JUMPSTART API FOR STM32F0xx and STM32F4xx

(Version: Nov 21 2015)

The design goal of the JumpStart API is to make it simple to access the peripherals of the microcontroller without sacrificing the power specific to each vendor’s offering. Thus, the JumpStart API is not a single API common to all the target devices, but rather it is specific to each device or device family.

This document describes the JumpStart API for the STM32F0xx and the STM32F4xx families. The same API should be mostly compatible with other STM32F families as well (e.g., STM32F1xx, STM32F2xx, ...), and we will be releasing JSAPI for these series shortly. Note that while the JSAPI for STM32F0xx coincides with the release of the JumpStart MicroBox hardware kit, JSAPI for STM32F0xx is usable for all STM32F0xx devices and boards, including the ST Discovery boards.

To fully understand the API, you need to consult the device family reference manual and the device’s datasheets from the vendor. Search terms such as “STM32F0 reference manual” and “STM32F401 datasheet” should locate the right files at http://st.com.

File Structure

To use JumpStart API, your project must do the following:

1. The source file `#include` the header file `jsapi.h`, and `jsapi_ACEShield.h` if it accesses the ACE Shield specific hardware devices. These header files include the class objects and function declarations.
2. The `Build Options->Paths` of the project must include paths to the header files as the locations are device family specific. See below.
3. The project must link with the JumpStart API library in the `Build Options->Target->Additional Lib.` edit box.

STM32F0xx:	file `libjsapi-stmf0.a`, or `jsapi-stmf0` in the edit box
STM32F4xx:	file `libjsapi-stmf4.a`, or `jsapi-stmf4` in the edit box

4. The project must include the interrupt vector file that references the JumpStart API interrupt handlers. Copies of the files are under `c:\iccv8cortex\libsrc.cortex\`. You can either add the file directly to your project, or make a copy of the file and put it in your project file list:

STM32F0xx:	file `stm32f0_vectors.s`
STM32F4xx:	file `stm32f4_vectors.s`

The examples under `c:\iccv8cortex\examples.JumpStartMicroBox\`, are setup correctly and you can use them as reference models.

To configure the path in the CodeBlocks IDE so that the compiler can locate the header file:

1. Invoke `Project->Build Options`, switch to the `Paths` tab.
2. For STM32F0xx devices, add the following:
   ```
   $(TARGET_COMPILER_DIR)\include\jsapi\ST.com-Files\CMSIS\Device\
   ST\STM32F0xx\Include
   $(TARGET_COMPILER_DIR)\include\jsapi\ST.com-Files\StdPeriph_Lib
   \stm32f0xx\inc
   $(TARGET_COMPILER_DIR)\include\jsapi\STM32F0
   $(TARGET_COMPILER_DIR)\include\jsapi\ACEShield
   ```

3. For STM32F4xx devices, add the following:

   ```
   $(TARGET_COMPILER_DIR)\include\jsapi\ST.com-Files\CMSIS\Device\
   ST\STM32F4xx\Include
   ```

```
$(TARGET_COMPILER_DIR)\include\jsapi\ST.com-Files\StdPeriph_Lib
\stm32f4xx\inc
$(TARGET_COMPILER_DIR)\include\jsapi\STM32F4
$(TARGET_COMPILER_DIR)\include\jsapi\ACEShield
```

Note that the header file paths reference the "`ST.com-Files`" directories. They are only needed if you need to access the ST StdPeriphLib in addition to the JumpStart API functions. The source code for the ST StdPeriphLib can be downloaded from st.com. There are copies installed under `C:\iccv8cortex\VendorLibs\ST-StdPeriphLib`[1].

For STM32F0xx family, you can also start a new project based on the "JumpStart MicroBox Template": invoking `File->New->Project…` and select `ImageCraft JumpStart Project`. Choose a directory where you want the project to be located, and the paths and other compiler configurations will be set up correctly for you.

We will create a template for the STM32F4xx family in the near future.

[1] All rights of the StdPeriphLib belong to ST. The files are provided unmodified from ST and provided as a courtesy to the users.

JumpStart API and ST Standard Peripheral Library

JumpStart API is superior to the ST Standard Peripheral Library in that it provides high-level object abstractions (e.g. I2C_WIRE and I2C objects) and it takes care of the low-level tedium (for example, it enables power to the RCC registers without work from the user). It is also written in efficient C code and does not use artificial “init” style structures. By requiring parameters to be specified as function call arguments, users can rely on the CodeBlocks IDE to pop up the function prototype as an aid to recall which parameters are needed.

Also, ImageCraft expects to port the JumpStart API to other Cortex-M devices. While JSAPI is specific to each device family, it still gives a better portability solution than vendor-provided solutions.

Finally, using the JumpStart API does not prevent a user from using ST’s Standard Peripheral Library. You can use the Standard Peripheral Library to utilize the features not supported by JSAPI.

“Doing It the Hard Way”

The JumpStart API hides some of the low-level tedium so that users can start working on more complicated topics faster. For example, the example and tutorial on the LED matrix shows how to light up the LED one at a time in a walking pattern. One of the suggested exercises is to light up the LED matrix creatively. For example, what interesting patterns can you create using the LED matrix? What if you couple it to a button push?

You might also learn the low-level subjects if you wish. Some of the topics you might want to tackle if you want to “do it the hard way” include: implementing the I2C protocol by using the Standard Peripheral Library and, if that is not challenging enough, implementing the I2C protocol by generating the SCL and SDA signals using GPIO pins and timers. You can use the same pins as the dedicated I2C pins so no hardware changes are needed.

Having examples that use JSAPI gives you confidence that “things are working,” which is not often the case. For example, if an LED does not light up, is it because the hardware is not connected correctly, or because the LED is defective? Toward that end, the JSAPI examples serve as reference materials.

Notes on the Examples

NOTE: the JumpStart API uses “C with Classes” to provide better modularization. This just means that API functions are declared as “member functions” of a C struct and they must be called using a variable of that struct type. For example:

```
// in jsapi.h
typedef struct JSAPI_CLOCK {
     …
     _Bool SystemInit(unsigned Mhz);
     …
} JSAPI_CLOCK;

extern JSAPI_CLOCK jsapi_clock;
…

// in your code
#include <jsapi.h>

int main(void)
     {
     // call the SystemInit function to initialize the clock
     // to 48 Mhz
     jsapi_clock.SystemInit(48);
     …
     }
```

API Syntax: In this document, the API functions are described using the C++ classes member function definition syntax. For example:

```
void JSAPI_GPIO::MakeAnalog(unsigned pin_no);
```

In this example, `JSAPI_GPIO::` is the class specifier. To call a member function, use the structure dereference syntax. For example:

```
extern JSAPI_GPIO porta;

porta.MakeAnalog(0);
```

In this example, `porta` is a JSAPI Class object. Most JSAPI classes have predefined objects, corresponding to the actual hardware components on the chip with the same or similar names. Some JSAPI classes are abstractions that do not have predefined objects. In those cases, you would declare variables of that type and then use an API function (usually in the form “MakeXYZ”) to initialize the object.

Return Values: Some functions return a value of the `_Bool` data type. A true value (“1”) signifies success and false (“0”) signifies failure. Some functions return `int` type success/failure codes. In those codes, 0 represents success and error codes are returned as negative values.

NOTE #1: Some examples may use features of JSAPI that will be described later. This chicken-and-egg problem cannot be avoided in these cases.

NOTE #2: The header file may evolve faster than this document. You should always refer to the released header file and the comments therein as the definitive reference document.

Selecting GPIO Pins for a Peripheral Function

Some GPIO pins have additional functions besides acting as digital input or output pins. For example, the STM32F030 I2C peripheral needs two pins: SCL for clock and SDA for bidirectional data. Rather than dedicating two pins for them, some GPIO pins can be configured as an I2C SCL or an I2C SDA pin. To provide flexibility, multiple GPIO pins can be used for these purpose so that the user can select which one is the most suitable in a particular system design.

To see which pin is suitable for a particular function, consult the device datasheet. For example, this is an excerpt of the STM32F030R8 datasheet:

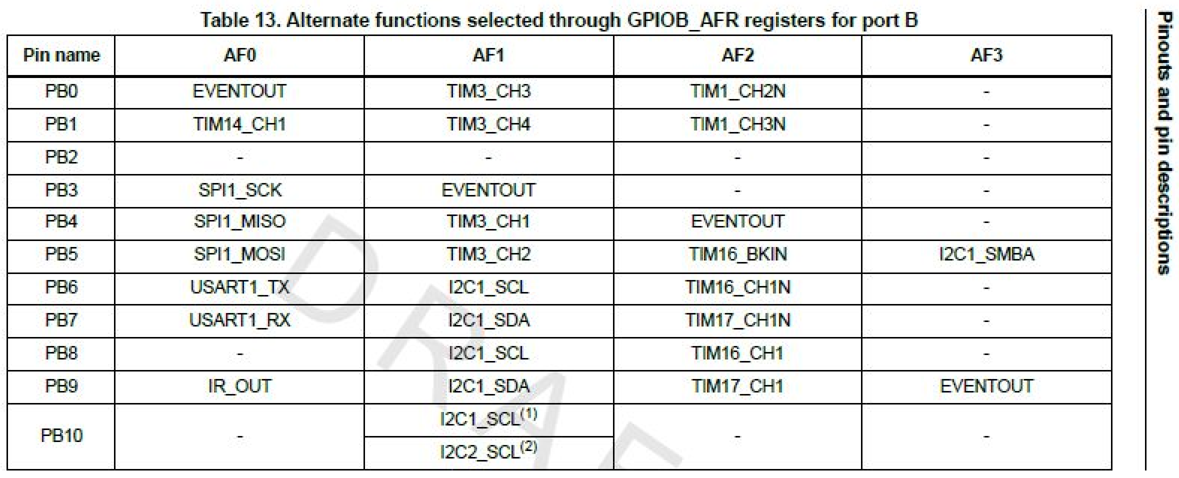


Table 13. Alternate functions selected through GPIOB_AFR registers for port B

Pin name	AF0	AF1	AF2	AF3
PB0	EVENTOUT	TIM3_CH3	TIM1_CH2N	-
PB1	TIM14_CH1	TIM3_CH4	TIM1_CH3N	-
PB2	-	-	-	-
PB3	SPI1_SCK	EVENTOUT	-	-
PB4	SPI1_MISO	TIM3_CH1	EVENTOUT	-
PB5	SPI1_MOSI	TIM3_CH2	TIM16_BKIN	I2C1_SMBA
PB6	USART1_TX	I2C1_SCL	TIM16_CH1N	-
PB7	USART1_RX	I2C1_SDA	TIM17_CH1N	-
PB8	-	I2C1_SCL	TIM16_CH1	-
PB9	IR_OUT	I2C1_SDA	TIM17_CH1	EVENTOUT
PB10	-	I2C1_SCL[1]	-	-
		I2C2_SCL[2]		

Pinouts and pin descriptions

PORTB has 16 I/O pins (like all the other ports in the STM32F030), the first 11, named PB0, PB1, etc., are listed here. Each column lists the *alternate function* that the particular pin can be configured as by using the *alternate function code*.

Thus, to fully specify a pin, you need to specify which port (e.g., PORTA, PORTB, etc.), the pin number (0 to 15), the alternate function code to use if applicable, and whether it's an input or output pin. This is reflected in the parameters to the JSAPI functions.

JSAPI_CLOCK: System Clock

One of the first thing a program must do is to set up the system clock. The STM32F devices can use a variety of internal and external oscillators and PLLs.

JSAPI Object

```
extern JSAPI_CLOCK jsapi_clock;
```

Type (STM32F0)

```
enum CLOCK_SRC { CLK_HSE, CLK_HSI, CLK_PLL, CLK_HSI48 };
```

HSE - High Speed External
HSI - High Speed Internal
PLL - Phase Lock Loop
HSI48 - High Speed Internal 48MHz

Type (STM32F4)

```
enum CLOCK_SRC { CLK_HSI, CLK_HSE, CLK_HSEBYP, CLK_PLL, CLK_PLLR };
```

API (STM32F4)

```
_Bool JSAPI_CLOCK::SetSystemClock(unsigned hsi_mhz, unsigned hse_mhz,
      _Bool hse_bypass, unsigned pll_mhz, unsigned flash_ws);
```

- sets the system clock using PLL if needed. You specify the initial clock (e.g. HSI or HSE) and the speed, and the desired PLL speed, or zero if you do not want to use the PLL. `flash_ws` is the wait state for the flash and is default to 5 when setting the PLL to 84Mhz or above

```
// ahbdiv is the divfactor SYSCLK -> HCLK
// apb1div/apb2div is the divfactor HCLK -> APBxCLK/PCLKx
// tim?mul: 0 - HCLK, otherwise 2 or 4 times PCLKx
void JSAPI_CLOCK::GetClockFactors(int *ahbdiv, int *apb1div,
      int *apb2div, int *tim1mul, int *tim2mul);
```

- Get the clock division factors. See the comments above

API (STM32F0)

```
_Bool JSAPI_CLOCK::SetSystemClock(unsigned hsi_mhz, _Bool hsi_div2,
      unsigned hse_mhz,_Bool hse_bypass, unsigned pll_mhz);
```

- sets the system clock using PLL if needed. You specify the initial clock (e.g. HSI or HSE) and the speed, and the desired PLL speed, or zero if you do not want to use the PLL

```
_Bool JSAPI_CLOCK::SystemInit(unsigned MHz);
```

- // deprecated. `Use SetSystemClock(8, 0, 0, 0, Mhz);`
- sets the system clock to the specified speed in megahertz

```
// ahbdiv is the divfactor SYSCLK -> HCLK
// apbdiv is the divfactor HCLK -> APBCLK/PCLK
// timmul is 1x or 2x PCLK
// null pointers can be passed if they are don't care
void JSAPI_CLOCK::GetClockFactors(int *ahbdiv, int *apbdiv,
     int *timmul);
```

- Get the clock division factors. See the comments above

API (COMMON)

```
unsigned JSAPI_CLOCK::GetSysClkFreq(void);
```

- returns the current system clock frequency in hertz

```
enum CLOCK_SRC JSAPI_CLOCK::GetCurrentClockSource(void);
```

- returns the current clock source

```
void JSAPI_CLOCK::RCC_GetClocksFreq(JSAPI_ClocksTypeDef *RCC_Clocks);
```

- returns various clock frequencies

Examples:

```
     jsapi_clock.SetSystemClock(8, 0, 0, 0, 48);
     printf("System running at %d
     Hz\n",jsapi_clock.GetSysClkFreq());

     static char *name[] = {
          "High Speed External",
          "High Speed Internal",
          "Phase Lock Loop",
          "High Speed Internal 48Mhz"
     };
     printf("clock source is '%s'\n",
                name[jsapi_clock.GetCurrentClockSource()]);
```

JSAPI_CORTEX_CORE: SysTick[2] Timer Functions

The SysTick timer is a simple 24-bit timer defined by ARM that is present in most Cortex-M implementations. It provides a portable method of invoking timer interrupts on any Cortex-M devices that include it.

JSAPI Object

```
extern JSAPI_CORTEX_CORE jsapi_cortex_core;
```

Type

```
enum SYSTICK_GRANULARITY { SYSTICK_10MICROSECOND,
     SYSTICK_100MICROSECOND, SYSTICK_MILLISECOND,
     SYSTICK_HUNDREDTHSECOND };
```

API

```
void JSAPI_CORTEX_CORE::SysTick_TimerConfig(
          enum SYSTICK_GRANULARITY granularity);
```

- sets the systick timer frequency of interrupt (tick).

```
void JSAPI_CORTEX_CORE::SysTick_TimerAddHook(void (*func)(void));
```

- adds a “hook” function.

Examples

```
     jsapi_cortex_core.SysTick_TimerConfig(SYSTICK_MILLISECOND);

     extern void periodic_interrupt(void);
     jsapi_cortex_core.SysTick_TimerAddHook(periodic_interrupt);
```

Notes

Use the `SysTick_TimerConfig` function to specify how fine the resolution of the SysTick timer should be using an `enum SYSTICK_GRANULARITY` value. The value specifies the value of each “tick.” At each tick, the SysTick timer generates an interrupt and JSAPI keeps track of the time passed. The SysTick timer can run as fast a 1 microsecond per “tick.” However, a Cortex-M0 core running at 48 MHz executes fewer than 50 instructions per microsecond and there are time overheads to service an interrupt. Therefore, a 1-microsecond timer is not very useful, and the finest granularity supported by JSAPI is 10 microseconds per tick.

If your program does not need fine resolution, you should choose a larger granularity value to lessen the interrupt overhead on the CPU. A value of `SYSTICK_MILLISECOND` is a good choice, as the STM32F030 can execute up to 48,000 instructions per millisecond.

[2] SysTick is written as SysTick and not Systick due to ARM’s convention.

You can add a “hook function” to the SysTick timer interrupt by calling the function `SysTick_TimerAddHook`. At every SysTick interrupt, the specified function will be called. You might use it to check for some hardware sensors or even implement a clock for an RTOS (Real Time Operating System) using a hook function. Only one hook function is allowed, and a call to `SysTick_TimerAddHook` overwrites any previously specified hook function.

Delay Functions

Delay functions (e.g., a busy wait for a tenth of a second) are commonly used in embedded programming. For example, to blink an LED, the LED should stay on for some amount of time before it is turned off, and a delay function is a simple method to accomplish that. Certain device programming — e.g., the I2C bus — also requires precise timing (although with a complex microcontroller such as the STM32Fxxx device with dedicated I2C circuitry, most of the timing needs for I2C programming are taken care by the microcontroller itself).

NOTE: These are the only JSAPI functions that do not use “C with Classes” syntax, since the Delay functions are generic and can be implemented using other features, such as. “busy loop” or device-specific timers.

API

```
void Delay10MicroSecs(unsigned delay);
void Delay100MicroSecs(unsigned delay);
void DelayMilliSecs(unsigned delay);
void DelayHundredth(unsigned delay);
void DelayTenth(unsigned delay);
void DelaySecs(unsigned delay);
```

The Delay functions busy-wait until the specific time is passed. Of course, it makes no sense to use a delay function with finer resolution than the one you specify with the JSAPI_CLOCK::`SysTick_TimerConfig` function.

For example, a blinking LED can be implemented as:

```
    JSAPI_IOPIN led;

    porta.MakeOutput(5);
    led.MakeIOPin(&porta, 5);

    while (1)
        {
        led.Toggle();
        DelayTenth(5);
        }
```

JSAPI_GPIO: GPIO (General Purpose Input Output) Functions

The GPIO pins on a STM32Fxxx device can be configured individually as input or output. In addition, some can be used for performing an ADC (Analog to Digital Converter) function. Some have alternate functions, such as acting as the SCL clock pin for the I2C module. In fact, a pin might have many different alternate functions, and the same function (e.g., I2C SCL) might be handled by several different pins, allowing flexibility on how the system can be designed.

JSAPI Objects (STM32F0)

```
extern JSAPI_GPIO porta, portb, portc, portd, porte, portf;
```

JSAPI Objects (STM32F4)

```
extern JSAPI_GPIO porta, portb, portc, portd, porte, portf, portg,
     porth, porti;
```

These objects correspond to the hardware GPIO ports of the same names. **NOTE:** not all ports are available on a particular device.

Types

For explanation of pull-up, pull-down, and the output type, please refer to the document <JumpStart MicroBox Hardware>.

```
enum OSPEED { OSPEED_LOW = 0, OSPEED_MED = 1, OSPEED_HIGH = 0x3 };
```

- Output port speed, corresponding to 2 MHz, 10 MHz, and 50 MHz.

```
enum PUPDR  { PUPDR_NONE = 0, PUPDR_UP, PUPDR_DOWN };
```

- Pull-up or Pull-down.

```
enum OTYPE  { OTYPE_PUSHPULL, OTYPE_OPENDRAIN };
```

- Push-pull output or Open Drain output.

API

There are 16 pins for each port.

```
void JSAPI_GPIO::MakeInput(unsigned pin_no);
```

- Sets the specified pin as an input pin.

```
void JSAPI_GPIO::MakeOutput(unsigned pin_no, enum OSPEED speed);
```

- Sets the specified pin as an output pin with the specified speed.

```
void JSAPI_GPIO::MakeAltFunction(unsigned pin_no, unsigned f,
                enum OSPEED ospeed);
```

- Sets the specified pin as an *alternate function pin* with the function code `f` and with the specified speed.

```
void JSAPI_GPIO::MakeAnalog(unsigned pin_no);
```

- Sets the specified pin as an ADC (analog to digital converter) pin. Not all port pins can be used. For the STM32F030, only some of the PORTA pins can be configured as ADC pins.

```
void JSAPI_GPIO::SetOType(unsigned pin_no, enum OTYPE otype);
```

- Sets the output type for the output pin.

```
void JSAPI_GPIO::SetPullUpDown(unsigned pin_no, enum PUPDR pupdr);
```

- Sets the pull-up or pull-down for the pin.

```
void JSAPI_GPIO::Set(unsigned pin_no);
```

- Sets the output pin state to “1”.

```
void JSAPI_GPIO::Clear(unsigned pin_no);
```

- Sets the output pin state to “0”.

```
void JSAPI_GPIO::JiggleLow(unsigned pin_no,
                unsigned millisecs_delay);
```

- Sets the output pin state to “0” for `millisecs_delay` milliseconds, then sets state to “1”. The function does not check the current pin state.

```
void JSAPI_GPIO::JiggleHigh(unsigned pin_no,
                unsigned millisecs_delay);
```

- Sets the output pin state to “1” for `millisecs_delay` milliseconds, then sets state to “0”. The function does not check the current pin state.

```
unsigned JSAPI_GPIO::Read(void);
```

- Reads the value of the entire port (16 bits).

```
void JSAPI_GPIO::Write(unsigned val);
```

- Writes the 16-bit argument to the port.

Notes
The STM32F030 has 6 GPIO ports, each one with 16 I/O pins. `jsaphi.h` declares them as `portX`, where X is `a..f`. These function are “port” centric, and the affected pin is specified as input to the member functions. Before you use an I/O pin, you need to call one of the `MakeXXX` functions to configure it.

JSAPI_IOPIN: I/O Pin Abstraction

Most embedded programs deal with I/O pins individually. The `JSAPI_IOPIN` class provides that abstraction; it is basically an object containing the GPIO port and the pin number.

JSAPI_IOPIN Object

There are no predefined `JSAPI_IOPIN` objects. While it is possible to predefine all the possible `JSAPI_IOPIN` objects, this would create 96 (16 pins times 6 ports) objects that are not used by most programs. In addition, it's more convenient to name your `JSAPI_IOPIN` with names such as `led0` or `switch5`, rather than `porta0`, `porta1`, etc.

You declare objects of type `JSAPI_IOPIN`, and then initialize them with the `MakeIOPin` function. See <Examples> below.

API

The output functions are applicable to output pins. However, there is no error checking, and the behavior is defined by the hardware. If you write to an input pin, in most cases, the behavior is that nothing will happen.

`void JSAPI_IOPIN::MakeIOPin(JSAPI_GPIO *gpio, unsigned pin_no);`

- Initializes a `JSAPI_IOPIN` object.

`unsigned JSAPI_IOPIN::Read(void);`

- Reads the status of an I/O pin. Returns 0 or 1.

`void JSAPI_IOPIN::Write(_Bool state);`

- Writes to the output pin. Only the LSB (1 or 0) is used.

`void JSAPI_IOPIN::Set(void);`

- Sets the output pin.

`void JSAPI_IOPIN::Clear(void);`

- Clears the output pin.

`void JSAPI_IOPIN::Toggle(void);`

- Toggles the output pin state.

`_Bool JSAPI_IOPIN::isSet(void);`

- Returns true if the state of the pin is set and false otherwise.

`_Bool JSAPI_IOPIN::isClear(void);`

- Returns true if the state of the pin is clear and false otherwise.

```
void JSAPI_IOPIN::JiggleLow(unsigned delay);
```

- “Jiggles” the state of the output to low (clear) for a specified number of milliseconds of delay.

```
void JSAPI_IOPIN::JiggleHigh(unsigned delay);
```

- “Jiggles” the state of the output to high (set) for a specified number of milliseconds of delay.

Examples
The LED blinker can be written as:

```
porta.MakeOutput(5, OSPEED_LOW);
JSAPI_IOPIN led;
led.MakeIOPin(&porta, 5);

while (1)
      {
      led.Toggle();
      DelayTenth(5);
      }
```

Notes
Most of the functions are simply a shorthand for making the equivalent calls using the JSAPI_GPIO functions.

JSAPI_EXTI: EXTI (External Interrupt) Functions

EXTI (External Interrupt) is a set of interrupts tied to the GPIO ports. For full details, please refer to the STM32F030 reference manual. There are six GPIO ports in the STM32F030 (GPIOA to GPIOF[3]), and each one has 16 port bits.

There are 16 EXTI for the GPIO ports, named EXTI0 to EXTI15. Each EXTIx can be configured to trigger on bit x of any of the GPIO port. For example, EXTI0 can be triggered by bit 0 of any of the GPIO port, and EXTI1 can be triggered by bit 1 of any of the GPIO port etc.

JSAPI_EXTI Objects

```
extern JSAPI_EXTI exti0, exti1, exti2,  exti3,  exti4,  exti5,
exti6,
     exti7, exti8, exit9, exti10, exti11, exti12, exti13, exti14,
     exti15;
```

These objects correspond to the 16 hardware EXTI of the same names.

Type

```
enum exti_edge {EXTI_RISING_EDGE = 1, EXTI_FALLING_EDGE = 2,
EXTI_BOTH_EDGES};
```

API

```
void JSAPI_EXTI::MakeEXTI(JSAPI_GPIO *port, enum PUPDR pupdr,
          enum exti_edge edge, int priority, void (*isr)(void));
```

- Sets the EXTI to the specified port. The pin is a property of the JSAPI_EXTI object; e.g., `exti0` is for pin zero.
- `pupdr` is the pull-up or pull-down value,
- `edge` is the edge the interrupt should trigger on,
- `priority` is the interrupt priority, which can be from 1 to 3 for Cortex-M0, and
- `isr` is the interrupt handler.

```
void JSAPI_EXTI::Debounce(int millisecs);
```

- Busy wait for the specified milliseconds, and then check the status of the pin until it is clear.

```
void JSAPI_EXTI::Disable(void);
```

- Disables the EXTI interrupt.

```
void JSAPI_EXTI::Enable(void);
```

- Enables the EXTI interrupt.

[3] STM32F030 does not allow EXTI on the GPIOE port.

Examples
When properly setup, this code fragment prints out ‘.’ indefinitely, and whenever a button tied to `PORTA`, pin 0, is pushed, would print out a ‘1.’

```
static void SW1(void)
      {
      putchar('1');
      exti0.Debounce();
      }

// in main()
…
exti0.MakeEXTI(&porta, PUPDR_UP, EXTI_FALLING_EDGE, 0, SW1);
while (1)
      putchar('.');
```

Notes
The single JSAPI function `MakeEXTI` does all the following: in addition to modifying the EXTI subsystem, it changes the STM32F030 SYSCFG subsystem and configures the NVIC (Nested Vectored Interrupt Controller) properly.

In fact, `MakeEXTI` does even more: Cortex-M devices use shared vector interrupt entries. For example, in the Cortex-M0, EXTI0 and EXTI1 share one interrupt entry, EXT2 and EXT3 share another entry, and EXT4 to EXT15 share another entry. This means that when an EXTI interrupt handler is entered, it will need to test which EXTI line the interrupt is really triggering. The JSAPI function does away with all that and allows the users to specify an individual interrupt handler for each EXTI.

This is a clear demonstration of using JumpStart API.

This has a small amount of runtime overhead using the JumpStart API, especially comparing to the case where you are only enabling one EXTI out of a shared group — for example, only enabling EXTI4 and none of the EXTI5 to EXTI15. In which case, no checking would have been needed, but the JSAPI function still performs the check. The simplicity outweighs the small amount of overhead.

JSAPI_TIMER: Timer Functions

You can use timers for a number of purposes. `JSAPI_TIMER` exposes the two commonly used features of timers: counting and generating PWM (Pulse Width Modulation) waveforms.

JSAPI_TIMER Objects (STM32F0)

```
extern JSAPI_TIMER timer1, timer2, timer3, timer6, timer7, timer14,
     timer15, timer16, timer17;
```

JSAPI_TIMER Objects (STM32F4)

```
extern JSAPI_TIMER timer1, timer2, timer3, timer4, timer5, timer6,
     timer7, timer8, timer9, timer10, timer11;
```

Not all timers are available on all devices.

API

```
_Bool JSAPI_TIMER::MakeTimer(unsigned Hz, void (*isr)(void));
```

- Initializes the timer to run with the specified frequency. If `isr` is non-null, then interrupt is enabled with `isr` as the interrupt handler.

```
void JSAPI_TIMER::ResetTimer(void);
```

- Resets the timer counter to zero.

```
unsigned JSAPI_TIMER::ReadTimer(void);
```

- Reads the value of the timer counter.

```
_Bool JSAPI_TIMER::SetPinsForPWM(JSAPI_GPIO *ch1_port,
                unsigned ch1_pin_no, unsigned ch1_af);
```

- Specific the GPIO port and pin number to use for PWM (Pulse Width Modulation) generation. Not all ports or pins can be used. Please consult the STM32F030 reference manual for details. `af` is the alternate function code.

```
void JSAPI_TIMER::MakePWM(unsigned Hz,
                unsigned duty_cycle_in_percent);
```

- Enables PWM output of specified frequency. `duty_cycle_in_percent` is the percentage (1 to 99, although the API does not check invalid argument) the waveform is high.

```
void JSAPI_TIMER::Enable(void);
```

- Enables the timer.

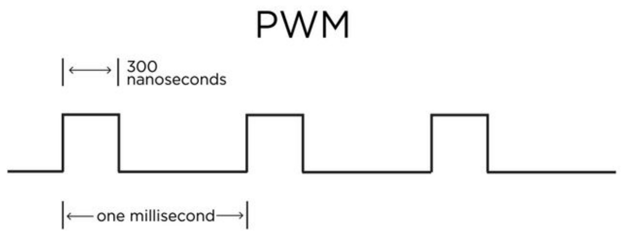
```
void JSAPI_TIMER::Disable(void);
```

- Disables the timer.

Examples
To use the timer to generate a PWM, use the function `SetPinsForPWM` to configure the pin, and then use `MakePWM` to specify the frequency of the wave. `duty_cycle_in_percent` specifies the amount of the time the signal should be active high. For example:

```
// use PA8 alternate function 2 for TIMER1 PWM output
timer1.SetPinsForPWM(&porta, 9, 2);
timer1.MakePWM(1000, 30);
```

The example `MakePWM` call generates a 1-kHz signal and 30% duty cycle. At 1 kHz, the time period of each wave is 1 millisecond. 30% duty cycle means that the high signal is 0.3 milliseconds, or 300 nanoseconds.



Notes
Timers are one of the most flexible peripheral subsystems in a microcontroller. A timer can be used for *input capture* (detecting the frequency of an input waveform), *output compare*, and many other functions. See the document <JumpStart MicroBox Hardware> for some descriptions and the STM32F030 reference manual for details.

JSAPI_SPI: SPI Functions

SPI is one of the most common embedded serial protocols. See the document <JumpStart MicroBox Hardware> for details. JSAPI supports the single,master single-slave SPI by default. You may add support for multiple SPI slaves by adding additional chip select pins and manage them yourself.

JSAPI_SPI Objects

```
extern JSAPI_SPI spi1, spi2;
```

Types

```
#define SPI_CPOL_HIGH      0b01
#define SPI_CPHA_HIGH      0b10
```

API

```
void JSAPI_SPI::SetPins(
     JSAPI_GPIO *sck_port, unsigned sck_pin_no, unsigned sck_af,
     JSAPI_GPIO *mosi_port, unsigned mosi_pin_no, unsigned mosi_af,
     JSAPI_GPIO *miso_port, unsigned miso_pin_no, unsigned miso_af,
     JSAPI_GPIO *cs_port, unsigned cs_pin_no,
     _Bool active_low);
```

- Sets the pins for the SPI. There are 4 pins used for SPI: SCK (clock), MOSI (Master Out Slave In), MISO (Master In Slave Out), and CS (Chip Select). Sometimes CS is called NSS (Negative Slave Select), as the signal is usually active low.

  For each pin function, you specify the port, the pin number, and the alternate function code.

  JSAPI_SPI uses *software slave management*, and you may specify any output pin for CS.

- `active_low` specifies whether CS is active low or not.

```
// mode = CPOL|CPHA, default: 0
void JSAPI_SPI::MakeSPI(unsigned bits, unsigned spi_mode,
                unsigned hz);
```

- Specifies the SPI protocol parameters and enables the SPI subsystem.
- `bits` is the number of bits, from 4 to 16.
- `spi_mode` is 0, 1, 2, or 3, specifying the `SPI_CPOL_HIGH` and/or `SPI_CPHA_HIGH` if appropriate. The default value of zero works with most SPI devices.
- `hz` is the frequency in Hertz. SPI can operate from 100 kHz to 10 MHz. Consult the slave device datasheet for details. This function sets the SPI clock close to but not

higher than the specified frequency, as limited by the hardware SPI clock prescaler values available.

`void JSAPI_SPI::ChipSelect(void);`

- Enables the Chip Select.

`void JSAPI_SPI::ChipDeselect(void);`

- Disables the Chip Select. This does not disable the SPI subsystem. See **Notes** below.

`void JSAPI_SPI::Enable(void);`

- Enables the SPI subsystem.

`void JSAPI_SPI::Disable(void);`

- Disables the SPI subsystem.

`unsigned JSAPI_SPI::Write(unsigned data);`

- Writes a value to the SPI bus. Returns a value from the slave. Either an 8-bit or 16-bit write will be done, depending on the number of bits you specify in the `MakeSPI` call.

`unsigned JSAPI_SPI::Read(void);`

- Reads a value from the slave. Since a SPI slave cannot writes back without the master writing to it, this calls writes a value of 0x00 from the microcontroller. Either an 8-bit or 16-bit read and returned, depending on the number of bits you specify in the `MakeSPI` call.

```
int JSAPI_SPI::WriteByteArray(unsigned char *readbuf,
     unsigned char *buf, int numbytes);
```

- Writes a byte array. Returned items will be stored in `readbuf` if it is not null. Valid only if the number of bits are set to 8 or less in the `MakeSPI` call.

`int JSAPI_SPI::ReadByteArray(unsigned char *readbuf, int numbytes);`

- Reads a byte array. Returned items will be stored in `readbuf`. Valid only if the number of bits are set to 8 or less in the `MakeSPI` call.

```
int JSAPI_SPI::WriteWordArray(unsigned short *readbuf,
     unsigned short *buf, int numwords);
```

- Writes a word (16-bit) array. Returned items will be stored in `readbuf` if it is not null. Valid only if the number of bits are set to 9 or more in the `MakeSPI` call.

`int JSAPI_SPI::ReadWordArray(unsigned short *readbuf, int numwords);`

- Reads a word (16-bit) array. Returned items will be stored in `readbuf`. Valid only if the number of bits are set to 9 or more in the `MakeSPI` call.

`void JSAPI_SPI::WriteBytes(unsigned char *buf, int numbytes, ...);`

- Writes `numbytes` of (8-bit) bytes to the slave. The returned bytes are stored in `buf` and must have at least `numbytes` element. The “...” is the C notation to specify a variable number of arguments, in this case, the list of bytes to be written to the slave.

`void JSAPI_SPI::WriteWords(unsigned short *buf, int numwords, ...);`

- Writes `numwords` of (16-bit) words to the slave. The returned words are stored in `buf` and must have at least `numwords` element. The “...” is the C notation to specify a variable number of arguments, in this case, the list of words to be written to the slave.

`void JSAPI_SPI::ReadBytes(unsigned char *buf, int numbytes);`

- Reads `numbytes` of (8-bit) bytes from the slave. A sequence of 0x00 is sent from the microcontroller to the slave to facilitate the reads.

`void JSAPI_SPI::ReadWords(unsigned short *buf, int numwords);`

- Reads `numwords` of (16-bit) words from the slave. A sequence of 0x00 is sent from the microcontroller to the slave to facilitate the reads.

Note
Some SPI slaves require a short delay after chip select to ensure that the bus signals are stable. In that case, you must call a delay function after you call `ChipSelect`. Usually a millisecond is sufficient.

The SD flash memory card can use the SPI protocol for communication. The initial handshaking involves write SPI data to the SPI bus while Chip Select is *DISABLED*.

JSAPI_I2C_WIRE: I2C Physical Wire Abstraction

As an I2C bus can have multiple slave devices that share the same physical I2C wires, I2C functions are split into two different sets of APIs: `JSAPI_I2C_WIRE` and `JSAPI_I2C`. The `JSAPI_I2C_WIRE` functions provide low-level access to the bus protocol.

JSAPI_I2C_WIRE Objects

```
extern JSAPI_I2C_WIRE i2c1, i2c1;
```

API

```
void SetPins(
     JSAPI_GPIO *scl_port, unsigned scl_pin_no,unsigned scl_af,
     JSAPI_GPIO *sda_port, unsigned sda_pin_no,unsigned sda_af);
```

- Sets the pins for I2C. There are two pins for I2C: SCL (Clock) and SDA (Data).

  For each pin function, you specify the port, the pin number, and the alternate function code.

  There is no chip select in I2C. Each set of I2C transactions includes a slave address in the beginning of the transaction, and the slave with the matching address will respond to the communication.

Note:
The half a dozen low-level functions in JSAPI_I2C_WIRE are omitted here, as they are mainly used by the JSAPI_I2C functions (described next) and user code normally does not need to access them.

JSAPI_I2C: I2C Slave Devices

You use JSAPI_I2C class and its member functions to access an actual I2C device. For example, in the ACE Shield, there are 4 devices that are attached to the I2C bus: the LED matrix, the RTC, the serial EEPROM, and the ATSHA204 crypto chip.

It is possible for the STM32F030 to act as an I2C slave and other device on the bus acting as the I2C master, but JSAPI does not support this functionality.

JSAPI_I2C Object

A JSAPI_I2C object is defined by its slave address and other attributes; thus, there are no such things as generic JSAPI_I2C objects. We do, however, provide extended JSAPI calls to support ACE Shield devices such as the RTC. See <JSAPI_RTC> in this document for details.

Type

```
enum I2C_SPEED { _10KHZ, _100KHZ, _400KHZ, _FASTMODE_PLUS };
```

This defines the SPI speeds supported by the STM32F030 subsystem. Most SPI devices can operate with these speeds. “Fast Mode” is 400 kHz and “Fast Mode Plus” is 1 MHz.

```
void JSAPI_I2C::MakeI2C(JSAPI_I2C_WIRE *i2c_wire,
                unsigned slave_address, enum I2C_SPEED ispeed,
                _Bool _2byte_command);
```

- Associates a JSAPI_I2C object with the JSAPI_I2C_WIRE object and specifies the I2C slave parameters.

  `slave_address` is the 7-bit address. Note: some I2C device documentation uses 8 bits for the slave address, so you must trim the low-order bit.

  Some I2C devices use a 10-bit slave address. The STM32F030 I2C subsystem can support this, but this is not supported in the JSAPI.

  `ispeed` is the I2C clock for the slave device. Note that different I2C devices on the same bus may use different speeds.

  `_2byte_command` specifies whether the slave commands are one byte (default and is the most common) or two bytes. Depending on the I2C devices, “commands” may be referred as “registers,” “commands,” “addresses,” or “memory addresses” in the device documentation.

```
int JSAPI_I2C::AcquireI2C(void);
```

- Takes control of the I2C bus for the device.

`int JSAPI_I2C::ReleaseI2C(void);`

- Releases control of the I2C bus.

`int JSAPI_I2C::Write(unsigned command, unsigned char *data, int len);`

- Writes `len` bytes from the buffer `data` to an I2C slave device. Note that this is a complete I2C transaction, starting with an I2C START condition, followed by the slave address, the command, and then the series of data, ending with the I2C STOP condition.

  As limited by the STM32F030 I2C subsystem, `len` must be 255 or less.

`int JSAPI_I2C::Read(unsigned command, unsigned char *data, int len);`

- Reads `len` bytes into the buffer `data` from an I2C slave device. This is a complete I2C transaction, starting with an I2C START condition, followed by the slave address, the command, then the I2C RESTART condition and the slave address again,[4] and then the series of data from the slave, ending with the I2C STOP condition.

  As limited by the STM32F030 I2C subsystem, `len` must be 255 or less.

`int JSAPI_I2C::ReadWithDefaultAddress(unsigned char *data, int len);`

- Reads `len` bytes into the buffer `data` from an I2C slave device. It differs from the `Read` function in that the slave immediately transfers data to the master (microcontroller) using the "current" address as defined by the slave device.

  As limited by the STM32F030 I2C subsystem, `len` must be 255 or less.

Examples

The following writes a string to a serial EEPROM:

```
#define HELLO   "Hello World"

// setup pins for the I2C
i2c1.SetPins(&portb, 8, 1, &portb, 9, 1);

JSAPI_I2C i2c_memory;

// create an I2C device with slave address 0b1010000
// and 2 byte commands
i2c_memory.MakeI2C(&i2c1, 0b1010000, _100KHZ, true);

// write a string to address 0xA
i2c_memory.AcquireI2C();
```

[4] The I2C read protocol is described in the document <JumpStart MicroBox Hardware>.

```
i2c_memory.Write(0xA, HELLO, strlen(HELLO)+1);
DelayTenth(1);
i2c_memory.ReleaseI2C();
```

JSAPI_USART: Universal Synchronous Asynchronous Receiver Transmitter

UART is a common serial communication protocol, especially for devices that are not situated on the same board. JSAPI_USART supports the USART in asynchronous mode only, which is sufficient for most needs.

By default, JSAPI_USART operates in *polled mode*; however *interrupt-driven mode* can be enabled via the `SetIntrMode` function. The function accepts the sizes of the internal transmit and receive buffers as arguments. However, these arguments are currently unused, and a hard-coded value of 12 bytes is used for the buffers. These arguments will be used in future releases when we add support for dynamic memory allocation.

By using interrupt-driven modes, UART *reads* and *writes* go through the internal receive and transmit buffers. As such, for a write operation (i.e. `putchar` function), the program does not need to wait for the (slow) hardware I/O unit to finish. For a read operation (i.e. `getchar` function), incoming data is buffered (up to the size limit of the receive buffer), and will not be lost even if the program is not reading the data.

The ST Nucleo board includes a USB port to the VCOM bridge and the STM32F030 USART2. You can use a terminal emulator program such as PuTTY to communicate with the STM device.

JSAPI_USART Objects

```
extern JSAPI_USART usart1, usart2;
```

Type

```
enum FLOW_CONTROL { FC_NONE, FC_HARDARE, FC_SOFTWARE };
```

API

```
void JSAPI_USART::SetPins(
          JSAPI_GPIO *tx_port, unsigned tx_pin_no, unsigned tx_af,
         JSAPI_GPIO *rx_port, unsigned rx_pin_no,unsigned rx_af);
```

- Sets the pins used for the USART. Two pins are used: TX (Transmit) and RX (Receive).

  For each pin function, you specify the port, the pin number, and the alternate function code.

```
void JSAPI_USART::MakeUSART(unsigned baud, unsigned bits,
          unsigned stop,
          enum FLOW_CONTROL fc);
```

- Sets the UART communication parameters:
  `baud` is the baud rate.

`bits` is the number of data bits, usually 8.
`stop` is the number of stop bits, usually 1.
`fc` specifies whether flow control is used. To enable hardware flow control, you must also specify the CTS and RTS pins. Currently we do not yet support software flow control (to be added).

`void JSAPI_USART::SetBaudrate(unsigned baud);`

- Change the baud rate.

`void JSAPI_USART::SetIntrMode(unsigned txlen, unsigned rxlen);`

- Enables interrupt-driven mode. `txlen` is the size of the transmit buffer to be used, and `rxlen` is the size of the receive buffer to be used. Currently both arguments are ignored, and a hard-coded value of 12 is used for both buffers.

`int JSAPI_USART::putchar(int ch);`

- Writes a byte. In interrupt-driven mode, the byte is written to the internal transmit buffer, and then written out to the hardware transmit output register only when the hardware transmitter is free. If the buffer is full, the character is lost, and a -1 is returned.

  In polled mode, the function waits until the hardware transmit unit is available, then the character is written out.

  If there is no error, then the original character is returned.

`int JSAPI_USART::getchar(void);`

- Reads a byte. In interrupt-driven mode, when a byte arrives at the hardware receive unit, it is written to the internal receive buffer. If the internal buffer is full, then the character is lost. When this function is called, if a character is available in the receive buffer, that character is returned. Otherwise, a -1 is returned.

  In polled mode, the function waits until a byte arrives at the hardware receive unit, and returns it.

`int JSAPI_USART::kbhit(void);`

- Returns 1 if a byte is available to be read, 0 otherwise.

`void JSAPI_USART::Enable(void);`

- Enables the USART.

`void JSAPI_USART::Disable(void);`

- Disables the USART.

.

Notes
You can send and receive characters using the `getchar` and `putchar` functions. These functions are named similar to the C standard I/O (`stdio`) functions. In fact, if you create a top level function called `putchar`, and have it calls the JSAPI_USART `putchar` function, then you can use `printf` and other standard I/O output functions:

```
int putchar(unsigned char ch)
      {
      if (ch == '\n')
      usart2.putchar('\r');
      usart2.putchar(ch);
      return ch;
      }

int main(void)
      {
      jsapi_clock.SystemInit(48);
      usart2.SetPins(&porta, 2, 1, &porta, 3, 1);
      usart2.MakeUSART(9600, 8, 1, true);

      printf("\r\nImageCraft JumpStart Education Kit... System "
  "running at %dMhz\n", jsapi_clock.GetSysClkFreq() / 1000000);
```

In this example fragment, `printf` uses the `%d` format character to print out the system frequency value.

The specified baud rate (9600 in this case) must agree with the baud rate value you specify in the terminal emulator program.

In the printf string,[5] `\r` is the carriage return (CR) and `\n` in the newline (NL). CR moves the cursor to the beginning of the line and NL moves the current one line lower but in the same column position. Some terminal emulators can be set to map a `\n` NL code to both CR and NL functions.

[5] `printf` is described in the “JumpStart Guide to C Programming” book.

JSAPI_ADC: Analog to Digital Converter

Some of the GPIO pins can be used to convert an analog signal, usually a voltage value, to an integer value. The input voltage value must be in reference to GND and a reference input Vdd.

JSAPI_ADC Object

```
extern JSAPI_ADC adc1;
```

There is one ADC in the STM32F030.

API

```
unsigned JSAPI_ADC::Convert(int pin_no, unsigned nsamples,
                unsigned *high, unsigned *low);
```

- Performs ADC on a single channel. Not all pins can be used for ADC, and you must use the call `JSAPI_GPIO::MakeAnalog` on the port pin prior to calling this function.
- `nsamples` is the number of samples to average the conversion.
- `high` is a pointer to an unsigned variable. The high value of the sampling will be written to this variable through the pointer.
- `low` is a pointer to an unsigned variable. The low value of the sampling will be written to this variable through the pointer.

```
void JSAPI_ADC::MultiConvert(unsigned pinmask, unsigned nsamples,
                unsigned results[]);
```

- Performs ADC on multiple channels. DMA is used to transfer data from the ADC to the `results` array for performance reasons.
- `pinmask` contains the bit mask of all the channels/pins to use for conversions. You must use the call `JSAPI_GPIO::MakeAnalog` on the port pins prior to calling this function.
- `nsamples` is the number of samples to average the conversion.
- `results` is the array to store the average results for each conversion. It must have space to store (`nsamples` * number of channels) entries.

Note

The ADC Convert function performs ADC conversion on an analog input pin. Rather than taking a single reading, which might be affected by a variety of environmental factors, the function lets you specify how many samples it should perform. It returns the average of the sample readings, in addition to the high and low values, so you can check how far off the average value is (doing a full standard deviation is more time-intensive and unlikely to be worthwhile).

For `MultiConvert`, assuming channels 0, 3, 7 are used and 4 samples each are specified, the results are stored in this order:

```
results[0]      channel 0 sample 1
results[1]      channel 0 sample 2
results[2]      channel 0 sample 3
results[3]      channel 0 sample 4
results[4]      channel 3 sample 1
results[5]      channel 3 sample 2
results[6]      channel 3 sample 3
results[7]      channel 3 sample 4
results[8]      channel 7 sample 1
results[9]      channel 7 sample 2
results[10]     channel 7 sample 3
results[11]     channel 7 sample 4
```

ACE Shield Hardware JSAPI Extensions

These API are built on top of the generic JSAPI functions.

JSAPI_RTC: DS1307 RTC Functions

These functions encapsulate the RTC features into a single JSAPI class.

JSAPI_RTC Object

None predefined. You must define an object with the type `JSAPI_RTC`.

Struct Members:

```
unsigned secs, mins, hours, date, month, year;
```

These are the structure members in the `JSAPI_RTC` class.

API

`void JSAPI_RTC::MakeRTC(JSAPI_I2C *i2c);`

- Associates an `JSAPI_I2C` object with the RTC. You must have initialized the `JSAPI_I2C` object with the proper slave address for the DS1307.

`void JSAPI_RTC::Set(void);`

- Sets the DS1307 clock data with the `JSAPI_RTC` struct members info.

`void JSAPI_RTC::Read(void);`

- Reads the DS1307 clock data and converts it into the `JSAPI_RTC` struct members format.

Notes

The DS1307 RTC stores the date information in an internal format that is based on BCD. The `Set` and `Read` functions are provided to convert between this internal format and a simpler C-friendly format.

Note that the DS1307 has 56 bytes of SRAM. If you use a battery to protect the content of the RTC, as is the case with the DS1307 on the ACE Shield, then you may use these SRAM bytes for storage that can survive system reset or power-off, as long as the battery provides enough power.

JSAPI_WS0010: OLED LCD Functions

The LCD on the ACE Shield is a 2-line OLED and has a built-in controller with the part name WS0010. This extended JSAPI function takes care of the timing and other issues to use the LCD in text mode.

The part can be used as a graphics LCD, but this feature is not (yet) supported by JSAPI.

JSAPI_WS0010 Object

None predefined. You must define an object with the type `JSAPI_WS0010`.

```
void JSAPI_WS0010::SetPins(
                JSAPI_GPIO *en_port, unsigned en_port_no,
                JSAPI_GPIO *rs_port, unsigned rs_port_no,
                JSAPI_GPIO *d7_port, unsigned d7_port_no,
                JSAPI_GPIO *d6_port, unsigned d6_port_no,
                JSAPI_GPIO *d5_port, unsigned d5_port_no,
                JSAPI_GPIO *d4_port, unsigned d4_port_no,
                JSAPI_GPIO *d3_port, unsigned d3_port_no,
                JSAPI_GPIO *d2_port, unsigned d2_port_no,
                JSAPI_GPIO *d1_port, unsigned d1_port_no,
                JSAPI_GPIO *d0_port, unsigned d0_port_no);
```

- The LCD requires 8 data bits and 2 control lines: EN for chip enable and RS to select between sending commands or sending data to the WS0010 controller.

```
void JSAPI_WS0010::MakeLCD(unsigned rows, unsigned columns);
```

- Declares the rows and columns attributes of the LCD.

```
void JSAPI_WS0010::SetMode(_Bool set_data_mode);
```

- Specifies whether to set the WS0010 to accept data (`set_data_mode` is true) or command (`set_data_mode` is false).

```
void JSAPI_WS0010::SendByte(unsigned val);
```

- Sends a byte to the WS0010. Uses current command or data mode setting.

```
void JSAPI_WS0010::putchar(unsigned char c);
```

- Sets the WS0010 to “accept data” mode, and then sends a byte to the LCD at current cursor position.

```
void JSAPI_WS0010::puts(int line_number, char *s);
```

- Writes a string to the LCD. `line_number` is the the number of rows specified in the `MakeLCD` call.

Any characters exceeding the number of columns are not displayed.

`int JSAPI_WS0010::printf(int line_number, char *fmt, … );`

- Writes a formatted output to the LCD. `line_number` is the the number of rows specified in the `MakeLCD` call.

  The rest of the arguments have the same meaning and function signatures as those of the standard `printf` function.