什么变成Kconfig选项

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

在确定某些内容属于Kconfig时，它有助于区分具有提示的符号和不具有提示的符号。

如果符号有提示符（例如``bool'启用foo“``），则用户可以在``menuconfig``界面中更改符号的值（或通过手动编辑配置文件）。因此，如果用户有意义更改其值，则只在符号上添加提示。

在Zephyr中，Kconfig配置在选择机器后完成，因此通常，在与特定于机器的固定设置相对应的符号上放置提示是没有意义的。通常，这些设置应该通过设备树（``.dts``）文件来处理。

没有提示的符号不能由用户直接配置（它们从其他符号中获取它们的值），因此对它们的限制较少。如果某些派生设置在Kconfig中比在例如在构建期间，然后在Kconfig中进行，但在记住有和没有提示的符号之间保持区别。

有关处理某些计算机上已修复且可在其他计算机上配置的设置的方法，请参阅“可选提示”部分。

``select``语句

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

每当另一个符号为“y”时，``select``语句用于强制一个符号为“y”。 例如，每当``USB\_CONSOLE``为``y``时，以下代码强制``CONSOLE``为``y``：

.. code-block :: none

    配置CONSOLE

    bool“控制台支持”

   ...

    配置USB\_CONSOLE

    bool“USB控制台支持”

    选择CONSOLE

本节介绍了``select``的一些缺陷和用法。

``选择``陷阱

===================

``select``最初可能看起来像一个普遍有用的功能，但如果过度使用会导致配置问题。

例如，假设一个新的依赖项被添加到上面的``CONSOLE``符号，由一个不知道``USB\_CONSOLE``符号的开发人员（或者简单地忘记它）：

.. code-block :: none

   配置CONSOLE

   bool“控制台支持”

   取决于STRING\_ROUTINES

现在启用``USB\_CONSOLE``会强制``CONSOLE``到``y``，即使``STRING\_ROUTINES``是``n``。

要解决这个问题，需要添加``STRING\_ROUTINES``依赖项

``USB\_CONSOLE``：

.. code-block :: none

   配置USB\_CONSOLE

   bool“USB控制台支持”

   选择CONSOLE

   取决于STRING\_ROUTINES

   ...

   配置STRING\_ROUTINES

   bool“包含字符串例程”

从``if``和``menu``语句继承的依赖关系的更多阴险案例很常见。

解决问题的另一种尝试可能是转变“取决于”

进入另一个``select``：

.. code-block :: none

   配置CONSOLE

   bool“控制台支持”

   选择STRING\_ROUTINES

   ...

   配置USB\_CONSOLE

   bool“USB控制台支持”

   选择CONSOLE

实际上，这通常会放大问题，因为添加到``STRING\_ROUTINES``的任何依赖项现在都需要复制到``CONSOLE``和``USB\_CONSOLE``。

通常，每当更新符号的依赖性时，也必须更新（直接或间接）选择它的所有符号的依赖性。这在实践中经常被忽视，即使对于上面最简单的情况也是如此。

除了简单的辅助符号之外，应特别避免使用符号链相互选择，如下所述：ref：`good\_select\_use`。

自由使用``select``也会使Kconfig文件更难以阅读，这既是由于额外的依赖性，也是由于``select``的非本地特性，这隐藏了符号可能被启用的方式。

替换``select``

==========================

对于上一节中的示例，更好的解决方案通常是将``select``转换为``依赖于``：

.. code-block :: none

   配置CONSOLE

   bool“控制台支持”

   ...

   配置USB\_CONSOLE

   bool“USB控制台支持”

   取决于CONSOLE

这使得无法生成无效配置，并且意味着只需要在单个位置更新依赖关系。

在这里使用``依赖于``的反对意见可能是启用``USB\_CONSOLE``的配置文件现在也需要启用``CONSOLE``：

.. code-block :: none

   CONFIG\_CONSOLE = Y

   CONFIG\_USB\_CONSOLE = Y

这归结为权衡，但如果启用``CONSOLE``是常态，那么缓解是使``CONSOLE``默认为``y``：

.. code-block :: none

   配置CONSOLE

   bool“控制台支持”

   默认的y

这只在配置文件中提供了一个赋值：

.. code-block :: none

   CONFIG\_USB\_CONSOLE = Y

请注意，现在不需要“CONSOLE”的配置文件必须明确禁用它：

.. code-block :: none

   CONFIG\_CONSOLE =正

.. \_good\_select\_use：

使用``select``作为助手符号

===================================

“select”的良好和安全使用是用于设置捕获某些条件的“帮助”符号。这样的帮助符号最好不应有提示或依赖。

例如，用于指示特定CPU / SoC具有FPU的帮助符号可以定义如下：

.. code-block :: none

   配置CPU\_HAS\_FPU

   布尔

   救命

   如果为y，则CPU具有FPU

   ...

   配置SOC\_FOO

   布尔“FOO SoC”

   选择CPU\_HAS\_FPU

   ...

   配置SOC\_BAR

   布尔“BAR SoC”

   选择CPU\_HAS\_FPU

这使得其他符号可以通用方式检查FPU支持，而无需查找特定的体系结构：

.. code-block :: none

   配置FLOAT

   bool“支持浮点运算”

   取决于CPU\_HAS\_FPU

另一种方法是拥有如下的依赖关系，可能会在以下几个方面重复：

.. code-block :: none

   配置FLOAT

   bool“支持浮点运算”

   取决于SOC\_FOO || SOC\_BAR || ...

不带“`select”的隐形辅助符号也很有用。例如，如果机器具有一些任意定义的“大”内存量，则以下代码定义一个具有值“y”的辅助符号：

.. code-block :: none

   配置LARGE\_MEM

   def\_bool MEM\_SIZE> = 64

.. 注意：：

   这是以下内容的简称：

   .. code-block :: none

      配置LARGE\_MEM

      布尔

      默认MEM\_SIZE> = 64

``select``推荐

==========================

总之，以下是``select``的一些推荐做法：

- 避免选择带有提示或依赖关系的符号。更喜欢``取决于``。如果``依赖于``在配置文件中引起恼人的膨胀，请考虑为最常见的值添加Kconfig默认值。

  罕见的例外情况可能包括您确定所选符号和所选符号的相关性永远不会偏离同步的情况，例如：当处理在同一个``if``中彼此接近定义的两个简单符号时。

  常识适用，但要注意``select``经常会在实践中引起问题。 ``依赖``通常是一种更清洁，更安全的解决方案。

- 选择简单的帮助符号，无需提示和依赖，但您喜欢。它们是简化Kconfig文件的绝佳工具。

（缺乏）有条件的包括

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

``if``块为``if``中的每个项添加依赖项，就像使用了``依赖于``。

与``if``相关的常见误解是认为以下代码有条件地包含文件：file：`Kconfig.other`：

.. code-block :: none

   如果DEP

   来源“Kconfig.other”

   万一

实际上，Kconfig中没有条件包含。 ``if``在``source``周围没有特殊含义。

.. 注意：：

   条件包含是不可能实现的，因为``if``条件可能包含（直接或间接）对尚未定义的符号的前向引用。

说：file：`Kconfig.other`上面包含这个定义：

.. code-block :: none

   配置FOO

   布尔“支持foo”

在这种情况下，```FOO``将以此定义结束：

.. code-block :: none

   配置FOO

   布尔“支持foo”

   取决于DEP

请注意，在`file：`Kconfig.other`中添加``依赖DEP``到``FOO``的定义是多余的，因为``DEP``依赖已经被添加了``if DEP ``。

通常，尽量避免添加冗余依赖项。它们可以使Kconfig文件的结构更难理解，并且还使更改更容易出错，因为很难发现相同的依赖关系被添加两次。

检查``menuconfig``中的更改

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

在添加新符号或对Kconfig文件进行其他更改时，最好在以后查找：ref：`menuconfig <override\_kernel\_conf>`接口中的符号。要快速找到符号，请使用menuconfig的跳转功能（按：kbd：`/`）。

以下是一些要检查的事项：

\*符号是否放在一个好位置？检查它们是否出现在有意义的菜单中，靠近相关符号。

  如果一个符号依赖于另一个符号，那么将它放在它所依赖的符号之后通常是个好主意。然后它将相对于它在``menuconfig``界面中依赖的符号进行缩进显示。如果在它们所依赖的符号之后放置了几个符号，这也有效。

\*从提示中猜出符号的作用是否容易？

\*如果添加了许多符号，是否可以将所有值组合设置为有意义？

  例如，如果添加两个符号“FOO\_SUPPORT”和“NO\_FOO\_SUPPORT”，并且两者都可以同时启用，那么这将构成一个无意义的配置。在这种情况下，拥有一个“FOO\_SUPPORT”符号可能更好。

\*是否存在重复的依赖关系？

  这可以通过选择符号并按：kbd：`？`来查看符号信息来检查。如果存在重复的依赖关系，则使用符号信息中显示的``Included via ...``路径来确定它们的来源。

风格建议和缩写

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

本节提供了一些样式建议，并解释了一些常见的Kconfig缩写。

考虑常见的依赖关系

=================================

如果一系列符号/选择共享一个共同的依赖关系，那么依赖关系可以用``if``来计算。

例如，请考虑以下代码：

.. code-block :: none

   配置FOO

   布尔“富”

   取决于DEP

   配置BAR

   布尔“酒吧”

   取决于DEP

   选择

   提示“选择”

   取决于DEP

   配置BAZ

   布尔“Baz”

   配置QAZ

   布尔“卡兹”

   endchoice

在这里，``DEP``依赖可以像这样考虑：

.. code-block :: none

   如果DEP

   配置FOO

   布尔“富”

   配置BAR

   布尔“酒吧”

   选择

   提示“选择”

   配置BAZ

   布尔“Baz”

   配置QAZ

   布尔“卡兹”

   endchoice

   endif #DEP

.. 注意：：

   在内部，代码的第二个版本转换为第一个版本。

如果具有共享依赖关系的符号/选项序列都在同一菜单中，则依赖关系可以放在菜单本身上：

.. code-block :: none

   菜单“Foo功能”

   取决于FOO\_SUPPORT

   配置FOO\_FEATURE\_1

   布尔“Foo功能1”

   配置FOO\_FEATURE\_2

   布尔“Foo功能2”

   endmenu

如果``FOO\_SUPPORT``是``n``，整个菜单就会消失。

冗余默认值

==================

``bool``符号默认为``n``，``string``符号默认为空字符串。因此，``default n``和``default'“``（几乎）总是多余的。

Zephyr中推荐的样式是跳过``bool``和``string``符号的冗余默认值。这也产生了更清晰的文档：（\*隐含默认为n \*而不是\* n，如果<依赖，可能继承> \*）。

.. 注意：：

   “default n`` /``default”“``不是多余的一种情况是在多个位置定义符号并想要覆盖例如在后面的定义中的``default y``。

默认值\*应该总是给``int``和``hex``符号，因为它们隐含地默认为空字符串。这部分是为了与C Kconfig工具的兼容性，尽管与其他符号类型相比，隐式0默认值可能不太可能是预期的。

常见的缩写

=================

Kconfig有两个简写来处理提示和默认值。

- ``<type>“提示符”``是一个表示符号/选择同时输入类型和提示的简写。这两个定义是相同的：

  .. code-block :: none

     配置FOO

     bool“foo”

  .. code-block :: none

     配置FOO

     布尔

     提示“foo”

  第一种风格，简写，在Zephyr中是首选。

- ```def\_ <type> <value>``是一个同时给出类型和值的简写。这两个定义是相同的：

  .. code-block :: none

     配置FOO

     def\_bool BAR && BAZ

  .. code-block :: none

     配置FOO

     布尔

     默认BAR && BAZ

在同一个定义中同时使用``<type>'提示符```和``def\_ <type> <value>``简写是多余的，因为它给出了两次类型。

``def\_ <type> <value>``速记通常只对没有提示的符号有用，而且有点模糊。

.. 注意：：

   对于在多个位置定义的符号（例如，在Zephyr的``Kconfig.defconfig``文件中），最好只给出符号“base”定义的符号类型，并使用``default` `（而不是``def\_ <type> value``）用于剩余的定义。这样，如果删除了符号的基本定义，则符号最终没有类型，这会生成指向其他定义的警告。这使得额外的定义更容易发现和删除。

鲜为人知/使用Kconfig功能

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

本节列出了一些可能仍然派上用场的更加模糊的Kconfig行为和功能。

“暗示``声明

=======================

``imply``语句类似于``select``，但尊重依赖关系并且不强制赋值。例如，以下代码可用于在FOO SoC上默认启用USB键盘支持，同时仍允许用户将其关闭：

.. code-block :: none

   配置SOC\_FOO

   布尔“FOO SoC”

   暗示USB\_KEYBOARD

   ...

   配置USB\_KEYBOARD

   bool“USB键盘支持”

``imply``就像一个建议，而``select``强制一个值。

可选提示

================

条件可以放在符号的提示符上，以使其可由用户选择。例如，在某些电路板上硬编码为0xFF并在其他电路板上可配置的值“MASK”可以表示如下：

.. code-block :: none

   配置MASK

   十六进制“位掩码”，如果HAS\_CONFIGURABLE\_MASK

   默认为0xFF

.. 注意：：

   这是以下内容的简称：

   .. code-block :: none

      配置MASK

      十六进制

      如果HAS\_CONFIGURABLE\_MASK，则提示“位掩码”

      默认为0xFF

板上会选择``HAS\_CONFIGURABLE\_MASK``辅助符号来表示``MASK``是可配置的。当``MASK``是可配置的时，它也将默认为0xFF。

可选的选择

================

使用``optional``关键字定义选项允许切换整个选项以不选择任何符号：

.. code-block :: none

   选择

   提示“使用旧协议”

   可选的

   config LEGACY\_PROTOCOL\_1

   bool“Legacy protocol 1”

   配置LEGACY\_PROTOCOL\_2

   bool“Legacy protocol 2”

   endchoice

在menuconfig界面中，将显示例如as`` [\*]使用遗留协议（LEGACY\_PROTOCOL\_1）--->``，可以切换选项以启用这两个符号。

``可见if``条件

=========================

在菜单上放置一个``visible if``条件会隐藏菜单及其中的所有符号，同时仍然允许符号默认值启动。

作为一个激励性的例子，请考虑以下代码：

.. code-block :: none

   菜单“Foo子系统”

   取决于HAS\_CONFIGURABLE\_FOO

   配置FOO\_SETTING\_1

   int“Foo设置1”

   默认1

   配置FOO\_SETTING\_2

   int“Foo设置2”

   默认2

   endmenu

当``HAS\_CONFIGURABLE\_FOO``是``n``时，没有为``FOO\_SETTING\_1``和``FOO\_SETTING\_2``生成配置输出，因为上面的代码在逻辑上等同于以下代码：

.. code-block :: none

   配置FOO\_SETTING\_1

   int“Foo设置1”

   默认1

   取决于HAS\_CONFIGURABLE\_FOO

   配置FOO\_SETTING\_2

   int“Foo设置2”

   默认2

   取决于HAS\_CONFIGURABLE\_FOO

如果我们希望符号仍然得到它们的默认值，即使``HAS\_CONFIGURABLE\_FOO``是``n``，但是用户也无法配置，那么我们可以使用``visible if``代替：

.. code-block :: none

   菜单“Foo子系统”

   如果HAS\_CONFIGURABLE\_FOO可见

   配置FOO\_SETTING\_1

   int“Foo设置1”

   默认1

   配置FOO\_SETTING\_2

   int“Foo设置2”

   默认2

   endmenu

这在逻辑上等同于以下内容：

.. code-block :: none

   配置FOO\_SETTING\_1

   int“Foo设置1”如果HAS\_CONFIGURABLE\_FOO

   默认1

   配置FOO\_SETTING\_2

   如果HAS\_CONFIGURABLE\_FOO，则为“Foo设置2”

   默认2

.. 注意：：

   有关提示条件的含义，请参阅“可选提示`”部分。

当``HAS\_CONFIGURABLE``是``n``时，我们现在得到符号的以下配置输出，而不是没有输出：

.. code-block :: none

   ...

   CONFIG\_FOO\_SETTING\_1 = 1

   CONFIG\_FOO\_SETTING\_2 = 2

   ...

.. \_kconfig-functions：

Kconfig函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Kconfiglib提供我们在Zephyr中使用的用户定义的预处理器函数，以将设备树信息公开给Kconfig。例如，我们可以从设备树中获取Kconfig符号的默认值。

设备树相关功能

=============================

.. code-block :: none

  dt\_int\_val（kconf，\_，name，unit）：

       此函数在DTS生成的“conf”样式数据库中查找“name”（<build\_dir> / zephyr / include / generated /中的generated\_dts\_board.conf）

       如果发现它将以十进制整数形式返回值。该函数将根据'unit'划分值：

           没有没有分裂

           'k'或'K'除以1024（1 << 10）

           'm'或'M'除以1,048,576（1 << 20）

           'g'或'G'除以1,073,741,824（1 << 30）

  dt\_hex\_val（kconf，\_，name，unit）：

       此函数在DTS生成的“conf”样式数据库（<build\_dir> / zephyr / include / generated /中的generated\_dts\_board.conf）中查找“name”，如果发现它将以十六进制整数形式返回该值。该函数将根据'unit'划分值：

           没有没有分裂

           'k'或'K'除以1024（1 << 10）

           'm'或'M'除以1,048,576（1 << 20）

           'g'或'G'除以1,073,741,824（1 << 30）

  dt\_str\_val（kconf，\_，name）：

       此函数在DTS生成的“conf”样式数据库（<build\_dir> / zephyr / include / generated /中的generated\_dts\_board.conf）中查找“name”，如果发现它将返回值为string。如果没有找到，我们返回一个空字符串。

示例用法

=============

以下示例显示了``dt\_int\_val``函数的用法：

.. code-block :: none

   板/臂/ mimxrt1020\_evk / Kconfig.defconfig

   配置FLASH\_SIZE

      默认$（dt\_int\_val，DT\_NXP\_IMX\_FLEXSPI\_402A8000\_SIZE\_1，K）

在此示例中，如果我们将生成的generated\_dts\_board.conf文件作为Zephyr构建的一部分进行检查，我们将找到以下条目：

.. code-block :: none

   DT\_NXP\_IMX\_FLEXSPI\_402A8000\_SIZE\_1 = 8388608

``dt\_int\_val``将搜索从板的dts派生的generated\_dts\_board.conf，并匹配``DT\_NXP\_IMX\_FLEXSPI\_402A8000\_SIZE\_1``条目。

该函数比将比例值“1024”更大。这有效导致上述情况如下：

.. code-block :: none

   配置FLASH\_SIZE

      默认8192

其他资源

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

`Kconfiglib docstring中的\*符号值简介\*部分

<https://github.com/ulfalizer/Kconfiglib/blob/master/kconfiglib.py>`\_详细说明符号值的计算方式。