Zaawansowane funkcje Linkera

Sekcja 13

Zaawansowane funkcje linkera

- **Linker "rozwiazuje" adresowanie absolutne i tworzy plik wykonywalny (.DXE)**

- **Advanced Linker Support**

  - **dzielenie pamieci**
    - wszystkie 21161's w klusterze uzyskac dostep do zalinkowanego symbola "dzilonej" pamieci zewnetrznej

  - **softwarowe overlays**
    - subroutine i dane moga '*zyc' w zewnetrznej pamieci* byc przeslane do pamieci wewnetrznej by 'run' bardziej efektywnie

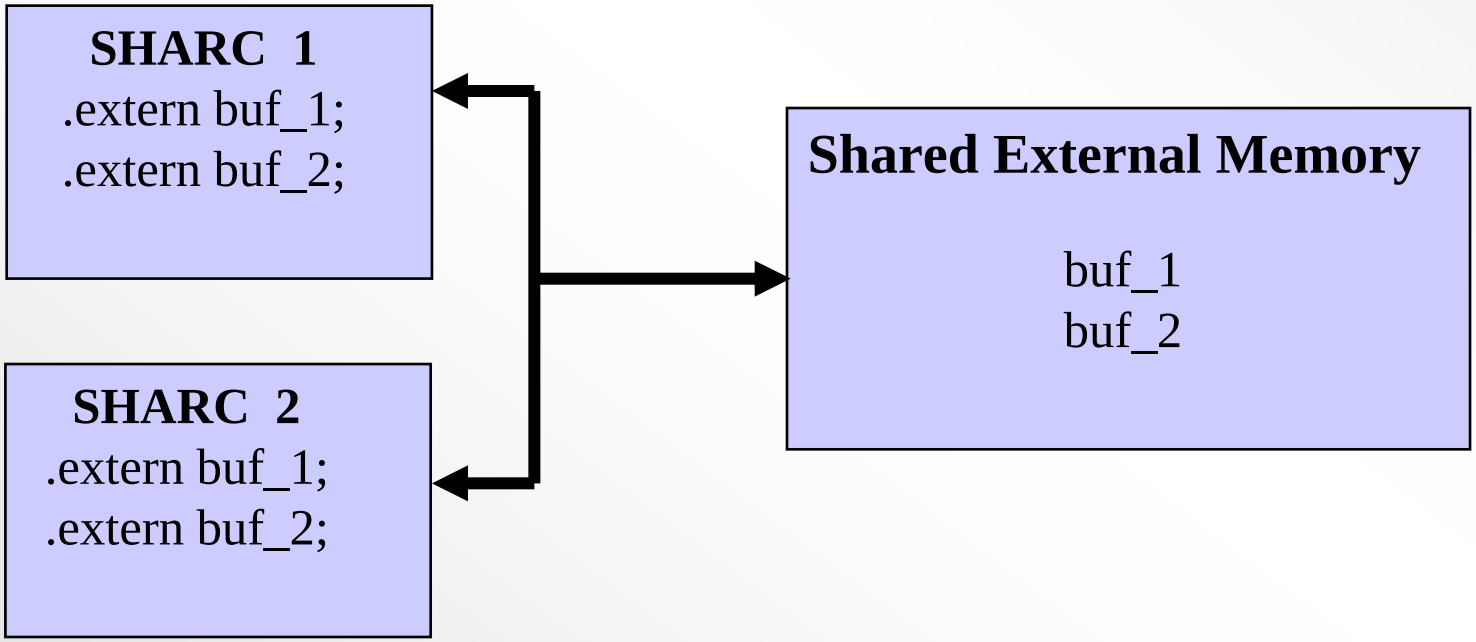
Shared Memory Support

- **Shared Memory**
  - SHARC cluster sharing an external block of memory
  - Data or subroutines can be accessed

- **Software support for shared memory systems**
  - The same buffer in external memory is accessible by all SHARCs via a common symbol.
  - Linker creates a “.SM” file that contains the resolved addresses for all the shared memory symbols in the project
  - The programmer can “ link against” .sm file/files to resolve the absolute addresses of shared memory symbols

Przyklad dzielenia pamieci

- **dwa SHARC moga miec dostep do tego samego bufora danych w pamieci zewnetrznej**
- **linker bedzie rozwiazywac wszystkie symbole adresu dla obu plikow wykonywalnych SHARC. Linker korzysta z informacji z LDF, ktore zaimplementowal programista.**



Przyklad dzielenia pamieci

Shared.asm

```
.global       buf_1;
.global       buf_2;
.section/dm         sm_data;
      .var    buf_1[100];
      .var    buf_2[50];
```

Variables can be defined in a separate file or within a SHARC's source code. This example uses variables defined in a separate file.

SHARC2.asm

```
.extern   buf_1;
.extern   buf_2;
```

Or, If variables defined within SHARC2's source code

```
.global   buf_1;
.global   buf_2;
.section/dm          sm_data;
          .var       buf_1[100];
          .var       buf_2[50];
```

```
.section/pm              pm_code;
          r0=dm(buf_1);
          r1=dm(buf_2);
```

SHARC1.asm

```
.extern   buf_1;
.extern   buf_2;
.section/pm              pm_code;
          r0=dm(buf_1);
          r1=dm(buf_2);
```

Przyklad dzielenia pamieci LDF

```
SHARED_MEMORY          /* Global command */
{     OUTPUT(common.sm)
   SECTIONS
   {  SM_1
            {  INPUT_SECTIONS( shared.doj(sm_data) )  } >Ext_dat
   }
} /*end shared-memory*/

PROCESSOR px1
{
   LINK_AGAINST( common.sm )
   OUTPUT( px1_fft.dxe )
   SECTIONS
   { include ”Sections1.h” }
}   /* end px1  */

PROCESSOR px2
{
   LINK_AGAINST( common.sm )
   OUTPUT( px2_fft.dxe )
   SECTIONS
   { include ”Sections2.h” }
}   /* end px2  */
```

Object Section name (defined in source file)

Assembled source

Memory-segment name (defined in LDF)

Overlay softwarowy

Softwerowy Overlay

Co jesli moj kod jest za duzy do pamieci wewnetrznej?

- **bezposrednie wykonanie w pamieci zewnetrznej**
  - **wykonalnosc pakietow jest wolniejsza**
  - **zajmuje cluster bus**

- **Overlay (nalozenie w pamieci)**
  - **kod/dana jest skladowana (“lives”) w pamieci zewnetrznej i, jesli potrzeba, jest transferowana (DMA) do pamieci wewnetrznej.**
  - **"Sekwentyzator" programu nie wie o overlay**
    - wszystkie overlay musza byc zarzadzane przez oprogramowanie
  - **kazde overlay jest przydzielone do “run space” i “live space” poprzez LDF**
  - **wielokrotne Run spaces sa mozliwe**
  - **Run space moze byc dzielony przez wiele overlay (poprzez tylko jedno overlay moze byc w danym momencie umieszczone w Run space)**

Podstawowe pytania

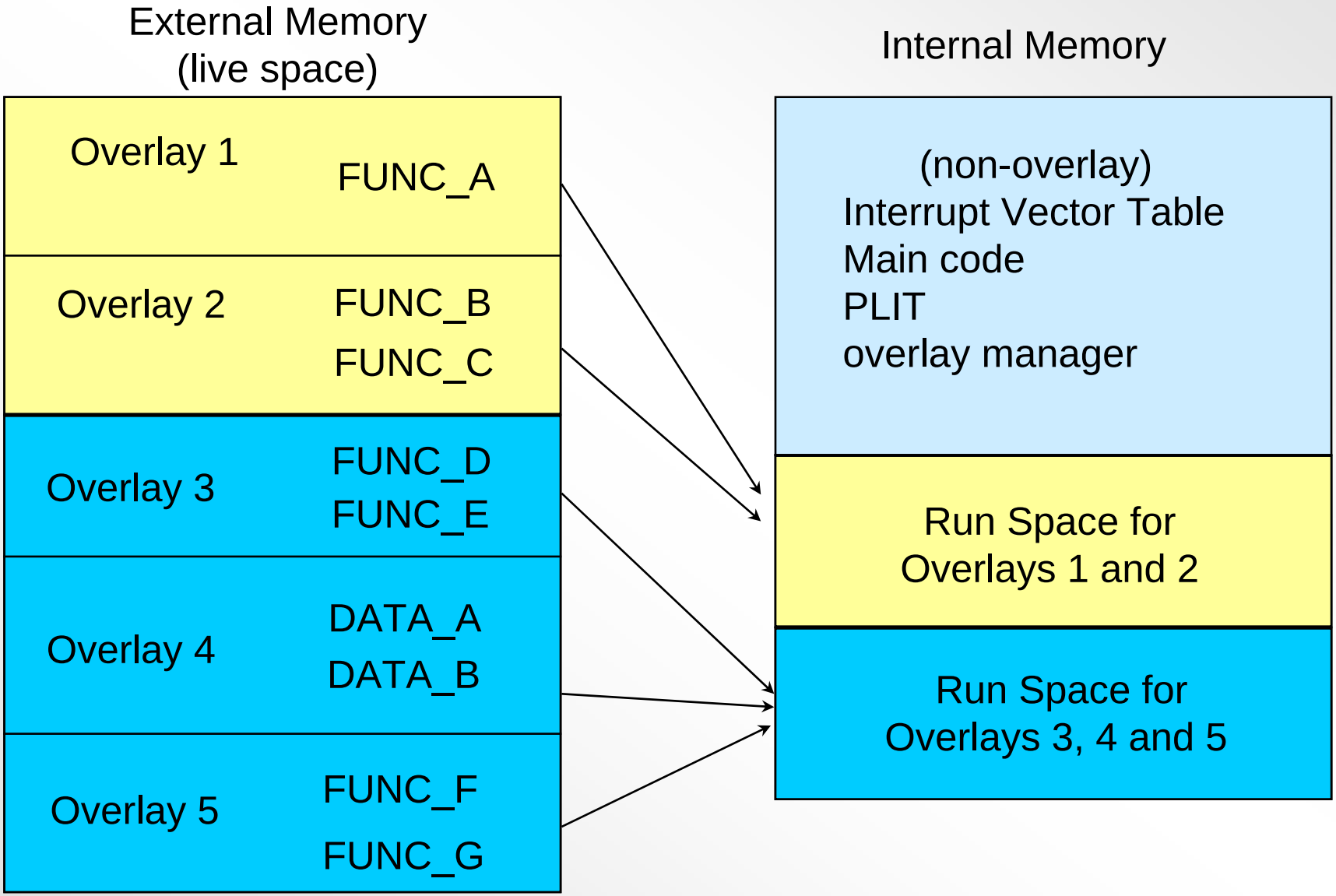
- **gdzie w pamieci zewnetrznej "zyje" overlay?**
  - LDF mowi linkerowi zakres adresu dla ***live space***
- **gdzie w wewnetrznej pamieci overlay sie rozpocznie?**
  - LDF mowi linkerowi zakres adresu dla ***run space***
- **jak dostac sie z live space do run space?**
  - maly "subroutine" nazywany ***overlay-manager**** ustawia ***DMA*** bazujac na informacji wygenerowanej przez linker
- **jak okreslic "subroutine"?**
  - Overlay-manager wykorzystuje informacje zwracane przez linker
- **jak otrzymac "subroutine", ktora chcemy uruchomic?**
  - Overlay-manager***** wykonuje skok uzywajac informacji zwracanej przez linker

* overlay manager jest kodem napisanym przez programujacego
W SW tools zalaczamy przyklad overlay managers

Linker Generated Overlay Support

- **wyszczegolnic overlay** *live space* **i** *run space* **(w LDF)**
- **wygenerowac stale dla** *overlay-manager*
  - Overlay begin and end live address
  - Overlay Ids
  - Overlay run size (bytes and words)
  - Overlay live size
- **Procedure Linkage Table (PLIT)**
  - prosta jump-table jest uzywana do konfigurowania *overlay-manager*
  - zawiera definiowany przez uzytkownika kod do wykonania przy kazdym dostepie do symbolu w overlay.
- **plik “.OVL” jest generowany dla kazdego overlay**

Overlay


External Memory
(live space)
Overlay 1
FUNC_A
Overlay 2
FUNC_B
FUNC_C
Overlay 3
FUNC_D
FUNC_E
Overlay 4
DATA_A
DATA_B
Overlay 5
FUNC_F
FUNC_G
Internal Memory
(non-overlay)
Interrupt Vector Table
Main code
PLIT
overlay manager
Run Space for
Overlays 1 and 2
Run Space for
Overlays 3, 4 and 5

Overlay - kontynuacja przykladu

```
MEMORY
{
  int_vect    { TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48) }
  pm_code  { TYPE(PM RAM) START(0x00040100) END(0x00040fff) WIDTH(48) }
  pm_code1  { TYPE(PM RAM) START(0x00041000) END(0x000414ff) WIDTH(48) }
  pm_code2  { TYPE(PM RAM) START(0x00041500) END(0x000419ff) WIDTH(48) }
  pm_data   { TYPE(PM RAM) START(0x00042a00) END(0x00043fff) WIDTH(32) }

  dm_data    { TYPE(DM RAM) START(0x00050000) END(0x00053fff) WIDTH(32) }

  ovl_code   { TYPE(DM RAM) START(0x00200000) END(0x002001ff) WIDTH(32) }
  ovl1_code { TYPE(DM RAM) START(0x00200200) END(0x002007ff) WIDTH(32) }

}
```

Umiejscowienie overlay w pamieci

```
Sections  {
    sec_1
    {
        INPUT_SECTIONS(main.doj(seg_pmco) ovl_mgr.doj(seg_pmco))

        OVERLAY_INPUT                  /* create an overlay file called OVLY_one.ovl  */
        { OVERLAY_OUTPUT(OVLY_one.ovl)
        INPUT_SECTIONS(  FUNC_A.doj(pm_code) )
        } >ovl_code

        OVERLAY_INPUT                  /* create an overlay file called OVLY_two.ovl  */
        { OVERLAY_OUTPUT(OVLY_two.ovl)
          INPUT_SECTIONS(  FUNC_B.doj(pm_code)  FUNC_C.doj(pm_code) )
        } >ovl_code

    }  >pm_code1

    sec_2 {INPUT_SECTIONS($OBJECTS(seg_dmda))}>mem_data
    // remaining input sections (PM data, etc.)
}
```

Memory segment where the overlays will live
(placed first-come, first-served basis)

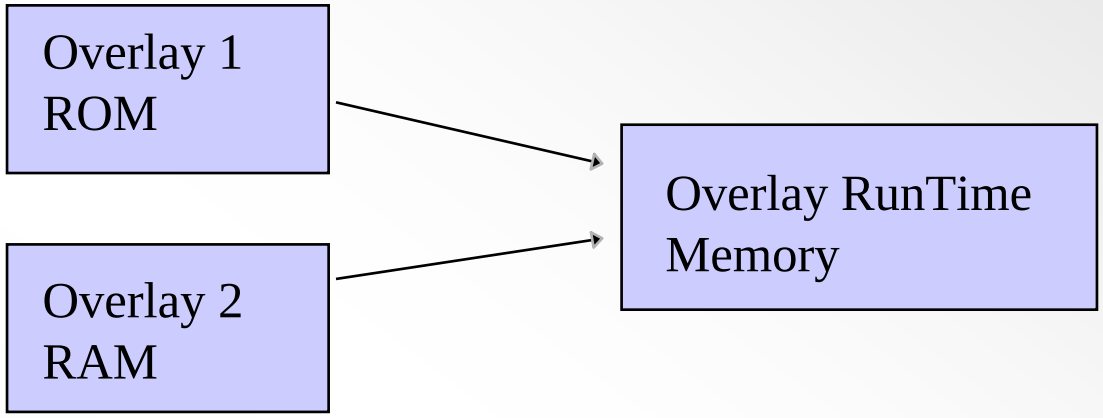
Memory segment where input-sections (including overlays) will run

Stale Linker Overlay

- **Linker automatycznie generuje stale dla kazdego symbolu overlay**

  _ov_startaddress_N
  _ov_endaddress_N
  _ov_size_N **(N is replaced by**
  _ov_word_run_size_N **the overlay ID)**
  _ov_word_live_size_N
  _ov_runtimestartaddress_N

- **Te stale sa zwykle zachowywane w pamieci i uzywanej przez nadzor overlay do ustawienia DMA dla overlay'a.**



- **Linker automatycznie generuje nastepujace symbole:**

_ov_startaddress_1	**_ov_startaddress_2**
_ov_endaddress_1	**_ov_endaddress_2**
_ov_size_1	**_ov_size_2**
_ov_word_live_size_1	**_ov_word_live_size_2**
_ov_word_run_size_1	**_ov_word_run_size_2**
_ov_runtimestartaddress_1	**_ov_runtimestartaddress_2**

Procedure Linkage Table (PLIT)

- **PLIT jest jump-table (podobnie do Interrupt-Vector-Table)**
  - dla kazdego overlay istnieje wejscie PLIT
  - kazdy PLIT ma identyczny kod
  - kod w PLIT jest definiowany przez uzytkownika w LDF
    (jakakolwiek instrukcja asemblerowa moze byc uzyta w PLIT)

- **Linker usuwa odwolania do funkcji overlay. Zastepuje bezposrednie odwolania, odwolaniami zwiazanymi z wejsciem PLIT**
  - "**`call FUNC1`**;" will be replaced with "**`call .plt_FUNC1;`**"

- **typowe entry PLITa:**
  - **okresla** *Overlay-ID*
  - **okresla** *Function's ddress*
  - **wywoluje** *overlay-manager.*

Note: The PLIT must be place in internal (non-overlay) memory

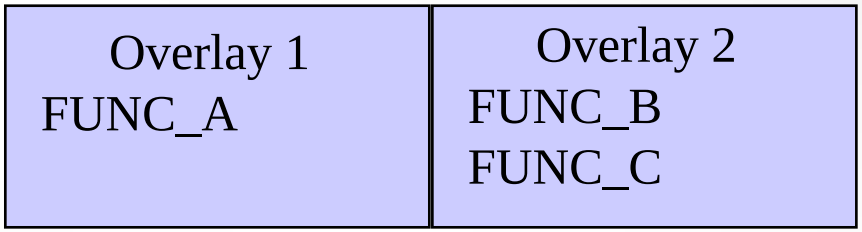
Komendy PLIT definiowane przez USERa

- **uzytkownik pisze instrukcje dla PLIT w LDF**
  - **kod PLIT okreslony w LDF jest duplikowany dla kazdego overlay (PLIT entry)**

```
PLIT        /* LDF Command */
 {
    R0 = PLIT_SYMBOL_OVERLAYID;   /* returns overlay ID in R0 */
    R1 = PLIT_SYMBOL_ADDRESS;     /* returns overlay function  */
                                  /* symbol address in R1     */
    JUMP _OverlayManager;
 }
```

Przyklad PLIT

- **Linker bedzie umieszczac zdefiniowany przez uzytkownika kod PLIT dla kazdego symbolu referencyjnego**




```
 /* Main code */                  /* PLIT */
Main:
call .plt_FUNC_A          .plt_FUNC_A:    r0=0x00001;
          .                                r1=0x41000;
          .                                jump OverlayManager;
          .               .plt_FUNC_B:     r0=0x00002;
call .plt_FUNC_C                           r1=0x41000;
call .plt_FUNC_B                           jump OverlayManager;
          .               .plt_FUNC_C:     r0=0x00002;
          .                                r1=0x41200;
                                           jump OverlayManager;
```

Umiejscowienie kodu PLIT

- **programista umieszcza PLIT w segmencie pamieci wewnetrznej**

```
SECTIONS
{
        .plit
        {
        } >seg_pmco
}
```

- **instruuje linker w przetwarzaniu kodu PLIT w segmencie pamieci seg_pmco.**

- **w celu uzyskania dalszych informacji o overlay i jego zarzadzaniu zobacz:**
  - **EE-66, EE-180, and EE-230**