Introducció.

L'objectiu d'aquest document és recollir aquells aspectes més rellevants de la programació de codi en "C" per a microcontroladors de la família MCS '51 fent servir el compilador 'ICC' de 'IAR Sytems'.

No volem, en cap cas, substituir una 'bona literatura' com el 'manual del usuari del compilador', el document que descriu el 'Sistema de desenvolupament DS-552' ni un llibre o 'datasheet' que descrigui els micros. Volem explicitar que els documents mencionats anteriorment són les eines que haureu de fer servir més sovint i que, aquesta 'guia' és només una introducció que permet 'començar a programar' sense invertir molt de temps en consultes a manuals extensos.

El llenguatge C, com tots els llenguatges d'alt nivell, ofereix al programador una sèrie de 'facilitats' a l'hora de codificar algorismes (abstracció, estructuració, portabilitat...). El principal motiu pel que hem triat aquest llenguatge és 'aparcar' l'assemblador i aprofitar la 'comoditat' a l'hora d'estructurar i escriure programes. Això ens permet fixar l'atenció en algorismes 'més grans', en quant a mida, evitant que la codificació d'aquests sigui 'avorrida' sense perdre l'accés als recursos de la màquina.

1) La sintaxi del 'C'.

El llenguatge C té una sintaxi ben definida i s'ha arribat a alguna mena d'estandardització encara que existeixen fortes discrepàncies en quant a diferents 'interpretacions' (o implementacions). Aquests detalls no ens han de preocupar per a la nostra feina.

Per començar a entendre la sintaxi d'aquest llenguatge res millor que prendre el 'Llenguatge C' de Joan Surrell en el que, a demés d'una introducció al llenguatge algorísmic, hi trobareu una breu però completa descripció de la sintaxi així com exemples clarificadors i referències a bibliografia.

2) El Compilador ICC de IAR Systems (versió 4).

Aquest 'paquet' ofereix, juntament amb el assemblador 'A8051' i l'enllaçador 'XLINK', un sistema de desenvolupament per microcontroladors de la família MCS '51 en 'ANSI C'. Permet generar codi altament optimitzat i s'acompanya d'extensions del llenguatge per accedir als recursos específics del maquinari des del codi font en C. S'acompanya d'una extensa documentació i exercicis 'tutorials' fent-lo molt adient per a desenvolupar aplicacions basades en aquests micro-controladors.

D'ara endavant suposarem que esteu familiaritzats amb l'arquitectura d'aquesta família i els conceptes bàsics de la programació d'aquests dispositius.

La línia de comandes bàsica per compilar un codi font escrit en C és la següent:

```
ICC8051 -o nomfitxer.R03 -a nomfitxer.S03 -l nomfitxer.LSC -q -mt -z nomfitxer.C
```

Això compila un ".C" amb el model de memòria tiny, genera un llistat ".LSC" i un assembler intermig ".S03", a més del fitxer objecte ".R03".

Si voleu saber què fan les altres o bé quines possibilitats hi ha crideu-lo sense paràmetres.

```
ICC8051
```

Si voleu consultar (i així volem que ho feu) les opcions de la línia de comandes ho trobareu a la secció 4.1 del manual (hi ha una còpia al laboratori), plana 274.

Per enllaçar el codi objecte (reubicable) executeu:

```
XLINK -x -l nomfitxer.LSX nomfitxer_1.R03 ... nomfitxer_n.R03 -f fitxer.XCL -o nomfitxer.A03
```

Això enllaça tots els fitxers ".R03" (objectes) en un ".A03" fent servir el fitxer de control ".XCL" (al ICC hi ha varis "models", no feu servir res "Banked") i genera un llistat ".LSX" (que no interfereix amb el ".LSC" del compilador).

Sense paràmetres també mostra les possibles opcions. La descripció d'aquestes són al manual del XLINK, secció 9.3 del manual, plana 238.

Recomanació: No proveu 'A8051' sense paràmetres.

3) Recursos del maquinari.

Es molt difícil obviar (o 'abstreure') la capa 'hardware' quan programem micro-controladors. Aquests dispositius, sovint, presenten uns recursos (memòria, E/S, velocitat...) molt 'petits'. No estem exercitant sobre algorismes 'portables' destinats a executar-se en grans computadors sinó, ben al contrari, accedint directament al maquinari i examinant els elements i mecanismes que van des de 'la electrònica' a la solució d'un problema. Per aquest motiu coneixem en profunditat el maquinari que estem programant i hem de saber quins recursos i de quina manera els fa servir el compilador.

A) Ús dels registres pel compilador.

El compilador fa servir per emmagatzemar resultats intermitjos els registres 'PSW', 'ACC', 'R0'..'R3', 'B' i 'DPTR'. El contingut d'aquests registres no cal que sigui preservat en rutines escrites en assemblador cridades des del C. Sí és necessari preservar el contingut d'aquests en rutines de servei a una interrupció escrites en assemblador així com qualsevol altre registre. No és aconsellable modificar-los des del C.

El compilador efectua el pas de paràmetres a les subrutines a través dels registres 'R4'..'R7' excepte en els tipus que no són escalars (tipus 'struct' i tipus 'union').

Pel valor de retorn de les funcions també fa servir els registres 'R4'..'R7'.

En ambdós casos es fan servir tants registres com es necessiti, a partir del que calgui, restant en R7 el byte menys significatiu.

Si es passa més d'un paràmetre o bé es necessita 'més espai' (cas d'estructures) el que es passa/retorna és una referència a un bloc (un punter que, com veurem més endavant, necessita 3 bytes).

El flag de Carry serveix com a valor de retorn de tipus booleà (un bit). També es fa servir per a passar el primer paràmetre de tipus bit a una subrutina.

El Punter de Pila (SP) apunta a la pila (nom 'ben triat') que conté les adreces de retorn de les funcions així com resultats temporals usats en l'avaluació d'expressions. La mala utilització d'aquest registre (i la estructura de pila) té resultats 'catastròfics' en l'execució de un programa. Recordem que aquest tipus d'errors (incloent-hi el desbordament de pila 'stack-overflow' són difícils de detectar i comporten molta feina de depuració).

Podeu ampliar aquesta informació a la secció 3.14 (plana 243) del manual.

B) Zones o àrees de memòria.

Directament lligat a les característiques dels microcontroladors distingirem diferents zones de memòria (o àrees).

A la zona de memòria RAM interna compresa entre l'adreça 00h i 7Fh, accessible amb adreçament directe o bé indirecte, ens hi referirem com a 'DATA' i 'IDATA' (depenent del accés). Recordem que es tracta d'una UNICA zona a la que s'hi pot accedir a través de dos modes d'adreçament diferents. Dins aquesta zona hi ha dues regions 'específiques': els bancs de registres i la zona adressable bit a bit. Els bancs de registres comencen a l'adreça 00h i ocupen tants blocs de vuit bytes com bancs fem servir (fins al màxim de quatre bancs de vuit registres). La zona adressable bit a bit SEMPRE es troba entre les adreces 20h i 2Fh i té 'nom propi'; s'anomena 'BITVAR' i aquí s'enllaçaran les variables declarades tipus 'bit'.

L'àrea del Special Function Register (anomenada 'SFR Space') és aquella compresa entre les adreces 80h i FFh i s'hi accedeix UNICAMENT amb adreçament directe. Els noms dels registres del SFR són definits en fitxers capçalera tipus '.H'.

Per aquells dispositius que en tinguin (acabats en '2'), la zona compresa entre les adreces 80h i FFh accessible amb adreçament indirecte (i que es tracta d'una memòria DIFERENT del SFR) formarà part de 'IDATA' (començant en 00h i acabant en FFh).

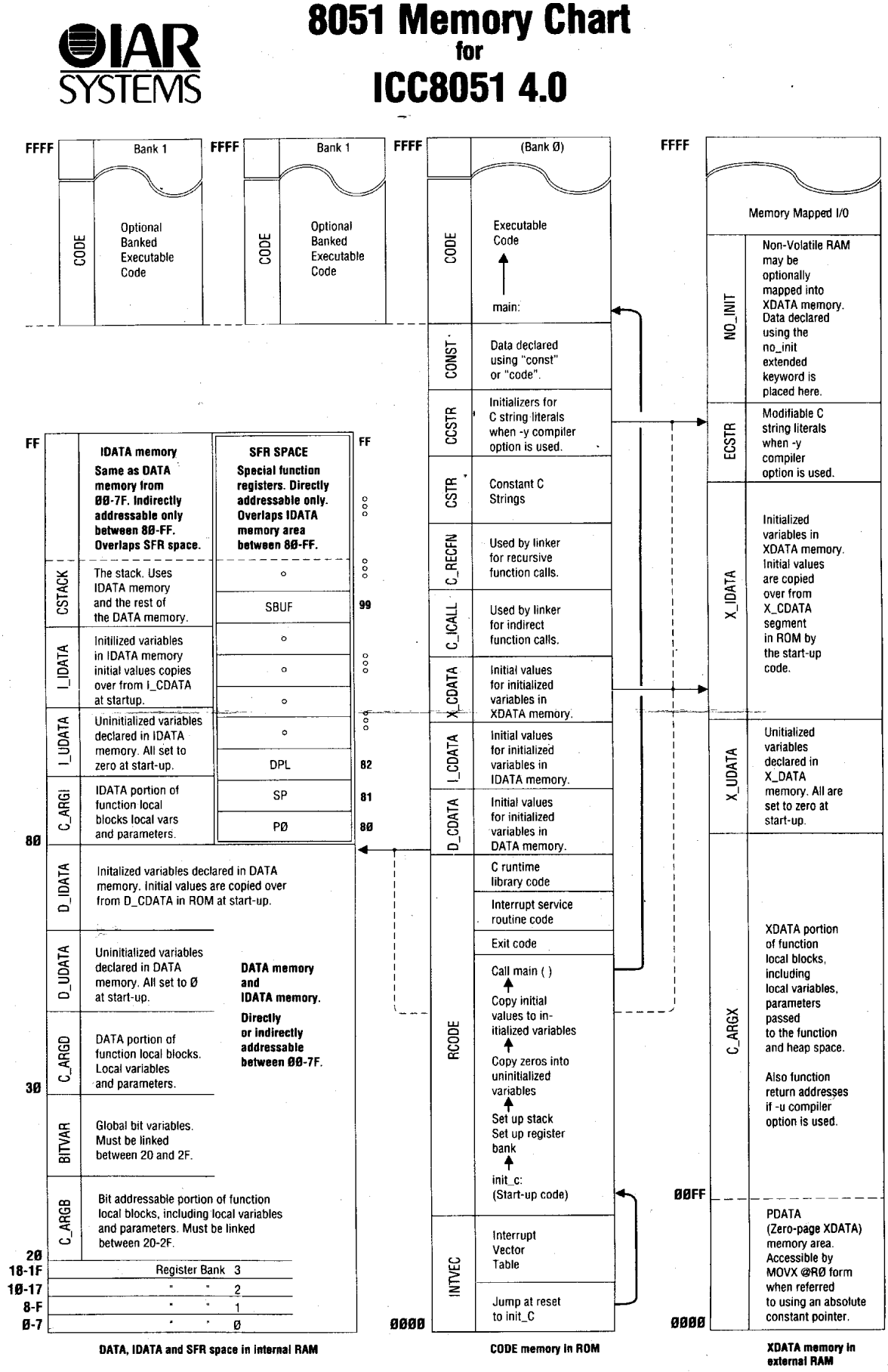
La memòria de codi (interna i/o bé externa, activada amb ~PSEN) l'anomenaren 'CODE' ocupa un espai entre les adreces 0000h i FFFFh. S'hi accedeix pel fet d'executar un programa o bé amb les instruccions 'MOVC'.

La última zona de memòria que ens falta per comentar és la RAM externa. Espai de 64 Kbytes, accessible a través de les instruccions 'MOVX' i controlada pels senyals ~RD i ~WR. S'anomena 'XDATA'.

Qualsevol variable (data object) es pot ubicar en qualsevol d'aquestes zones de memòria des del codi font, en C, a través de les extensions del llenguatge (bàsicament prefixes i directives).

Dins aquestes zones de memòria s'hi defineixen 'segments' atenent les característiques de la informació que contenen. Els diferents segments de cada àrea s'enllaçaran contiguament dins cada zona de memòria. El control d'aquest enllaçat s'efectua a través dels fitxers '.XCL' (XLINK control files).

Al següent gràfic hi trobareu tots els noms d'aquests segments, la seva ubicació i 'mides':



La descripció complerta d'aquests aspectes és a la secció 3.1 (plana 100) del manual.

4) Models de memòria.

Especificar un 'model de memòria', informa al compilador quins recursos del nostre maquinari són disponibles i això permet usar-los d'una forma eficient. Per entendre aquest aspecte cal fer referència, entre altres, al concepte de variables 'locals' i 'globals' (anomenades 'file-level'). En els llenguatges d'alt nivell parlem de 'àmbits' (en el C, allò que va entre dos claus {...}). Direm que un objecte (variable, funció...) és **local** a un àmbit si ha estat definida dins d'aquest. Aquesta 'localitat' té implicacions que, en aquest moment, no ens han de preocupar (visibilitat, 'temps de vida'...). Els **globals** seran aquells objectes definits fora del àmbit superior (en el nostre cas, el 'main()').

Depenent del model de memòria triat, el compilador ubicarà els objectes definits en diferents zones de memòria i generarà el codi necessari per accedir-hi. Les diferents opcions queden reflexades a la taula següent:

Memory model	tiny	small	compact	medium	large	banked
ICC8O51 option	-mt	-ms	-mc	-mm	-ml	-mb
File-level	DATA	IDATA	XDATA	XDATA	XDATA	XDATA
Local	DATA	IDATA	DATA	IDATA	XDATA	XDATA
External RAM	no	no	yes	yes	yes	yes
Code size	64K	64K	64K	64K	64K	>1M
Typical chip	8051	8052	8031/8032	8032	8032	
C library	cl805lt	cl805ls	cl805lc	cl8051m	cl8051l	cl8051b

Aquesta 'ubicació' és la que assignarà el compilador, per defecte, segons el model de memòria triat. Es pot 'forçar' a ubicar un objecte en qualsevol zona de memòria segons ens convingui. Això es fa amb el prefix adient a l'hora de declarar l'objecte ('data' 'idata' o bé 'xdata').

Considerem les implicacions en el temps d'accés a una variable (i el codi generat) depenent la zona on s'ubiqui. Davant la declaració:

```
int i;          /* int són 16 bits */
i = 530;        /* (530 = 10 0001 1000 b = 0x218) */
```

El codi generat i el temps d'execució, depenent de la zona de memòria on s'ubica la variable 'i' és el següent:

Zona de memòria	DATA	IDATA	XDATA

Codi generat	MOV i,#2
	MOV i+1,#18
		MOV R0,#i
		MOV @R0,#2
		INC R0
		MOV @R0,#18
			MOV DPTR,#i
			MOV A,#2
			MOVX @DPTR,A
			INC DPTR
			MOV A,#18
			MOVX @DPTR,A

Bytes/cicles	6/48	7/48	10/108
Rang adreces	0-7F	O-FF	O-FFFF

5) Tipus de dades i la seva representació en memòria.

Els tipus de dades bàsics (derivats del ANSI C) suportats per quest compilador, el seu rang així com l'espai requerit pel seu emmagatzemament és el següent:

Tipus de dada	Espai	Rang
bit	1 bit	0 o bé 1
sfr	1 byte	0 a 255
char	1 byte	0 a 255
enum	1 ó 2 bytes	
unsigned char	1 byte	0 a 255
signed char	1 byte	-128 a 127
short	2 bytes	-32768 a 32767
unsigned short	2 bytes	0 a 65535
int	2 bytes	-32768 a 32767
unsigned int	2 bytes	0 a 65535
long	4 bytes	-2147483648 a 2147483648
unsigned long	4 bytes	0 a 4294967295
float	4 bytes	+- 1.18 E-38 a +-3.39 E+38
double	4 bytes	+- 1.18 E-38 a +-3.39 E+38
long double	4 bytes	+- 1.18 E-38 a +-3.39 E+38
punters/adreces	veure més endavant.	

L'emmagatzemament d'aquests valors, en cas que es requereixi més d'un byte, s'efectua amb el byte més significatiu a l'adreça més baixa. Si heu de consultar al manual, secció 4.2 (plana 327).

Punters.
Els punters, en C, són en essència l'adreçament indirecte del assemblador. Donades les característiques (maquinari i repertori d'instruccions) dels micros MCS '51 el compilador en fa un tractament 'especial' (suposem per 'homogeneïtzar' a nivell de punter). Tots els punters s'emmagatzemen com a "3-byte absolute pointers". El primer byte indica a quina àrea de memòria apuntem i els dos següents són, sempre, un punter absolut de setze bits.

DATA, IDATA	0	Internal RAM
XDATA	1	External data memory
CODE	2	ROM memory
PDATA	3	MOVX @R0 accessed external data memory

Exemple:

```
Amb   "#define XDATA_AMB_OFFSET ((char*) (0x018050));"
  i   "#define XDATA_PORT      (*(char*) (0x018000));"
```

Estem definint un punter a la memòria externa (01) que "comença" a l'adreça "8050" i un punter a l'adreça 8000 (ambdós a memòria externa).

Amb aquest podem fer:

```
 XDATA_PORT=0x20;        /* Escrivim un 20Hex a l'adreça 8000 (externa de dades)*/
for (i=0;i<20;i++)
 XDATA_AMB_OFFSET[i]=0x030; /* Omplim 20 Bytes amb "30Hex" començant per l'adreça 8050 */
```

Cal recordar que la "plana 0" és aquella en la que el byte alt d'adreces és zero (ens podem imaginar la memòria externa com 256 "planes" (adreces altes) de 256 bytes (adreces baixes). A aquesta zona (plana 0) s'hi pot accedir "ràpidament" amb la instrucció específica del micro "MOVX @R0..".

6) Llistats i fitxers de control.

A part de cridar el compilador directament des de la línia de comandes és preferible crear-nos els nostres propis fitxers de procés per lots (.BAT) per tal d'automatitzar les tasques bàsiques de compilat i enllaçat (com hem anat fent fins ara). Existeixen altres 'denominacions' (depenent de factors com el sistema operatiu) per aquesta funcionalitat, si ho preferiu, els podem anomenar 'scripts'. També existeixen 'utilitats' per tal de 'flexibilitzar' aquesta tasca (la utilitat 'MAKE') basades en 'MAKEFILES', fitxers amb descripcions de les tasques a realitzar (inclús de resoldre dependències entre paquets) per automatitzar els processos d'enllaçat de grans projectes. Feu-los servir si ho creieu convenient.

Un altre aspecte molt important és la generació de llistats amb informació relativa al codi generat, especialment els llistats 'intermitjos' en assemblador. Si seguiu les instruccions que us hem recomanat veureu que 'forcem' a generar aquests fitxers (fitxers '.LSC' i '.LSX'). Examineu-los amb detall i interpreteu-ne les 'traduccions' d'estructures algorísmiques (bucles, crides...) a codi assemblador. És un bon exercici que pot 'optimitzar' indirectament la vostra manera de programar.

Per acabar aquest 'capítol' només cal fer referència als fitxers de control del enllaçador ('.XCL'). En aquests fitxers s'efectua el control del enllaçat de tots els segments. Es tracta d'un fitxer de text en el que s'hi indiquen, entre d'altres coses, a partir de quines adreces es comencen a enllaçar els diferents segments, quins poden ser 're-ubicats' i quins han d'anar a una adreça fixe. Per a realitzar les pràctiques de manera satisfactòria haureu de revisar-los.

7) Altres consideracions puntuals.

Per tal que els nostres programes escrits en 'C' s'executin correctament cal que s'hi 'afegeixi' una sèrie de codi que efectua certes inicialitzacions i és cridat quan cal. Bàsicament aquest codi és el format pel 'inicialitzador' i el 'runtime'. El codi que s'encarrega d'inicialitzar tot el necessari és exactament el que hi ha al fitxer 'CSTARTUP.S03'. Aquest codi inicialitza els segments, variables, constants... i acaba cridant al nostre 'main()'. Examineu-lo i interpreteu-lo. D'altrabanda la llibreria de 'runtime' és una llibreria associada al model de memòria amb funcions bàsiques (inicialitzar estructures, efectuar operacions...) que es cridarà i serà enllaçat amb el nostre codi.

Existeix una funció 'exit()' que es crida quan s'acaba el nostre codi. En el nostre cas (el ICC) aquesta funció deixa el micro 'penjat' en un bucle infinit. Així, si el nostre 'main()' acaba (no es tracta d'un bucle infinit) el micro restarà 'aparentment aturat'.

Aquests aspectes fan que el codi generat (per senzill que sigui) tingui una mida considerablement més gran que el seu 'homòleg' en assemblador.

Aquest compilador NO permet la generació de codi 're-entrant'. Això vol dir que una rutina no es pot cridar a si mateixa. Durant les crides, s'emmagatzemen resultats temporals en registres específics (i no a la pila). Dues crides 'niuades' desencadenarien en una 'corrupció' d'aquestes dades apareixent errors en temps d'execució.

Escriure rutines de servei a una interrupció és en 'C' una tasca 'fàcil'. Només cal tenir en compte que no es poden passar paràmetres ni retornen cap valor (no es pot garantir en quin moment són cridades, si es pot parlar de crida, simplement s'executen 'quan toca'). s'han de declarar seguint el següent patró:

interrupt [*vector*] void *nom_rutina*(void);

On 'vector' és l'adreça del vector d'interrupció associat i 'nom_rutina' aquell que li volem donar. El compilador s'encarrega de preservar els registres necessaris (si es fan servir dins la nostra rutina) i de restaurar-los abans de retornar. Recordeu que la única manera d'accedir a dades del programa principal (o retornar valors) és a través de variables globals. Cal esmentar que el fet que aquest compilador no sigui 're-entrant' fa que, des de una RSI, NO es pugui cridar a cap rutina que pogués estar executant-se en el moment de produir-se la interrupció.

8) Eficiència en C.

El llenguatge C ofereix al programador certs avantatges enfront als llenguatges de "baix nivell". Aporta la possibilitat d'estructurar còmodament sense perdre l'accés directe a les capes més "properes" a la màquina (gairebé a qualsevol recurs hardware). El terme "eficiència" es refereix a la mida final i a la "velocitat" d'execució del codi (màquina) generat.

A continuació teniu una 'traducció-síntesi' de la secció 3.10 (plana 197) del manual.

> La principal raó per utilitzar un llenguatge d'alt nivell com el C és allunyar-se del llenguatge assemblador i el codi específic de cada màquina.
>
> **Consells generals:**
>
> Concentreu els vostres esforços d'optimització el aquells llocs on hi hagi una repercussió més gran en la mida i la velocitat. Recordeu que "doblar la velocitat" d'execució d'una rutina que "ocupa" el 2% del total del temps d'execució, només aporta un increment "global" del 1%.
>
> Emprant eines de simulació (com el C-SPY) que ofereixen la possibilitat d'acotar cicles d'execució per diferents porcions de codi ens permetrà veure en quines d'elles hem d'invertir més esforç. Aïllant codi en mòduls, podem veure la mida de cadascun d'ells i advertir seccions desmesuradament grans.
>
> En cap cas, "trucs" o "receptes màgiques" poden substituir l'ús del algorisme òptim per a cada tasca. La majoria dels problemes ja han estat resolts i documentats, un o dos dies de "recerca" en "bones" publicacions poden estalviar setmanes de maldecaps.
>
> Conèixer els recursos del processador és bàsic. El '51 no té suport per operacions matemàtiques amb signe. Feu servir variables 8bit "char" sempre que sigui possible en comptes de variables 16 bit "int".

El pre-processador efectuarà operacions matemàtiques bàsiques sobre les constants, estalviant espai i temps al "run-time". En expressions "compostes" agrupeu les constants per tal de que pugin ser reconegudes i tractades. Per exemple, és preferible fer servir "j=i+k+(CONST1+CONST2);" en comptes de "j=i+CONST1+k+CONST2;"

El codi necessari per una crida a funció, és aproximadament una constant més un increment lineal per cada paràmetre passat. Així una crida amb molts paràmetres "triga" més que una amb pocs. També una "repetició de crides" a una funció requereix més codi que una "rèplica" de codi de funcions "in-line".

Els bucles requereixen estructures de control per actualitzar i comparar els comptadors d'iteració. Per aquesta raó, "desenrotllar" bucles en codi lineal augmenta notablement la velocitat en detriment de la mida del codi generat. Per exemple, usant cinc instruccions d'assignació "in-line" per omplir una petita matriu serà molt més ràpid que assignar amb un índex d'un bucle.

Com que les "Macros" no suposen temps de crida, re-escriure petites funcions com a MACROS, ens estalviarà temps d'execució.

Empreu els operadors "++" i "--" en comptes de "+1", fan servir instruccions INC i DEC en comptes d'efectuar la suma d'una unitat (amb el conseqüent estalvi).

Efectueu els "tests més ràpids" primer. La resta d'un test no s'avalua si la primera condició ja és certa (pel cas d'una OR) o bé falsa (en cas d'una AND).

És preferible multiplicar per potències de 2 en comptes d'utilitzar el "vell recurs" del shift. El pre-processador ja ho fa i no cal recórrer a aquest mètode. Per "desplaçar 4" cal iterar, mentre que el processador farà servir SWAP (nibbles).

Punts específics del ICC8051 Versió 4.0:

Fer servir el model de memòria més petit possible i, si és necessari, re-definir el segment per defecte d'algunes variables (adequadament triades).

Triar l'àrea de memòria més adient per a les variables (data, idata, xdata).

Hi ha directives de compilació com -s (Speed) i -z (siZe) que optimitzen en mida o bé velocitat.

El recurs del assemblador. Hi ha una creença errònia en que l'assemblador és generalment ràpid. Encara que sigui "més directe", només cal re-escriure en assemblador seccions crítiques.. Típicament només cal re-escriure entre el dos i el deu per cent del codi total per aconseguir la millora desitjada.

Conclusió.

La realitat d'aquesta història és identificar els punts crucials del teu codi i examinar el codi generat pel compilador. Si algunes construccions generen molt de codi, intenteu redefinir-lo d'una altra manera per veure si es pot trobar un "esquema" que el compilador "manegui" més bé.

Totes aquestes tècniques només són un punt de partida per "escriure codi més eficient en C per microcontroladors". Per ampliar el tema podeu consultar (com sempre la bona literatura) "Efficient C" de T. Plum (Plum Hall, 1985) i "Writting Efficient Programs" de J.L. Bentley
(Prentice Hall, 1982).