Az alapötlet **2**

Felépítés **2**
COG 3
HUB 5
I/O lábak 6
Rendszerszámláló 6
CLK regiszter 6
Közös memória (Main Memory) 8
Közös RAM (Main RAM) 8
Szemaforok (Locks) 9
Közös ROM (Main ROM) 9
Karakter készlet ($8000-$BFFF) 9
Logaritmus és anti-logaritmus táblák ($C000-$CFFF és $D000-$DFFF) 11
Szinusz tábla ($E000-$F001) 11
Boot Loader és Spin Interpreter ($F002-$FFFF) 11
Lássuk egyben az egészet! – A Propeller teljes blokkvázlata 12
A Propeller elindítása 13
A Propeller leállítása 14

Lábkiosztás **15**

Specifikációk **16**

Hardver kialakítás **17**
A minimál hardver 17
A Parallax cég gyakorló panele 18

Programozás **19**
Spin nyelv 19
Objektumok 19
Blokkok 20
Propeller Assembly 20
Utasítások felépítése 21
Utasítás végrehajtás hatása a flag-ekre és a célterületre 21
Utasítás végrehajtási feltételek 22
Címzés 23

A Parallax Propeller ismertetése

Az alapötlet

A propeller csipet arra tervezték, hogy nagy sebességű adatfeldolgozást biztosítson beágyazott rendszerek számára, mindezt alacsony fogyasztással és kicsi fizikai méretben.
A Propeller flexibilitását és teljesítményét nyolc processzor adja, amelyeket COG-oknak nevezünk. A COG-ok képesek egymástól független illetve egymással kooperáló taszkokat futtatni. A tok Propeller assembly és Spin nyelven is programozható. Mindkét nyelvet a Parallax cég fejlesztette ki és számos újdonságot tartogatnak számunkra. A csipet úgy alkották meg, hogy a fejlesztőnek egyszerű dolga legyen, ezért a következő megoldásokat vetették be:

- A memória címzést úgy alakították ki, hogy ne legyen szükség memória lapok használatára. Ezzel gyorsabbá és kényelmesebbé tették az alkalmazások fejlesztését.

- Az aszinkron események kezelését egyszerűbben oldották meg, mint a megszakítást használó eszközöknél. Tehát itt nem használnak megszakítást, minden periféria kezeléséhez külön COG-ot, vagy COG-okat használnak, ezzel biztosítják a perifériák gyors kiszolgálását. A perifériák nincsenek hardveresen kialakítva a mikrovezérlőn belül, mint azt már sok típusnál megszokhattuk. Általános célú I/O portok vannak és mikrovezérlőben szoftveresen kell megoldani perifériáink kezelését.

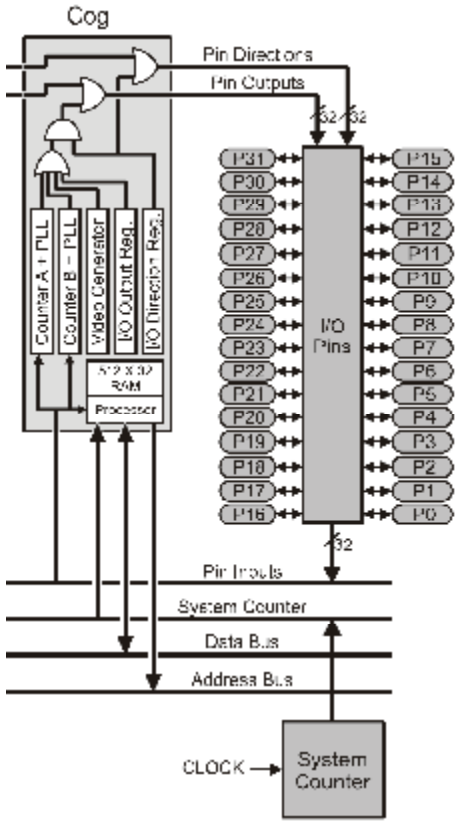
- Gépi kódban minden utasítás elé tehetünk futási feltételt és opcionálisan megjelölhetjük –az utasítás mögött- az eredmény keletkezésének módját. Így a többszörös elágazásokat használó kritikus sebességű programrészek időzítése egységesebbé válik, illetve az eseménykezelők kevésbé lesznek hajlamosak jitterre.

Felépítés

A tok nyolc önállóan működő processzort (COG) tartalmaz, amelyeket 0-7-ig számozunk. A COG-ok használhatnak saját, illetve megosztott erőforrásokat. A saját erőforrásokat a COG bármikor elérheti, a megosztott erőforrásokat egy időben csak egy COG érhet el. A közös erőforrások használatának a vezérlését a HUB végzi.

COG

A nyolc COG felépítése pontosan megegyezik és egymástól függetlenül képesek működni.

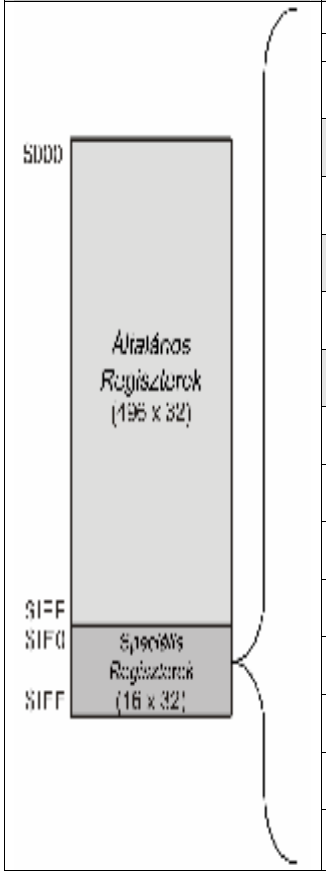


Minden COG tartalmaz:

- Egy processzort
- 512x32bit RAM –ot (COG RAM)
- Két számlálót PLL-el
- Egy videó generátort
- Speciális regisztereket
  (az 1. táblázatban láthatók)

Mind a nyolc COG közös órajelet kap, így egyszerre ugyanabban az időben végeznek műveleteket az éppen aktív COG-ok. Mindegyik hozzáférhet a közös erőforrásokhoz (I/O lábak, Közös RAM/ROM, Rendszerszámláló). A COG-ok programból elindíthatók és leállíthatók, képesek taszkokat futtatni egymástól függetlenül, illetve egymással együttműködve a közös memórián keresztül.
Amikor egy COG elindul, akkor a COG RAM $000-$1EF feltöltődik a közös RAM-ból és a speciális regiszterek kinullázódnak ($1F0-$1FF). Betöltés után a COG RAM $000 címén lévő utasítás végrehajtása kezdődik. A program végrehajtása mindaddig folytatódik, amíg valamelyik COG (akár saját maga) le nem állítja, vagy újra nem tölti, illetve nem reseteljük a tokot.

1. táblázat - A COG RAM kiosztása és a speciális regiszterek

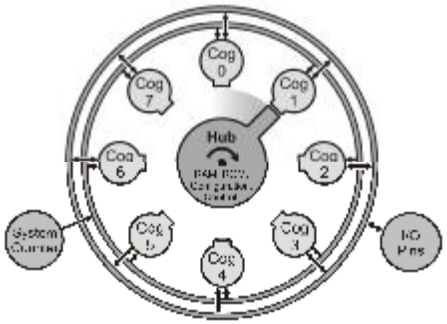
A COG RAM kiosztása	Cím	Név	Típus	Leírás
	$1F0	PAR	Csak olvasható	Boot paraméter
	$1F1	CNT	Csak olvasható	Rendszer számláló
	$1F2	INA	Csak olvasható	Bemeneti regiszter a P31-P0 portokhoz
	$1F3	INB	Csak olvasható	Bemeneti regiszter a P63-P32 portokhoz
	$1F4	OUTA	Írható/olvasható	Kimeneti regiszter a P31-P0 portokhoz
	$1F5	OUTB	Írható/olvasható	Kimeneti regiszter a P63-P32 portokhoz
	$1F6	DIRA	Írható/olvasható	Irány regiszter a P31-P0 portokhoz
	$1F7	DIRB	Írható/olvasható	Irány regiszter a P63-P32 portokhoz
	$1F8	CTRA	Írható/olvasható	Az **A** számláló vezérlő regisztere
	$1F9	CTRB	Írható/olvasható	A **B** számláló vezérlő regisztere
	$1FA	FRQA	Írható/olvasható	Az **A** számláló frekvencia regisztere
	$1FB	FRQB	Írható/olvasható	A **B** számláló frekvencia regisztere
	$1FC	PHSA	Írható/olvasható	Az **A** számláló fázis regisztere
	$1FD	PHSB	Írható/olvasható	A **B** számláló fázis regisztere
	$1FE	VCFG	Írható/olvasható	Videó konfiguráció regiszter
	$1FF	VSCL	Írható/olvasható	Videó arányregiszter

A megjelölt **INB**, **OUTB**, **DIRB** regiszterek a későbbi fejlesztés számára vannak fenntartva. A jelenleg gyártott eszközökön csak egy darab 32 bites port van (**PORT A**).

Mindegyik speciális regiszter három módon érhető el:

- Fizikai címmel (pl.: **MOV $1F4, #$FF**)
- Előre definiált név segítségével (pl.: **MOV OUTA, #$FF**)
- A 16 elemű (0-15) SPR regisztertömb segítségével (pl.: **SPR[$4]:=$FF**)

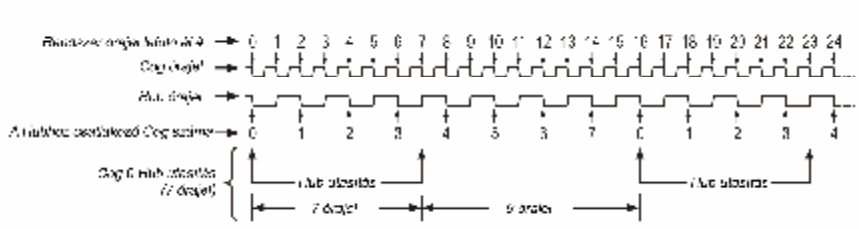
HUB



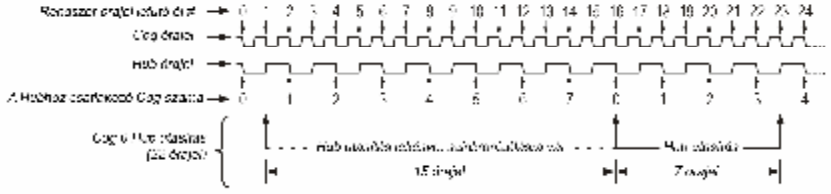
Az erőforrások megfelelő elosztásáért a HUB felel. Mivel a kölcsönösen egymást kizáró (megosztott) erőforrásokat egyszerre csak egy COG használhatja, ezért a HUB időszeleteket ad COG-oknak, amelyben elérhetik ezeket. A COG-ok sorban egymás után (COG0..7) kapják meg ezen erőforrások elérési jogát, majd a COG7 után természetesen megint a COG0 következik. Az éppen nem működő COG-okat nem hagyja ki a HUB, ezért mindig ugyanannyi idő alatt ér körbe. A körkörös vezérlés miatt hasonlították a repülőgép légcsavarjához és innen kapta a Propeller chip a nevét. A HUB és a busz, amit vezérel, a rendszer órajel frekvenciájának felével működik. Ez azt jelenti, hogy minden egyes COG 16 rendszer órajel periódusonként képes az egymást kölcsönösen kizáró erőforrásokhoz hozzáférni. Az assembly HUB utasítások lefuttatásához 7 órajelre van szükség, előbb azonban a HUB-nak az adott COG-hoz kell érnie. Ehhez maximum 15 órajelet kell várnunk (16-1 → éppen elhaladtunk a kérdéses COG mellett) és még 7-et az utasítás végrehajtásához. Tehát egy HUB utasítás végrehajtásához 7..22 órajel szükséges.

A következő ábrákon különböző esetek láthatók.

A legjobb eset, ilyenkor a HUB pont az adott COG-on áll:



A legrosszabb eset, amikor épp túlhaladtunk a COG-on:



I/O lábak

A Propellernek 32 I/O lába van, amelyből 28 teljesen általános felhasználású. A maradék négy I/O lábnak (28-31) speciális szerepe van a mikrovezérlő elindításánál, utána normál I/O lábként használhatók. Mivel minden I/O láb közös erőforrás, ezért ezeket a COG-ot bármikor elérhetik, akár egyszerre is. Ezért a programozónak nagyon oda kell arra figyelni, hogy miként használja ezeket.
Minden COG-nak van saját 32 bites I/O irány és kimeneti regisztere. Az összes COG I/O irány regisztere logikai VAGY kapcsolatban van egymással. Az előzőhöz hasonlóan, az összes I/O kimeneti regiszter is logikai VAGY kapcsolatban van egymással.

A portok kezelésére a következő szabályok vonatkoznak:

!

- Egy láb csak akkor lehet bemenet, ha egy aktív COG sem állította be kimenetnek.
- Egy láb csak akkor lehet alacsony szinten, ha minden COG alacsony szintre állította azt.
- Egy láb magas szinten lesz, ha bármely COG magas szintre állítja.
- Ha egy COG nem fut, akkor annak az irány és a kimeneti regisztere 0, így nem szól bele a portok vezérlésébe.

Minden COG-nak van bemeneti regisztere is, ami látszólagos, mindig a láb logikai aktuális állapotát találjuk benne, függetlenül attól, hogy bemenet vagy kimenet.

Rendszerszámláló

A Rendszerszámláló egy globális, csak olvasható, 32 bites regiszter, amely a Propeller elindítását követően minden egyes Rendszer órajel hatására megnő eggyel. A rendszerszámláló közös erőforrás, tehát minden COG tudja olvasni bármikor (a CNT regiszteren keresztül) - akár egy időben is. A Rendszerszámláló értékét felhasználva eseményeket időzíthetünk, ezt használja fel a WAITCNT utasítás is. A Rendszerszámláló nem törlődik ha programot töltünk egy COG-ba, hiszen akkor a többi programot futtató COG-ok egymáshoz képesti időzítését megzavarnánk . A Rendszerszámláló csak bekapcsolásnál és újraindításnál törlődik.

CLK regiszter

A CLK regiszter a Rendszer órajel előállításának módjáért felelős. Itt kell beállítani az órajel forrását, tulajdonságait. Ez a regiszter vezérli a belső RC oszcillátort, a PLL-t, a kristály oszcillátort és az órajel kiválasztó áramköröket. Fordítási időben a **_CLKMODE** direktíva segítségével állíthatjuk be, valamint futási időben a **CLKSET** utasítással változtathatjuk meg. Ha a CLK regiszterbe írunk, akkor a változás csak kb. 75µs múlva fog bekövetkezni.
Amikor a regiszter értéke megváltozik, az órajel előállításának módja, illetve Rendszer órajel frekvenciájának értéke a közös RAM-ba íródik. Ezekre a CLKMODE és CLKFREQ függvényekkel lehet hivatkozni, amelyek hasznosak lehetnek időzítések számításánál. Ha lehetséges, ajánlott a CLKSET paranccsal átállítani az órajelet, mivel ez az utasítás korrekt módon felülírja a CLKMODE és a CLKFREQ értékeket, így kisebb a tévedés esélye.

A CLK regiszter felépítése								
Bit	7	6	5	4	3	2	1	0
Név	RESET	PLLENA	OSCENA	OSCM1	OSCM0	CLKSEL2	CLKSEL1	CLKSEL0

RESET bit (7)	
Bit	Hatás

0	Semmi
1	Újraindítja a chipet. Ugyanaz, mint a hardver reset. A REBOOT spin parancs egybe állítja ezt a bitet.

PLLENA bit (6)	
Bit	Hatás
0	Letiltja a PLL áramkört
1	Engedélyezi a PLL áramkört

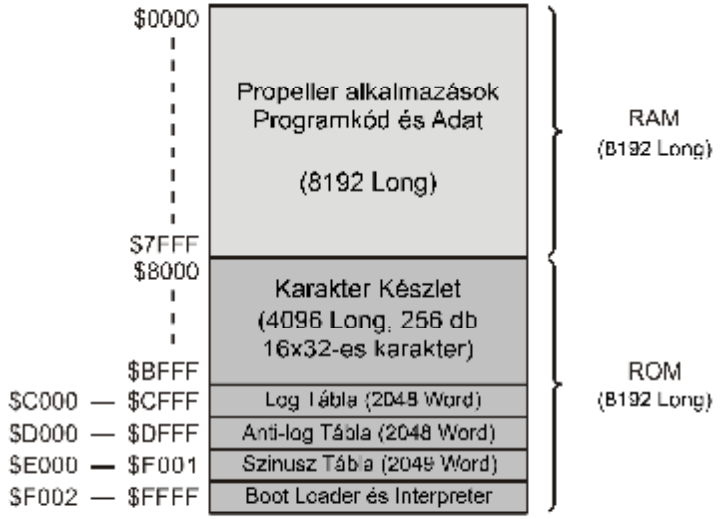
OSCENA bit (5)	
Bit	Hatás
0	Letiltja a kristály oszcillátort
1	Engedélyezi a kristály oszcillátort.

OSCMx bitek (4-3)					
OSCMx		_CLKMODE	XOUT	XIN/XOUT	Frekvencia tartomány
1	0	beállítás	ellenállás	kapacitás	
0	0	XINPUT	∞	6 pF	DC - 128MHz (külső)
0	1	XTAL1	2 kΩ	36 pF	4 - 16 MHz (kristály/rezonátor)
1	0	XTAL2	1 kΩ	26 pF	8 - 32 MHz (kristály/rezonátor)
1	1	XTAL3	500 Ω	16 pF	20 - 60 MHz (kristály/rezonátor)

CLKSELx bitek (2-0)						
CLKSELx			_CLKMODE	Rendszer	Forrás	Megjegyzés
2	1	0	beállítás	órajel		
0	0	0	RCFAST	Kb. 12 MHz	Belső	Külső alkatrész nem szükséges. (8-20MHz között)
0	0	1	RCSLOW	Kb. 20 kHz	Belső	Nagyon alacsony fogyasztás. (13-33kHz között)
0	1	0	XINPUT	XIN	OSC	Az OSCENA bitnek 1-nek kell lenni
0	1	1	XTALx és PLL1x	XIN · 1	OSC+PLL	Az OSCENA és a PLLENA bitnek 1-nek kell lenni
1	0	0	XTALx és PLL2x	XIN · 2	OSC+PLL	Az OSCENA és a PLLENA bitnek 1-nek kell lenni
1	0	1	XTALx és PLL4x	XIN · 4	OSC+PLL	Az OSCENA és a PLLENA bitnek 1-nek kell lenni
1	1	0	XTALx és PLL8x	XIN · 8	OSC+PLL	Az OSCENA és a PLLENA bitnek 1-nek kell lenni
1	1	1	XTALx és PLL16x	XIN · 16	OSC+PLL	Az OSCENA és a PLLENA bitnek 1-nek kell lenni

Közös memória (Main Memory)

Tokon belül található a közös memória, nagysága 64 kbyte (16k long). Ez a blokk megosztott erőforrásként érhető el a HUB-on keresztül. 32 kbyte RAM-ból és 32 kbyte ROM-ból áll. A 32 kbyte-os RAM szabadon felhasználható, a rendszer indításakor feltöltődik a külső 32 kbyte-os I2C epromból, vagy a soros vonalról. A ROM területen találhatóak azok az erőforrások, amik igazán erőssé teszik a Propellert. Ezek az erőforrások a Karakter Készlet, a Logaritmus Táblázat, az Anti-logaritmus Táblázat, a Boot Loader és az Interpreter.



Közös RAM (Main RAM)

A közös memória első felében helyezkedik el ($0000-$7FFF), ide kerülhetnek a programok és az adatok. Amikor programot töltünk a Propellerbe a külső epromból, vagy soros vonalon keresztül, akkor az egész memória terület felülíródik. Az első 16 byte ($0000-$000F) tartalmazza az inicializálási adatokat, amit Boot Loader és az Interpreter használ. Ebből következik, hogy a programunk $0010 címen kezdődhet. Az adatok és a stack a program után foglalhatnak helyet, egészen $7FFF-ig. Az inicializálási adatok közül kettő fontos lehet a számunkra. A **CLKFREQ** (LONG[$0]) az órajel frekvenciáját Hz-ben, valamint a **CLKMODE** (BYTE[$4]) az oszcillátor beállítását tartalmazza. A **CLK** regiszter módosításakor mindkét memóriahelyet felül kell írni, ezért célszerű a **CLOCKSET** utasítást használni, mivel az automatikusan megteszi.

Szemaforok (Locks)

8 szemafor bit van a Propellerben ami az egymást kizáró közös perifériák kezelését segíti. Ha egy memória-tartományban kettő vagy több COG dolgozik egy időben és egy bájtnál nagyobb szélességű adatokat használunk (word, long), akkor ezeknek az adatoknak a kezeléséhez több írási/olvasási műveletre van szükség. Megeshet az, hogy miközben egy adatot épp kiolvasunk, egy másik COG felülírja az egyik bájtját. Ez a jelenség hibás íráshoz és kiolvasáshoz vezethet. A szemaforok segítenek megoldani a problémát. Ezek a bitek egyszerű flag bitekként működnek, amivel a COG-ok üzenni tudnak egymásnak.
A szemafor bitek globálisan elérhetők a HUB-on keresztül, a következő HUB utasítások által: **LOCKNEW**, **LOCKRET**, **LOCKSET**, **LOCKCLR**. Pont azért mert csak HUB utasítással érhetők el, egyszerre csak egy COG használhatja ezeket. A HUB nyilvántartást vezet arról, hogy melyik szemaforok vannak éppen használatban és mi az állapotuk.
A COG-ok a következő szemafor műveleteket végezhetik:

Utasítás	Leírás
id:=**LOCKNEW**	Új szemafort igényelése. Ha van szabad szemafor, akkor annak az azonosítójával, ha nincs, akkor -1 –el tér vissza a függvény.
LOCKRET (id)	A korábban igényelt szemafor visszaadása a HUB-nak. A függvény a visszaadni kívánt szemafor azonosítóját várja bemenetként.
LOCKSET (id)	1-re állítja az id azonosítójú szemafor bitet és annak előző értékével tér vissza a függvény
LOCKCLR (id)	0-ra állítja az id azonosítójú szemafor bitet és annak előző értékével tér vissza a függvény

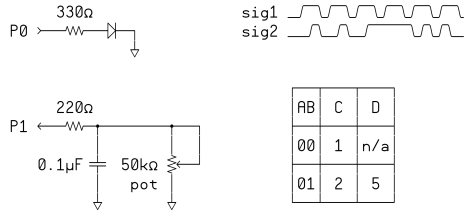
Közös ROM (Main ROM)

A ROM területen találhatóak azok az erőforrások, amik igazán erőssé teszik a Propellert. Ezek az erőforrások a Karakter Készlet, a Logaritmus Táblázat, az Anti-logaritmus Táblázat, a Boot Loader és az Interpreter.

Karakter készlet ($8000-$BFFF)

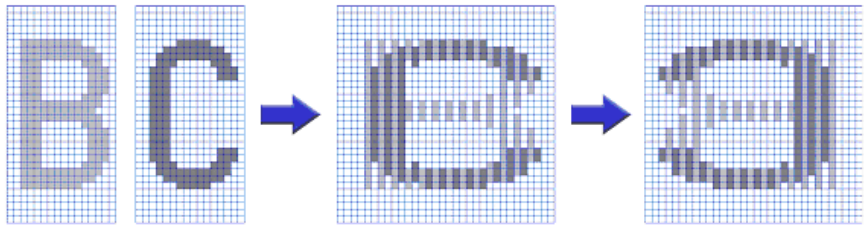
A ROM első felében található a karakter készlet. 256 darab, 32 pixel magas és 16 pixel széles karakter tartalmaz. A karakter készlet az észak amerikai és nyugat európai karaktereken alapul, valamint számos speciális karaktert tartalmaz, ahogy a képen is látható.

A speciális karakterek jól használhatók kapcsolási rajzok, négyszögjelek, gombok, keretek megjelenítéséhez.



AB	C	D
00	1	n/a
01	2	5

Fizikailag a karakterek párosával vannak elhelyezve a ROM-ban, egy pár 32 long-ot foglal el. A karakter párok soronként vannak összefűzve, ami azt jelenti, hogy egy longban a páratlan karakter a páratlan (1,3,5,..,31), páros karakter a páros (0,2,4,..,30) biteket használja.



A karakter definíciók mindegyike ilyen módon van tárolva, mivel a COG videó generátora ezeket közvetlenül tudja kezelni. Azt, hogy a karakterpárok közül a páros vagy a páratlan kerüljön megjelenítésre, a színválasztással tudjuk meghatározni.

Egy karakterpárhoz négyféle színinformáció tartozik:

1. **Páratlan karakter színe**
2. **Páratlan karakter háttérszíne**
3. **Páros karakter színe**
4. **Páros karakter háttérszíne**

Ezeknek a nem megfelelő kezelésével elég vad dolgokat is kaphatunk végeredményül, például egymásra írhatjuk a páratlan és a páros karaktert, vagy csak egy négyzetet látunk a megjeleníteni kívánt karakter helyett.
Léteznek olyan karakterek kódok is, amelyek speciális jelentéssel bírnak, mint például a 9-es Tab, a 10-es soremelés vagy a 13-as kocsi vissza karakterek. Ezeknek a karaktereknek a definíciója eltér a többitől. Speciális négyszínű karakterekként alkalmazhatjuk, amelyeket 3D-s hatású keretek (pl. gombok) éleinek megjelenítésére használhatunk. Egy-egy ilyen karakter 16 x 16-os pixelből áll, ellentétben a közönséges karakterek 16 x 32-es pixelével. Ezek a karakterek a 0-1, 8-9, 10-11 és 12-13 karakterpárok.

Logaritmus és anti-logaritmus táblák ($C000-$CFFF és $D000-$DFFF)

A logaritmus és anti-logaritmus táblázatok segítségével számokat konvertálhatunk át logaritmikus alakba, és vissza.
Ezzel a trükkel jelentősen egyszerűsödik pár - egyébként csak körülményesen megvalósítható - matematikai művelet. A logaritmikus alakkal például a következő műveleteket végezhetjük:

A logaritmikus számmal végzett művelet	Mi történik?
Összeadás	Szorzás
Kivonás	Osztás
Balra eltolás 1-el	Négyzetre emelés
Jobbra eltolás 1-el	Gyökvonás
Szorzás 3-al	Köbgyök-vonás

Logaritmikus alakból visszakonvertálás segítségével kapjuk meg a végeredményt. Az eljárás nem tökéletes, viszont meglehetősen gyors.

Szinusz tábla ($E000-$F001)

A szinusz táblázat 2049 darab előjel nélküli 16 bites mintát tartalmaz 0-tól 90 fokig (I. negyed) , 0,0439 fokos felbontással. A többi negyed ($90° < \alpha < 360°$), valamint a cosinus értékek kiszámításához csak egyszerű matematikai transzformációkra van szükség.

Boot Loader és Spin Interpreter ($F002-$FFFF)

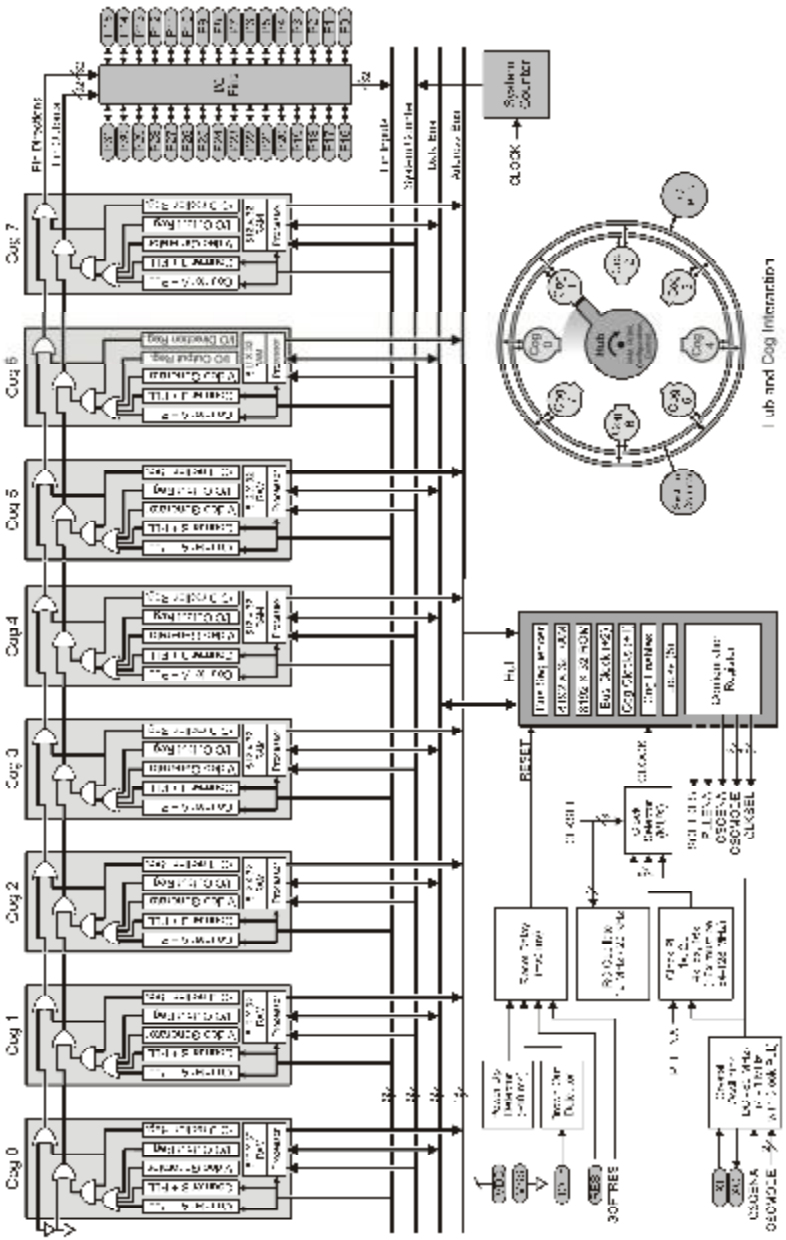
A ROM végében kapott helyet a Boot Loader és a Spin Interpreter. Ezek olyan assembly nyelven írt programok, amelyek a chip működése szempontjából létfontosságúak.

A Propeller bekapcsolásakor/újraindításakor a Boot Loader bekerül a COG0-ba és elindul. Először megvizsgálja van-e soros kapcsolat. Ha van, akkor ezen keresztül tölthetünk a közös RAM-ba. Kérhetjük a tartalom átírását az EEPROM-ba is. Ha nem talál soros kapcsolatot, akkor az EEPROM tartalma töltődik a közös RAM-ba. (Amennyiben EEPROM-ot sem talál, a Propeller kikapcsol.) Amikor a betöltés befejeződött, akkor a Spin Interpreter kerül a COG0-ba (felülírva a Boot Loadert) és elkezdődik a programunk végrehajtása.

A Spin Interpreter feladata a Spin nyelven írt program utasításainak futási időben való értelmezése és futtatása. Hogy több COG is képes legyen párhuzamosan Spin programot futtatni, a Spin Interpreter minden olyan COG-ba betöltődik, ahol szükség van rá.

Lássuk egyben az egészet! – A Propeller teljes blokkvázlata

Az előző részekben végignéztük a Propeller részegységeinek működését, most lássuk egyben az egészet! A következő ábra bemutatja hogy kapcsolódnak a COG-ok a HUB-hoz, milyen módon érthetjük el a közös erőforrásokat, miként állítja elő a chip az órajelet, valamint láthatjuk milyen sorrendben választja ki a HUB a COG-okat.



A Parallax Propeller teljes blokkvázlata

A Propeller elindítása

Bekapcsolás után (+100ms), RESn lábra kapcsolt felfutó él hatására (hardver reset), vagy szoftver reset hatására a következők történnek a Propeller chip-ben:

1. A chip belső oszcillátora alacsony sebességű módban (kb. 20 kHz) indul, 50 ms-ot vár (reset késleltetés), majd gyors módba kapcsol (kb. 12MHz) és betölti a COG0-ba a Boot Loadert, valamint elindítja azt.

2. A Boot Loader a következő feladatokat hajthatja végre ebben a sorrendben:

   a. Megvizsgálja, hogy van-e soros kommunikáció a számítógép és a Propeller között. Amennyiben van, a chip a soros vonalon keresztül betölti a programot a közös RAM-ba, illetve – ha kívánjuk- a külső 32 kbyte-os EEPROM-ba is.

   b. Ha nem talál soros kommunikációt, akkor megnézi, hogy talál-e külső 32 kbyte-os EEPROM-ot (24LC256). Abban az esetben, ha talál, betölti mind a 32 kbyte-ot a közös RAM-ba.

   c. Amennyiben EEPROM-ot sem talál, a Boot Loader és a chip leáll, minden I/O láb bemenet lesz.

3. Ha a 2a, vagy a 2b művelet sikerrel jár – valamint nem kapunk felfüggesztési parancsot a HUB-tól-, akkor a COG0-ba betöltődik a Spin Interpreter és az futtatni kezdi programot a közös memóriából.

Ha a későbbiekben új COG-ot indítunk, akkor csak abban az esetben töltődik bele Spin Interpreter, ha Spin nyelvű programot fogunk futtatni rajta.

A Propeller leállítása

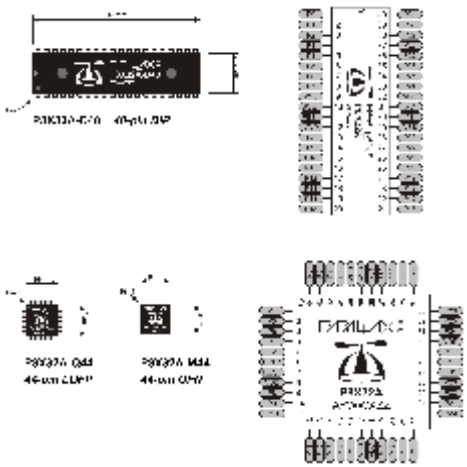
Amikor a Propellert leállítjuk, megszünteti a belső órajelet, ami miatt az összes COG rögtön leáll. Az I/O lábak nagyimpedanciás módba kerülnek. A chip leállítását a következő események idézhetik elő:

1. A tápfeszültség a brown-out határérték alá esik (kb. 2,7 V), amikor a brown-out áramkör engedélyezve van. (A chip feléled, amennyiben a tápfeszültség a brown-out határérték fölé emelkedik és a RESn láb magas szinten van.)

2. A RESn lábra alacsony szintet kapcsolunk.

3. Az alkalmazás kéri az újraindítást (REBOOT parancs)

Lábkiosztás



A P8X32A Propeller chip lábkiosztása		
Láb neve	**Irány (Be/Ki)**	**Funkció**
P0 – P31	Be/Ki	Általános célú I/O port (**Port A**). 30mA-t képes elnyelni, vagy szolgáltatni 3,3V-on. Több lábat párhuzamosítva ezt az értéket 100mA-re emelhetjük. A komparálási szint a tápfeszültség felénél van (1,65V 3,3V-os tápfeszültség mellett) Az alábbi lábak a bekapcsolásnál/resetnél speciális funkciókat látnak el, majd később általános ki/bementekként használhatók: **P28** – I2C SCL csatlakozás egy opcionális külső 32 kbyte-os EEPROM számára **P29** – I2C SDA csatlakozás egy opcionális külső 32 kbyte-os EEPROM számára **P30** – Soros adatvonal TX **P31** – Soros adatvonal RX
BOEn	Be	Brown-out áramkör engedélyezés (alacsony szintre aktív). A VDD vagy VSS lábra kell kapcsolni. Amennyiben a VSS lábra (Föld) kötjük, akkor a RESn láb gyenge kimenetként szolgál – 5 kΩ-os belső ellenálláson keresztül figyelheti a Brown-out áramkör a tápfeszültséget. Ekkora belső ellenállásra azért van szükség, hogy a RESn lábat alacsony szintre tudjuk húzni (külső reset). Ha a BOEn lábat VDD-re kötjük, akkor a RESn láb egy sima Schmitt triggeres CMOS bemenetként működik.
RESn	Be/Ki	Reset (alacsony szintre aktív). Ha alacsony szinten van, akkor minden COG le van tiltva és a kimenetek lebegnek. Az alacsony-magas átmenetet követően 50ms múlva a Propeller chip újraindul.
XI	Be	Kristály bemenet. Ide csatlakoztatható egy külső oszcillátor (ilyenkor az **XO** lábat szabadon kell hagyni), vagy egy kristály egyik kivezetése (ebben az esetben a kristály másik kivezetését az **XO** lábra kell kötni) A CLK regiszterben be kell állítani, hogy melyiket választottuk. Külső ellenállás és kondenzátor alkalmazása nem szükséges.
XO	Ki	Kristály kimenet. Ide a kristály egyik kivezetését csatlakoztathatjuk(ebben az esetben a kristály másik kivezetését az **XI** lábra kell kötni), vagy szabadon kell

		hagyni. A CLK regiszterben be kell állítani, hogy melyiket választottuk. Külső ellenállás és kondenzátor alkalmazása nem szükséges.
VDD	---	3,3 VDC tápfeszültség (2,7 - 3,3VDC)
VSS	---	Föld

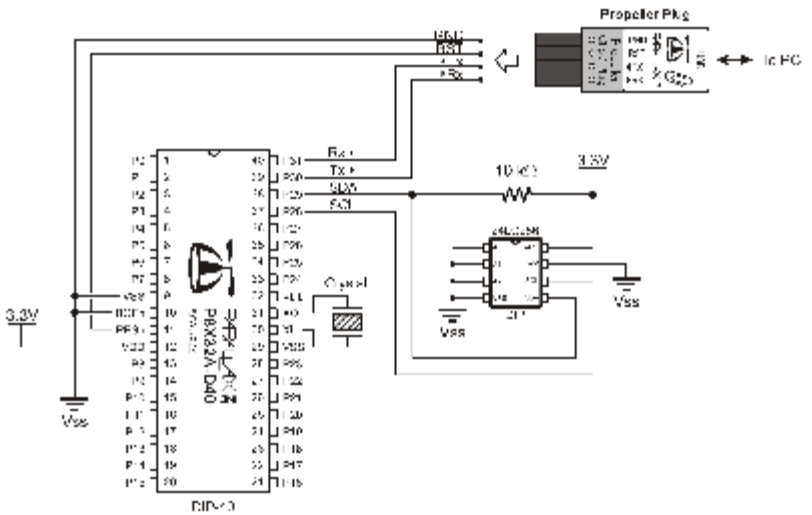
Specifikációk

A P8X32A típusú mikrovezérlő specifikációja	
Tápfeszültség	3,3V DC
Külső órajel frekvencia	DC-től 80MHz-ig
Rendszer órajel frekvencia	DC-től 80MHz-ig
Belső RC oszcillátor frekvencia	Kb. 12Mhz (8-20MHz tartományon belül) vagy Kb. 20Khz (13-33kHz tartományon belül)
Fő RAM/ROM	64 Kbyte, 32Kbyte RAM + 32Kbyte ROM
COG RAM	512x32bit minden COG-ban
RAM/ROM szervezés	Long (32-bit), Word (16-bit), Byte (8-bit) hosszúságú adatok címezhetők.
I/O lábak	32 CMOS be/kimenet, VDD/2 komparálási szinttel.
Áram terhelhetőség I/O lábanként	30 mA
Áram terhelhetőség 8 I/O lábra	100 mA
A tok áramfelvétele 3,3V DC tápfeszültség és 21°C környezeti hőmérséklet mellett	500µA/MIPS (MIPS = f (MHz) / 4 · Aktív COG-ok száma)

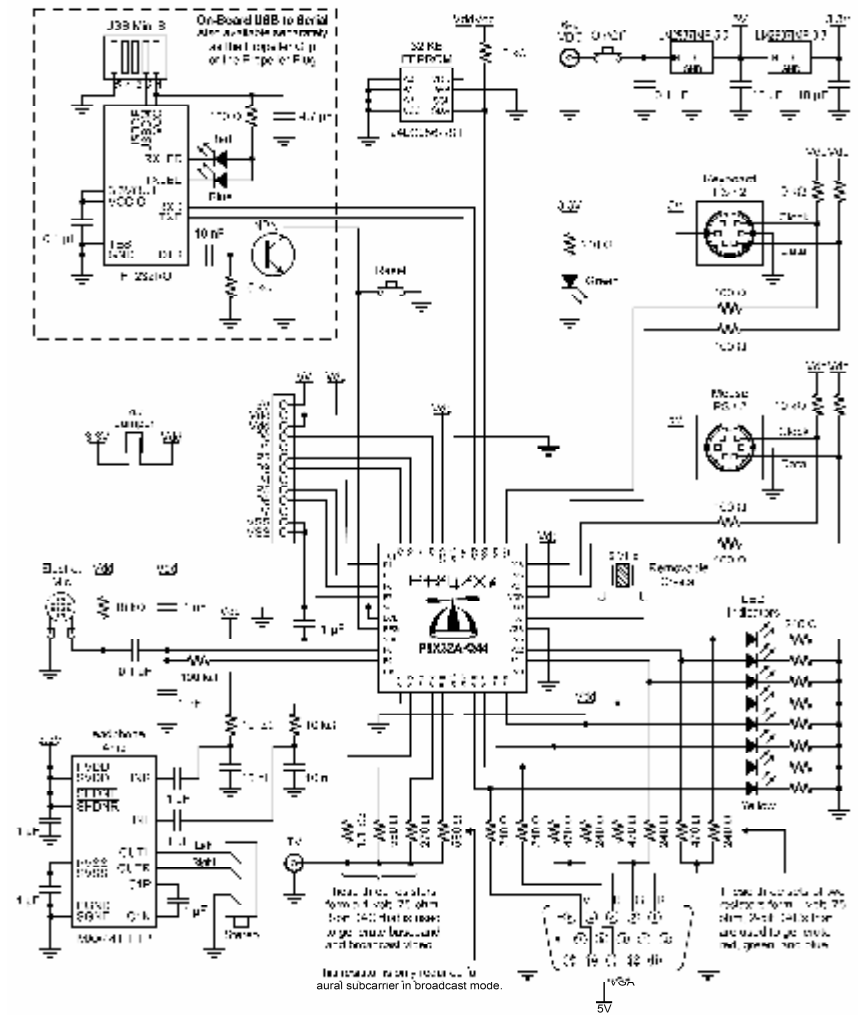
Hardver kialakítás

A minimál hardver

Propeller chipből könnyedén építhetünk áramköröket, hiszen kevés számú külső alkatrész alkalmazásával már működőképes a tok. Az alábbi ábrán látható a Propeller működtetéséhez szükséges minimális hardver. Tulajdonképpen a külső EEPROM is elhagyható, viszont ilyenkor csak a soros vonalon keresztül tölthetünk programot az eszközbe, aminek gyakorlati szempontból nincs sok értelme. A képen látható Propeller Plug valójában egy egyszerű USB/Soros átalakító.



A Parallax cég gyakorló panele



A Parallax cég gyakorlójának kapcsolási rajzán látható, hogy milyen egyszerűen csatlakoztathatók ehhez a chiphez a perifériák. Csupán pár alkatrész szükséges az illesztéshez. Nézzük meg például a VGA kimenetet! Nyolc darab ellenállás és egy csatakozó. Ezzel a megoldással színcsatornánként négy árnyalatot képes előállítani, így összesen 2^{12}=4096 féle színt jeleníthetünk meg. Hogy mit csatlakoztatunk a Propeller 32 I/O lábához, annak csupán a fantáziánk és a portok sebessége szab határt. Ez könnyű, egyszerű és rugalmas fejlesztést tesz lehetővé. A Parallax cég számos előre kidolgozott perifériaillesztési megoldást kínál, eszközéhez – méghozzá ingyen. Ezek egyszerűen használhatók, illetve jó alapot nyújthatnak saját feladataink megoldásához is.

Programozás

A Propeller két nyelven is programozható, ezek a Spin és a Propeller Assembly. Nem kívántam teljes nyelvi referenciát leírni, hiszen az a Propeller Manualban megtalálható (szabadon letölthető a http://www.parallax.com/propeller/ weboldalról) és minimális angol nyelvtudás segítségével használható. A következőkben mindkét nyelv legfontosabb tulajdonságait összegeztem.
Alkalmazásokat Propeller Tool nevű szoftver segítségével készíthetünk. Könnyen, gyorsan, jól áttekinthetően fejleszthetünk ebben a környezetben, viszont semmilyen debuggolási lehetőséget nem kínál. Létezik hozzá egy kezdetleges stádiumban lévő – nem a Parallax által fejlesztett - szimulátor, ami Gear névre hallgat (http://sourceforge.net/projects/gear-emu/). Ebben töréspontok, watch-ok nélkül futtathatjuk alkalmazásunkat. Ezzel a fapados verzióval sajnos elég nehézkes a hibakeresés, de használható. Előnyei között említhető hogy megírták hozzá a VGA és TV plugint, amelynek segítségével a szimulátorunk képes megjeleníteni a programunk által generált képet. Bízzunk benne, hogy továbbfejlesztik…

Spin nyelv

A Spin nyelv egy objektum alapú magas szintű programnyelv. Megalkotója Chip Gracey, egy könnyen használható, egyszerű szintaxist használó nyelv kifejlesztését tűzte ki célul. Felfedezhető ebben a szintaktikában sokféle programnyelv hatása.

Bevezettek két érdekes és új megoldást:

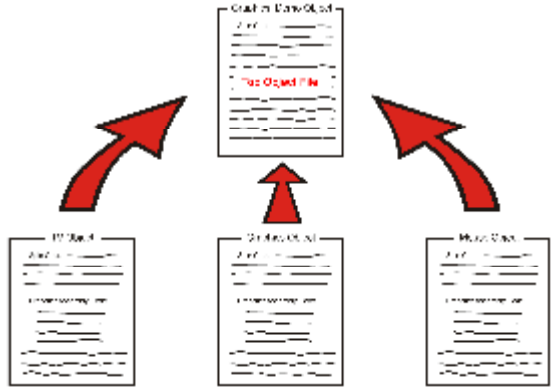
- Programon belül a blokkok határolását a sorbehúzás mértékével oldották meg.
- Kétféle megjegyzést definiáltak, az egyik fajta csak a dokumentációban, a másik csak a forráskódban jelenik meg.

Objektumok

Az objektumok olyan programok, amelyek:

- Önállóan működő egységek
- Egy konkrét feladatot végeznek el
- Újra felhasználhatók más alkalmazásokban

Fordítás előtt ki kell jelölnünk egy ún. Top Object File-t, amely a fordítás kezdőpontja lesz.
Objektumokként kapjuk például az összes gyárilag implementált periféria meghajtó programot. Az objektumok használatával a fejlesztés jelentősen leegyszerűsödik és felgyorsul.
A Spin programokat a fordító ún. byte kódra fordítja. Ezt a kódot a Spin Interpreter dolgozza fel és fordítja ASM utasításokra futási időben. Egy Spin utasítás 20-40 ASM utasítás futtatását is eredményezheti.



Az ábrán megfigyelhető az objektumok hierarchiája. A Graphics_Demo objektum a Top Object File, ami a TV, a Graphics és a Mouse objektumokat használja.

Blokkok

A Spin nyelvű programot különféle blokkok alkotják. Ezek a következők lehetnek:

Blokk Típus	Leírás
CON	Itt definiálhatjuk konstansainkat. A fordításkor fordító behelyettesíti ezeket az értékeket, ezért nem foglal helyet a memóriából.
VAR	Ebben a blokkban definiálhatjuk azokat a változóinkat, amiket a közös memóriában kívánunk elhelyezni. Programindításkor minden itt definiált változó értéke 0 lesz.
OBJ	Objektumok beillesztését valósíthatjuk meg itt.
PUB	A PUB blokktípus jelölővel public-típusú függvényt hozhatunk létre, amely a fájl tartományán kívülről is elérhető azon objektumok számára, amelyek importálják a PUB-ot tartalmazó fájlt. A Spin program végrehajtása mindig a legfelső szinten elhelyezkedő fájl elsőként felfedezhető PUB bejegyzésénél kezdődik.
PRI	A PRI használata megegyezik a PUB-éval, azzal a különbséggel, hogy ez private típusú függvényt hoz létre, ami az adott objektum tartományán kívül nem elérhető.
DAT	A DAT blokk-kijelölő egy adatszegmens kezdetét határozza meg, amely a következő blokktípus-kijelölőig tart. A DAT-szegmenseket adatok és táblázatok definiálására használjuk a SPIN-ben. A DAT-szegmensek ASM-kódok megadására is szolgálnak, amihez deklarálnunk kell egy DAT-szegmenst, majd pedig elhelyezzük az ASM-kódunkat. Ennek az az oka, hogy az ASM-kódot a fordító adatként látja, ezért egyszerűen a végleges objektumba "olvasztja", majd pedig egy adott helyen átadjuk egy COG-nak. Az ASM kód mérete soha nem lehet nagyobb 512 LONG-nál (valójában 512-16 LONG-nál, a COG-modulok memóriájának végén elhelyezkedő regiszterállományok miatt).

Minden blokk a következő blokk kezdetéig tart. A Spin programnak legalább egy PUB blokkot kell tartalmaznia. A különböző típusú blokkokat a Propeller Tool más-más színnel jelzi a könnyebb áttekinthetőség érdekében.

Propeller Assembly

Amennyiben a propellert assembly nyelven kívánjuk programozni, be kell érnünk COG-onként maximum 496 utasítással. Ez abból adódik, hogy egy COG-ban 512 long memória található, amiből 16 long-ot elfogyasztanak a memória végén lévő regiszterek. Minden Propeller utasítás 32 bit (1 long)

helyet foglal. A kódot futás közben módosíthatjuk, vagy felülírhatjuk, ez segít a viszonylag kis memória jobb kihasználásában.

Utasítások felépítése

Az assembly utasítások 32 bit hosszúságúak és felépítésük eltér a szokásos utasítás szerkezettől. Meghatározhatjuk, hogy az utasítás végrehajtása után melyik jelzőbitek állítódhatnak át, illetve, hogy az utasítás milyen feltételek mellett futhat le. Ezekkel a trükkökkel tömör és hatékony programot írhatunk.

Parallax Propeller 32 bites assembly utasítások felépítése								
Név	Műveleti kód	Z flag frissítés	C flag frissítés	Eredmény frissítés	Forrás #	Végrehajtási feltétel	Cél-regiszter	Forrás-regiszter
Bitek	**31..26**	**25**	**24**	**23**	**22**	**21..18**	**17..9**	**8..0**
Jelölés	**iiiiii**	**z**	**C**	**r**	**i**	**cccc**	**ddddddddd**	**sssssssss**

- Ha a Z flag frissítés bit be van kapcsolva, akkor az utasítás végrehajtásakor a zero flag értéke frissül.
- Ha a C flag frissítés bit be van kapcsolva, akkor az utasítás végrehajtásakor a carry flag értéke frissül.
- Ha az Eredmény frissítés bit be van kapcsolva, akkor az utasítás végrehajtásakor a célként megjelölt helyre kerül az eredmény. Ellenkező esetben az eredményt nem tárolja a COG.
- Csak akkor hajtja végre azt utasítást a COG, ha a végrehajtási feltételben meghatározott feltétel teljesül. Alapértelmezésben az utasítás mindenféleképp végrehajtódik (IF_ALWAYS).

Utasítás végrehajtás hatása a flag-ekre és a célterületre

A következő táblázatban a z,C,r biteket állító jelölések találhatók. Ezeket a jelöléseket az utasítás mögé kell írni – ha többet használunk, akkor vesszővel kell elválasztani.

Jelölés	Hatás
NR	Nem tárolja az eredményt (a célként megjelölt hely nem íródik felül)
WR	Tárolja az eredményt (a célként megjelölt hely felülíródik)
WC	A carry flag írásának engedélyezése
WZ	A zero flag írásának engedélyezése

Utasítás végrehajtási feltételek

Az alábbi táblázatban az utasítások összes lehetséges futási feltételét felsoroltam. Ezek közül sok egymásnak megfelelő van, ezek a programozó kényelmét –a könnyebb áttekinthetőséget- szolgálják. Ezeket a feltételeket az utasítás elé kell írni és csak egyet használhatunk egy utasításhoz.

Feltétel	A feltétel igaz …
IF_ALWAYS	Mindig igaz feltétel (ez az alapértelmezett)
IF_NEVER	Mindig hamis feltétel
IF_E	Ha egyenlő (Z=1) (ua. mint IF_Z)
IF_NE	Ha nem egyenlő (Z=0) (ua. mint IF_NZ)
IF_A	Ha nagyobb (C =0 és Z=0) (ua. mint IF_NC_AND_NZ, IF_NZ_AND_NC)
IF_B	Ha kisebb (C=1) (ua. mint IF_C)
IF_AE	Ha nagyobb, vagy egyenlő (C=0) (ua. mint IF_NC)
IF_BE	Ha kisebb vagy egyenlő (C=1 vagy Z =1) (ua. mint IF_C_OR_Z, IF_Z_OR_C)
IF_C	Ha a carry flag 1 (ua. mint IF_BE)
IF_NC	Ha a carry flag 0 (ua. mint IF_AE)
IF_Z	Ha a zero flag 1 (ua. mint IF_E)
IF_NZ	Ha a zero flag 0 (ua. mint IF_NE)
IF_C_EQ_Z	Ha a carry flag értéke megegyezik a zero flag értékével (ua. mint IF_Z_EQ_C)
IF_C_NE_Z	Ha a carry flag értéke nem egyezik meg a zero flag értékével (ua. mint IF_Z_NE_C)
IF_C_AND_Z	Ha a carry és a zero flag is 1 (ua. mint IF_Z_AND_C)
IF_C_AND_NZ	Ha a carry flag 1 és a zero flag 0 (ua. mint IF_NZ_AND_C)
IF_NC_AND_Z	Ha a carry flag 0 és a zero flag 1 (ua. mint IF_Z_AND_NC)
IF_NC_AND_NZ	Ha a carry és a zero flag is 0 (ua. mint IF_NC_AND_NC)
IF_C_OR_Z	Ha vagy a carry flag vagy a zero flag 1 (ua. mint IF_Z_OR_C)
IF_C_OR_NZ	Ha vagy a carry flag 1 vagy a zero flag 0 (ua. mint IF_NZ_OR_C)
IF_NC_OR_Z	Ha vagy a carry flag 0 vagy a zero flag 1 (ua. mint IF_Z_OR_NC)
IF_NC_OR_NZ	Ha vagy a carry flag vagy a zero flag 0 (ua. mint IF_NZ OR NC)
IF_Z_EQ_C	Ha a zero flag értéke megegyezik a carry flag értékével (ua. mint IF_C_EQ_Z)
IF_Z_NE_C	Ha a zero flag értéke nem egyezik meg a carry flag értékével (ua. mint IF_C_NE_Z)
IF_Z_AND_C	Ha a zero és a carry flag is 1 (ua. mint IF_C_AND_Z)
IF_Z_AND_NC	Ha a zero flag 1 és a carry flag 0 (ua. mint IF_NC_AND_Z)
IF_NZ_AND_C	Ha a zero flag 0 és a carry flag 1 (ua. mint IF_C_AND_NZ)
IF_NZ_AND_NC	Ha a zero és a carry flag is 0 (ua. mint IF_NC_AND_NZ)
IF_Z_OR_C	Ha vagy a zero flag vagy a carry flag 1 (ua. mint IF_C_OR_Z)
IF_Z_OR_NC	Ha vagy a zero flag 1 vagy a carry flag 0 (ua. mint IF_NC_OR_C)
IF_NZ_OR_C	Ha vagy a zero flag 0 vagy a carry flag 1 (ua. mint IF_C_OR_NZ)
IF_NZ_OR_NC	Ha vagy a zero flag vagy a carry flag 0 (ua. mint IF_NC_OR_NZ)

Egy egyszerű példán keresztül szeretném bemutatni ezt a szokatlan szintaktikát:

```
                mov    t1,     pins    wz
if_nz   mov     dira, t1
```

Az első sor betölti pins tartálmát t1-be, és ha az érték 0 volt, akkor 1-re állítja a zero flaget. A wz jelzi, hogy az utasítás írhatja a zero flaget. A következő sor t1 tartalmát tölti dira-ba, de csak akkor, ha a zero flag 0 – tehát az előzőleg végrehajtott utasításban nem 0-t töltöttünk t1-be.

Címzés

A COG RAM (512 long) címzéséhez 9 bit szükséges. Mivel az utasításban a forrás és a cél számára is egy 9 bites mezőt hagytak, ezért a COG RAM forrás és cél területe is közvetlenül címezhető egy utasításon belül.
Más a helyzet akkor, ha a közös memóriához szeretnénk fordulni.
Elvileg a 64 kbyte byte-os címzéséhez 16, a word-ös címzéséhez 15 és a long-os címzéséhez 14 bit elegendő lenne, de a Propellerben azonban ezt másképp oldották meg.
Ha a közös memóriához szeretnénk fordulni, akkor egy 32 bites regiszterben kell megadnunk az elérni kívánt címet (amiből 16 bitet használ). Pontosabban az adatunk kezdő byte-jának a címét. Ezzel a megoldással bármely címtől kezdődően írhatunk/olvashatunk byte/word/long hosszúságú adatokat.

!

FONTOS!

A megfelelő címszámításról saját magunknak kell gondoskodni. Tehát ha például egymás után következő adatokkal végzünk műveletet, akkor a címszámlálóként használt regiszterünket az adathossznak megfelelően kell növelni:

- Byte hosszúságú adatnál **1**-el
- Word hosszúságú adatnál **2**-vel
- Long hosszúságú adatnál **4**-el

Ha nem megfelelően számítjuk ki a címeket, akkor az adatokat elcsúszva egymásra írhatjuk, és/vagy hibásan olvashatjuk ki. Ez végzetes következményekkel járhat mind a program, mind a vezérelt perifériák számára is!