AN OSEK/VDX-BASED MULTI-JVM FOR AUTOMOTIVE APPLIANCES

Christian Wawersich, Michael Stilkerich, Wolfgang Schröder-Preikschat
University of Erlangen-Nuremberg
Distributed Systems and Operating Systems
Erlangen, Germany
E-Mail: wawi@cs.fau.de, stilkerich@cs.fau.de, wosch@cs.fau.de

Abstract The automotive industry has recent ambitions to integrate multiple applications from different micro controllers on a single, more powerful micro controller. The outcome of this integration process is the loss of the physical isolation and a more complex monolithic software. Memory protection mechanisms need to be provided that allow for a safe co-existence of heterogeneous software from different vendors on the same hardware, in order to prevent the spreading of an error to the other applications on the controller and leaving an unclear responsibility situation.

With our prototype system KESO, we present a Java-based solution for robust and safe embedded real-time systems that does not require any hardware protection mechanisms. Based on an OSEK/VDX operating system, we offer a familiar system creation process to developers of embedded software and also provide the key benefits of Java to the embedded world.

To the best of our knowledge, we present the first Multi-JVM for OSEK/VDX operating systems. We report on our experiences in integrating Java and an embedded operating system with focus on the footprint and the real-time capabilities of the system.

Keywords: Java, Embedded Real-Time Systems, Automotive, Isolation, KESO

Introduction

Modern cars contain a multitude of micro controllers for a wide area of tasks, ranging from convenience features such as the supervision of the car's audio system to safety relevant functions such as assisting the braking system of the car. The diversity of hardware and software, likely to stem from a variety of manufacturers, complicates the integration to a connected and cooperating system.

Integrating multiple tasks on fewer, but more powerful micro controllers reduces diversity and costs of production. This approach, however, introduces new problems. In a system of dedicated micro controllers, the deployed software is physically isolated from each other. This isolation lacks among tasks sharing the hardware, which enables a malfunctioning task to corrupt the memory of other tasks, spreading the error and possibly resulting in a failure of other tasks running on the micro controller. The impact of such a failure depend on the duties assigned to the software and can be catastrophic. Software development in C and Assembler tends to be error-prone, yet they dominate the development of embedded software. This, along with the ever-increasing software complexity, even worsens the problem and accentuates the need for isolation concepts in embedded systems.

Micro controllers that are equipped with a memory protection unit (MPU) or a memory management unit (MMU) can isolate tasks, however, the hardware-based approach has several shortcomings compared to software-based protection. Beside these micro controllers being more expensive than controllers without an MPU/MMU, software development in a type-safe language such as Java avoids many errors that cannot be detected by unsafe languages, resulting in a more robust software. Moreover, hardware-based solutions are heterogeneous and require different programming on different hardware. Software-based solutions offer a uniform environment independent of the underlying hardware.

Though the approach of software integration arises the discussed problems of error-prone software development and lacking isolation, these problems are well-known from the area of personal computing and concepts have been developed to solve these problems. These concepts can be adapted and migrated to embedded systems. Object-oriented programming (OOP) facilitates coping with complex software and makes the software-development process more robust. Virtual machines such as the Java Virtual Machine (JVM) [8] allow to run applications in an isolated environment and provide memory protection through type-safety without the need for a MPU or even a MMU, preventing the use of arbitrary values as memory references.

Migrating these concepts to an embedded JVM [1, 10, 4, 3] provides less error-prone software-development using Java as an object-oriented programming language and isolates tasks (or threads, to speak in terms of Java) to a certain degree. Yet the approach of a single JVM has several shortcomings. First, static class fields allow the wandering of references. Second, resources are shared equally among the threads, which is not tolerable in real-time systems.

Our prototype system *KESO* will demonstrate that both, a robust development process for embedded software with OOP and the safe integration of tasks on a single micro controller by means of strong isolation, can be achieved while maintaining real-time capabilities and a small footprint suitable for the limited resources available on embedded systems. KESO is built on top of a standard

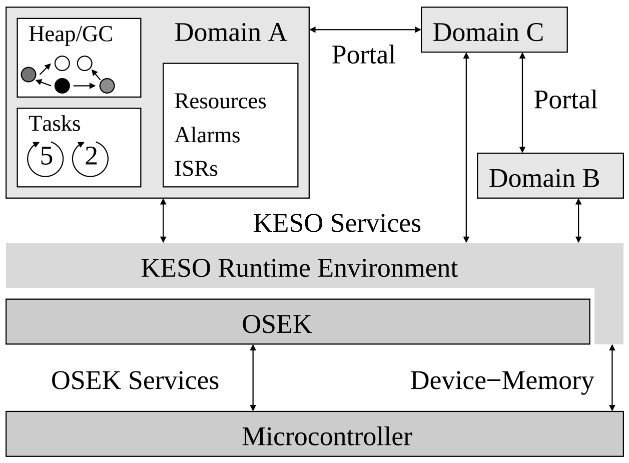


Figure 1. System Architecture

OSEK/VDX [9] operating system. These operating systems are widely used by the automotive industry. KESO thus provides a similar system configuration and creation process and a familiar interface to OSEK system services for embedded software engineers. We adopted the concept of isolated domains as it was first introduced by JX [7] and redesigned the architecture to fit more closely to the needs of resource-constrained environments.

The remainder of this paper is structured as follows: In section 1, we describe the overall KESO architecture and the build process. We evaluated our system with two example applications, which we describe in section 2. The paper is concluded with a discussion of related work in section 3.

1. KESO Architecture

The architecture of KESO is illustrated in figure 1. The KESO runtime environment is based on top of an OSEK/VDX operating system, which affects the KESO design in several aspects.

Scheduling. OSEK/VDX operating systems use the concept of tasks as the basic schedulable unit. OSEK scheduling is based on static priorities. Consequently, KESO also uses the notion of priority-assigned tasks to represent threads of control rather than Java threads, whereby each task is assigned to a KESO domain. Scheduling is handled by the OSEK/VDX scheduler.

Synchronization. OSEK/VDX provides a synchronization mechanism that utilizes a priority ceiling protocol. We decided not to fully support Java monitors because they would degrade the block-free properties of this protocol. We provide limited support for the Java *synchronize* primitive: Entering and leaving a monitor are mapped to acquiring and releasing the special OSEK scheduler resource, that has a ceiling priority equal or higher than the priority of the highest-priority task in the system. We do otherwise encourage the usage of the OSEK priority ceiling API that allows the complete exploitation of priority ceiling synchronization.

Domains. Each domain appears as a self-contained JVM to the user application and represents the fundamental unit of memory protection in a KESO system. The static class fields and the object heap of each domain are separated, whereby each domain can choose a management strategy for its heap.

Strong isolation of the domains is achieved by restricting the scope of each object to exactly one domain, the heap of which the object is allocated from. Each task belongs to exactly one domain. Upon a task switch the domain context is switched to the domain of the task, meaning that new objects are allocated from the heap of the respective domain and the task has a view on the static fields of its domain. This guarantees that a reference never crosses a domain boundary.

Domains are also the fundamental unit of allocation of the resource memory, in what domains differ from Java isolates [2] as used in Squawk VM [11].

Portals. Inter-domain communication is possible via portals. A domain can provide a *portal service*. A portal service consists of a Java interface that offers *service methods* to other domains. The service class that provides the implementation of the portal interface is exported via a static global name. Both the class and the name are configured in the system configuration. Tasks of other domains (*client tasks*) can invoke methods of the portal. The execution of a service method takes place in the environment of the service domain.

The parameters of a portal call are passed by value. If an object is passed as a parameter to a service method, a copy of the object is allocated on the heap of the service domain. Thereby the property of the separated heaps is preserved. The service is executed by the calling OSEK task, which is migrated to the service domain for the duration of the service. The code executed by the migrated task will execute with the scope of the service domain, i.e. has the same view on the system as any regular task of the service domain.

This solution has the advantage that the service method is executed with the priority of the calling task. This conforms to the OSEK priority model. The alternate solution of a dedicated service task in the service domain would require a service task of the same priority as the client task. Thus a service task would have to be created for every possible client task to conform to the priority model. This would use reasonable more resources than the task migration.

OSEK/VDX API. The OSEK/VDX API of KESO [12] allows the user applications to access OSEK services such as the scheduler and the synchronization primitives. This API does also provide mechanisms that allow to restrict the access to the system services to a particular domain in order to guarantee strong isolation.

Hardware Access. Embedded applications often require access to the hardware. In KESO, access to memory mapped device registers is possible through the *Memory* interface. This interface provides methods to access a specific region of memory with methods similar to raw memory access. The memory region accessible through a Memory object can be bounded to prevent a breakout from the Java protection mechanisms. Port-based access to hardware is possible through a similar interface.

These interfaces allow the implementation of the device drivers in Java. No references can be stored through this interface, which prevents the transition of a reference through a commonly accessed region of device memory. Memory objects do also provide a way to efficiently share large amounts of data between different domains without the need to copy.

Heap Management. KESO currently provides three different flavors of heap management: no garbage collection at all, a stop-the-world garbage collector and a highly preemptable garbage collector (RTGC). Each domain can choose an appropriate strategy depending on its needs, which is possible due to the strict separation of the domain heaps. The real-time specification for Java (RTSJ) [5] addresses a similar topic with the introduction of immortal and scoped memory, however, the handling of object references of different heap types in the RTSJ is costly why we opted for a clear separation.

Code Generation Concept. The user applications are developed in Java and available as Java bytecode after having been processed by a Java compiler. The bytecode is compiled to C source code ahead of time by the *KESO builder*. The generation process of a KESO system is illustrated in figure 2.

The generated C code does not only contain the compiled class files, but also the KESO runtime data structures. Moreover, additional code is inserted to retain the properties of a JVM, such as `null` reference checks and array boundary checks, and the code of other services of the KESO runtime layer, such as the garbage collector and the portal services.

Code Size Optimizations. KESO is a static system, that does not allow the dynamic loading of classes at runtime, which opens more optimization potential for reducing the size of the generated system.

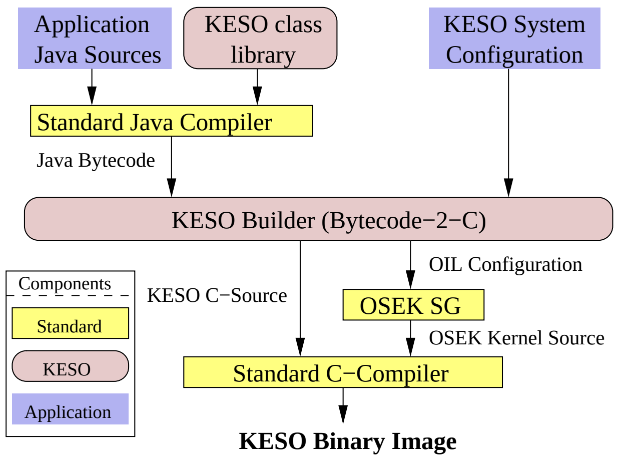


Figure 2. System Generation Process

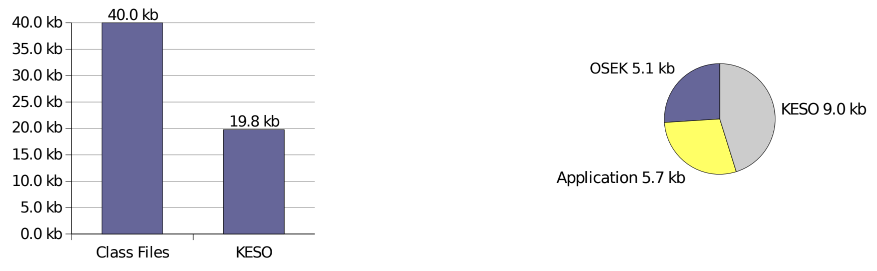
The KESO compiler performs a reachability analysis on the bytecode and eliminates classes, methods and fields, that are not accessed by the user application. This reduces both, the code size of the generated system and the size of the KESO data structures. Additionally, only the parts of the KESO abstraction layer, that are actually used by the application, are added to the generated code, e.g. if an application does not use OSEK resources, the data structures and code for the resource-related services are not added to the generated system.

The data structures of the runtime environment are automatically generated for the smallest memory footprint. The class type information only consumes a few bytes per class, depending on the number of classes, and can be moved completely into the ROM or flash memory. Virtual method calls are eliminated by the compiler as far as possible and the remaining virtual method calls can be entirely replaced by conditional branches.

While the KESO builder is performing global optimization, the generated C source code is also enhanced with additional compiler hints for better code generation. For instance, the C compiler can perform common sub-expression elimination with C functions if the return value of a function only depends on its arguments and the function does not access global memory.

2. Evaluation

Garbage Collector. Figure 3 shows a footprint comparison for a test application that was used to test the real-time garbage collector. The test application processes an infinite loop and maintains a FIFO of fixed size. In each



(a) Total System (b) KESO Components

Figure 3. Footprint comparison of the garbage collector test application

loop cycle, a new element is allocated and added to the FIFO, and another element is removed. The task is activated by an OSEK event triggered by a cyclic OSEK alarm after each loop cycle. The garbage collector can reclaim memory in the waiting phases of the task. As the task enters the waiting state, the garbage collector has to scan the stack of the task. OSEK resources are used to synchronize the stack scanning with the task. Hence the test application requires an almost full-featured OSEK operating system. The applications makes extensive used of string manipulation operations, which is the main reason for the large code size, but was convenient because a lot of memory is allocated during these operations.

Figure 3(a) opposes the size of the class files actually used by the test application and the size of the resulting KESO image. The KESO image is only 50% of the size of the class files, and already includes the KESO VM and the OSEK OS. Looking at the portion the optimized and compiled files take in the binary image (Figure 3(b)), the 40 kB of class files are compiled to only 5.7 kB which is 14.25% of the original size. For comparison, Squawk VM [11] achieves a size reduction to 38% on average by converting class files to an internal suite file format.

Figure 3(b) shows the composition of the KESO system. The bulk of the system is posed by the KESO runtime, that consists almost half (4 kB) of the code of the garbage collector that will remain constant for larger applications. The other major part of the KESO layer is posed by the runtime data structures. These data structures grow with the number of classes and methods used in the system. The size of the OSEK system also depends on the needs of the user applications, however, the test application already makes use of OSEK resources, alarms and events. Thereby, even for larger applications, only a decent increase of the OSEK component has to be expected. The most variable component is posed by the user application. With larger applications, the application component will increase in size most, and the fraction of the other two components in the size of the entire system will shrink.

The RTGC disables the interrupts in several critical sections. As this can delay the reaction to external events, it is important to determine the worst-case latency that can be caused by the RTGC. All critical sections that are secured by disabling the interrupts are O(1). We measured an execution time of 8 μs for the longest critical section of the RTGC on a Tricore controller clocked at 150 MHz. This is less than the time spent in the critical section of the frequently used `ActivateTask()` service of the OSEK OS (12 μs). Our RTGC does therefore not increase the worst-case latency to events.

Small Real-World Application. For a first evaluation of our system with a real world application, we ported an application that was created by students as an exercise to a local real-time systems lecture to KESO. The original application is written in C and runs on top of the same OSEK/VDX implementation that is used for KESO.

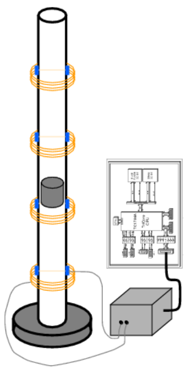
The application is a control software to an automated version of an experiment similar to the ring-the-bell game. Figure 4(a) shows the schematic construction. An iron projectile can be raised and lowered in a plastic pipe using seven electric coils and gravity. The coils are mounted in spacings of 230 mm on the pipe. A photo sensor is attached to each of the coils to detect the movement of the projectile within the pipe. Both the photo sensors and the coils are connected to the general purpose I/O ports of a an Infineon Tricore TC1796b micro controller.

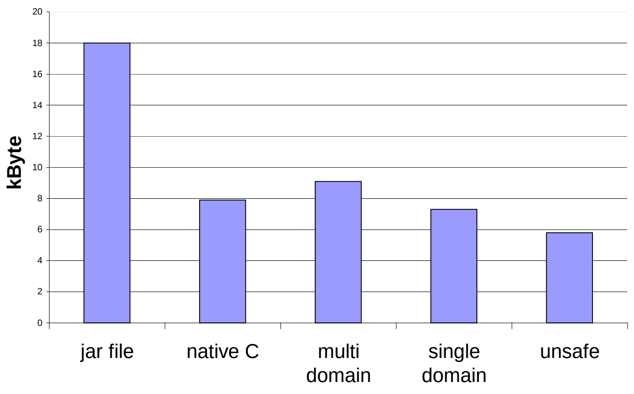
The application implements a finite state machine (FSM). State transition is either caused by interrupts of the photo sensors or a timer (using an OSEK alarm), depending on the state of the FSM. The experiment represents a hard real-time problem. If the application fails to (de)activate a coil in order to catch or decelerate a fast ascending projectile, the projectile will be shot out of the pipe and the programmed sequence cannot be completed any more.

We determined a maximum latency of 240 μs to react to the interrupt of a photo sensor. The successful execution of the program cannot be guaranteed with higher latencies anymore. The measured interrupt frequency ranges from 10 Hz to 27 Hz.

The ported application is implemented very similar to the original C application to ensure comparability. The application does only allocate memory in the startup phase and does therefore not require a garbage collector. As there is only a single OSEK task, the KESO system was configured with a single domain only.

Figure 4(b) shows a comparison between the used Java classes, the original C code and the different configurations of the KESO system. The jar file only contains the classes which are actually used by the program and have to be


(a) Construction



(b) Footprint

Figure 4. Ring-the-bell Simulator

loaded by the JVM. The original C application has a code size of 7.9 kB and a multi domain KESO configuration has 9.1 kB. We expect that the overhead is growing slower than the code size of bigger applications.

The evaluated application does not require multi domain support. The single domain configuration with only 7.3 kB code size is thus the comparable image. The smaller code size was achieved by dead code elimination and auto generated compiler hints for C compiler. This compensated the extra runtime checks added for null pointer and boundary checks. An unsafe system configuration without the runtime checks has only a code size of 5.8 kB, which serves as a basis of comparison to see the amount of code eliminated by the reachability analysis and compiler hints.

3. Related Work

The AJACS Project. The AJACS (Applying Java to Automotive Control Systems) [1] project did general research on deploying Java in automotive control systems, i.e. on static embedded systems, and therefore has the same target platforms as the KESO system. The main objective was developing and defining an open technology that is based on existing standards of the automotive industry, explicitly naming OSEK/VDX operating systems. The expected benefits of using Java on automotive control systems were restricted to a single JVM approach, particularly to software structuring, reusability, dependability, portability and robustness benefits. KESO mainly differs from AJACS in the Multi-JVM approach, that puts the main focus on the isolation of the tasks integrated on the controller.

Squawk VM. The Squawk VM [11] is a small JVM written in Java that runs without an operating system. Java bytecode is converted to a Squawk specific *Suite File* format, that incorporates space, execution and garbage collection simplification optimizations.

Squawk implements an isolation mechanism similar to that of Java Specification Request (JSR) 121 [6, 2]. Squawk encapsulates applications into so-called *Isolates*, whereby each Isolate maintains an own copy of mutable data such as static class variables. An Isolate may contain multiple threads, therefore the Isolate concept shows some similarities to the domain concept used in KESO. Contrary to domains, all Isolates allocate new objects from the same heap, therefore Isolates are no separate units of memory allocation.

Squawk further differs from KESO in the thread and scheduling concept, and the non-preemptable system code including the garbage collector, that can have a negative impact on the interrupt handling latency.

JamaicaVM. The commercial JVM JamaicaVM [10] is designed as a base for embedded software that provides support for the RTSJ [5]. The Jamaica VM build tools also generate native code ahead of time. The main objective of Jamaica VM is to remove the non-determinism from Java in order to create a JVM that is suitable for hard real-time purposes. Jamaica VM as a JVM implementation does not offer concepts for thread isolation.

AUTOSAR. AUTOSAR [**?**] is a joint effort of the automotive industry to develop a standardized software infrastructure and the specification of compatible functional interfaces. The goal of the initiative is the replacement of the component-oriented development process by a function-oriented process.

AUTOSAR applications need to be specified as *software components* that can be transparently mapped to electronic controller units (ECU) in a network of ECUs. To enable this, AUTOSAR provides a *basic software* that abstracts from the hardware by providing a comprehensive set of drivers and hardware abstraction layers. Components communicate through the *virtual functional bus*. The communication over this bus is uniform regardless whether the communicating software components are placed on the same ECU or on different ECUs of the network.

Though outside the scope of this paper, KESO pursuits similar objectives. In an earlier paper [**?**], we presented a variant of KESO where domains can be distributed across ECUs in a network transparently to the application and communicate using the uniform portal mechanism. Contrary to KESO, AUTOSAR does not provide the benefits of a type-safe language that allows the safe integration of components on an ECU.

Conclusion

KESO is the first embedded Multi-JVM for the automotive environment. It eases the transition from dedicated micro controllers to the integration of multiple applications on the same micro controller providing application isolation on micro controllers without the need of hardware-based memory protection. Hardware-based memory protection tends to more expensive hardware and the existing heterogeneous hardware solutions leading to a more complicated software integration process. With Java, we offer a robust and comfortable software development process.

We decided to use an OSEK/VDX system as the base of our development. OSEK/VDX systems are established in the automotive area and known for their small code size and good real-time properties. In KESO, the OSEK/VDX concepts were preferred over the Java compatibility to conserve these properties and to provide a familiar API to the software developer.

While KESO also offers powerful features such as garbage collection, these can be disabled for the entire system or only parts of the system, so that the overhead introduced by these features is only added to the system where required. Our powerful tools mostly automatically generate a system that is closely fitted to the needs of the applications.

The small controller application that we ported from C to KESO shows, that it is possible to produce similar code size in spite of the fact that we added runtime type and boundary checks. This was achieved because of the type-safe Java bytecode, which is easier to analyze and therefore has better attributes for global optimizations.

We are currently working on device drivers for KESO that allow a more generic and high-level access to the hardware and greatly increase the portability of KESO applications. In the future, we hope to provide a common runtime environment to embedded software regardless of the underlying controller hardware. For software that was already certified for one micro controller, this will hopefully ease or even obsolete the certification process of the same software for other micro controllers.

References

[1] AJACS: Applying Java to Automotive Control Systems Concluding Paper V2.0. Technical report, AJACS consortium, July 2002. http://www.ajacs.org/.

[2] JSR 121 Application Isolation API Specification. Technical report, Palo Alto, CA, USA, June 2006. http://jcp.org/aboutJava/communityprocess/final/jsr121/.

[3] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes. A real-time java virtual machine for avionics - an experience report. In *IEEE Real Time Technology and Applications Symposium*, pages 384–396, Washington, DC, USA, 2006. IEEE.

[4] D. Beuche, L. Büttner, D. Mahrenholz, W. Schröder-Preikschat, and F. Schön. JPure - purified java execution environment for controller networks. In *International IFIP TC10 W'shop on Distributed and Parallel Embedded Systems (DIPES '00)*. Kluwer, Oct. 2000.

[5] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. AW, 1st edition, Jan. 2000.

[6] G. Czajkowski. Application isolation in the Java virtual machine. In *15th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '00)*, pages 354–366, New York, NY, USA, 2000. ACM.

[7] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *2002 USENIX TC*, pages 45–58, Berkeley, CA, USA, June 2002. USENIX.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. AW, second edition, 1999.

[9] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, Feb. 2005. http://www.osek-vdx.org/.

[10] F. Siebert. The impact of realtime garbage collection on realtime Java programming. *isorc*, 00:33–40, 2004.

[11] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java$^{\text{TM}}$ on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *2nd USENIX Int. Conf. on Virtual Execution Environments (VEE '05)*, pages 78–88, New York, NY, USA, 2006. ACM.

[12] M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, A. Gal, and M. Franz. OSEK/VDX API for Java. In *Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS '06)*, pages 13–17, San Jose, California, USA, Oct. 2006. ACM.