

**San Francisco Bay University**

**CE450 Fundamentals of Embedded Engineering**

**2023 Summer Final Exam**

**Student Name: Melvin Divine Pritchard ID: 19857**

**Part I Embedded System Design Theory**

1. Describe the application areas of the real time operating system (RTOS)

Real-Time Operating Systems (RTOS) have a widespread impact across critical industries due to their ability to ensure precise timing, reliability, and deterministic behavior. RTOS finds application in various sectors, including automotive systems for controlling engines, brakes, and driver-assistance features; industrial automation for managing manufacturing processes and robotics; aerospace and defense for avionics and military systems; medical devices for patient monitoring and diagnostics; and telecommunications for network management and data communication. Additionally, RTOS is vital in consumer electronics, IoT devices, energy management, railway systems, financial trading, and entertainment systems. Its versatility allows it to cater to real-time requirements in these domains, guaranteeing safety, efficiency, and responsiveness.

RTOS, in essence, serves as the foundation for designing embedded systems across industries, allowing for flawless synchronization, task prioritization, and inter-task communication. This technology enables applications as diverse as autonomous vehicles, medical devices, communication networks, and gaming consoles, greatly contributing to the evolution and dependability of modern technology solutions.

1. Explain why the middleware is needed and where.

Middleware is a crucial abstraction layer in embedded system design, serving as a bridge between hardware and application software. Its significance lies in its ability to enhance communication and interaction among different applications within an embedded device. Positioned above the operating system kernel or device drivers, middleware offers standardized communication protocols, security mechanisms, real-time support, and user interface development tools. It finds application in various scenarios, including interoperability with diverse systems, networking, real-time scheduling, security enforcement, and user interface creation. Additionally, middleware streamlines application complexity and aids in device management and scalability, ensuring efficient development and operation of embedded systems.

In short, the aim of middleware in embedded systems is to simplify and optimize interactions by providing features such as connectivity, security, and real-time responsiveness. Its strategic placement and multiple features make it an essential component in a variety of fields, allowing embedded devices to communicate, interface with users, and easily integrate into larger networks or ecosystems.

1. Describe each component’s function of any operating system.

I chose to discuss the components and functions in FreeRTOS, which is a widely used real-time operating system in embedded systems: FreeRTOS offers a comprehensive set of components that together form a robust real-time operating system for embedded systems. Its task management, scheduler, synchronization mechanisms, memory management, and various other features enable developers to build responsive and reliable applications that meet real-time requirements

See below for FreeRTOS Components and their Functions:

**Task Management:**

FreeRTOS provides a task management mechanism that allows you to create multiple tasks, each with its own execution context. Tasks can run concurrently and have their own priorities, ensuring that higher-priority tasks preempt lower-priority ones. The task management component includes functions for task creation, deletion, suspension, resumption, and priority control.

**Scheduler:**

The scheduler in FreeRTOS is responsible for task scheduling and switching. It determines which task should run based on their priorities and scheduling algorithms (such as preemptive or cooperative). Context switching between tasks is managed by the scheduler to ensure efficient execution.

**Synchronization and Communication:**

FreeRTOS provides synchronization mechanisms for tasks to coordinate and communicate effectively. This includes functions for using semaphores, mutexes, and event flags to manage access to shared resources and prevent data conflicts. Additionally, message queues and task notifications allow tasks to exchange information.

**Memory Management:**

FreeRTOS offers memory management capabilities tailored to the needs of embedded systems. Dynamic memory allocation and deallocation functions are provided, allowing tasks to request memory resources from the FreeRTOS heap.

**Time Management and Timers:**

The time management component includes functions to manage delays, timeouts, and timers. Tasks can be made to wait for a specific amount of time, and software timers can be created to execute functions periodically or after a certain interval.

**Interrupt Service Routines (ISRs) Handling:**

FreeRTOS provides interrupt-safe mechanisms to handle ISRs. The ISR handling component allows you to notify tasks from ISRs using task notifications, ensuring that time-critical operations can be efficiently managed.

**Event Groups:**

Event groups allow tasks to wait for combinations of events to occur before proceeding. This component provides functions to set, clear, wait for, and query event group states, enabling efficient synchronization between tasks.

**Queue and Stream Buffers:**

FreeRTOS offers queue and stream buffer functionalities that enable tasks to communicate and share data in a thread-safe manner. These components allow tasks to send and receive messages or data streams efficiently.

**Tick and Tickless Idle:**

FreeRTOS uses a tick interrupt to keep track of time and schedule tasks. The tickless idle feature reduces power consumption by allowing the processor to enter low-power modes when there are no tasks to run. It wakes up the processor only when necessary.

**Portability and Hardware Abstraction Layer (HAL):**

FreeRTOS provides a hardware abstraction layer that allows the operating system to be easily ported to different hardware platforms. The HAL includes platform-specific functions that interface with hardware components like timers, interrupts, and context switching.

1. What general functions are there in any of device drivers, including the description for each?

Device drivers encompass a range of general functions that enable the interaction between hardware components and an operating system. These functions ensure proper hardware initialization, control, and communication. Here are the key general functions found in device drivers, along with their descriptions:

**Hardware Initialization:**

The hardware initialization function sets up the hardware device when it's powered on or reset. It configures the device to a known state, initializes registers, and prepares it for operation.

**Hardware Shutdown:**

The hardware shutdown function is responsible for properly shutting down the hardware device when the system is powered off or reset. It ensures that the device enters a power-off state and is ready for the next power-on cycle.

**Hardware Enable and Disable:**

Hardware enable and disable functions provide the ability to activate or deactivate the hardware device on-the-fly. This allows software components to control the device's operational status dynamically.

**Exclusive Access (Acquire and Release):**

The exclusive access functions ensure that only one process or software component can have access to the hardware device at a given time. The acquire function locks the hardware for exclusive use, and the release function unlocks it once the operation is complete.

**Data Read and Write:**

Data read and write functions facilitate the transfer of data between the hardware device and the operating system or application. The read function retrieves data from the device, while the write function sends data to the device.

**Hardware Install and Uninstall:**

The hardware install function allows for the dynamic installation of new hardware devices while the system is running. Conversely, the uninstall function facilitates the removal of installed hardware components without requiring a system restart.

**Configuration and Control:**

Configuration and control functions allow software components to modify the behavior of the hardware device. This includes setting parameters, adjusting settings, and configuring operational modes.

**Interrupt Handling:**

Device drivers often need to handle interrupts generated by hardware events. Interrupt handling functions manage the response to these interrupts, ensuring proper synchronization and execution of corresponding tasks.

**Error Handling and Reporting:**

The error handling functions detect and manage errors that may occur during hardware operation. They report errors to the operating system or application and take appropriate actions to recover or handle the situation.

**Resource Management:**

Resource management functions allocate and manage resources, such as memory or I/O channels, required for the operation of the hardware device. They ensure efficient utilization of available resources.

**Part II Python Programming**

1. Write a function as a decorator of other function calls for the following operations.

***def******trc1(g):***

*""" YOUR SOURCE CODE HRER """*

***@trc1***

***def sqr(x):***

*return x\*x*

***@trc1***

***def sum\_sqr(n):***

*""" YOUR SOURCE CODE HRER """*

*>>> sqr(3)*

***Calling*** *<function* ***sqr*** *at 0x7f73e7ce8620> on argument* ***3***

***9 # 9 = 3^2***

*>>> sum\_sqr(3)*

***Calling*** *<function* ***sum\_sqr*** *at 0x7f73e7c410d0> on argument* ***3***

***Calling*** *<function* ***sqr*** *at 0x7f73e7c41158> on argument* ***1***

***Calling*** *<function* ***sqr*** *at 0x7f73e7c41158> on argument* ***2***

***Calling*** *<function* ***sqr*** *at 0x7f73e7c41158> on argument* ***3***

***14 # 14 = 1^2 + 2^2 + 3^2***

*Hint: sqr(3) with a decorator @trc1 will be coming trc1(sqr)(3), likewise sum\_sqr(3) should be trc1(sum\_sqr)(3)*

***Program***

def trc1(g):

def wrapper(arg):

fun = g(arg)

print(fun)

return wrapper

return wrapper

@trc1

def sqr(x):

return x\*x

sqr(3)

@trc1

def sum\_sqr(n):

res = 0

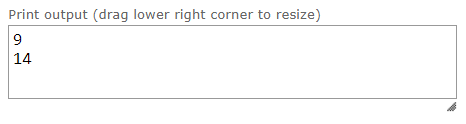
for i in range(1, n+1):

res += i\*i

return res

sum\_sqr(3)

**Running result**



1. Generate a function to implement the following operations.

***def*** ***verify\_add(m, ls):***

*"""Returns True if addition of any two different elements in ls is m.*

*>>> verify\_add (100, [1, 2, 3, 4, 5])*

*False*

*>>> verify\_add (7, [1, 2, 3, 4, 2])*

*True # 7 = 3 +4*

*>>> verify\_add (10, [5, 5])*

*False*

*>>> verify\_add (151, range(0, 200000, 3))*

*False*

*>>>verify\_add(50004, range(0, 200000, 4))*

*True # 50004 = 50000 + 4*

*"""*

*""" YOUR SOURCE CODE HRER """*

**Program**

def verify\_add(m, ls):

unique\_elements = set(ls)

for num in unique\_elements:

complement = m - num

if complement != num and complement in unique\_elements:

return True

return False

def main():

test\_cases = [

(100, [1, 2, 3, 4, 5]),

(7, [1, 2, 3, 4, 2]),

(10, [5, 5]),

(151, range(0, 200000, 3)),

(50004, range(0, 200000, 4))

]

for m, ls in test\_cases:

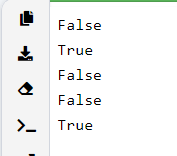
result = verify\_add(m, ls)

print(result)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Running result**



1. Write a function to implement deep-reverse for taking a (possibly deep) tuple argument and reverses it including deep tuple element.

***def deep\_rvrs(tup):***

*"""Reverses tuple with possible tuple elements*

*>>> a = (11, 12, 13, 14)*

*>>> deep\_rvrs (a)*

*(14, 13, 12, 11)*

*>>>tpl = (11, (12, (13,113), 14), 15)*

*>>> deep\_rvrs (tpl)*

*(15, (14, (113, 13), 12), 11))*

*"""*

*""" YOUR SOURCE CODE HRER """*

***Program***

*def deep\_rvrs(tup):*

*if isinstance(tup, tuple):*

*new\_tup = tuple(deep\_rvrs(item) for item in reversed(tup))*

*return new\_tup*

*else:*

*return tup*

*def main():*

*test\_cases = [*

*(5, 6, 7, 8),*

*(5, (6, (7, 77), 8), 9)*

*]*

*for tpl in test\_cases:*

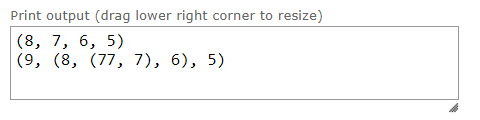
*result = deep\_rvrs(tpl)*

*print(result)*

*if \_\_name\_\_ == "\_\_main\_\_":*

*main()*

***Running Result***



1. Write a Fibonacci class to calculate next number in the ***'Fibonacci'***class by the

***'nxt'*** method. In this class, the ***'val'*** member is a ***'Fibonacci'*** number. The ***'nxt'*** method will return a ***'Fibonacci'*** object whose value is the next number in Fibonacci series.

***class*** *Fibonacci ():*

*"""A Fibonacci number.*

*>>> a = Fibonacci():*

*>>> a*

*0*

*>>> a.nxt()*

*1*

*>>> a.nxt().nxt()*

*1*

*>>> a.nxt().nxt().nxt()*

*2*

*>>> a.nxt().nxt().nxt().nxt()*

*3*

*>>> a.nxt().nxt().nxt().nxt().nxt()*

*5*

*>>> a.nxt().nxt().nxt().nxt().nxt().nxt()*

*8*

*"""*

***def*** ***\_\_init\_\_(self):***

*self.val = 0*

***def******nxt(self):***

*""" YOUR SOURCE CODE HRER """*

***def*** ***\_\_repr\_\_(self):***

*return str(self.val)*

***program***

class Fibonacci():

"""A Fibonacci number.

>>> a = Fibonacci()

>>> a

0

>>> a.nxt()

1

>>> a.nxt().nxt()

1

>>> a.nxt().nxt().nxt()

2

>>> a.nxt().nxt().nxt().nxt()

3

>>> a.nxt().nxt().nxt().nxt().nxt()

5

>>> a.nxt().nxt().nxt().nxt().nxt().nxt()

8

"""

def \_\_init\_\_(self, val=0, next\_val=1):

self.val = val

self.next\_val = next\_val

def nxt(self):

return Fibonacci(self.next\_val, self.val + self.next\_val)

def \_\_repr\_\_(self):

return str(self.val)

def main():

a = Fibonacci()

print(a)

print(a.nxt())

print(a.nxt().nxt())

print(a.nxt().nxt().nxt())

print(a.nxt().nxt().nxt().nxt())

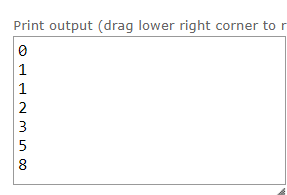
print(a.nxt().nxt().nxt().nxt().nxt())

print(a.nxt().nxt().nxt().nxt().nxt().nxt())

if \_\_name\_\_ == "\_\_main\_\_":

main()

***Result***



*Hint: A new* ***'Fibonacci'*** *object is needed to create and assign****'val'*** *and****'pre'*** *members within****'nxt'*** *method.*

1. Create a class *'****Student'*** first and construct objects with student ***'name'*** and ***'number'*** of course(s) she/he is taking in the current semester. The following operations can be allowed by using magic methods (*'dunder'* method), such as ***\_\_add\_\_(), \_\_str\_\_(), \_\_repr\_\_(), \_\_lt\_\_(), \_\_eq\_\_(), \_\_ne\_\_(), and \_\_gt\_\_()*.**

***class*** *Student():*

*"""*

*>>> a= Student ('Peter', 3)*

*>>> b= Student ('Mike', 4)*

*>>> c= Student ('John', 5)*

*>>> d= Student ('Kelvin', 3)*

*>>> a+b+d*

*10*

*>>> a!=d*

*False*

*>>> b<c*

*True*

*"""*

**Program**

class Student:

def \_\_init\_\_(self, name, num\_courses):

self.name = name

self.num\_courses = num\_courses

def \_\_add\_\_(self, other):

if isinstance(other, Student):

total\_courses = self.num\_courses + other.num\_courses

return Student(f"{self.name} + {other.name}", total\_courses)

else:

raise TypeError("Unsupported operand type(s) for +")

def \_\_str\_\_(self):

return f"{self.name} is taking {self.num\_courses} course(s)"

def \_\_repr\_\_(self):

return f"Student('{self.name}', {self.num\_courses})"

def \_\_lt\_\_(self, other):

return self.num\_courses < other.num\_courses

def \_\_eq\_\_(self, other):

return self.num\_courses == other.num\_courses

def \_\_ne\_\_(self, other):

return self.num\_courses != other.num\_courses

def \_\_gt\_\_(self, other):

return self.num\_courses > other.num\_courses

def main():

a = Student('Peter', 3)

b = Student('Mike', 4)

c = Student('John', 5)

d = Student('Kelvin', 3)

e = Student ('Divine', 5)

result = a + b + d

print(result)

print(a != d)

print(b < c)

print(e > d)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Running result

