

*Making tool-chain independent
libraries for ARM platforms*

By Ketan Patel





Making tool-chain independent libraries for ARM platforms

Most of the SoC manufactures' customers use different software tool chains to develop applications on top of SoC. This leads to maintaining multiple source code trees. ARM based SoC can support multiple software tool chains to develop application. Support for various tool chains leads to maintaining multiple source code trees. Eventually it becomes very difficult to synchronize, updates / new features, across different branches. This tool tip will talk about how to generate / make tool-chain independent libraries for a given ARM platform.

Audio-video algorithm porting efforts are made to translate the software targeted for the source execution environment into a form that can be executed in destination environment. Once the audio-video algorithm is built on particular tool chain, modifying its assembly and C code to support other tool chains is a challenging task.

Various tool chains have different environments that support different mnemonics. Translation of the source code's assembler directives and mnemonics between different tool chains requires major efforts. The differences between various versions of tool chains lie among compiler, assembler and linker.

Compiler and assembler generate symbols that have different definitions for various tool chains. In order to port a library, the compiler and assembler should be modified to have common definition for all linkers. Modification of the behavior of compiler, assembler and linker require in-depth understanding of various tool chains.

The same library can be used on different tool chains by changing opcode entries in symbol table and relocation table. A simple patch allows library built for ARM platform to link to different tool chains. For eg., GreenHills MULTI, Linux, Arm Developer Suite (ADS).

ARM library symbol table

For every function and variables, defined in file, compiler makes an entry in symbol table. Symbol table contains information needed to locate and relocate a program's symbolic definitions and references. The symbols in library files convey specific information to the linker and loader. Below is a format of symbol table entry for ARM elf library.

```
<Base address> <Global/Local><Function/Object/Data> <Section> <Size> <Function Name>
```



ARM library relocation table

Relocation connects symbolic references with symbolic definitions. When a symbol is not found at compile time, compiler adds entry for that symbol in relocation table. Relocation table entry for ARM elf library is as under:

<OFFSET> 4 bytes	<TYPE> 1 byte	<VALUE> 3 byte symbol + 4 byte offset
---------------------	------------------	--

A relocation section refers to two other sections: a symbol table and a section to modify. The section header's sh_info and sh_link members specify these relationships.

When we build library on GreenHills MULTI tool chain for ARM platform it will not link directly on ARM-Linux and ADS. Understanding of compiler and linker is different from one tool chain to another. Below are some problem definitions and solutions.

Problems with Branch instruction B (conditional and unconditional):

Branch instruction (B) branches program counter to a particular offset within the code segment. Branches (B) to labels that have entry in relocation table, sometimes jumps to any incorrect memory location when it is linked on different tool chain. This is due to the offset available in opcode. On the flip side, all branches (B) to labels that are not present in relocation table will link to correct address.

$$\text{Wrong branch address} = (\text{<offset in opcode> * 4}) + 8 + \text{<address of label (from symbol table)>}$$

This address is incorrect as it refers to an offset which is interpreted differently for different tool-chains. If we remove this offset, the linker will fetch the address of this label from symbol table.

Problems with temporary labels (.L<number>):

.L<number> is a temporary label defined by compiler that is generated when the instruction to jump is at distant offset.

A compiler does not replace all function calls by eb000000 opcode (BL instruction). A function call may be replaced by ldr (load register with address) and a branch instruction. Compiler creates a temporary label name .L<number>, and makes a corresponding entry in the relocation table. The type field of this



entry is R_ARM_ABS32 (0x02). A linker locates the entry in relocation table corresponding to the label and replaces actual address of function at temporary label. Upon execution, the ldr instruction loads value from temporary label and branches to this value, assuming that loaded value is the correct address of a function.

For example c code :

```
func3()
{
    func1();
}
Disassembly:
func3:                                // code section
20c: e59fc434    ldr    ip, [pc, #1076]; 648 <.L208>
210: e12fff3c    blx   ip
.
.
.
00000648 <.L208>:
648: 01435a32    ....
                                648: R_ARM_ABS32    func1 + 0x0
```

The relocation table contains an entry for the opcode at 648, as described above. On the address 648, some garbage value is stored i.e. 0x01435a32. At the time of linking this value is replaced with,

Value at 648 = 0x01435a32 + <address of func1 function (from symbol table)>

To resolve this problem, replace the offset in opcode with the value specified in relocation table entry. Linker will then add this offset (same as that available in relocation table) to the base address of label.

CONCLUSION

By changing opcodes, entries in symbol table and relocation table it is possible to translate a library in order to support different tool chains. This concept has been implemented and tested thoroughly. Further it is also found that this concept does not compromise with performance.