MSP430G2553 için assembler düzeyinde SIC/SIC-XE mimarileriyle karşılaştırmalı olarak aşağıdaki kavramları araştıracağım:

1. Literals
2. EQU direktifi
3. Expressions (ifadeler)
4. Program Blocks
5. Relocation (yeniden konumlandırma)
6. Nesne kodu üretimi ve formatı
7. SIC/SIC-XE'e kıyasla yeni komut ve direktifler

Araştırmayı tamamladığımda ayrıntılı bir karşılaştırmalı rapor sunacağım.

**MSP430 Assembler ile SIC/SIC-XE Karşılaştırması**

**1. Literaller (Literals)**

**MSP430 Assembler:** MSP430 mimarisinde sabit değerler doğrudan **immediate operand** olarak kullanılabilir. Örneğin, bir değeri bir registre yüklemek için MOV #10, R5 gibi bir sentaks kullanılır; burada #10 doğrudan 10 sayısal literalini temsil eder. MSP430 assembler, bu tür sabitleri doğrudan makine komutunun parçası olarak işler ve genelde ayrı bir *literal havuzu* oluşturmaz. Yani literaller için özel bir = önekine ihtiyaç yoktur, sabit değerler anında operand şeklinde verilir. Mimarinin 16 bitlik yapısı sayesinde 16 bitlik immediate değerler doğrudan komut içinde saklanabilir. Eğer bir sabitin kod içinde depolanması gerekirse, geliştirici bunu .byte, .word gibi veri tanımlama direktifleriyle bellek içerisinde tanımlar; assembler bu verileri uygun bölüme yerleştirir. Sonuç olarak MSP430 assembler’da literaller geliştirici açısından basittir ve *literal pool* kavramı görünmez.

**SIC/SIC-XE Assembler:** SIC (Simplified Instructional Computer) mimarisinde orijinal SIC işlemcisi **doğrudan immediate operand modunu desteklemez**; bu özellik yalnızca genişletilmiş SIC/XE modelinde mevcuttur[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20The%20operand%20value%20is,001A%20ENDFIL%20LDA%20%3DC%E2%80%99EOF%E2%80%99%20032010). SIC kodunda sabit bir değeri kullanmak için assembler’ın *literal* mekanizması vardır. Programcı literalleri = işaretiyle belirtir, örneğin LDA =C'EOF' ifadesinde 'EOF' stringi bir literal olarak tanımlanır. Assembler, literalleri programın başka bir adresinde sabit değer olarak oluşturur ve ilgili komutun operandı olarak bu değerin adresini kullanır[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Literals%20%01%20The%20assembler,3%20010003). Tüm literaller genellikle programın sonunda bir *literal havuzu* içinde depolanır. SIC assembler, kaynak kodda karşılaştığı literalleri toplar ve varsayılan olarak **END direktifinden sonra** bunların bir listesini oluşturur[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%06%20Literal%20Pools%20%01%20Normally,containing%20all). Gerekirse assembler, LTORG direktifi ile literalleri programın ortasında bir havuza yerleştirebilir; bu komut, önceki LTORG’dan beri biriken tüm literalleri o noktada belleğe yazar[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Assembler%20directive%20LTORG%20%01,addressing%20may%20not%20be%20allowed). Bu yöntem, özellikle PC-bağıl adresleme kullanan SIC/XE komutlarında literallerin hedef komuta çok uzak kalmamasını sağlar[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=it%20generates%20a%20literal%20pool,addressing%20may%20not%20be%20allowed). SIC/XE modelinde # öneki ile immediate adresleme eklenmiştir, ancak orijinal SIC’te immediate yoktu; bu nedenle literaller orijinal SIC için sabit değer kullanmanın bir yoludur.

**Karşılaştırma:** MSP430’da literaller doğrudan komut içinde verilir ve ek bir havuz yönetimi gerektirmez. SIC/SIC-XE’de ise assembler’ın yönettiği bir literal konsepti vardır. Örneğin, SIC’te LDA =5 yazıldığında assembler programın sonunda 5 değeri için bir alan ayırır ve komutu o alana yönelik yükleme olarak makine koduna çevirir. SIC assembler çıktısında literaller tek kopya olarak yer alır, aynı literal birden fazla kullanılsa bile assembler bunları tek bir bellek konumunda tutar[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=literal%20operands%20used%20since%20previous,addressing%20may%20not%20be%20allowed). MSP430’da buna ihtiyaç yoktur çünkü aynı sabit değeri defalarca kullansanız da her seferinde doğrudan komuta yerleştirilebilir. Ayrıca MSP430’da küçük sabitler (0, 1, 2, 4, 8 gibi) *constant generator* sayesinde özel kısayollarla yüklenebilir (bu mimariye özgü bir detay); SIC tarafında böyle bir özellik yoktur. Özetle, literallerin yönetimi SIC/SIC-XE’de assembler tarafından literal havuzları ile yapılırken, MSP430’da mimari destek sayesinde daha şeffaftır.

**Not:** Aşağıdaki tabloda MSP430 ve SIC/SIC-XE’de literal kullanımının farkları özetlenmiştir:

| **Literal Kullanımı** | **MSP430 Assembler** | **SIC/SIC-XE Assembler** |
| --- | --- | --- |
| **Immediate (Doğrudan) Değer** | # öneki ile desteklenir; sabit değer direkt komut içinde yer alır. Ör: MOV #100, R10. | Orijinal SIC'te desteklenmez; SIC/XE’de # öneki kullanılabilir (immediate adresleme sadece SIC/XE’de vardır). |
| **Literal Sabit (=)** | Ayrı bir literal tanımı gerekmez; gerekli durumda .word/.byte ile veri tanımlanır. | = ile tanımlanır, assembler bu sabiti bellek havuzunda depolar. Ör: LDA =X'05' literal havuza 0x05 değeri yazar. |
| **Literal Havuzu** | Konsept olarak yok (assembler otomatik yerleşim yapar). | Var. Literaller genelde program sonunda havuzda toplanır; LTORG ile ara havuz açılabilir. |
| **Örnek** | MOV #0x55, R5 (0x55 değeri doğrudan R5’e yüklenir). | LDA =C'EOF' (’EOF’ stringi için assembler program sonunda bellek ayırır ve LDA komutu oraya yönelik yükleme yapar). |

**2. EQU Direktifi (Sabit Tanımlama)**

**MSP430 Assembler:** MSP430 assembly dilinde bir sabit değeri bir sembole eşlemek için .equ ya da eş anlamlısı olan .set direktifi kullanılır. Söz dizimi: Sembol .equ İfade. Bu şekilde bir sembole bir ifade sonucu bağlanır ve bu sembol assembly sırasında sabit gibi kullanılabilir[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20The%20,Set%20bval%20%3D%200100h). Örneğin, MAX\_LEN .equ 100 dediğimizde assembler, MAX\_LEN sembolünü 100 değeriyle ilişkilendirir. Bu sembol daha sonra bir komutta kullanıldığında, sanki sayısal sabit 100 yazılmış gibi işlem görür. .equ ile tanımlanan semboller assembly zamanında hesaplanır ve **nesne koduna herhangi bir veri yazmaz**, sadece sembol tablosunda değer tutar[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=match%20at%20L5413%20The%20,asg). MSP430 assembler’da .equ ile atanan semboller yeniden tanımlanamaz; aynı isim için ikinci bir değer atamasına izin verilmez (sabit tanımı bir kereliktir)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20The%20,Set%20bval%20%3D%200100h). Ayrıca .equ ifadesinde başka semboller kullanılıyorsa, bunların önceden tanımlanmış olması gerekir; ileriye yönelik (forward) referanslara izin verilmez[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=,in%20the%20current%20source%20module). .equ ile iliştirilen değer mutlak bir sabit olabileceği gibi, bir adres ifadesi de olabilir. Eğer ifade tanımlı bir label adresiyse ve link aşamasında belirlenecekse, assembler bu sembolü *relocatable* (taşınabilir) olarak işaretler. Örneğin BUF\_END .equ BUFFER+100 gibi bir tanımda BUFFER başlangıç adresi daha sonra belirlenecekse, BUF\_END de relocatable olarak işaretlenir[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=Undefined%20external%20symbols%20and%20symbols,is%20assigned%20is%20also%20relocatable). MSP430 assembler ayrıca .global ile birlikte .equ kullanarak bu sabitleri diğer modüllere de açığa çıkarma imkânı sunar (global sabit tanımı)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=Symbols%20defined%20with%20,can%20define%20global%20absolute%20constants)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=Symbols%20defined%20with%20,you%20can%20define%20global%20absolute).

**SIC/SIC-XE Assembler:** SIC assembler’da EQU direktifi benzer şekilde sembollere değer atamak için kullanılır. Söz dizimi: Sembol EQU Değer şeklindedir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Assembler%20directive%20EQU%20%01,%01%20with%20MAXLEN%20EQU%204096). Programcı, magic number denilen sabit sayıların kod içinde doğrudan kullanılmasını önlemek için EQU ile anlamlı isimler tanımlayabilir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Allows%20the%20programmer%20to,MAXLEN). Örneğin, SIC programında MAXLEN EQU 4096 tanımı yapılıp sonra +LDT #MAXLEN komutu kullanılarak, doğrudan #4096 yerine daha okunaklı bir sembolle yazım sağlanır[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20To%20improve%20the%20program,RMO%20A%2CX%01%20X%20EQU%201). EQU ayrıca registerlere isim vermek için de kullanılabilir; örneğin X EQU 1 diyerek indeks register’ine X sembolü atanabilir, böylece kodda TIX X şeklinde anlaşılır bir kullanım elde edilir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%2BLDT%20,BUFFER). SIC assembler, EQU ifadesindeki sembollerin değerini pass 1 sırasında çözebilmek için genellikle ifade içindeki sembollerin daha önce tanımlanmış olmasını şart koşar (ileriye referanslar tipik olarak izinli değildir, ya da en azından değerin hesaplanabilir olması gerekir). SIC’te EQU ile atanan semboller ya mutlak bir değer taşır ya da bir adres ifadesi olabilir. Eğer ifade bir adresler farkı ise (ör. BUF\_SIZE EQU BUFEND-BUFFER), assembler bu hesaplamayı yapar ve sonucu mutlak bir değer olarak atar[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Expressions%20can%20be%20%01,one%20can%20be%20paired%20as). Genel kural, SIC assembler’da bir **ifadenin relocatable sayılmaması için adresli terimlerin birbirini götürmesi** gerekir: Örneğin BUFEND - BUFFER gibi bir ifade, iki bellek adresinin farkı olduğu için program başlangıcından bağımsız mutlak bir uzunluk verir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Only%20absolute%20terms%20%01,must%20have%20a%20positive%20sign). Buna karşılık TABLE EQU OPTAB + (BUFEND-BUFFER) gibi bir ifade eğer tek bir bellek adresi netice veriyorsa relocatable olabilir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20MAXLEN%20EQU%20BUFEND,BUFEND%20%E2%80%93%20BUFFER).

**Karşılaştırma:** Her iki assembler’da da EQU, **sembolik sabit** tanımlamak içindir ve programın okunabilirliğini artırır. Temel farklar sözdizimi ve kapsam ile ilgilidir. MSP430 assembler’da direktif .equ (veya .set) şeklindeyken, SIC assembler’da EQU olarak yazılır (noktasız). İşlevsel olarak çok benzerler: Değerler atanır ve nesne koduna gömülmez, sadece sembol olarak kullanılır. MSP430 assembler modern bir araç olduğundan .equ ifadesinde daha zengin hesaplamalar ve farklı türde sabitler kullanılabilir (örneğin ileri seviye ifadeler, yerel/harici semboller ayrımı). SIC assembler ise temel aritmetik operatörlerle sınırlıdır ve harici bağlam olmadığı sürece tüm semboller tek bir program içindedir. SIC/XE bağlamında, eğer program birden fazla control section’a bölünmüşse (EXTREF/EXTDEF ile), bir control section içindeki EQU ile tanımlı semboller diğerinde kullanılamaz (harici sembol olarak tanıtılmadıkça). MSP430’da ise .global/.def/.ref direktifleriyle .equ sembolleri modüller arası kullanılabilir kılınır[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=The%20,symbol%20in%20the%20symbol%20table)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=A%20global%20symbol%20is%20defined,module%20actually%20uses%20the%20symbol). Sonuç olarak, EQU kullanımındaki farklar büyük oranda sözdizimsel; her iki assembler da sabit tanımlamada benzer davranır.

**Örnek:**  
*SIC:* ALEN EQU BUFEND - BUFFER – Buffer alanının uzunluğunu hesaplar, ALEN sembolüne atar (relokasyonsuz mutlak değer)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Only%20absolute%20terms%20%01,must%20have%20a%20positive%20sign).  
*MSP430:* ALEN .equ (BUFEND - BUFFER) – Benzer şekilde, ALEN hesaplanıp sabit olarak tanımlanır (linker aşamasına kadar BUFEND adresi bilinmese bile bu fark sabit bir uzunluk olduğundan absolute kabul edilir).

**3. İfadeler (Expressions)**

**MSP430 Assembler:** MSP430 assembly’de hemen hemen her yerde *ifade* kullanılabilir. Sayılar, semboller, bellek adresleri ve operatörler kullanarak assembler zamanında hesaplanacak değerler yazmak mümkündür[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=Nearly%20all%20values%20and%20operands,expressions%20that%20are%20referred%20to)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=A%20constant%20expression%20is%20any,invocation%20on%20a%20constant%20expression). MSP430 assembler, C benzeri zengin bir operatör setini destekler: dört işlem (+ - \* /), mod alma (%), bit kaydırma (<< >>), bitwise AND/OR/XOR (& | ^), mantıksal değil (!), birlerin tümleci (~), karşılaştırmalar (== != < > <= >=) vb. Bu operatörlerin öncelik sırası ve birleşim kuralları da C diline benzer şekilde tanımlanmıştır[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20a%20literal%20constant%20%E2%80%A2,For%20ELF%2C%20such%20expressions%20may)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20an%20external%20symbol%20%E2%80%A2,expression%20may%20be%20known%20at). Örneğin, assembler içinde VAL .equ 5+3\*2 yazıldığında VAL = 11 olarak hesaplanır. Parantez kullanımı da desteklenir ve parantez içi öncelikle değerlendirilir. İfadelerde semboller kullanılabilir; ancak MSP430 assembler bir ifadenin değerlendirilebilmesi için, içindeki tüm sembollerin o ana dek tanımlanmış olmasını bekler[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=,in%20the%20current%20source%20module). Bu nedenle, .equ veya benzeri direktiflerde ileriye dönük sembol kullanımı kısıtlanmıştır. MSP430 assembler, ifadeleri **mutlak (absolute)** veya **yeniden yerleştirilebilir (relocatable)** olarak sınıflandırır[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=A%20constant%20expression%20may%20be,following%3A%20%E2%80%A2%20a%20literal%20constant)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20a%20built,be%20any%20of%20the%20following). Bir ifade eğer tamamen sayısal ya da aynı bölüm içindeki adres farklarından oluşuyorsa mutlak kabul edilir ve değeri assembly zamanında kesinleşir. Eğer ifade harici bir sembol içeriyorsa veya bir adres değeri barındırıyorsa, bu ifade relocatable olarak işaretlenir; değeri assembly sırasında kısmen bilinse de kesin adres link aşamasında belirlenecektir[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20a%20built,be%20any%20of%20the%20following). MSP430 nesne dosyasında bu tür ifadeler için relocation girişleri oluşturulur (aşağıdaki *Relocation* bölümüne bakınız). Örneğin JMP LABEL+4 şeklinde bir kullanımda, eğer LABEL aynı dosyada tanımlıysa assembler 4 eklentili adresi hesaplamaya çalışır; fakat genelde dallanma ofseti zaten assembler tarafından çözülecektir. İfadeler, sadece sabit değer gereken yerlerde değil, bazı komut operandlarında da kullanılabilir. Örneğin, bir bellek adresine offset eklenmesi gereken durumlarda MSP430 assembler “adres sabiti ifadeleri”ni destekler: MOV X(R4), R5 gibi bir operand, R4 registrinin içeriğine X sembolünün değeri kadar offset uygulayarak bellek adresine erişir (X burada sabit bir ifade olmalıdır). Kısacası MSP430 assembler, yüksek seviyeli bir dildeki sabit ifadelere benzer esneklikte işlem yapabilir; bu sayede tabloların boyutları, sabit çarpanlar vb. rahatça tanımlanabilir.

**SIC/SIC-XE Assembler:** SIC assembler’da da ifadeler kullanılabilir ancak kapsam ve karmaşıklığı daha sınırlıdır. SIC dili, semboller ve sayılar üzerinde temel aritmetik işlemlere izin verir. Örneğin, bir *RESB* direktifinde uzunluğu dinamik hesaplamak için BUFFER RESB 80+20 yazmak mümkün olup bu, 100 byte’lık bir yer ayırır. En tipik kullanım, EQU direktifinde veya WORD sabiti tanımlarken adres farklarıdır: TABLE\_SIZE EQU BUFEND - BUFFER gibi bir ifade, iki label arası mesafeyi hesaplar. SIC/SIC-XE assembler, ifadeleri **mutlak** veya **göreli (relative)** olarak kategorize eder. Mutlak bir ifade yalnız sabit sayılardan ya da kendi içinde adresli terimlerin birbirini götürdüğü kombinasyonlardan oluşur[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Expressions%20can%20be%20%01,one%20can%20be%20paired%20as). Relative (göreli) bir ifade ise program başlangıcına göre bir adres değerini temsil eder. SIC örneğinde programın başlangıç adresi 0 kabul edilirse, bir label’ın değeri relative bir ifadedir. Kural olarak, bir ifadede **en fazla bir adet göreli terim** saf halde kalabilir; diğer adresli terimler çiftler halinde çıkarma ile birbirini yok etmelidir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Relative%20terms%20in%20pairs,BUFEND%20%E2%80%93%20BUFFER). Örneğin:

* MAXLEN EQU BUFEND - BUFFER ifadesinde BUFEND ve BUFFER adresli terimler çıkarma ile birbirini yok ederek sonuç mutlak bir değere (buffer uzunluğu) dönüşür[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Only%20absolute%20terms%20%01,must%20have%20a%20positive%20sign). Bu nedenle MAXLEN mutlak bir sabit olur.
* PTR EQU BUFEND + START gibi bir ifade ise iki ayrı adres toplandığı için anlamsız kabul edilir, assembler hata verir (iki göreli terim artı işaretiyle birleşemez).
* SYMTAB EQU OPTAB + (BUFEND - BUFFER) ifadesinde BUFEND-BUFFER kısmı mutlak bir değerdir (diyelim 30 çıktı), OPTAB ise program içindeki bir adres (göreli). Bu ifade neticede OPTAB + 30 şeklinde **göreli bir değer** oluşturur; SYMTAB sembolü relocatable (program başlangıcı değişirse SYMTAB da değişecektir) olarak kabul edilir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20All%20the%20relative%20terms,BUFEND%20%E2%80%93%20BUFFER).

SIC/XE assembler, ifadelerde orijinal SIC ile aynı kuralları izler, ancak ilave olarak EXTREF/EXTDEF harici semboller de bulunabileceğinden, bir ifade harici bir sembol içeriyorsa o da relocatable olarak değerlendirilir. Yine de kısıt aynıdır: tek bir external veya local adres terimi kalabilir. Örneğin SUM EQU ALPHA + BETA (ALPHA ve BETA farklı control section’dan geliyorsa) bu da çözümlenemez bir ifade olur. Genel olarak SIC ortamında operatör kümesi MSP430 kadar geniş değildir; genellikle + - \* / desteklenir, bit düzeyinde işlemler veya karşılaştırma operatörleri assembler seviyesinde yoktur. Mantıksal bağlamda 0 dışı değer “true” kabul et vb. gibi .if yapıları da olmadığından, < ya da == gibi operatörler kullanılmaz.

**Karşılaştırma:** MSP430 assembler ifadelere yönelik çok daha **zengin** bir dil sunar. Hem operatör çeşitliliği fazladır hem de 16 bitlik ve 32 bitlik sayılar, hatta kayan nokta sabitleri (.float ile) ifadelerde yer alabilir. Ayrıca MSP430, makro dilde .if gibi yapılarla koşullu derleme yapabildiği için karşılaştırma operatörlerini de bu bağlamda kullanabilir (örneğin .if CONST <= 10 şeklinde şart kontrolü yapılabilir)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=IF_5%3A%20.if%20SYM1%20,byte%20SYM1%20%3B%20Greater%20than). SIC assembler ise ifadeleri sadece adres ve sayısal sabit hesaplamaları için kullanır, program kontrolü için kullanmaz. Her iki assembler da relocatable vs absolute ayrımını yapar, ancak MSP430’un ELF tabanlı yapısında “relocatable ifade” genelde “içinde harici bir sembol var, linker halledecek” demektir[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=A%20constant%20expression%20may%20be,following%3A%20%E2%80%A2%20a%20literal%20constant)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20a%20built,be%20any%20of%20the%20following). SIC’te relocatable demek genellikle “bu bir adres, loader başlangıç adresini eklemeli” demektir. Sonuç olarak MSP430’da ifadeler daha esnek ve güçlüdür; SIC/SIC-XE’de ise daha kısıtlı ama anlaşılırdır. İki tarafta da geçerli temel kural, **ifadelerin assembler tarafından hesaplanabilir olması** ve adres içeriği barındırıyorsa uygun şekilde işaretlenmesidir.

**4. Program Blokları ve Bölümler (Program Blocks / Sections)**

**MSP430 Assembler (Program Bölümleri – Sections):** MSP430 kodu, derleyici ve assembler düzeyinde **segmentlere (section)** ayrılarak yönetilir. Varsayılan olarak TI assembler, kod için .text bölümü, başlatılmış veri için .data bölümü ve başlatılmamış veriler için .bss bölümü kullanır[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=Object%20files%20usually%20contain%20three,for%20how%20sections%20are%20handled). Geliştirici, isterse .sect "isim" direktifiyle özel bir adla yeni bir bölüm tanımlayabilir ya da .usect ile özel bir *uninitialized* bölüm açabilir[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=Mnemonic%20and%20Syntax%20Description%20See,bss)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=.sect%20,size%20in%20bytes). Bu bölümler assembler çıktısında ayrı segmentler olarak yer alır. Linker, bu segmentleri hedef işlemcinin bellek haritasına uygun şekilde yerleştirir (**placement** adı verilen süreç)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=One%20of%20the%20linker%27s%20functions,using%20sections%20can%20help%20you). Örneğin MSP430’da .text genellikle flash (ROM) bölgesine, .data ve .bss RAM bölgesine konulur. Tüm bölümlerin adresleri ve birleştirilmesi linker tarafından yapılır, assembler sadece her bölüm için sıfırdan başlayan bir **lokasyon sayacı (location counter)** kullanarak kodu konumlandırır. Section kullanımı sayesinde, kod ve veri farklı bellek türlerine sahip gömülü sistemlerde doğru yerlere dağıtılabilir. MSP430 assembler, bölüm direktifleri haricinde **kontrol bölümleri (control sections)** kavramını ayrıca desteklemez; zira her assembly kaynağı kendi bölümlerini tanımlar ve birleştirme linker ile olur. Ancak bir program birden fazla kaynak dosyaya bölünebilir, bu durumda her modülün .text, .data vs. segmentleri linker aşamasında birleşecektir. MSP430 assembler’da segmentlerin sırası kaynakta yazıldığı gibi olmak zorunda değildir; her bölüm kendi adres alanında başlar ve linker kurallarına göre yerleştirilir. Ayrıca TI assembler, .intvec gibi özel direktiflerle kesme vektör tablosu girişi oluşturmaya imkân tanır (bu da bir named section yaratır)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=Mnemonic%20and%20Syntax%20Description%20See,bss). Özetle, MSP430 ortamında program blokları kavramı, **bölümler** (sections) şeklinde uygulanır ve çok esnektir: geliştirici isterse tüm kodu tek bir .text içinde tutabilir veya farklı amaçlarla birden fazla bölüm kullanabilir. Bölümlerin nihai adresleri ve birleşimi otomatik olarak linker ile çözülür.

**SIC/SIC-XE Assembler (Program Blocks):** SIC assembler, temel haliyle tüm kod ve veriyi tek bir blokta işler. Fakat Leland Beck’in *System Software* anlatımında, assembler’ın gelişmiş bir özelliği olarak **program blocks** kavramı tanıtılmıştır. Program blokları, bir assembly kaynağı içinde **USE** direktifi kullanılarak birden fazla kısma ayrılabilir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Assembler%20directive%3A%20USE%20%01,11). Başlangıçta tüm kod *unnamed (isimsiz)* varsayılan bloğa aittir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Assembler%20directive%3A%20USE%20%01,11). Örneğin, USE CDATA yazarak yeni bir blok başlatılabilir; sonrasında gelen talimat ve veriler CDATA adlı blokta toplanacaktır. Aynı kaynak dosyada tekrar USE ile başka bir bloğa veya varsayılan bloğa geçmek mümkündür[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Assembler%20directive%3A%20USE%20%01,11). Assembler her blok için ayrı bir location counter tutar. Assembly tamamlandığında, assembler bu blokları birleştirerek nihai adresleri atar. Genellikle, blokların sıralaması tanımlandıkları sıraya veya programcının belirttiği bir düzene göre yapılır. Beck’in kitabındaki örnekte, üç blok kullanılmıştır: **default** (ana kod için), **CDATA** (küçük veriler için) ve **CBLKS** (büyük sürekli bellek blokları için)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Three%20blocks%20are%20used,of%20larger%20blocks%20of%20memory)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Example%3A%20pp,11). Assembler, bu segmentlerin parçalarını bir araya getirip her bloğa bir başlangıç adresi verir. Örneğin default bloğu programın başlangıç adresinden başlar; CDATA bloğu default blok kodunun hemen bitiminden sonraki adrese yerleştirilir; CBLKS ise en sona konulur. Sonuçta, object program içinde her blok kendi alanında toplanmış olur[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%02%02Program%20Blocks%20%01%20Assembler%20directive%3A,11)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=). **Program readability (okunabilirlik)** açısından, bloklar sayesinde geliştirici kaynak kodunda veriyi ve kodu mantıksal olarak yakın tutabilir (örneğin bir tabloyu tanımlamaları, kullanıldığı kod bloğu ile aynı yerde yazmak ancak farklı bir blokta saymak)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%02%02Program%20Blocks%20%01%20Assembler%20directive%3A,11)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=). Assembler bunları sonunda birleştirir, böylece büyük veriler programın sonunda toplanabilir gibi optimizasyonlar yapılabilir. SIC program blokları, tek bir object file içinde, tek bir **control section** olarak yüklenir; yani loader açısından halen tek programdır, sadece adres atlamaları assembler tarafından halledilmiştir. Program bloklarını karıştırmamak gereken kavram **control section (CSECT)** kavramıdır: CSECT, SIC/XE’de ayrı yüklenebilir program parçalarıdır ve EXTREF/EXTDEF ile birbirine bağlanır. Program blokları ise tek bir control section içindeki alt bloklardır. USE direktifi ile ayrılan bloklar, **aynı object dosyada birlikte yer alır** ancak adresleri ayrı hesaplanır.

**Karşılaştırma:** MSP430 assembler’daki bölüm yaklaşımı ile SIC assembler’ın program blokları benzer amaca hizmet eder: kodu ve veriyi lojik segmentlere ayırmak. Ancak uygulama şekli farklıdır. MSP430’da bölüm ayrımı daha formaldir ve ELF nesne dosyası içinde ayrı segmentler olarak yer alır; linker bu bölümleri donanım bellek haritasına yerleştirir (örneğin .text’i ROM’a, .data’yı RAM’e)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=,two%20basic%20types%20of%20sections)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=The%20smallest%20unit%20of%20an,segment%20information%2C%20but%20does%20not). SIC’te ise bloklar assembler seviyesinde çözümlenir, nesne kodunda blok bilgisi dolaylı olarak adres hesaplarından anlaşılır (her Instruction’ın adresi ilgili blok kombinasyonuyla belirlenmiştir). SIC program bloklarında adres hesaplama manuel sayılabilir: Her blok için başlangıç adresini programcı verebilir ya da assembler ilk bloğu START adresinden, sonraki blokları bir öncekinin bittiği yerden başlatır. MSP430’da programcı bu detaya karışmaz, linker yerleştirme dosyasındaki kurallara göre bunu yapar. Ayrıca MSP430 bölümleri tamamen farklı adres alanlarına da gidebilir (örneğin .text 0x4000’de, .data 0x200’de başlayabilir), oysa SIC blokları genelde art arda yerleştirilir (hepsi tek contiguous yükleme imajı oluşturur).

Aşağıdaki tablo, iki yaklaşımın temel farklarını özetlemektedir:

| **Özellik** | **MSP430 – Sections (Bölümler)** | **SIC – Program Blocks** |
| --- | --- | --- |
| Tanımlama Yöntemi | .text, .data, .bss gibi ön tanımlı veya .sect "isim" ile özel bölüm oluşturma. | USE blockname direktifi ile blok seçme/oluşturma (başta default blok vardır). |
| Amaç | Kod ve veriyi bellek türlerine göre ayırmak; bağlayıcıya yol göstermek. | Kodu mantıksal parçalara ayırmak (ör. verileri ayrı blokta tutup program sonunda yer vermek, okunabilirliği artırmak). |
| Adresleme | Her bölüm kendi adres ofsetiyle başlar (0’dan); linker her bölüme uygun nihai adres atar. | Her blok assembler tarafından sıfırdan adreslenir; assembly sonunda bloklar arka arkaya dizilir ve adresleri hesaplanır. |
| Yükleme (Loader) | ELF segmentleri olarak çıktı; loader her segmenti hedef adresine yükler (farklı bellek bölgelerine olabilir). | Tek bir program imajı; blokların birleşimi loader tarafından tek parça olarak yüklenir. Loader, blok ayrımını bilmez, birleşik kod alır. |
| Örnek Kullanım | Kodlar .text içinde, küçük sabit tablolar .data içinde, büyük boş RAM alanları .bss içinde tanımlanır. Bu bölümler otomatik uygun yerlere gider (Flash vs RAM). | **default** blok: çalıştırılabilir talimatlar; **CDATA** blok: küçük veriler; **CBLKS** blok: büyük buffer’lar tanımlı. Assembler, CDATA ve CBLKS bloklarını default bloktan sonra yerleştirir. |

**5. Relocation (Yeniden Konumlandırma)**

**MSP430 Assembler – Relocatable Kod:** MSP430 için derlenen assembly kodu, *bağlanabilir nesne kodu* olarak üretilir. Bu nesne dosyası ELF formatındadır ve içinde çözülmemiş adres referansları için **relocation kayıtları** içerir[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20Adjusting%20references%20to%20relocated,The%20assembler%20creates%20a). MSP430 assembler, bir komut ya da veri başka bir sembolün adresine referans içeriyorsa, bu adresin tam değeri assembly aşamasında bilinmiyorsa, o konum için bir relocation entry oluşturur[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20Adjusting%20references%20to%20relocated,The%20assembler%20creates%20a). Örneğin, MOV.W #VAR, R6 gibi bir komutta VAR sembolü aynı modülde daha sonra tanımlıysa veya harici bir sembolse, assembler R6’ya yükleme için nesne koduna şimdilik bir yer tutucu değer koyar ve nesne dosyasına "VAR adresi buraya eklenecek" şeklinde bir relocation kaydı ekler. Linker aşamasında, tüm bu relocation girişleri işlenir: Linker her bir sembolün nihai adresini bilir ve ilgili nesne kodu konumlarına bu adresleri ekler/düzeltir[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=begin%20at%20address%200%20in,linker%20can%20relocate%20sections%20by)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20Adjusting%20references%20to%20relocated,The%20assembler%20creates%20a). MSP430 ELF formatında relocation, genelde "bir sembolün değeri bu offset’e eklenecek" biçimdedir. Yani her relocation kaydı bir sembol referansını ve bunu içeren kod konumunu gösterir. Link edilirken, eğer sembol kendi modülünüzdense sadece bölümün baz adresine göre ayarlanır, eğer harici bir modüldense linker sembolün adresini bulup yazar. MSP430 assembler, tüm kodu 0 adresinden başlatarak varsayılan relocatable output üretir; programın gerçek yükleme adresi önemli değildir çünkü ELF içinde bölüm baz adresleri 0 kabul edilir, sonra linker bunları hedef bellekte nereye koyacağına karar verir[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=begin%20at%20address%200%20in,linker%20can%20relocate%20sections%20by). Özetle, MSP430 ortamında relocatable kod doğal bir durumdur: Assembler adresleri sabitler yerine sembollerle temsil ederek işaretler, linker düzeltir. Geliştirici isterse tamamen **konumdan bağımsız (position-independent)** kod da yazabilir, ancak genelde buna gerek kalmaz çünkü linker her şeyi uygun şekilde ayarlar.

**SIC/SIC-XE – Relocation Mekanizması:** SIC ve SIC/XE assembler, ürettiği object programda relocatable adresler için **Modification record (M kaydı)** adı verilen kayıtlar kullanır[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Modification%20record%20%01%20Col,to%20be%20added%20to%20or). SIC’te program tek bir parça olarak yüklenir, ancak programın derlendiği başlangıç adresi (HEADER kaydındaki) ile fiili yükleme adresi farklı olursa, mutlak adres içeren alanların ayarlanması gerekir. Örneğin, SIC makinesinde bir komut 3 byte’lık formatta direkt bir adres taşıyorsa (ör: LDA SOMEADDR), eğer program 0x1000 adresine yüklenmişse bu SOMEADDR sabit adresine yükleme komutu yanlış yere işaret edebilir. Loader bu durumda, assembler’ın bıraktığı ipuçlarını kullanarak adresi düzeltir. SIC object programı üç temel kayıt içerir: **H (Header)**, **T (Text)** ve **E (End)**. SIC/XE’de ek olarak **M (Modification)** kayıtları ve harici referanslar için **D/R** kayıtları vardır. Relocation’ın asıl gerçekleştiği yer M kayıtlarıdır. Assembler, çözülmemiş veya yükleme konumuna göre değişmesi gereken bir adres gördüğünde, bir M kaydı yazar. Basit SIC programlarında, genellikle başlangıç adresi dışında harici sembol olmadığından, M kayıtları *programın tüm adresini yükleme adresiyle düzeltmek* için kullanılır. Örneğin, program 0x0000 adresinden derlendiyse ama loader 0x1000’e yüklüyorsa, her mutlak adres için M kaydı mevcut olur ve loader her birine 0x1000 ekler. SIC assembler bu M kayıtlarını şu formatta yazar: **M^adres^boyut^+/-Sembol**. Eğer program içinde harici referans yoksa, Sembol olarak genelde programın başlangıç adresini temsil eden sabit bir değer kullanılır (veya H kaydındaki başlangıç adresini loader otomatik uygular). SIC/XE’de harici semboller (EXTREF/EXTDEF) varsa, M kayıtlarında hangi sembolün değeri ekleneceği de belirtilir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Modification%20record%20%01%20Col,to%20be%20added%20to%20or). Örneğin +JSUB FARFUNC (format 4 uzun adresli çağrı) kullanılmışsa, assembler o konum için M kaydı koyar: “bu adres alanına FARFUNC sembolünün değeri eklenecek” şeklinde (çünkü FARFUNC başka bir control section’dan gelecek). Loader, FARFUNC’ın program yükleme adresini D kayıtlarından bulur ve M kaydına göre ilgili yere ekler. Orijinal SIC’de EXTREF olmadığı durumlarda, M kaydı sadece başlangıç adresinin ekleneceğini belirtir.

Relocation konusunda SIC ve SIC/XE arasında mimari bir fark da şudur: **SIC/XE, PC-relative ve Base-relative adresleme modları sunar**, bu sayede birçok komutun hedef adresi komutun konumuna göre göreli belirtilebilir. Bu durumda, programın taşınması (relocation) gerekse bile, PC’ye göre verilen ofset aynı kalır, komut çalışırken PC değeri değiştiğinden otomatik adapte olur. Örneğin SIC/XE’de LDA LABEL komutu, eğer Format 3 (3 byte) kullanıldıysa PC-relative 12 bitlik ofsetle kodlanır. Program başka adrese taşınsa bile, LABEL ile o komut arasındaki mesafe değişmediği için bu kod çalışmaya devam eder, relocation gerektirmez. Buna karşın, **format 4** (4 byte) yani doğrudan 20 bit adres içeren komutlar relocation gerektirir. Orijinal SIC’te tüm adresler 15 bitlik mutlak adresler olarak komutta saklandığından, programın yükleme adresi değişirse bu adreslerin hepsi düzeltilmelidir. Bu nedenle orijinal SIC loader’ı, her **Text kaydı** için başlangıç adresini kullanarak içerdiği adresleri ofsetleyebilir veya M kayıtlarından yararlanır. Beck’in tasarımında, text kayıtlar içindeki her adres için ayrı relocation biti öngörülmüştür (her 3 byte’lık kelime için 1 bit, adres alanı mı değil mi diye). M kaydı aslında harici semboller için kullanılır, saf relocation ise genellikle basitçe tüm programı ofsetleme şeklinde anlatılır. Fakat sistematik yaklaşım olarak, object programlarda relocation M kayıtları ile ifade edilir.

**Karşılaştırma:** MSP430 assembler/linker sistemi ile SIC loader sistemi arasında relocation konusunda yaklaşım farkı vardır:

* MSP430 (ELF) sisteminde relocation işlemi **bağlayıcı (linker)** tarafından yapılır, kaynak kod modüler olabilir ve her bir reference için detaylı relocation girişleri tutulur[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=%E2%80%A2%20Adjusting%20references%20to%20relocated,The%20assembler%20creates%20a). Bu modern yöntem, birden çok dosyanın link edilmesini ve adres çözümlemeyi otomatikleştirir. Relocation bilgisi ikili ELF dosyasında saklıdır ve loader genelde bunlarla uğraşmaz; linker son aşamada tüm relocations’ı çözüp yürütülebilir dosya üretir.
* SIC sisteminde relocation doğrudan **loader** tarafından gerçekleştirilir. Assembler, text kayıtları üretirken programın sıfırdan başlayan adreslerini kullanır ve yükleme adresine göre düzeltilecek kısımları M kayıtlarıyla belirtir[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Modification%20record%20%01%20Col,to%20be%20added%20to%20or). Linker ayrı bir adım olmadığı için (basit SIC için), tek modüllü programda loader sadece bu M kayıtlarına bakarak adres ekleme yapar. SIC/XE’de birden fazla control section varsa *linking loader* devreye girer, D (Define) ve R (Refer) kayıtlarıyla semboller bağlanır ve M kayıtları harici sembollere göre çözülür. Yani SIC dünyasında relocation, loader’ın sorumluluğudur ve metin tabanlı kayıtlar üzerinden gerçekleşir.

Mimarinin etkisine gelirsek, MSP430 tüm dallanma komutlarını görece kısa ofsetli (±512 veya ±1024 bayt) PC-relative yaparak zaten konum-bağımsızlık sağlar; benzer şekilde SIC/XE’de PC-relative mod tercih edilerek M kaydı ihtiyacı azaltılır. Fakat mutlak adres kullanan komutlarda (MSP430’da örneğin interrupt vektör adresleri sabittir, SIC/XE’de format4 kullanımı) relocation şarttır. Sonuç olarak her iki assembler da relocatable kod destekler ancak **SIC/SIC-XE’de bu destek assembler çıktısında M kayıtları ile açıkça belirtilirken, MSP430’da ELF içindeki relocation tablo girişleri ile kapalı biçimde ifade edilir**. Kullanıcı açısından MSP430’da relocation otomatik ve görünmezdir; SIC’te ise program yazarı, örneğin başlangıç adresini değiştirince hangi adreslerin düzeltilmesi gerektiğini en azından kavramsal olarak bilir (çünkü o adresler için M kaydı üretilir).

**6. Nesne Kodu Üretimi (Object Code Formatı)**

**MSP430 Assembler Çıktısı:** MSP430 assembler, derlenen program için bir **nesne dosyası** üretir. Bu dosya **ELF (Executable and Linking Format)** formatındadır[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=2,later%20versions%20of%20the%20TI). ELF, modern derleyici sistemlerinin kullandığı standart bir ikili formattır. MSP430 için üretilen ELF içinde, programın bölümlerinin (section) içerikleri, sembol tablosu, relocation tablosu, debug bilgileri (varsa) vb. yer alır[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=The%20object%20files%20created%20by,If)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=2,the%20same%20property%2C%20such%20as). ELF formatı insan tarafından kolay okunabilir değildir; tipik olarak objdump gibi araçlarla incelenir. TI’ın derleyici seti başlangıçta COFF formatını da destekliyordu ancak v15.6 itibariyle sadece ELF EABI kullanılmaktadır[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=COFF%20object%20files%20are%20not,or%20gABI). Bir MSP430 nesne dosyasında, örneğin .text bölümünün makine kodu, .data bölümünün başlatılmış verileri bulunur. Bu bölümlerin her biri için tablo girişleri vardır (isim, boyut, konum vs.). Semboller (label’lar, .global ile işaretlenmiş değişkenler vs.) ayrı bir tablodadır. Eğer program birden fazla dosyaya bölünmüşse, her biri kendi ELF nesne dosyasını üretir ve **linker bu dosyaları birleştirerek** son yürütülebilir dosyayı oluşturur. ELF formatının bir özelliği de relocatable olması; yani ELF içindeki adresler genelde 0’dan başlar, her şey “belirsiz” bir baz adresine göre tanımlıdır, bu sayede linker bunları kolayca farklı adreslere yerleştirebilir. MSP430 assembler çıktısı tipik olarak .o uzantılı bir dosyadır ve doğrudan MCU’ya yüklenemez; link edilip bir **.elf veya .hex** yürütülebilir dosyaya çevrilmesi gerekir. Linker, tüm nesne dosyalarını alır, tanımlı/başvurulan sembolleri eşleştirir, segmentleri birleştirir ve nihai makine kodunu üretir. Son aşamada, istenirse ELF, Intel HEX veya TI-TXT gibi bir formatta dışa aktarılıp programlayıcıya verilir. Özetle, MSP430 assembler **object code olarak makine diline çok yakın, zengin bilgili bir ELF** üretir; bu format SIC’in basit metin formatından oldukça farklıdır.

**SIC/SIC-XE Nesne Kodu:** SIC assembler, çıktısını metin tabanlı bir **object program** şeklinde üretir. Bu object program 3 ana kayıt içerir (SIC standart model için): **Header (H)**, **Text (T)** ve **End (E)** kayıtları. SIC/XE genişletilmiş modelde ise ilave olarak **Define (D)**, **Refer (R)** ve **Modification (M)** kayıtları da olabilir (harici sembollere bağlı olarak). Bu format, Leland Beck’in kitabında detaylı açıklanmıştır:

* **H kaydı (Header):** Programın adı, başlangıç yükleme adresi ve toplam program uzunluğunu içerir[stackoverflow.com](https://stackoverflow.com/questions/74611391/how-can-i-obtain-the-length-of-text-from-object-code-in-sic-assembler#:~:text=,address%20of%20the%20next%20instruction). Örneğin H COPY 001000 00107A gibi bir header satırı, program adının "COPY", başlangıç adresinin 0x1000, uzunluğun 0x107A olduğunu belirtir. Program adı 6 karakterlik alana sığmazsa boşluk ile pad edilir[stackoverflow.com](https://stackoverflow.com/questions/74611391/how-can-i-obtain-the-length-of-text-from-object-code-in-sic-assembler#:~:text=,address%20of%20the%20next%20instruction).
* **T kayıtları (Text Records):** Programın makine kodunu içerir. Her T kaydı, bir başlangıç adresi ve o adresden itibaren gelen baytları verir. SIC formatında bir T kaydı en fazla 30 bayt kod taşır (kitapta örneklendiği üzere). Örneğin T 001000 1E 141033281... gibi bir satır, 0x1000 adresine 0x1E byte uzunluğunda kod yükleneceğini ve devamında bu kodun byte değerlerini listeler. T kayıtları adres sırasına göre gelmek zorunda değildir, loader her birini belirtilen adrese yüklercs.bilkent.edu.trcs.bilkent.edu.tr. Bu esneklik, program blokları kullanıldığında T kayıtlarının adres sırası karışabileceği durumlar içindir (ancak loader hepsini doğru yere koyar)cs.bilkent.edu.tr. Ayrıca ardışık gelen boş alanlar (RESW/RESB) T kayıtlarında atlanır; assembler ya bu kısım için T kaydı yazmaz ya da dolgu yapar. Örneğin, uzun bir RESW varsa assembler bir T kaydını sonlandırıp bir sonrakini boşluğun sonundan başlatabilir[stackoverflow.com](https://stackoverflow.com/questions/74611391/how-can-i-obtain-the-length-of-text-from-object-code-in-sic-assembler#:~:text=with%20spaces,which%2C%20I%20assume).
* **M kayıtları (Modification):** Relocation gerektiren adres alanlarını belirtir (özellikle SIC/XE’de harici referanslar için). Formatı: M ADDRSS LENGTH [+/-]SYM. Örneğin M 001007 05 +COPY bir modification kaydıdır: loader’a, 0x1007 adresindeki 5 yarı-byte uzunluğundaki alanın değerine COPY sembolünün adresini eklemesini söyler[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp2-3.pdf#:~:text=%01%20Modification%20record%20%01%20Col,to%20be%20added%20to%20or). Eğer + yerine - olsaydı, çıkarması gerektiğini belirtirdi. Basit SIC programlarında sembol kısmı genelde programın yükleme başlangıcını temsil eder (program adı sembol olarak kullanılır).
* **E kaydı (End):** Programın giriş noktası adresini içerir ve object kodun sonunu belirtir. Örneğin E 001000, programın 0x1000’den başlayarak yürütüleceğini söyler.

SIC object code **textual (metin)** bir temsildir. Bu nedenle bir metin dosyası olarak saklanabilir ve incelenebilir. Örnek bir SIC object program (COPY programı için) şöyle görünebilir:

H COPY 001000 00107A

T 001000 1E 141033... (ilk 30 byte kod)

T 00101E 15 69F914... (devam kodları)

T 001080 1A 4720... (devam kodları)

M 00106D 05 +COPY (relocation kaydı örneği)

E 001000

Harici semboller varsa, D kayıtları ile tanıtılır (her D satırı bir sembol adı ve relative adresini verir), R kayıtları ile referanslar listelenir (her R satırı bu modülde kullanılan harici sembolleri belirtir). Bu sayede linking loader, birden çok object programı bağlayabilircs.bilkent.edu.trcs.bilkent.edu.tr.

**Karşılaştırma:** MSP430’un ELF formatı ile SIC’in H/T/E metin formatı oldukça farklıdır:

* **Biçim:** ELF ikili bir formattır (binary); SIC object ise ASCII karakterlerle temsil edilir (text). ELF insan tarafından doğrudan okunmazken, SIC object programı bir metin editörüyle bile anlaşılabilir (opcode’lar hex şekilde görünür).
* **İçerik:** ELF’te bölüm tablosu, sembol tablosu, relocation tablosu gibi ayrıntılı ve **yapılandırılmış** bilgiler bulunur[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=The%20smallest%20unit%20of%20an,segment%20information%2C%20but%20does%20not)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=Object%20files%20usually%20contain%20three,for%20how%20sections%20are%20handled). SIC formatında bu bilgiler satırlara serpiştirilmiştir: H satırında program adı/uzunluk, D/R satırlarında harici semboller, T satırlarında kod ve M satırlarında relocation bilgisi vardır. Örneğin ELF’te her sembol için bir tablo girdisi vardır; SIC’te sadece harici semboller listelenir, dahili semboller loader tarafından kullanılmaz.
* **Yeniden Yerleştirme:** ELF’te relocation girdi olarak tutulur ve bağlayıcı tarafından uygulanır; SIC’te relocation M kayıtlarında belirtilir ve loader tarafından uygulanır (yukarıda anlatıldığı gibi). SIC loader’ı çok daha basit bir işlem gerçekleştirir (adresleri toplayıp yazmak), ELF linker’ı ise daha karmaşık bir sembol çözümleme yapar.
* **Çoklu Modül Bağlama:** ELF formatı, birden fazla nesne dosyasını birleştirmek üzere tasarlanmıştır. SIC formatı ise linking loader tarafından birleştirilebilir; bunun için D ve R kayıtları kullanılır. Bu, SIC/XE’nin harici sembol desteğiyle mümkün kılınmıştır. Yine de bu bağlama işlemi ELF linkine göre daha sınırlıdır (Yalnızca sembol adreslerini toplamak/çıkarmak şeklinde).
* **Boş Alanlar:** ELF’te .bss gibi başlatılmamış bölümler, dosyada yer kaplamaz sadece boyut bilgisi taşır. SIC object programında RESB/RESW ile oluşturulan boş alanlar T kayıtlarında yer almaz; bu kısım E kaydındaki program uzunluğuna dahil edilse de, loader o aralığı sıfırlarla doldurabilir ya da atlayabilir (genellikle atlar, çünkü loader sadece belirtilen adreslerdeki belirtilen kodları yazar). Stack alanı gibi alanlar da programın parçası olarak object code’da temsil edilmez.
* **Ölçeklenebilirlik:** ELF, büyük projelerde binlerce sembol ve bölümle çalışabilir. SIC formatı eğitim amaçlı olduğundan, bir programda çok fazla parça olduğunda kayıt yönetimi karmaşıklaşır. Örneğin 10’dan fazla harici sembol varsa D/R kayıtlarının satır uzunluğu artar vb. ELF’te böyle sınırlar pratikte yoktur.

Kısaca, **MSP430 nesne kodu formatı endüstriyel standart bir formattır (ELF)** ve derleyici araç zincirinde otomatik işlenir. **SIC/SIC-XE nesne kodu formatı ise metin tabanlı, eğitimsel bir formattır** ve basit bir loader tarafından doğrudan okunup yüklenebilecek şekilde tasarlanmıştır. Kullanıcı için MSP430 nesne dosyasının detayları önemli olmazken, SIC object programını anlamak eğitim sürecinin bir parçası olabilir.

**7. MSP430’da Yeni Komutlar ve Direktifler (SIC/SIC-XE’e Göre)**

MSP430, gerçek bir mikrodenetleyici mimarisi olduğu için, basitleştirilmiş SIC/SIC-XE’ye kıyasla çok daha geniş bir komut setine ve assembler direktiflerine sahiptir. Aşağıda, **SIC/SIC-XE’de bulunmayan veya onlardan çok daha gelişmiş olan MSP430 komut ve direktiflerinden bazıları** listelenmiştir:

* **Ek Adresleme Modları ve İmmediate Kullanımı:** MSP430 tüm komutlarında immediate (#), register direkt, register dolaylı (@Rn), autoincrement (@Rn+), indeksli (X(Rn) şeklinde offsetli) ve sembolik (PC relative) adresleme modlarını destekler. Örneğin bir değeri hafızadan yüklemek için MOV 2(R5), R6 (R5 + 2 offsetindeki bellekten R6’ya) gibi esnek biçimler mevcuttur. SIC orijinal modelde yalnızca direkt bellek adreslemesi ve indeksleme (,X kullanarak X register ile) sunuyordu; SIC/XE bunlara PC-relative ve Base-relative modlarını ekledi. Autoincrement (örn. @R14+) ve register dolaylı adresleme SIC/XE’de yoktur, MSP430’da özellikle **stack pointer R1** üzerinde @SP+ şeklinde kullanımı yaygındır. Bu sayede MSP430, döngülerde bellek üzerinden kolayca ilerleyebilir (adres pointer register’ı otomatik artar). SIC ortamında benzer bir işlemi yapmak için her seferinde adresi programcı artırmak zorundaydı (TIX vs.). Immediate operanda gelince, MSP430’da immediate her yerde kullanılabilirken SIC’de sadece XE modelinde ve sınırlı uzunlukta kullanılabildi.
* **Daha Fazla Genel Maksatlı Register:** MSP430’da R4–R15 arası **12 adet çalışma registeri** vardır (R0-PC, R1-SP, R2-SR/CG, R3-CG sabit jeneratörü olarak özel kullanımlar). SIC standart modelde sadece **A (Accumulator)** ve **X (Index)** registerlerini genel maksatlı kullanıma açmıştı (L register sadece link/dönüş için). SIC/XE ise A, X’e ek olarak B, S, T adında 3 genel amaçlı register daha getirdi ve ayrıca F (Floating-point accumulator) getirdi. Ancak SIC/XE register-to-register işlemler bile sınırlıdır (format 2 komutları). MSP430’da tüm aritmetik ve mantıksal işlemler registerler arasında veya register-bellek arasında yapılabilir. Örneğin, ADD R5, R6 (R6 = R6 + R5) mümkündür. SIC’de benzer bir işlem ya bellek üzerinden (ADD m, A = A + mem) ya da XE’nin ADDR komutuyla (iki register toplayıp birine koyan özel komut) yapılabiliyordu[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=MNEMONIC%20OPERAND%20OPCODE%20EXPLANATION%20ADDR,R1). MSP430, esnek komut yapısıyla registerleri istediği gibi kaynak/hedef kullanmaya izin verirken SIC/SIC-XE’de kısıtlı özel komutlar vardı (örn. CLEAR R1 gibi sabit opkodlar)[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=MNEMONIC%20OPERAND%20OPCODE%20EXPLANATION%20ADDR,R1)[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=CLEAR%20R1%2004%20R1%20%3D,left%20shifts%20R1%20n%20times).
* **Yığın (Stack) İşlemleri ve Altprogram Çağrıları:** MSP430, donanım stack pointer’ı (R1) ve stack bazlı altprogram çağrı mekanizmasına sahiptir. **PUSH** ve **POP** talimatları sayesinde herhangi bir register veya immediati stack’e itip çekmek mümkündür. Örneğin PUSH R12 register R12’nin değerini yığına atar. **CALL** komutu, bir altprogram adresine (register ya da bellek olabilir) dallanırken PC’yi stack’e push eder (yani fonksiyon çağrısı)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=CALL%20Subroutine%20call%3B%20push%20PC,0%200%200%200%200). **RET** (return) bir altprogram dönüş komutudur, stack’ten PC değerini pop eder (assembler’da RET emule bir komut olup MOV @SP+, PC şeklinde gerçekleştirilir). SIC mimarisinde donanım stack yoktur: Altprogram çağrıları için **JSUB**, dönüş için **RSUB** kullanılır ve tek bir dönüş adresi L register’ında tutulur. Bu, iç içe (nested) altprogram çağrılarını yazılım seviyesinde yönetmeyi zorlaştırırdı (her seferinde L’nin saklanması gerekirdi)[gear.kku.ac.th](https://gear.kku.ac.th/~watis/courses/188231/sp1-1.pdf#:~:text=,one%20layer%20of%20function%20call). SIC/XE’de de stack pointer kavramı yoktur (SIC/XE ek register B, S, T getirse de S genellikle stack pointer olarak kullanılabilir diye düşünülmüş ama donanım desteği yok). Dolayısıyla MSP430’un **push/pop** talimatları ve otomatik stack kullanımı (interruptlar dahil) SIC’e göre önemli bir yeniliktir.
* **Koşullu Dallanma (Branch) Komutları:** SIC’te koşullu dallanma *Condition Code (CC)* ile ve yalnızca üç şekilde yapılır: **JLT, JEQ, JGT** (A <0, =0, >0 durumu için) ve bunları ayarlayan **COMP** (karşılaştır) talimatı ile. Bu kısıtlı yapıda farklı koşullar kontrol etmek için bazen ek işlemler gerekiyordu. MSP430 ise işlemci durum registerindeki bayraklara göre bir dizi dal talimatı sunar: **JZ/JEQ (sıfır ise zıpla)**, **JNZ/JNE (sıfır değil ise)**, **JC (carry = 1 ise)**, **JNC (carry = 0 ise)**, **JN (negatif ise)**, **JGE (signed >=)**, **JL (signed <)**, **JMP (koşulsuz)** gibi geniş bir yelpaze vardır[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=JNE%2FJNZ%20Jump%20if%20not%20equal%2Fzero,operand%20arithmetic%201)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=JN%20Jump%20if%20negative%200,operand%20arithmetic%201). Örneğin, bir sayının çift olup olmadığını kontrol etmek için MSP430’da önce bir bit test edip sonra JNZ diyerek sıfır değilse dalmak kolaydır. SIC’te benzer bir işlem için belki değeri 2’ye bölüp kalanına bakmak gerekebilirdi veya farklı mantıklar kurmak zorunlu olabilirdi. Koşullu dallanmaların ofsetleri MSP430’da 10 bitlik PC-relative verilir (±512 word aralığı)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=JNE%2FJNZ%20Jump%20if%20not%20equal%2Fzero,operand%20arithmetic%201). SIC/XE’de de PC-relative dallanma 12 bit ofset ile vardı. Ancak MSP430’un koşul seti daha zengin ve doğrudandır (ayrı bir COMP gerekmeden, örneğin CMP de bir talimattır ama bazı basit kontroller BIT ile yapılabilir). Ayrıca MSP430’da **döngü** işlemlerini kolaylaştıran DJNZ benzeri bir komut olmamasına rağmen, DEC Rn; JNZ label gibi kombinasyonlarla istenen her koşul kodlanabilir. SIC/XE’de döngü için özel TIX ve TIXR komutları mevcuttu ama bunlar sadece X register ile sınırlıydı[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=STT%20M%2084%20M%20%3D,1%3B%20compares%20X%20and%20R1); MSP430’da istenilen register ile döngü sayacı yapılabilir.
* **Bit Manipülasyon ve Mantıksal Komutlar:** MSP430 mimarisinde donanım kayıtları ve bellek üzerindeki bitleri manipüle etmeyi kolaylaştıran özel komutlar vardır. **BIS (Bit Set)** ve **BIC (Bit Clear)** komutları, hedef operanda belirtilen bitleri 1 yapar veya temizler. Özellikle durum registerini yönetmek veya I/O portlarının bitlerini değiştirmek için kullanılırlar. Örneğin BIS #0x08, SR komutu durum registerindeki GIE bitini 1 yaparak kesmeleri etkinleştirir (assembler bunu EINT emüle komutu olarak sağlar)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=SXT%20Sign%20extend%20byte%20to,0%200%200%200%200)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=As%20destination%20ADD%20Add%20source,1%201%200%20source%20A). Benzer şekilde BIC #0x08, SR kesmeleri kapatır (DINT emülasyonu)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=W%20As%20source%20SXT%20Sign,0%201%200%20As%20source)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=As%20destination%20ADD%20Add%20source,1%201%200%20source%20A). Bu tarz tek bit ayarlama/silme işlemleri SIC/SIC-XE’de doğrudan yoktur. Programcı, örneğin bir bayrak bitini temizlemek için bir AND maskesi yapacaksa, bunu ACC içinde gerçekleştirmeli ve sonra geri yazmalıdır. XE’de eklenen CLEAR Rn komutu sadece tüm registre 0 yazar, tek bit değil[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=MNEMONIC%20OPERAND%20OPCODE%20EXPLANATION%20ADDR,R1)[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=CLEAR%20R1%2004%20R1%20%3D,M%2068%20B%20%3D%20M). MSP430 ayrıca **BIT (bit test)** komutuna sahiptir; bu aslında src AND dest işlemi yapıp sonucu atmaz, sadece bayrakları ayarlar (yani bir AND testi)【44†Table 3-11】. Bu da SIC’te olmayan ama MSP430 programcılarının sık kullandığı bir kolaylıktır (bir değerin belli bitleri sıfır mı değil mi kontrolü). Mantıksal işlemlerde MSP430, **AND, OR, XOR** gibi tam komutlar sunar (OR için MSP430’da genelde **BIS** kullanılır; XOR ayrı komuttur)【44†Table 3-11】. SIC/SIC-XE’de temel bir **AND** komutu yoktur (XE belki COMPR ile register karşılaştırması, veya test için TIXR gibi kısıtlı şeyler sunar ama genel amaçlı XOR, OR opcode’u listede yok). Bu nedenle MSP430’un mantıksal komut seti (AND, XOR, BIS, BIC vs.) SIC’e göre çok daha zengindir ve donanım kontrolünde işlevseldir.
* **8-bit ve 16-bit Operasyon Ayrımı:** MSP430 komutları, .B (byte) ve .W (word) olmak üzere iki boyutta çalışabilir. Örneğin MOV.B bir bayt taşır, MOV.W 16 bitlik bir word taşır. Birçok aritmetik ve mantıksal komutta da isteğe bağlı olarak byte modu kullanılabilir (örneğin ADD.B 8 bitlik toplama yapar, taşı bayrağını etkiler)【44†Table 3-11】. Bu özellik, 8-bit verilerle işlem yapmayı kolaylaştırır ve bellek/veri tasarrufu sağlar. SIC makinesi 24-bit kelime boyutuna sahipti; karakter işlemleri için sadece **LDCH/STCH** komutları mevcuttu (A registerinin düşük baytını hafıza ile transfer ediyordu). SIC bir bayt üzerinde aritmetik işlem yapamaz, en fazla bir baytı yükleyip ACC’nin geri kalanı temiz kalarak işlenmiş olur. MSP430 ise doğrudan 8 bitlik ALU işlemlerini destekler. Bu, SIC’e göre önemli bir gelişimdir. Örneğin MSP430’da CMP.B # 'a', R5 ile bir karakter karşılaştırması tek komutta yapılabilir; SIC’te önce ACC’ye 3 byte’lık kelime olarak yüklemek, maskalamak vs. gerekebilirdi.
* **Özel Tek Operandlı Komutlar:** MSP430, bazı faydalı tek operandlı komutlara sahiptir. Örneğin **SWPB** (Swap Bytes) komutu, bir registerin lower-high byte’larını kendi içinde değiştirir (16 bitlik registerin üst 8 biti ile alt 8 bitini takas eder). **SXT** komutu, bir registerdeki alt 8 bit (byte) değerini işaret bitini kopyalayarak 16 bite genişletir (sign-extend)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=SWPB%20Swap%20bytes%200%200,PC%20and%20move%20source%20to). Bu, özellikle bir byte’ı işaretli olarak word’e çevirmek için kullanışlıdır. **RRA** ve **RRC** komutları, aritmetik ve döngülü sağ kaydırma yapar (LSB -> Carry, ya da Carry üzerinden)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=As%20source%20RRC%20Rotate%20right,1%201%200%20As%20source). Bu şekilde bölme veya bit kaydırma işlemleri kolaylaşır. **RLAM/ RRUM** (rotate left arithmetic, rotate right through carry) gibi sol döndürme yok fakat ADDC/SUBC ile carry üzerinden işlem yaparak sol kaydırma da yapılabilir. SIC/XE’de benzer şekilde **SHIFTL/SHIFTR** komutları eklenmiştir ancak bunlar sadece register çifti üzerinden belli sayıda bit kaydırabilir ve sadece XE registerlarına özgüdür[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=MULR%20R1%2C%20R2%2098%20R2,1%3B%20compares%20X%20and%20R1). MSP430’da herhangi bir registeri 1 bit kaydırmak bir komuttur, çoklu bit kaydırmak döngü ile yapılabilir; XE’de tek komutta n bit kaydırma vardı fakat çok esnek değildi. **NOP** (No Operation) MSP430’da vardır (assembler bunu MOV R3,R3 şeklinde uygular), SIC’te de etkisiz bir işlem yoktu ama format 1 olarak FIX vs. gibi bazı komutlar etkisiz gibi kullanılabilirdi. Yine de MSP430’da NOP açık bir mnemonik olarak bulunur.
* **Donanım Kontrol ve Durum Komutları:** MSP430 işlemcisinde kesme yönetimi ve durum bayrakları için özel talimatlar bulunur. Aslında bunlar core instruction set’te BIC/BIS işlemlerinin durum registerine uygulamalarıdır, ancak assembler **emulated instruction** olarak kolay isimler sağlar. **EINT (Enable Interrupts)** assembler düzeyinde bir komut gibi yazılabilir; bu, aslında BIS #8, SR (durum registerinin GIE bitini 1 yap) şeklinde makine kodudur【52†Table 1b】. **DINT** benzer şekilde BIC #8, SR ile kesmeleri maskeler. **SETC, CLRC** komutları carry bayrağını 1 yapmak veya temizlemek için assembler tarafından sağlanır (gerçekte SR’ın bit0’sine BIS/BIC uygular)【52†Table 1b】. **SETN, SETZ, CLRZ** gibi durum bayrağı ayarlama komutları da mevcuttur (N ve Z bitleri için). SIC mimarisinde programcı CC (condition code) bitselerini doğrudan ayarlayamaz; sadece COMP veya aritmetik işlemler sonucu CC belirlenir. XE’de de CC’yi direkt manipüle eden komut yoktur. Bu bakımdan MSP430, düşük seviyede bayrak kontrolü de sunar (bu, gömülü sistemlerde önemli olabilir, örneğin bir durum bayrağını yapay olarak set edip bir dallanmayı tetiklemek gibi).
* **Donanım Çarpma Desteği:** Orijinal MSP430 core set’inde bir donanım çarpma talimatı yoktur, ancak çoğu MSP430 çipi ayrı bir donanım multiplier modülüne sahiptir ve bu modül, özel fonksiyon registerleri üzerinden kullanılabilir. Bunun assembler seviyesine yansıması, bu registerlere yazma okuma şeklindedir (yani belirli adreslere .word yazıp sonucu beklemek). Ancak MSP430X genişletmeleriyle (MSP430X ISA), muhtemelen **MUL, MULS** gibi çarpma komutları eklenmiştir. SIC orijinal modelinde **MUL** komutu hiç yoktu, sadece ADD ve SUB vardı. SIC/XE’de floating-point birimi hariç, fix-noktada **MULR** (register \* register) ve **MULF** (float multiply) gibi komutlar eklendi[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=DIVR%20R1%2C%20R2%209C%20R2,M%2078%20M%20%3D%20B). Yine de MSP430’un sabit nokta çarpma modülünü kullanımı SIC’e göre çok daha hızlı ve basittir (16x16->32 bit çarpımı birkaç clock sürer). Benzer şekilde **DIV** komutu MSP430’da yok (modül olarak yoktur) ancak SIC/XE’de DIVR (register bölme) ve DIVF (float) mevcuttu[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=CLEAR%20R1%2004%20R1%20%3D,left%20shifts%20R1%20n%20times). MSP430 için derleyici yazılım olarak bölmeyi halleder. Bu bir farklılık olsa da, temel olarak MSP430’un mikrodenetleyici olması nedeniyle donanım çarpma modülü bulundurması bir avantajdır.
* **Emülasyon (Pseudo) Komutlar ve Makrolar:** MSP430 assembler, gerçek işlemci komutları dışında birçok *pseudo-op* sağlar. Bunlar assembler tarafından bir veya birkaç gerçek komuta çevrilen kolaylık komutlarıdır:
  + **CLR Rn:** Bir registerı sıfırlar (aslında MOV #0, Rn ya da eşdeğeri SUB Rn, Rn). XE’de benzer CLEAR R1 gibi bir komut opkod olarak vardı[geeksforgeeks.org](https://www.geeksforgeeks.org/instruction-set-used-in-sic-xe/#:~:text=MNEMONIC%20OPERAND%20OPCODE%20EXPLANATION%20ADDR,R1), MSP430’da bu bir emülasyondur. Aynı şekilde **CLR.b X** bir bellek byte’ını sıfırlamak için MOV.b #0, X şeklinde yapılır.
  + **INC/DEC:** Artırma/azaltma için kısa yazımlar. Örneğin INC R5 MSP430 assembler’da ADD #1, R5 olarak genişletilir. XE’de de ADDR ile iki register arasında toplama yapabilirdi ama tek register increment yoktu (TIX bir nevi X’i artırır).
  + **ADC/SBC:** Aslında MSP430 core set’inde *ADD with carry* (ADDC) ve *SUB with borrow* (SUBC) komutları vardır, bunlar bayrak dahil toplama/çıkarma yapar. SIC’te taşıyıcıyı ayrı kontrol etme yoktu (floating ops hariç). Emulate olarak bazen ADC.x pseudo kullanılabiliyor.
  + **BR (Branch) ve NOP:** BR label MSP430’da aslında koşulsuz atlama JMP label demektir; assembler eşanlamlı olarak kabul eder. NOP hiçbir şey yapma komutu, assembler bunu MOV R3,R3 (constant 0 registerini kendine taşı, etki yok) şeklinde uygular【52†Table 1b】. SIC’te NOP karşılığı genelde olmuyordu (belki TD ile cihaz olmayan bir üniteyi test edip CC’yi etkilemeden devam edilebilir gibi hackler vardı).
  + **RETI:** MSP430’da kesme servis rutinden dönüş komutu RETI’dir, bu bir core instruction’dır (SR ve PC’yi stack’ten çekip geri döner)[phas.ubc.ca](https://phas.ubc.ca/~michal/phys319/MSP430Reference-RyansEdit.pdf#:~:text=CALL%20Subroutine%20call%3B%20push%20PC,0%200%200%200%200). SIC mimarisi kesmeleri desteklemediği için böyle bir komut yoktu.
  + **Macros:** MSP430 assembler, SIC assembler’ın temelinde olmayan güçlü bir makro sistemine sahiptir. .macro ... .endm direktifleri ile parametre alabilen kod blokları tanımlanabilir. Bu, tekrarlayan kodlar için kullanılır. SIC assembler’da makro dili kavramı kitapta ayrı bölüm olarak vardır ama makro işlemleri assembler’ın bir üst katmanı gibi görülür. MSP430’da ise TI assembler içinde makro direktifleri (``.macro, .exitm, .mend` vb) kullanarak kod üretimini otomatikleştirmek mümkündür. Örneğin, bir porttan belirli bitleri okuma/yazma işlemi bir makro ile tanımlanabilir ve program içinde çağrılabilir. Bu, SIC’in iki geçişli basit assembler’ında olmayan ileri bir özelliktir.
* **Assembler Direktif Zenginliği:** MSP430 assembler, SIC/SIC-XE’den çok daha fazla sayıda assembler direktifi sunar. Bunların bazıları:
  + **Veri Tanımlama:** SIC’te BYTE ve WORD direktifleri vardı. MSP430 assembler’da bunların karşılığı **.byte, .word, .long, .float, .double, .string, .uchar, .ushort** gibi pek çok çeşitte direktif vardır (farklı boy ve tiplerde veriyi hafızaya yerleştirmek için)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=,cstring%20directives)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=The%20,For%20more%20information%2C%20see%20the). Örneğin .string "HELLO" ardışık karakter kodlarını bellek yerleştirir, .long 0x12345678 32 bitlik sabit atar vb. SIC’te 32 bitlik bir sayı koymak ancak iki WORD yazarak mümkündü.
  + **Hizalama (Alignment):** MSP430 assembler’da **.align** direktifiyle veya .even gibi kısayollarla bellek adresini belirli bir sınırın sonraki değerine ayarlamak mümkündür. Örneğin .align 2 adresi bir üst çift adrese yuvarlar. SIC assembler’da belirli bir alignment direktifi yoktu; programcı gerekirse bir baytlık doldurma yaparak (X’00’ gibi) kendisi align edebilirdi.
  + **Space Reservation:** SIC’te RESB ve RESW direktifleri başlatılmamış bellek alanı ayırırdı. MSP430’da karşılığı **.space** (belirtilen bayt kadar 0 doldur) ve **.bes** (belirtilen bayt kadar ayır, son adres noktasına label yerleştir) gibi direktifler vardır[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=27%20000a%200035%20.word%20NSYMS)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=Description). Örneğin .space 100 mevcut sekmede 100 bayt ileri gider (0 yazarak) – bu RESB 100’e denktir.
  + **Section Yönetimi:** Yukarıda anlatılan .text, .data, .bss, .sect, .usect, .global, .def, .ref gibi direktifler SIC’te ya hiç yoktu ya da farklı isimlerle sınırlıydı. SIC/XE’de benzer işi yapan CSECT (yeni kontrol bölümü), EXTDEF/EXTREF (global sembol tanım ve referansı) direktifleri vardı. MSP430 assembler’da .global/.def/.ref ile semboller global ilan edilir (SIC’te EXTDEF/EXTREF ile benzer)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=The%20,symbol%20in%20the%20symbol%20table)[downloads.ti.com](https://downloads.ti.com/docs/esd/SLAU131K/Content/SLAU131K_HTML/assembler_directives.html#:~:text=A%20global%20symbol%20is%20defined,module%20actually%20uses%20the%20symbol). Fakat .bss, .usect gibi bölümsel ayırımlar SIC’te program blokları ile sınırlıydı.
  + **Koşullu Derleme:** MSP430 assembler, C preprocessor benzeri koşullu derleme direktiflerine sahiptir: **.if, .else, .elseif, .endif**. Bu sayede assembly kodu içinde belli koşullara göre bazı kısımlar derlenip derlenmeyebilir. Örneğin:

.if MYCONST > 5

MOV #1, R12

.else

MOV #0, R12

.endif

* + Bu kabiliyet, SIC assembler’da yoktu. SIC’te assembly düzeyinde koşullu derlemeyi yapmak ancak makro işlemcisi ile veya harici bir araçla mümkündü. MSP430 assembler ayrıca **.loop/.endloop** yapısına da izin verir (belirtilen sayıda kod bloğunu tekrarlama)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=match%20at%20L4717%20,loop%20topic)[ti.com](https://www.ti.com/lit/ug/slau131r/slau131r.pdf#:~:text=,the%20assembler%20to%20repeatedly%20assemble). Bu da SIC’te bulunmayan bir özelliktir.
  + **Diğer Direktifler:** MSP430 assembler’da bulunan ve SIC’te olmayan daha birçok direktif vardır: **.include** (başka bir dosyayı dahil etme), **.org** (belli bir adres ofsetine atlama – gerçi TI dokümanında .org geçmiyor, .space ile yapılabilir), **.symval, .length** gibi sembol bilgisi alan, **.option** ile assembler ayarları değiştiren direktifler, **.label** gibi özel kullanım, **.struct/.endstruct** ile veri yapıları tanımlama (yapı üyelerini offset hesaplayarak) vb. Bu zenginlik SIC assembler’da bulunmaz çünkü o minimal bir assembler idi.

**Özetle:** MSP430’un komut seti ve assembler direktifleri, SIC ve SIC/XE’e göre son derece gelişmiştir. MSP430’da **27 çekirdek komut + 24 emüle komut** olmak üzere 50’yi aşkın mnemonik bulunmaktadır (toplama, çıkarma, mantıksal, döndürme, push/pop, çağrı, dönüş, bit set/clear, koşullu dallanma vb.) – SIC’te ise yaklaşık 25 kadar temel mnemonik vardı[ninova.itu.edu.tr](https://ninova.itu.edu.tr/en/courses/faculty-of-computer-and-informatics/4142/blg-351/ekkaynaklar?g867880#:~:text=%5BPDF%5D%203.4%20Instruction%20Set%20,instructions%20that%20have%20unique). MSP430 assembler, modern bir araç olarak, programcının işini kolaylaştıran birçok özelliğe (makrolar, şartlı derleme, çeşitli veri tipleri, bölümler) sahiptir. SIC assembler ise öğretici bir basitlikte olduğundan bu tür “yeni” özelliklerin çoğu yoktur. SIC/XE mimarisi elbette SIC’e göre bir adım ileri birkaç komut ve adresleme eklemiştir (ör. CLEAR, TIXR, SHIFTL gibi komutlar, float işlemler, base register, format4 adresleme vs.), ancak bunlar bile MSP430’un sunduklarıyla kıyaslandığında sınırlı kalır.

Aşağıdaki tablo, **bazı önemli yeni komutların** karşılaştırmasını yapar:

| **Komut/Özellik** | **MSP430 (destekliyor)** | **SIC/SIC-XE (durumu)** |
| --- | --- | --- |
| Immediate operand (doğrudan sabit) | Evet – #value her yerde (16-bit). | SIC: Hayır, SIC/XE: Evet (12/20 bit sınır). |
| Register-to-register işlemler | Evet – tüm aritmetik/mantıksal MOV/ADD/AND vs. | SIC: Sadece A ile bellek; XE: Sınırlı (ADDR,SUBR vs.). |
| Stack ve Call/Ret | Evet – R1 stack pointer, PUSH/POP, CALL, RETI var. | SIC: Donanım stack yok; JSUB/RSUB ile tek seviye çağrı. |
| Koşullu dallanma çeşitliliği | Çok – JZ, JNZ, JC, JNC, JN, JGE, JL, JMP vb.. | Az – Sadece COMP sonrası JLT, JEQ, JGT (CC’lere göre). |
| Bit set/clear (bayrak veya I/O) | Evet – BIS/BIC (örn. BIC #1,SR carry bayrağını temizler)【52†Table 1b】. | Yok – Bit manipülasyonu ancak AND/OR maskesiyle manuel. |
| Mantıksal komutlar | Evet – AND, XOR, NOT (emnule: INVX), BIT (test) mevcut. | Sınırlı – Mantıksal doğrudan komut yok (COMP/TEST kısıtlı). |
| Byte ve Word ayrımı | Evet – bir çok komutta .B ve .W seçenekleri (8-bit/16-bit). | Kısmen – SIC: 24-bit sabit, LDCH/STCH ile 8-bit transfer. XE: 8-bit işlem desteği yok. |
| Özel komutlar (SWPB, SXT, RRA, RRC) | Evet – Byte takas, işaret genişletme, kaydırma komutları var. | XE: SHIFTL/SHIFTR (yalnız reg), CLEAR var; SIC: yok. |
| Kesme yönetim komutları | Evet – EINT, DINT, RETI (kesme dönüşü) vb. bulunur【52†Table 1b】. | Yok – Kesme mekanizması tanımlı değil (SIC). |
| Çarpma/Bölme talimatı | Donanım multiplier modül var (yalın çekirdekte MUL yok, MSP430X’de olabilir). | SIC: MUL yok, XE: MULR, DIVR (register arası) eklendi. |
| Makro ve şartlı derleme | Evet – .macro/.endm, .if/.else vb. zengin dil desteği. | Yok – (SIC makro işlemleri ayrı, assembler’ın parçası değil). |
| Assembler direktifleri | Çok zengin – .text, .data, .align, .space, .global, .include, .struct, vb. | Sınırlı – START, END, BYTE/WORD, RESB/RESW, EQU, ORG, EXTDEF/EXTREF, CSECT vb. temel düzey. |

Yukarıdaki karşılaştırmalardan görüldüğü üzere, **MSP430 assembler’ı** gerçek bir ticari işlemci için olduğundan, **SIC/SIC-XE assembler’ına göre çok daha güçlü ve esnektir**. SIC/SIC-XE tasarımı, eğitim amaçlı basitliği hedeflerken, MSP430 tüm modern özellikleri barındırır. Bu durum, komut setinden bellek yönetimine, nesne kod formatından direktiflere kadar her başlıkta belirgin şekilde ortaya çıkmaktadır.