MIKROPROCESSZOR PROGRAMOZAS

Tartalom

Assembly programozás: Alapok

Bevezetés az assembly nyelv alapjaiba

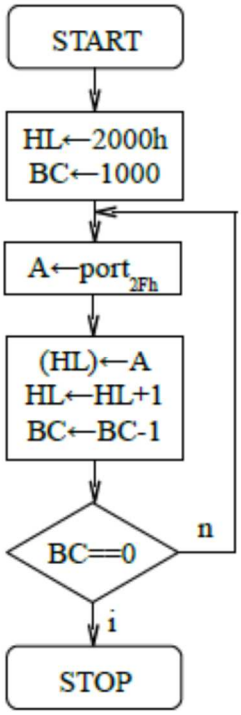
Assembly nyelvű program írása

Alapfogalmak
Assembly nyelv: gépközeli programozási nyelv, egy utasítása megfelel egy gépi utasításnak. Utasításait szimbolikus nevekkel illetjük, ami az adott utasítás angol nevének rövidítése (mnemonik). Utasításokon kívül tartalmaz még direktívákat (lásd a programok felépítésénél).
Assembler: fordítóprogram, ami assembly nyelvről tárgykódra (object code) fordít.
Linker (linkage editor): kapcsolatszerkesztő: az a program, ami a különböző magas szintű nyelvek
(pl. Pascal, C, C++) és az assembly gépközeli nyelv programjainak fordítási egységeiből készült tárgykódú modulokat egy programmá szerkeszti össze.
Megjegyzések:
1. Egyes mikroszámítógépes assemblerek közvetlenül gépi kódú programot készítenek és azt a gép memóriájában helyezik el.
2. Cross compilernek hívjuk az olyan fordítóprogramot, amely az őt végrehajtó gép architektúrájától eltérő architektúrájú gépre fordít. (Ilyen fordítóprogramot használunk például, ha PC-n valamilyen mikrokontrollerre fejlesztünk programot.)
Assembly nyelvű programok felépítése
8080/8085 assembly esetén a program egy sora megfelelhet egy gépi utasításnak, állhat pusztán megjegyzésből, illetve tartalmazhat direktívát. Az első esetben a sor a következő részeket tartalmazhatja:
<címke mező> <műveleti mező> <operandus mező> <megjegyzés rész>
A **címke** betűvel kezdődik, betűket és számokat tartalmazhat, utána kettőspont áll.
A **műveleti mező**ben valamely gépi utasítás mnemonikja áll.
Az **operandus mező** tartalmát a gépi utasítás fajtája határozza meg, üres is lehet.
A **megjegyzés rész** opcionális, ha van, akkor pontosvesszővel kezdődik.
Ha egy sor pontosvesszővel kezdődik, akkor a fordító az egész sort megjegyzésnek tekinti.
A lehetséges direktívákat nézzük meg a Referenciakártyán!
Egy assembly nyelvű programnál mindig meg kell adnunk az ORG direktívával, hogy a gépi kódú program milyen memóriacímtől kezdődjön.
Változóink számára a megfelelő direktívákkal bájt vagy szó méretű helyeket tudunk lefoglalni... Lehetőség van szimbolikus konstansok használatára is.
Assembly nyelvű programok írása
Nem teljesen triviális feladatok esetén a feladatot részekre bontjuk és megtervezzük a részek együttműködését (például: melyik szubrutin milyen funkcióért felelős, ezekből hogyan épül fel a teljes program; paraméterek átadásának módja, konkrétan mit és hogyan adunk át, stb.).
Egyszerű feladat (vagy összetett feladat kellően kicsi, így már egyszerű része) esetén célszerűen folyamatábrát készítünk, amelybe lehetőleg olyan tevékenységeket és feltételvizsgálatokat írunk, amit 1-1 gépi utasítással vagy legfeljebb néhánnyal (erre látunk példát) meg tudunk oldani.
Összetett feladatokra szubrutinhívást alkalmazunk.

(Ezt betenni gyakorlási példának 8085'-re ??????????????)

A 8085 regiszter- és utasításkészletének ismeretében kis gyakorlattal eldönthetjük, hogy mely változókat tároljunk memóriában, illetve regiszterben/regiszterpárban, és azok közül is melyikben.
Programok írásakor mindig használjuk a Referenciakártyát!

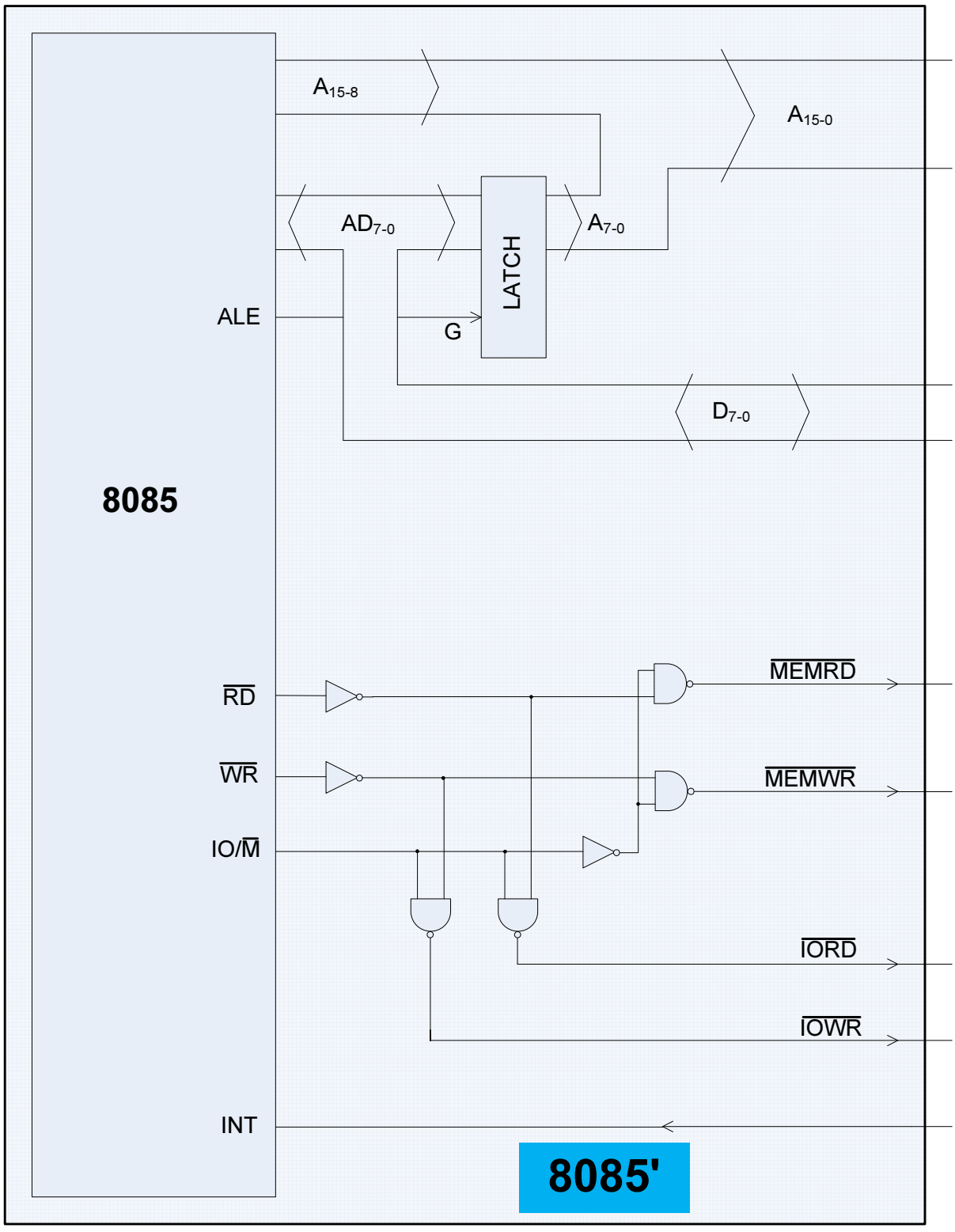
Mintafeladat
Olvassunk be 1000 bájt adatot a 2Fh I/O portról, és tároljuk el a 2000h címtől a memóriában!
A feladat *egy* lehetséges megoldása:




```
        ORG 0000H
;
; kezdőértékek beállítása
        LXI H,2000h
        LXI B,1000
;
; beolvasás a perifériáról
input:  IN 2Fh
;
; tárolás, mutató++, számláló--
        MOV M,A
        INX H
        DCX B
;
; BC regiszterpár értéke 0-e?
        MOV A,B
        ORA C
        JNZ input
;
        END
```

További feladatok
1. feladat: *Másoljuk át a memória 1000h címtől kezdődő 1kB méretű részét a 2000h címtől kezdően!*
2. feladat: *Mint az első feladat, de a másolandó blokk mérete legyen 5kB! Mit veszünk észre?*
3. feladat: *Specifikáljunk és írjunk szubrutint, amely 2 tetszőleges hosszúságú számot összead! Egy lehetséges specifikáció: A számokat binárisan ábrázoljuk (kettes komplemens kódban) az alvég bájtsorrendet követve (legkisebb helyiértékű bájt van elöl), hosszuk a BC, az első operandus címe a HL, a másodiké a DE regiszterpárban található. Az eredmény az első operandus helyén képződjön.*
Bemenő átvitel (carry) nincs, a kimenő átvitel értéke a legyen a C (carry) jelzőbitben!
Adjunk olyan specifikációt is, ahol az operandusok értéke NEM változik meg (az eredmény máshol képződik)!

A 8085’ hipotetikus mikroprocesszor blokkvázlata
HYPO85Z



Assembly programzás: Makro szubrutin

– HLT (**Halt**)...megállító utasítás. Ha rákerül a vezérlés a számítógép futása megáll.

– NOP (**No Operation**)... nincs művelet. Ha erre az utasításra kerül a vezérlés, akkor nem történik semmi. Így – a program fejlesztése során – tetszőleges utasítás NOP-al történő cseréje estén tetszőleges utasítások programjának funkciója kihagyható, illetve visszaállítható.

MK műveleti kód	KMK (Kiegészítő MK)	m (Lépésszám)

1.20. ábra Nem memóriára hivatkozó utasításformátum

Megjegyzések: Az aritmetikai /logikai, a mozgató és vezérlésátadó utasítások a műveleti kód mellett egy címet – pontosabban címkonstanst – tartalmaznak. Az ilyen utasításokat hívjuk **memória-referenseknek**. A léptető/forgató, a HLT és NOP utasítások nem tartalmaznak memóriacímet, így kódolásuk – az utasítás formátuma – kissé más mint az az 1.18. ábrán látszott. A címkonstans helyére a **kiegészítő műveletkód** és a *lépésszám (m)* került (1.20. ábra).

1.2.3. AZ ASSEMBLY[21] PROGRAM

Az eddig ismertetett utasítások felhasználásával: „Töltsük fel egy számítógép memóriáját 1000H címtől 100 rekesz hosszan zérusokkal!"

a) A feladat

Az „ADAT" – szimbolikus nevet viselő rekesz – tartalmát kell elhelyezni a „CÍM" pointer által mutatott rekeszbe. (1.21. ábra)

Az „ADAT" értéke végig zérus. A „CÍM" kezdetben 1000H-ra a végén 1000H + 100=1063H-ra mutat. A „DARAB" rekesz mindig arról tájékoztat, hogy hányszor kell még a feladatot elvégezni, vagyis kezdetben 100, a végén 0 a tartalma. (Ez a „zérus" lesz a leállítás feltétele.)

A program során elvégzendő operációk nagyon egyszerűek:

– kezdeti beállítások

– tárolás a pointerben meghatározott címre, vagyis **indirekt címzés.**

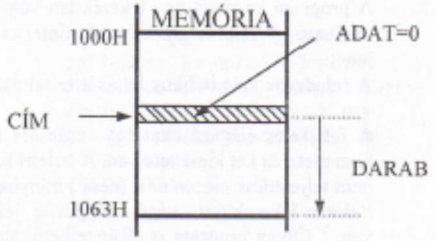
– a pointer növelése

– számláló rekesz csökkentése

és

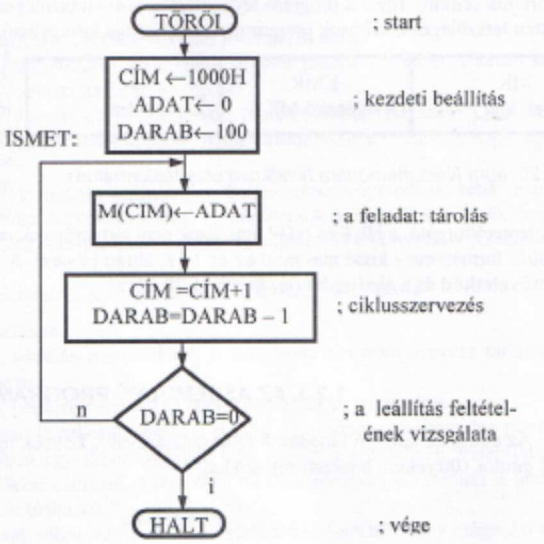
– zérus értékének komparálása

– a számítógép leállítása.



1.21. ábra 100 memória rekesz törlése 1000H-tól

[21] Az assembly olyan programozási nyelv, mely szorosan kapcsolódik a hardverhez. Minden utasítása közvetlenül gépi utasítássá fordítható.



1.22. ábra 100 rekesz törlésének folyamatábrája

b) A folyamatábra

Az 1.22. ábrán látható folyamatábra négy féle szimbólumot használ:

- A program kezdetét egy lekerekített végű négyszög jelzi, amelyben a program neve található. A szimbólumnak csak kimenete van és minden programban csak egy ilyen lehet.
- A feladatok szimbolikus leírását tartalmazó négyszögeknek egy be és egy kimenetük van.
- A feltételes elágazásokat egy csúcsára állított rombusz szimbolizálja, melynek egy bemenete és két kimenete van. A beleírt feltétel teljesülése esetén az i (igen) irányban, nem teljesülése esetén az n (nem) irányban folytatódik a program.
- Szintén lekerekített végű négyszög jelzi a program végét. Ennek csak bemenete van. (Olyan program is elképzelhető, amely – ellentétben az itt bemutatottal – több helyen fejeződik be.)

A haladási – futási – irányokat nyilak jelzik. Találkozási pontoknak szimbolikus nevek – **címkék** – adhatók. (Ilyen pont a példaprogramban az ISMET:) A címkék kettősponttal végződnek, betűvel kezdődnek és nem tartalmaznak ékezetes betűt, valamint írásjelet. (Egy – egy konkrét programozási nyelvben a címke maximális hossza korlátozva van.)

c) A program

Az 1.23. ábrán látható assembly program sorokból áll. Egy-egy sor felépítése a következő:

	címke mező	műveleti mező	operandus mező	megjegyzésrész
(Az angol elnevezések:	label	operation	operandus	comments.)

Minden sorban a „ ; ” szeparátor[22] utáni szöveg megjegyzésnek tekintendő, ha a sor „ ; ”-vel kezdődik , akkor a teljes sor megjegyzés.

Egy programsor tartalmazhat **utasítást**, melyet a számítógép végrehajt, vagy **direktívát**, mely a fordítóprogram számára jelent információt. Az 1. 22. folyamatábra sorai jól követhetők az 1. 23. ábra forráslistájában.

```
CIMKE     MÜVELET   OPERANDUS      MEGJEGYZÉS

          ORG       50H            ;Kezdődjön az 50H címtől
;törlő program
TOROL:    LOAD      H1000
          STORE     CIM            ;CIM ← 1000H
          LOAD      NULLA
          STORE     ADAT           ;ADAT ← 0
          LOAD      C100
          STORE     DARAB          ;DARAB ← 100
ISMET:    LOAD      ADAT
          STORE  I  CIM            ;MEM(CIM) ← ADAT
          LOAD      CIM
          ADD       C1
          STORE     CIM            ;CIM=CIM+1
          LOAD      DARAB
          SUB       C1
          STORE     DARAB          ;DARAB=DARAB-1
          JUMP      NZ,ISMET       ;DARAB=0?
          HALT                     ;igen
;konstansok és változók
CIM:      DC        0              ;a cím pointer helye
DARAB:    DC        0              ;a hossz számláló helye
ADAT:     DC        0              ;a beírandó információ helye
H1000:    DC        1000H          ;konstans: 1000 hexa
C100:     DC        100            ;konstans: 100 decimális
C1:       DC        1              ;konstans: 1
NULLA:    DC        0              ;konstans: 0
;vége a programnak
          END
```

1.23. ábra Forráslista

A mintaprogramok direktívái:

(i) ORG 50H (Origin)

A direktíva megadja, hogy a programot a memória 50H rekeszétől kezdődően kell folyamatosan elhelyezni. (Egy programban több ORG is elképzelhető.)

[22] Szétválasztójel: itt pontosvessző

(ii) CIMKE: DC 1000H (DC=**D**efine **C**onstant)
Egy memóriarekeszbe egy 1000Hexa konstanst helyez el, amely a továbbiakban a CIMKE szimbolikus névvel hívható. A konstansok lehetnek decimálisak (DC 100), oktálisak (DC 117Q), hexadecimálisak (DC 1FBH), binárisak (DC 101101B) vagy szövegek (DC „START"). Az utóbbi esetben 5 byte hosszú ASCII kódú[23] szöveget helyez el a memóriában a fordítóprogram.

(iii) END
A forrásprogram vége, eddig nézi át a fordítóprogram a szövegfile-t.

Megjegyzés: a konstansok és a változók helye tetszőleges a memóriában. Egy hosszabb feladatnál azonban nem ismerhetjük előre az összes szükséges rekeszt, ezért célszerű a program végén elhelyezni őket.

Most már csak az a kérdés:
„Hogyan érti meg és hajtja végre a programot a számítógép?"

Ezt a folyamatot az 1. 24. ábra – **az operációs rendszer** – mutatja.

1.2.4. FORDÍTÁS, BETÖLTÉS, FUTTATÁS

A felhasználó által megírt PASCAL, C, CLIPPER, FOXPRO... vagy (ahogy az 1.23. ábrán is látható) ASSEMBLY nyelvű programokat **forrásprogram**oknak hívjuk. A forrásprogramok szimbolikus címeket (címkéket), az utasítások és direktívák emlékeztető (mnemonikus) kódjait, műveleti jeleket és megjegyzéseket (commenteket) tartalmaznak. A forrásnyelvű programokat valamilyen – erre alkalmas **szövegszerkesztő** programmal (texteditorral) egy elsődleges tároló eszközre – hajlékony mágneslemez (FD), mágnesszalag (MT), esetleg egyenesen a diszkre (HD) visszük fel. Ezekről az eszközökről kerülhet fel a program a **forráskönyvtár**ba (source) szintén a forráskönyvtárban találhatók a MAKRO-k. (Ezekről majd a későbbiekben lesz szó.) A számítógép könyvtárai mindig a háttértárolón elhelyezett azonos funkciójú programok csoportját jelentik.

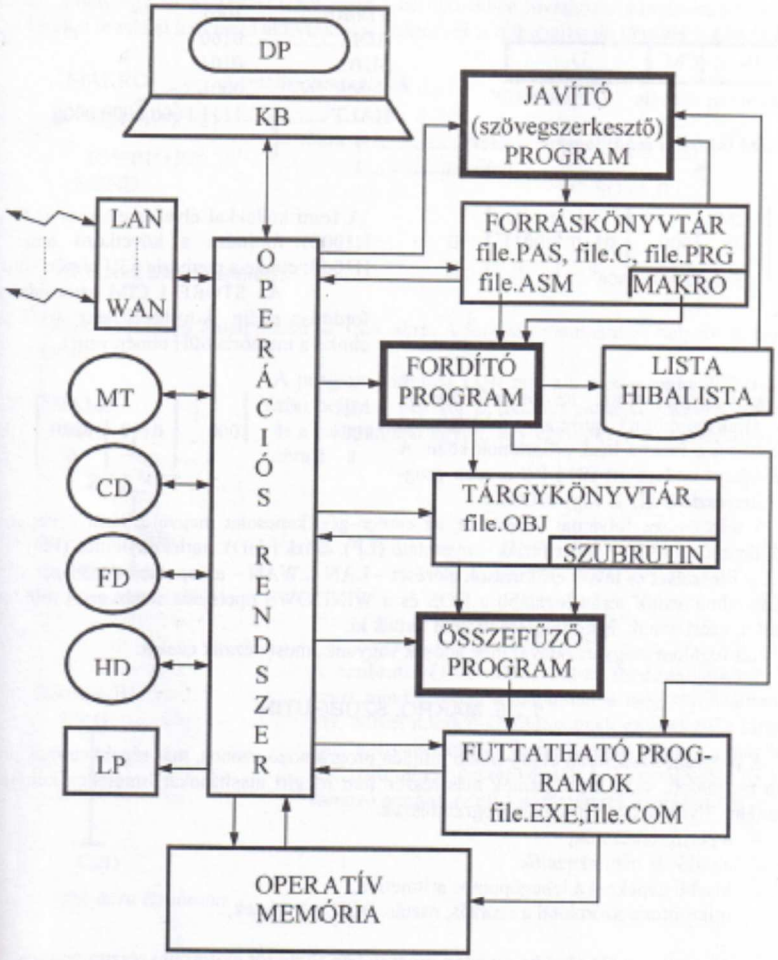
A forrásnyelvű (a felhasználó számára érthető formában leírt) programot gépi kóddá (a gép számára érthető formátummá) a **fordítóprogram** alakítja. Az így keletkező és már csak bináris számokat tartalmazó formátumot **tárgyprogram**nak (objectnek) hívjuk. A tárgyprogramok a tárgykönyvtárban találhatók. Minden számítógépnek annyi fordítóprogramja van, ahány nyelven programozható. Egy IBM PC-nek például van PASCAL fordítója, ez a .PAS kiterjesztésű file-okat fordítja le, vagy van C fordítója, ez a .C kiterjesztésű file-okat., a CLIPPER vagy a FOXPRO fordító pedig a .PRG kiterjesztésű forrásnyelvű programokból csinál .OBJ kiterjesztésű tárgyprogramot. Egy adott számítógépnek csak egyféle tárgyprogramja (tárgykódja) van! (Az assembly szokásos kiterjesztése .ASM, fordító programját pedig ASSEMBLER-nek hívjuk.)

Minden fordító, azonos feladat megoldására azonos kódot generál, természetesen bizonyos redundanciával. Minél magasabb szintű egy nyelv, annál redundánsabb lesz az általa generált tárgyprogram. (Például azonos feladat megoldására megírt C program 2 – 4 -szeres, PASCAL program 3…10 -szeres redundanciát tartalmaz az ASSEMBLY programhoz képest. Azonban egy feladat megoldása ASSEMBLY nyelven legalább ennyiszer több időt igényel.

A tárgyprogramokat a **tárgykönyvtár**ban tároljuk. Itt találhatók meg a **szubrutin**ok is, melyekről később lesz szó.

[23] ASCII – American Standard Code for Information Interchange

A fordítás során keletkezik a tárgykódon kívül a forrásprogram listája és hibalistája is. A hibalista a **szintaktikus** hibákat, – például rossz mnemonik, túl nagy konstans, nem definiált címkére hivatkozás -- tartalmazza.

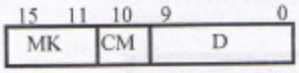


1.24. ábra Az operációs rendszer kapcsolatrendszere

A hibalista alapján a forrásnyelvű **javító** (editor) program segítségével javíthatjuk ki, és vihetjük ismét fel a forráskönyvtárba a javított programokat. Az újabb fordítás újabb tárgykódot, listát, hibalistát generál, s ezt a folyamatot addig ismételjük, amíg a programból el nem fogynak a hibák.

Példaként nézzük meg, hogyan fordítódik le az 1.23. ábrán látható program néhány sora!
Az utasításábrázolási forma és a mnemonikok kódolása:

A MK (műveleti kódok) legyenek:
LOAD.....1000
STORE.....1001
ADD.....0100
SUB.....0101
JUMP NZ.....0001
HALT 1111 0000 0000 0000



CM (címzési mód) legyen: direkt.....0
indirekt.....1

MK	CM		D		
1000	0	000	0110	0011	B
8		0	6	3	H

A fenti kódokkal élve az első sor (LOAD H1000) fordítása a következő lesz: (a H1000: címke a memória 63H címén van!)

A STORE I CIM assembly sor fordítása pedig következő lesz: (A CIM: címke a memória 60H címén van!)

MK	CM		D		
1001	1	000	0110	0000	B
8		8	6	0	H

A tárgyprogramok az **összefűző-szerkesztő** (linkage-editor) program feldolgozása után kerülnek a futásra kész programok közé. A már példaként emlegetett IBM PC-n ezen programok kiterjesztése .EXE vagy .COM.

A leírt összes folyamat vezérlését, az ember–gép kapcsolat megvalósítását – képernyő (DP), billentyűzet (KB) – a perifériák – nyomtató (LP), diszk (HD), hajlékonylemez (FD), CD-ROM ... – illesztését és távoli erőforrások elérését – LAN ...WAN – az operációs rendszer végzi. Eddigi tanulmányaink során legalább a DOS és a WINDOWS operációs rendszerrel mindenki találkozott, ezért annak ismertetésére itt nem térünk ki.

Két fogalom magyarázatával még adósak vagyunk, most nézzük ezeket:

1.2.5. MAKRO, SZUBRUTIN

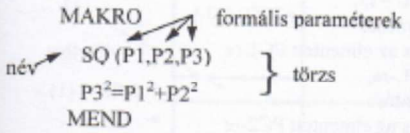
A program írása során előbb-utóbb minden programozó azonos, már régebben más programban megoldott, vagy a programok más részén már megírt utasításokat ismételten kénytelen felhasználni. Ilyen sokszor használt programrészek:

- a perifériakezelők,
- az idő- és dátumkezelők,
- kisebb gépeknél a lebegőpontos aritmetika,
- mikroprocesszoroknál a szorzás, osztás, BCD műveletek, ...
- ...
- és a felhasználó által írt, az adott programban többször előforduló részprogramok.

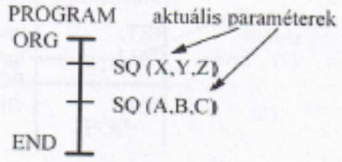
Azért, hogy ilyen esetekben, ne kelljen mindig minden utasítást újra leírni, valamint, hogy a mások által elkészített programrészek bárki számára könnyen felhasználhatók legyenek, létezik két szabványos programrészírási eljárás: a **makro** és a **szubrutin**.

a) Makró

A makró *forrásnyelvi* programrész, ezért mindig a forráskönyvtárban van katalogizálva. Az 1.25. ábra mutatja a **makródefiniálás** szabályát. A MAKRO-MEND direktívák közé kerül a makrótörzs, amely egy nevet, (ezzel lehet majd a későbbiekben hivatkozni a makróra,) formális paramétereket (ezekkel kell majd aktivizálni a makrót) és a végrehajtandó utasításokat tartalmazza:

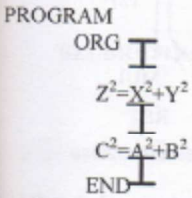


1.25. ábra Makródefiniálás



1.26. ábra Makróhívás

Makróhívására mutat példát az 1.26. ábra. A formális paraméterek helyére az aktuálisakat kell írni.



1.27. ábra Makro kifejtés

A program fordítása **előtt** történik meg a **makrókifejtés.** A hívási helyekre bekerül az aktuális paraméterekkel a makrótörzs, és a programmal együtt, egy egységként lesz lefordítva. (1.27. ábra.)

b) Szubrutin

1.28. ábra Szubrutin

A szubrutin (1.28. ábra) **külön fordítási modult** képez, ezért mindig tárgyformátumban a tárgykönyvtárban tároljuk. Mivel minden számítógépnek csak egyféle tárgykódja van, ezért tetszőleges forrásnyelven megírt és már lefordított szubrutinok összefűzhetők és használhatók, ha a programozó gondoskodott a paraméterek átadásáról.

Az 1.29. ábra a szubrutin hívására mutat példát.

A program és a szubrutin összefűzését az összefűző-szerkesztő program végzi. Két feladatot kell elvégeznie:

- bevinni a memóriába a programokat,
- a szubrutinok hívása helyére beírni a szubrutin betöltési címét. (1.30. ábra)

A szubrutin hívása CALL vagy BAL (Branch and Link) utasítással történik. A branch a szubrutinra való ugrásra, a **link** – csatolás – a visszatérési cím elmentésére utal. A hívás helyére mindig az utasításszámláló (PC) mutat, ezért mindig a PC-t kell elmenteni.

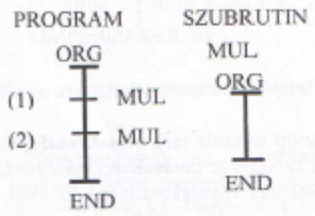
Amint az az 1.30. ábrából is látható máshova kell visszaadni a vezérlést a szubrutin végén az (1) hívásnál, és máshová kerül vissza a vezérlés a (2) hívás kiszolgálása után. A szubrutinból való **visszatérés** utasítása a RET (Return). Nézzük részletesebben:

(1) hívás: CALL: - ugrás MUL – ra,
- PC1 elmentése.

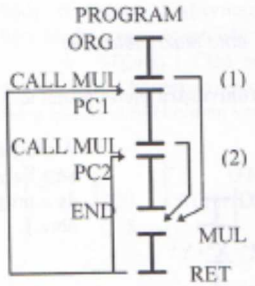
RET: - visszatérés az elmentett PC1-re

(2) hívás: CALL - ugrás MUL-ra,
- PC2 elmentése

RET: - visszatérés az elmentett PC2-re



1.29. ábra Szubrutinhívás



1.30. ábra Szubrutin beszerkesztése

Kisebb számítógépekben egyetlenegy Link (csatoló) regiszter található és ez minden CALL utasítás hatására feltöltődik a hívó utasításra mutató, (vagy az azt követő) PC értékével, majd minden RET utasítás hatására a Link regiszter tartalma íródik vissza a PC-be. (Ugrás a link alapján.) Ennél az elrendezésnél problémát okoz, ha szubrutinban újabb szubrutint hívunk, mert ebben az esetben a második CALL utasítás felülírja a Link regisztert és elveszik az első visszatérési cím. A problémán a Link regiszter tartalmának az újabb szubrutin hívása előtti **elmentésével** és a szubrutinból való visszatérés utáni **visszaállításával** lehet segíteni.

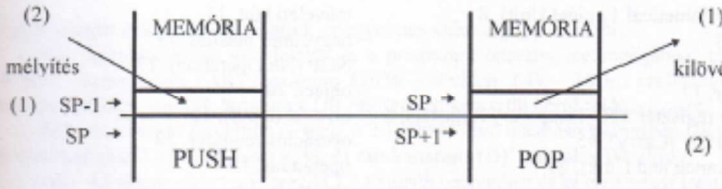
Tetszőleges mélységű egymásba skatulyázott szubrutinok visszatérési címeinek elmentését és visszaállítását automatizálja a **stack**memória. Korszerű számítógépekben, a stack az operatív memória egy kijelölt része. A központi feldolgozó egységben (CPU) csak egy regiszter, az úgynevezett **stack pointer** (SP) található. Nézzük a stackműveleteket: (1.31. ábra)

PUSH: (beírás, mélyítés)
- (1) a stackpointer értéke eggyel csökken (SP – 1)
- (2) az új SP által mutatott memóriarekeszbe beírjuk a kívánt adatot

POP: (kilökés)
- (1) az SP által mutatott memóriarekesz tartalmát kiolvassuk
- (2) az SP értékét 1-gyel megnöveljük (SP+1)

Az eddigiek alapján ha a CALL utasítás végrehajtása során az aktuális PC értéket egy PUSH PC művelettel a stackbe mentjük és a RET utasítás végrehajtása során mindig egy POP PC utasítással vesszük elő a visszatérési címet, akkor a szubrutinok egymásba skatulyázhatóságának csak a stack mérete szab határt.

A stack nemcsak szubrutin hívásnál és visszatérésnél, hanem:
- szubrutin paraméterátadásnál,
- rekurzív szubrutinhívásnál és
- tetszőleges helyen megszakítható és újraindítható szubrutinok (időosztásos rendszerek) estén is jól felhasználható.



1.31. ábra Stackműveletek

Végezetül foglaljuk össze a makró és a szubrutin néhány tulajdonságát:

MAKRO:
1. Forráskönyvtárban tároljuk
2. Többszörös hívás esetén sok helyet igényel, mert minden hívásnál be kell építeni.
3. Gyors, mert futás során nincs paraméterátadás.

SZUBRUTIN:
1. Tárgykódban tároljuk. (Kevesebb hely; tetszőleges forrásnyelv)
2. Külön fordítási egység: külön javítható, élesztheto.
3. Kicsi a memóriaigénye, csak egyszer van a memóriában elhelyezve.
4. A futása a paraméterek átadása miatt lassú.

Mindig az adott feladat specifikációja alapján kell eldöntenünk, hogy makrót, vagy szubrutint használunk. Ha egy feladat során csak egyszer használjuk fel a makrót, vagy szubrutint, akkor is érdemes megírni külön, hátha **holnap is, másnak is** kell!

1.3. ELLENŐRZŐ KÉRDÉSEK

1. Hasonlítsa össze a regiszterek huzalozott és sínes kapcsolatát!
2. Milyen adatokkal jellemezhető a számítógép memóriája?
3. Milyen különbségek vannak egy magas szintű és az assembly program között?
4. Milyen mezőket tartalmaz egy 4, 3, 2, 1 címes utasítás?
5. Milyen célt szolgál a PC, és milyet az ACC regiszter?
6. Mi a különbség direkt és indirekt címzés között?
7. Mi az utasítás és mi a direktíva? Soroljon fel mindkettőből néhányat!
8. Mi a különbség assembly és assembler között?
9. Hasonlítsa össze a forrás- és a tárgykönyvtárat!
10. Mi a közös és mi a különböző a szubrutinban és a makróban?
11. (•)[24] Számítsa ki – az 1.22. ábra alapján -, hogy hány százalékára csökken a forrásprogram a fordítás során! (A forrásprogram minden karakterének tárolására egy bytenyi memóriára van szükség.)

[24] A *-gal megjelölt feladatok nehezebbek, ezért utoljára célszerű hozzákezdeni megoldásukhoz.

Mikrovezérlők programozási nyelvei

Forrás: ----------## JEGYZET NAGYON JÓÓ§§ --MyStudy-- PIC Harvard MÓDOSÍTOTT HARVARD Mikroprocesszorok alkalmazaasa 2012 .docx

Az assembly programozás alapgondolata

A processzortól általunk elvárt feladatokat programokba foglaljuk, legyen a processzor egy számítógép processzora, vagy egy mikrovezérlő. Az első számítógépek programozása a számításokat végző elemek huzalozásával történt, programváltáskor a huzalozást változtatták. Ezután már a processzorok utasításaihoz rendelt bináris számok bevitelével (gépi kóddal) lehetett programozni. Ezeket a számokat eleinte bináris majd később már hexadecimális számrendszerben ábrázolták, illetve ábrázolják.

A programok megírásánál többféle programnyelvet használhatunk, ezek a programnyelvek lehetnek az emberi kommunikációhoz hasonló szövegesebb, vagy inkább a gépi kódú logikához hasonlóak. Az előzőt magas, míg a gépi kódhoz hasonló programnyelveket az alacsony szintű programnyelvekhez soroljuk.

A különböző szintű programozási nyelvek szolgáltatásait tekintve látható, hogy az alacsony szinttől a magasabb szint felé haladva a programok egyre közérthetőbbek, jobban strukturáltabbak, a programok megírása során sokkal könnyebben hajthatunk végre komplex feladatokat. Ez a magas szintű programnyelvek általános programozói feladatokhoz kifejlesztett utasításainak, függvényeinek köszönhető. Felmerül a kérdés, hogy akkor miért is használjuk mégis az assembly programnyelvet. Több indok is létezik, az első, hogy a magas szintű programnyelveket is meg kell írni valamiben. Egy másik indok az általánosabb feladatokat ellátó programok esetében pedig gyakran a hatékonyság növelése. A magas szintű programnyelveken megírt programok általános felhasználásra lettek megírva, gyakran a mi feladatunk szempontjából hasztalan funkciókkal. Ezek vonatkozhatnak általánosított hardver elemekre, szoftverkörnyezetre stb. Ha mi magunk egy adott környezetre írunk egy programot, számos vizsgálatot kihagyhatunk, aminek hatására a programunk gyorsabb lesz és kevesebb helyet fog foglalni a memóriában. Példának tekintsünk egy egyszerű programrészletet, amikor a programunk rajzol egy pontot a képernyő közepére, ahhoz, hogy tudja, mik a képernyő közepének koordinátái, először beolvassa annak a felbontását. Ez a programrészlet sokkal rövidebb, ha a felbontásunk fix, egyféle képernyőt használunk, és mi magunk beírjuk a koordinátákat. Cserébe viszont a programot nem lehet rugalmasan használni más hardver-, szoftverkörnyezetben. Mikrovezérlők programozásánál nagy általánosságban ez nem is szokott cél lenni, a mikrovezérlő programját leggyakrabban adott hardverre írjuk.

Az assembly programozási nyelv egy általános célú, alacsony szintű, gépközeli programozási nyelv. Az assembly nyelv nagyfokú hasonlatossága ellenére sem keverendő össze a gépi kóddal, az assembly egy programozási nyelv, míg a gépi kód egy tárgykód. Az assembly nyelven írt program a processzor utasításkészletéből választott végrehajtható utasításaiból áll, a sorok általában egy gépi kódú utasításnak felelnek meg. A gépi kód viszont az a bináris gépi szavakból álló tárgykód, ami a processzorok utasításmemóriájába letölthető. A gépi kódot a programozási nyelvek fordító programjai végeredményként kapjuk.

Az assembly programnyelv főbb jellemzői:

- nagyon egyszerű, elemi műveletek
- típustalanság
- rögzített utasításkészlet
- világos, egyszerű szintaxis
- kevés vezérlési szerkezet
- nagyon kevés adattípus; ha több is van, akkor általában egymásból származtathatók

Az assembly nyelvben az adott processzor utasításai használhatóak, emiatt a nyelv utasításai processzorról processzorra változnak, de a gyártók által megadott assembly nyelvű szintaxis általában hasonló irányelvekre épül. A nyelvben általában nincsenek programkonstrukciók, típusok, osztályok stb., viszont lehetnek benne makrók, fordító direktívák stb., amik a programírást megkönnyítik.

Az assembly program utasításokból, direktívákból és pszeudoutasításokból állnak. Az utasítások a processzor utasításkészletének elemei. Az utasítást a rá jellemző cselekvés néhány betűs rövidítéseivel, az úgynevezett mnemonikokkal helyettesítjük a forráskódban. A direktívákkal a program memóriában történő elhelyezése, felépítése, belépési pontjának meghatározása, a változók deklarálása stb. vezérelhető. A pszeudoutasítások a fordítás/kódgenerálás vezérlése használt ál-utasítások, amelyek a lefordított programlistában nem jelennek meg, csak a lista generálását befolyásolják.

A C nyelvű programozása alapgondolatai

A C programozási nyelvet az AT&T keretein belül 1969 és 1973 között fejlesztették ki. A legnagyobb fejlesztési eredmények Dennis Ritchie-nek és Ken Thomsonnak – a B nyelv kifejlesztőjének – köszönhetőek. A C nyelvet UNIX operációs rendszerekhez fejlesztették ki, azonban, mivel a C nyelv nem kötődik egyetlen operációs rendszerhez vagy számítógéphez sem, használhatóságából kifolyólag ma már jóformán minden operációs rendszerre megtalálható a megfelelő C fordító. A C programozási nyelv felhasználói és rendszerprogramozáshoz egyaránt használható.

A C nyelvet az 1980-as években az IBM PC számítógépekre is átültették, így a C nyelv népszerűsége ugrásszerűen emelkedni kezdett és kezdte felváltani a BASIC nyelvet. Ebben az időben a Bell Labs munkatársai – Bjarne Stroustrup és társai – elkezdték kibővíteni a C nyelvet objektum-orientált nyelvi elemekkel. Ez a nyelv a C++ nevet kapta, manapság a legelterjedtebb programozási nyelv a Microsoft Windows operációs rendszereken. A C a UNIX és a mikrovezérlők világában viszont megőrizte népszerűségét.

Éveken keresztül A C programozási nyelv első kiadásában szereplő referencia-kézikönyv volt a C nyelv definíciója. 1983-ban az Amerikai Nemzeti Szabványügyi Intézet (ANSI) létrehozott egy bizottságot a C nyelv modern, átfogó definiálására. 1989-re elkészült a C nyelv szabványosítása (egy évvel az első C++ ANSI szabvány után!) és jóváhagyták, mint: ANSI X3.159–1989 „A C programozási nyelv”. A nyelvnek ezt a verzióját nevezik ANSI C-nek. 1990-ben az ANSI C szabványt (néhány apróbb módosítással) átvette a Nemzetközi Szabványügyi Szervezet (angolul: International Organization for Standardization, röviden ISO) mint ISO/EC 9899:1990.

Az ANSI C szabványosítás egyik célja az volt, hogy a Kernighan–Ritchie-féle C-ből és a különböző, nem hivatalos bővítésekből egy egységeset alakítson ki. A szabványosított C fordító az eredeti referencia-kézikönyvön alapszik, azonban számos új megoldás is helyet kapott, mint például függvény prototípus (a C++ nyelvből) valamint egy fejlettebb előfordító (preprocessor). Az ún. szabványos fejlécek (headerek) gyűjteménye lehetővé teszi a függvény- és adattípus-deklarációk egységes kezelését. Ezt a könyvtárat használó programok kompatibilis módon fognak együttműködni a befogadó rendszerrel. Az ANSI C-t szinte minden manapság használt fordító támogatja.

A legtöbb C kód, mely manapság íródott, az ANSI C-n alapul. Vannak azonban programok, melyek csak adott platformon vagy adott fordítóval fordíthatók le, az általuk használt nem szabvány függvénygyűjtemények miatt.

Mivel a C nyelvben alkalmazott adattípusok és vezérlési szerkezetek alkalmazását a legtöbb számítógép közvetlenül támogatja, az önmagában zárt programok formájában megvalósított futtatási könyvtár kicsi. A standard könyvtár függvényeit csak explicit módon hívjuk, így minden további nélkül elhagyhatók, ha nincs szükség rájuk. A függvények többsége C nyelven íródott és – az operációs rendszerhez tartozó részek kivételével – más gépre is átvihető.

A C nyelv sokféle számítógép adottságaihoz illeszkedik, mégis bármilyen konkrét számítógép felépítésétől független, ezért viszonylag kis fáradsággal írhatunk hordozható, azaz változtatás nélkül különféle számítógépeken futtatható, C programokat. A szabvány a hordozhatóságot explicit módon megköveteli, és azon számítógép jellemzésére, amelyen a program futtatható egy paraméterhalmazt ír elő.

A C nyelv nem erősen típusos nyelv, de a fejlődése során a típusellenőrzés erősödött. A C eredeti definíciója, eléggé el nem ítélhető módon, megengedte a mutatók és az egész típusú adatok keverését. Ezt a hiányosságot már régen kiküszöbölték, és a szabvány már megköveteli a megfelelő deklarációt és az explicit típuskonverziót, amit a jó fordítóprogramok ki is kényszerítenek. A C nyelv alap adattípusai a karakterek, valamint a különböző méretű egész és lebegőpontos számok. Ezekhez járul a származtatott adattípusok hierarchiája, amelyekbe a mutatók, tömbök, struktúrák és unionok tartoznak. A kifejezések operátorokból és operandusokból állnak, és bármely kifejezés – beleértve az értékadást vagy a függvényhívást is – lehet önálló utasítás. A mutatókkal végzett műveletekhez a nyelv egy géptől független címaritmetikát használ. A fordítóprogramok a legtöbb típusillesztési hibára figyelmeztetnek és inkompatíbilis adatok között nincs automatikus típuskonverzió. Bárhogyan is nézzük, a C megtartotta az alapfilozófiáját, miszerint a programozónak csak tudnia kell, hogy mit csinál, és a C nyelv csak azt igényli, hogy a szándékát egyértelműen fogalmazza meg.

A C nyelv tartalmazza a strukturált programozáshoz szükséges vezérlési szerkezeteket: az összetartozó utasításokat egyetlen csoportba foglaló utasítás-zárójelet, a döntési szerkezetet (if-else), a lehetséges esetek egyikének kiválasztását (switch), az elöltesztelt ciklust (while, for) és a hátultesztelt ciklust (do), valamint a ciklusból való feltétel nélküli kilépést (break).

A függvények értéke visszatéréskor az alap adattípusok egyike, ill. struktúra, union vagy mutató lehet. Bármely függvény rekurzívan hívható és lokális változói általában „automatikusak", vagyis a függvény minden hívásakor újra generálódnak. A függvénydefiníciók nem ágyazhatók egymásba, de a változók blokkstruktúrában is definiálhatók. Egy C program függvényei önálló forrásállományban is elhelyezhetők és külön is fordíthatók. A függvények változói belső (internal), külső, de csak egyetlen forrásállományban ismert (external) vagy a teljes programban ismert (globális) típusúak lehetnek.

A C nyelvű programok fordításához egy előfeldolgozó menet is kapcsolódik, ami lehetővé teszi a program szövegében a makrohelyettesítést (más forrásállományokat is beleértve), valamint a feltételes fordítást.

A C viszonylag alacsony szintű nyelv. Ezt a kijelentést nem pejoratív értelemben használjuk, hanem egyszerűen csak azt akarjuk kifejezni vele, hogy a C nyelv – a legtöbb számítógéphez hasonlóan – karakterekkel, számokkal és címekkel dolgozik. Ezek az alapobjektumok az adott számítógépen értelmezett aritmetikai és logikai műveletekkel kombinálhatók és mozgathatók.

A C nyelv nem tartalmaz műveleteket az összetett objektumok (karakterláncok, halmazok, listák, tömbök) közvetlen kezelésére, vagyis hiányzanak a teljes tömb vagy karakterlánc manipulálására alkalmas műveletek, bár a struktúrák egy egységenkénti másolása megengedett. A nyelvben csak a statikus és a függvények lokális változóihoz használt verem típusú tárfoglalási lehetőség létezik, és nincs a más nyelvekben megszokott heap vagy garbage collection (a felszabaduló tárterületeket összegyűjtő és hasznosító mechanizmus) típusú dinamikus tárkezelés. Bár ezeknek a lehetőségeknek a hiánya komoly hiányosságnak tűnhet („Két karakterlánc összehasonlításához egy függvény szükséges?"), a nyelv szigorú korlátozása valójában előnyös. Mivel a C viszonylag „kis" nyelv, ezért tömören leírható és gyorsan megtanulható. A programozótól elvárható, hogy ismerje és értse, valamint szabályosan használja a teljes nyelvet.

Végül pedig a C nyelvben nincs adatbeviteli és adatkiviteli lehetőség, azaz nincs READ vagy WRITE utasítás, valamint nincsenek beépített állományelérési módszerek sem. Mindezeket a magasabb szintű tevékenységeket explicit függvényhívásokkal kell megvalósítani. A legtöbb C fordító csomag szerencsére már tartalmazza az ilyen célokra alkalmazható függvények gyűjteményét, az ún. standard könyvtárat.

A fenti előnyök miatt a C nyelv alkalmazása széles körben elterjedt a mikrovezérlők körében. Általánosságban elmondható, hogy ma már minden mikrovezérlőhöz található C fordító, sőt, a nagyobb teljesítményű mikrovezérlőkhöz már C++ fordító is (MPLAB XC32++)!

Forráskód lefordítása gépi kódra

Az assembly, C, vagy más nyelven megírt forráskódból futtatható kódot a forráskód lefordításával (compile) és összeszerkesztésével (link) kapunk. A forráskód elkészítésére szinte minden rendszer biztosít egy szövegszerkesztőt (editor), amely a begépelt szöveget egy állományba menti. A forráskód elkészítése után a következő lépés a forráskód gépi kódra fordítása. A forráskód alapján azt a gépi kódú programot, amely az adott processzor által ismert utasításokat tartalmazza a fordító program (compiler) készíti el. A fordító egy tárgykódú (object) modult készít, amelyben az ugrások, változók abszolút címei helyére még a modul belépési pontjához viszonyított relatív címet helyettesít, mintha a program a 0-ik memóriacímtől kezdődne. Az abszolút címek már csak azért sem szerepelhetnek a tárgykódban, mivel egy program rendszerint több, külön fordított modulból áll(hat) és ezek a különálló modulok is tartalmaznak memóriahivatkozásokat, ugrásokat stb. el kell helyezni őket a programmemóriában (esetleg speciális memóriacímre). A forráskód fordítása után a szerkesztés a következő folyamat. A szerkesztő (linker) feladata a tárgykódú modulok címeinek összehangolása, a kereszthivatkozások feloldása, a futtatható, mikrovezérlőbe letölthető program előállítása.

A programkészítés fent vázolt menetét a korszerű integrált fejlesztőrendszerek észrevehetetlenné teszik.

Az Intel HEX formátum

Mikrovezérlők esetében a megírt és lefordított program eredménye egy, a mikrovezérlőbe letölthető fájl lesz. Ez a fájl gyakorlatilag a mikrovezérlő programmemóriájába írandó bájtokat tartalmazza. Ennek a fájlnak a formátuma

gyártófüggő, nem szabványosított. Egy igen elterjedt formátum a kezdetben az Intel által használt ún. Intel HEX formátum. Ez a formátum az 1970-es évektől fogva használatos az EPROM-ok, mikrovezérlők programozására. A formátumnak három típusa van: a 8, 16 és a 32 bites. Ezek a bájtok sorrendjében térnek el egymástól.

Az adatsorok felépítése

Minden adatsor hat mezőből áll:

Adatmező vagy oszlop	Tartalom	Hossz	Leírás
1	Startkód	1 bájt	egy karakter, egy ASCII kettőspont ":"
2	Bájtok száma	1 bájt	Az adatbájtok száma. Általában 16 (0x10) vagy 32 (0x20) bájtnyi adat.
3	Cím	2bájt	Az adatok kezdőcíme a memóriában. A cím 16 bites, így a maximális címtartomány 64 kB. ezt úgy lehet túllépni, hogy a további bájtokat rekordtípusokon keresztül adjuk meg. A cím "big endian" formátumú, a cím végén van az MSB.
4	Rekord típusa	1 bájt	az adatmező típusát adja meg, értéke 0x00-tól 0x05-ig terjed
5	Adat	a 2. adatmezőben megadott számú bájt	a 2. adatmező-ben megadott számú bájt
6	Ellenőrző összeg (CheckSum)	1 bájt	Az első adatmezőt (Startkód) kivéve az összes bájt összegének a kettes komplemensét képezzük. Az ellenőrző összeg ennek a legalsó bájtja lesz.

16.3.1.1. ábra

A 4. adatmező rekordtípusai:

0x00	adatrekord
	Adatot és 16 bites címet tartalmaz.
0x01	End Of File (fájl vége) rekord
	A fájl végét jelölő rekord. Adatot nem tartalmaz. Ennek kell lennie a fájl utolsó sorának, csak egy ilyen engedélyezett fájlonként. Általában ':00000001FF'.
0x02	Kiterjesztett szegmenscím rekord, szegmens báziscím.
	Akkor használatos, amikor a 16 bites címtartomány nem elég, megfelel a 8086 valós módú címzésnek. A cím a 02-es rekordban megszorzódik 16-tal és hozzáadódik a következő 00-ás rekordban megadott címhez. Így a maximális megcímezhető tartomány 1 MB-ra növekszik. Ennek a rekordnak Cím mezője 0x0000-t kell – hogy tartalmazzon, a Bájtok száma pedig 0x02-t (mivel egy szegmens 16 bites). A szegmenscím legalacsonyabb értékű számjegye mindig 0x00.
0x03	Kezdő szegmenscím rekord
	A 8086 processzoroknál meghatározza a CS:IP regiszterek kezdőtartalmát. A Cím mező 0000, a Bájtok száma 04, az első két bájt a kódszegmens (CS), az utolsó kettő pedig az utasítás mutató (IP) értékét adja meg.
0x04	Kiterjesztett lineáris cím rekord
	Teljes 32 bites címzést tesz lehetővé. A Cím mező 0000, a Bájtok száma 02. A két adatbájt ebben az esetben a 32 bites cím felső 16 bitjét adja meg, az alsó 16 bit pedig a 00 rekord Cím mezőjében található.
0x05	Lineáris címzés kezdete rekord
	A Cím mező 0000, a Bájtok száma mező 04. A 4 adatbájt tartalmazza a 32 bites címet a 80386 vagy attól újabb processzorok EIP regisztere számára.

16.3.1.2. ábra

A Microchip PIC mikrovezérlőiben a 8 és a 32 bites változatot használják.

Az alábbi példán egy három soros program lefordított kódját mutatja be. A programrészlet:

```
ORG 0

MAIN

MOVLW 0x00

MOVWF 0x06

GOTO MAIN

END
```

Az ORG 0 csak fordító direktíva, azt mondja meg a fordítónak, hogy az ezt követő parancs kerüljön az ORG direktíva után megjelölt programmemória címre. Jelen esetben a 0. címre, így a MAIN címke is a 0. programmemória címre fog utalni. Ezután a programmemória címzése lineárisan növekszik. Az ezt követő parancsok 14 bites megfelelői a parancsok leírásai alapján (7. fejezet):

MOVLW 0x00 $\Rightarrow$ 11 0000 0000 0000 = 0x3000

MOVWF 0x06 $\Rightarrow$ 00 0000 1000 1010 = 0x0086

GOTO MAIN $\Rightarrow$ 10 1000 0000 0000 = 0x2800

Az END szintén fordító direktíva, nem fog bekerülni a lefordított kódba. A fordító által létrehozott HEX fájl tartalma:

```
:02 0000 04 0000 FA

:06 0000 00 003086000028 1C

:00 0000 01 FF
```

Látható, hogy minden sor a „:” Start rekorddal kezdődik. Az első sor két adatbájtot tartalmaz, a rekordtípus szerint kiterjesztett lineáris címzést használunk. Az itt megadott adatbájt a programmemória cím felső 16 bitjét határozza meg (mivel összesen három sort tartalmaz a programmemória, ez az érték értelemszerűen 0x0000). Példánkban a második sor tartalmazza a programmemóriába írandó három lefordított parancsot LSB, MSB sorrendben. Ez összesen hat bájt. A harmadik sor pedig a fájl végét jelző rekord.

Programozáskor a programozó ennek a fájlnak a tartalmát tölti le a mikrovezérlőbe soros kommunikáció segítségével.