ThinkOS - User Guide

Bob Mittmann
bobmittmann@gmail.com

January 2017

Revision 0.2

Contents

List of Figures

List of Tables

Revision History

Revision	Date	Author(s)	Description
0.1	21.01.17	BM	created
0.2	22.03.18	BM	patially fixed redundant sections

1 Introduction

THIS user guide provides an overview of the **ThinkOS** real-time operating system. The target audience for this guide are embedded software developers knowledgeable in C language. Notice that no mention of C++ is made in this text.

ThinkOS was designed specifically for the **Cortex-M** family of microprocessors from **ARM**. These are all 32bits CPUs with mostly 16bits instruction set, which produces very compact code, hence suitable for deep embedded systems with stringent memory and other peripheral resources constraints.

At this moment **ThinkOS** supports **M3**, **M4** and **M7** cores.

The popular **M0** family lacks some features needed for some operations in the kernel. Porting to these platforms is feasible but needs someone with time to do that. Any volunteer?

ThinkOS is a Real Time Operating System designed for the ARM Cortex-M core. It takes advantage of some unique features of this processor to enable for very low task switch latency.

1.1 Why Another OS?

The real answer for this question perhaps is simply: "Just because I can". But as most people won't be satisfied with this shallow reply here are some justifications:

1.2 Why Cortex-M

The ARM Cortex-M family of microprocessors is the standard 32 bits CPU for most of the microcontrollers in the market today.

1.3 Can it be ported to another processor?

In theory yes. But probably its not worth the effort...

1.4 Where do I can get ThinkOS from?

Repository: [https://github.com/bobmittmann/thinkos]
Git HTTPS: [https://github.com/bobmittmann/thinkos.git]

1.5 How to use it?

How and what is need in order to compile **ThinkOS** ....

- Download the source code
- Install the Tools
- Configure / or chose and example
- Compile
- Load and run

1.6 Performance

The main bottleneck for performance is usually the scheduler. The ThinkOS running in a 120MHz Cortex-M3 (STM32F270) have a measured latency time of 0.5 microseconds.

There are some tests and calculations that need to be done to determine the latency of other elements of the system...

1.7 Source Code

One feature that distinguishes the **ThinkOS** from most of other Operating Systems is the way the control structures for the several types of objects are held by kernel.

The usual approach for holding this information is to allocate a structure specific for a certain object and pass a pointer to the kernel. The kernel then stores this in a list or array.

ThinkOS in other hand holds the core data structure in a Structure of Arrays (SOA) representation. The cost of this representation is the readability of the code, (it may look a little obscure at the beginning). The idea here is simplify the race condition avoidance in the kernel and speed-up the scheduler.

1.7.1 Coding Style

ThinkOS source uses the Linux kernel coding style and I encourage all developers to follow it. This is just little more than a matter of preference, please read the style guide to know the rational behind the rules before despising them :) ...

1.8 Compiling

There are basically two ways of using **ThinkOS** :

As a library linked against your application. (Need a term to decribe this like embedded mode...). In this type of usage there is usually no bootloader, the application boots and configure itself. In this scenario the firmware is the application. The advatnages of this mode are:

- Smallest footprint. The linker will strip away unused (unreferenced) portions of the kernel.
- Kernel calls and services are tailored for the application.

As part of the bootloader separated from the application. This mode will be called ”standalone” mode, or ”bootloader” mode. In this case the application will be a separated entity (file, binary) and run in a separated memory space. The bootloader will be residing on the platform at all times and will be responsible to initialize the platform and to load the application. It’s possible then to upload upgrade and change the application without changing the bootlader and consequently the operating system. Advantages:

- No need to configure the OS (need to elaborate...) to match the application. At least initially.
- In field firmware upgrades are safer as the bootloader is the responsible for the upgrade.
- The Debug Monitor features can be used for development.

- Kernel data and code memory are protected.

1.9 Modules

Overview of the different modules.

- Kernel
- Debug Monitor
- GDB server
- Console

1.10 Threads

ThinkOS is a preemptive time-sharing kernel. Each thread has its own stack and a timer. This timer is used to block the thread's execution for a certain period of time as in `thinkos_sleep()`.

1.11 Scheduler

ThinkOS scheduler was designed to reduce the context switch time to a minimum. The most important piece on the Kernel is the Scheduler....

1.12 Time Sharing

All threads with priority higher than 0 go into a round-robin time sharing scheduling policy, when this feature is enabled. Their execution time will be inversely proportional to their priority.

1.13 Interrupt Model

There are two types of interrupt handlers with ThinkOS:

Implicit ISR Thread interrupt handlers that executes as normal threads. This is a simple call to `thinkos_irq_wait(IRQ)` inside a regular task or function. The thread will block until this interrupt is raised. This mechanism provides a convenient and powerful tool to design drivers. Devices that benefit most of this type of ISR are simplex I/O channels like I2C or SPI, hardware accelerators like cryptographic ...

Native ISR Normal interrupt handlers that shares a common stack. These handlers are subject to to some constraints regarding API calls. It is recommended to use this type of interrupt handler when low latency is a must. But bear in mind that if the handler is too complex the actual latency due to loading registers and housekeeping may be equivalent to a context switch in which case it will be equivalent of a threaded handler in terms of performance. Another case will be a high priority interrupt you want to make sure it will won't be preempted...

Configuration is simply a matter of creating your custom configuration header file: `config.h` and enabling or disabling the features the application requires.

The file `include/thinkos_sys.h` contains the definition of the structures used by the OS. In this file there is a series of macros used to set default values for the configuration options.

It is not recommended to change the values directly in `thinkos_sys.h`, create a `config.h` file instead, and define the macro `CONFIG_H` at the compiler's: `-DCONFIG_H`.

Note: if you are using the **ThinkOS** as part of the YARD-ICE there is a `config.h` file located at the source root.

2 Quick Start

This guide will help you to install the necessary tools to compile and run a ThinkOS demo application on a development board.

2.1 Windows Tools

These are the basic steps to follow in order to install the tools:

- Install GNU Toolchain for ARM Processros for Windows hosts.
- Install GNU Make for Windows.
- Install GNU CoreUtils for Windows.
- Install the Java JRE from Oracle.
- Install the Eclipse for C++ Developers IDE.
- Install the GNU ARM Eclipse Plug-ins.
- Install the C/C++ GDB Hardware Debugging Plug-in.
- Install the STM32 ST-LINK Utility from STMicroelectronics.

Detailed instructions for each one of the steps can be found in the next sections of this document.

2.2 Demo Application

There are some demo applications using **ThinkOS** in the source tree. This is a guide on how to compile and run ThinkOS programs in the STM32F3Discovery board.

- Get the **ThinkOS** Source Code from GitHub.
- Import the Eclipse projects for the target board into your workspace.
- Compile the **ThinkOS** BootLoader project using Eclipse IDE.
- Load the **ThinkOS** BootLoader on the board using the STM32 ST-LINK Utility.
- Install the Virtual COM Driver on the PC to access the ThinkOS Debug-Monitor.
- Compile the SDK Libraries (sdklib) for the target board using Eclipse.
- Compile the demo application project using Eclipse.
- Run the application with GDB Remote Debugging.

2.3 ThinkOS Basics

In this tutorial the most basic **ThinkOS** functions are described with some code examples. This is a summary of what it will be covered in this section:

- How to create and run threads.
- Create, lock and unlock mutexes.
- Create, signal and wait on semaphores.
- Wait for time intervals (sleep).
- Create a new thread

2.4 Creating a Thread

The easiest way to create and run a thread is to call `thinkos_thread_create()`. Here is an example:

Listing 1: Simple Thread Create Example

```
static int my_task(void * arg)
{
    for (;;) {
        /* do something ... */
         thinkos_sleep(50);
    }
    return 0;
}

static uint32_t my_stack[256]; /* 1KB of stack is enough for most porposes */

void my_module_init(void)
{
    thinkos_thread_create(my_task, NULL, my_stack, sizeof(my_stack));
}
```

In this example we define a function to be the entry point of the thread, in this case `my_task()`. We also define an array of 32bit words, amounting to 1KB, to be used as stack for the thread (`my_stack`). We then call `thinkos_thread_create()` which will allocate and run the thread. The return of this function is the thread id or a negative error code in case something goes wrong.

2.4.1 Using semaphores

ThinkOS semaphores allow processes and threads to synchronize their actions.

A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (`thinkos_sem_post()`); and decrement the semaphore value by one (`thinkos_sem_wait()`). If the value of a semaphore is currently zero, then a `thinkos_sem_wait()` operation will block until the value becomes greater than zero. A semaphore should be allocated with `thinkos_sem_alloc()` which will return a descriptor to be used in subsequent calls to `thinkos_sem_post()`, and `thinkos_sem_wait()`.

In the example below, two treads are created: a producer and a consumer. The producer signals the semaphore periodically twice a second, whereas the consumer waits for the signal and prints a message when receives it.

Listing 2: Using semaphores example

```
/* Global semaphore descriptor */

int my_sem;

/* Producer thread */
static int producer_task(void * arg)
{
    for (;;) {
        /* Wait 500 milliseconds... */
        thinkos_sleep(500);
        /* Signal the semaphore. */
        thinkos_sem_post(my_sem);
    }
    return 0;
}

/* Consumer thread */
static int consumer_task(void * arg)
{
    for (;;) {
        /* Wait on the semaphore... */
        thinkos_sem_wait(my_sem);
        /* Do something... */
        printf("Signal received\n");
    }
    return 0;
}


static uint32_t consumer_stack[256];
static uint32_t producer_stack[256];

void my_test_init(void)
{
    /* Allocate a semaphore with 0 initial count. */
    my_sem = thinkos_sem_alloc(0);
    /* Create a thread to signal the semaphore */
    thinkos_thread_create(producer_task, NULL, producer_stack, sizeof(
        producer_stack));
    /* Create a thread to wait on the semaphore */
    thinkos_thread_create(consumer_task, NULL, consumer_stack, sizeof(
        consumer_stack));
}
```

2.4.2 Using MUTEXes

A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

A call to `thinkos_mutex_lock()` locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and `thinkos_mutex_lock()` returns imme-

diately. If the mutex is already locked by another thread, `thinkos_mutex_lock()` suspends the calling thread until the mutex is unlocked by calling `thinkos_mutex_unlock()`. A mutex should be allocated with `thinkos_mutex_alloc()` which will return a descriptor to be used in subsequent mutex calls.

In the following example the serial port is a resource that needs to be protected to ensure the packets are transmitted in full.

Listing 3: Using MutExes examples

```
/* Link Layer private data */
static struct {
    uint8_t addr; /* this panel address */


    struct serial_dev * serial; /* serial device driver */
    int mutex; /* protect the link layer send call */
} rs485;

void netlnk_send(unsigned int daddr, void * data, unsigned int len)
{
    struct lnkhdr hdr;

    /* lock the mutex, blocking the access to the serial port... */
    thinkos_mutex_lock(rs485.mutex);
    /* prepare header */
    hdr.sync = PKT_SYNC;
    hdr.daddr = daddr;
    hdr.saddr = rs485.addr;
    hdr.datalen = len;
    /* send header */
    serial_send(rs485.serial, &hdr, sizeof(struct lnkhdr));
    /* send payload */
    serial_send(rs485.serial, data, len);
    /* unlock mutex, other thread can use the serial port now. */
    thinkos_mutex_unlock(rs485.mutex);
}

void netlnk_init(unsigned int addr)
{
    /* Open the serial port */
    rs485.serial = stm32f_uart1_serial_init(9600, SERIAL_8N1);
    /* Set the local address */
    rs485.addr = addr;
    /* Allocate a mutex */
    rs485.mutex = thinkos_mutex_alloc();
}
```

2.4.3 Time Wait

Use the `thinkos_sleep()` function to wait for a period of time. The calling thread suspends its execution until at least the time specified as parameter has elapsed. The time period is specified in milliseconds. The example that follows a LED will blink repeatedly 4 times per second:

Listing 4: Time Wait Example

```
int blink_task(void * arg)
```

```
{
    for (;;) {
        led_on();           /* Turn LED on */
        thinkos_sleep(100); /* Wait for 100 milliseconds */
        led_off();          /* Turn LED off */
        thinkos_sleep(150); /* Wait for 150 milliseconds */
    }

    return 0;
}
```

2.4.4 Compiling the Code

In order to compile the code in a Windows machine you must first install these tools:

- GNU Toolchain for ARM Processors for Windows hosts.
- GNU Make for Windows.

2.4.5 Compiling Using the Command Shell

You can use the windows Command Prompt or a MinGW/MSYS terminal to compile the code. Open a windows shell: Start-¿All Programs-¿Accessories-¿Command Prompt Change to the directory in the ThinkOS source tree. Type: make

3 ThinkOS Kernel

- Data structures.
  - Design decisions.
  ...

3.1 Exception Priorities

3.2 Kernel Objects

3.2.1 System Calls

3.2.2 Library functions

4 API

The ThinkOS API can be separated according to the type of object the operation is performed ...

Core initialization configuration and debug

Threads functions related to threads

Mutexes

Conditional Variables

Semaphores

Events

Interrupts

Time

4.1 Threads

int thinkos_thread_create (int(*task_ptr)(void *), void *task_arg, void *stack_ptr, unsigned int opt)
create a new thread
int thinkos_thread_create_inf (int(*task_ptr)(void *), void *task_arg, const struct thinkos_thread_inf *inf)
create a new thread with extend information
int const thinkos_thread_self (void)
obtaind a handler for the calling thread
int thinkos_cancel (unsigned int thread_id, int code)
request a thread to terminate
int thinkos_exit (int code)
cause the thread to terminate
int thinkos_join (unsigned int thread_id)
join with a terminated thread
int thinkos_pause (unsigned int thread_id)
pause the thread execution
int thinkos_resume (unsigned int thread_id)
resume a thread operation
void thinkos_yield (void)
causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

4.2 Time related calls

4.2.1 Functions

int thinkos_sleep (unsigned int ms)
pause for a specified amount of time
uint32_t thinkos_clock (void)
retrieve the ThikOS clock time
int thinkos_alarm (uint32_t clock)
pause until a specified alarm time

4.2.2 Detailed Description

4.2.3 Function Documentation

int thinkos_thread_create (int(*)(void *) task_ptr, void * task_arg, void * stack_ptr, unsigned int opt) #include <thinkos.h>

create a new thread

task_ptr
task_arg
stack_ptr
opt

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_thread_create_inf (int(*)(void *) task_ptr, void * task_arg, const struct thinkos_thread_inf * inf) #include <thinkos.h>

create a new thread with extend information

task_ptr
task_arg
inf

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int const thinkos_thread_self (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>

obtaind a handler for the calling thread

Returns: This function always succeeds, returning the calling thread's ID.

```
int thinkos\_cancel (unsigned int ~thread\_id, int ~code)

\#include thinkos.h
```

request a thread to terminate

thread_id	thread handler
code	return code

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_exit (int code) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
cause the thread to terminate

code	return code

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_join (unsigned int thread_id) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
join with a terminated thread
The thinkos_join() function waits for the thread specified by thread_id to terminate. If that thread has already terminated, then thinkos_join() returns immediately.

thread_id	thread handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_pause (unsigned int thread_id) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
pause the thread execution

thread_id	thread handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_resume (unsigned int thread_id) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
resume a thread operation

thread_id	thread handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

void thinkos_yield (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

4.3 Time related calls

4.3.1 Functions

int thinkos_sleep (unsigned int ms)
pause for a specified amount of time
uint32_t thinkos_clock (void)
retrieve the ThikOS clock time
int thinkos_alarm (uint32_t clock)
pause until a specified alarm time

4.3.2 Detailed Description

4.3.3 Function Documentation

int thinkos_sleep (unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
pause for a specified amount of time
The thinkos_sleep() function suspends execution of the calling thread for (at least) ms milliseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timer.

ms	waiting time in milliseconds

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

uint32_t thinkos_clock (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
retrieve the ThikOS clock time
The ThikOS clock is a tick counter with a resolution of 1 milisecond, which cannot be set and is allways running. It represents monotonic time since the system power up.

Returns: the ThinkOS monotonic clock.

int thinkos_alarm (uint32_t clock) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
pause until a specified alarm time
This function should be used in conjucntion with thinkos_clock(). . . .

clock	time for the thread to be waked up.

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

4.4 Mutexes

4.4.1 Functions

int thinkos_mutex_alloc (void)
alloc a mutex
int thinkos_mutex_free (int mutex)
release a mutex
int thinkos_mutex_lock (int mutex)
lock a mutex, blocking if already locked.
int thinkos_mutex_trylock (int mutex)
try to lock a mutex, non blocking.
int thinkos_mutex_timedlock (int mutex, unsigned int ms)
lock a mutex, blocking for limited time.
int thinkos_mutex_unlock (int mutex)
unlock the mutex

4.4.2 Detailed Description

4.4.3 Function Documentation

int thinkos_mutex_alloc (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
alloc a mutex

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_mutex_free (int mutex) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
release a mutex

mutex	mutual exclusion handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_mutex_lock (int mutex) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
lock a mutex, blocking if already locked.

mutex	mutual exclusion handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_mutex_trylock (int mutex) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
try to lock a mutex, non blocking.

mutex	mutual exclusion handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_mutex_timedlock (int mutex, unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
lock a mutex, blocking for limited time.

mutex	mutual exclusion handler
ms	waiting time in milliseconds

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_mutex_unlock (int mutex) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
unlock the mutex

mutex	mutual exclusion handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

4.5 Conditional Variables

4.5.1 Functions

int thinkos_cond_alloc (void)
alloc a conditional variable
int thinkos_cond_free (int cond)
release a conditional variable
int thinkos_cond_wait (int cond, int mutex)
wait for a conditional variable
int thinkos_cond_timedwait (int cond, int mutex, unsigned int ms)
wait for a conditional variable or timeout
int thinkos_cond_signal (int cond)
signal a conditional variable wake a single thread
int thinkos_cond_broadcast (int cond)
signal a conditional variable wake all waiting threads

4.5.2 Detailed Description

4.5.3 Function Documentation

int thinkos_cond_alloc (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
alloc a conditional variable

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_cond_free (int cond) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
release a conditional variable

cond	conditional variable handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_cond_wait (int cond, int mutex) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
wait for a conditional variable

cond	conditional variable handler
mutex	mutex handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_cond_timedwait (int cond, int mutex, unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
wait for a conditional variable or timeout

cond	conditional variable handler
mutex	mutex handler
ms	waiting time in milliseconds

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_cond_signal (int cond) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal a conditional variable wake a single thread

cond	conditional variable handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_cond_broadcast (int cond) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal a conditional variable wake all waiting threads

cond	conditional variable handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

4.6 Semaphores

4.6.1 Functions

int thinkos_sem_alloc (unsigned int val)
alloc a semaphore
int thinkos_sem_free (int sem)
release a semaphore
int thinkos_sem_init (int sem, unsigned int val)
set the initial value of a semaphore
int thinkos_sem_wait (int sem)
wait for a semaphore to be signaled
int thinkos_sem_timedwait (int sem, unsigned int ms)
wait for a semaphore to be signaled or timeout
int thinkos_sem_post (int sem)
signal a semaphore
void thinkos_sem_post_i (int sem)
signal a semaphore inside an interrupt handler

4.6.2 Detailed Description

4.6.3 Function Documentation

int thinkos_sem_alloc (unsigned int val) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
alloc a semaphore

val	initial semaphore value

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_sem_free (int sem) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
release a semaphore

sem	semaphore handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_sem_init (int sem, unsigned int val) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
set the initial value of a semaphore

sem	semaphore handler
val	

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_sem_wait (int sem) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
wait for a semaphore to be signaled

sem	semaphore handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_sem_timedwait (int sem, unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/thinl
wait for a semaphore to be signaled or timeout

sem	semaphore handler
ms	

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_sem_post (int sem) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal a semaphore

sem	semaphore handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

void thinkos_sem_post_i (int sem) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal a semaphore inside an interrupt handler

sem	semaphore handler

Parameters:

4.7 Event sets

4.7.1 Functions

int thinkos_ev_alloc (void)
alloc an event set
int thinkos_ev_free (int set)
release an event set
int thinkos_ev_wait (int set)
wait for an event
int thinkos_ev_timedwait (int set, unsigned int ms)
wait for an event or timeout
int thinkos_ev_raise (int set, int ev)
signal an event
void thinkos_ev_raise_i (int set, int ev)
signal an event from inside an interrupt handler
int thinkos_ev_mask (int set, int ev, int val)
mask/unmask an event
int thinkos_ev_clear (int set, int ev)
clear an event

4.7.2 Detailed Description

4.7.3 Function Documentation

int thinkos_ev_alloc (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
alloc an event set

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_ev_free (int set) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
release an event set

set	event set handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_ev_wait (int set) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
wait for an event

set	event set handler

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, an event otherwise.

int thinkos_ev_timedwait (int set, unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/thinkos
wait for an event or timeout

set	event set handler
ms	time to wait in milliseconds

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, an event otherwise.

int thinkos_ev_raise (int set, int ev) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal an event

set	event set handler
ev	event identifier

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

void thinkos_ev_raise_i (int set, int ev) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal an event from inside an interrupt handler

set	event set handler
ev	event identifier

Parameters:

int thinkos_ev_mask (int set, int ev, int val) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>

mask/unmask an event

set	event set handler
ev	event identifier
val	mask value. 1 enable the event, 0 disable the event.

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_ev_clear (int set, int ev) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
clear an event

set	event set handler
ev	event identifier

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

4.8 Flags

4.8.1 Functions

int thinkos_flag_alloc (void)
alloc a flag
int thinkos_flag_free (int flag)
release a flag
int thinkos_flag_val (int flag)
return the flag value
int thinkos_flag_set (int flag)
set a flag
int thinkos_flag_clr (int flag)
clear a flag
int thinkos_flag_give (int flag)
signal a flag
void thinkos_flag_give_i (int flag)
signal a flag form inside an interrupt handler
int thinkos_flag_take (int flag)
wait for flag to be signaled

int thinkos_flag_timedtake (int flag, unsigned int ms)
wait for flag or timeout

4.8.2 Detailed Description

4.8.3 Function Documentation

int thinkos_flag_alloc (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
alloc a flag

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_flag_free (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
release a flag

---

flag

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_flag_val (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
return the flag value

---

flag

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_flag_set (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
set a flag

---

flag

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_flag_clr (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>

clear a flag

---

flag

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_flag_give (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal a flag

---

flag

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

void thinkos_flag_give_i (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
signal a flag form inside an interrupt handler

---

flag

Parameters:

int thinkos_flag_take (int flag) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
wait for flag to be signaled

---

flag

Parameters:

int thinkos_flag_timedtake (int flag, unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/think
wait for flag or timeout

---

flag
ms

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

4.9 Gates

4.9.1 Functions

int thinkos_gate_alloc (void)
Alloc a gate synchronization object.
int thinkos_gate_free (int gate)
Frees the gate synchronization object.
int thinkos_gate_wait (int gate)
Wait for a gate to open.
int thinkos_gate_timedwait (int gate, unsigned int ms)
Wait for a gate to open or a timeout.
int thinkos_gate_open (int gate)
Open or signal the gate.
void thinkos_gate_open_i (int gate)
Open or signal the gate from inside an interrupt handler.
int thinkos_gate_close (int gate)
Close the gate if the gate is OPEN or remove pending signaling it the gate is @ LOCKED.
int thinkos_gate_exit (int gate, unsigned int open)
Exit the gate, leaving the gate, optionally leaving it open or closed.

4.9.2 Detailed Description

Gates are syncronization objects which provide a convenient way of creating mutual exclusion acess to code blocks signaled by interrupt handlers...

A gate have a lock flag and a signal flag. A gate can be in one of the following states:

CLOSED: no threads crossed the gate yet.

LOCKED: a thread entered the gate, closed and locked it.

OPENED: no threads are waiting in the gate, the first thread to call thinkos_gate_wait() will cross the gate.

SIGNALED: a thread crossed the gate and locked it, but the gate received a signal to open. When the thread exits the gate the gate will stay open.

4.9.3 Function Documentation

int thinkos_gate_alloc (void) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
Alloc a gate synchronization object.

Returns: return a handler for a new gate object, or a negative value if an error ocurred.
Errors:
THINKOS_ENOSYS if the system call is not enabled.
THINKOS_ENOMEM no gates left in the gate pool.

int thinkos_gate_free (int gate) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
Frees the gate synchronization object.

gate	handler for a gate object which must have been returned by a previous call to thinkos_gate_alloc() .

Parameters:

Returns: returns THINKOS_OK on sucess. On error a negative code value is returned. an error ocurred.
Errors:
THINKOS_EINVAL gate is not a valid gate handler.
THINKOS_ENOSYS not implemented.

int thinkos_gate_wait (int gate) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
Wait for a gate to open.
If the gate is open this function return imediatelly, otherwise it will block the calling thread.

gate	The gate descriptor.

Parameters:

Returns: THINKOS_EINVAL if gate is invalid, THINKOS_OK otherwise.

int thinkos_gate_timedwait (int gate, unsigned int ms) #include <C:/devel/yard-ice/src/sdk/include/thin
Wait for a gate to open or a timeout.
If the gate is open this function return imediatelly, otherwise it will block the calling thread.

gate	The gate descriptor.
ms	Timeout ins milliseconds.

Parameters:

Returns: THINKOS_OK is returned on sucess. On error a negative code value is returned.

THINKOS_EINVAL: gate is not a valid handler.
THINKOS_ETIMEDOUT: timer expired before the gate opens.
THINKOS_ENOSYS: syscall not implemented.

int thinkos_gate_open (int gate) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
Open or signal the gate.
The resulting gate's state will depend on the current gate state and whether there are threads waiting at the gate. There are four possible scenarios . . . :
1. the gate is open already, then this function does nothing.
2. the gate is closed and no threads are waiting it will open the gate, allowing the next thread to call gate_open() to enter the gate.
3. the gate is closed and at least one thread is waiting it will allow the thread to cross the gate, in this case the gate will be locked.
4. a thread crossed the gate (gate state is LOCKED), then the gate will be signaled to open when the gate is unlocked.

gate	The gate descriptor.

Parameters:

Returns: THINKOS_EINVAL if gate is invalid, THINKOS_OK otherwise.

void thinkos_gate_open_i (int gate) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
Open or signal the gate from inside an interrupt handler.
This call is similar to the thinkos_gate_open() except that it is safe to ba called from inside an interrupt handler.

gate	The gate descriptor.

Parameters: Warning: no argument validation is performed.

int thinkos_gate_close (int gate) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
Close the gate if the gate is OPEN or remove pending signaling it the gate is @ LOCKED.

gate	The gate descriptor.

Parameters:

Returns: THINKOS_EINVAL if gate is invalid, THINKOS_OK otherwise.

int thinkos_gate_exit (int gate, unsigned int open) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h
Exit the gate, leaving the gate, optionally leaving it open or closed.

gate	The gate descriptor.
open	Indicate the state of the gate on exit.

Parameters: open > 0, the gate will be left open, allowing for another thread to enter the gate.
open == 0, the gate will stay closed if not signaled, in wich case it will open accordingly.

Returns: THINKOS_EINVAL if gate is invalid, THINKOS_OK otherwise.

4.10 Interrupt Requests

4.10.1 Functions

int thinkos_irq_wait (int irq)
wait for interrupt
int thinkos_irq_register (int irq, int pri, void(*isr)(void))
reqister an interrupt handler

4.10.2 Detailed Description

4.10.3 Function Documentation

int thinkos_irq_wait (int irq) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
wait for interrupt

irq

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_irq_register (int irq, int pri, void(*)(void) isr) #include <C:/devel/yard-ice/src/sdk/include/thi
reqister an interrupt handler

irq

pri
isr

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

4.11 OS Monitor and Cotrol.

4.11.1 Functions

int thinkos_clocks (uint32_t *clk[])
get hardware clocks frequencies
int thinkos_udelay_factor (int32_t *factor)
get udelay calibration factor
void __attribute__ ((noreturn)) thinkos_abort(void)
abort the operating system.

4.11.2 Detailed Description

These calls return information about the ThinkOS configuratiion and operational parameters.

4.11.3 Function Documentation

int thinkos_clocks (uint32_t * clk[]) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
get hardware clocks frequencies

clk[]	pointer to an array to receive the clocks frequency list.

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

int thinkos_udelay_factor (int32_t * factor) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
get udelay calibration factor

factor	pointer to an integer.

Parameters:

Returns: THINKOS_ENOSYS if call is not implemented, THINKOS_OK otherwise.

void __attribute__((noreturn)) #include <C:/devel/yard-ice/src/sdk/include/thinkos.h>
abort the operating system.

Returns: This function does not return.

5 Data Structure Documentation

5.1 thinkos_thread_inf Struct Reference

#include <thinkos.h>

5.1.1 Data Fields

```
void * stack_ptr
    union {
     uint32_t opt
     struct {
       uint16_t stack_size
       uint8_t priority
       uint8_t thread_id: 7
       uint8_t paused: 1
     }
    };
    char tag [8]
```

5.1.2 Field Documentation

void* thinkos_thread_inf::stack_ptr

uint32_t thinkos_thread_inf::opt

uint16_t thinkos_thread_inf::stack_size

uint8_t thinkos_thread_inf::priority

uint8_t thinkos_thread_inf::thread_id

uint8_t thinkos_thread_inf::paused

union { **...** }

char thinkos_thread_inf::tag[8]

6 File Documentation

6.1 C:/devel/yard-ice/src/sdk/include/thinkos.h

ThinkOS API.

```
#include <stdint.h>
#include <thinkos_svc.h>
```

6.1.1 Data Structures

struct thinkos_thread_inf

6.1.2 Macros

```
#define IRQ_PRIORITY_HIGHEST (1 << 5)
#define IRQ_PRIORITY_VERY_HIGH (2 << 5)
#define IRQ_PRIORITY_HIGH (3 << 5)
#define IRQ_PRIORITY_REGULAR (4 << 5)
#define IRQ_PRIORITY_LOW (5 << 5)
#define IRQ_PRIORITY_VERY_LOW (6 << 5)
#define THINKOS_OPT_PRIORITY(VAL) (((VAL) & 0xff) << 16)
#define THINKOS_OPT_ID(VAL) (((VAL) & 0x07f) << 24)
#define THINKOS_OPT_PAUSED (1 << 31) /* don't run at startup */
#define THINKOS_OPT_STACK_SIZE(VAL) ((VAL) & 0xffff)
```

6.1.3 Enumerations

```
enum thinkos_err{
    THINKOS_OK= 0,
    THINKOS_ETIMEDOUT= -1,
    THINKOS_EINTR= -2,
    THINKOS_EINVAL= -3,
    THINKOS_EAGAIN= -4,
    THINKOS_EDEADLK= -5,
    THINKOS_EPERM= -6,
    THINKOS_ENOSYS= -7,
    THINKOS_EFAULT= -8,
    THINKOS_ENOMEM= -9
    }
    enum thinkos_obj_kind {
    THINKOS_OBJ_READY = 0,
    THINKOS_OBJ_TMSHARE,
    THINKOS_OBJ_CLOCK,
    THINKOS_OBJ_MUTEX,
    THINKOS_OBJ_COND,
    THINKOS_OBJ_SEMAPHORE,
    THINKOS_OBJ_EVENT,
    THINKOS_OBJ_FLAG,
```

THINKOS_OBJ_GATE,
THINKOS_OBJ_JOIN,
THINKOS_OBJ_CONREAD,
THINKOS_OBJ_CONWRITE,
THINKOS_OBJ_PAUSED,
THINKOS_OBJ_CANCELED,
THINKOS_OBJ_FAULT,
THINKOS_OBJ_INVALID
}

6.1.4 Functions

int thinkos_init (unsigned int opt)
Initializes the ThinkOS library.
int thinkos_thread_create(int(*task_ptr)(void *), void *task_arg, void *stack_ptr, unsigned int opt)
create a new thread
int thinkos_thread_create_inf(int(*task_ptr)(void *), void *task_arg, const struct thinkos_thread_inf *inf)
create a new thread with extend information
int const thinkos_thread_self(void)
obtaind a handler for the calling thread
int thinkos_cancel (unsigned int thread_id, int code)
request a thread to terminate
int thinkos_exit (int code)
cause the thread to terminate
int thinkos_join (unsigned int thread_id)
join with a terminated thread
int thinkos_pause (unsigned int thread_id)
pause the thread execution
int thinkos_resume (unsigned int thread_id)
resume a thread operation
void thinkos_yield (void)
causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.
int thinkos_sleep (unsigned int ms)
pause for a specified amount of time
uint32_t thinkos_clock (void)
retrieve the ThikOS clock time
int thinkos_alarm (uint32_t clock)
pause until a specified alarm time
int thinkos_mutex_alloc (void)
alloc a mutex
int thinkos_mutex_free (int mutex)
release a mutex
int thinkos_mutex_lock (int mutex)
lock a mutex, blocking if already locked.
int thinkos_mutex_trylock (int mutex)

try to lock a mutex, non blocking.
int thinkos_mutex_timedlock (int mutex, unsigned int ms)
lock a mutex, blocking for limited time.
int thinkos_mutex_unlock (int mutex)
unlock the mutex
int thinkos_cond_alloc (void)
alloc a conditional variable
int thinkos_cond_free (int cond)
release a conditional variable
int thinkos_cond_wait (int cond, int mutex)
wait for a conditional variable
int thinkos_cond_timedwait (int cond, int mutex, unsigned int ms)
wait for a conditional variable or timeout
int thinkos_cond_signal (int cond)
signal a conditional variable wake a single thread
int thinkos_cond_broadcast (int cond)
signal a conditional variable wake all waiting threads
int thinkos_sem_alloc (unsigned int val)
alloc a semaphore
int thinkos_sem_free (int sem)
release a semaphore
int thinkos_sem_init (int sem, unsigned int val)
set the initial value of a semaphore
int thinkos_sem_wait (int sem)
wait for a semaphore to be signaled
int thinkos_sem_timedwait (int sem, unsigned int ms)
wait for a semaphore to be signaled or timeout
int thinkos_sem_post (int sem)
signal a semaphore
void thinkos_sem_post_i (int sem)
signal a semaphore inside an interrupt handler
int thinkos_ev_alloc (void)
alloc an event set
int thinkos_ev_free (int set)
release an event set
int thinkos_ev_wait (int set)
wait for an event
int thinkos_ev_timedwait (int set, unsigned int ms)
wait for an event or timeout
int thinkos_ev_raise (int set, int ev)
signal an event
void thinkos_ev_raise_i (int set, int ev)
signal an event from inside an interrupt handler
int thinkos_ev_mask (int set, int ev, int val)
mask/unmask an event
int thinkos_ev_clear (int set, int ev)
clear an event
int thinkos_flag_alloc (void)

alloc a flag
int thinkos_flag_free (int flag)
release a flag
int thinkos_flag_val (int flag)
return the flag value
int thinkos_flag_set (int flag)
set a flag
int thinkos_flag_clr (int flag)
clear a flag
int thinkos_flag_give (int flag)
signal a flag
void thinkos_flag_give_i (int flag)
signal a flag form inside an interrupt handler
int thinkos_flag_take (int flag)
wait for flag to be signaled
int thinkos_flag_timedtake (int flag, unsigned int ms)
wait for flag or timeout
int thinkos_gate_alloc (void)
Alloc a gate synchronization object.
int thinkos_gate_free (int gate)
Frees the gate synchronization object.
int thinkos_gate_wait (int gate)
Wait for a gate to open.
int thinkos_gate_timedwait (int gate, unsigned int ms)
Wait for a gate to open or a timeout.
int thinkos_gate_open (int gate)
Open or signal the gate.
void thinkos_gate_open_i (int gate)
Open or signal the gate from inside an interrupt handler.
int thinkos_gate_close (int gate)
Close the gate if the gate is OPEN or remove pending signaling it the gate is @ LOCKED.
int thinkos_gate_exit (int gate, unsigned int open)
Exit the gate, leaving the gate, optionally leaving it open or closed.
int thinkos_irq_wait (int irq)
wait for interrupt
int thinkos_irq_register (int irq, int pri, void(*isr)(void))
reqister an interrupt handler
int thinkos_clocks (uint32_t *clk[])
get hardware clocks frequencies
int thinkos_udelay_factor (int32_t *factor)
get udelay calibration factor
void __attribute__ ((noreturn)) thinkos_abort(void)
abort the operating system.

Author: Robinson Mittmann bobmittmann@gmail.com

6.1.5 Macro Definition Documentation

#define IRQ_PRIORITY_HIGHEST (1 << 5)

#define IRQ_PRIORITY_VERY_HIGH (2 << 5)

#define IRQ_PRIORITY_HIGH (3 << 5)

#define IRQ_PRIORITY_REGULAR (4 << 5)

#define IRQ_PRIORITY_LOW (5 << 5)

#define IRQ_PRIORITY_VERY_LOW (6 << 5)

#define THINKOS_OPT_PRIORITY(VAL) (((VAL) & 0xff) << 16)

#define THINKOS_OPT_ID(VAL) (((VAL) & 0x07f) << 24)

#define THINKOS_OPT_PAUSED (1 << 31) /* don't run at startup */

#define THINKOS_OPT_STACK_SIZE(VAL) ((VAL) & 0xffff)

6.1.6 Enumeration Type Documentation

enum thinkos_err Enumerator
- THINKOS_OK No error
- THINKOS_ETIMEDOUT System call timed out
- THINKOS_EINTR System call interrupted out
- THINKOS_EINVAL Invalid argument
- THINKOS_EAGAIN Non blocking call failed
- THINKOS_EDEADLK Deadlock condition detected
- THINKOS_EPERM
- THINKOS_ENOSYS Invalid system call
- THINKOS_EFAULT
- THINKOS_ENOMEM Resource pool exausted

enum thinkos_obj_kind Enumerator
- THINKOS_OBJ_READY
- THINKOS_OBJ_TMSHARE
- THINKOS_OBJ_CLOCK
- THINKOS_OBJ_MUTEX
- THINKOS_OBJ_COND
- THINKOS_OBJ_SEMAPHORE
- THINKOS_OBJ_EVENT

THINKOS_OBJ_FLAG
THINKOS_OBJ_GATE
THINKOS_OBJ_JOIN
THINKOS_OBJ_CONREAD
THINKOS_OBJ_CONWRITE
THINKOS_OBJ_PAUSED
THINKOS_OBJ_CANCELED
THINKOS_OBJ_FAULT
THINKOS_OBJ_INVALID

6.1.7 Function Documentation

int thinkos_init (unsigned int opt) Initializes the ThinkOS library.
On return the current program execution thread turns into the first thread of the system.

Returns: THINKOS_OK

7 Module Index

7.1 Modules

Here is a list of all modules
Threads
Time related calls
Mutexes
Conditional Variables
Semaphores
Event sets
Flags
Gates
Interrupt Requests
OS Monitor and Control.

8 Data Structure Index

8.1 Data Structures

Here are the data structures with brief descriptions:
thinkos_thread_inf

9 Modules Summary

10 Demo Using Windows

11 Tools

12 Tools

12.1 MSYS-2

MSYS2 is software distribution and a building platform for Windows. It provides a Unix-like environment, a command-line interface and a software repository making it easier to install, use, build and port software on Windows.

The official website is:
MSYS2 Website

Useful information can be found at the project's wiki pages at:
MSYS2 Introduction

12.2 GNU Toolchain for ARM Processors

ThinnkOS on Windows Host - this document is a guide on how to set-up the tools and environment to develop ThinkOS applications using a Windows computer as host.....

The GNU toolchain is a broad collection of programming tools developed by the GNU Project. These tools form a toolchain (a suite of tools used in a serial manner) used for developing software applications and operating systems.

If you are a windows user, you can download and install the GNU Toolchain for ARM Processors from the site: https://launchpad.net/gcc-arm-embedded. At the time of writing this document the latest version known to work with YARD-ICE was 4.9-2015-q3, which can be downloaded directly from:

GNU Arm Embedded Toolchain at Launchpad: [https://launchpad.net/gcc-arm-embedded]

[GNU Arm Embedded Toolchain]
Downloads: [https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

Run the downloaded file *gcc-arm-none-eabi-6-2017-q2-update-win32-sha2.exe*, or newer, to start the installation. In the next steps use the default values suggested by the wizard installer. In the last screen, after the files were copied, select the "Add path to environment variable" option.

Alternatively a compressed (.zip) file can be downloaded (*gcc-arm-none-eabi-6-2017-q2-update-win32.zip*) and extracted in a directory of choice...

Embedded systems ... In some circumstances there are some pieces of code can be ... better performance comprises of a repetition of code that is tedious to type and error prone. Also in some contexts a generic solution that can be computed at runtime can be too expensive in terms of CPU usage to be computed in real-time. Or may require too much memory. A common solution for these problems is to use lookup tables and to interpolate intermidiate values, store coefficients in FLASH. Even if you could afford to calculate a complex dataset some systems require a very fast boot-up response, so storing pre computed values is a sure way of impproving things. We present here two typical scenarios where an automated generated code is benneficial for embedded systems.

14.2.1 Fixed-point sine function

Example sine function using Python.

14.2.2 IIR filter

In this example we use a Julia script to calculate and automatically generate a very efficient C code for a low pass filter. By automating the code generation its easy to change the filter parameters, like cur off frequency, filter order without having to worry about synchronizing the design specs with your embedded code.

14.2.3 Embedded HTML pages

Here HTML pages, or any pages for that matter, are automatically compressed, converted into C arrays. Makefile rules take care of the dependency of your code and the web pages, so the code is regenerated and compiled whenever a page is modified.

14.2.4 Autogenerated Code Comments

Always add a comment in the code indicating that it is automatically generated and a summary of instructions on how to regenerate it.

Example:

14.3 Makefiles

14.4 Text Editor

14.4.1 Multiple editor windows

14.5 Git - Revision Control

15 References

15.1 Gnu Tools for Window

More information on the GNU Make and CoreUtils for Windows package can be found at:

- http://gnuwin32.sourceforge.net/packages/make.htm
- http://gnuwin32.sourceforge.net/packages/coreutils.htm

15.2 GNU ARM Eclipse Plug-Ins

Quick info on GNU ARM Eclipse Plug-Ins:

- Name: '*GNU ARM Eclipse Plug-ins*'
- Location: http://gnuarmeclipse.sourceforge.net/updates
- Website: http://gnuarmeclipse.livius.net/blog

15.3 GNU MCU Eclipse plug-ins

These plug-ins provide Eclipse CDT (C/C++ Development Tooling) extensions for GNU embedded toolchains like the GNU Tools for ARM Embedded Processors used as reference in this document.

- Eclipse update site: http://gnu-mcu-eclipse.netlify.com/v4-neon-updates
- Website: https://gnu-mcu-eclipse.github.io

15.4 Git Extensions

From the project's website:

> Git Extensions is a toolkit aimed at making working with Git under Windows more intuitive (note that Git Extensions is also available on Linux and Macintosh OS X using Mono). The shell extension will integrate in Windows Explorer and presents a context menu on files and directories.

Website: Git Extensions

15.5 YARD-ICE

The YARD-ICE project is a community based, open source hardware and software platform for a standalone remote debugger. The embedded platform uses **ThinkOS** as bootloader and real-time operating system.

- YARD-ICE Project Location: https://github.com/bobmittmann/yard-ice
- Latest Release: https://github.com/bobmittmann/yard-ice/archive/0.24.zip
- Git Repository (HTTPS): https://github.com/bobmittmann/yard-ice.git
- Git Repository (SSH):git@github.com:bobmittmann/yard-ice.git

15.6 Useful Web References

Real-time operating system - Wikipedia - March, 2017

The GNU Project

GCC, the GNU Compiler Collection

MSYS2

Git

Linux kernel coding style