

青风带你玩蓝牙 nRF52832 系列教程.....	3
-----作者: 青风.....	3
作者: 青风.....	4
出品论坛: www.qfv8.com	4
淘宝店: http://qfv5.taobao.com	4
QQ 技术群: 346518370.....	4
硬件平台: 青云 QY-nRF52832 开发板.....	4
2.29 蓝牙 FLASH 存储之 FDS.....	4
1 蓝牙 FLASH 存储 FDS 模块介绍.....	4
2 FDS 库函数 API 详解.....	5
3 FDS 方式编程方法.....	10
3.1 FDS 配置参数.....	10
3.2 FDS 配置流程.....	11
4 应用与调试.....	19
4.1 下载.....	19
4.2 测试.....	19

青风带你玩蓝牙 nRF52832 系列教程.....	3
-----作者: 青风.....	3
作者: 青风.....	4
出品论坛: www.qfv8.com	4
淘宝店: http://qfv5.taobao.com	4
QQ 技术群: 346518370.....	4
硬件平台: 青云 QY-nRF52832 开发板.....	4
2.29 蓝牙 FLASH 存储之 FDS.....	4
1 蓝牙 FLASH 存储 FDS 模块介绍.....	4
2 FDS 库函数 API 详解.....	5
3 FDS 方式编程方法.....	10
3.1 FDS 配置参数.....	10
3.2 FDS 配置流程.....	11
4 应用与调试.....	19
4.1 下载.....	19
4.2 测试.....	19

青风带你玩蓝牙 nRF52832 系列教程

-----作者: 青风

出品论坛: www.qfv8.com 青风电子社区



作者: 青风

出品论坛: www.qfv8.com

淘宝店: <http://qfv5.taobao.com>

QQ 技术群: 346518370

硬件平台: 青云 QY-nRF52832 开发板

2.29 蓝牙 FLASH 存储之 FDS

很多朋友和客户希望能够实现内部的 FLASH 的存储, 内部 FLASH 的存储官方提供两种方式, 一种就是 **fstorage** 方式, 另外一种就是在第一次方式基础上的 **fds** 方式。本讲将深入探讨第二种方式 **fds** 存储

这里我们通过一个简单的例子: 蓝牙 BLE 的 FLASH 的 FDS 存储, 来进行一个简单的思路验证。注意本例在蓝牙串口的基础上进行修改。

1 蓝牙 FLASH 存储 FDS 模块介绍

Flash 数据存储 (FDS) 模块是芯片上闪存的最小化文件系统, 它可以最小化数据损坏的风险, 并简化了与持久存储的交互。它通过在文件中组织数据来实现这一点, 这些数据由一个或多个**记录**组成。这些**记录**包含实际的数据, 可以被写入、删除、更新或检索。

将数据作为文件处理的概念, 提供了一个高水平的抽象方案。您可以使用 **FDS** 模块, 而不需要详细了解内部使用的实际数据格式。相反, 您可以只处理文件和记录, 并将**模块用作黑盒**。

该 FDS 模块化的设计方法会提供以下好处:

- 通过不断的验证的方式来最小化访问损坏数据的风险: 在掉电后, 数据可能没有被完全写入, 这时认为数据为无效数据, 通过验证来确保 **FDS** 识别无效的数据, 并且不将损坏的数据返回给用户。

- 在打开记录时提供 (可选的) **CRC** 校验, 以确保数据写入以后没有发生变化。

- 最小化 **flash** 操作 (更新和删除): **FDS** 没有删除完整的页面的操作, 而是先存储新数据的副本, 然后通过一个字写操作来使旧的数据失效。

- 基本的损耗均衡: 内部 **FLASH** 的写入次数虽然非常的多, 但是, 如果超过写入次数限制也会损坏 **FLASH**, **FDS** 模式通过顺序写和垃圾收集提供了一个平均相当的

flash 的使用级别,也就是把读写均匀的分布在内部 FLASH 空间上,使得不会出现哪块位置由于过度的读写造成损坏。

●在不需要复制数据的情况下可以轻松访问数据,这使得访问数据的影响与数据的大小无关。

●通过灵活选择数据块的大小来最小化的使用内存,而不是需要存一个固定长度。

●对数据的内容不加限制(这意味着它可以包含特殊字符)。

FDS 使用实验性的:闪存作为后端写入 Flash。Flash 存储反过来依赖于软设备来执行写入。Flash 数据存储支持同步读操作和异步写操作。

2 FDS 库函数 API 详解

FDS 已经是脱离了底层的内部 FLASH 读写方式,这种方式需要通过官方提供的 FDS 驱动库编写。那么首先我们就来认识下 FDS 驱动库提供的 API 函数:

1: FDS 注册函数 `fds_register`

函数: `ret_code_t fds_register(fds_cb_t cb);`

*功能: 功能是注册一个 FDS 事件回调处理函数。

* 你可以注册最大数量的回调函数,这个最大数量的值可以通过配置 `fds_config.h` 文件中的参数 `FDS_MAX_USERS` 来实现。

* 参数: `cb` 事件回调处理功能

*返回值: `FDS_SUCCESS` 如果回调处理函数注册成功

*返回值: `FDS_ERR_USER_LIMIT_REACHED` 如何达到可最大数量的回调注册函数

2: FDS 统计函数 `fds_stat`

函数: `ret_code_t fds_stat(fds_stat_t * p_stat);`

*功能: 检索文件系统统计信息的函数

* 这个函数检索文件系统统计信息,例如打开记录的数量,可以通过垃圾收集回收的空间,等等。

* 参数[out] `p_stat` 文件系统统计数据。

* 返回值: `FDS_SUCCESS` 如果统计数据被成功返回。

* 返回值: `FDS_ERR_NOT_INITIALIZED` 如果模块没有初始化。

* 返回值: FDS_ERR_NULL_ARG

如果参数 p_stat 是空的.

3: FDS 初始化函数 fds_init

函数: ret_code_t fds_init(void);

*功能 初始化功能模块组件 Function for initializing the module.

*这个功能是初始化功能组件和安装文件系统（除非文件系统已经安装）

* 这个功能是异步的，完成报告通过事件来实现。在你调用函数 **fds_init** 初始化之前，确保首先调用 **fds_register** 函数，这样你就能收到完成事件。

* 返回值 FDS_SUCCESS 如果操作被成功排队。

* 返回值 FDS_ERR_NO_PAGES 如果在闪存 flash 中没有可用的空间来安装文件系统。

4:FDS 记录搜索函数 fds_record_find

函数: ret_code_t fds_record_find(uint16_t file_id,
uint16_t record_key,
fds_record_desc_t * p_desc,
fds_find_token_t * p_token);

*功能描述: 在文件中通过给定记录键 **key** 搜索记录的函数。

*这个函数的功能找到了具有给定记录键 **key** 的文件中的第一个记录。要在文件中使用相同的键 **key** 搜索下一个记录，再次调用该函数并提供相同的功能。从最后的记录中恢复搜索的 **fds_find_token_t** 结构。

* 参数[in] file_id 文件标识 ID。

* 参数[in] record_key 记录键 key。

* 参数[out] p_desc 找到的记录的描述符。

* 参数[out] p_token 包含关于操作进展的信息的记录。

* 返回值: FDS_SUCCESS 如果找到一个记录

* 返回值: FDS_ERR_NOT_INITIALIZED 如果模块没有初始化。

* 返回值: FDS_ERR_NULL_ARG 如果参数 p_desc 或者参数 p_token 是空的.

* 返回值: FDS_ERR_NOT_FOUND 如果没有找到匹配的记录

5: FDS 记录删除函数 `fds_record_delete`

函数: `ret_code_t fds_record_delete(fds_record_desc_t * p_desc);`

*功能描述 删除记录的函数

* 删除的记录不能使用 参数 `fds_record_find`, 参数 `fds_record_find_by_key`, or 参数 `fds_record_find_in_file`. 此外, 他们不能再使用 参数 `fds_record_open`.

*注意, 删除一个记录并不能释放它在闪存中占用的空间。要回收被删除的记录所使用的 `flash` 空间, 调用 `@ref fds_gc` 去进行垃圾回收。

*这个函数是异步的。完成是通过一个事件报告的, 该事件被发送到注册的事件处理程序函数。

* 参数[in] `p_desc` 应该删除的记录的描述符。

*返回值: `FDS_SUCCESS` 如果操作成功地排队。

*返回值: `FDS_ERR_NOT_INITIALIZED` 如果模块没有初始化。

*返回值: `FDS_ERR_NULL_ARG` 如果指定的记录描述符参数 `p_desc` 为空。

*返回值: `FDS_ERR_NO_SPACE_IN_QUEUES` 如果操作队列满了。

6.FDS 记录写入函数 `fds_record_write`

函数: `ret_code_t fds_record_write(fds_record_desc_t * p_desc,
fds_record_t const * p_record);`

功能描述 : 写入一个记录到闪存的函数。

* 这个函数对于文件 ID 和记录键 `key` 没有限制,除了记录键必须不同于参数 `FDS_RECORD_KEY_DIRTY` 和文件 ID 必须不同于参数 `FDS_FILE_ID_INVALID`。

* 特别地, 对于文件 ID 或记录键 `key` 的唯一性没有任何限制。所有具有相同文件 ID 的记录都被分组到一个文件中。如果没有带有指定 ID 的文件, 则会创建它。在一个文件中可以有多个记录, 具有相同的记录键 `key`。

*有些模块需要使用特定的文件 id 和记录键 `key`。看参数 `lib_fds_functionality_keys` 相关详细信息。

* 记录数据可以由多个块组成。数据必须与 4 字节的边界对齐, 并且由于它没有内部缓冲, 所以必须将其保存在内存中, 直到收到操作的回调为止。数据的长度不能超过参数 `FDS_VIRTUAL_PAGE_SIZE` 的字长度减去 14 字节。

*这个函数是异步的。完成是通过一个事件报告的, 该事件被发送到注册的事件处理程序函数。

- * 参数[out] p_desc 所写的记录的描述符。如果您不需要描述符，则传递 NULL。
- * 参数[in] p_record 将被写入 flash 的记录。

- * 返回值: FDS_SUCCESS 如果操作成功地排队。
- * 返回值: FDS_ERR_NOT_INITIALIZED 如果模块没有初始化。
- * 返回值: FDS_ERR_NULL_ARG 如果参数 p_record 是空。
- * 返回值: FDS_ERR_INVALID_ARG 如果文件 ID 或记录键无效。
- * 返回值: FDS_ERR_UNALIGNED_ADDR 如果记录数据不对齐到 4 字节的边界。
- * 返回值: FDS_ERR_RECORD_TOO_LARGE 如果记录数据超过了最大长度。
- * 返回值: FDS_ERR_NO_SPACE_IN_QUEUES 如果操作队列满了，或者有更多的记录块，而不是可以缓冲的。
- * 返回值: FDS_ERR_NO_SPACE_IN_FLASH 如果在 flash 中没有足够的空闲空间来存储记录。

7.FDS 的记录阅读函数 fds_record_open

函数: ret_code_t fds_record_open(fds_record_desc_t * p_desc,
fds_flash_record_t * p_flash_record);

功能描述 用于打开阅读记录的功能。

* 这个函数打开一个存储在 flash 中的记录，这样就可以读取它了。这个函数初始化一个参数 fds_flash_record_t 结构，它可以用来访问记录数据以及它的相关元数据。在参数 fds_flash_record_t 结构中提供的指针是指向闪存的指针。

* 用操作 fds_record_open 打开一个记录，可以防止垃圾收集在存储记录的虚拟闪存页面上运行，这样，只要记录保持打开，就保证在参数 fds_flash_record_t 中由字段指向的内存内容不会被修改。

* 当您完成读取记录时，请调用操作 fds_record_close 关闭它。垃圾收集可以在记录存储的虚拟页面上回收空间。请注意，您必须为操作 fds_record_close 提供相同的描述符，就像您为本函数所做的那样。

- * 参数[in] p_desc 打开的记录的描述符。
- * 参数[out] p_flash_record 记录，存储在 flash 中。

- * 返回值: FDS_SUCCESS 如果记录成功打开。
- * 返回值: FDS_ERR_NULL_ARG 如果参数 p_desc 或者参数 p_flash_record 是空的。
- * 返回值: FDS_ERR_NOT_FOUND 如果没有找到记录。它可能已经被删除了，或者它可能

还没有被写出来。

* 返回值: `FDS_ERR_CRC_CHECK_FAILED` 如果对记录的 CRC 校验失败了。

8.FDS 记录关闭函数 `fds_record_close`

函数: `ret_code_t fds_record_close(fds_record_desc_t * p_desc);`

功能描述 闭合记录的功能

* 关闭一个记录可以让垃圾收集在记录存储的虚拟页面上运行（如果该页面上没有其他记录）。作为一个参数传递的描述符必须与使用操作 `fds_record_open` 打开记录的那个相同。

* 请注意，关闭一个记录并不能使其描述符无效。您仍然可以向所有接受记录描述符作为参数的函数提供描述符。

* 参数[in] `p_desc` 记录的描述符被关闭。

* 返回值: `FDS_SUCCESS` 如果记录被成功关闭。

* 返回值: `FDS_ERR_NULL_ARG` 如果参数 `p_desc` 为空的。

* 返回值: `FDS_ERR_NO_OPEN_RECORDS` 如果记录没有打开。

* 返回值: `FDS_ERR_NOT_FOUND` 如果无法找到记录。

9.FDS 记录更新函数 `fds_record_update`

函数: `ret_code_t fds_record_update(fds_record_desc_t * p_desc, fds_record_t const * p_record);`

功能描述: 用于更新记录的函数。

* 更新一个记录首先会写一个新的记录（参数 `precord`）到 `flash`，然后删除旧记录（由参数 `p_desc` 识别）。

* 文件 ID 和记录钥匙 `key` 没有任何限制，除了记录键必须与参数 `FDS_RECORD_KEY_DIRTY` 不同，并且文件 ID 必须与参数 `FDS_FILE_ID_INVALID` 不同。特别地，对于文件 ID 或记录钥匙 `key` 的唯一性没有任何限制。所有具有相同文件 ID 的记录都被分组到一个文件中。如果没有带有指定 ID 的文件，则会创建它。在一个文件中可以有多个记录，具有相同的记录键。

* 记录数据可以由多个块组成。数据必须与 4 字节的边界对齐(字对齐)，并且由于它没有内部缓冲，所以必须将其保存在内存中，直到收到操作的回调为止。数据的长度不能超过参数 `FDS_VIRTUAL_PAGE_SIZE` 的定义字长减去 14 字节。

* 这个函数是异步的。完成是通过一个事件报告的，该事件被发送到注册的事件处理程序函数。

* 参数[in, out] `p_desc` 记录更新的描述符。当函数返回 `FDS_SUCCESS` 时，该参数包含

新写入记录的描述符。

* 参数[in] p_record 更新的记录将被写入 flash。

* 返回值: FDS_SUCCESS 如果操作成功地排队。

* 返回值: FDS_ERR_NOT_INITIALIZED 如果模块没有初始化。

* 返回值: FDS_ERR_INVALID_ARG 如果文件 ID 或记录键 key 无效。

* 返回值: FDS_ERR_UNALIGNED_ADDR 如果记录数据不对齐到 4 字节的边界（不是字对齐）。

* 返回值: FDS_ERR_RECORD_TOO_LARGE 如果记录数据超过了最大长度。

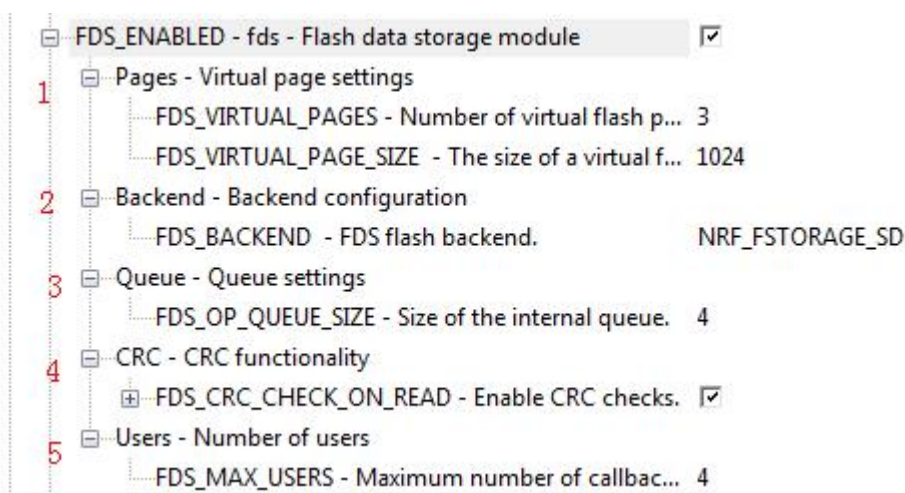
* 返回值: FDS_ERR_NO_SPACE_IN_QUEUES 如果操作队列满了，或者有更多的记录块，而不是可以缓冲的。

* 返回值: FDS_ERR_NO_SPACE_IN_FLASH 如果 flash 中没有足够的空闲空间来存储更新的记录。

3 FDS 方式编程方法

3.1 FDS 配置参数

对应 FDS 初始化的时候实际上需要首先在 sdk_config.h 文件中对 FDS 进行使能，同时配置下面 5 种参数，我们先进行介绍下：



◎#define FDS_ENABLED 1

对 FDS 进行使能，在实现 FDS 库函数之前，需要首先将其设置为 1。

◎#define FDS_VIRTUAL_PAGES 3

#define FDS_VIRTUAL_PAGE_SIZE 1024

这两个参数用于配置要使用的虚拟页面数量及其大小。

FDS_VIRTUAL_PAGES -要使用的虚拟 flash 页面的数量。系统为垃圾收集预留了一个虚拟页面。因此，最少是两个虚拟页面:一个用于存储数据的页面和一个用于系统垃圾收集的页面。FDS 使用的闪存总量为@ref FDS_VIRTUAL_PAGES * @ref FDS_VIRTUAL_PAGE_SIZE * 4 字节

FDS_VIRTUAL_PAGE_SIZE -虚拟 flash 页面的大小。用 4 字节的倍数表示。默认情况下，虚拟页面的大小与物理页面相同。虚拟页面的大小必须是物理页面大小的倍数。

◎ **#define FDS_BACKEND 2**

配置 nrf_fstorage 后台被 FDS 模式用于写入 flash。

参数选择为 NRF_FSTORAGE_NVMC 时，FDS_BACKEND 定义为 1

参数选择为 NRF_FSTORAGE_SD 时，FDS_BACKEND 定义为 2

使用 SoftDevice API 使用 NRF_FSTORAGE_SD 实现后端，如果你使用协议栈的时候，就用这个。

使用外设的时候使用 nrf_fstorage_nvmc 实现后端，如果你没有使用协议栈的时候，就用这个。

◎**#define FDS_OP_QUEUE_SIZE 4**

内部队列的大小。如果您经常得到同步的 FDS_ERR_NO_SPACE_IN_QUEUES 错误，请增加这个值。

◎**#define FDS_CRC_CHECK_ON_READ 1**

#define FDS_CRC_CHECK_ON_WRITE 0

FDS_CRC_CHECK_ON_READ -使能 CRC 检查。当记录写入闪存时保存记录的 CRC，并在记录打开时检查它。使用 FDS 函数的用户仍然可以“看到”不正确的 CRC 记录，但是不能打开它们。此外，它们在被删除之前不会被垃圾收集。

FDS_CRC_CHECK_ON_WRITE -对新记录进行 CRC 检查。此设置可用于确保记录数据在写入 flash 时不会发生更改。

◎**#define FDS_MAX_USERS 4**

可以注册的回调的最大数量。

3.2 FDS 配置流程

第一步： FDS 配置首先是注册一个 FDS 事件回调处理函数。 你可以注册最大数量的回调函数，这个最大数量的值可以通过前面配置 fds_config.h 文件中的参数 FDS_MAX_USERS 来实现。

第二步：就是初始化 FDS 功能组件和安装文件系统，FDS 内部实际上是把文件存储和读取当做类似的一个 filesystem 文件系统处理。这个功能是异步的，完成报告通过事件来实现。

FDS_EVT_INIT 事件报告初始化完成，完成后我们把标志位 m_fds_initialized 置位。因此需要通过判断 m_fds_initialized 是否置位来判断初始化是否成功，没成功之前系统代码，处于 system on 状态。

(注意: 在你调用函数 `fds_init` 初始化之前, 确保首先调用 `fds_register` 函数, 这样你就能收到完成事件。)

第三步: 用 `fds_stat` 函数检索文件系统统计信息, 例如打开记录的数量, 可以通过垃圾收集回收的空间等。这些数据提供一个结构体, 其结构如下所示, 可以通过这个函数观察此时 FDS 所处的状态:

```
01. typedef struct
02. {
03.     uint16_t pages_available;    //可用页数
04.     uint16_t open_records;      //打开记录的数量
05.     uint16_t valid_records;     //有效记录的数量
06.     uint16_t dirty_records;     //删除(“脏”)记录的数量
07.     uint16_t words_reserved;    //通过参数函数 fds_reserve()保留的字数.
08.     uint16_t words_used;        //给 flash 写的字的数量, 包括那些预留给将来写的字
09.     //该参数表示文件系统中最大的自由连续字数。这个数字表示 FDS 可以存储的最大记录。
10.     //它考虑了为将来写所预留的空间。
11.     uint16_t largest_contig;
12.     //碎片回收后可以释放的最大的字节数。如果在运行垃圾收集时, 记录是打开的, 那么碎片
    收集后释放的实际空间数量可能小于此值。
13.     uint16_t freeable_words;
14.     //检测到文件系统损坏。检测到一个或多个损坏的记录。FDS 将自动修复文件系统,
15.     下次碎片收集时, 可能会丢失一些数据。此标志与 CRC 是否正确无关
16.     bool corruption;
17. } fds_stat_t;
```

第四步: 开始测试。可以先使用函数 `fds_record_find` 找到指定记录, 再使用 `fds_record_delete` 先清除所有记录。然后再声明记录描述符 `fds_record_desc_t` 和操作进展的信息的令牌 `fds_find_token_t`。这两个描述符简单进行说明:

功能: 用来操作记录的记录描述符结构。FDS 模式使用这种结构。在操作现有记录时, 必须向 FDS 模块提供描述符。但是, 您不应该修改它或使用它的任何字段。永远不要为不同的记录重用相同的描述符。

typedef struct

```
{
    uint32_t      record_id;        //唯一的记录 ID。
    uint32_t const * p_record;      //在 flash 中最后已知的记录的位置。
    uint16_t      gc_run_count;     //运行碎片收集的次数
    bool          record_is_open;   //记录是否现在打开。
} fds_record_desc_t;
```

功能: 一个令牌, 用来保存关于 `fds_record_find` 进程的信息, `fds_record_find_by_key` 和

`fds_record_find_in_file`总是在第一次使用令牌之前对其进行零初始化。永远不要重用相同的令牌来搜索不同的记录。

```
typedef struct
{
    uint32_t const * p_addr;

    uint16_t      page;
} fds_find_token_t;
```

这两个参数是用来标志操作记录的和操作的进程。表征了本次操作，用于后面整体打开、删除、关闭本次记录所使用。

测试 FDS 模块的数据流程就是：写入数据，然后查找数据，找到对应记录后读取数据，对比写入和读出的数据是否正确。

1. 写入调用 FDS 记录写入 API 函数 `fds_record_write`。该函数的工作就是向前面声明的操作记录写入对应的数据。如何使用可以参考前面 API 函数的说明，这里着重讨论写入数据的格式，写入数据格式以结构体 `fds_record_t` 表征，该结构体如下所示：

```
/*要写入到 flash 中的记录*/
typedef struct
{
    uint16_t file_id;          //记录所属的文件的 ID 一个
    uint16_t key;              //记录 key.

    struct
    {
        void      const * p_data;//数据指针
        uint32_t   length_words;//数据长度
    } data;//数据
} fds_record_t;
```

对于文件 ID 或记录键 `key` 的唯一性没有任何限制。所有具有相同文件 ID 的记录都被分组到一个文件中。在一个文件中可以有多个记录，具有相同的记录键 `key`。记录数据可以由多个块组成。但数据必须与 4 字节的边界对齐，并且由于它没有内部缓冲，所以必须将其保存在内存中，直到收到操作的回调为止。数据的长度不能超过参数 `FDS_VIRTUAL_PAGE_SIZE` 的字长度减去 14 字节。所有写入之前需要对数据结构进行配置，代码如下：

```
01. static fds_record_t const m_dummy_record =
02. {
```

```

03.     .file_id = CONFIG_FILE,//文件 ID
04.     .key     = CONFIG_REC_KEY,//记录钥匙
05.     .data.p_data= &m_dummy_cfg;//仿真数据配置
06.     /* The length of a record is always expressed in 4-byte units (words). 数据长度一般超过 4 个字
    节*/
07.     .data.length_words = (sizeof(m_dummy_cfg) + 3) / sizeof(uint32_t),
08. };

```

2. 查找数据，调用 FDS 记录搜索 API 函数 `fds_record_find`。这个函数的功能是找到给定文件标识 `file_id` 和 `record_key` 记录键 `key` 的文件系统中的第一个记录。要在文件中使用相同的键 `key` 搜索下一个记录，再次调用该函数并提供相同的功能。从最后的记录中恢复搜索的 `fds_find_token_t` 结构。
3. 如果成功找到记录，则使用函数 `fds_record_open` 打开记录。 这个函数打开一个存储在 `flash` 中的记录，这样就可以读取它了。记录的数据以及它的相关元数据放到一个参数 `fds_flash_record_t` 结构指针函数中。在参数 `fds_flash_record_t` 结构中提供的指针是指向闪存的指针。用 `fds_record_open` 操作打开一个记录，可以防止碎片收集在存储记录的虚拟闪存页面上运行破坏数据，只要记录保持打开，就保证在参数 `fds_flash_record_t` 中由字段指向的内存内容不会被修改。当您完成读取记录时，请调用操作 `fds_record_close` 关闭它。此时碎片收集可以在记录存储的虚拟页面上回收空间。请注意，您必须为操作 `fds_record_close` 提供相同的描述符，就像你执行 `fds_record_open` 函数那样。下面谈下 `fds_flash_record_t` 结构体，对比存入的数据结构体 `fds_record_t`。

/@brief** 可用于读取存储在 `flash` 中的记录内容的结构。这个结构没有反映 `flash` 中记录的物理布局，但是它指向了记录头(元数据)和记录数据存储的位置。*/

typedef struct

{

 fds_header_t const * p_header; //记录头在 `flash` 中的位置

 void const * p_data; //记录数据在 `flash` 中的位置

} fds_flash_record_t;

/@brief** 存储在 `flash` 中的记录数据单元格式。警告：不要编辑或重排此结构中的字段 */

typedef struct

{

 uint16_t record_key; //记录密钥 `key`

 uint16_t length_words; //记录数据的长度 (in 4-byte words).

 uint16_t file_id; //记录所属的文件的 ID

 uint16_t crc16; //CRC16 效验检测值.

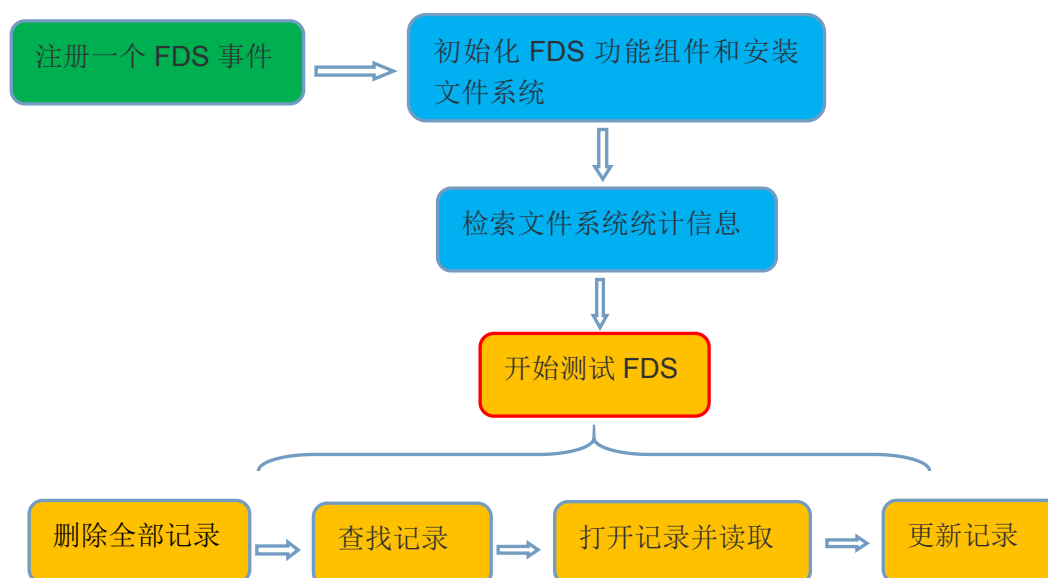

```
/*CRC 是计算效验整个存储在 flash 中的记录, 除了 CRC 字段本身  
之外, 还包括记录元数据。 */
```

```
uint32_t record_id;    //唯一记录 ID (32 bits).  
} fds_header_t;
```

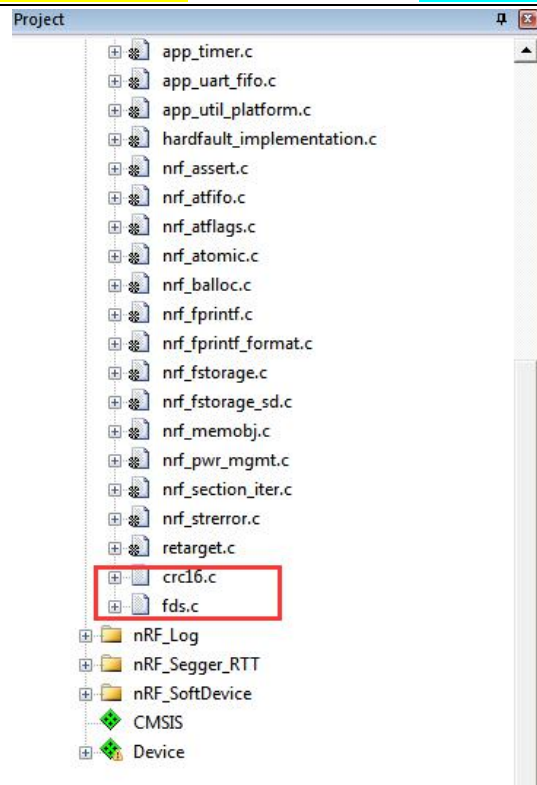
写入的 `fds_flash_record_t` 结构体数据对比存入的 `fds_record_t` 数据结构体数据, 会发现写入的 `fds_flash_record_t` 结构体数据的 `*p_data`、`length_words`、`record_key`、`file_id` 参数全部都会存储到存入的 `fds_record_t` 数据结构体数据内, 同时 `fds_record_t` 数据中会加入 `record_id` 记录描述符内的记录 ID、`crc16` 传输效验。

- 使用 `fds_record_update` 函数对 FDS 记录进行更新。更新一个记录首先会写一个新的记录 (参数 `precord`) 到 flash, 然后删除旧记录 (由参数 `p_desc` 识别)。

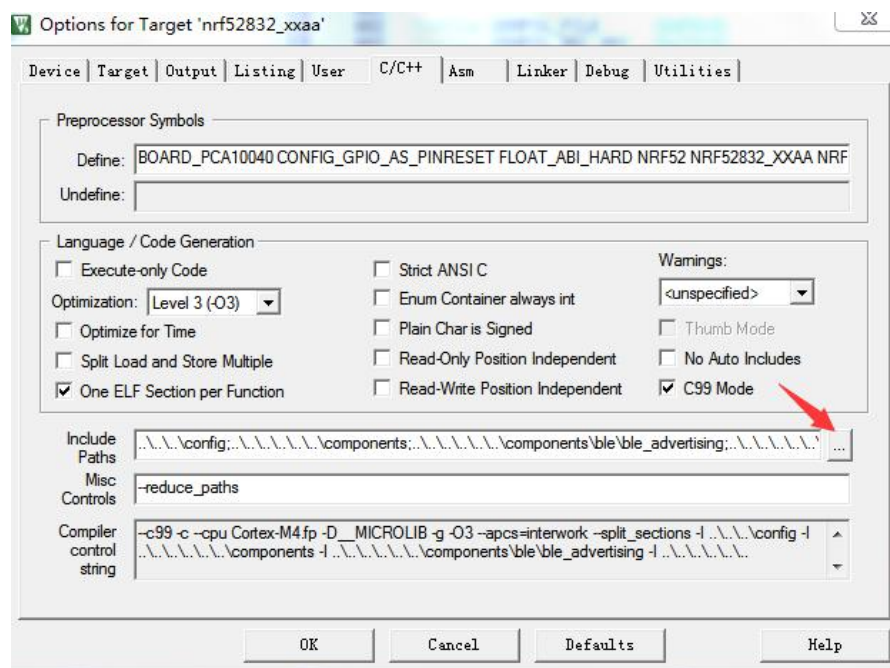
总结整个步骤如下图所示:

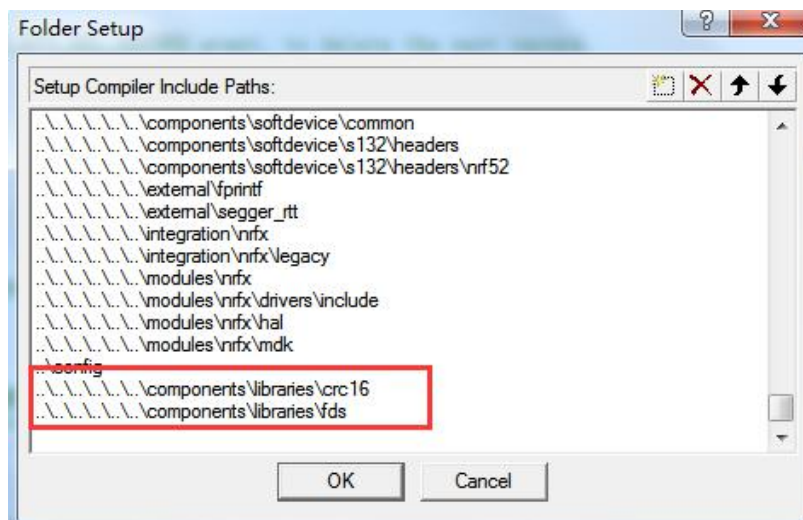


按照上面思路, 我们以蓝牙串口例子为模板, 首先在工程中添加 FDS 的库文件和 CRC 的传输效验文件, 如下图所示:



同时打开工程文件 C/C++ 文件路径设置，添加刚刚加入的文件的路径，如下图所示：





按照上面步骤我们通过编写一个 `flash_test` 测试段程序对设计思路的验证。采用 FDS 方式测试 `flash` 的思路比较简单，具体代码如下所示，下面来进行详细分析：

```

01. void fds_test(void)
02. {
03.     ret_code_t rc;
04.     uint32_t *data;
05.     (void) fds_register(fds_evt_handler); //FDS 注册
06.     rc = fds_init(); //fds 初始化
07.
08.     APP_ERROR_CHECK(rc);
09.
10.     while (!m_fds_initialized) //等待初始化完成
11.     {
12.         sd_app_evt_wait(); //等待过程中待机
13.     }
14.
15.     fds_stat_t stat = {0};
16.     rc = fds_stat(&stat); //设置统计数据
17.     APP_ERROR_CHECK(rc);
18.
19.     record_delete_next(); //把所有记录清零
20.     fds_record_desc_t desc = {0}; //用来操作记录的描述符结构清零
21.     fds_find_token_t tok = {0}; //保存秘钥的令牌清零
22.
23.     rc = fds_record_write(&desc, &m_dummy_record); //写记录和数据
24.     APP_ERROR_CHECK(rc);
25.     //对应 KEY 记录查找数据
26.     rc = fds_record_find(CONFIG_FILE, CONFIG_REC_KEY, &desc, &tok);
27.
28.     if (rc == FDS_SUCCESS) //如果查找成功

```

```
29.     {
30.         /* A config file is in flash. Let's update it. */
31.         fds_flash_record_t config = {0}; //把配置清零
32.
33.         /* Open the record and read its contents. */
34.         rc = fds_record_open(&desc, &config); //打开记录读取数据
35.         APP_ERROR_CHECK(rc);
36.
37.         /* 复制数据到 m_dummy_cfg */
38.         memcpy(&m_dummy_cfg, config.p_data, sizeof(configuration_t));
39.
40.         NRF_LOG_INFO("Found Record ID = %d", desc.record_id);
41.         NRF_LOG_INFO("Data = ");
42.         data = (uint32_t *)config.p_data;
43.         for (uint16_t i=0; i<config.p_header->length_words; i++)
44.         {
45.             NRF_LOG_INFO("0x%8x ", data[i]); //打印输出数据
46.         }
47.         NRF_LOG_INFO("\r\n");
48.
49.         /*当读的时候需要关闭记录. */
50.         rc = fds_record_close(&desc);
51.         APP_ERROR_CHECK(rc);
52.
53.         /* 更新 flash 上的记录*/
54.         rc = fds_record_update(&desc, &m_dummy_record);
55.         APP_ERROR_CHECK(rc);
56.     }
57.     else
58.     {
59.         /* System config not found; write a new one. */
60.         NRF_LOG_INFO("Writing config file...");
61.
62.         rc = fds_record_write(&desc, &m_dummy_record); //重新写记录
63.         APP_ERROR_CHECK(rc);
64.     }
65. }
```

主函数直接调用测试函数，代码如下：

```
66. int main(void)
67. {
68.     bool erase_bonds;
69.
70.     // Initialize.
```

```
71.     uart_init();
72.     .....
73.     .....
74.     fds_test();//fds 测试函数
75.     .....
76.     .....
77.     // Enter main loop.
78.     for (;;)
79.     {
80.         idle_state_handle();
81.
82.     }
83. }
```

4 应用与调试

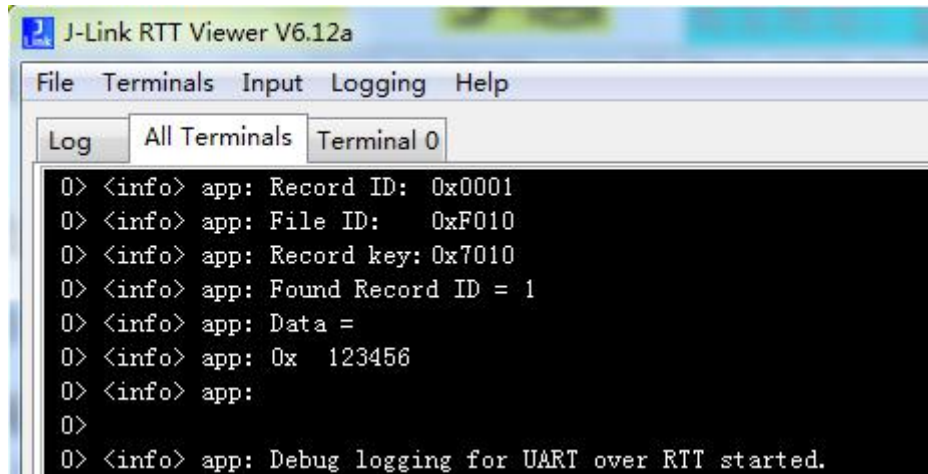
4.1 下载

本例使用的协议栈为 S132 版本，打开 NRFgo 进行下载，可以参考前面的章节。首先整片擦除，后下载协议栈，下载完协议栈后可以下载工程，首先把工程编译一下，通过后点击 KEIL 上的下载按钮。下载成功后提示如图，程序开始运行，同时开发板上广播 LED 开始广播：



4.2 测试

打开 jlink-RTT 观察窗口，可以观察写和读的结果对比，如下图所示，如图两次写入和读的内容相同，则表明写入成功：



The screenshot shows the J-Link RTT Viewer V6.12a interface. The 'Terminal 0' tab is selected, displaying a series of debug logs. The logs indicate that the application has successfully recorded information, including the Record ID (0x0001), File ID (0xF010), and Record key (0x7010). It also shows the Found Record ID (1) and the Data (0x123456). The final log entry states that debug logging for UART over RTT has started.

```
0> <info> app: Record ID: 0x0001
0> <info> app: File ID: 0xF010
0> <info> app: Record key: 0x7010
0> <info> app: Found Record ID = 1
0> <info> app: Data =
0> <info> app: 0x 123456
0> <info> app:
0>
0> <info> app: Debug logging for UART over RTT started.
```