AltOS

Altos Metrum Operating System

Keith Packard

AltOS: Altos Metrum Operating System

Keith Packard

Table of Contents

Chapter 1. Overview

AltOS is a operating system built for a variety of microcontrollers used in Altus Metrum devices. It has a simple porting layer for each CPU while providing a convenient operating enviroment for the developer. AltOS currently supports three different CPUs:

- STM32L series from ST Microelectronics. This ARM Cortex-M3 based microcontroller offers low power consumption and a wide variety of built-in peripherals. Altus Metrum uses this in the TeleMega, MegaDongle and TeleLCO projects.

- CC1111 from Texas Instruments. This device includes a fabulous 10mW digital RF transceiver along with an 8051-compatible processor core and a range of peripherals. This is used in the TeleMetrum, TeleMini, TeleDongle and TeleFire projects which share the need for a small microcontroller and an RF interface.

- ATmega32U4 from Atmel. This 8-bit AVR microcontroller is one of the many used to create Arduino boards. The 32U4 includes a USB interface, making it easy to connect to other computers. Altus Metrum used this in prototypes of the TeleScience and TelePyro boards; those have been switched to the STM32L which is more capable and cheaper.

Among the features of AltOS are:

- Multi-tasking. While microcontrollers often don't provide separate address spaces, it's often easier to write code that operates in separate threads instead of tying everything into one giant event loop.

- Non-preemptive. This increases latency for thread switching but reduces the number of places where context switching can occur. It also simplifies the operating system design somewhat. Nothing in the target system (rocket flight control) has tight timing requirements, and so this seems like a reasonable compromise.

- Sleep/wakeup scheduling. Taken directly from ancient Unix designs, these two provide the fundemental scheduling primitive within AltOS.

- Mutexes. As a locking primitive, mutexes are easier to use than semaphores, at least in my experience.

- Timers. Tasks can set an alarm which will abort any pending sleep, allowing operations to time-out instead of blocking forever.

The device drivers and other subsystems in AltOS are conventionally enabled by invoking their _init() function from the 'main' function before that calls ao_start_scheduler(). These functions initialize the pin assignments, add various commands to the command processor and may add tasks to the scheduler to handle the device. A typical main program, thus, looks like:

```
void
main(void)
{
        ao_clock_init();

        /* Turn on the LED until the system is stable */
        ao_led_init(LEDS_AVAILABLE);
        ao_led_on(AO_LED_RED);
        ao_timer_init();
        ao_cmd_init();
        ao_usb_init();
```

```
        ao_monitor_init(AO_LED_GREEN, TRUE);
        ao_rssi_init(AO_LED_RED);
        ao_radio_init();
        ao_packet_slave_init();
        ao_packet_master_init();
        #if HAS_DBG
        ao_dbg_init();
        #endif
        ao_config_init();
        ao_start_scheduler();
}
```

As you can see, a long sequence of subsystems are initialized and then the scheduler is started.

Chapter 2. AltOS Porting Layer

AltOS provides a CPU-independent interface to various common microcontroller subsystems, including GPIO pins, interrupts, SPI, I2C, USB and asynchronous serial interfaces. By making these CPU-independent, device drivers, generic OS and application code can all be written that work on any supported CPU. Many of the architecture abstraction interfaces are prefixed with ao_arch.

Low-level CPU operations

These primitive operations provide the abstraction needed to run the multi-tasking framework while providing reliable interrupt delivery.

ao_arch_block_interrupts/ao_arch_release_interrupts

```
static inline void
ao_arch_block_interrupts(void);

static inline void
ao_arch_release_interrupts(void);
```

These disable/enable interrupt delivery, they may not discard any interrupts. Use these for sections of code that must be atomic with respect to any code run from an interrupt handler.

ao_arch_save_regs, ao_arch_save_stack, ao_arch_restore_stack

```
static inline void
ao_arch_save_regs(void);

static inline void
ao_arch_save_stack(void);

static inline void
ao_arch_restore_stack(void);
```

These provide all of the support needed to switch between tasks.. ao_arch_save_regs must save all CPU registers to the current stack, including the interrupt enable state. ao_arch_save_stack records the current stack location in the current ao_task structure. ao_arch_restore_stack switches back to the saved stack, restores all registers and branches to the saved return address.

ao_arch_wait_interupt

```
#define ao_arch_wait_interrupt()
```

This stops the CPU, leaving clocks and interrupts enabled. When an interrupt is received, this must wake up and handle the interrupt. ao_arch_wait_interrupt is entered with interrupts disabled to ensure that there is no gap between determining that no task wants to run and idling the CPU. It must sleep the CPU, process interrupts and then disable interrupts again. If the CPU doesn't have any reduced power mode, this must at the least allow pending interrupts to be processed.

GPIO operations

These functions provide an abstract interface to configure and manipulate GPIO pins.

GPIO setup

These macros may be invoked at system initialization time to configure pins as needed for system operation. One tricky aspect is that some chips provide direct access to specific GPIO pins while others only provide access to a whole register full of pins. To support this, the GPIO macros provide both port+bit and pin arguments. Simply define the arguments needed for the target platform and leave the others undefined.

ao_enable_output

```
#define ao_enable_output(port, bit, pin, value)
```

Set the specified port+bit (also called 'pin') for output, initializing to the specified value. The macro must avoid driving the pin with the opposite value if at all possible.

ao_enable_input

```
#define ao_enable_input(port, bit, mode)
```

Sets the specified port/bit to be an input pin. 'mode' is a combination of one or more of the following. Note that some platforms may not support the desired mode. In that case, the value will not be defined so that the program will fail to compile.

- AO_EXTI_MODE_PULL_UP. Apply a pull-up to the pin; a disconnected pin will read as 1.
- AO_EXTI_MODE_PULL_DOWN. Apply a pull-down to the pin; a disconnected pin will read as 0.
- 0. Don't apply either a pull-up or pull-down. A disconnected pin will read an undetermined value.

Reading and writing GPIO pins

These macros read and write individual GPIO pins.

ao_gpio_set

```
#define ao_gpio_set(port, bit, pin, value)
```

Sets the specified port/bit or pin to the indicated value

ao_gpio_get

```
#define ao_gpio_get(port, bit, pin)
```

Returns either 1 or 0 depending on whether the input to the pin is high or low.

Chapter 3. Programming the 8051 with SDCC

The 8051 is a primitive 8-bit processor, designed in the mists of time in as few transistors as possible. The architecture is highly irregular and includes several separate memory spaces. Furthermore, accessing stack variables is slow, and the stack itself is of limited size. While SDCC papers over the instruction set, it is not completely able to hide the memory architecture from the application designer.

When built on other architectures, the various SDCC-specific symbols are #defined as empty strings so they don't affect the compiler.

8051 memory spaces

The __data/__xdata/__code memory spaces below were completely separate in the original 8051 design. In the cc1111, this isn't true—they all live in a single unified 64kB address space, and so it's possible to convert any address into a unique 16-bit address. SDCC doesn't know this, and so a 'global' address to SDCC consumes 3 bytes of memory, 1 byte as a tag indicating the memory space and 2 bytes of offset within that space. AltOS avoids these 3-byte addresses as much as possible; using them involves a function call per byte access. The result is that nearly every variable declaration is decorated with a memory space identifier which clutters the code but makes the resulting code far smaller and more efficient.

__data

The 8051 can directly address these 128 bytes of memory. This makes them precious so they should be reserved for frequently addressed values. Oh, just to confuse things further, the 8 general registers in the CPU are actually stored in this memory space. There are magic instructions to 'bank switch' among 4 banks of these registers located at 0x00 - 0x1F. AltOS uses only the first bank at 0x00 - 0x07, leaving the other 24 bytes available for other data.

__idata

There are an additional 128 bytes of internal memory that share the same address space as __data but which cannot be directly addressed. The stack normally occupies this space and so AltOS doesn't place any static storage here.

__xdata

This is additional general memory accessed through a single 16-bit address register. The CC1111F32 has 32kB of memory available here. Most program data should live in this memory space.

__pdata

This is an alias for the first 256 bytes of __xdata memory, but uses a shorter addressing mode with single global 8-bit value for the high 8 bits of the address and any of several 8-bit registers for the low 8 bits. AltOS uses a few bits of this memory, it should probably use more.

__code

All executable code must live in this address space, but you can stick read-only data here too. It is addressed using the 16-bit address register and special 'code' access opcodes. Anything read-only should live in this space.

__bit

The 8051 has 128 bits of bit-addressible memory that lives in the __data segment from 0x20 through 0x2f. Special instructions access these bits in a single atomic operation. This isn't so much a separate address space as a special addressing mode for a few bytes in the __data segment.

__sfr, __sfr16, __sfr32, __sbit

Access to physical registers in the device use this mode which declares the variable name, its type and the address it lives at. No memory is allocated for these variables.

Function calls on the 8051

Because stack addressing is expensive, and stack space limited, the default function call declaration in SDCC allocates all parameters and local variables in static global memory. Just like fortran. This makes these functions non-reentrant, and also consume space for parameters and locals even when they are not running. The benefit is smaller code and faster execution.

__reentrant functions

All functions which are re-entrant, either due to recursion or due to a potential context switch while executing, should be marked as __reentrant so that their parameters and local variables get allocated on the stack. This ensures that these values are not overwritten by another invocation of the function.

Functions which use significant amounts of space for arguments and/or local variables and which are not often invoked can also be marked as __reentrant. The resulting code will be larger, but the savings in memory are frequently worthwhile.

Non __reentrant functions

All parameters and locals in non-reentrant functions can have data space decoration so that they are allocated in __xdata, __pdata or __data space as desired. This can avoid consuming __data space for infrequently used variables in frequently used functions.

All library functions called by SDCC, including functions for multiplying and dividing large data types, are non-reentrant. Because of this, interrupt handlers must not invoke any library functions, including the multiply and divide code.

__interrupt functions

Interrupt functions are declared with with an __interrupt decoration that includes the interrupt number. SDCC saves and restores all of the registers in these functions and uses the 'reti' instruction at the end so that they operate as stand-alone interrupt handlers. Interrupt functions may call the ao_wakeup function to wake AltOS tasks.

__critical functions and statements

SDCC has built-in support for suspending interrupts during critical code. Functions marked as __critical will have interrupts suspended for the whole period of execution. Individual statements may also be marked as __critical which blocks interrupts during the execution of that statement. Keeping critical sections as short as possible is key to ensuring that interrupts are handled as quickly as possible. AltOS doesn't use this form in shared code as other compilers wouldn't know what to do. Use ao_arch_block_interrupts and ao_arch_release_interrupts instead.

Chapter 4. Task functions

This chapter documents how to create, destroy and schedule AltOS tasks.

ao_add_task

```
void
ao_add_task(__xdata struct ao_task * task,
            void (*start)(void),
            __code char *name);
```

This initializes the statically allocated task structure, assigns a name to it (not used for anything but the task display), and the start address. It does not switch to the new task. 'start' must not ever return; there is no place to return to.

ao_exit

```
void
ao_exit(void)
```

This terminates the current task.

ao_sleep

```
void
ao_sleep(__xdata void *wchan)
```

This suspends the current task until 'wchan' is signaled by ao_wakeup, or until the timeout, set by ao_alarm, fires. If 'wchan' is signaled, ao_sleep returns 0, otherwise it returns 1. This is the only way to switch to another task.

Because ao_wakeup wakes every task waiting on a particular location, ao_sleep should be used in a loop that first checks the desired condition, blocks in ao_sleep and then rechecks until the condition is satisfied. If the location may be signaled from an interrupt handler, the code will need to block interrupts around the block of code. Here's a complete example:

```
ao_arch_block_interrupts();
while (!ao_radio_done)
        ao_sleep(&ao_radio_done);
ao_arch_release_interrupts();
```

ao_wakeup

```
void
ao_wakeup(__xdata void *wchan)
```

Wake all tasks blocked on 'wchan'. This makes them available to be run again, but does not actually switch to another task. Here's an example of using this:

```
if (RFIF & RFIF_IM_DONE) {
        ao_radio_done = 1;
        ao_wakeup(&ao_radio_done);
        RFIF &= ~RFIF_IM_DONE;
}
```

Note that this need not block interrupts as the ao_sleep block can only be run from normal mode, and so this sequence can never be interrupted with execution of the other sequence.

ao_alarm

```
void
ao_alarm(uint16_t delay);

void
ao_clear_alarm(void);
```

Schedules an alarm to fire in at least 'delay' ticks. If the task is asleep when the alarm fires, it will wakeup and ao_sleep will return 1. ao_clear_alarm resets any pending alarm so that it doesn't fire at some arbitrary point in the future.

```
ao_alarm(ao_packet_master_delay);
ao_arch_block_interrupts();
while (!ao_radio_dma_done)
        if (ao_sleep(&ao_radio_dma_done) != 0)
                ao_radio_abort();
ao_arch_release_interrupts();
ao_clear_alarm();
```

In this example, a timeout is set before waiting for incoming radio data. If no data is received before the timeout fires, ao_sleep will return 1 and then this code will abort the radio receive operation.

ao_start_scheduler

```
void
ao_start_scheduler(void);
```

This is called from 'main' when the system is all initialized and ready to run. It will not return.

ao_clock_init

```
void
ao_clock_init(void);
```

This initializes the main CPU clock and switches to it.

Chapter 5. Timer Functions

AltOS sets up one of the CPU timers to run at 100Hz and exposes this tick as the fundemental unit of time. At each interrupt, AltOS increments the counter, and schedules any tasks waiting for that time to pass, then fires off the sensors to collect current data readings. Doing this from the ISR ensures that the values are sampled at a regular rate, independent of any scheduling jitter.

ao_time

```
uint16_t
ao_time(void)
```

Returns the current system tick count. Note that this is only a 16 bit value, and so it wraps every 655.36 seconds.

ao_delay

```
void
ao_delay(uint16_t ticks);
```

Suspend the current task for at least 'ticks' clock units.

ao_timer_set_adc_interval

```
void
ao_timer_set_adc_interval(uint8_t interval);
```

This sets the number of ticks between ADC samples. If set to 0, no ADC samples are generated. AltOS uses this to slow down the ADC sampling rate to save power.

ao_timer_init

```
void
ao_timer_init(void)
```

This turns on the 100Hz tick. It is required for any of the time-based functions to work. It should be called by 'main' before ao_start_scheduler.

Chapter 6. AltOS Mutexes

AltOS provides mutexes as a basic synchronization primitive. Each mutexes is simply a byte of memory which holds 0 when the mutex is free or the task id of the owning task when the mutex is owned. Mutex calls are checked—attempting to acquire a mutex already held by the current task or releasing a mutex not held by the current task will both cause a panic.

ao_mutex_get

```
void
ao_mutex_get(__xdata uint8_t *mutex);
```

Acquires the specified mutex, blocking if the mutex is owned by another task.

ao_mutex_put

```
void
ao_mutex_put(__xdata uint8_t *mutex);
```

Releases the specified mutex, waking up all tasks waiting for it.

Chapter 7. DMA engine

The CC1111 and STM32L both contain a useful bit of extra hardware in the form of a number of programmable DMA engines. They can be configured to copy data in memory, or between memory and devices (or even between two devices). AltOS exposes a general interface to this hardware and uses it to handle both internal and external devices.

Because the CC1111 and STM32L DMA engines are different, the interface to them is also different. As the DMA engines are currently used to implement platform-specific drivers, this isn't yet a problem.

Code using a DMA engine should allocate one at startup time. There is no provision to free them, and if you run out, AltOS will simply panic.

During operation, the DMA engine is initialized with the transfer parameters. Then it is started, at which point it awaits a suitable event to start copying data. When copying data from hardware to memory, that trigger event is supplied by the hardware device. When copying data from memory to hardware, the transfer is usually initiated by software.

CC1111 DMA Engine

ao_dma_alloc

```
uint8_t
ao_dma_alloc(__xdata uint8_t *done)
```

Allocate a DMA engine, returning the identifier. 'done' is cleared when the DMA is started, and then receives the AO_DMA_DONE bit on a successful transfer or the AO_DMA_ABORTED bit if ao_dma_abort was called. Note that it is possible to get both bits if the transfer was aborted after it had finished.

ao_dma_set_transfer

```
void
ao_dma_set_transfer(uint8_t id,
void __xdata *srcaddr,
void __xdata *dstaddr,
uint16_t count,
uint8_t cfg0,
uint8_t cfg1)
```

Initializes the specified dma engine to copy data from 'srcaddr' to 'dstaddr' for 'count' units. cfg0 and cfg1 are values directly out of the CC1111 documentation and tell the DMA engine what the transfer unit size, direction and step are.

ao_dma_start

```
void
```

```
ao_dma_start(uint8_t id);
```

Arm the specified DMA engine and await a signal from either hardware or software to start transferring data.

ao_dma_trigger

```
void
ao_dma_trigger(uint8_t id)
```

Trigger the specified DMA engine to start copying data.

ao_dma_abort

```
void
ao_dma_abort(uint8_t id)
```

Terminate any in-progress DMA transaction, marking its 'done' variable with the AO_DMA_ABORTED bit.

STM32L DMA Engine

ao_dma_alloc

```
uint8_t ao_dma_done[];

void
ao_dma_alloc(uint8_t index);
```

Reserve a DMA engine for exclusive use by one driver.

ao_dma_set_transfer

```
void
ao_dma_set_transfer(uint8_t id,
void *peripheral,
void *memory,
uint16_t count,
uint32_t ccr);
```

Initializes the specified dma engine to copy data between 'peripheral' and 'memory' for 'count' units. 'ccr' is a value directly out of the STM32L documentation and tells the DMA engine what the transfer unit size, direction and step are.

ao_dma_set_isr

```
void
ao_dma_set_isr(uint8_t index, void (*isr)(int))
```

This sets a function to be called when the DMA transfer completes in lieu of setting the ao_dma_done bits. Use this when some work needs to be done when the DMA finishes that cannot wait until user space resumes.

ao_dma_start

```
void
ao_dma_start(uint8_t id);
```

Arm the specified DMA engine and await a signal from either hardware or software to start transferring data. 'ao_dma_done[index]' is cleared when the DMA is started, and then receives the AO_DMA_DONE bit on a successful transfer or the AO_DMA_ABORTED bit if ao_dma_abort was called. Note that it is possible to get both bits if the transfer was aborted after it had finished.

ao_dma_done_transfer

```
void
ao_dma_done_transfer(uint8_t id);
```

Signals that a specific DMA engine is done being used. This allows multiple drivers to use the same DMA engine safely.

ao_dma_abort

```
void
ao_dma_abort(uint8_t id)
```

Terminate any in-progress DMA transaction, marking its 'done' variable with the AO_DMA_ABORTED bit.

Chapter 8. Stdio interface

AltOS offers a stdio interface over USB, serial and the RF packet link. This provides for control of the device locally or remotely. This is hooked up to the stdio functions by providing the standard putchar/ getchar/flush functions. These automatically multiplex the available communication channels; output is always delivered to the channel which provided the most recent input.

putchar

```
void
putchar(char c)
```

Delivers a single character to the current console device.

getchar

```
char
getchar(void)
```

Reads a single character from any of the available console devices. The current console device is set to that which delivered this character. This blocks until a character is available.

flush

```
void
flush(void)
```

Flushes the current console device output buffer. Any pending characters will be delivered to the target device.

ao_add_stdio

```
void
ao_add_stdio(char (*pollchar)(void),
                  void (*putchar)(char),
                  void (*flush)(void))
```

This adds another console device to the available list.

'pollchar' returns either an available character or AO_READ_AGAIN if none is available. Significantly, it does not block. The device driver must set 'ao_stdin_ready' to 1 and call ao_wakeup(&ao_stdin_ready) when it receives input to tell getchar that more data is available, at which point 'pollchar' will be called again.

'putchar' queues a character for output, flushing if the output buffer is full. It may block in this case.

'flush' forces the output buffer to be flushed. It may block until the buffer is delivered, but it is not required to do so.

Chapter 9. Command line interface

AltOS includes a simple command line parser which is hooked up to the stdio interfaces permitting remote control of the device over USB, serial or the RF link as desired. Each command uses a single character to invoke it, the remaining characters on the line are available as parameters to the command.

ao_cmd_register

```
void
ao_cmd_register(__code struct ao_cmds *cmds)
```

This registers a set of commands with the command parser. There is a fixed limit on the number of command sets, the system will panic if too many are registered. Each command is defined by a struct ao_cmds entry:

```
struct ao_cmds {
        char  cmd;
        void  (*func)(void);
        const char *help;
};
```

'cmd' is the character naming the command. 'func' is the function to invoke and 'help' is a string displayed by the '?' command. Syntax errors found while executing 'func' should be indicated by modifying the global ao_cmd_status variable with one of the following values:

ao_cmd_success	The command was parsed successfully. There is no need to assign this value, it is the default.
ao_cmd_lex_error	A token in the line was invalid, such as a number containing invalid characters. The low-level lexing functions already assign this value as needed.
ao_syntax_error	The command line is invalid for some reason other than invalid tokens.

ao_cmd_lex

```
void
ao_cmd_lex(void);
```

This gets the next character out of the command line buffer and sticks it into ao_cmd_lex_c. At the end of the line, ao_cmd_lex_c will get a newline ('\n') character.

ao_cmd_put16

```
void
```

```
ao_cmd_put16(uint16_t v);
```

Writes 'v' as four hexadecimal characters.

ao_cmd_put8

```
void
ao_cmd_put8(uint8_t v);
```

Writes 'v' as two hexadecimal characters.

ao_cmd_white

```
void
ao_cmd_white(void)
```

This skips whitespace by calling ao_cmd_lex while ao_cmd_lex_c is either a space or tab. It does not skip any characters if ao_cmd_lex_c already non-white.

ao_cmd_hex

```
void
ao_cmd_hex(void)
```

This reads a 16-bit hexadecimal value from the command line with optional leading whitespace. The resulting value is stored in ao_cmd_lex_i;

ao_cmd_decimal

```
void
ao_cmd_decimal(void)
```

This reads a 32-bit decimal value from the command line with optional leading whitespace. The resulting value is stored in ao_cmd_lex_u32 and the low 16 bits are stored in ao_cmd_lex_i;

ao_match_word

```
uint8_t
ao_match_word(__code char *word)
```

This checks to make sure that 'word' occurs on the command line. It does not skip leading white space. If 'word' is found, then 1 is returned. Otherwise, ao_cmd_status is set to ao_cmd_syntax_error and 0 is returned.

ao_cmd_init

```
void
ao_cmd_init(void
```

Initializes the command system, setting up the built-in commands and adding a task to run the command processing loop. It should be called by 'main' before ao_start_scheduler.

Chapter 10. USB target device

AltOS contains a full-speed USB target device driver. It can be programmed to offer any kind of USB target, but to simplify interactions with a variety of operating systems, AltOS provides only a single target device profile, that of a USB modem which has native drivers for Linux, Windows and Mac OS X. It would be easy to change the code to provide an alternate target device if necessary.

To the rest of the system, the USB device looks like a simple two-way byte stream. It can be hooked into the command line interface if desired, offering control of the device over the USB link. Alternatively, the functions can be accessed directly to provide for USB-specific I/O.

ao_usb_flush

```
void
ao_usb_flush(void);
```

Flushes any pending USB output. This queues an 'IN' packet to be delivered to the USB host if there is pending data, or if the last IN packet was full to indicate to the host that there isn't any more pending data available.

ao_usb_putchar

```
void
ao_usb_putchar(char c);
```

If there is a pending 'IN' packet awaiting delivery to the host, this blocks until that has been fetched. Then, this adds a byte to the pending IN packet for delivery to the USB host. If the USB packet is full, this queues the 'IN' packet for delivery.

ao_usb_pollchar

```
char
ao_usb_pollchar(void);
```

If there are no characters remaining in the last 'OUT' packet received, this returns AO_READ_AGAIN. Otherwise, it returns the next character, reporting to the host that it is ready for more data when the last character is gone.

ao_usb_getchar

```
char
ao_usb_getchar(void);
```

This uses ao_pollchar to receive the next character, blocking while ao_pollchar returns AO_READ_AGAIN.

ao_usb_disable

```
void
ao_usb_disable(void);
```

This turns off the USB controller. It will no longer respond to host requests, nor return characters. Calling any of the i/o routines while the USB device is disabled is undefined, and likely to break things. Disabling the USB device when not needed saves power.

Note that neither TeleDongle nor TeleMetrum are able to signal to the USB host that they have disconnected, so after disabling the USB device, it's likely that the cable will need to be disconnected and reconnected before it will work again.

ao_usb_enable

```
void
ao_usb_enable(void);
```

This turns the USB controller on again after it has been disabled. See the note above about needing to physically remove and re-insert the cable to get the host to re-initialize the USB link.

ao_usb_init

```
void
ao_usb_init(void);
```

This turns the USB controller on, adds a task to handle the control end point and adds the usb I/O functions to the stdio system. Call this from main before ao_start_scheduler.

Chapter 11. Serial peripherals

The CC1111 provides two USART peripherals. AltOS uses one for asynch serial data, generally to communicate with a GPS device, and the other for a SPI bus. The UART is configured to operate in 8-bits, no parity, 1 stop bit framing. The default configuration has clock settings for 4800, 9600 and 57600 baud operation. Additional speeds can be added by computing appropriate clock values.

To prevent loss of data, AltOS provides receive and transmit fifos of 32 characters each.

ao_serial_getchar

```
char
ao_serial_getchar(void);
```

Returns the next character from the receive fifo, blocking until a character is received if the fifo is empty.

ao_serial_putchar

```
void
ao_serial_putchar(char c);
```

Adds a character to the transmit fifo, blocking if the fifo is full. Starts transmitting characters.

ao_serial_drain

```
void
ao_serial_drain(void);
```

Blocks until the transmit fifo is empty. Used internally when changing serial speeds.

ao_serial_set_speed

```
void
ao_serial_set_speed(uint8_t speed);
```

Changes the serial baud rate to one of AO_SERIAL_SPEED_4800, AO_SERIAL_SPEED_9600 or AO_SERIAL_SPEED_57600. This first flushes the transmit fifo using ao_serial_drain.

ao_serial_init

```
void
```

```
ao_serial_init(void)
```

Initializes the serial peripheral. Call this from 'main' before jumping to ao_start_scheduler. The default speed setting is AO_SERIAL_SPEED_4800.

Chapter 12. CC1111 Radio peripheral

Radio Introduction

The CC1111 radio transceiver sends and receives digital packets with forward error correction and detection. The AltOS driver is fairly specific to the needs of the TeleMetrum and TeleDongle devices, using it for other tasks may require customization of the driver itself. There are three basic modes of operation:

1. Telemetry mode. In this mode, TeleMetrum transmits telemetry frames at a fixed rate. The frames are of fixed size. This is strictly a one-way communication from TeleMetrum to TeleDongle.

2. Packet mode. In this mode, the radio is used to create a reliable duplex byte stream between TeleDongle and TeleMetrum. This is an asymmetrical protocol with TeleMetrum only transmitting in response to a packet sent from TeleDongle. Thus getting data from TeleMetrum to TeleDongle requires polling. The polling rate is adaptive, when no data has been received for a while, the rate slows down. The packets are checked at both ends and invalid data are ignored.

   On the TeleMetrum side, the packet link is hooked into the stdio mechanism, providing an alternate data path for the command processor. It is enabled when the unit boots up in 'idle' mode.

   On the TeleDongle side, the packet link is enabled with a command; data from the stdio package is forwarded over the packet link providing a connection from the USB command stream to the remote TeleMetrum device.

3. Radio Direction Finding mode. In this mode, TeleMetrum constructs a special packet that sounds like an audio tone when received by a conventional narrow-band FM receiver. This is designed to provide a beacon to track the device when other location mechanisms fail.

ao_radio_set_telemetry

```
void
ao_radio_set_telemetry(void);
```

Configures the radio to send or receive telemetry packets. This includes packet length, modulation scheme and other RF parameters. It does not include the base frequency or channel though. Those are set at the time of transmission or reception, in case the values are changed by the user.

ao_radio_set_packet

```
void
ao_radio_set_packet(void);
```

Configures the radio to send or receive packet data. This includes packet length, modulation scheme and other RF parameters. It does not include the base frequency or channel though. Those are set at the time of transmission or reception, in case the values are changed by the user.

ao_radio_set_rdf

```
void
ao_radio_set_rdf(void);
```

Configures the radio to send RDF 'packets'. An RDF 'packet' is a sequence of hex 0x55 bytes sent at a base bit rate of 2kbps using a 5kHz deviation. All of the error correction and data whitening logic is turned off so that the resulting modulation is received as a 1kHz tone by a conventional 70cm FM audio receiver.

ao_radio_idle

```
void
ao_radio_idle(void);
```

Sets the radio device to idle mode, waiting until it reaches that state. This will terminate any in-progress transmit or receive operation.

ao_radio_get

```
void
ao_radio_get(void);
```

Acquires the radio mutex and then configures the radio frequency using the global radio calibration and channel values.

ao_radio_put

```
void
ao_radio_put(void);
```

Releases the radio mutex.

ao_radio_abort

```
void
ao_radio_abort(void);
```

Aborts any transmission or reception process by aborting the associated DMA object and calling ao_radio_idle to terminate the radio operation.

Radio Telemetry

In telemetry mode, you can send or receive a telemetry packet. The data from receiving a packet also includes the RSSI and status values supplied by the receiver. These are added after the telemetry data.

ao_radio_send

```
void
ao_radio_send(__xdata struct ao_telemetry *telemetry);
```

This sends the specific telemetry packet, waiting for the transmission to complete. The radio must have been set to telemetry mode. This function calls ao_radio_get() before sending, and ao_radio_put() afterwards, to correctly serialize access to the radio device.

ao_radio_recv

```
void
ao_radio_recv(__xdata struct ao_radio_recv *radio);
```

This blocks waiting for a telemetry packet to be received. The radio must have been set to telemetry mode. This function calls ao_radio_get() before receiving, and ao_radio_put() afterwards, to correctly serialize access to the radio device. This returns non-zero if a packet was received, or zero if the operation was aborted (from some other task calling ao_radio_abort()).

Radio Direction Finding

In radio direction finding mode, there's just one function to use

ao_radio_rdf

```
void
ao_radio_rdf(int ms);
```

This sends an RDF packet lasting for the specified amount of time. The maximum length is 1020 ms.

Radio Packet Mode

Packet mode is asymmetrical and is configured at compile time for either master or slave mode (but not both). The basic I/O functions look the same at both ends, but the internals are different, along with the initialization steps.

ao_packet_putchar

```
void
```

```
ao_packet_putchar(char c);
```

If the output queue is full, this first blocks waiting for that data to be delivered. Then, queues a character for packet transmission. On the master side, this will transmit a packet if the output buffer is full. On the slave side, any pending data will be sent the next time the master polls for data.

ao_packet_pollchar

```
char
ao_packet_pollchar(void);
```

This returns a pending input character if available, otherwise returns AO_READ_AGAIN. On the master side, if this empties the buffer, it triggers a poll for more data.

ao_packet_slave_start

```
void
ao_packet_slave_start(void);
```

This is available only on the slave side and starts a task to listen for packet data.

ao_packet_slave_stop

```
void
ao_packet_slave_stop(void);
```

Disables the packet slave task, stopping the radio receiver.

ao_packet_slave_init

```
void
ao_packet_slave_init(void);
```

Adds the packet stdio functions to the stdio package so that when packet slave mode is enabled, characters will get send and received through the stdio functions.

ao_packet_master_init

```
void
ao_packet_master_init(void);
```

Adds the 'p' packet forward command to start packet mode.