Simulavr

0.1.2.6

Generated by Doxygen 1.7.3

Wed Feb 2 2011 20:17:00

Contents

1 Simulavr Internals

1.1 Introduction

This chapter documents the internals of simulavr for those wishing to work with the source code to fix bugs, add features, or to just see how it all works. If you only wish to know how to use simulavr, you don't need to read this.

Internals Topics:

- Memory Management
- Objects
- Instruction Decoder
- Interrupts
- Virtual Devices
- External Devices
- Breakpoints and Watchpoints

2 Memory Management

Every program ever written has had to deal with memory management. Simulavr is no exception. For portability and to potentially aid in memory debugging, simulavr supplies it's own functions and macros for handling the allocation and releasing of memory resources.

For memory which could be used by many differing parts of the simulator, an object referencing system has been implemented (see Objects).

2.1 Memory Functions

The following functions provide wrappers for all library functions that return memory which simulavr must manage.

- avr_malloc()

- avr_malloc0()
- avr_realloc()
- avr_strdup()
- avr_free()

All functions which return allocated memory will only return if the allocation was successful. If the allocation failed, an error message is issued and the program is aborted. Thus, the developer does not need write any code to check the returned value.

2.2 Memory Macros

The following C-preprocessor macro definitions are provided for convenience and should be used instead of the underlying functions. These macros relieve the programmer from having to perform manual type-casting thus making the code easier to read.

- avr_new()
- avr_new0()
- avr_renew()

3 Objects

Simulavr uses a simple object oriented system for handling the data structure creation and destruction. Since simulavr is written in C, a class system must be manually implemented and the basis for this class system is the AvrClass structure. All higher level structures are ultimately based on the AvrClass structure.

How the AvrClass structure is defined is not as import as how it is used as a base or parent class structure. A concrete example of simulavr's object system will be discussed (see Derived Class Example), but before jumping into the example, the AvrClass method functions will be introduced.

3.1 AvrClass Methods

The following functions provide the user interfaces to the AvrClass structure.

- class_new()
- class_construct()

- class_destroy()
- class_overload_destroy()
- class_ref()
- class_unref()

All classes must provide their own creation function, <klass>_new(). The purpose of the creation function is to:

- Allocate memory for the class's data structure.
- Call class_overload_destroy() to install the class's own destroy method.
- Call the class's constructor method to fill in the data structure information.

3.2 Derived Class Example

Simulavr's inheritance mechanism is a little more complicated than that of C++, but is still relatively easy to use once it is understood. An example should make it clear how the system works.

First we need to create some objects. Assume that we need to add two new objects to simulavr, `foo` and `bar`. To keep things simple, they are both integers. Another requirement is that any time we need to access a `foo`, we'll also need to access a `bar`, but sometimes we only need a `bar` without a `foo`. Thus, we will have a class hierarchy `FooClass->BarClass->AvrClass`, or `FooClass` derives from `BarClass` which derives from `AvrClass`. To achieve this, we create the following two data structures:

```
// Define BarClass with AvrClass as parent

typedef struct _BarClass BarClass;
struct _BarClass {
    AvrClass parent;
    int      bar;
};

// Define FooClass with BarClass as parent

typedef struct _FooClass FooClass;
struct _FooClass {
    BarClass parent;
    int      foo;
};
```

Notice that in both struct definitions, the parent element is not a pointer. When you allocate memory for a `BarClass`, you automatically allocate memory for an `AvrClass`

at the same time. It's important that the parent is always the first element of any derived class structure.

The trick here is that once we have a class object, we can get at any object in it's class hierarchy with a simple type-cast.

```
void func( void )
{
    int num;
    FooClass *Foo = foo_new( 12, 21 );

    // get foo from FooClass
    num = Foo->foo;

    // get bar from BarClass
    num = ((BarClass *)Foo)->bar;

    class_unref( (AvrClass *)Foo );
}
```

Although the example above works, it assumes that the programmer knows what the `FooClass` and `BarClass` structures look like. The programmer has broken the encapsulation of both `FooClass` and `BarClass` objects. To solve this problem, we need to write method functions for both classes.

Here's the methods for `BarClass`:

```
// BarClass allocator
BarClass *bar_new( int bar )
{
    BarClass *bc;

    bc = avr_new( BarClass, 1 );
    bar_construct( bc, bar );
    class_overload_destroy( (AvrClass *)bc, bar_destroy );

    return bc;
}

// BarClass constructor
void bar_construct( BarClass *bc, int bar )
{
    class_construct( (AvrClass *)bc );
    bc->bar = bar;
}

// BarClass destructor
void bar_destroy( void *bc )
{
    if (bc == NULL)
        return;

    class_destroy( bc );
}
```

```
// BarClass public data access methods
int  bar_get_bar( BarClass *bc )          { return bc->bar; }
void bar_set_bar( BarClass *bc, int val ) { bc->bar = val;  }
```

And here's the methods for `FooClass`:

```
// FooClass allocator
FooClass *foo_new( int foo, int bar )
{
    FooClass *fc;

    fc = avr_new( FooClass, 1 );
    foo_construct( fc, foo, bar );
    class_overload_destroy( (AvrClass *)fc, foo_destroy );

    return fc;
}

// FooClass constructor
void foo_construct( FooClass *fc, int foo, bar )
{
    bar_construct( (BarClass *)fc, bar );
    fc->foo = foo;
}

// FooClass destructor
void foo_destroy( void *fc )
{
    if (fc == NULL)
        return;

    class_destroy( fc );
}

// FooClass public data access methods

int  foo_get_foo( FooClass *fc )          { return fc->foo; }
void foo_set_foo( FooClass *fc, int val ) { fc->foo = val;  }

int  foo_get_bar( FooClass *fc )
{
    return bar_get_bar( (BarClass *)fc );
}

void foo_set_bar( FooClass *fc, int val )
{
    bar_set_bar( (BarClass *)fc, val );
}
```

Take a good look at the `*_new()`, `*_construct()` and `*_destroy()` functions in the above examples and make sure you understand what's going on. Of particluar importance is how the constructor and destructor functions are chained up along the various classes. This pattern is used extensively throughout the simulavr source code and once understood, makes some complicated concepts incredibly easy to implement.

Now that we have the method functions, we can rewrite our original example function without the broken encapsulation.

```
void func( void )
{
    int num;
    FooClass *Foo = foo_new( 12, 21 );

    num = foo_get_foo( Foo );
    num = foo_get_bar( Foo );

    class_unref( (AvrClass *)Foo );
}
```

Now that's better, but you might think that we are breaking encapsulation when we cast `Foo` to `AvrClass`. Well, in a way we are, but since *all* class objects *must* be derived from `AvrClass` either directly or indirectly, this is acceptable.

3.3 Object Referencing

You may have noticed by this point that we haven't called avr_free() to free the memory we allocated for our objects. We called class_unref() instead. This mechanism allows us to store many references to a single object without having to keep track of all of them.

The only thing we must do when we store a reference to an object in a new variable, is call class_ref() on the object. Then, when that stored reference is no longer needed, we simply call class_unref() on the object. Once the reference count reaches zero, the object's destroy method is automatically called for us. The only hard part for us is knowing when to ref and unref the object.

Here's an example from the simulavr code for callbacks:

```
void callback_construct( CallBack *cb,
                         CallBack_FP func,
                         AvrClass *data )
{
    if (cb == NULL)
        avr_error( "passed null ptr");

    class_construct( (AvrClass *)cb );

    cb->func = func;

    cb->data = data;
    class_ref( data );
}

void callback_destroy( void *cb )
{
    CallBack *_cb = (CallBack *)cb;
```

```
    if (cb == NULL)
        return;

    class_unref( _cb->data );

    class_destroy( cb );
}
```

Notice that `data` is a pointer to `AvrClass` and thus can be any class defined by simulavr. `CallBack` is another class which happens to store a reference to `data` and must therefore call class_ref() on the `data` object. When the callback is destroyed (because the reference count reached zero), the callback destroy method calls class_unref() on the `data` object. It is assumed that the original reference to `data` still exists when the callback is created, but may or may not exist when the callback is destroyed.

4 Instruction Decoder

Instruction decoding and processing is implemented in the `decode.c` file.

The heart of the instruction decoder is the decode_opcode() function.

The decode_opcode() function examines the given opcode to determine which instruction applies and returns a pointer to a function to handle performing the instruction's operation. If the given opcode does not map to an instruction handler, `NULL` is returned indicating an invalid instruction.

Nearly every instruction in Atmel's Instruction Set Data Sheet will have a handler function defined. Each handler will perform all the operations described in the data sheet for a given instruction. A few instructions have synonyms. For example, `CBR` is a synonym for `ANDI`.

This should all be fairly straight forward.

5 Interrupts

FIXME: empty place holder

6 Virtual Devices

FIXME: empty place holder

7 External Devices

FIXME: empty place holder

8 Breakpoints and Watchpoints

Using gdb, it is possible to set breakpoints.

Watch points are not currently implemented.

This is only an idea right now: The way breakpoints are implemented within simulavr, it would be possible at init time to read a file containing breakpoint information. Then, as breakpoints are reached, have something happen which allows the user to check the state of the program. Anyone interested in implementing this, please step forward.

9 Todo List

Global avr_core_run(AvrCore *core) Should add some basic breakpoint handling here. Maybe allow continuing, and simple breakpoint management (disable, delete, set)

File ports.c Remove the pins argument and the mask field. That's handled at a higher level so is obsolete here now.

10 Deprecated List

Global mem_get_vdevice_by_name(Memory *mem, char *name)

11 File Index

11.1 File List

Here is a list of all documented files with brief descriptions:

12 File Documentation

12.1 adc.c File Reference

Module to simulate the AVR's ADC module.

Functions

- VDevice * adc_int_create (int addr, char *name, int rel_addr, void *data)
- ADCIntr_T * **adc_intr_new** (int addr, char *name, int rel_addr)
- void adc_intr_construct (ADCIntr_T *adc, int addr, char *name, int rel_addr)
- void adc_intr_destroy (void *adc)
- VDevice * adc_create (int addr, char *name, int rel_addr, void *data)
- ADC_T * **adc_new** (int addr, char *name, uint8_t uier, int rel_addr)
- void adc_construct (ADC_T *adc, int addr, char *name, uint8_t uier, int rel_addr)
- void adc_destroy (void *adc)
- uint16_t **adc_port_rd** (uint8_t mux)
- void **adc_port_wr** (uint8_t val)

12.1.1 Detailed Description

Module to simulate the AVR's ADC module.

Definition in file adc.c.

12.1.2 Function Documentation

12.1.2.1 VDevice* adc_int_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new ADC interrupt.

Definition at line 79 of file adc.c.

12.1.2.2 void adc_intr_construct (ADCIntr_T * *adc*, int *addr*, char * *name*, int *rel_addr*)

Constructor for adc interrupt object.

Definition at line 99 of file adc.c.

References avr_error, and vdev_construct().

12.1.2.3 void adc_intr_destroy (void * *adc*)

Destructor for adc interrupt object.

Definition at line 138 of file adc.c.

References vdev_destroy().

12.1.2.4 VDevice* adc_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new ADC structure.

Definition at line 292 of file adc.c.

References avr_error.

12.1.2.5 void adc_construct (ADC_T * *adc*, int *addr*, char * *name*, uint8_t *uier*, int *rel_addr*)

Constructor for ADC object.

Definition at line 318 of file adc.c.

References avr_error, and vdev_construct().

12.1.2.6 void adc_destroy (void * *adc*)

Destructor for ADC object.

Definition at line 357 of file adc.c.

References vdev_destroy().

12.2 avrclass.c File Reference

Methods to provide user interfaces to the AvrClass structure.

Functions

- AvrClass * class_new (void)
- void class_construct (AvrClass *klass)
- void class_destroy (void *klass)
- void class_overload_destroy (AvrClass *klass, AvrClassFP_Destroy destroy)
- void class_ref (AvrClass *klass)
- void class_unref (AvrClass *klass)

12.2.1 Detailed Description

Methods to provide user interfaces to the AvrClass structure. This module provides the basis for simulavr's object mechanism. For a detailed discussion on using simulavr's class mechanism, see the simulavr users manual. FIXME: [TRoth 2002/03/19] move the discussion here.

Definition in file avrclass.c.

12.2.2 Function Documentation

12.2.2.1 AvrClass* class_new (void)

This function should never be used.

The only potential use for it as a template for derived classes. Do Not Use This Function!

Definition at line 46 of file avrclass.c.

References avr_new, and class_construct().

12.2.2.2 void class_construct (AvrClass * *klass*)

Initializes the AvrClass data structure.

A derived class should call this function from their own <klass>_construct() function. All classes should have their constructor function call their parent's constructor function.

Definition at line 61 of file avrclass.c.

References avr_error, class_destroy(), and class_overload_destroy().

Referenced by class_new(), mem_construct(), stack_construct(), and vdev_construct().

12.2.2.3 void class_destroy (void * *klass*)

Releases resources allocated by class's <klass>_new() function.

This function should never be called except as the last statement of a directly derived class's destroy method. All classes should have their destroy method call their parent's destroy method.

Definition at line 78 of file avrclass.c.

References avr_free().

Referenced by avr_core_destroy(), class_construct(), mem_destroy(), stack_destroy(), and vdev_destroy().

12.2.2.4 void class_overload_destroy (AvrClass * *klass*, AvrClassFP_Destroy *destroy*)

Overload the default destroy method.

Derived classes will call this to replace class_destroy() with their own destroy method.

Definition at line 92 of file avrclass.c.

References avr_error.

Referenced by avr_core_new(), class_construct(), flash_new(), hwstack_new(), mem_new(), memstack_new(), stack_new(), and vdev_new().

12.2.2.5 void class_ref (AvrClass * *klass*)

Increments the reference count for the klass object.

The programmer must call this whenever a reference to an object is stored in more than one place.

Definition at line 106 of file avrclass.c.

References avr_error.

Referenced by mem_attach(), and memstack_construct().

12.2.2.6 void class_unref (AvrClass * *klass*)

Decrements the reference count for the klass object.

When the reference count reaches zero, the class's destroy method is called on the object.

Definition at line 120 of file avrclass.c.

References avr_error.

Referenced by avr_core_destroy(), dlist_add(), dlist_delete(), dlist_delete_all(), mem_-destroy(), and memstack_destroy().

12.3 avrcore.c File Reference

Module for the core AvrCore object, which is the AVR CPU to be simulated.

Defines

- #define **BREAK_OPCODE** 0x9598

Functions

- void avr_core_dump_core (AvrCore *core, FILE *f_core)
- int avr_core_load_program (AvrCore *core, char *file, int format)
- int avr_core_load_eeprom (AvrCore *core, char *file, int format)

AvrCore handling methods

- AvrCore * avr_core_new (char *dev_name)
- void avr_core_destroy (void *core)
- void avr_core_get_sizes (AvrCore *core, int *flash, int *sram, int *sram_start, int *eeprom)
- void avr_core_attach_vdev (AvrCore *core, uint16_t addr, char *name, VDevice *vdev, int flags, uint8_t reset_value, uint8_t rd_mask, uint8_t wr_mask)
- VDevice * avr_core_get_vdev_by_name (AvrCore *core, char *name)
- VDevice * avr_core_get_vdev_by_addr (AvrCore *core, int addr)
- void avr_core_set_state (AvrCore *core, StateType state)

- int avr_core_get_state (AvrCore *core)
- void avr_core_set_sleep_mode (AvrCore *core, int sleep_mode)
- int avr_core_get_sleep_mode (AvrCore *core)

Program Memory Space Access Methods

Data Memory Space Access Methods

Status Register Access Methods

- int avr_core_sreg_get_bit (AvrCore *core, int b)
- void avr_core_sreg_set_bit (AvrCore *core, int b, int v)

RAMPZ access methods

- uint8_t avr_core_rampz_get (AvrCore *core)
- void avr_core_rampz_set (AvrCore *core, uint8_t v)

Direct I/O Register Access Methods

IO Registers are mapped in memory directly after the 32 (0x20) general registers.

- void avr_core_io_display_names (AvrCore *core)
- uint8_t avr_core_io_read (AvrCore *core, int reg)
- void avr_core_io_write (AvrCore *core, int reg, uint8_t val)

Stack Methods

- uint32_t avr_core_stack_pop (AvrCore *core, int bytes)
- void avr_core_stack_push (AvrCore *core, int bytes, uint32_t val)

Program Counter Methods

- int32_t avr_core_PC_size (AvrCore *core)
- void avr_core_PC_incr (AvrCore *core, int val)

Methods for accessing CK and instruction Clocks

- uint64_t avr_core_CK_get (AvrCore *core)
- void avr_core_CK_incr (AvrCore *core)
- int avr_core_inst_CKS_get (AvrCore *core)
- void avr_core_inst_CKS_set (AvrCore *core, int val)

Interrupt Access Methods.

- IntVect * avr_core_irq_get_pending (AvrCore *core)
- void avr_core_irq_raise (AvrCore *core, unsigned int irq)

- void avr_core_irq_clear (AvrCore *core, IntVect *irq)
- void avr_core_irq_clear_all (AvrCore *core)

Break point access methods.

- void avr_core_insert_breakpoint (AvrCore *core, int pc)
- void avr_core_remove_breakpoint (AvrCore *core, int pc)
- void avr_core_disable_breakpoints (AvrCore *core)
- void avr_core_enable_breakpoints (AvrCore *core)

Program control methods

- int avr_core_step (AvrCore *core)
- void avr_core_run (AvrCore *core)
- void avr_core_reset (AvrCore *core)

Callback Handling Methods

- void avr_core_add_ext_rd_wr (AvrCore *core, int addr, PortFP_ExtRd ext_rd, PortFP_ExtWr ext_wr)
- void avr_core_clk_cb_add (AvrCore *core, CallBack *cb)
- void avr_core_async_cb_add (AvrCore *core, CallBack *cb)
- void avr_core_clk_cb_exec (AvrCore *core)
- void avr_core_async_cb_exec (AvrCore *core)

Variables

- int global_debug_inst_output = 0

12.3.1 Detailed Description

Module for the core AvrCore object, which is the AVR CPU to be simulated.

Definition in file avrcore.c.

12.3.2 Function Documentation

12.3.2.1 AvrCore* avr_core_new (char * *dev_name*)

Allocate a new AvrCore object.

Definition at line 355 of file avrcore.c.

References avr_core_destroy(), avr_new, class_overload_destroy(), and dev_supp_lookup_device().

12.3.2.2 void avr_core_destroy (void * *core*)

Destructor for the AvrCore class.

Not to be called directly, except by a derived class. Called via class_unref.

Definition at line 549 of file avrcore.c.

References class_destroy(), class_unref(), and dlist_delete_all().

Referenced by avr_core_new().

12.3.2.3 void avr_core_get_sizes (AvrCore * *core*, int * *flash*, int * *sram*, int * *sram_start*, int * *eeprom*)

Query the sizes of the 3 memory spaces: flash, sram, and eeprom.

Definition at line 572 of file avrcore.c.

References flash_get_size().

12.3.2.4 void avr_core_attach_vdev (AvrCore * *core*, uint16_t *addr*, char * *name*, VDevice * *vdev*, int *flags*, uint8_t *reset_value*, uint8_t *rd_mask*, uint8_t *wr_mask*)

Attach a virtual device into the Memory.

12.3.2.5 VDevice* avr_core_get_vdev_by_name (AvrCore * *core*, char * *name*)

Returns the `VDevice` with the name *name*.

12.3.2.6 VDevice* avr_core_get_vdev_by_addr (AvrCore * *core*, int *addr*)

Returns the `VDevice` which handles the address *addr*.

12.3.2.7 void avr_core_set_state (AvrCore * *core*, StateType *state*)

Sets the device's state (running, stopped, breakpoint, sleep).

12.3.2.8 int avr_core_get_state (AvrCore * *core*)

Returns the device's state (running, stopped, breakpoint, sleep).

Referenced by avr_core_step().

12.3.2.9 void avr_core_set_sleep_mode (AvrCore * *core*, int *sleep_mode*)

Sets the device to a sleep state.

Parameters

core	Pointer to the core.
sleep_mode	The BITNUMBER of the sleepstate.

12.3.2.10 int avr_core_get_sleep_mode (AvrCore * *core*)

Return the device's sleepmode.

12.3.2.11 int avr_core_sreg_get_bit (AvrCore * *core*, int *b*)

Get the value of bit `b` of the status register.

12.3.2.12 void avr_core_sreg_set_bit (AvrCore * *core*, int *b*, int *v*)

Set the value of bit `b` of the status register.

12.3.2.13 uint8_t avr_core_rampz_get (AvrCore * *core*)

Get the value of the rampz register.

12.3.2.14 void avr_core_rampz_set (AvrCore * *core*, uint8_t *v*)

Set the value of the rampz register.

12.3.2.15 void avr_core_io_display_names (AvrCore * *core*)

Displays all registers.

Definition at line 733 of file avrcore.c.

References display_io_reg_name(), and mem_io_fetch().

12.3.2.16 uint8_t avr_core_io_read (AvrCore * *core*, int *reg*)

Reads the value of a register.

Parameters

core	Pointer to the core.
reg	The registers address. This address is counted above the beginning of the registers memory block (0x20).

12.3.2.17 void avr_core_io_write (AvrCore * *core*, int *reg*, uint8_t *val*)

Writes the value of a register. See avr_core_io_read() for a discussion of *reg*.

12.3.2.18 uint32_t avr_core_stack_pop (AvrCore * *core*, int *bytes*)

Pop 1-4 bytes off of the stack.

See stack_pop() for more details.

12.3.2.19 void avr_core_stack_push (AvrCore * *core*, int *bytes*, uint32_t *val*)

Push 1-4 bytes onto the stack.

See stack_push() for more details.

12.3.2.20 int32_t avr_core_PC_size (AvrCore * *core*)

Returns the size of the Program Counter in bytes.

Most devices have a 16-bit PC (2 bytes), but some larger ones (e.g. mega256), have a 22-bit PC (3 bytes).

12.3.2.21 void avr_core_PC_incr (AvrCore * *core*, int *val*)

Increment the Program Counter by val.

val can be either positive or negative.

If the result of the incrememt is outside the valid range for PC, it is adjusted to fall in the valid range. This allows addresses to wrap around the end of the insn space.

12.3.2.22 uint64_t avr_core_CK_get (AvrCore * *core*)

Get the current clock counter.

Referenced by avr_core_run().

12.3.2.23 void avr_core_CK_incr (AvrCore * *core*)

Increment the clock counter.

Referenced by avr_core_step().

12.3.2.24 int avr_core_inst_CKS_get (AvrCore * *core*)

Get the number of clock cycles remaining for the currently executing instruction.

12.3.2.25 void avr_core_inst_CKS_set (AvrCore * *core*, int *val*)

Set the number of clock cycles for the instruction being executed.

Referenced by avr_core_reset().

12.3.2.26 IntVect* avr_core_irq_get_pending (AvrCore * *core*)

Gets the first pending irq.

Definition at line 846 of file avrcore.c.

12.3.2.27 void avr_core_irq_raise (AvrCore * *core*, unsigned int *irq*)

Raises an irq by adding it's data to the irq_pending list.

Definition at line 854 of file avrcore.c.

References avr_message.

12.3.2.28 void avr_core_irq_clear (AvrCore * *core*, IntVect * *irq*)

Calls the interrupt's callback to clear the flag.

Definition at line 867 of file avrcore.c.

12.3.2.29 void avr_core_irq_clear_all (AvrCore * *core*)

Removes all irqs from the irq_pending list.

Referenced by avr_core_reset().

12.3.2.30 void avr_core_insert_breakpoint (AvrCore * *core*, int *pc*)

Inserts a break point.

Definition at line 884 of file avrcore.c.

References flash_read(), and flash_write().

12.3.2.31 void avr_core_remove_breakpoint (AvrCore * *core*, int *pc*)

Removes a break point.

Definition at line 898 of file avrcore.c.

References flash_write().

12.3.2.32 void avr_core_disable_breakpoints (AvrCore * *core*)

Disable breakpoints.

Disables all breakpoints that where set using avr_core_insert_breakpoint(). The breakpoints are not removed from the breakpoint list.

Definition at line 955 of file avrcore.c.

12.3.2.33 void avr_core_enable_breakpoints (AvrCore * *core*)

Enable breakpoints.

Enables all breakpoints that where previous disabled.

Definition at line 968 of file avrcore.c.

12.3.2.34 int avr_core_step (AvrCore * *core*)

Process a single program instruction, all side effects and peripheral stimulii.

Executes instructions, calls callbacks, and checks for interrupts.

Definition at line 1093 of file avrcore.c.

References avr_core_async_cb_exec(), avr_core_CK_incr(), avr_core_clk_cb_exec(), and avr_core_get_state().

Referenced by avr_core_run().

12.3.2.35 void avr_core_run (AvrCore * *core*)

Start the processing of instructions by the simulator.

The simulated device will run until one of the following occurs:

- The state of the core is no longer STATE_RUNNING.
- The simulator receives a SIGINT signal.
- A breakpoint is reached (currently causes core to stop running).
- A fatal internal error occurs.

Note

When running simulavr in gdb server mode, this function is not used. The avr_core_step() function is called repeatedly in a loop when the continue command is issued from gdb. As such, the functionality in this loop should be kept to a minimum.

Todo

Should add some basic breakpoint handling here. Maybe allow continuing, and simple breakpoint management (disable, delete, set)

Definition at line 1150 of file avrcore.c.

References avr_core_CK_get(), avr_core_reset(), avr_core_step(), avr_message, get_program_time(), signal_has_occurred(), signal_watch_start(), and signal_watch_stop().

12.3.2.36 void avr_core_reset (AvrCore * *core*)

Sets the simulated CPU back to its initial state.

Zeroes out PC, IRQ's, clock, and memory.

Definition at line 1204 of file avrcore.c.

References avr_core_inst_CKS_set(), avr_core_irq_clear_all(), display_clock(), and mem_reset().

Referenced by avr_core_run().

12.3.2.37 void avr_core_add_ext_rd_wr (AvrCore * *core*, int *addr*, PortFP_ExtRd *ext_rd*, PortFP_ExtWr *ext_wr*)

For adding external read and write callback functions.

rd and wr should come in pairs, but it is safe to add empty function via passing a NULL pointer for either function.

Parameters

core	A pointer to an AvrCore object.
port_id	The ID for handling the simulavr inheritance model.
ext_rd	Function for the device core to call when it needs to communicate with the external world via I/O Ports.
ext_wr	Function for the device core to call when it needs to communicate with the external world via I/O Ports.

Definition at line 1244 of file avrcore.c.

References avr_warning, mem_get_vdevice_by_addr(), and port_add_ext_rd_wr().

12.3.2.38 void avr_core_clk_cb_add (AvrCore * *core*, CallBack * *cb*)

Add a new clock callback to list.

12.3.2.39 void avr_core_async_cb_add (AvrCore * *core*, CallBack * *cb*)

Add a new asynchronous callback to list.

12.3.2.40 void avr_core_clk_cb_exec (AvrCore * *core*)

Run all the callbacks in the list.

If a callback returns non-zero (true), then it is done with it's job and wishes to be removed from the list.

The time argument has dual meaning. If the callback list is for the clock callbacks, time is the value of the CK clock counter. If the callback list is for the asynchronous callback, time is the number of milliseconds from some unknown, arbitrary time on the host system.

Referenced by avr_core_step().

12.3.2.41 void avr_core_async_cb_exec (AvrCore * *core*)

Run all the asynchronous callbacks.

Referenced by avr_core_step().

12.3.2.42 void avr_core_dump_core (AvrCore * *core*, FILE * *f_core*)

Dump the contents of the entire CPU core.

Parameters

core	A pointer to an AvrCore object.
f_core	An open file descriptor.

Definition at line 1294 of file avrcore.c.

References flash_dump_core(), and mem_dump_core().

12.3.2.43 int avr_core_load_program (AvrCore * *core*, char * *file*, int *format*)

Load a program from an input file.

Definition at line 1307 of file avrcore.c.

References flash_load_from_file().

12.3.2.44 int avr_core_load_eeprom (AvrCore * *core*, char * *file*, int *format*)

Load a program from an input file.

Definition at line 1314 of file avrcore.c.

References mem_get_vdevice_by_name().

12.3.3 Variable Documentation

12.3.3.1 int global_debug_inst_output = 0

Flag for enabling output of instruction debug messages.

Definition at line 64 of file avrcore.c.

12.4 avrerror.c File Reference

Functions for printing messages, warnings and errors.

Defines

- #define avr_message(fmt, args...) private_avr_message(__FILE__, __LINE__, fmt, ## args)
- #define avr_warning(fmt, args...) private_avr_warning(__FILE__, __LINE__-, fmt, ## args)
- #define avr_error(fmt, args...) private_avr_error(__FILE__, __LINE__, fmt, ## args)

12.4.1 Detailed Description

Functions for printing messages, warnings and errors. This module provides output printing facilities.

Definition in file avrerror.c.

12.4.2 Define Documentation

12.4.2.1 #define avr_message(*fmt, args...*) private_avr_message(__FILE__, __LINE__, fmt, ## args)

Print an ordinary message to stdout.

Definition at line 42 of file avrerror.c.

Referenced by avr_core_irq_raise(), avr_core_run(), and decode_init_lookup_table().

12.4.2.2 #define avr_warning(*fmt, args...*) private_avr_warning(__FILE__, __LINE__, fmt, ## args)

Print a warning message to stderr.

Definition at line 46 of file avrerror.c.

Referenced by avr_core_add_ext_rd_wr(), display_open(), display_send_msg(), flash_load_from_file(), mem_read(), mem_write(), signal_has_occurred(), signal_watch_start(), signal_watch_stop(), vdev_add_addr(), and vdev_def_AddAddr().

12.4.2.3 #define avr_error(*fmt, args...*) private_avr_error(__FILE__, __LINE__, fmt, ## args)

Print an error message to stderr and terminate program.

Definition at line 50 of file avrerror.c.

Referenced by adc_construct(), adc_create(), adc_intr_construct(), avr_malloc(), avr_malloc0(), avr_realloc(), avr_strdup(), class_construct(), class_overload_destroy(), class_ref(), class_unref(), display_eeprom(), display_flash(), display_send_msg(), display_sram(), dlist_delete(), dlist_iterator(), dlist_lookup(), flash_construct(), gdb_interact(), get_program_time(), hwstack_construct(), mem_attach(), mem_construct(), mem_get_vdevice_by_name(), memstack_construct(), ocreg16_construct(), ocreg16_create(), spi_construct(), spi_intr_construct(), stack_construct(), timer0_construct(), timer16_construct(), timer16_create(), timer_int_create(), timer_intr_construct(), uart_construct(), uart_int_create(), uart_intr_construct(), usb_construct(), usb_intr_construct(), usbi_create(), and vdev_construct().

12.5 avrmalloc.c File Reference

Memory Management Functions.

Defines

- #define avr_new(type, count) ((type *) avr_malloc ((unsigned) sizeof (type) * (count)))
- #define avr_new0(type, count) ((type *) avr_malloc0 ((unsigned) sizeof (type) * (count)))
- #define avr_renew(type, mem, count) ((type *) avr_realloc (mem, (unsigned) sizeof (type) * (count)))

Functions

- void * avr_malloc (size_t size)
- void * avr_malloc0 (size_t size)
- void * avr_realloc (void *ptr, size_t size)
- char * avr_strdup (const char *s)
- void avr_free (void *ptr)

12.5.1 Detailed Description

Memory Management Functions. This module provides facilities for managing memory.

There is no need to check the returned values from any of these functions. Any memory allocation failure is considered fatal and the program is terminated.

We want to wrap all functions that allocate memory. This way we can add secret code to track memory usage and debug memory leaks if we want. Right now, I don't want to ;).

Definition in file avrmalloc.c.

12.5.2 Define Documentation

12.5.2.1 #define avr_new(*type, count*) ((type *) avr_malloc ((unsigned) sizeof (type) * (count)))

Macro for allocating memory.

Parameters

type	The C type of the memory to allocate.
count	Allocate enough memory hold count types.

This macro is just a wrapper for avr_malloc() and should be used to avoid the repetitive task of casting the returned pointer.

Definition at line 57 of file avrmalloc.c.

Referenced by avr_core_new(), class_new(), flash_new(), hwstack_new(), memstack_new(), and stack_new().

12.5.2.2 #define avr_new0(*type, count*) ((type *) avr_malloc0 ((unsigned) sizeof (type) * (count)))

Macro for allocating memory and initializing it to zero.

Parameters

type	The C type of the memory to allocate.
count	Allocate enough memory hold count types.

This macro is just a wrapper for avr_malloc0() and should be used to avoid the repetitive task of casting the returned pointer.

Definition at line 67 of file avrmalloc.c.

Referenced by decode_init_lookup_table(), display_send_msg(), hwstack_construct(), mem_construct(), mem_new(), and vdev_new().

12.5.2.3 #define avr_renew(*type, mem, count*) ((type *) avr_realloc (mem, (unsigned) sizeof (type) * (count)))

Macro for allocating memory.

Parameters

type	The C type of the memory to allocate.
mem	Pointer to existing memory.
count	Allocate enough memory hold count types.

This macro is just a wrapper for avr_malloc() and should be used to avoid the repetitive task of casting the returned pointer.

Definition at line 78 of file avrmalloc.c.

12.5.3 Function Documentation

12.5.3.1 void* avr_malloc (size_t *size*)

Allocate memory and initialize to zero.

Use the avr_new() macro instead of this function.

There is no need to check the returned value, since this function will terminate the program if the memory allocation fails.

No memory is allocated if passed a size of zero.

Definition at line 93 of file avrmalloc.c.

References avr_error.

12.5.3.2 void* avr_malloc0 (size_t *size*)

Allocate memory and initialize to zero.

Use the avr_new0() macro instead of this function.

There is no need to check the returned value, since this function will terminate the program if the memory allocation fails.

No memory is allocated if passed a size of zero.

Definition at line 117 of file avrmalloc.c.

References avr_error.

12.5.3.3 void* avr_realloc (void * *ptr*, size_t *size*)

Wrapper for realloc(). x Resizes and possibly allocates more memory for an existing memory block.

Use the avr_renew() macro instead of this function.

There is no need to check the returned value, since this function will terminate the program if the memory allocation fails.

No memory is allocated if passed a size of zero.

Definition at line 143 of file avrmalloc.c.

References avr_error.

12.5.3.4 char* avr_strdup (const char * *s*)

Wrapper for strdup().

Returns a copy of the passed in string. The returned copy must be free'd.

There is no need to check the returned value, since this function will terminate the program if the memory allocation fails.

It is safe to pass a NULL pointer. No memory is allocated if a NULL is passed.

Definition at line 168 of file avrmalloc.c.

References avr_error.

12.5.3.5 void avr_free (void * *ptr*)

Free malloc'd memory.

It is safe to pass a null pointer to this function.

Definition at line 187 of file avrmalloc.c.

Referenced by class_destroy(), display_send_msg(), hwstack_destroy(), and mem_-destroy().

12.6 decoder.c File Reference

Module for handling opcode decoding.

Enumerations

- enum decoder_operand_masks {
  mask_Rd_2 = 0x0030,
  mask_Rd_3 = 0x0070,
  mask_Rd_4 = 0x00f0,
  mask_Rd_5 = 0x01f0,
  mask_Rr_3 = 0x0007,
  mask_Rr_4 = 0x000f,
  mask_Rr_5 = 0x020f,
  mask_K_8 = 0x0F0F,
  mask_K_6 = 0x00CF,
  mask_k_7 = 0x03F8,
  mask_k_12 = 0x0FFF,
  mask_k_22 = 0x01F1,
  mask_reg_bit = 0x0007,
  mask_sreg_bit = 0x0070,
  mask_q_displ = 0x2C07,
  mask_A_5 = 0x00F8,
  mask_A_6 = 0x060F }

Functions

- int **avr_op_UNKNOWN** (AvrCore *core, uint16_t opcode, unsigned int arg1, unsigned int arg2)
- void decode_init_lookup_table (void)
- struct opcode_info * decode_opcode (uint16_t opcode)

Variables

- struct opcode_info * **global_opcode_lookup_table**

12.6.1 Detailed Description

Module for handling opcode decoding. The heart of the instruction decoder is the decode_opcode() function.

The decode_opcode() function examines the given opcode to determine which instruction applies and returns a pointer to a function to handler performing the instruction's operation. If the given opcode does not map to an instruction handler, NULL is returned.

Nearly every instruction in Atmel's Instruction Set Data Sheet will have a handler function defined. Each handler will perform all the operations described in the data sheet for a given instruction. A few instructions have synonyms. For example, CBR is a synonym for ANDI.

This should all be fairly straight forward.

Definition in file decoder.c.

12.6.2 Enumeration Type Documentation

12.6.2.1 enum decoder_operand_masks

Masks to help extracting information from opcodes.

Enumerator:

mask_Rd_2 2 bit register id (R24, R26, R28, R30)

mask_Rd_3 3 bit register id (R16 - R23)

mask_Rd_4 4 bit register id (R16 - R31)

mask_Rd_5 5 bit register id (R00 - R31)

mask_Rr_3 3 bit register id (R16 - R23)

mask_Rr_4 4 bit register id (R16 - R31)

mask_Rr_5 5 bit register id (R00 - R31)

mask_K_8 for 8 bit constant

mask_K_6 for 6 bit constant

mask_k_7 for 7 bit relative address

mask_k_12 for 12 bit relative address

mask_k_22 for 22 bit absolute address

mask_reg_bit register bit select

mask_sreg_bit status register bit select

mask_q_displ address displacement (q)

mask_A_5 5 bit register id (R00 - R31)

mask_A_6 6 bit IO port id

Definition at line 77 of file decoder.c.

12.6.3 Function Documentation

12.6.3.1 void decode_init_lookup_table (void)

Initialize the decoder lookup table.

This is automatically called by avr_core_construct().

It is safe to call this function many times, since if will only create the table the first time it is called.

Definition at line 3869 of file decoder.c.

References avr_message, and avr_new0.

12.6.3.2 struct opcode_info* decode_opcode (uint16_t *opcode*) [read]

Decode an opcode into the opcode handler function.

Generates a warning and returns NULL if opcode is invalid.

Returns a pointer to the function to handle the opcode.

12.7 device.c File Reference

VDevice methods.

Functions

- VDevice * vdev_new (char *name, VDevFP_Read rd, VDevFP_Write wr, VDevFP_-Reset reset, VDevFP_AddAddr add_addr)

- void vdev_def_AddAddr (VDevice ∗dev, int addr, char ∗name, int related_addr, void ∗data)
- void vdev_construct (VDevice ∗dev, VDevFP_Read rd, VDevFP_Write wr, VDevFP_-Reset reset, VDevFP_AddAddr add_addr)
- void vdev_destroy (void ∗dev)
- uint8_t vdev_read (VDevice ∗dev, int addr)
- void vdev_write (VDevice ∗dev, int addr, uint8_t val)
- void vdev_reset (VDevice ∗dev)
- void vdev_set_core (VDevice ∗dev, AvrClass ∗core)
- AvrClass ∗ vdev_get_core (VDevice ∗dev)
- void vdev_add_addr (VDevice ∗dev, int addr, char ∗name, int rel_addr, void ∗data)

12.7.1 Detailed Description

VDevice methods. These functions are the base for all other devices mapped into the device space.

Definition in file device.c.

12.7.2 Function Documentation

12.7.2.1 VDevice∗ vdev_new (char ∗ *name,* VDevFP_Read *rd,* VDevFP_Write *wr,* VDevFP_Reset *reset,* VDevFP_AddAddr *add_addr*)

Create a new VDevice.

Definition at line 62 of file device.c.

References avr_new0, class_overload_destroy(), vdev_construct(), and vdev_destroy().

12.7.2.2 void vdev_def_AddAddr (VDevice ∗ *dev,* int *addr,* char ∗ *name,* int *related_addr,* void ∗ *data*)

Default AddAddr method.

This generate a warning that the should let the developer know that the vdev needs to be updated.

Definition at line 80 of file device.c.

References avr_warning.

12.7.2.3 void vdev_construct (VDevice * *dev*, VDevFP_Read *rd*, VDevFP_Write *wr*, VDevFP_Reset *reset*, VDevFP_AddAddr *add_addr*)

Constructor for a VDevice.

Definition at line 89 of file device.c.

References avr_error, and class_construct().

Referenced by adc_construct(), adc_intr_construct(), ocreg16_construct(), spi_construct(), spi_intr_construct(), timer0_construct(), timer16_construct(), timer_intr_construct(), uart_construct(), uart_intr_construct(), usb_construct(), usb_intr_construct(), and vdev_new().

12.7.2.4 void vdev_destroy (void * *dev*)

Destructor for a VDevice.

Definition at line 105 of file device.c.

References class_destroy().

Referenced by adc_destroy(), adc_intr_destroy(), spi_destroy(), spi_intr_destroy(), timer0_destroy(), timer_intr_destroy(), uart_destroy(), uart_intr_destroy(), usb_destroy(), usb_intr_destroy(), and vdev_new().

12.7.2.5 uint8_t vdev_read (VDevice * *dev*, int *addr*)

Reads the device's value in the register at *addr*.

Definition at line 161 of file device.c.

Referenced by mem_io_fetch(), and mem_read().

12.7.2.6 void vdev_write (VDevice * *dev*, int *addr*, uint8_t *val*)

Writes an value to the register at *addr*.

Definition at line 168 of file device.c.

Referenced by mem_write().

12.7.2.7 void vdev_reset (VDevice * *dev*)

Resets a device.

Definition at line 175 of file device.c.

Referenced by mem_reset().

12.7.2.8 void vdev_set_core (VDevice * *dev*, AvrClass * *core*)

Set the core field.

Definition at line 182 of file device.c.

12.7.2.9 AvrClass* vdev_get_core (VDevice * *dev*)

Get the core field.

12.7.2.10 void vdev_add_addr (VDevice * *dev*, int *addr*, char * *name*, int *rel_addr*, void * *data*)

Inform the vdevice that it needs to handle another address.

This is primarily used when creating the core in dev_supp_create_core().

Definition at line 195 of file device.c.

References avr_warning.

12.8 devsupp.c File Reference

Contains definitions for device types (i.e. at90s8515, at90s2313, etc.)

Functions

- int **dev_supp_has_ext_io_reg** (DevSuppDefn *dev)
- int **dev_supp_get_flash_sz** (DevSuppDefn *dev)
- int **dev_supp_get_PC_sz** (DevSuppDefn *dev)
- int **dev_supp_get_stack_sz** (DevSuppDefn *dev)

- int **dev_supp_get_vtab_idx** (DevSuppDefn ∗dev)
- int **dev_supp_get_sram_sz** (DevSuppDefn ∗dev)
- int **dev_supp_get_eeprom_sz** (DevSuppDefn ∗dev)
- int **dev_supp_get_xram_sz** (DevSuppDefn ∗dev)
- DevSuppDefn ∗ dev_supp_lookup_device (char ∗dev_name)
- void dev_supp_list_devices (FILE ∗fp)
- void **dev_supp_attach_io_regs** (AvrCore ∗core, DevSuppDefn ∗dev)

12.8.1 Detailed Description

Contains definitions for device types (i.e. at90s8515, at90s2313, etc.) This module is used to define the attributes for each device in the AVR family. A generic constructor is used to create a new AvrCore object with the proper ports, built-in peripherals, memory layout, registers, and interrupt vectors, etc.

Definition in file devsupp.c.

12.8.2 Function Documentation

12.8.2.1 DevSuppDefn∗ dev_supp_lookup_device (char ∗ *dev_name*)

Look up a device name in support list.

Returns

An opaque pointer to DevSuppDefn or NULL if not found.

Definition at line 272 of file devsupp.c.

Referenced by avr_core_new().

12.8.2.2 void dev_supp_list_devices (FILE ∗ *fp*)

Print a list of supported devices to a file pointer.

Definition at line 292 of file devsupp.c.

12.9 display.c File Reference

Interface for using display coprocesses.

Enumerations

- enum { **MAX_BUF** = 1024 }

Functions

- int display_open (char *prog, int no_xterm, int flash_sz, int sram_sz, int sram_start, int eeprom_sz)
- void display_close (void)
- void display_send_msg (char *msg)
- void display_clock (int clock)
- void display_pc (int val)
- void display_reg (int reg, uint8_t val)
- void display_io_reg (int reg, uint8_t val)
- void display_io_reg_name (int reg, char *name)
- void display_flash (int addr, int len, uint16_t *vals)
- void display_sram (int addr, int len, uint8_t *vals)
- void display_eeprom (int addr, int len, uint8_t *vals)

12.9.1 Detailed Description

Interface for using display coprocesses. Simulavr has the ability to use a coprocess to display register and memory values in near real time.

Definition in file display.c.

12.9.2 Function Documentation

12.9.2.1 int display_open (char * *prog*, int *no_xterm*, int *flash_sz*, int *sram_sz*, int *sram_start*, int *eeprom_sz*)

Open a display as a coprocess.

Parameters

prog	The program to use as a display coprocess.
no_xterm	If non-zero, don't run the disply in an xterm.
flash_sz	The size of the flash memory space in bytes.
sram_sz	The size of the sram memory space in bytes.
sram_start	The addr of the first byte of sram (usually 0x60 or 0x100).
eeprom_sz	The size of the eeprom memory space in bytes.

Try to start up a helper program as a child process for displaying registers and memory. If the prog argument is NULL, don't start up a display.

Returns an open file descriptor of a pipe used to send data to the helper program.

Returns -1 if something failed.

Definition at line 85 of file display.c.

References avr_warning.

12.9.2.2 void display_close (void)

Close a display and send coprocess a quit message.

Definition at line 181 of file display.c.

References display_send_msg().

12.9.2.3 void display_send_msg (char * *msg*)

Encode the message and send to display.

Parameters

msg	The message string to be sent to the display process.

Encoding is the same as that used by the gdb remote protocol: '$...#CC' where '...' is msg, CC is checksum. There is no newline termination for encoded messages.

FIXME: TRoth: This should be a private function. It is only public so that dtest.c can be kept simple. dtest.c should be changed to avoid direct use of this function. [dtest.c has served it's purpose and will be retired soon.]

Definition at line 220 of file display.c.

References avr_error, avr_free(), avr_new0, and avr_warning.

Referenced by display_clock(), display_close(), display_eeprom(), display_flash(), display_io_reg(), display_io_reg_name(), display_pc(), display_reg(), and display_sram().

12.9.2.4 void display_clock (int *clock*)

Update the time in the display.

Parameters

clock	The new time in number of clocks.

Definition at line 255 of file display.c.

References display_send_msg().

Referenced by avr_core_reset().

12.9.2.5 void display_pc (int *val*)

Update the Program Counter in the display.

Parameters

val	The new value of the program counter.

Definition at line 269 of file display.c.

References display_send_msg().

12.9.2.6 void display_reg (int *reg*, uint8_t *val*)

Update a register in the display.

Parameters

reg	The register number.
val	The new value of the register.

Definition at line 284 of file display.c.

References display_send_msg().

12.9.2.7 void display_io_reg (int *reg*, uint8_t *val*)

Update an IO register in the display.

Parameters

reg	The IO register number.
val	The new value of the register.

Definition at line 299 of file display.c.

References display_send_msg().

Referenced by mem_write().

12.9.2.8 void display_io_reg_name (int *reg*, char * *name*)

Specify a name for an IO register.

Parameters

reg	The IO register number.
name	The symbolic name of the register.

Names of IO registers may be different from device to device.

Definition at line 316 of file display.c.

References display_send_msg().

Referenced by avr_core_io_display_names().

12.9.2.9 void display_flash (int *addr*, int *len*, uint16_t * *vals*)

Update a block of flash addresses in the display.

Parameters

addr	Address of beginning of the block.
len	Length of the block (number of words).
vals	Pointer to an array of *len* words.

The display will update each addr of the block to the coresponding value in the *vals* array.

Each address in the flash references a single 16-bit wide word (or opcode or instruction). Therefore, flash addresses are aligned to 16-bit boundaries. It is simplest to consider the flash an array of 16-bit values indexed by the address.

Definition at line 340 of file display.c.

References avr_error, and display_send_msg().

Referenced by flash_write().

12.9.2.10 void display_sram (int *addr*, int *len*, uint8_t * *vals*)

Update a block of sram addresses in the display.

Parameters

addr	Address of beginning of the block.
len	Length of the block (number of bytes).
vals	Pointer to an array of *len* bytes.

The display will update each addr of the block to the coresponding value in the *vals* array.

Definition at line 372 of file display.c.

References avr_error, and display_send_msg().

12.9.2.11 void display_eeprom (int *addr*, int *len*, uint8_t * *vals*)

Update a block of eeprom addresses in the display.

Parameters

addr	Address of beginning of the block.
len	Length of the block (number of bytes).
vals	Pointer to an array of *len* bytes.

The display will update each addr of the block to the coresponding value in the *vals* array.

Definition at line 404 of file display.c.

References avr_error, and display_send_msg().

12.10 flash.c File Reference

Flash memory methods.

Functions

- uint16_t flash_read (Flash *flash, int addr)
- void flash_write (Flash *flash, int addr, uint16_t val)
- void flash_write_lo8 (Flash *flash, int addr, uint8_t val)

- void flash_write_hi8 (Flash *flash, int addr, uint8_t val)
- Flash * flash_new (int size)
- void flash_construct (Flash *flash, int size)
- void flash_destroy (void *flash)
- int flash_load_from_file (Flash *flash, char *file, int format)
- int flash_get_size (Flash *flash)
- void flash_dump_core (Flash *flash, FILE *f_core)

12.10.1 Detailed Description

Flash memory methods. This module provides functions for reading and writing to flash memory. Flash memory is the program (.text) memory in AVR's Harvard architecture. It is completely separate from RAM, which is simulated in the memory.c file.

Definition in file flash.c.

12.10.2 Function Documentation

12.10.2.1 uint16_t flash_read (Flash * *flash*, int *addr*)

Reads a 16-bit word from flash.

Returns

A word.

Referenced by avr_core_insert_breakpoint(), and flash_dump_core().

12.10.2.2 void flash_write (Flash * *flash*, int *addr*, uint16_t *val*)

Reads a 16-bit word from flash.

Parameters

flash	A pointer to a flash object.
addr	The address to which to write.
val	The byte to write there.

Definition at line 86 of file flash.c.

References display_flash().

Referenced by avr_core_insert_breakpoint(), and avr_core_remove_breakpoint().

12.10.2.3 void flash_write_lo8 (Flash * *flash*, int *addr*, uint8_t *val*)

Write the low-order byte of an address.

AVRs are little-endian, so lo8 bits in odd addresses.

Definition at line 98 of file flash.c.

12.10.2.4 void flash_write_hi8 (Flash * *flash*, int *addr*, uint8_t *val*)

Write the high-order byte of an address.

AVRs are little-endian, so hi8 bits in even addresses.

Definition at line 109 of file flash.c.

12.10.2.5 Flash* flash_new (int *size*)

Allocate a new Flash object.

Definition at line 117 of file flash.c.

References avr_new, class_overload_destroy(), flash_construct(), and flash_destroy().

12.10.2.6 void flash_construct (Flash * *flash*, int *size*)

Constructor for the flash object.

Definition at line 131 of file flash.c.

References avr_error.

Referenced by flash_new().

12.10.2.7 void flash_destroy (void * *flash*)

Destructor for the flash class.

Not to be called directly, except by a derived class. Called via class_unref.

Definition at line 153 of file flash.c.

Referenced by flash_new().

12.10.2.8 int flash_load_from_file (Flash * *flash*, char * *file*, int *format*)

Load program data into flash from a file.

Definition at line 164 of file flash.c.

References avr_warning.

Referenced by avr_core_load_program().

12.10.2.9 int flash_get_size (Flash * *flash*)

Accessor method to get the size of a flash.

Definition at line 210 of file flash.c.

Referenced by avr_core_get_sizes().

12.10.2.10 void flash_dump_core (Flash * *flash*, FILE * *f_core*)

Dump the contents of the flash to a file descriptor in text format.

Parameters

flash	A pointer to a flash object.
f_core	An open file descriptor.

Definition at line 223 of file flash.c.

References flash_read().

Referenced by avr_core_dump_core().

12.11 gdbserver.c File Reference

Provide an interface to gdb's remote serial protocol.

Functions

- void gdb_interact (GdbComm_T *comm, int port, int debug_on)

12.11.1 Detailed Description

Provide an interface to gdb's remote serial protocol. This module allows a program to be used by gdb as a remote target. The remote target and gdb communicate via gdb's remote serial protocol. The protocol is documented in the gdb manual and will not be repeated here.

Hitting Ctrl-c in gdb can be used to interrupt the remote target while it is processing instructions and return control back to gdb.

Issuing a 'signal SIGxxx' command from gdb will send the signal to the remote target via a "continue with signal" packet. The target will process and interpret the signal, but not pass it on to the AVR program running in the target since it really makes no sense to do so. In some circumstances, it may make sense to use the gdb signal mechanism as a way to initiate some sort of external stimulus to be passed on to the virtual hardware system.

Signals from gdb which are processed have the following meanings:

- `SIGHUP` Initiate a reset of the target. (Simulates a hardware reset)

Definition in file gdbserver.c.

12.11.2 Function Documentation

12.11.2.1 void gdb_interact (GdbComm_T * *comm*, int *port*, int *debug_on*)

Start interacting with gdb.

Parameters

comm	A previously initialized simulator comm
port	Port which server will listen for connections on.
debug_on	Turn on gdb debug diagnostic messages.

Start a tcp server socket on localhost listening for connections on the given port. Once a connection is established, enter an infinite loop and process command requests from gdb using the remote serial protocol. Only a single connection is allowed at a time.

Definition at line 1320 of file gdbserver.c.

References avr_error, signal_has_occurred(), signal_watch_start(), and signal_watch_-stop().

12.12 memory.c File Reference

Memory access functions.

Functions

- Memory * mem_new (int gpwr_end, int io_reg_end, int sram_end, int xram_-end)
- void mem_construct (Memory *mem, int gpwr_end, int io_reg_end, int sram_-end, int xram_end)
- void mem_destroy (void *mem)
- void mem_attach (Memory *mem, int addr, char *name, VDevice *vdev, int flags, uint8_t reset_value, uint8_t rd_mask, uint8_t wr_mask)
- VDevice * mem_get_vdevice_by_addr (Memory *mem, int addr)
- VDevice * mem_get_vdevice_by_name (Memory *mem, char *name)
- void **mem_set_addr_name** (Memory *mem, int addr, char *name)
- uint8_t mem_read (Memory *mem, int addr)
- void mem_write (Memory *mem, int addr, uint8_t val)
- void mem_reset (Memory *mem)
- void mem_io_fetch (Memory *mem, int addr, uint8_t *val, char *buf, int bufsiz)
- void mem_dump_core (Memory *mem, FILE *f_core)

12.12.1 Detailed Description

Memory access functions. This module provides functions for reading and writing to simulated memory. The Memory class is a subclass of AvrClass.

Definition in file memory.c.

12.12.2 Function Documentation

12.12.2.1 Memory* mem_new (int *gpwr_end*, int *io_reg_end*, int *sram_end*, int *xram_end*)

Allocates memory for a new memory object.

Definition at line 66 of file memory.c.

References avr_new0, class_overload_destroy(), mem_construct(), and mem_destroy().

12.12.2.2 void mem_construct (Memory * *mem*, int *gpwr_end*, int *io_reg_end*, int *sram_end*, int *xram_end*)

Constructor for the memory object.

Definition at line 80 of file memory.c.

References avr_error, avr_new0, and class_construct().

Referenced by mem_new().

12.12.2.3 void mem_destroy (void * *mem*)

Descructor for the memory object.

Definition at line 99 of file memory.c.

References avr_free(), class_destroy(), and class_unref().

Referenced by mem_new().

12.12.2.4 void mem_attach (Memory * *mem*, int *addr*, char * *name*, VDevice * *vdev*, int *flags*, uint8_t *reset_value*, uint8_t *rd_mask*, uint8_t *wr_mask*)

Attach a device to the device list.

Devices that are accessed more often should be attached last so that they will be at the front of the list.

A default virtual device can be overridden by attaching a new device ahead of it in the list.

Definition at line 130 of file memory.c.

References avr_error, and class_ref().

12.12.2.5 VDevice* mem_get_vdevice_by_addr (Memory * *mem*, int *addr*)

Find the VDevice associated with the given address.

Definition at line 159 of file memory.c.

Referenced by avr_core_add_ext_rd_wr(), and memstack_construct().

12.12.2.6 VDevice* mem_get_vdevice_by_name (Memory * *mem*, char * *name*)

Find the VDevice associated with the given name.

Deprecated

Definition at line 169 of file memory.c.

References avr_error, and dlist_lookup().

Referenced by avr_core_load_eeprom().

12.12.2.7 uint8_t mem_read (Memory * *mem*, int *addr*)

Reads byte from memory and sanity-checks for valid address.

Parameters

mem	A pointer to the memory object
addr	The address to be read

Returns

The byte found at that address addr

Definition at line 220 of file memory.c.

References avr_warning, and vdev_read().

12.12.2.8 void mem_write (Memory * *mem*, int *addr*, uint8_t *val*)

Writes byte to memory and updates display for io registers.

Parameters

mem	A pointer to a memory object
addr	The address to be written to
val	The value to be written there

Definition at line 255 of file memory.c.

References avr_warning, display_io_reg(), and vdev_write().

12.12.2.9 void mem_reset (Memory * *mem*)

Resets every device in the memory object.

Parameters

mem	A pointer to the memory object.

Definition at line 292 of file memory.c.

References vdev_reset().

Referenced by avr_core_reset().

12.12.2.10 void mem_io_fetch (Memory * *mem*, int *addr*, uint8_t * *val*, char * *buf*, int *bufsiz*)

Fetch the name and value of the io register (addr).

Parameters

mem	A pointer to the memory object.
addr	The address to fetch from.
val	A pointer where the value of the register is to be copied.
buf	A pointer to where the name of the register should be copied.
bufsiz	The maximum size of the the buf string.

Definition at line 330 of file memory.c.

References vdev_read().

Referenced by avr_core_io_display_names().

12.12.2.11 void mem_dump_core (Memory * *mem*, FILE * *f_core*)

Dump all the various memory locations to a file descriptor in text format.

Parameters

mem	A memory object.
f_core	An open file descriptor.

Definition at line 491 of file memory.c.

Referenced by avr_core_dump_core().

12.13 ports.c File Reference

Module for accessing simulated I/O ports.

Functions

- VDevice * port_create (int addr, char *name, int rel_addr, void *data)
- void port_ext_disable (Port *p)
- void port_ext_enable (Port *p)
- void port_add_ext_rd_wr (Port *p, PortFP_ExtRd ext_rd, PortFP_ExtWr ext_wr)

12.13.1 Detailed Description

Module for accessing simulated I/O ports. Defines an abstract Port class as well as subclasses for each individual port.

Todo

Remove the pins argument and the mask field. That's handled at a higher level so is obsolete here now.

Definition in file ports.c.

12.13.2 Function Documentation

12.13.2.1 VDevice* port_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Create a new Port instance.

This should only be used in DevSuppDefn initializers.

Definition at line 100 of file ports.c.

12.13.2.2 void port_ext_disable (Port * p)

Disable external port functionality.

This is only used when dumping memory to core file. See mem_io_fetch().

Definition at line 200 of file ports.c.

12.13.2.3 void port_ext_enable (Port * p)

Enable external port functionality.

This is only used when dumping memory to core file. See mem_io_fetch().

Definition at line 210 of file ports.c.

12.13.2.4 void port_add_ext_rd_wr (Port * p, PortFP_ExtRd ext_rd, PortFP_ExtWr ext_wr)

Attaches read and write functions to a particular port.

I think I may have this backwards. Having the virtual hardware supply functions for the core to call on every io read/write, might cause missed events (like edge triggered). I'm really not too sure how to handle this.

In the future, it might be better to have the core supply a function for the virtual hardware to call when data is written to the device. The device supplied function could then check if an interrupt should be generated or just simply write to the port data register.

For now, leave it as is since it's easier to test if you can block when the device is reading from the virtual hardware.

Definition at line 231 of file ports.c.

Referenced by avr_core_add_ext_rd_wr().

12.14 sig.c File Reference

Public interface to signal handlers.

Functions

- void signal_watch_start (int signo)

- void signal_watch_stop (int signo)
- int signal_has_occurred (int signo)
- void signal_reset (int signo)

12.14.1 Detailed Description

Public interface to signal handlers. This module provides a way for the simulator to process signals generated by the native host system. Note that these signals in this context have nothing to do with signals or interrupts as far as a program running in the simulator is concerned.

Definition in file sig.c.

12.14.2 Function Documentation

12.14.2.1 void signal_watch_start (int *signo*)

Start watching for the occurrance of the given signal.

This function will install a signal handler which will set a flag when the signal occurs. Once the watch has been started, periodically call signal_has_occurred() to check if the signal was raised.

Definition at line 67 of file sig.c.

References avr_warning.

Referenced by avr_core_run(), and gdb_interact().

12.14.2.2 void signal_watch_stop (int *signo*)

Stop watching signal.

Restores the default signal handler for the given signal and resets the signal flag.

Definition at line 97 of file sig.c.

References avr_warning, and signal_reset().

Referenced by avr_core_run(), and gdb_interact().

12.14.2.3 int signal_has_occurred (int *signo*)

Check to see if a signal has occurred.

Returns

Non-zero if signal has occurred. The flag will always be reset automatically.

Definition at line 119 of file sig.c.

References avr_warning.

Referenced by avr_core_run(), gdb_interact(), and signal_reset().

12.14.2.4 void signal_reset (int *signo*)

Clear the flag which indicates that a signal has ocurred.

Use signal_reset to manually reset (i.e. clear) the flag.

Definition at line 142 of file sig.c.

References signal_has_occurred().

Referenced by signal_watch_stop().

12.15 spi.c File Reference

Module to simulate the AVR's SPI module.

Functions

- VDevice * spii_create (int addr, char *name, int rel_addr, void *data)
- SPIIntr_T * **spi_intr_new** (int addr, char *name)
- void spi_intr_construct (SPIIntr_T *spi, int addr, char *name)
- void spi_intr_destroy (void *spi)
- VDevice * spi_create (int addr, char *name, int rel_addr, void *data)
- SPI_T * **spi_new** (int addr, char *name, int rel_addr)
- void spi_construct (SPI_T *spi, int addr, char *name, int rel_addr)
- void spi_destroy (void *spi)
- uint8_t **spi_port_rd** (int addr)
- void **spi_port_wr** (uint8_t val)

12.15.1 Detailed Description

Module to simulate the AVR's SPI module.

Definition in file spi.c.

12.15.2 Function Documentation

12.15.2.1 VDevice* spii_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new SPI interrupt.

Definition at line 78 of file spi.c.

12.15.2.2 void spi_intr_construct (SPIIntr_T * *spi*, int *addr*, char * *name*)

Constructor for spi interrupt object.

Definition at line 98 of file spi.c.

References avr_error, and vdev_construct().

12.15.2.3 void spi_intr_destroy (void * *spi*)

Destructor for spi interrupt object.

Definition at line 134 of file spi.c.

References vdev_destroy().

12.15.2.4 VDevice* spi_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new SPI structure.

Definition at line 247 of file spi.c.

12.15.2.5 void spi_construct (SPI_T * *spi*, int *addr*, char * *name*, int *rel_addr*)

Constructor for SPI object.

Definition at line 267 of file spi.c.

References avr_error, and vdev_construct().

12.15.2.6 void spi_destroy (void * *spi*)

Destructor for SPI object.

Definition at line 300 of file spi.c.

References vdev_destroy().

12.16 stack.c File Reference

Module for the definition of the stack.

Functions

- Stack * stack_new (StackFP_Pop pop, StackFP_Push push)
- void stack_construct (Stack *stack, StackFP_Pop pop, StackFP_Push push)
- void stack_destroy (void *stack)
- uint32_t stack_pop (Stack *stack, int bytes)
- void stack_push (Stack *stack, int bytes, uint32_t val)
- HWStack * hwstack_new (int depth)
- void hwstack_construct (HWStack *stack, int depth)
- void hwstack_destroy (void *stack)
- VDevice * sp_create (int addr, char *name, int rel_addr, void *data)
- MemStack * memstack_new (Memory *mem, int spl_addr)
- void memstack_construct (MemStack *stack, Memory *mem, int spl_addr)
- void memstack_destroy (void *stack)

12.16.1 Detailed Description

Module for the definition of the stack. Defines the classes stack, hw_stack, and mem_-stack.

FIXME: Ted, I would really really really love to put in a description of what is the difference between these three classes and how they're used, but I don't understand it myself.

Definition in file stack.c.

12.16.2 Function Documentation

12.16.2.1 Stack* stack_new (StackFP_Pop *pop*, StackFP_Push *push*)

Allocates memory for a new Stack object.

This is a virtual method for higher level stack implementations and as such should not be used directly.

Definition at line 82 of file stack.c.

References avr_new, class_overload_destroy(), stack_construct(), and stack_destroy().

12.16.2.2 void stack_construct (Stack * *stack*, StackFP_Pop *pop*, StackFP_Push *push*)

Constructor for the Stack class.

This is a virtual method for higher level stack implementations and as such should not be used directly.

Definition at line 99 of file stack.c.

References avr_error, and class_construct().

Referenced by hwstack_construct(), memstack_construct(), and stack_new().

12.16.2.3 void stack_destroy (void * *stack*)

Destructor for the Stack class.

This is a virtual method for higher level stack implementations and as such should not be used directly.

Definition at line 116 of file stack.c.

References class_destroy().

Referenced by hwstack_destroy(), memstack_destroy(), and stack_new().

12.16.2.4 uint32_t stack_pop (Stack * *stack*, int *bytes*)

Pops a byte or a word off the stack and returns it.

Parameters

stack	A pointer to the Stack object from which to pop
bytes	Number of bytes to pop off the stack (1 to 4 bytes).

Returns

The 1 to 4 bytes value popped from the stack.

This method provides access to the derived class's pop() method.

Definition at line 133 of file stack.c.

12.16.2.5 void stack_push (Stack * *stack*, int *bytes*, uint32_t *val*)

Pushes a byte or a word of data onto the stack.

Parameters

stack	A pointer to the Stack object from which to pop.
bytes	Size of the value being pushed onto the stack (1 to 4 bytes).
val	The value to be pushed.

This method provides access to the derived class's push() method.

Definition at line 146 of file stack.c.

12.16.2.6 HWStack* hwstack_new (int *depth*)

Allocate a new HWStack object.

This is the stack implementation used by devices which lack SRAM and only have a fixed size hardware stack (e.i., the at90s1200)

Definition at line 163 of file stack.c.

References avr_new, class_overload_destroy(), hwstack_construct(), and hwstack_-destroy().

12.16.2.7 void hwstack_construct (HWStack * *stack*, int *depth*)

Constructor for HWStack object.

Definition at line 177 of file stack.c.

References avr_error, avr_new0, and stack_construct().

Referenced by hwstack_new().

12.16.2.8 void hwstack_destroy (void * *stack*)

Destructor for HWStack object.

Definition at line 191 of file stack.c.

References avr_free(), and stack_destroy().

Referenced by hwstack_new().

12.16.2.9 VDevice* sp_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Create the Stack Pointer VDevice.

This should only be used in the DevSuppDefn io reg init structure.

Definition at line 272 of file stack.c.

12.16.2.10 MemStack* memstack_new (Memory * *mem*, int *spl_addr*)

Allocate a new MemStack object.

Definition at line 388 of file stack.c.

References avr_new, class_overload_destroy(), memstack_construct(), and memstack_-destroy().

12.16.2.11 void memstack_construct (MemStack * *stack*, Memory * *mem*, int *spl_addr*)

Constructor for MemStack object.

Definition at line 402 of file stack.c.

References avr_error, class_ref(), mem_get_vdevice_by_addr(), and stack_construct().

Referenced by memstack_new().

12.16.2.12 void memstack_destroy (void * *stack*)

Destructor for MemStack object.

Definition at line 423 of file stack.c.

References class_unref(), and stack_destroy().

Referenced by memstack_new().

12.17 timers.c File Reference

Module to simulate the AVR's on-board timer/counters.

Functions

- VDevice * timer_int_create (int addr, char *name, int rel_addr, void *data)
- TimerIntr_T * **timer_intr_new** (int addr, char *name, uint8_t func_mask)
- void timer_intr_construct (TimerIntr_T *ti, int addr, char *name, uint8_t func_-mask)
- void timer_intr_destroy (void *ti)
- VDevice * timer0_create (int addr, char *name, int rel_addr, void *data)
- Timer0_T * **timer0_new** (int addr, char *name, int rel_addr)
- void timer0_construct (Timer0_T *timer, int addr, char *name, int rel_addr)
- void timer0_destroy (void *timer)

16 Bit Timer Functions

- VDevice * timer16_create (int addr, char *name, int rel_addr, void *data)
- Timer16_T * **timer16_new** (int addr, char *name, int rel_addr, Timer16Def timerdef)
- void timer16_construct (Timer16_T *timer, int addr, char *name, int rel_addr, Timer16Def timerdef)

16 Bit Output Compare Register Functions

- VDevice * ocreg16_create (int addr, char *name, int rel_addr, void *data)
- OCReg16_T * **ocreg16_new** (int addr, char *name, OCReg16Def ocrdef)
- void ocreg16_construct (OCReg16_T *ocreg, int addr, char *name, OCReg16Def ocrdef)

12.17.1 Detailed Description

Module to simulate the AVR's on-board timer/counters. This currently only implements the timer/counter 0.

Definition in file timers.c.

12.17.2 Function Documentation

12.17.2.1 VDevice* timer_int_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new timer interrupt.

Definition at line 138 of file timers.c.

References avr_error.

12.17.2.2 void timer_intr_construct (TimerIntr_T * *ti*, int *addr*, char * *name*, uint8_t *func_mask*)

Constructor for timer interrupt object.

Definition at line 163 of file timers.c.

References avr_error, and vdev_construct().

12.17.2.3 void timer_intr_destroy (void * *ti*)

Destructor for timer interrupt object.

Definition at line 205 of file timers.c.

References vdev_destroy().

12.17.2.4 VDevice* timer0_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new timer/counter 0.

Definition at line 350 of file timers.c.

12.17.2.5 void timer0_construct (Timer0_T * *timer*, int *addr*, char * *name*, int *rel_addr*)

Constructor for timer/counter 0 object.

Definition at line 370 of file timers.c.

References avr_error, and vdev_construct().

12.17.2.6 void timer0_destroy (void * *timer*)

Destructor for timer/counter 0 object.

Definition at line 387 of file timers.c.

References vdev_destroy().

12.17.2.7 VDevice* timer16_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new 16 bit timer/counter.

Definition at line 582 of file timers.c.

References avr_error.

12.17.2.8 void timer16_construct (Timer16_T * *timer*, int *addr*, char * *name*, int *rel_addr*, Timer16Def *timerdef*)

Constructor for 16 bit timer/counter object.

Definition at line 608 of file timers.c.

References avr_error, and vdev_construct().

12.17.2.9 VDevice* ocreg16_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new 16 bit Output Compare Register.

Parameters

ocrdef	The definition struct for the *OCR* to be created

Definition at line 909 of file timers.c.

References avr_error.

12.17.2.10 void ocreg16_construct (OCReg16_T * *ocreg*, int *addr*, char * *name*, OCReg16Def *ocrdef*)

Constructor for 16 bit Output Compare Register object.

Definition at line 935 of file timers.c.

References avr_error, and vdev_construct().

12.18 uart.c File Reference

Module to simulate the AVR's uart module.

Functions

- VDevice * uart_int_create (int addr, char *name, int rel_addr, void *data)
- UARTIntr_T * **uart_intr_new** (int addr, char *name, void *data)
- void uart_intr_construct (UARTIntr_T *uart, int addr, char *name)
- void uart_intr_destroy (void *uart)
- VDevice * uart_create (int addr, char *name, int rel_addr, void *data)
- UART_T * **uart_new** (int addr, char *name, int rel_addr)
- void uart_construct (UART_T *uart, int addr, char *name, int rel_addr)
- void uart_destroy (void *uart)
- uint16_t **uart_port_rd** (int addr)
- void **uart_port_wr** (uint8_t val)

Variables

- unsigned int **UART_Int_Table** [ ]
- unsigned int **UART0_Int_Table** [ ]
- unsigned int **UART1_Int_Table** [ ]

12.18.1 Detailed Description

Module to simulate the AVR's uart module.

Definition in file uart.c.

12.18.2 Function Documentation

12.18.2.1 VDevice* uart_int_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new uart interrupt.

Definition at line 95 of file uart.c.

References avr_error.

12.18.2.2 void uart_intr_construct (UARTIntr_T * *uart*, int *addr*, char * *name*)

Constructor for uart interrupt object.

Definition at line 128 of file uart.c.

References avr_error, and vdev_construct().

12.18.2.3 void uart_intr_destroy (void * *uart*)

Destructor for uart interrupt object.

Definition at line 179 of file uart.c.

References vdev_destroy().

12.18.2.4 VDevice* uart_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new uart structure.

Definition at line 335 of file uart.c.

12.18.2.5 void uart_construct (UART_T * *uart*, int *addr*, char * *name*, int *rel_addr*)

Constructor for uart object.

Definition at line 356 of file uart.c.

References avr_error, and vdev_construct().

12.18.2.6 void uart_destroy (void * *uart*)

Destructor for uart object.

Definition at line 389 of file uart.c.

References vdev_destroy().

12.18.3 Variable Documentation

12.18.3.1 unsigned int UART_Int_Table[ ]

Initial value:

```
 {
    irq_vect_table_index (UART_RX),
    irq_vect_table_index (UART_UDRE),
    irq_vect_table_index (UART_TX)
}
```

Definition at line 74 of file uart.c.

12.18.3.2 unsigned int UART0_Int_Table[ ]

Initial value:

```
 {
    irq_vect_table_index (USART0_RX),
    irq_vect_table_index (USART0_UDRE),
    irq_vect_table_index (USART0_TX)
}
```

Definition at line 80 of file uart.c.

12.18.3.3 unsigned int UART1_Int_Table[ ]

Initial value:

```
 {
    irq_vect_table_index (USART1_RX),
    irq_vect_table_index (USART1_UDRE),
    irq_vect_table_index (USART1_TX)
}
```

Definition at line 86 of file uart.c.

12.19 usb.c File Reference

Module to simulate the AVR's USB module.

Functions

- void **usb_port_wr** (char *name, uint8_t val)
- uint8_t **usb_port_rd** (char *name)
- VDevice * usbi_create (int addr, char *name, int rel_addr, void *data)
- USBInter_T * **usb_intr_new** (int addr, char *name, uint8_t func_mask)
- void usb_intr_construct (USBInter_T *usb, int addr, char *name, uint8_t func_mask)
- void usb_intr_destroy (void *usb)
- VDevice * usb_create (int addr, char *name, int rel_addr, void *data)
- USB_T * **usb_new** (int addr, char *name)
- void usb_construct (USB_T *usb, int addr, char *name)
- void usb_destroy (void *usb)

12.19.1 Detailed Description

Module to simulate the AVR's USB module.

Definition in file usb.c.

12.19.2 Function Documentation

12.19.2.1 VDevice* usbi_create (int *addr*, char * *name*, int *rel_addr*, void * *data*)

Allocate a new USB interrupt.

Definition at line 78 of file usb.c.

References avr_error.

12.19.2.2 void usb_intr_construct (USBInter_T * *usb*, int *addr*, char * *name*, uint8_t *func_mask*)

Constructor for usb interrupt object.

Definition at line 103 of file usb.c.

References avr_error, and vdev_construct().

12.19.2.3 void usb_intr_destroy (void * *usb*)

Destructor for usb interrupt object.

Definition at line 181 of file usb.c.

References vdev_destroy().

12.19.2.4 VDevice* usb_create (int *addr,* char * *name,* int *rel_addr,* void * *data*)

Allocate a new USB structure.

Definition at line 301 of file usb.c.

12.19.2.5 void usb_construct (USB_T * *usb,* int *addr,* char * *name*)

Constructor for new USB object.

Definition at line 321 of file usb.c.

References avr_error, and vdev_construct().

12.19.2.6 void usb_destroy (void * *usb*)

Destructor for USB object.

Definition at line 480 of file usb.c.

References vdev_destroy().

12.20 utils.c File Reference

Utility functions.

Functions

- int str2ffmt (char ∗str)
- uint8_t set_bit_in_byte (uint8_t src, int bit, int val)
- uint16_t set_bit_in_word (uint16_t src, int bit, int val)
- uint64_t get_program_time (void)
- DList ∗ dlist_add (DList ∗head, AvrClass ∗data, DListFP_Cmp cmp)
- DList ∗ dlist_add_head (DList ∗head, AvrClass ∗data)
- DList ∗ dlist_delete (DList ∗head, AvrClass ∗data, DListFP_Cmp cmp)
- void dlist_delete_all (DList ∗head)
- AvrClass ∗ dlist_lookup (DList ∗head, AvrClass ∗data, DListFP_Cmp cmp)
- AvrClass ∗ dlist_get_head_data (DList ∗head)
- DList ∗ dlist_iterator (DList ∗head, DListFP_Iter func, void ∗user_data)

12.20.1 Detailed Description

Utility functions. This module provides general purpose utilities.

Definition in file utils.c.

12.20.2 Function Documentation

12.20.2.1 int str2ffmt (char ∗ *str*)

Utility function to convert a string to a FileFormatType code.

Definition at line 50 of file utils.c.

12.20.2.2 uint8_t set_bit_in_byte (uint8_t *src*, int *bit*, int *val*)

Set a bit in src to 1 if val != 0, clears bit if val == 0.

12.20.2.3 uint16_t set_bit_in_word (uint16_t *src*, int *bit*, int *val*)

Set a bit in src to 1 if val != 0, clears bit if val == 0.

12.20.2.4 uint64_t get_program_time (void)

Return the number of milliseconds of elapsed program time.

Returns

an unsigned 64 bit number. Time zero is not well defined, so only time differences should be used.

Definition at line 76 of file utils.c.

References avr_error.

Referenced by avr_core_run().

12.20.2.5 DList* dlist_add (DList * *head*, AvrClass * *data*, DListFP_Cmp *cmp*)

Add a new node to the end of the list.

If cmp argument is not NULL, use cmp() to see if node already exists and don't add node if it exists.

It is the responsibility of this function to unref data if not added.

Definition at line 159 of file utils.c.

References class_unref().

12.20.2.6 DList* dlist_add_head (DList * *head*, AvrClass * *data*)

Add a new node at the head of the list.

Definition at line 196 of file utils.c.

12.20.2.7 DList* dlist_delete (DList * *head*, AvrClass * *data*, DListFP_Cmp *cmp*)

Conditionally delete a node from the list.

Delete a node from the list if the node's data matches the specified data. Returns the head of the modified list.

Definition at line 215 of file utils.c.

References avr_error, and class_unref().

Referenced by dlist_iterator().

12.20.2.8 void dlist_delete_all (DList * *head*)

Blow away the entire list.

Definition at line 266 of file utils.c.

References class_unref().

Referenced by avr_core_destroy().

12.20.2.9 AvrClass* dlist_lookup (DList * *head*, AvrClass * *data*, DListFP_Cmp *cmp*)

Lookup an item in the list.

Walk the list pointed to by head and return a pointer to the data if found. If not found, return NULL.

Parameters

head	The head of the list to be iterated.
data	The data to be passed to the func when it is applied.
cmp	A function to be used for comparing the items.

Returns

A pointer to the data found, or NULL if not found.

Definition at line 291 of file utils.c.

References avr_error.

Referenced by mem_get_vdevice_by_name().

12.20.2.10 AvrClass* dlist_get_head_data (DList * *head*)

Extract the data from the head of the list.

Returns the data element for the head of the list. If the list is empty, return a NULL pointer.

Parameters

head	The head of the list.

Returns

A pointer to the data found, or NULL if not found.

Definition at line 321 of file utils.c.

12.20.2.11 DList* dlist_iterator (DList * *head*, DListFP_Iter *func*, void * *user_data*)

Iterate over all elements of the list.

For each element, call the user supplied iterator function and pass it the node data and the user_data. If the iterator function return non-zero, remove the node from the list.

Parameters

head	The head of the list to be iterated.
func	The function to be applied.
user_data	The data to be passed to the func when it is applied.

Returns

A pointer to the head of the possibly modified list.

Definition at line 357 of file utils.c.

References avr_error, and dlist_delete().

Index