Спецификация внутрифайловых метаданных

для проекта

FX-RTOS

версия 1.1

Prosoft®

15.04.2015

Contents

1 Введение

1.1 Назначение

В настоящем документе описаны функции и формат метаданных, используемых внутри файлов исходных текстов и заголовочных файлов ОСРВ FX-RTOS.

1.2 Предметная область

Язык С не содержит встроенной поддержки метаданных, тем не менее, часто возникает ситуация, когда коду на С требуется сопоставить некоторую информацию, которая могла бы быть использована внешними инструментами (например, информация о зависимостях, свойства функций, импортируемые (ожидаемые) макроопределения и т.п. Целесообразно размещать эту информацию внутри исходных текстов и заголовочных файлов, делая тем самым исходные тексты набором независимых функциональных единиц – модулей, все внешние зависимости которых могут быть проанализированы программным образом (в т.ч. конфигурационные: зависимости от внешних макроопределений).

1.3 Ссылки

Спецификация YAML - http://yaml.org/

Спецификация JSON - http://json.org/

2 Общее описание

2.1 Формат

Для размещения метаданных используется предопределенный макрос FX_METADATA, этот макрос должен быть определен извне (ключом компилятора или принудительно включаемым файлом) как пустой. Макрос имеет один аргумент, поэтому аргумент заключается в двойные круглые скобки, чтобы исключить трактовку запятых внутри метаданных как разделителей аргументов макроса:

```
FX_METADATA((…))
```

Метаданные могут располагаться в нескольких строках (при этом конфигуратор должен трактовать их как односрочную запись).

```
FX_METADATA((   …
                …
                …
                …))
```

Выравнивание не имеет значения.

Текст в круглых скобках должен быть представлен в формате JSON (пары ключ-значение) с единственным исключением – ключи не должны заключаться в кавычки. Т.к. метаданные всегда должны быть представлены в виде словаря, размещаемый в кавычках текст всегда должен иметь

формат словаря (заключаться в фигурные скобки).

Пример:

```
FX_METADATA(({ key1: scalar_value,
               key2: [list_val1, list_val2]
               key3: { key4: val1, key5: "val2" }))
```

Все скалярные значения (даже те, которые не заключены в кавычки) рассматриваются как текст. Возможности записи чисел в различных форматах (восьмеричном, шестнадцатиричном и т.д.) определяются внешним инструментом, анализирующим метаданные.

Ключи верхнего уровня (в корневом словаре, который содержится в круглых скобках) могут быть как объединены в одной директиве, так и содержаться в нескольких. Например:

```
FX_METADATA(({ key1: val1 }))
FX_METADATA(({ key2: val2 }))
FX_METADATA(({ key3: val3 }))
```

Может быть описано как:

```
FX_METADATA(({ key1: val1, key2: val2, key3: val3 }))
```

Хотя, при извлечении метаданных, имеется возможность различать эти случаи, рекомендуется, чтобы внешний инструмент производил поиск ключа во всех записях метаданных, которые встречаются в файле.

2.2 Размещение метаданных в исходных текстах

Не накладывается никаких ограничений на размещение метаданных в тексте программ и заголовочных файлов. Один файл может содержать несколько записей метаданных (каждая запись должна представлять собой словарь). Рекомендуется ограничивать использование метаданных в стандарте кодирования.

3 Описание элементов метаданных

Данная спецификация описывает три основных компонента метаданных, используемых в настоящий момент в проекте FX-RTOS – описание экспортируемых интерфейсов (имён), конструкторов и опций модулей.

3.1 Экспорт интерфейсов

Совокупность определений типов данных, прототипов функций и прочей документированной информации, находящейся в заголовочном файле называется интерфейсом. Этот интерфейс имеет сопоставляемое ему имя (не зависящее от имени заголовочного файла), которое должно использоваться в директивах #include. Заголовочный файл должен содержать явную метку экспортируемого интерфейса, а файл исходного текста – явную метку реализуемого интерфейса.

Для одного модуля, имена экспортируемого и реализуемого интерфейса (имена, содержащееся в заголовочных файла и файлах исходных текстов соответственно) должны совпадать.

Экспортируемый интерфейс доступен по ключу interface, по которому должен быть доступен список, содержащий как минимум 2 элемента: имя интерфейса и имя реализации:

```
interface: [<interface_name>, <implementation_name>]
```

Пример:

```
interface: [MY_INTERFACE, MY_IMPLEMENTATION]
```

Метка interface может размещаться **только** в заголовочных файлах (записи в файлах исходных текстов игнорируются). Целиком запись метаданных будет выглядеть так:

```
FX_METADATA({ interface: [MY_INTERFACE, MY_IMPLEMENTATION] })
```

3.2 Реализация интерфейсов

Запись метаданных, соответствующая рализации интерфейса может быть размещена только в файлах исходных текстов (записи в заголовочных файлах игнорируются).

Описание реализации аналогично описанию экспорта интерфейса в заголовочных файлах, отличается только ключ – в описании реализации ключом является implementation. Таким образом, реализация интерфейса выглядит так:

```
implementation: [<interface_name>, <implementation_name>]
```

Пример:

```
implementation: [MY_INTERFACE, MY_IMPLEMENTATION]
```

Целиком запись метаданных в файлах исходных текстов выглядит так:

```
FX_METADATA({implementation: [MY_INTERFACE,MY_IMPLEMENTATION]})
```

3.3 Опции модулей

Опции являются внешними зависимостями, которые должны быть указаны пользователем. Они представляют собой константы, которые обычно содержатся в конфигурационном заголовочном файле. Например, максимальные размеры каких-либо массивов, максимальные значения каких-то конфигурационных параметров, все то, что обычно указывается в виде константных директив #define. Из-за возможности заменять реализации модулей (каждый из которых может иметь свои, либо не иметь опций), использование единого заголовочного файла содержащего все опции нецелесообразно, набор нужных опций определяется конфигуратором и нужный заголовочный файл генерируется им же. Опции доступны в качестве интерфейса CFG_OPTIONS (если в интерфейсе описаны опции, он должен включать этот интерфейс, чтобу получить их значения).

3.3.1 Общий формат

Опции доступны по ключу options. По этому ключу должен быть доступен список опций. Опция описывается как именованный словарь – имя должно соответствовать макроопределению define (которое будет использоваться в коде модуля), словарь содержит описание опции. Таким образом общий формат выглядит следующим образом:

```
options: [ option1: {<описание опции1>}, option2: {<описание опции2>}, …]
```

В описании опции должны быть 3 обязательных элемента: **type** (тип опции), **default** (значение по-умолчанию, его интерпретация зависит от типа), **description** (текстовое описание опции). В зависимости от типа опции могут требоваться дополнительные парметры. Все обязательные элементы являются скалярными значениями.

3.3.2 Целочисленные константы

Целочисленные константы являются наиболее широко используемыми опциями. Опция данного типа сопоставляет константу некоторому имени, то есть, в конечном итоге, отображается на строку в консфигурационном файле, которая имеет вид:

```
#define OPTION_NAME <константа>
```

Например, опция STACK_ADDRESS со значением 0x00004000:

```
#define STACK_ADDRESS 0x00004000
```

В дополнение к стандартным компонентам описания опция, целочисленная опция содержит еще ключ range, который должен содержать список из двух элементов, содержащих минимальное, и максимальное значения для данной опции. Ключ type должен иметь значение int.

Пример описания целочисленной опции STACK_ADDRESS:

```
STACK_ADDRESS: { type: int, range: [0, 0xffff], default: 0x4000,
description: “Stack address.” }
```

В полном виде строка метаданных будет выглядеть следующим образом:

```
FX_METADATA(({ options: [
   STACK_ADDRESS: {
         type: int, range: [0, 0xffff], default: 0x4000,
         description: “Stack address.” } ]}))
```

В текущей реализации, целочисленные опции могут иметь как десятичные, так и шестнадцатиричные значения (последние предваряются префиксом 0x).

Значение default интерпретируется как значение константы по-умолчанию.

3.3.3 Перечисления

Бывает необходимо сопоставить макроопределению только определенный фиксированный набор значений (например, включить/отключить какое-то свойство), в этом случае должны использоваться перечисления.

В дополнение к стандартном компонентам записи опции, перечисление добавляет список values, которое содержит пары дружественных человеку названий значений и итогового значения (которое будет использовано в #define). Ключ type для перечисления должен содержать значение enum.

Пример описания опции-перечисления:

```
MY_FEATURE: { type: enum, values: [Disabled: 0, Enabled: 1], default: 0,
description: "My feature." }
```

В конфигураторе этой опции должен соответствовать comboBox, значениями которого будут "Disabled" и "Enabled", а при генерации конфигурационного файла, в качестве значения макроопределения, будут подставлены 0 или 1, в зависимости от того, что было выбрано в comboBox.

Значение default интерпретируется как индекс элемента в списке values, который соответствует значению опции по-умолчанию.

При необходимости, если элементы списка values содержат несколько слов, они могут заключаться в кавычки:

```
MY_FEATURE: { type: enum, values: ["Feature disabled": 0, "Feature
enabled": 1], default: 0, description: "My feature." }
```

Важно также отметить, что значения элементов подставляются в #define в текстовом виде, то есть не обязаны быть числами и типы этих значений могут различаться даже в пределах одной опции.

Например, если в одном случае, значение опции должно определяться пользователем, а в другом – должно использоваться значение некоторого дргугого макроса, опция может выглядеть так:

```
MY_FEATURE: { type: enum, values: [Disabled: 0, Enabled: ANOTHER_DEFINE],
default: 0, description: "My feature." }
```

Типы значений, перечисленных в списке values не совпадают, в одном случае используется число 0, а в другом – строка ANOTHER_DEFINE. При выборе пользователем первого или второго варианта, будут сгенерированы, соответственно, следующие макросы:

```
#define MY_FEATURE 0                    // If user selected "Disabled"
#define MY_FEATURE ANOTHER_DEFINE       // If user selected "Enabled"
```

Целиком описание опций в коде будет выглядеть следующий образом (пример с двумя опциями, целочисленной и перечислением):

```
FX_METADATA(({ options: [
   STACK_ADDRESS: {
         type: int, range: [0, 0xffff], default: 0x4000,
         description: "Stack address." },
   MY_FEATURE: {
         type: enum,
         values: [Disabled: 0, Enabled: ANOTHER_DEFINE],
         default: 0, description: "My feature." } ]}))
```

3.4 Конструкторы модулей

Каждый модуль может содержать конструктор, функцию – которая должна быть вызвана для инициализации внутренних структур данных. До того, как отработал конструктор, пользоваться API модуля нельзя. Поскольку каждый модуль может импортироваться многократно, явно вызывать конструктор из модулей, которые его используют нецелесообразно, т.к. требует затрат памяти на предотвращение повторного вызова конструктора из другого модуля. В данной реализации используется генерация кода функции, содержащей вызовы конструкторов, в порядке импорта интерфейсов (если модуль А использует модуль Б, конструктор модуля Б должен быть вызван раньше конструктора А). Это накладывает ограничения на циклические зависимости интерфейсов.

Конструктор описывается с помощью ключа ctor, по которому должен быть доступен список состоящий из двух элементов – имени функции конструктора и типа. Тип указывает, должен ли быть конструктор вызван только один раз, или контекст модуля содержит по одному экземпляру на каждый процессор, и, таким образом, конструктор должен быть вызван на каждом процессоре.

```
ctor: [<function_name>, on_boot_cpu или on_each_cpu]
```

Пример конструктора для модуля MY_MODULE, который содержит конструктор my_module_ctor, который должен быть вызван на загрузочном процессоре один раз при старте системы:

```
ctor: [my_module_ctor, on_boot_cpu]
```

Если конструктор должен быть вызован на каждом процессоре в системе, он описывается так:

```
ctor: [my_module_ctor, on_each_cpu]
```

3.5 Аспекты

Код модулей может содержать сквозную функциональность, либо может требоваться знание о конфигурации, которое может быть получено только после сборки системы. Например, перечисление модулей определенного типа, с сопоставлением каждому уникального номера или ключа, генерация таблиц указателей на функции, состав которых зависит от модулей, которые входят в конечную сборку. Т.к. модули не обладают информацией о конечной системе и об общем количестве модулей в системе, эта информация может быть получена только с помощью внешнего инструмента. Рассредоточенная по модулям информация, которая приобретает окончательный вод после конфигрирования системы называется аспектами. Все аспекты в системе доступны в виде интерфейса CFG_ASPECTS.

Аспекты определяются как массив пар "ключ"-"массив значений".

```
aspects: [ <aspect_pairs> ]
```

В качестве аспекта используется хэш-массив с одним ключом и списком значений:

```
aspects: [ {<key1-values1>}, {<key2-values2>}, {<key3-values2>} ]
```

Например:

```
aspects: [  { "key1": [  "value1",  "value2", "value3" ] } ]
```

После обработки исходных текстов, все аспекты из всех модулей, которые входят в целевую систему объединяются по следующему принципу:

Каждый ключ порождает макроопределение define, имя которого совпадает с именем ключа, а значением являются все значения, соответствующие этому ключу, собранные по всем модулям, входящим в проект.

Например, если в проект входят модули Module1.c и Module2.c, причем первый содержит метаданные вида:

```
aspects: [  { "key": [  "mod1_value1",  "mod1_value2" ] } ]
```

А второй модуль:

```
aspects: [  { "key": [  "mod2_value1",  "mod2_value2" ] } ]
```

Тогда результирующий интерфейс CFG_ASPECTS будет содержать следующее макроопределение:

```
#define key \
   mod1_value1 \
   mod1_value2 \
   mod2_value1 \
   mod2_value2
```

При генерации макроопределения не используются знаки препинания, поэтому они должны быть учтены на этапе описания аспекта.

Значение ключа с т.з. генератора аспектов является строкой символов, поэтому допустима генерация функциональных макросов вида:

```
#define key(a, b) \
   a##mod1_value1##b \
   a##mod1_value2##b
```

В случае генерации с помощью аспектов таких конструкций как поля структур, элементов массивов и т.д. следует использовать "замыкающий элемент", который позволяет избавиться от синтаксических ошибок из-за наличия конечного знака препинания в описании аспектов.

Для подсчета аспектов можно использовать перечисление вида:

```
enum
{
   MY_ASPECTS_DEFINE
   ASPECTS_COUNT
};
```

В данном примере MY_ASPECTS_DEFINE - аспектный макрос, содержащий набор элементов, который из которых включает в себя также запятую. ASPECTS_COUNT содержит количество аспектов.

4 Специальные вопросы

4.1 Упрощенный формат

Стандартный вариант использования метаданных предполагает предварительную обработку файла препроцессором, перед извлечением метаданных. В целях повышения

производительности, существует также «упрощенный формат», который является подмножеством обычного формата. В упрощенном формате не поддерживаются опции, а экспорт/реализация интерфейса и конструктор должны быть описаны в однострочной директиве. Такой формат позволяет разобрать метаданные с использованием регулярного выражения (без использования парсера JSON/YAML).

```
FX_METADATA({{interface: [I, VER1],ctor: [my_ctor, on_boot_cpu]}})
```

Реализация:

```
FX_METADATA({{implementation: [I, VER1]}})
```

При использовании опций, они должны быть описаны в отдельных директивах #pragma.

Предполагается, что пользователь уже имеет конфигурационный файл, содержащий все опции для данной конфигурации, поэтому инструмент, использующий упрощенный синтаксис, должен использоваться только для сборки и генерации списка конструкторов.

Конструкторы включены в упрощенный синтаксис из-за того, что отсутствие вызова конструктора (в отличие от отсутствия опции) не вызывает ошибку компиляции, поэтому желательно чтобы генерация конструкторов поддерживалась всеми инструментами и производилась автоматически.

4.2 Метаданные в комментариях

Рекомендуется обрабатывать файлы препроцессором, перед извлечение метаданных, это исключает возможность использования метаданных в комментариях. Вместе с тем, некоторые инструменты могут использовать упрощенный формат записи, и читать необработанный текст из файла с его разбором регулярными выражениями, такие инструменты могут увидеть метаданные, размещенные внтури комментариев. Для избежания путаницы, написание метаданных внутри комментариев запрещено.

4.3 Зарезервированные значения

Ключ dependencies в корневом словаре неявно используется для хранения зависимостей данного модуля, поэтому не должен использоваться в пользовательских метаданных.