SIR CMPWN TECHNOLOGIES

KnightOS for Developers

Coding multithreaded programs for KOS

Drew DeVault

10/21/2010

This document is a beta release and is subject to and likely to change.

Contents

Introduction

Thank you for choosing KnightOS for your development needs. KnightOS is a multi-threaded operating system, which puts considerably more requirements on the developer. However, there are several routines built into KOS to make this easier on you as the developer.

There are two different ways of executing code under KnightOS – libraries and programs. Libraries are utilities used by programs to perform certain tasks, similar to a shell under TIOS. KnightOS provides several libraries for your use, stored at /lib/. You can also create your own libraries. KOS supports up to 20 libraries loaded simultaneously. Libraries run from RAM, and SMC is allowed, although not recommended unless it is safe to execute the routines more than once.

The other method of programming under KOS is with programs. Programs also run from RAM, and SMC is completely safe, unlike potential complications with libraries, as multiple instances of a program will are allocated memory twice. Programs have their own set of registers, including shadow and index registers, all of which are safe to use without disabling interrupts. In fact, disabling interrupts is frowned upon, because the user will be unable to switch programs and other tasks will not be given CPU time.

Memory Layout

The following diagram explains how memory is laid out in KnightOS:

// TODO

Programs

Programs are run from RAM. However, due to the multitasking nature of KnightOS, programs do not know where in RAM they will execute until runtime. This makes certain tasks harder. However, KnightOS provides several helper routines to ease this process. First, let's talk about the header. [Subject to change] It includes a stack size. All programs have a stack unique to them. The first byte of the header describes the size of the stack, in bytes. This is a typical program header:

Code **z80**

```
.db 10 ; 10 bytes for stack
Program:
.org 0
    ; Your code here
```

In order to accommodate for position-independent code, the opcodes jp, call, and occasionally ld are unavailable. KnightOS provides several routines to handle this, and KnightOS.inc has the following macros to use these opcodes:

Macro	Function
kjp Address	Jumps to Address, plus the offset of the program in RAM
kjpc Condition, Address	If Condition is true, jumps to Address, plus the offset of the program in RAM
kcall Address	Pushes PC to the stack and jumps to Address, plus the offset of the program in RAM
kcall Condition, Address	If Condition is true, pushes PC to the stack and jumps to Address, plus the offset of the program in RAM
kld Register, Address	Loads the value of Address into Register
kldp Register, Address	Loads the value at Address into Register

The kernel uses self-modifying code to modify these routines at runtime so that they only have to run once. It will run slower the first time one of these is executed, but the subsequent times will run quickly.

You can find an example program in /samples/program.asm.

Libraries

Libraries run from RAM as well as programs. Libraries provide functionality that is common among several programs, similar to a shell. The OS should provide several libraries for programs to access GUI and other common routines. Like programs, they must be position-independent, and like programs, there are routines to help this work. Each library has an ID word associated with it. This word is used to reference the library when programs make calls to it, as well as within libraries themselves to call internal routines. It should be a number unique to your library. Be sure to register your library's number at http://knightos.sourceforge.net/. This is an example header for a library:

Code **z80**

```
.dw 0 ; Library ID
Library:
 .org 0
    ; Your code here
```

As libraries are required to be location-independent, the following macros are provided in KnightOS.inc to help you achieve this. These are the same macros you can use inside a program to call library functions:

Macro	Function
ljp ID, Address	Jumps to Address plus the offset of the library with the matching ID
ljpc ID, Condition, Address	If Condition is true, jumps to Address plus the offset of the library with the matching ID
lcall ID, Address	Pushes PC to the stack and jumps to Address plus the offset of the library with the matching ID
lcallc ID, Condition, Address	If Condition is true, pushes PC to the stack and jumps to Address plus the offset of the library with the matching ID
lld ID, Register, Address	Loads Address plus the offset of the library with the matching ID into Register
lldp ID, Register, Address	Loads the value at Address plus the offset of the library with the matching ID into Register

Like programs, the kernel uses SMC to make each subsequent run of the code faster after the first time.

An example library can be found in /samples/lib.asm

Clipboard

KnightOS features a global clipboard shared among all programs. It consists of three bytes of Safe RAM that describe what is in the clipboard and where to find it. The first byte is the ID byte, which represents what kind of data is currently copied. The next two bytes are the address of the data in RAM. It is generally good practice to copy the data elsewhere, rather than to point to where it is at the moment, so that the user can modify the copied data without modifying the clipboard. You may register your type ID on http://knightos.sourceforge.net/, so that two programs do not have conflicting data types. Here are the data types used by KnightOS that your program may take advantage of:

ID	Description
0x00	Plain Text
0x01	Image (See Images for more information)

Images

KnightOS uses a special format for storing images, in order to create a standard for programs to use when exchanging data. This is the same format that should appear on the clipboard for type 0x01 data. The pointer word in the clipboard should point to a valid image, which is structured as follows:

Offset	Size	Data
0x0000:	2 bytes	Width
0x0002:	2 bytes	Height
0x0004:	1 byte	Levels of grayscale
0x0005:	X bytes	Data

The data should have each buffer in order.

Kernel Routines

These routines are provided by the kernel, and exist on ROM page 00. Page 00 is always swapped in, so they may safely be run with CALL or JP. When specified, they may be run with RST.

Graphics			
FastCopy	FastCopySafe		
Multitasking			
KillCurrentThread	KillThread		

KillThread

Multitasking

Kills a specific thread and frees all resources associated with it. You shouldn’t use this to kill the currently executing thread, instead use KillCurrentThread.

Input:

A: The thread to destroy

Output:

None

Destroys:

All resources associated with specified thread.

Example Usage

```
ld a, 1
call KillThread ; Kills the thread with an ID of 1
```

KillCurrentThread

Multitasking

Kills the current thread and frees all resources associated with it. Use KillThread to kill a thread other than your own. It will not return to your program after execution, so use JP instead of CALL to save a byte.

Thread safety: **Thread Safe**

Input:

None

Output:

Current thread stops execution

Destroys:

All registers and stack

Example Usage
jp KillCurrentThread ; End this thread

FastCopy

Graphics

Copies the contents of LCDBuffer to the screen. May be run with RST rFastCopy. Safe for all models of calculators, even with newer LCDs.

Thread safety: **Thread Unsafe**. If you do not have exclusive control of the processor, use FastCopySafe.

Inputs:

None

Outputs:

LCDBuffer is copied to the screen

Destroys:

None

Code: Example Usage

```
ld hl, Picture
ld de, LCDBuffer
ld bc, 768
ldir ; Copy an image to the buffer
rst rFastCopy ; And copy the buffer to the screen
```

FastCopySafe

Graphics

Copies LCDBuffer to the screen only if your thread has access to the screen.

Inputs:

None

Outputs:

LCDBuffer is copied to the screen.

Destroyed:

None

Code: Example Usage

```
ld hl, Picture
ld de, LCDBuffer
ld bc, 768
ldir ; Copy an image to the buffer
call FastCopySafe ; And copy the buffer to the screen
```