```
 * performed as part of the advertisement.
 *
 * Subscription will fail if the topic is not known to the system, i.e.
 * there is nothing in the system that has defined the topic and thus it
 * can never be published.
 *
 * @param meta          The uORB metadata (usually from the ORB_ID() macro)
 *                      for the topic.
 * @return              ERROR on error, otherwise returns a handle
 *                      that can be used to read and check the topic for
updates.
 *                      If the topic in question is not known (due to an
 *                      ORB_DEFINE_OPTIONAL with no corresponding
ORB_DECLARE)
 *                      this function will return -1 and set errno to ENOENT.
 */
    externint    orb_subscribe(conststruct orb_metadata *meta);
```

如果一个 optional topic 没有被提供，那么对它的订阅将失败，但是别的订阅依然会成功，并且创建这个 topic 即使它还没被通告。这很大程度低降低了系统启动顺序的安排难度。

一个任务中订阅者的数量没有上限。

取消订阅一个 topic，使用如下函数：

```
    /**
 * Unsubscribe from a topic.
 *
 * @param handle        A handle returned from orb_subscribe.
 * @return              OK on success, ERROR otherwise with errno set
accordingly.
 */
    externint    orb_unsubscribe(int handle);
```

从 topic 中复制出数据：订阅者并不直接调用 ORB 中存储的数据，也不直接

与其他订阅者共享它，而是将其从 ORB 中复制到一个临时缓存中。这个复制过程避免了死锁问题，并且使得发布和订阅函数很简单。并且也允许订阅者直接修改数据（如果需要的话）而不影响其他订阅者。

当一个订阅者想复制一份最新的副本时，使用如下函数：

```
    /**
 * Fetch data from a topic.
 *
 * @param meta          The uORB metadata (usually from the ORB() macro)
 *                      for the topic.
 * @param handle        A handle returned from orb_subscribe.
 * @param buffer        Pointer to the buffer receiving the data.
 * @return              OK on success, ERROR otherwise with errno set
accordingly.
 */
    extern int    orb_copy(const struct orb_metadata *meta, int
handle, void *buffer);
```

检查更新：订阅者可以使用如下函数来检查从它上次订阅后，topic 是否被再次发布。

```
    /**
 * Check whether a topic has been published to since the last orb_copy.
 *
 * This check can be used to determine whether to copy from the topic when
 * not using poll(), or to avoid the overhead of calling poll() when the
 * topic is likely to have updated.
 *
 * Updates are tracked on a per-handle basis; this call will continue to
 * return true until orb_copy is called using the same handle. This interface
 * should be preferred over calling orb_stat due to the race window between
 * stat and copy that can lead to missed updates.
 *
 * @param handle        A handle returned from orb_subscribe.
 * @param updated       Set to true if the topic has been published since
the
```

```
 *                      last time it was copied using this handle.
 * @return              OK if the check was successful, ERROR otherwise with
 *                      errno set accordingly.
 */
    extern int    orb_check(int handle, bool *updated);
```

当一个 topic 在它被通告（advertise）之前被发布，这个函数将返回没有更新直到它被通告。

发布时间戳：订阅者可以使用如下函数检查最新的发布发生的时间：

```
    /**
 * Return the last time that the topic was published.
 *
 * @param handle        A handle returned from orb_subscribe.
 * @param time          Returns the time that the topic was published, or
zero if it has
 *                      never been published/advertised.
 * @return              OK on success, ERROR otherwise with errno set
accordingly.
 */
    extern int    orb_stat(int handle,uint64_t*time);
```

调用这个函数前要额外小心，因为无法保证这个 topic 不会在这个函数被调用后立即被发布。

等待更新：一个将发布者发出的信息作为信息来源的订阅者（subscriber）可以同时等待多个发布者的发布。这个和等待一个 file descriptor 一样，也是使用 poll()函数（这是一个标准通用函数,并非 PX4 独有，可以在百度上搜索其用法）这个方法可行，是因为实际上订阅也是一个 file descriptor。

下面的例子展示了一个等待三个发布者的订阅者的情况。如果一秒钟内没有更行出现，则一个 timeout 计数器将被更新并发布出去。

color_counter.h

```
ORB_DECLARE(color_red);
ORB_DECLARE(color_green);
ORB_DECLARE(color_blue);
ORB_DECLARE(color_timeouts);
/* structure published to color_red, color_green, color_blue and color_timeouts */
struct color_update
{
	int number;
};
```

color_counter.c

```
#include <poll.h>
ORB_DEFINE(color_timeouts,struct color_update);

void
subscriber(void)
{
	int	sub_red, sub_green, sub_blue;
	int	pub_timeouts;
	int	timeouts =0;
	struct color_update cu;

	/* subscribe to color topics */
	sub_red = orb_subscribe(ORB_ID(color_red));
	sub_green = orb_subscribe(ORB_ID(color_green));
	sub_blue = orb_subscribe(ORB_ID(color_blue));

	/* advertise the timeout topic */
	cu.number=0;
	pub_timeouts = orb_advertise(ORB_ID(color_timeouts),&cu);

	/* loop waiting for updates */
	for(;;){

		/* wait for updates or a 1-second timeout */
		struct pollfd fds[3]={
			{ .fd= sub_red,   .events= POLLIN },
			{ .fd= sub_green, .events= POLLIN },
			{ .fd= sub_blue,  .events= POLLIN }
		};
		int ret =poll(fds,3,1000);
```

```
                /* check for a timeout */
                if(ret ==0){
                        puts("timeout");
                        cu.number=++timeouts;
                        orb_publish(ORB_ID(color_timeouts), pub_timeouts,&cu);

                /* check for color updates */
                }else{

                        if(fds[0].revents& POLLIN){
                                orb_copy(ORB_ID(color_red), sub_red,&cu);
                                printf("red is now %d\n",cu.number);
                        }
                        if(fds[1].revents& POLLIN){
                                orb_copy(ORB_ID(color_green), sub_green,&cu);
                                printf("green is now %d\n",cu.number);
                        }
                        if(fds[2].revents& POLLIN){
                                orb_copy(ORB_ID(color_blue), sub_blue,&cu);
                                printf("blue is now %d\n",cu.number);
                        }
                }
        }
    }
```

限制更新速率：一个订阅者可能想要限制它们接收的 topic 的更新速率，这个可以通过下面的函数实现：

```
    /**
 * Set the minimum interval between which updates are seen for a subscription.
 *
 * If this interval is set, the subscriber will not see more than one update
 * within the period.
 *
 * Specifically, the first time an update is reported to the subscriber a
timer
 * is started. The update will continue to be reported via poll and orb_check,
but
 * once fetched via orb_copy another update will not be reported until the
timer
 * expires.
```

```
 *
 * This feature can be used to pace a subscriber that is watching a topic
that
 * would otherwise update too quickly.
 *
 * @param handle        A handle returned from orb_subscribe.
 * @param interval      An interval period in milliseconds.
 * @return              OK on success, ERROR otherwise with ERRNO set
accordingly.
 */
    externint    orb_set_interval(int handle,unsigned interval);
```

速率限制是针对某一个特定的订阅者的，单个 topic 可以对多个订阅者有多种限制速率。

添加一个 **uORB topic** 和 **mavlink** 解析程序

参考链接：http://pixhawk.org/dev/add_uorb_topic

步骤一：添加一个 uORB topic

在 src/modules/uORB/topics 文件夹下添加一个新的名为“ca_trajectory_msg.h”的头文件，其内容如下：

```
#ifndef TOPIC_CA_TRAJECTORY_MSG_H
#define TOPIC_CA_TRAJECTORY_MSG_H

#include <stdint.h>
#include "../uORB.h"

/** global 'actuator output is live' control. */
struct ca_traj_struct_s {

        uint64_t timestamp;
        uint64_t time_start_usec;///< starting time of the trajectory.
        uint64_t time_stop_usec;///< stopping time of the trajectory.
        floatcoefficients[28];///< coefficients of the polynomial trajectory.
```

```
        uint16_t seq_id;///< sequence id of the sent trajectory piece.
};

ORB_DECLARE(ca_trajectory_msg);

#endif
```

编辑 src/modules/uORB/objects_common.cpp 文件并添加如下内容：

```
#include "topics/ca_trajectory_msg.h"
ORB_DEFINE(ca_trajectory_msg,struct ca_traj_struct_s);
```

步骤二：添加一个 mavlink 解析程序

这将会将一个输入的 mavlink 消息解析并传入 uORB topic 中。假设 mavlink 消息和 uORB topic 有着相同的结构。(注意 :下文列出的示例程序中 ,行首出现“+”表示在原程序中添加该行，“-”表示删除原程序中该行)

编辑 src/modules/mavlink/mavlink_bridge_header.h

```
-#include <v1.0/common/mavlink.h>
+#include <v1.0/ca_pixhawk/mavlink.h>
```

在 src/modules/mavlink/mavlink_messages.cpp 中添加如下行：

```
+#include <uORB/topics/ca_trajectory_msg.h>
```

在 src/modules/mavlink/mavlink_receiver.h 中添加如下行：

```
+#include <uORB/topics/ca_trajectory_msg.h>

+void handle_message_ca_trajectory_msg(mavlink_message_t *msg);

+orb_advert_t _ca_traj_msg_pub;
```

在 src/modules/mavlink/mavlink_receiver.cpp 中添加下列函数：