + _DEBUGCODELEN
6  #DEFINE _DATAEND _DEBUGDATASTART - 1
7  #DEFINE _DEND _DATAEND + _DEBUGDATALEN
8
9  LIBPATH .
10
11 #IFDEF _CRUNTIME
12   #IFDEF _EXTENDEDMODE
13     FILES c018i_e.o
14     FILES clib_e.lib
15     FILES p18f47j53_e.lib
16
17   #ELSE
18     FILES c018i.o
19     FILES clib.lib
20     FILES p18f47j53.lib
21   #FI
22
23 #FI
24
25 #IFDEF _DEBUGCODESTART
26   CODEPAGE   NAME=page       START=0x0                 END=_CODEEND
27   CODEPAGE   NAME=debug      START=_DEBUGCODESTART     END=_CEND          PROTECTED
28 #ELSE
29   CODEPAGE   NAME=page       START=0x0                 END=0x1FFF7
30 #FI
31
32 CODEPAGE   NAME=config     START=0x1FFF8             END=0x1FFFF        PROTECTED
33 CODEPAGE   NAME=devid      START=0x3FFFFE            END=0x3FFFFF       PROTECTED
34
35 #IFDEF _EXTENDEDMODE
36   DATABANK   NAME=gpre       START=0x0                 END=0x5F
37 #ELSE
38   ACCESSBANK NAME=accessram  START=0x0                 END=0x5F
39 #FI
40
41 DATABANK   NAME=gpr0       START=0x60                END=0xFF
42 DATABANK   NAME=gpr1       START=0x100               END=0x1FF
43 DATABANK   NAME=gpr2       START=0x200               END=0x2FF
44 DATABANK   NAME=gpr3       START=0x300               END=0x3FF
45 DATABANK   NAME=gpr4       START=0x400               END=0x4FF
46 DATABANK   NAME=gpr5       START=0x500               END=0x5FF
47 DATABANK   NAME=gpr6       START=0x600               END=0x6FF
48 DATABANK   NAME=gpr7       START=0x700               END=0x7FF
49 DATABANK   NAME=gpr8       START=0x800               END=0x8FF
50 DATABANK   NAME=gpr9       START=0x900               END=0x9FF
51 DATABANK   NAME=gpr10      START=0xA00               END=0xAFF
52 DATABANK   NAME=gpr11      START=0xB00               END=0xBFF
53
54 #IFDEF _DEBUGDATASTART
55   DATABANK   NAME=gpr12      START=0xC00               END=_DATAEND
56   DATABANK   NAME=dbgspr     START=_DEBUGDATASTART     END=_DEND          PROTECTED
57 #ELSE //no debug
58   DATABANK   NAME=gpr12      START=0xC00               END=0xCFF
59 #FI
60
61 DATABANK   NAME=gpr13      START=0xD00               END=0xDFF
62 DATABANK   NAME=gpr14      START=0xE00               END=0xEAF
63 DATABANK   NAME=sfr14      START=0xEB0               END=0xEFF          PROTECTED
64 DATABANK   NAME=sfr15      START=0xF00               END=0xF5F          PROTECTED
65 ACCESSBANK NAME=accesssfr  START=0xF60               END=0xFFF          PROTECTED
66
67 #IFDEF _CRUNTIME
68   SECTION    NAME=CONFIG     ROM=config
69   #IFDEF _DEBUGDATASTART
70     STACK SIZE=0x100 RAM=gpr11
71   #ELSE
72     STACK SIZE=0x100 RAM=gpr12
73   #FI
74 #FI
```

Definizioni per le estensioni dei banchi di memoria

Inclusioni di object files e librerie

Strutturazione della NVM flash per il codice

1.

2.

3.

2.

4.

5.

Strutturazione della RAM

Linker Script originale.jpg

Prima di commentare la RAM vorrei solo far notare che, nella parte dedicata alla strutturazione della memoria NV per il codice, in genere si trova la definizione dell'area per il bootloader.

La RAM

I banchi di memoria possono essere definiti DATABANK o ACCESSBANK. Mentre i DATABANK sono i banchi di RAM standard, gli ACCESSBANK rappresentano un potenziamento architetturale che permette alle operazioni eseguite all'interno di queste aree di girare più velocemente grazie alla riduzione dei tempi di accesso (non c'è bisogno di preselezionare il banco). L'estensione di questa memoria particolare varia a secondo del microcontrollore ma per i PIC18 è sempre localizzata nei lower bytes del banco 0 per i dati, negli upper bytes del banco 15 per gli SFR. Per chi volesse approfondire rimando alla sezione 6.3 del PIC® 18C MCU Family Reference Manual (DS39500) e più nello specifico alla sezione 6.3 del Datasheet del PIC18F47J53

1. Sorvolando sull'Extended Intruction Set, qui vengono definiti i 96 bytes di RAM di tipo ACCESS nel banco 0. Come si vede, a quest'area viene assegnato un nome, un indirizzo di partenza e uno di fine. Queste caratteristiche si ripetono per tutte le definizione dei banchi.
2. Qui vengono definiti i banchi di memoria nei registri G.eneral P.urpose R.egister da 0 a 14. GPR0 parte adiacentemente alla fine della accessram e ognuno largo 256 bytes (tranne ovviamente proprio gpr0).
3. GPR12 è particolare perché, se ha avuto luogo una compilazione per debug, esso si estende dalla fine di GPR11 fino alla fine della memoria allocata per le variabili, che però non esaurisce la RAM lasciando posto a dbgspr ove risiederanno le informazioni di debug. Se non siamo in debug GPR12 è costituito semplicemente da altri 256 byte di memoria
4. Gli ultimi banchi sono RESERVED (quindi non allocabili a meno di una esplicita indirizzazione che vedremo a breve) per gli s.pecial f.unction r.egister(s) di cui gli ultimi 160 bytes sono di tipo ACCESS.
5. Per il runtime viene dichiarata una SECTION (delle quali tra poco ne vedremo l'utilità) che punta direttamente al CODEPAGE di nome config alla linea 32 di questo linker script e viene inoltre definita l'area destinata allo stack che, in caso di debug si trova nel banco 11, altrimenti al 12: 256 bytes.

Customizzazione

Da quello fino a qui detto appare evidente che, volendo specificare (vedremo come) un banco per l'allocazione di una variabile, questa non potrà superare le dimensioni del banco stesso, ovvero 256 bytes. In altre parole non possono esserci vettori (o strutture di memorie in genere) più grandi di 256 byte. Questo limite è aggirabile e vedremo come.

Modifichiamo questo layout secondo i nostri bisogni.

Ipotizziamo questo scenario:

1. Vogliamo lasciare immutata l'area di tipo ACCESS
2. Vogliamo impedire al compilatore di allocare variabili nel banco 0
3. Vogliamo allocare variabili arbitrarie in gpr1
4. Vogliamo lasciar decidere l'allocazione di alcune variabili al compilatore
5. Vogliamo allocare un vettore INIZIALIZZATO di 2101 bytes a partire dall'indirizzo 0x307 (Tosta? No, vedrete..)

Ecco il linker script, commentiamolo:

```
35 #IFDEF _EXTENDEDMODE
36   DATABANK   NAME=gpre       START=0x0              END=0x5F
37 #ELSE
38   ACCESSBANK NAME=accessram  START=0x0              END=0x5F
39 #FI
40
41 DATABANK   NAME=gpr0       START=0x60             END=0xFF     PROTECTED
42 DATABANK   NAME=gpr1       START=0x100            END=0x1FF    PROTECTED  1.
43 DATABANK   NAME=gpr2       START=0x200            END=0x2FF
44 DATABANK   NAME=gpr3       START=0x300            END=0x306    2.
45 DATABANK   NAME=OurVectReg START=0x307            END=0xB3B    PROTECTED  3.
46 //DATABANK   NAME=gpr4       START=0x400            END=0x4FF
47 //DATABANK   NAME=gpr5       START=0x500            END=0x5FF
48 //DATABANK   NAME=gpr6       START=0x600            END=0x6FF
49 //DATABANK   NAME=gpr7       START=0x700            END=0x7FF    4.
50 //DATABANK   NAME=gpr8       START=0x800            END=0x8FF
51 //DATABANK   NAME=gpr9       START=0x900            END=0x9FF
52 //DATABANK   NAME=gpr10      START=0xA00            END=0xAFF
53 DATABANK   NAME=gpr11      START=0xB3C            END=0xBFF  5.
54
55 #IFDEF _DEBUGDATASTART
56   DATABANK   NAME=gpr12      START=0xC00            END=_DATAEND
57   DATABANK   NAME=dbgspr     START=_DEBUGDATASTART  END=_DEND          PROTECTED
58 #ELSE //no debug
59   DATABANK   NAME=gpr12      START=0xC00            END=0xCFF
60 #FI
61
62 DATABANK   NAME=gpr13      START=0xD00            END=0xDFF
63 DATABANK   NAME=gpr14      START=0xE00            END=0xEAF
64 DATABANK   NAME=sfr14      START=0xEB0            END=0xEFF          PROTECTED
65 DATABANK   NAME=sfr15      START=0xF00            END=0xF5F          PROTECTED
66 ACCESSBANK NAME=accesssfr  START=0xF60            END=0xFFF          PROTECTED
67
68 SECTION     NAME=Bank1          RAM=gpr1
69 SECTION     NAME=OurVectSec     RAM=OurVectReg  6.
70
```

Linker Script modificato.jpg

1. Per gpr0 e 1 l'attributo PROTECTED ci assicura che il compilatore non vi allochi variabili.
2. I banchi gpr2 e 3 sono lasciati liberi per il compilatore. GPR3 inoltre ha uno span di soli 7 bytes perché a 0x307 vogliamo che inizi il nostro mega-vettore.
3. L'area di memoria per il nostro vettore comincia a 0x307 ed è ESATTAMENTE 2101 bytes, per non sprecare spazio.
4. I gpr da 4 a 10 sono stati eliminati perché li abbiamo unificati in OurVectReg

5. E' sempre buona regola inserire gpr di padding nel caso in cui l'unificazione di alcuni banchi non esaurisca i bytes dell'ultimo banco usato, ed è sempre buona regola dare il nome relativo a tale banco.
6. Ed ecco finalmente le SECTION. Sono delle sezioni LOGICHE il cui nome viene usato nel codice per l'indirizzamento in fase di dichiarazione delle variabili.

Verifica

Vediamo adesso uno snippet di codice per verificare i risultati ottenuti. Creato un progetto in MPLAB X, incluso il linker script modificato e DISABILITATO TUTTE LE OTTIMIZZAZIONI DEL COMPILATORE per essere sicuri di avere tutto sotto controllo, scriviamo due righe di codice

```
58  volatile int a; 1.
59
60  #pragma udata Bank1
61  volatile int b;
62  #pragma udata
63
64  #pragma idata OurVectSec
65  volatile unsigned char c[2101] = {0x0, 0x1, 0x2, 0x3,
66                                    0x4, 0x5, 0x6, 0x7,
67                                    0x8, 0x9, 0xA, 0xB,
68                                    0xC, 0xD, 0xE, 0xF};
69  #pragma idata
70
71  void main(void) {
72      unsigned char *ptr = &c[0]; 4.
73
74      a = 1;
75      b = 2;
76      ptr[7] = 0xA; 5.
77
79      while(1);
80  }
81
```

2. 3.

Watches

Name	Type	Address	Value
a	int	0x300	0x0001
b	int	0x100	0x0002
c	unsigned char[2101]	0x307	"\u0x0\u0x1\u0x2\u0x3\u0x4\u0x5\u0x6\u0xa\u0x8\u0x9\u
ptr	unsigned char*	0xC01	0x0307
<Enter new watch>			

Code snippet.jpg

Tutte le variabili sono dichiarate "volatile" per essere sicuri che il compilatore le allochi anche se non vengono utilizzate (in realtà questo dovebbe già essere garantito dalla disattivazione delle ottimizzazioni).

1. Questa è la variabile che diamo in pasto al compilatore affinché la allochi dove più gli garba.

2. Con la direttiva per il precompilatore "#pragma udata" invece specifichiamo che vogliamo che la variabile u.ninitialized(data) venga allocata nella SECTION Bank1 che rimanda al banco gpr1 all'indirizzo 0x100
3. La direttiva "#pragma idata" è identica a "udata" con la sola differenza che è dedicata alle variabili i.nitialized(data). Questo deve essere il nostro vettore di 2101 bytes allocato nella SECTION OurVectSec che corrisponde al banco OurVectReg @ 0x307 e inizializzato con i valori che vedete
4. Per l'accesso a variabili "spalmate" su più banchi bisogna usare dei puntatori di appoggio definiti ad esempio così (o anche solo "unsigned char *ptr = c"). L'indirizzazione canonica (col nome proprio della variabile in oggetto), sotto determinate circostanze può dare luogo a comportamenti indefiniti.
5. L'indicizzazione può comunque avvenire nel modo consueto.

Osserviamo adesso nella finestra delle watches quello che abbiamo ottenuto:

Variabile "a": il compilatore decide di allocarla in 0x300 ovvero in gpr3 che avevamo lasciato libero.

Variabile "b": come richiesto è allocata in 0x100 ovvero in gpr1.

Vettore "c": allocato in 0x307 è lungo 2101.

Puntatore "ptr": il compilatore decide di allocarlo all'indirizzo 0xC01, anche questo libero. Questi punta a 0x307, proprio la posizione del nostro vettore.

Tutte le variabili hanno i valori che gli abbiamo assegnato nel codice

cvd.

XC8

Se è vero che "gallina vecchia fa buon brodo" è altrettanto vero che si dice "largo ai giovani" e fra un attimo capirete perché.

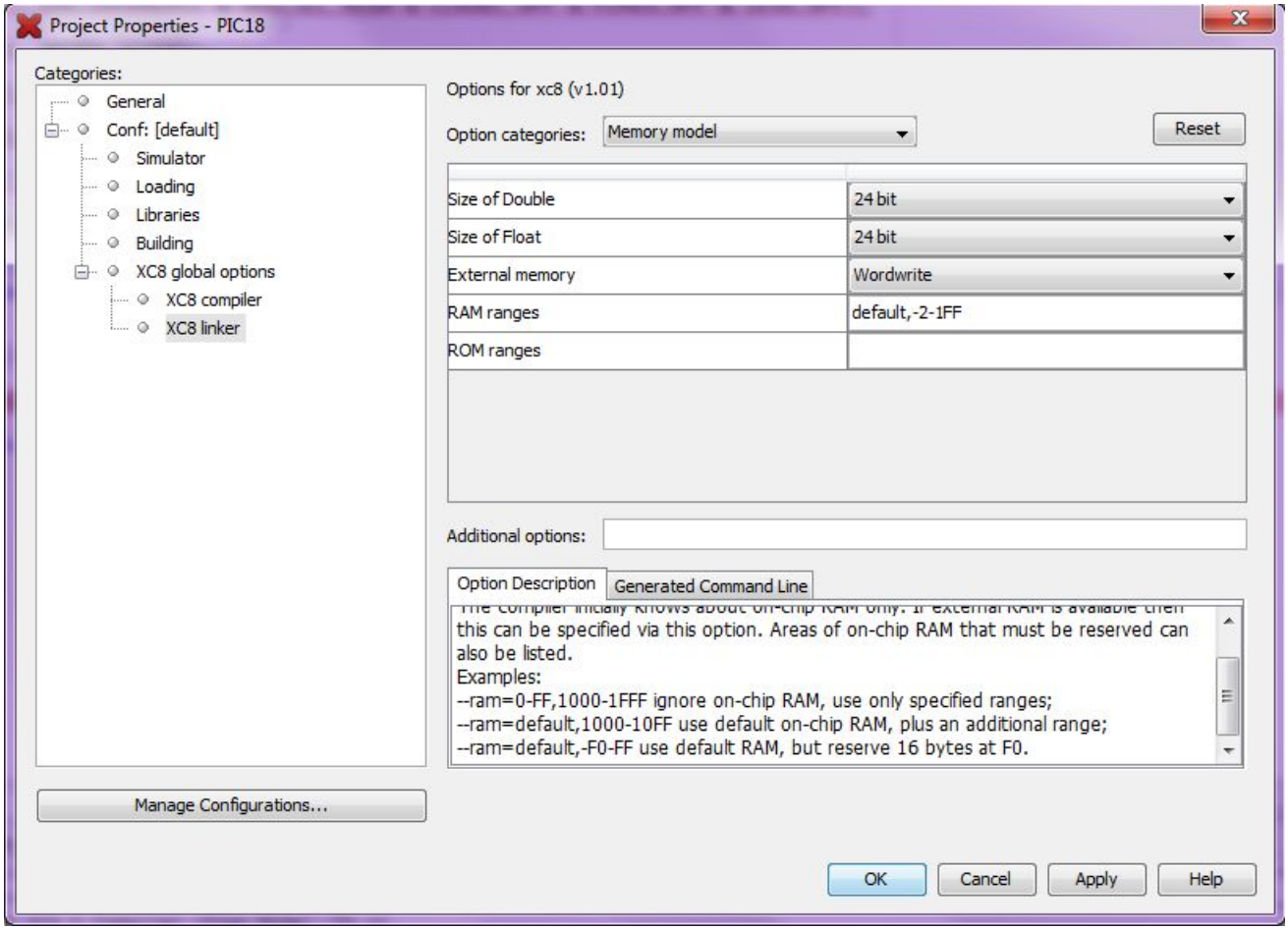
Customizzazione

Ripetiamo le ipotesi... ctrl+c, ctrl+v

1. Vogliamo lasciare immutata l'area di tipo ACCESS (0x0 - 0x5F). Stavolta però la vogliamo interdire al compilatore, lasciandogli solo i primi 2 bytes del banco 0 dei quali, per questo esempio, ha bisogno.
2. Vogliamo impedire al compilatore di allocare variabili nel banco 0 (0x60 - 0xFF)
3. Vogliamo allocare variabili arbitrarie in gpr1 (0x100 - 0x1FF)
4. Vogliamo lasciar decidere l'allocazione di alcune variabili al compilatore

5. Vogliamo allocare un vettore INIZIALIZZATO di 2101 bytes a partire dall'indirizzo 0x307 (Tosta? No, vedrete..) (0x307 - 0xB3B)

Con l'XC8 i linker script diventano obsoleti, per raggiungere il nostro scopo, dopo aver creato il progetto apposito, basta aprirne le proprietà, selezionare sull'albero di sinistra "XC8 Linker" e sulla ComboBox "Option categories" di destra "Memory model". "RAM ranges" è ciò che stiamo cercando.



XC8 Linker.jpg

Come brillantemente descritto nella finestrella in basso, dobbiamo specificare i ranges di memoria che vogliamo lasciare gestire al Linker. Analizziamo la stringa inserita:

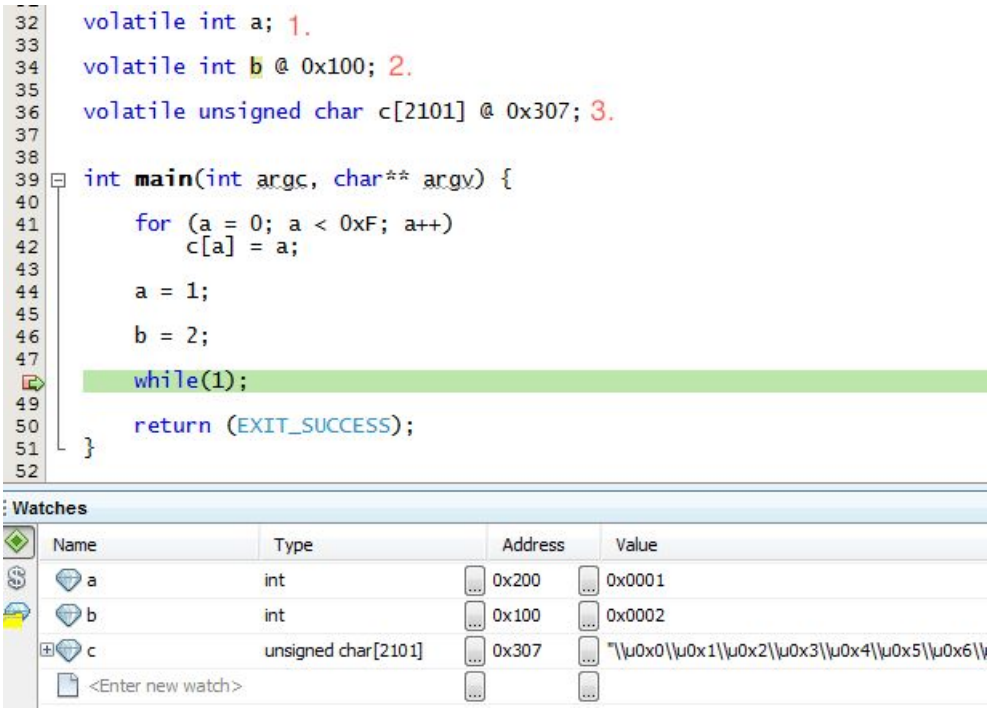
"default,": indica la struttura standard della ram.

"-2-1FF": il primo trattino è un segno "meno" col quale indichiamo che vogliamo sotrarre al controllo del compilatore il range di indirizzi 0x2 - 0x1FF, come da ipotesi 1. + 2. + 3.

Per il mega vettore non vi è bisogno di riservarne lo spazio in memoria, ci penserà la dichiarazione stessa.

Verifica

Queste le righe di codice con watches allegate.



Code snippet XC8.jpg

1. Solita variabile da lasciar gestire al compilatore.
2. Le direttive "#pragma udata" e "idata" scompaiono, per l'indirizzazione adesso basta un semplice "@ [address]". Terrific. (Attenzione!) Il linker non fa il check degli overlap. Questo può essere sia un bene che un male: un male perché diventa un sistema prono agli errori umani, un bene perché è possibile definire aree di memoria condivise tra più variabili senza l'utilizzo delle "union"
3. Il nostro mega vettore. Unica "pecca", con l'XC8 non è possibile inizializzare le varibili con scope global, a meno di definirle "const", ma in questo caso esse non sarebbero modificabili runtime e verrebbero harcoded nella NVM del PIC e non più nella RAM. Ciò esula da questo articolo. I vettori più grossi di un banco vengno allocati automaticamente su più banchi e con "cognizione di causa" per cui non c'è bisogno di puntatori di appoggio.

Uno sguardo alle watch:

Variabile "a": in quanto int definita con un parallelismo a 8 bit, questa occupa 2 byte di memoria. Il Linker sembra essere molto più efficiente del suo predecessore: la piazza veramente alla prima posizione di memoria disponibile con questo span, 0x200 nel nostro caso.

Variabile "b": alla posizione richiesta.

Variabile "c": della dimensione e alla posizione richieste.

Conclusioni

L'XC8 sembra essere più efficiente e "intelligente" del suo predecessore. Esso ha forti influenze dal HI-Tech C Compiler e la nuova IDE MPLAB X ne tira fuori quasi tutte le sue potenzialità, come abbiamo visto con il "RAM ranges" che non fa altro che generare un opzione di linking utilizzata in fase di build. Analogamente vi è, nelle proprietà del progetto, una sezione GUI quasi per ogni opzione di compilazione e linking messe a disposizione dall'XC8.

Non bisogna però cadere nell'errore di snobbare il C18: le conoscenze più radicate e fondamentali di un programmatore o un softwarista in genere, nonché l'importantissima capacità di risolvere i problemi o aggirare le limitazioni delle GUI, derivano sempre dalla conoscenza delle piattaforme dalle quali le nuove versioni derivano.

Qualunque commento sarà ben accetto e, visto che si tratta del mio primo articolo, le critiche lo saranno ancora di più.

Fonti

MPLAB C18 C Compiler User's Guide

MPLAB XC8 C Compiler User's Guide

PIC® 18C MCU Family Reference Manual (DS39500)

Datasheet del PIC18F47J53

Esperienza personale :)