Zephyr® Project
Developer Summit 2022

DYNAMIC MODULE LOADING

Chen Peng, Intel Corporation

peng1.chen@intel.com

Agenda

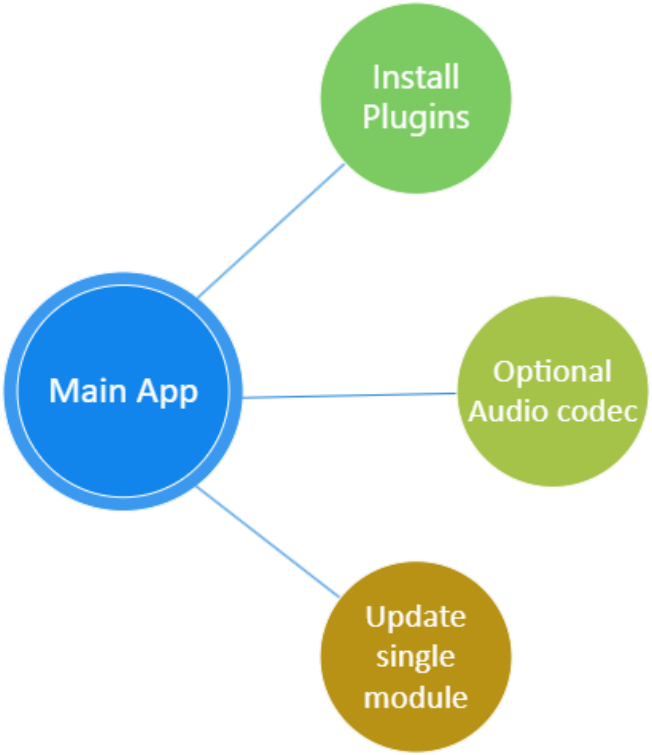
- Requirement
- Usage
- Implementation proposal in Zephyr

Requirement

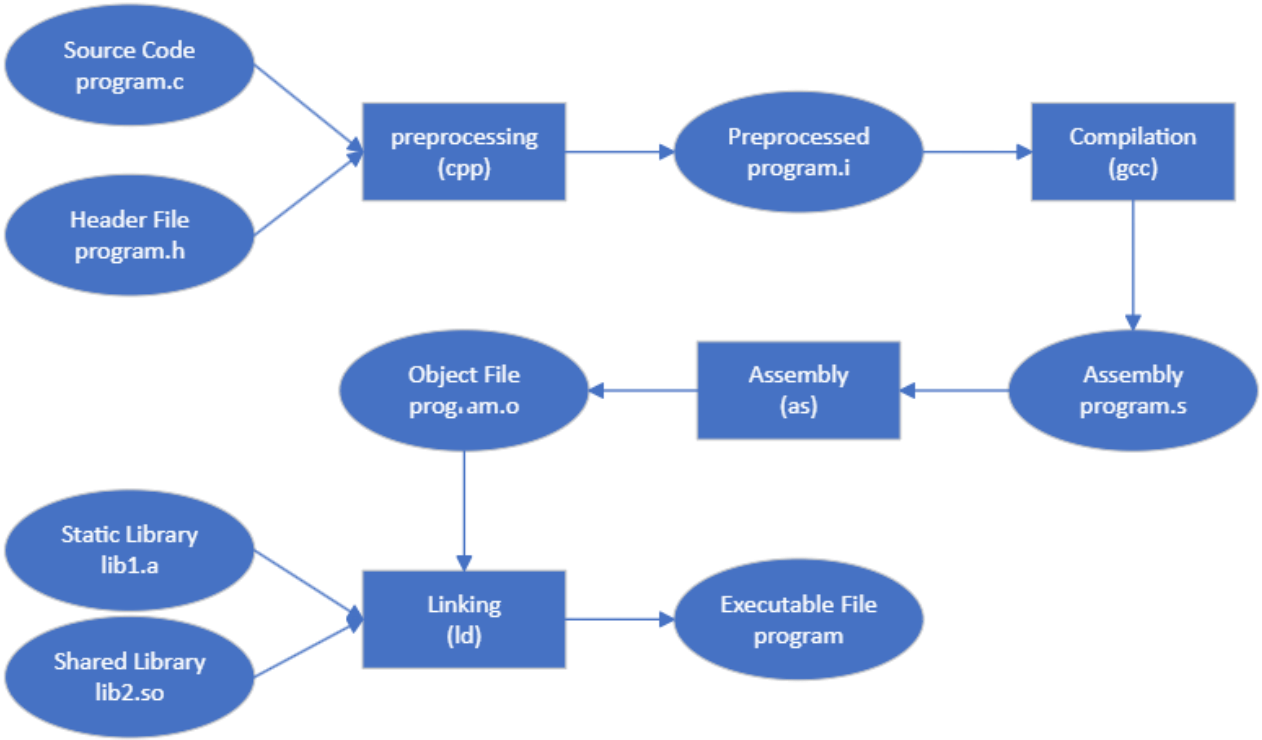
- Load a library at run time
- Execute functions or access variables in the loaded library
- Unload the library at run time

Usage

- Expand functionality
- Select optional module
- OTA(Over the Air) Update



Compile and Link Process

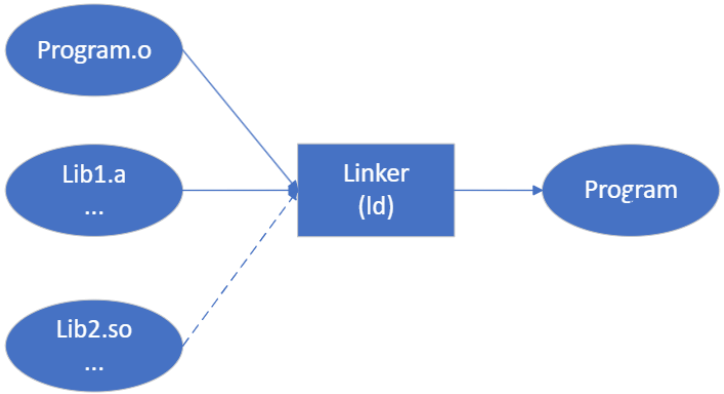


Linking Process

Combine one or more object files into a single executable, library file or other object file.

Mainly include two steps:

- Address and memory allocation
- Symbol relocation

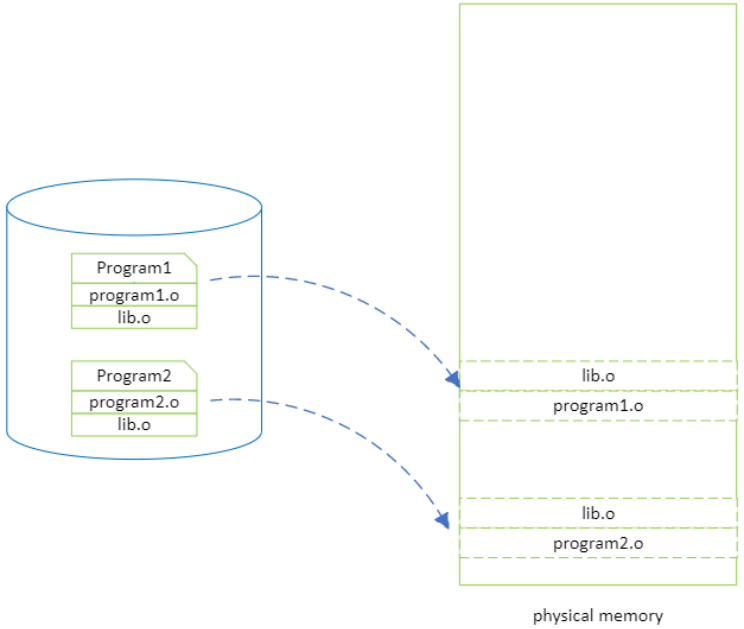


Static Linking

Linking all object files at compile time.

Disadvantages:

- memory wasting
- hard to update

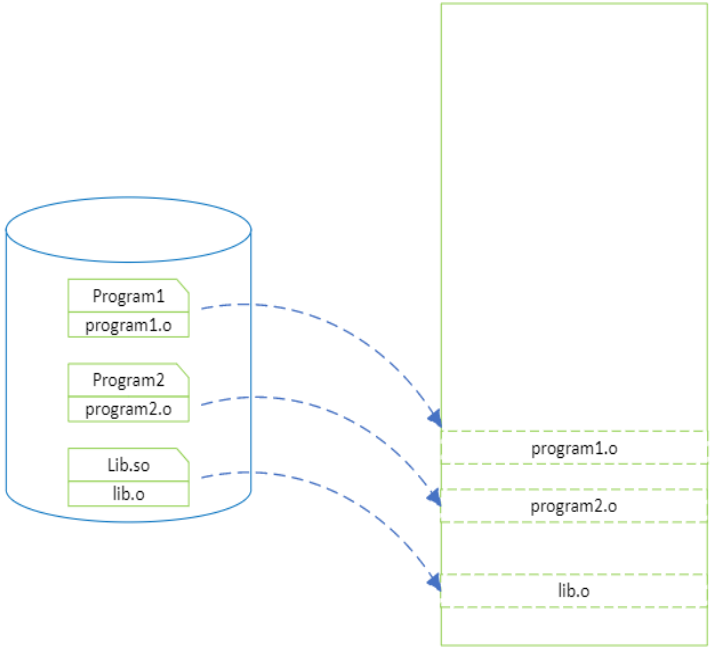


Dynamic Linking

Postpone the linking process to run time.

- Load main app into memory
- Find and load all shared libraries
- Symbol relocation
- Program starts to execute

All of this is based on a dynamic linker provided by OS. (In Linux, it's ld.so.)



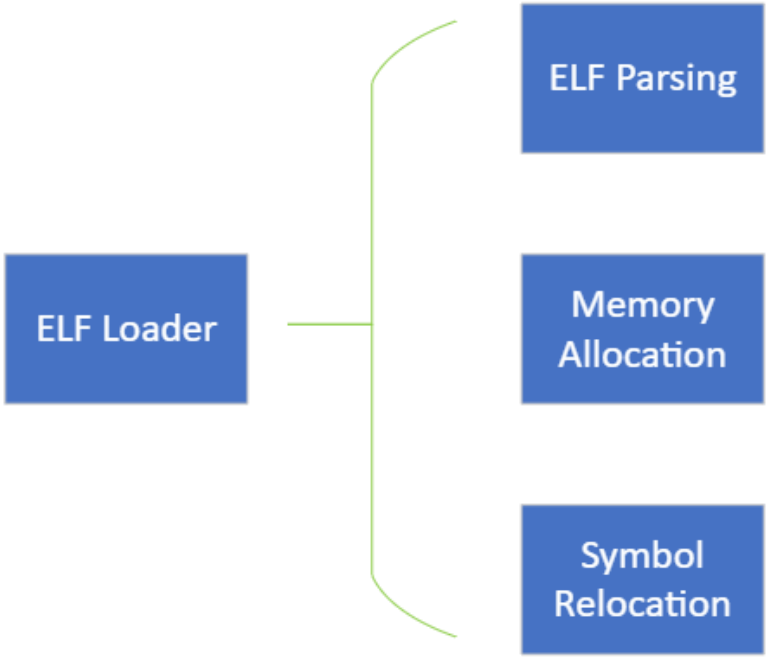
Dynamic Loading

Libraries loading and symbols relocation are triggered by application, rather than OS dynamic linker.

OS need to provide a dynamic loading API to achieve the same function with dynamic linker, like the DL API in Linux.

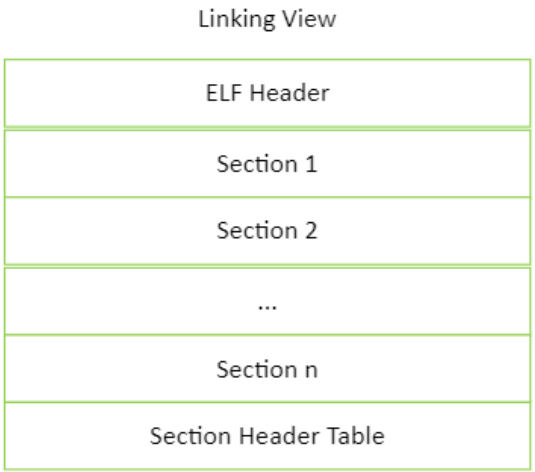
Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a dlopen ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

Implementation proposal in Zephyr



ELF Parsing

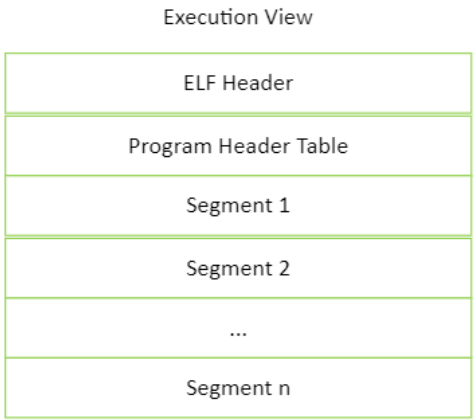
ELF File Format




```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            00000154 000154 000024 00   A  0   0  4
  [ 2] .gnu.hash         GNU_HASH        00000178 000178 000024 04   A  3   0  4
  [ 3] .dynsym           DYNSYM          0000019c 00019c 000090 10   A  4   1  4
  [ 4] .dynstr           STRTAB          0000022c 00022c 000079 00   A  0   0  1
  [ 5] .gnu.version      VERSYM          000002a6 0002a6 000012 02   A  3   0  2
  [ 6] .gnu.version_r    VERNEED         000002b8 0002b8 000020 00   A  4   1  4
  [ 7] .rel.dyn          REL             000002d8 0002d8 000050 08   A  3   0  4
  [ 8] .rel.plt          REL             00000328 000328 000008 08  AI  3  20  4
  [ 9] .init             PROGBITS        00001000 001000 000024 00  AX  0   0  4
  [10] .plt              PROGBITS        00001030 001030 000020 04  AX  0   0 16
  [11] .text             PROGBITS        00001050 001050 00014c 00  AX  0   0 16
  [12] .fini             PROGBITS        0000119c 00119c 000018 00  AX  0   0  4
  [13] .eh_frame_hdr     PROGBITS        00002000 002000 00002c 00   A  0   0  4
  [14] .eh_frame         PROGBITS        0000202c 00202c 000098 00   A  0   0  4
  [15] .init_array       INIT_ARRAY      00003ef8 002ef8 000004 04  WA  0   0  4
  [16] .fini_array       FINI_ARRAY      00003efc 002efc 000004 04  WA  0   0  4
  [17] .data.rel.ro      PROGBITS        00003f00 002f00 000004 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         00003f04 002f04 0000e0 08  WA  4   0  4
  [19] .got              PROGBITS        00003fe4 002fe4 00001c 04  WA  0   0  4
  [20] .got.plt          PROGBITS        00004000 003000 000010 04  WA  0   0  4
```

ELF Parsing

ELF File Format




```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x00330 0x00330 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x001b4 0x001b4 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x000c4 0x000c4 R   0x1000
  LOAD           0x002ef8 0x00003ef8 0x00003ef8 0x00118 0x00120 RW  0x1000
  DYNAMIC        0x002f04 0x00003f04 0x00003f04 0x000e0 0x000e0 RW  0x4
  NOTE           0x000154 0x00000154 0x00000154 0x00024 0x00024 R   0x4
  GNU_EH_FRAME   0x002000 0x00002000 0x00002000 0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002ef8 0x00003ef8 0x00003ef8 0x00108 0x00108 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt
   01     .init .plt .text .fini
   02     .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .data.rel.ro .dynamic .got .got.plt .bss
   04     .dynamic
   05     .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .data.rel.ro .dynamic .got
```

All segments with PT_LOAD type should be copied into memory.

Memory Allocation

Problem Statement:

Zephyr only supports to allocate a continuous virtual space for platforms with MMU(x86 platforms and some arm platforms), without MMU, how do we allocate a continuous memory for modules loading?

Proposal:

Reserve a memory in advance, when Zephyr boots up, use this memory to load modules.

```
K_HEAP_DEFINE(_module_mem_heap, CONFIG_MODULE_MEMORY_SIZE);
```

Symbol Relocation

Problem Statement:

Lots of architectures have a lot of very different relocation types, it's difficult to cover all.

Proposal:

Use GOT only relocation type to simplify symbol relocation process.

GOT Only Relocation

Precondition:

Need to add flag “-shared –fPIC –fno-plt” to compile module object files.

- -shared: generate DYN (shared object file)
- -fPIC: generate Position Independent Code
- -fno-plt: disable PLT (Procedure Linkage Table)

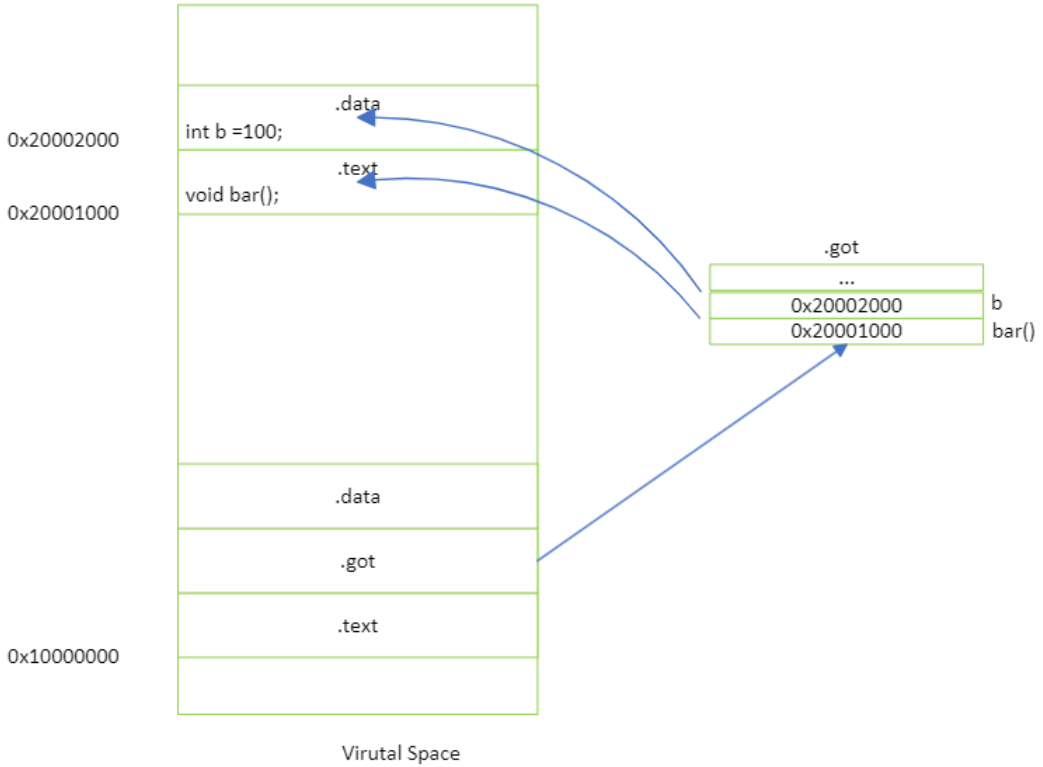
GOT: Global Offset Table, used to find symbols’ address which are defined in other modules.

GOT Relocation Example

```
extern int b;
extern void bar();

void foo()
{

   b = 1;
   bar();
}
```



WIP PR

https://github.com/zephyrproject-rtos/zephyr/pull/41700

Any comments or suggestion are welcome!

THANKS

Zephyr® Project
Developer Summit 2022