Assembler Annotations

This document describes the new macros for annotation of data and code in assembly. In particular, it contains information about `SYM_FUNC_START`, `SYM_FUNC_END`, `SYM_CODE_START`, and similar.

Rationale

Some code like entries, trampolines, or boot code needs to be written in assembly. The same as in C, such code is grouped into functions and accompanied with data. Standard assemblers do not force users into precisely marking these pieces as code, data, or even specifying their length. Nevertheless, assemblers provide developers with such annotations to aid debuggers throughout assembly. On top of that, developers also want to mark some functions as *global* in order to be visible outside of their translation units.

Over time, the Linux kernel has adopted macros from various projects (like `binutils`) to facilitate such annotations. So for historic reasons, developers have been using `ENTRY`, `END`, `ENDPROC`, and other annotations in assembly. Due to the lack of their documentation, the macros are used in rather wrong contexts at some locations. Clearly, `ENTRY` was intended to denote the beginning of global symbols (be it data or code). `END` used to mark the end of data or end of special functions with *non-standard* calling convention. In contrast, `ENDPROC` should annotate only ends of *standard* functions.

When these macros are used correctly, they help assemblers generate a nice object with both sizes and types set correctly. For example, the result of `arch/x86/lib/putuser.S`:

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
 25: 0000000000000000    33 FUNC    GLOBAL DEFAULT    1 __put_user_1
 29: 0000000000000030    37 FUNC    GLOBAL DEFAULT    1 __put_user_2
 32: 0000000000000060    36 FUNC    GLOBAL DEFAULT    1 __put_user_4
 35: 0000000000000090    37 FUNC    GLOBAL DEFAULT    1 __put_user_8
```

This is not only important for debugging purposes. When there are properly annotated objects like this, tools can be run on them to generate more useful information. In particular, on properly annotated objects, `objtool` can be run to check and fix the object if needed. Currently, `objtool` can report missing frame pointer setup/destruction in functions. It can also automatically generate annotations for :doc:`ORC unwinder <x86/orc-unwinder>` for most code. Both of these are especially important to support reliable stack traces which are in turn necessary for :doc:`Kernel live patching <livepatch/livepatch>`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\[linux-master][Documentation]asm-annotations.rst`, line 41); *backlink*

Unknown interpreted text role "doc".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\[linux-master][Documentation]asm-annotations.rst`, line 41); *backlink*

Unknown interpreted text role "doc".

Caveat and Discussion

As one might realize, there were only three macros previously. That is indeed insufficient to cover all the combinations of cases:

- standard/non-standard function
- code/data
- global/local symbol

There was a discussion and instead of extending the current `ENTRY/END*` macros, it was decided that brand new macros should be introduced instead:

```
So how about using macro names that actually show the purpose, instead
of importing all the crappy, historic, essentially randomly chosen
debug symbol macro names from the binutils and older kernels?
```

Macros Description

The new macros are prefixed with the `SYM_` prefix and can be divided into three main groups:

1. `SYM_FUNC_*` -- to annotate C-like functions. This means functions with standard C calling conventions. For example, on x86, this means that the stack contains a return address at the predefined place and a return from the function can happen in a standard way. When frame pointers are enabled, save/restore of frame pointer shall happen at the start/end of a function, respectively, too.

Checking tools like objtool should ensure such marked functions conform to these rules. The tools can also easily annotate these functions with debugging information (like *ORC data*) automatically.

2. SYM_CODE_* -- special functions called with special stack. Be it interrupt handlers with special stack content, trampolines, or startup functions.

   Checking tools mostly ignore checking of these functions. But some debug information still can be generated automatically. For correct debug data, this code needs hints like UNWIND_HINT_REGS provided by developers.

3. SYM_DATA* -- obviously data belonging to .data sections and not to .text. Data do not contain instructions, so they have to be treated specially by the tools: they should not treat the bytes as instructions, nor assign any debug information to them.

Instruction Macros

This section covers SYM_FUNC_* and SYM_CODE_* enumerated above.

objtool requires that all code must be contained in an ELF symbol. Symbol names that have a .L prefix do not emit symbol table entries. .L prefixed symbols can be used within a code region, but should be avoided for denoting a range of code via SYM_*_START/END annotations.

- SYM_FUNC_START and SYM_FUNC_START_LOCAL are supposed to be **the most frequent markings**. They are used for functions with standard calling conventions -- global and local. Like in C, they both align the functions to architecture specific __ALIGN bytes. There are also _NOALIGN variants for special cases where developers do not want this implicit alignment.

  SYM_FUNC_START_WEAK and SYM_FUNC_START_WEAK_NOALIGN markings are also offered as an assembler counterpart to the *weak* attribute known from C.

  All of these **shall** be coupled with SYM_FUNC_END. First, it marks the sequence of instructions as a function and computes its size to the generated object file. Second, it also eases checking and processing such object files as the tools can trivially find exact function boundaries.

  So in most cases, developers should write something like in the following example, having some asm instructions in between the macros, of course:

  ```
  SYM_FUNC_START(memset)
      ... asm insns ...
  SYM_FUNC_END(memset)
  ```

  In fact, this kind of annotation corresponds to the now deprecated ENTRY and ENDPROC macros.

- SYM_FUNC_ALIAS, SYM_FUNC_ALIAS_LOCAL, and SYM_FUNC_ALIAS_WEAK can be used to define multiple names for a function. The typical use is:

  ```
  SYM_FUNC_START(__memset)
      ... asm insns ...
  SYN_FUNC_END(__memset)
  SYM_FUNC_ALIAS(memset, __memset)
  ```

  In this example, one can call __memset or memset with the same result, except the debug information for the instructions is generated to the object file only once -- for the non-ALIAS case.

- SYM_CODE_START and SYM_CODE_START_LOCAL should be used only in special cases -- if you know what you are doing. This is used exclusively for interrupt handlers and similar where the calling convention is not the C one. _NOALIGN variants exist too. The use is the same as for the FUNC category above:

  ```
  SYM_CODE_START_LOCAL(bad_put_user)
      ... asm insns ...
  SYM_CODE_END(bad_put_user)
  ```

  Again, every SYM_CODE_START* **shall** be coupled by SYM_CODE_END.

  To some extent, this category corresponds to deprecated ENTRY and END. Except END had several other meanings too.

- SYM_INNER_LABEL* is used to denote a label inside some SYM_{CODE,FUNC}_START and SYM_{CODE,FUNC}_END. They are very similar to C labels, except they can be made global. An example of use:

  ```
  SYM_CODE_START(ftrace_caller)
      /* save_mcount_regs fills in first two parameters */
      ...

  SYM_INNER_LABEL(ftrace_caller_op_ptr, SYM_L_GLOBAL)
      /* Load the ftrace_ops into the 3rd parameter */
      ...

  SYM_INNER_LABEL(ftrace_call, SYM_L_GLOBAL)
      call ftrace_stub
      ...
      retq
  SYM_CODE_END(ftrace_caller)
  ```

Data Macros

Similar to instructions, there is a couple of macros to describe data in the assembly.

- `SYM_DATA_START` and `SYM_DATA_START_LOCAL` mark the start of some data and shall be used in conjunction with either `SYM_DATA_END`, or `SYM_DATA_END_LABEL`. The latter adds also a label to the end, so that people can use `lstack` and (local) `lstack_end` in the following example:

  ```
  SYM_DATA_START_LOCAL(lstack)
      .skip 4096
  SYM_DATA_END_LABEL(lstack, SYM_L_LOCAL, lstack_end)
  ```

- `SYM_DATA` and `SYM_DATA_LOCAL` are variants for simple, mostly one-line data:

  ```
  SYM_DATA(HEAP,     .long rm_heap)
  SYM_DATA(heap_end, .long rm_stack)
  ```

  In the end, they expand to `SYM_DATA_START` with `SYM_DATA_END` internally.

Support Macros

All the above reduce themselves to some invocation of `SYM_START`, `SYM_END`, or `SYM_ENTRY` at last. Normally, developers should avoid using these.

Further, in the above examples, one could see `SYM_L_LOCAL`. There are also `SYM_L_GLOBAL` and `SYM_L_WEAK`. All are intended to denote linkage of a symbol marked by them. They are used either in `_LABEL` variants of the earlier macros, or in `SYM_START`.

Overriding Macros

Architecture can also override any of the macros in their own `asm/linkage.h`, including macros specifying the type of a symbol (`SYM_T_FUNC`, `SYM_T_OBJECT`, and `SYM_T_NONE`). As every macro described in this file is surrounded by `#ifdef` + `#endif`, it is enough to define the macros differently in the aforementioned architecture-dependent header.