AUTONOMOUS NETWORKED TACTICAL SENTRIES*

John B. Bowles†, Larry M. Stephens†, Michael N. Huhns†,
Richard W. Tobaben††, Sanath Yekollu†, and Hareesh Kolpuru†

† Electrical and Computer Engineering
University of South Carolina
Columbia, SC 29208
e-mail: bowles@engr.sc.edu

†† Raytheon Systems Company
Mail Stop 8461
Plano, Texas 75023

1. INTRODUCTION

The objective of the Autonomous Networked Tactical Sentries (ANTS) program is to develop a suite of sensors of different types and capabilities that can be deployed as an array on a battlefield. Each sensor is self-contained and equipped with its own microprocessor, memory, communications electronics, and power source; all contained in a cylindrical module approximately 14.6 centimeters (5.7 in.) long by 4 centimeters (1.6 in.) in diameter.

The array of sensors will be able to detect the position and movement of enemy forces and materiel with minimal involvement or risk for friendly forces. Within the array of sensors, coordination and control is distributed with the local processor analyzing the local data collected, handling exceptions, and communicating the results to a higher level entity for further analysis or the appropriate personnel to make strategic and tactical decisions about the battlefield.

The Autonomous Networked Software is an enabling technology for ANTS. It consists of Java-like applets residing on a Universal Sensor Interface Chip (USIC) that interfaces with and controls the associated sensors. Figure 1 illustrates the overall software architecture for the system.

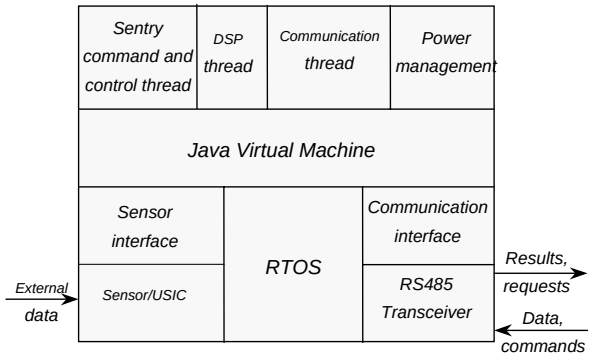

Figure 1. ANTS Software Architecture

Its major components are:

- Sentry software interpreter for Java bytecodes (Java virtual machine);
- Real-time operating system, including interrupt service routines, and low-level interface code;
- Java application threads for communications, power management, signal processing, and sensor command and control.

In addition, a Java class file linker-loader preprocesses of the Java class files so that they can be interpreted directly by the ANTS Java Virtual Machine (JVM) and loads the software onto the sensor. The underlying hardware has been designed for power conservation, modular configurability, and maximum versatility.

1.1 Why Java

Java was originally designed for systems that perform networked computing and communications. Java source code is compiled into byte codes, which are then interpreted by the Java Virtual Machine (JVM), which runs on the host computer. This process provides a great deal of machine independence and flexibility. Applications can be developed on one system, downloaded, combined with pieces of Java programs from other machines, and run on a different system.

Using an interpreted high-level language such as Java for microprocessor-based sentries has several advantages. First, software for an ANTS sentry can be developed on any platform and downloaded to the sentries. This provides a level of flexibility unmatched by other high-level languages.

Second, several features of the Java language itself are advantageous for ANTS:

* This work was supported in part by the US Defense Advanced Research Projects Agency under contract F04701-97-C-0016 for the Technology for Tactical Sensors and in Small Unit Operations Program

- Java is multithreaded, which makes it straightforward to construct ANTS applications that must manage sensor signal sampling, signal processing, communications, and power management concurrently. This multi-thread aspect of Java is particularly well suited to the ANTS multi-processing requirements. Other languages such as C and C++ do not support concurrency directly.
- Java is a pure, object-oriented language, enabling ANTS software to be developed and managed in a modular fashion, and enabling modules to be defined that correspond naturally to real-life objects.
- Java byte code is compact, requiring half as much space to store and half as much bandwidth to download a program over a network, as does machine language.

Third, new or modified Java threads can be downloaded dynamically to reconfigure an ANTS sensor array in the field. For other applications, ANTS sensors can function as web servers, enabling them to be interrogated and controlled from standard web browsers.

Finally, since the use of Java for ANTS is based on off-the-shelf Java compilers, application development is more cost effective.

The major disadvantage of Java is execution speed. Interpreted software is much slower than software compiled into a processor's native instruction set. Preliminary studies of ANTS software indicate that equivalent functions might be a factor of 10 slower in Java than in native, machine code. This speed disadvantage can be substantially mitigated by writing low-level interrupt-service routines in assembly language and computationally intensive analysis routines in C and using Java primarily for control of the microprocessor, where its robustness and flexibility are most important.

2. JAVA BYTECODE INTERPRETER (JAVA VIRTUAL MACHINE)

The Java Virtual Machine (JVM) is an abstract computer that runs compiled Java programs. It is "virtual" because it is implemented in software on top of a "real" hardware platform and operating system. Java programs are compiled into instructions called Java bytecodes that are interpreted by the JVM. Each instruction consists of a one-byte opcode followed by zero or more operands.

The JVM has a stack-oriented architecture that helps to keep both the number of instructions and the size of each instruction small. The JVM has five basic parts:

- the stack,
- the registers,
- the garbage-collected heap, and
- the method area.

The set of registers, the stack, the heap, and the method area constitute the "virtual hardware" of the Java Virtual Machine. These parts are abstract, just like the machine they compose, but they must exist in some form in every JVM implementation.

2.1 The Stack and Associated Registers

Information in the Java stack is organized in "stack frames". Each stack frame contains the state of one Java method invocation. When a program invokes a new method, the Java Virtual Machine pushes a new frame onto that program's stack. The stack frame, as shown in Figure 2, contains space for the method's local variables, its operand stack, its parameters, static link and return value. The maximum number and size of the local variables are calculated at compile-time and given to the interpreter, so that the interpreter knows how much memory will be needed by the method's stack frame. When the interpreter invokes a method, it creates a stack frame of the proper size for that method. When the method completes, the JVM removes the corresponding frame.

The Java stack is used to store parameters for bytecode instructions and the results of bytecode operations, to pass parameters to and return values from methods, and to keep the state of each method invocation. The stack is word-based. Each time a value is pushed onto the Java stack, it goes on as a 32-bit word (although *longs* and *doubles* actually go on as two words).

The registers of the JVM hold the machine's state and

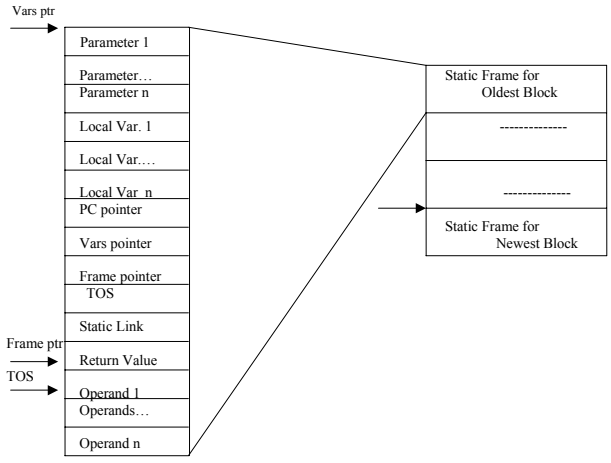

Figure 2. ANTS JVM Stack Structure.

control its operation; they are updated after each bytecode is executed. The JVM has a program counter, called the *pc* register, and three registers that manage the stack — the *optop*, *vars*, and *frame* registers. The JVM needs only a few registers because the bytecode instructions operate primarily on the stack.

Optop is a pointer to the top of the operand stack. This is always the topmost section of the stack; hence the *optop* register always points to the top of the entire Java stack.

The operand stack is used as a workspace by bytecode instructions; both parameters for the bytecode instructions and the results of operations on those parameters are placed on the top of the stack. For example, the *iadd* instruction adds two integers by popping two *ints* off the top of the operand stack, adding them, and pushing the result back onto the stack. Since each word on the stack is 32 bits, *Optop* is implemented as a *pointer to integer* in the ANTS JVM.

The local variables section of the stack contains all the local variables, including any parameters passed to the method, for the current method invocation. *Vars* is a pointer to the first local variable of the currently executing method. The local variables section of the Java stack is treated as an array of words starting at the location pointed to by the vars register. Byte codes that deal with local variables generally include an array index, which is an offset from the *vars* register. Values of type *int*, *float*, *reference*, and *return value* occupy one entry each in the local variables array. Values of type *byte*, *short*, and *char* are converted to *int* and padded before being stored into the local variables and values of *double*, and *long* occupy two locations on the stack.

Frame is a pointer to the execution environment of the current method. This section is used to maintain the operations of the stack itself. When a running thread invokes a Java method, the calling-method first pushes any parameters to be passed to the called-method onto the stack. The JVM then creates and pushes a new frame onto the thread's Java stack. This frame then becomes the current stack frame. The JVM saves the calling-method's execution environment (*pc*, *vars*, *frame*, and *optop* registers) in the new stack frame, and updates those registers for the new method.

2.2 The Method Area and Heap

The method area is where the byte-codes reside and the heap is used for allocating new objects in memory. The Java language does not allow memory to be freed directly; instead it keeps track of the references to each object on the heap, and automatically frees the memory when the object is no longer referenced. The heap and method areas are managed for the JVM by the ANTS Real Time Operating System.

Our implementation of the JVM does not support garbage collection because of real-time scheduling constraints and memory space limitations; hence, there are some limitations on how a program can allocate and dereference objects.

2.3 Java Bytecode Interpreter

The JAVA bytecode interpreter implements the JVM. It acts as a "virtual processor" and executes the instructions in the stream of bytecodes. The interpreter is written in C.

The overall structure of the interpreter is shown in Figure 3. The interpreter is initialized by setting the *pc*, *optop*, *frame*, and *var* registers to point to the address of the Java bytecode program to execute, the top of the operations stack, the execution environment (frame), and the local variables, respectively.

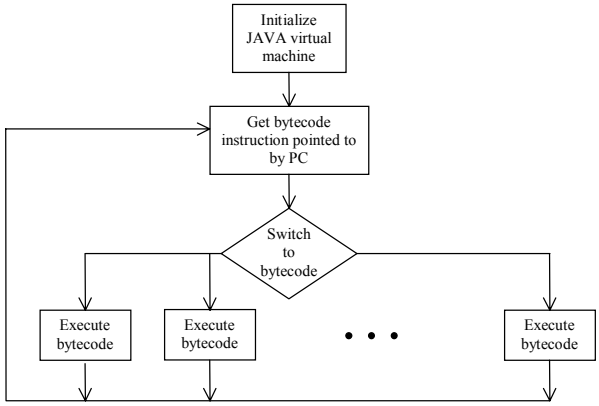

Figure 3. Architecture for the Java Virtual Machine Interpreter

The interpreter then enters a basic fetch-execute cycle. It fetches the bytecode pointed to by *pc*, decodes the instruction, and switches to the code to execute that instruction. Once the bytecode instruction has been executed the *pc* is incremented to point to the next instruction and that instruction is fetched. The inner loop of the interpreter is essentially:

```
Do {
    Fetch a byte
    Execute an action based on
        the value of the byte
} While (there is more to do);
```

The performance of the JVM is improved by declaring *pc*, *optop*, and *vars* as "register variables", thus associating these virtual registers with the real registers of the underlying hardware.

The ANTS Java interpreter implements the complete core instruction set of 201 bytecodes from Sun Microsystems JVM specification.

2.4 Class Loader

The Java class file is the format that a compiled Java class is saved in. Class files are the Java equivalent of the object files produced by other compilers, but the instructions in them are intended to be interpreted dynamically rather than as static opcodes. A Java class file contains several data items required by the runtime system; these include the virtual machine code for each method provided by the class, a symbolic reference to the superclass of the class, a list of fields defined by the class, and a constant pool containing littorals and symbols used by the class. Bytecode instructions that make symbolic references use the constant pool table (in the Java class file) to map to a string representing the reference which is then resolved dynamically. For example, the instruction *invokevirtual* uses an index into the constant pool to find the class name, method name, and method signature (number and type of parameters, and return type) of an instance method. "Resolution" of this reference requires locating the appropriate class file and parsing it until the correct method is found. In addition, several bytecode instructions (e.g., *getfield*, *invokspecial*) obtain information such as what method to execute, or the amount of space to allocate for the local variables of a method from the class file.

In ANTS the dynamic resolution of symbolic references is not feasible because: 1) there is no file system in which to store the class file; and 2) the system does not have enough memory to store the class files. Hence, the ANTS Linker-Loader must resolve all the symbolic references when the program is linked and provide the real memory addresses to the JVM as operands following the bytecode.

The Java JVM currently defines 201 byte-code instructions. Fourteen of these instructions require the resolution of symbolic addresses before they can be executed.

2.5 ANTS Implementation Differences from "Standard" Java

There are several differences between the ANTS Java Virtual Machine and the "standard" Java Virtual Machine. These differences were imposed by the system architecture and the manner in which the ANTS sensor modules will be deployed. The most important differences are:

1. The Java Application Programming Interface (API) is a set of runtime libraries that provides a standard way for Java programs to access the resources of the host system. The API's functionality must be implemented for a particular platform before that platform can host Java programs. The ANTS JVM currently has no implemented API.
2. Normally, a Java program is interpreted and symbolic references, such as those for method invocation, object field access, object creation, and certain stack manipulations are resolved at runtime by a dynamic class-file loader. Since the ANTS JVM does not have access to the class file, all of these symbolic references are resolved by the ANTS Linker-Loader and passed to the JVM as parameters following the corresponding bytecode.
3. The Java virtual machine specification provides only general rules for multi-threading. The ANTS JVM calls the RTOS for creating, scheduling and destroying threads and hence depends completely on the underlying operating system for this functionality. Since the current implementation of the ANTS JVM cannot manage the execution of its threads it does not provide compatibility with the Java language thread class.

   The ANTS JVM provides the capability of multi-threading using internal threads of the ANTS RTOS. Scheduling of Java threads is also done by the RTOS which provides priority driven preemptive task scheduling. Implementation of multi-threading in the ANTS JVM has been done using re-entrant code so that one thread does not interfere with other running threads. Each thread also maintains its own heap for dynamic allocation of memory.

3. ANTS REAL TIME OPERATING SYSTEM

The ANTS Real Time Operating System (RTOS) manages the execution of the ANTS application software. It provides the functionality needed for real-time system operation and to support the Java virtual machine. This includes:

- managing Java threads and other tasks;
- providing communication between threads;
- making static allocation of memory for all tasks in the system.

In addition, the RTOS provides routines to respond to hardware and timer interrupts and to boot the system. It also interfaces to the development system and provides the capability to download programs and data.

3.1 Tasks and Thread Management

Real-time systems require multi-tasking for the concurrent execution of essential tasks. The logical correctness of the system depends on both the correctness and the timeliness of the outputs.

The ANTS RTOS uses a Task Control Block (TCB) model to keep track of the various tasks in the system. Although this model is more complex than the stack model used in some embedded systems applications, it allows task priorities to change dynamically and it allows more sophisticated scheduling algorithms to be used. The TCB model also helps the kernel with keeping track of the status (executing, ready, suspended, or dormant) of each task.

A Task Control Block is created for each task when the task is initialized by the RTOS. Separate stack structures are also reserved for each task. The TCB contains sufficient information, to enable the task to resume execution from where it left off when it is suspended.

As tasks run they transition between four states:

1. Executing,
2. Ready,
3. Suspended, and
4. Dormant.

Figure 4 shows the state transitions in the TCB model system. The *executing* task is the task that is currently using the processor resources. Tasks in the *ready* state are those which are ready to run but are not running — usually because another, higher priority task is running. Tasks that are waiting on a particular resource, and hence are not ready, are in the *suspended* state. The *dormant* state refers to the state of a task which exists, but is unavailable to the operating system.

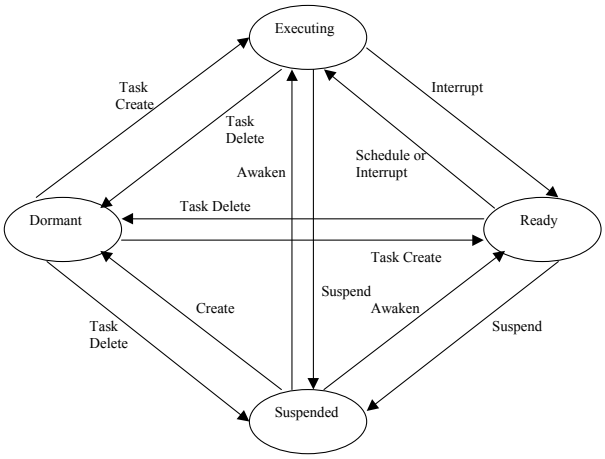

Figure 4. State transitions in a TCB model.

A task can enter the *executing* state when it is created (if no other tasks are ready) or from the *ready* state (if it is eligible to run based on its priority). A task enters the ready state if it was executing and its time slice runs out, or if it is preempted by a higher priority task. When a task is *suspended*, it can enter the ready state if the event that it is waiting on occurs. If the task is in the dormant state, then it enters the ready state upon creation (if another task is executing).

Java thread management must be priority-driven. The RTOS utilizes a priority-driven preemptive task scheduler for thread management with each Java thread corresponding to a separate task. The scheduler implements 10 levels of priorities (compatible with the 10 priority levels in standard Java) numbered from 0 to 9. Whenever a higher-priority thread becomes *ready*, the current thread is preempted. Equal priority threads execute in a round-robin manner until they run to completion.

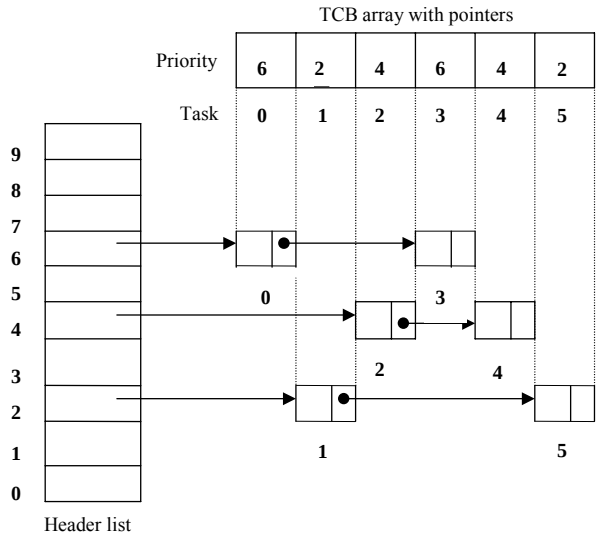

Figure 5. Priority ordering of task control blocks.

Figure 5 shows the ordering of TCBs according to their priority levels. The scheduling algorithm scans the header list starting with the highest priority level and traverses through the TCB links to build a round-robin queue of all equal priority tasks. The queue is first set up when the system is initialized and it is rebuilt when any change in priorities occurs. For the tasks shown in Figure 5, the algorithm produces a queue having two tasks, 0 and 3, both of priority 6.

Task priority assignments are based on the relative importance of the tasks or their deadline requirements, but they do not make any assumptions about the task execution times or possible resource hogging

situations. This results in a finite schedulability bound for each task in the system.

Priority inheritance is used to protect against priority inversion caused when a high priority task must wait on a resource held by a low priority task and the low priority task is preempted by a medium priority task.

In addition to the task scheduling function, the RTOS also tracks the status of the tasks in the suspended list and it maintains a set of allocation tables which are used to arbitrate between tasks that are pending on the same resource. If a resource becomes available to a pending task, then the resource tables are updated and the eligible task is moved from the suspended list to the ready list.

The task dispatcher actually allocates the processor time to the next ready task in the round robin list. It selects the ready task from the ready list prepared by the scheduler and prepares the task for execution by loading its state from its TCB onto the processor registers and starting the time-slice counter for that task.

3.2 Intertask Communication and Synchronization

Communicating data between tasks and synchronizing tasks is an important problem in any multi-tasking system. A related issue is the sharing of certain resources that can only be used by one task at a time. Semaphores, mailboxes and message passing are all used to solve these problems according to the functionality required.

Message passing is the primary means of communication among the various tasks in the system for ANTS. Tasks use this mechanism to signal conditions to other waiting tasks and to pass information among themselves. Tasks that are waiting on interrupts also use message passing as a synchronization primitive. Semaphores are used for mutual exclusion.

The message queue structure is composed of two separate queues. One is a queue of messages that have been sent to the message queue, but have not yet been received by a task. The other is a queue of tasks that are waiting for a message from the message queue.

3.3 RTOS-JVM Interaction

Although the JVM isolates the Java code from the hardware specific and operating system details, an embedded Java implementation still requires tight coupling between the JVM and the RTOS. This coupling enables the JVM to request underlying kernel services in order to be real-time compliant and to support the language features.

The Java language provides method constructs for multi-threading, thread synchronization and communication. To implement these, the JVM must map the Java threads to individual kernel tasks and rely on the kernel services to schedule and synchronize the threads in accordance with the Java thread priorities. The JVM seeks support for hardware related functions through native function calls to device driver routines and interrupt handlers. Hence, the RTOS has to handle both native code tasks and Java based threads as part of its management function. Both requirements are simplified in ANTS by writing the JVM in reentrant code and executing each Java thread as a separate instance of the JVM. Each JVM execution instance is mapped to a separate RTOS task.

In ANTS a fixed amount of heap memory is allocated to each JVM task during system start-up. This static allocation of heap memory circumvents the need for garbage collection to dynamically reclaim freed up memory. Thus it avoids the indeterminacy caused by garbage collection, but it leads to strict programming restrictions imposed on the user. For example it is not possible to create a large number of new objects in a loop.

4. ANTS DEVELOPMENT SYSTEM

The ANTS development environment consists of a compiler for the Java language, a set of libraries containing signal processing routines and control routines for the microprocessor, and a linker-loader for configuring the compiled code into a form suitable for loading into the processor, where it can be executed by the JVM. The principal components of this environment are shown in Figure 6.

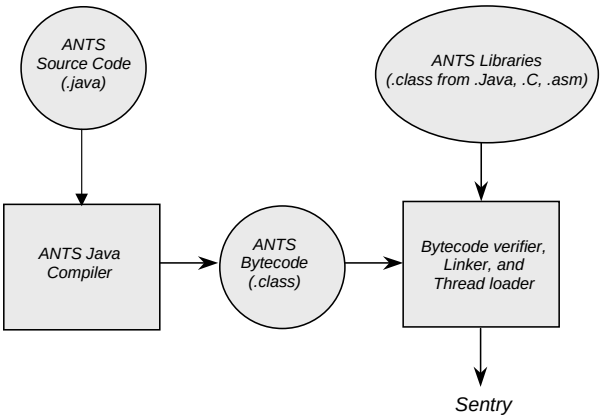

Figure 6. Major components of the ANTS software development system.

Applications for ANTS written in Java source code are compiled into *class* files using a standard Java compiler. Each class file contains the Java bytecodes for the included methods. These methods (and methods in other class files) are referenced symbolically in the *constant pool table*—the Java virtual machine symbol table for a class file. In standard Java, these symbolic references are resolved at *run time*.

Since one design decision was to eliminate garbage collection (and dynamic allocation of memory space), the linker-loader must resolve the symbolic references before the code is loaded. In addition, the linker-loader must identify all required memory space for the objects themselves before the code can be run.

The requirement to identify memory space for objects at link time imposes certain restrictions on allowable Java code. Primarily, the system must control the creation of an arbitrary number of new objects by monitoring and restricting the use of the Java *new* statement in loops and conditional statements. (In contrast, the Java *execution* stack can grow to a specified limit.)

The Java language specification allows flexibility as to when resolution and linking occur. To improve deterministic runtime performance, ANTS JVM resolves and links all classes before they are downloaded into the memory. (Other virtual machines usually defer resolution and linking until they are needed at runtime.)

The current form of ANTS JVM cannot run applications relying on the libraries from the Java programming language.

5. ANTS HARDWARE

The ANTS hardware has been designed for low power, modularity, autonomous reconfigurability, and maximum versatility. Hardware for ANTS consists of the sensor transducers, the Universal Sensor Interface Module (USIM), antenna, and battery pack. The battery pack is configurable to meet the power consumption needs of the mission and size limitations of the delivery mechanism. The antenna is planned to be shared between a wireless LAN transceiver and a GPS receiver. The USIM is the control, communication, data acquisition, and processing element of the ANTS hardware. It is intended to allow any of 10 different sensor transducer types to be attached in a variety of configurations and combinations. The 10 sensor transducer types are:

1. Acoustic
2. Seismic
3. Magnetic
4. Infrared Imaging
5. Atmospheric Temperature
6. Wind Velocity
7. Atmospheric Humidity
8. Rain
9. Barometric Pressure
10. Visual Imaging

For high data rates or tightly coupled control sensor transducer types such as infrared or visual imaging and magnetometers, only one device is attached to a USIM. For the other sensor transducer types, the USIM incorporates a "non-imaging" interface that supports up to 16 channels of incoming analog signal with conversion for digital processing. The 16 channels can be allocated to these "non-imaging" sensor transducers in mission optimal configurations and combinations.

5.1 Universal Sensor Interface Module (USIM)

The conceptual design of the USIM is depicted in Figure 7 and Figure 8 is a high level schematic of its architecture. In its minimal configuration, the USIM consists of three Universal Sensor Interface Cards (USIC): a master processor USIC, a Transceiver/GPS USIC, and a Non-imaging USIC. In a maximum configuration, three slave processor USICs are added to the minimal configuration and the Non-imaging USIC might be replaced with an Imaging USIC. The system design is founded upon software control of USIM resources to optimize problem solution vs. power consumption. The ANTS modules may be deployed in an array or "anthill" up to 250 meters in diameter. Data acquired by one ANTS node is combined with data from other nodes of like sensor transducers and fused with data from unlike sensor transducers to provide reliable coordinated detection, tracking and recognition of target objects ranging from personnel to large motorized and tracked vehicles. The ANTS nodes provide reliable, low-cost, unattended micro-electronic sensors that can give advanced warning of hostile presence in the area up to and including identification and targeting information.

5.2 Master Processor USIC

The Master Processor USIC is the primary control card in the USIM. The USIC design is based on the Motorola MPC555 PowerPC microprocessor that features:

- software controlled clock rate from 1 to 40 MHz,
- full PowerPC instruction set including double precision floating point,
- dual timer control units,
- dual CAN network buss interfaces,
- three serial ports,
- 32 channels of 10 bit A/D converters,
- 28 Kbytes of dynamic RAM, and
- 384 Kbytes of flash memory on the chip.

This powerful processor is augmented with an additional 2 Mbytes of flash memory and 3 Mbytes of static RAM. The USIC also contains 256 Kbytes of dual ported RAM and Ethernet communication chips as well as other supporting electronics. Because the processor power consumption is a function of the CPU clock speed, the software controlled clock selection allows the ANTS software to match power consumption to processing cycle time requirements. The PowerPC instruction set permits the software to utilize the most effective digital signal processing algorithms to accomplish mission objectives. The dual timer units allow very accurate timing control independent of the CPU. The CAN network buss interfaces allow networking of the processors within the USIM for control and internal communication. Digitized data is delivered to one side of the dual port memory from either the imaging or non-imaging interface where it can be delivered to a slave processor USIC for analysis or transmitted out of the ANTS node for external analysis.

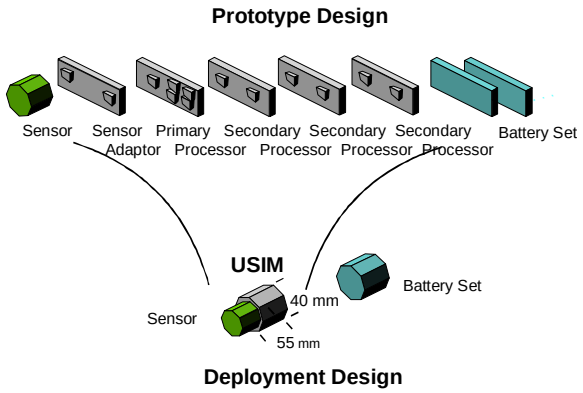

Figure 7 Universal Sensor Interface Module

5.3 Slave Processor USIC

The Slave Processor USIC is the data processing element of the USIM. A USIM can be configured with up to three Slave Processor USICs depending upon the anticipated processing requirements of the application. The Slave Processor USIC has resources similar to those of the Master Processor but it emphasizes the data processing functions more than command and control. Like the Master Processor USIC, the Slave Processor USIC is based on the MPC555 PowerPC and has available all of the same on-chip resources. The Slave Processor USIC does not have the Ethernet capability, but the static RAM is increased to 5 Mbytes along with the 256 Kbytes of dual port RAM.

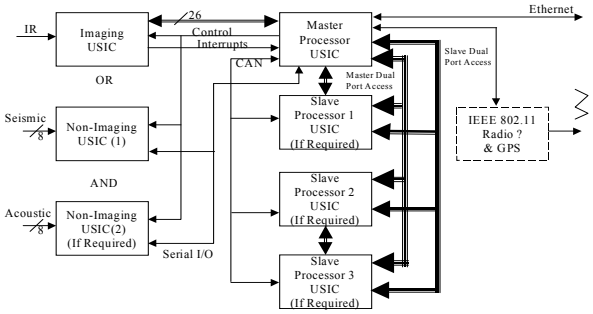

Figure 8. ANTS USIM Block Diagram

Since the same CPU is used as in the Master Processor USIC, the Slave Processor USIC can also vary its clock rate and, therefore, its power consumption to match the processing load and cycle time requirements. The Master Processor USIC makes data available to each Slave Processor USIC through the Master Processor USIC's dual port RAM which is directly addressable by the slave. Results are conveyed back to the Master Processor USIC through each Slave Processor's dual port RAM that has local Slave Processor access on one side and Master Processor access on the other.

5.4 GPS/Transceiver USIC

The GPS/Transceiver USIC is not included in this phase of development. However, suitable integrated circuits have been identified for both low power GPS and Transceiver functions. The Transceiver IC accepts the output from the Ethernet IC on the Master Processor USIC and transmits in accordance with the IEEE 802.11 protocol at 2 Mbps. The Transceiver wireless LAN bandwidth is expected to increase to 10-11 Mbps by next year. This increased communication bandwidth is a key component in the ANTS

implementation that will allow viable data communication and control of an Anthill network to accomplish coordinated detection, tracking, and identification. The GPS function will allow precise relative and absolute position determination for each USIM. Precise relative position allows coordinated data processing for tracking and identification. Precise absolute position allows anthill reporting to external agents to contain precise target object location.

5.5 Non-Imaging USIC

The low power Non-Imaging USIC accepts 8 channels of input analog signal, delta-sigma analog to digital conversion with up to 22 bits of noise free resolution with data rates programmable from 7.5 to 3840 Hz. A USIM may be configured with from zero to two Non-Imaging USICs depending on the number and type of non-imaging sensor transducers to be attached to the USIM. Some non-imaging sensor transducers may be connected directly to the MPC555 ports to take advantage of the many peripherals embedded in the processor chip itself.

5.6 Imaging USIC

There are currently two instances of imaging USIC planned for ANTS:

- 160x120 pixel Miniature InfraRed Camera (MIRC)
- 320x240 pixel Infrared Camera

Each of these USIC contains electronics to deliver digitized data from their associated sensor transducer to the Master Processor USIC of the USIM for processing. These two infrared cameras can provide an image frame at rates up to 30 Hz.

5.7 Summary of ANTS Hardware

ANTS is designed to meet the need for low power, high sensitivity detection with long active life, and autonomous embedded data analysis and reporting capabilities. The pivotal component, the USIM, is configurable to meet this task with different types of sensor transducers and variable local processing capacity and it is software controllable to accommodate the varying power and processing requirements.

6. PAST AND FUTURE DEVELOPMENTS

ANTS is very much a work in progress. The original system specifications called for only a single sensor, a TMS320C Digital Signal Processor, 32K 16-bit words Flash EPROM on chip memory, and 64K 16-bit words of external RAM. The TMS320C has a 16 bit, fixed-point processor and a Harvard architecture (separate program and data address spaces). Use of the TMS320C placed many restrictions on the system design. For example, the Java bytecodes had to be stored in 16-bit words, followed by their operands, rather than in 8-bit bytes; float, double, and long byte-code types were not implemented; and the stack was defined as 16- rather than the standard 32-bits wide. These differences placed severe restrictions on what Java programs could be run.

However, as is true of many "real-world" projects, the operational concept evolved as the development proceeded. The present concept (described here, and reflecting the state of the design during the summer, 1999) allows up to 10 types of sensors and as many as 4 processors in a sentry. In addition, primarily for reasons of supportability and maintenance, a Commercial-Off-The-Shelf (COTS) operating system, instead of the custom built RTOS system, will be used. Also, to reduce manufacturing complexity, all of the software will be loaded into ROM, which is relatively slow, and when the sensors are attached, the module will configure itself automatically by downloading the RAM with the appropriate software from the ROM. To support these changes ANTS now uses the Motorola MPC555 PowerPC as its processor.

The new processor brings significant improvements. It has a 32-bit Von Neumann architecture (instructions and data in the same address space), which simplifies the design and memory allocation tasks, higher speed, and it supports multiple processor configurations. Also, since it addresses memory in bytes rather than words it provides a better match with the JVM software.

REFERENCES

1. T. Lindholm and F. Yellin, "The Java™ Virtual Machine Specification", Sun Microsystems Inc.
2. J. Meyer and T. Downing, *Java Virtual Machine*, 1st Ed., O'Reilly Publications, ISBN: 1-562592-194-1, 1997
3. K. Ramamritham and J. A. Stankovic, *Scheduling algorithms and Operating Systems Support for Real-time Systems*.