CatOS

Author: Evan Higgins

Instructor: Christian Hassard

Revision 0.6

ECE270, Fall, 2021

DigiPen Institute of Technology

Abstract

CatOS is a Real-Time Operating System designed for use in embedded systems. This document details all the necessary information and cautions that users should know when using CatOS. This includes intended/supported devices, implementation details, functions and macros, as well as necessary considerations and warnings to the user.

A special thanks to Christian Hassard for teaching me about Real-Time Operating Systems and guiding me through the development of my own. Thanks as well to all other students in the DigiPen Fall 2021 ECE270 class!

Table of Contents

1 Introduction

1.1 About This Document

This document provide an overview and guidelines for usage of CatOS. This includes details about its capabilities, intended architecture(s), and the philosophies behind its design.

1.1.1 Intended Audience

This document is intended for those intending to use and develop tasks for CatOS.

1.1.2 Document Version History

Document Version	CatOS Version	Changes
v1.0	v0.6	Initial Release

1.2 RTOS Model

The **Real-Time Operating System** model involves absolute deadlines of tasks differentiating it from a **General Purpose Operating System** model. This strict constraint on the Requirementstiming correctness of the system requires many characteristics to be present within the system.

1.3 Philosophy

The characteristics that make up the **real-time** philosophy applied to RTOSs (including CatOS) involve reliability, predictability, optimized performance, compactness, and scalability. These characteristics create an operating system that is application-specific, highly optimized, and robust.

The importance of timing correctness also means that scheduling policies/algorithms for these systems are tailored for **real-time** applications. These algorithms aim to allow all given tasks to achieve their deadlines.

1.4 Hardware Abstraction

The scalability of RTOSs means that they must be usable on a wide variety of systems. To achieve this a **Hardware Abstraction Layer (HAL)** is

implemented (sometimes called a **Board Support Package (BSP)**. This abstraction layer directly interfaces with the hardware while maintaining the same API and usage for the user.

For each device, a different HAL/BSP is necessary to account for the differences between devices. However, the usage remains the same for the user. (See: Available Devices).

2 Design Specifications

2.1 Functional Requirements

Functional Correctness of the system is very important for any application. The inclusion of a strict timing requirement does not change that. The system is required to run without major error and will report if one occurs so that a graceful failure or recovery may be attempted.

The functional correctness of the tasks given is not affected by the operating system. CatOS does not interfere with the functionality of the given tasks. It only manages the resources on the system and ensures that the tasks achieve their defined deadlines.

To ensure this functional correctness of its tasks, the kernel does not interact with the internal code of the task and gives control of the CPU to the task for its allotted time frame.

As for functional requirements, the user must give information to the kernel regarding the deadline and expected execution time of the tasks to be executed. To ensure functional requirements of the kernel objects it is advised that all precautions and guidelines for the kernel objects outlined in this document be followed.

2.2 Timing Requirements

Timing correctness is extremely important in a real-time system. CatOS ensures all tasks reach their deadlines if the set of tasks can. This is achieved through the sec:Scheduling Algorithmearliest deadline first scheduling algorithm.

Some amount of variability is expected due to limitations in various areas of the system. Due to potential variance and resource limitations it is advised to only schedule sets of tasks that meet the following expected CPU utilization specifications:

$$U_b = n(2^{1/n} - 1)$$

$$U = \sum_i C_i/P_i$$

$$U \leq U_b < 1$$

where: $P_i = periodofTask_i$, C = Execution time of $Task_i$, U = Utilization, U_b = Utilization Bound. (Any $U > 1$ will be unable to meet all deadlines)

3 Platform Specifications

3.1 Available Devices

CatOS' intended device is the STM32F4-Discovery board. the STM32 features a 32-bit Arm® Cortex®-M4 processor with FPU Core. 1Mb Flash memory and 192Kb RAM in an LQFP100 package. CatOS is highly compatible with other Cortex®-M devices and will likely see more Cortex®-M HAL/BSP inclusions.

3.2 Software Dependencies

CatOS is written in embedded C as well as ARM assembly. An ARM processor is expected on any device that runs CatOS. No extra libraries outside of the C standard library are used.

3.2.1 Assembly Instructions

The Cortex-M4 processor implements the ARMv7-M Thumb instruction set. While most embedded C code used in the project can be recompiled without modification to use a different processor, that processor must support the following instructions:

Operation	Description	Assembler	Cycles
Move	Register	MOV Rd, ⟨*op2*⟩	1
	16-bit immediate	MOV Rd, ⟨*op2*⟩	1
	Immediate into top	MOV Rd, ⟨*op2*⟩	1
	To PC	MOV Rd, ⟨*op2*⟩	1 + P
Push	Push	PUSH {⟨*reglist*⟩}	1 + N
	Push with link register	PUSH {⟨*reglist*⟩, LR}	1 + N
Pop	Pop	PUSH {⟨*reglist*⟩}	1 + N
	Pop and return	POP {⟨*reglist*⟩, LR}	1 + N + P
Add	Add	ADD Rd, Rn, ⟨*op2*⟩	1
	Add to PC	ADD PC, PC, Rm	1 + P
	Add with carry	ADC Rd, Rn, ⟨*op2*⟩	1
	Form address	ADR Rd, ⟨*label*⟩	1

Load	Word	LDR Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	To PC	LDR PC, [Rn, ⟨*op2*⟩]	$2^{[b]} + P$
	Halfword	LDRH Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Byte	LDRB Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Signed halfword	LDRSH Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Signed byte	LDRSB Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	User word	LDRT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User halfword	LDRHT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User byte	LDRBT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User signed halfword	LDRSHT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User signed byte	LDRSBT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	PC relative	LDR Rd, [PC, #⟨*imm*⟩]	$2^{[b]}$
	Doubleword	LDRD Rd, Rd, [Rn, #⟨*imm*⟩]	1 + N
	Multiple	LDM Rn, {⟨*reglist*⟩}	1 + N
	Multiple including PC	LDM Rn, {⟨*reglist*⟩,PC}	1 + N + P
Store	Word	STR Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Halfword	STRH Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Byte	STRB Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Signed halfword	STRSH Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	Signed byte	STRSB Rd, [Rn, ⟨*op2*⟩]	$2^{[b]}$
	User word	STRT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User halfword	STRHT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User byte	STRBT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User signed halfword	STRSHT Rd, [Rn, #⟨*imm*⟩]	$2^{[b]}$
	User signed byte	STRSBT Rd, [Rn, #⟨*imm*⟩]	2^{b}
	Doubleword	STRD Rd, [Rn, #⟨*imm*⟩]	1 + N
	Multiple	STM Rn, {⟨*reglist*⟩}	1 + N
Branch	Conditional	B⟨*cc*⟩ ⟨*label*⟩	1 or $1 + P^{C}$
	Unconditional	B ⟨*label*⟩	1 + P
	With link	BL ⟨*label*⟩	1 + P
	With exchange	BX Rm	1 + P
	With link and exchange	BLX Rm	1 + P
	Branch if zero	CBZ Rn, ⟨*label*⟩	1 or 1 + P
	Branch if non-zero	CBNZ Rn, ⟨*label*⟩	1 or 1 + P
	Byte table branch	TBB [Rn, Rm]	2 + P
	Halfword table branch	TBH [Rn, Rm, LSL#1]	2 + P

State Change	Supervisor call	SVC #⟨*imm*⟩	-
	If-then-else	IT... ⟨*cond*⟩	1[⌊*d*]
	Disable interrupts	CPSID ⟨*flags*⟩	1 or 2
	Enable interrupts	CPSIE ⟨*flags*⟩	1 or 2
	Read special register	MRS Rd, ⟨*op*2⟩	1 or 2
	Write special register	MSR ⟨*specreg*⟩, Rn	1 or 2
	Breakpoint	BKPT #⟨*imm*⟩	-
Logical	AND	AND Rd, Rn, ⟨*op*2⟩	1
	Exclusive OR	EOR Rd, Rn, ⟨*op*2⟩	1
	OR	ORR Rd, Rn, ⟨*op*2⟩	1
	OR NOT	ORN Rd, Rn, ⟨*op*2⟩	1
	Bit clear	BIC Rd, Rn, ⟨*op*2⟩	1
	Move NOT	MVN Rd, ⟨*op*2⟩	1
	AND test	TST Rd, ⟨*op*2⟩	1
	Exclusive OR test	TEQ Rd, ⟨*op*1⟩	1
Shift	Logical shift left	LSL Rd, Rn, #⟨*imm*⟩	1
	Logical shift left	LSL Rd, Rn, Rs	1
	Logical shift right	LSR Rd, Rn, #⟨*imm*⟩	1
	Logical shift right	LSR Rd, Rn, Rs	1
	Arithmetic shift right	ASR Rd, Rn, #⟨*imm*⟩	1
	Arithmetic shift right	ASR Rd, Rn, Rs,	1
Compare	Compare	CMP Rn, ⟨*op*2⟩	1
	Negative	CMN Rn, ⟨*op*2⟩	1

For a full list of Cortex®-M4 instructions see online ARM documentation

3.3 Hardware Dependencies

The intended hardware for this device is the Cortex®-M4 processor found on the TM32F4-Discovery board. The kernel for this operating system only requires the processor from the board. More requirements such as memory and storage requirements will be found here in a later iteration of this document along with the performance statistics.

3.4 Processor Context

The Cortex®-M4 has the following 32-bit registers

- 13 general-purpose registers, r0-r12
- Stack Pointer (SP) alias of banked registers, SP_process and SP_main
- Link Register (LR), r14
- Program Counter (PC), r15
- Special-purpose Program Status Registers,(xPSR)

For more information regarding the Cortex®-M4 processor registers see online ARM documentation

4 Implementation Details

4.1 Theory of Operation

CatOS can manage several tasks at a time each depending on one or more resources also managed by CatOS. The scheduling algorithm ensures that if all tasks can achieve their defined deadlines then they will all execute within their required deadlines (Assuming no catastrophic failure or significant externally caused delay)(see Scheduling Algorithm).

These deadlines are defined within each task's Task Control Block (TCB). Along with these deadlines, the TCB stores information such as the task's expected execution time, priority level, current task state, and each TCB also stores the task's stack pointer.

These stack pointers refer the task to its own stack space where its necessary variables for operation are stored. The size of this stack is configurable at startup (see Configuration).

4.2 Interrupt Timings

CatOS' kernel uses a timer interrupt-based preemption system where each task is given an uninterrupted 1ms of time (assuming 16MHz clock speed) before the CPU preempts the task to execute the scheduling algorithm as well as other necessary kernel functions. Task deadlines and execution times are assigned as milliseconds.

4.3 Scheduling Algorithm

The scheduling algorithm implemented by CatOS is an **Earliest Deadline First** algorithm. This algorithm first prioritizes tasks whose deadline is the soonest and then prioritizes based on the assigned task priorities.

This algorithm will successfully schedule any set of tasks such that all deadlines are met if such as schedule is possible. This algorithm does not take external factors or potential delays from resource limitation into account so it is suggested to follow the guidelines found in Timing Requirements for any tasks scheduled.

5 Usage

5.1 Use Cases

CatOS is intended to be used in an embedded system where multiple tasks will be present and multiple resources shared between the tasks. CatOS will manage the tasks given and resources necessary for the tasks to ensure that a **Real-Time** system is upheld.

5.2 Configuration

To configure the operating system the user simply provides the necessary parameters to the initialization function: *OS_InitKernel(numTasks, stackSize)*. By default, the heap size is set to 512 * sizeof(unsigned int). The numTasks argument given determines the max number of tasks that can be created for the system. The stackSize argument given determines the stack size for each task.

5.3 How to Initialize

After configuring the kernel with the necessary parameters the user can create the tasks using the *OS_CreateTask(address,priority,executionTime,Deadline)* function (For more information on tasks see [sec:Working With Tasks]Working With Tasks). Once all configuration and tasks are completed simply call $OS_Start()to start CatOS$.

5.4 Working With Tasks

Tasks are the user's code that CatOS works to schedule and manage. Internally they are defined by a ***Task Control Block*** *(TCB) that stores information regarding a task's status, priority, deadlines, execution time, and stack information. To create a task thee user simply calls OS_CreateTask(address, priority, executionTime, Deadline).*

The tasks are automatically scheduled, by the kernel, based on their deadlines and priorities on creation (see [sec:Scheduling Algorithm]Scheduling Algorithm. These tasks will run on the CPU in increments of 1ms until their execution time is reached. Once its execution time is reached the task will be

held until its next executable time defined by its deadline. Once its deadline is reached it will be re-scheduled.

This process will repeat as long as the kernel is running. All tasks will run asynchronously unless blocked by a required resource. These resources are defined by semaphores or mutexes (For more information on synchronization of tasks see [sec:Synchronization]sec:Synchronization

During the execution of the tasks, a task-specific stack is provided for local variables. The size of these stacks can be given during the configuration (see [sec:Configuration]Configuration). Separation of each task's stack ensures no contamination between stacks. For dynamically allocated memory there is a heap for users to utilize as well (see [sec:Memory]Memory).

5.5 Synchronization

Synchronization of tasks is done using the kernel's synchronization structures: ***Semaphores*** *and* ***Mutexes****. Mutexes are treated as recursive mutexes with ownership when acquired by a task.*

In order to create a mutex/semaphore the user calls OS_SemCreate(type, tokenStart, tokenMax). This function will create the provided type and return the indexed ID for the semaphore that should be provided whenever acquiring or releasing the structure. The types of structures are defined by a provided enumerator. These types are ***MUTEX*** *and* ***COUNTING****.* ***MUTEX*** *creates a recursive mutex.* ***COUNTING*** *creates a counting semaphore with the provided token counts. (note: recursive mutexes are uncapped and the token count information does not affect the functionality of mutexes when used as a recursive mutex)*

In order to acquire a mutex/semaphore task must call OS_SemAcquire(SemID) where SemID is the ID given when the mutex/semaphore was originally created. If the mutex/semaphore is available, the task will continue running. If the mutex/semaphore is not available, the task will be placed in a blocking state until the semaphore is available. note: multiple tasks can be waiting for the same resource. In such a situation the tasks waiting are treated as a queue where the first task that was placed into a blocked state will be given access first)

In order to release a mutex/semaphore task must call OS_SemRelease(SemID) where SemID is the ID given when the mutex/semaphore was originally created. If the structure is a mutex then the owner of the mutex will be compared to the task attempting to release it to ensure mutual exclusion. If the structure

is a semaphore then the tokenMax will be compared to the current number of tokens to ensure it is not exceeded. note: any task may release a semaphore so special care should be taken when using counting semaphore to ensure tasks do not release semaphores unnecessarily)

5.6 Memory

For dynamically allocated memory, CatOS provides an allocation system using OS_Malloc(size) to allocate a block of size bytes. This is managed internally by a heap to optimize allocation times as well as a statically allocated array, mapping blocks of memory to the heap for optimized deallocation.

To allocate memory a task must call OS_Malloc(size) with the size of the block to allocate. A void pointer will be returned for use by the user's task.

To free memory, a task must call OS_Free(void ptr) where ptr is the pointer to be freed. This will result in the freeing of that block of memory. The pointer will be returned however it is considered invalid memory and usage of that memory will cause undefined behavior.*

5.7 Macros

These macros are defined in the kernel's header files and usable by the user.

Macro	*Description*	*Value*
OS_MAX_STACK_SIZE	*Maximum stack size supported by OS*	*64*
OS_MAX_HEAP_SIZE	*Maximum heap size supported by OS*	*512*
OS_MAX_TASKS	*Maximum number of tasks supported by OS*	*4*
OS_MAX_SEMS	*Maximum number of synchronization structures supported by OS*	*10*
OS_STACK_MARKER	*Marker for end of stack*	*0xDEABEEF*

kernelErrors		
Macro	*Description*	*Value*
NO_ERROR	*No error occurred*	*0*
UNDEFINED_ERROR	*an unknown error occurred*	*1*
STACK_SIZE_TOO_LARGE	*requested stack size too large*	*2*
TASK_MAX_TOO_LARGE	*requested max tasks too large*	*3*
TASK_MAX_REACHED	*maximum number of tasks reached and another was requested*	*4*
SEM_UNKNOWN_TYPE	*Unknown synchronization structure type given*	*5*
SEM_ZERO_TOKEN	*Attempted to set max tokens of semaphore to 0*	*6*
SEM_TOO_MANY_TOKENS	*Attempted to release a semaphore too many times*	*7*
SEM_COUNT_MAX	*maximum number of semaphores reached and another was requested*	*8*
SEM_TOKEN_MAX_REACHED	*requested stack size too large*	*9*
SEM_INCORRECT_OWNER	*Task attempted to acquire an owned synchronization structure (Common error when using synchronization)*	*10*
HEAP_BAD_BLOCK_SIZE	*Bad block size given to initialize heap*	*11*
ALLOC_BAD_SIZE	*Bad size given for allocation*	*12*
ALLOC_NO_MEM	*Not enough memory to allocate requested memory*	*13*
FREE_INVALID	*Tried freeing invalid pointer*	*14*
kernelObjects		
MUTEX	*Mutual Exclusion Synchronization type*	*0*
COUNTING	*Counting Semaphore Synchronization type*	*1*

Many functions return a 0 on an error and OS_GetError(void) can be used to retrieve what the error was.

5.8 Functions

Function	*Description*	*Input*	*Output*
OS_InitKernel	*Initialize Kernel*	*unsigned numTasks unsigned stackSize*	*Error Code unsigned*
OS_CreateTask	*Create Task*	*unsigned priority unsigned executionTime unsigned deadline*	*Error Code unsigned*
OS_Start	*Start OS*	*void*	*void*
OS_SemCreate	*Create synchronization structure*	*unsigned type unsigned tokenStart unsigned TokenMax*	*Error Code unsigned*
OS_SemAcquire	*Acquire synchronization structure*	*unsigned SemID*	*Error Code unsigned*
OS_SemRelease	*Release synchronization structure*	*unsigned SemID*	*Error Code unsigned*
OS_Malloc	*Allocate Memory*	*unsigned size*	*Pointer to allocated memory void**
OS_Free	*Free Memory*	*void* ptr*	*Unsigned from pointer freed unsigned*
OS_GetError	*Get Last Error*	*void*	*last error that occurred kernelErrors*

6 Conclusion

6.1 Summary

CatOS is a lightweight RTOS that prioritizes deadlines and speed. It is designed for embedded systems where many tasks and resources need to be handled to meet task deadlines. This document covers all current functionalities and necessary details regarding CatOS and will be updated along with CatOS.

6.2 Performance

CatOS's processing time is mostly given to the user's tasks. The tasks are only preempted by the kernel during system calls or every 1ms for scheduling purposes. This means that all processor overhead is isolated to the performance of task switching.

6.3 Future Features

CatOS is by no means finished and will continue to grow. Future features include support for more devices (HALs/BSPs), Optimized system calls, and support for the usage of the currently on board screen for user tasks.

6.4 System Flaws

CatOS is by no means perfect and any users should know the potential flaws so they can be avoided or contingencies can be made.

Synchronization using mutexes and semaphores should be used carefully as they do not always guarantee that the task's deadlines are met. This synchronization of tasks and their deadlines are then left to the user to coordinate so that the kernel does not need to step into the tasks' space with extra overhead.

If synchronization is necessary, it is suggested that the [sec:Timing Requirements]Timing Requirements section be reviewed thoroughly.

As CatOS's features grow more considerations will be added to this document to warn the user in regards to any potential troubles that may occur when using CatOS.