# Channeled Micro-System (CHANSYS) Design

1st Author
1st author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
1st author's email address

2nd Author
2nd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
2nd E-mail

3rd Author
3rd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
3rd E-mail

## ABSTRACT
This article introduces new scalable operating system architecture for real-time parallel computations. The operating system implements a novel Resource-Owner-Service (ROS) programming model and, according to the model, provides means of task execution, communication and synchronization. The model defines the concept of a channel as key communication and synchronization entity, and illustrates the use and efficiency of channels in parallel operations. Inherent scalability of an ROS-based system makes it a perfect candidate for deployment in wireless sensor networks, known for their constrained resources and growing popularity. Sections below give an overview of design goals and principles behind the operating system, discuss the ROS programming model in detail, describe core operating system functionality, and furnish examples of parallel communications, typical channel topologies, statically and dynamically controllable task scheduling scheme, exemplary system layouts to fit different hardware resource usage scenarios, as well as possible requirements for programming languages to efficiently support the new operating system architecture.

## Categories and Subject Descriptors
D.4.0 [**Operating Systems**]: General. D.4.7 [**Operating Systems**]: Organization and Design - *Real-time systems and embedded systems*.

## General Terms
Algorithms, Performance, Design, Languages

## Keywords
Operating systems, multi-tasking, parallel programming models, task communications.

## 1. INTRODUCTION
Designing new operating system architectures usually pursues the following goals of (a) achieving low resource consumption, (b) maintaining scalability with regard to processor power, concurrency level, and storage capacity, that is, being equally applicable (with reasonable implementation tradeoffs) to low-performance micro-computers and powerful servers, (c) providing unified solution for heterogeneous execution environments, i.e., networks comprised of multiple machines of different architectures, level of parallelism, and performance, and (d) establishing easier implementation, deployment, and update processes, minimizing the use of specific languages, compilers and development environments.

Many of the aforementioned objectives become of extreme significance when choosing a platform for a wireless sensor network. Wireless sensor networks are specifically mentioned here because, with the advancement of the technology, we can expect a burst of production and adoption of compact, low-power processor, memory, and transmission devices.

Given the nature of the problems they solve, designing wireless sensors is always a tradeoff between the physical constraints and computing power. With that in mind, two major trends can be foreseen in the future: either further miniaturization keeping the processor power at best the same, or inevitable increase of computing power when decreasing physical parameters does not add value to the system (standardized connection ports/pipe diameters, solar/inductive non-accumulative power supply, and so forth). In the latter case, an adaptive, parallel design is a must for any operating system.

## 2. DESIGN PRINCIPLES
This section introduces primary concepts of the operating system, discusses the basic design principles behind the chansys architecture, and explains solutions to key operating system design issues and functional blocks implementation problems.

### 2.1 Resource-Owner-Service Model
Let us begin with an example of Figure 1. Suppose, we need to design a network-connected system comprising a receiver-transmitter, a local memory storage and capable of dynamic software update.

According to the ROS model, the system may look as follows: a first, privileged task may be responsible for network interface and packet sorting services, a second, user-level task may be responsible for data processing, a third packet-consuming task should be responsible for handling code update packets. The third task should establish some means of communication with the second task in order to verify the version and capability of the new code and to signal the old task to complete. A fourth task may be responsible for memory storage access. The third task may pass the received and verified code to the storage manager for the actual (physical) update of the binary image. The new code may then transparently take over interfaces originally serviced by the second task, in accordance with ROS principles described below.

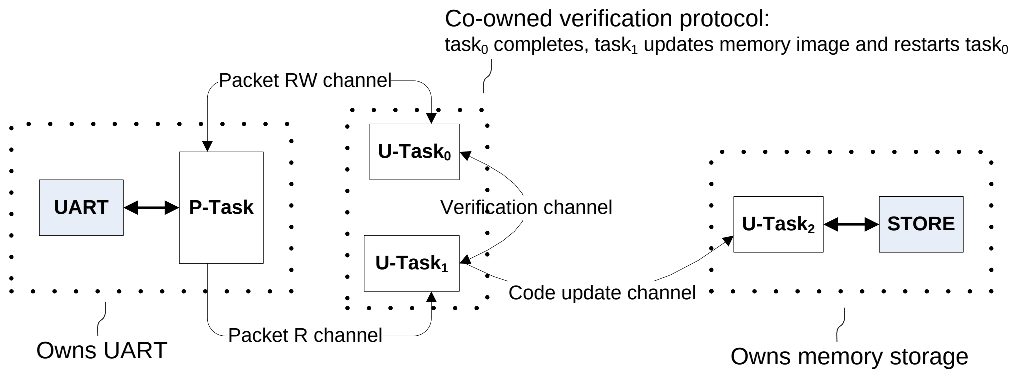


**Figure 1: An exemplary network-connected system**

The main properties of such design are: (a) all hardware (and software) resources (devices, storage space, or computational capabilities – that is, specific algorithmic solutions to certain problems) are assigned strict ownership, and (b) all resource owners communicate with each other by servicing each other's requests and providing access to the owned resources.

The following definitions may be useful. By owner we mean a resource-controlling agent (software program/task); service is a communication protocol between the owner and consumer agents.

Now we can name the basic principles of the ROS model: resource ownership – all computational resources (hardware devices or algorithmic parts) are viewed as resources and are divided between owners (software programs/tasks); service orientation – all inter-task communications are performed by means of requesting and providing services.

All the above resembles the traditional client-server design so far, except for the third principle – service instantiation. The service in ROS model is not a mere fact of communication or a data exchange interface, but rather becomes both a communication and synchronization entity.

That is what a channel is: a shared object established for data and control transfer between tasks. The channels make ROS different from the client-server programming model: there is no strict distinction between the communicating agents any more, as all of them instantiate their resources/services and may be mutual service providers in accordance with their communication protocol.

Yet another important property of the ROS model is communicational synchronization. Any control transfer between tasks (i.e., sharing execution time in a parallel or sequential (mutually exclusive) form) – something that cannot be done without operating system's assistance, by definition – can be performed only in connection with a communication session, in other words, while accepting/providing some service; and since any service is always backed up by a material object (channel), the channel becomes a synchronization object unambiguously identifying the place and purpose (i.e., algorithmic role) of synchronization.

The above statement does not deny the hardware-assisted mutual exclusion in multi-processor systems when a function running on one processor needs to synchronize its execution with the copies of the same function (or functions of the same channel) running on other processors.

## 2.2 Cooperative Preemption

In order not to waste computer resources (to reduce memory consumption for stack allocations, and to increase the performance by eliminating register saving instructions), each task can notify the chansys kernel that there is no need to preserve its context.

Instead, the task provides a four-pointer execution environment in the {func(chan, sys, loc)} form, so that the kernel can release the task's stack memory, and then, prior to the task's activation, allocate a new stack (or give the current free stack) for the task, and call the specified function with the provided parameters. Note that register values other than those used in parameter transfer may be either undefined or cleared, which is in any case faster than the correct value preservation/restoration. Also note that the system clears the four-pointer context before activating the task to make sure it cannot be erroneously preempted when not ready for it.

The cooperative preemption also means that if all tasks within the system adhere to the cooperative scheme, there may be no more than one stack per processor under ideal execution circumstances. Extra stacks will have to be allocated when a task is preempted prematurely, before it reaches its safe state and initializes the four-pointer context.

## 2.3 Yield-To Model

The chansys design, unlike traditional operating systems, does not provide explicit wait or mutual exclusion functions. Instead, all inter-task synchronization is accomplished in the form of explicit control transfers to other tasks identified by the channel pointer and the task's in-channel number. That is, the control is transferred not to some particular task with a certain task ID but rather to any agent that happened to provide a certain service, and happened to be indexed in a certain manner within its service channel.

This type of synchronization implies that it is the communicating agents' responsibility to establish such a communication protocol through their channel that will enable them to effectively transfer control to each other (or other dependent agents), to respond to mutual requests and exclude such inefficient means of synchronization as variable-polling.

The operating system may support multiple yield strategies, such as yield-to-any, yield-as-specified, and yield-as-specified-excluding-self. The latter strategy indicates that the "yielding" task does not intend to yield control at the current processor, so the new task should be activated in parallel with the current one. Note that yielding to a disconnected channel is ignored (the system call returns).

## 2.4 Deterministic Task Schedule

Each task (pretty much like in all modern operating systems) can be activated by a system timer, and since the chansys design should be applicable to real-time systems, it is decided that each task specifies its real-time requirements by means of the period of activation, and the duration of activation. Those two parameters are used to simplify the estimation of the anticipated system load, the number of expected task conflicts, and thus, to pre-allocate the necessary resources (e.g., stacks).

Ideally, the sum of durations of all tasks should be less or equal to the smallest activation period, and the periods should be multiples of the smallest one. That would mean there would be no scheduled conflicts. Nevertheless, in the reality, some tasks may exceed their requested durations, in which case the tasks could be preempted by other ready tasks and placed on the ready list to be executed in the future during a free time slot.

## 2.5 Performance-Driven Communication

As everything is resource/service oriented, and there may be several providers of the same service (on a network or even locally), the only criterion to differentiate between multiple providers is their performance.

To enable program performance management, the system channel contains active time and wait time counters providing information on the time of operation of the current task and the time of inactivity, respectively.

Since it is usually known which task (channel) the control was transferred to, the channel performance may be easily estimated by sampling the wait time counter value each time the control is transferred back to the requesting task. By comparing performance of multiple service channels the task may hop to faster service providers, which is especially convenient when trying to adapt to dynamically changing communication conditions (vast networks, physically unstable environments, etc.).

## 2.6 Channel Naming Convention

The chansys design does not make any assumptions with regard to the type or purpose of each channel. So it is the sole responsibility of communicating agents to choose such channel identifiers that will enable correct differentiation of channel type, purpose, group, or whatever semantics may be required by the nature of communication.

## 2.7 Channel Topology

Every channel has a two-ended topology. The chansys design introduces two classes of channels: dual-channels, and multi-channels. The former are those channels whose number of connected tasks cannot exceed 2, while the latter class of channels have no limitation with regard to the number of connected tasks (limited by particular operating system implementations and the memory size).

Tasks connected to the opposite ends of a channel are called channel exporter and channel importer. There is not much substantial difference between these two types of tasks, both exporting and importing tasks should instantiate the channel, that is, allocate memory, and thus have a complete channel object present in their address spaces. The explicit differentiation between those two types of tasks is introduced to let tasks with conventional consumer (client) semantics connect to conventional producer (server) semantics, in case multiple dual channels of the same ID are used in the system (not to let one channel to be connected to by two consumers, what may lead to deadlocks if the channel usage semantics is not flexible).

In case of multi-channels, those semantics have no meaning as all tasks will operate on a common channel object, and it is their responsibility to disambiguate their mutual accesses to the channel. Note that every channel, independent of the channel topology, may contain any combination of code and data.

## 2.8 Simplified Code Structure

Because of the channel-oriented design, and since the channels are dynamic data structures (are shared between address spaces, and thus can be moved from their initial location by the operating system kernel), the in-channel program code (and the task program code, in general) should be written in a position-independent manner and should not rely on statically allocated data (or should locate such data relatively to the accessing code).

This lack of statically allocated structures, given the general four-pointer state specification plus position-independence, denies the need in complex program headers and determines the simplest possible program structure: every program is represented by a solid chunk of code and data (with optional differentiation between the two to enable access protection). The beginning of the chunk is the task start function that accepts the standard three-pointer parameter set.

## 3. CORE FUNCTIONALITY

This section provides an overview of the internal system design, describes the structure and interaction of the functional blocks, and emphasizes the operation of some of the blocks where necessary.

## 3.1 Core Functional Diagram

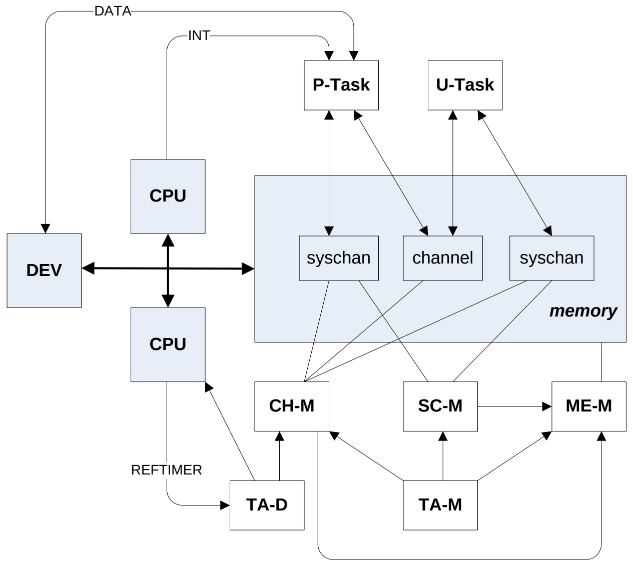


**Figure 2: CHANSYS functional diagram**

According to the above diagram, the chansys operating system implements the following functional blocks: (a) memory manager (ME-M), (b) channel manager (CH-M), (c) system channel manager (SC-M), (d) task manager (TA-M), and (e) task dispatcher (TA-D).

The operating system kernel owns processor and memory resources. The rest of system resources may be owned by privileged tasks, which, in their turn, provide service to user-level tasks by means of communication channels. The kernel identifies tasks and services task requests by means of system channels. The channels comprise shared memory locations, certain interconnection topology, and associated task scheduling services.

## 3.2 Task Creation

A task can be created from any piece of code. The address of the beginning of the code is treated as the task start function address. The task start function adheres to the four-pointer prototype {func(chan, sys, loc)}, wherein the chan and loc arguments are optional and sys is a pointer to the newly created task's system channel. Note that the task code is copied to a new physical location to simplify further channel operations (to avoid sharing conflict when exporting and then disconnecting the code as part of a channel).

The tasks are organized as illustrated in the figure below.

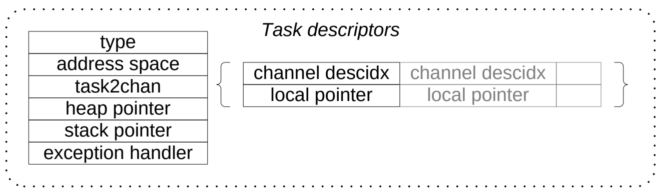


**Figure 3: Task descriptor table**

Each task is associated with a task descriptor comprising information on the type of task (e.g., user-level or privileged) and providing pointers to the task's address space page tables, local memory, stack, and exception handler chain.

Additionally, the task descriptor comprises a channel mapping list, which is a list of channel descriptor indices associated with channel body pointers to locate the channels within the task's address space. In case the tasks share a common address space, the channel pointers may be omitted for the sake of memory usage optimization.

The first element of the channel mapping list describes the task's system channel. The system channel stores the task's execution context and other properties as will be described further in this section.

## 3.3 Task Dispatching

The chansys design provides for two types of task activation: (a) by the system timer according to a requested schedule; and (b) by explicit control transfer from another task.

The activation schedule may be requested in terms of activation period and activation duration. Those two parameters determine the following task state diagram.

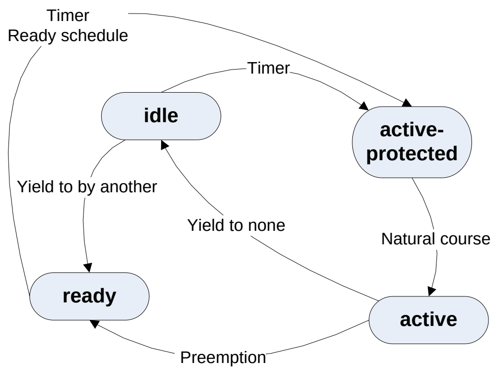


**Figure 4: Task state change graph**

Initially, when a task is activated, it acquires a protected state (cannot be preempted). After the task operates longer than its specified duration, the task becomes eligible for preemption. In case another task is on the list of activation (ready list), the system may preempt the first task, place its descriptor on the ready list and activate the second task. The newly activated task resumes execution in either active (in case it was previously preempted) or protected state (in case of scheduled execution). Particular implementations of the operating system task dispatchers may always resume the tasks in the protected state.

It is important to note here that the period-oriented scheduling specification enables the system kernel to guarantee a certain frequency of activation to the real-time tasks, rather than a certain activation deadline. Thus, the system is allowed to "shift" tasks in time as long as it does not break the activation frequency. The specification of the duration enables to avoid complicated task priority issues by defining the pre-scheduled (pre-requested) on-off time ratio. Particular system implementations may limit the duration to the maximal length of time quantum to prevent "greedy" tasks from occupying the processor infinitely. All the above simplifies the scheduler design and hopefully provides for faster task scheduling.

Explicit control transfers may be performed by means of `yield()` system call by specifying new task identifiers (channel pointer/in-channel index) and the processors, on which to activate the tasks. The operating system kernel may also support control transfer strategies to allow tasks to yield execution to any unspecified ready task or to run other tasks in parallel, without actually yielding the control.

In order to optimally utilize system resources, each task may signal to the system kernel that it reached a steady state and does not require preservation of registers and stack data (cooperative preemption). The operating system in this case may free the task's stack and discard registers upon `yield()` system call. When activating such a task, the system calls the specified four-pointer prototype `{func(chan, sys, loc)}` function at an empty stack and undefined register context.

Along with the aforementioned cooperative preemption, the chansys design implies resource-efficient task scheduling scheme. Since task schedules are specified as activation period and duration, the operating system task dispatcher may detect inevitable scheduling conflicts (see the diagram below) and either deny the scheduling request or relax the real-time scheduling constraints.

Each scheduling conflict makes the operating system kernel allocate extra resources (a new stack for the activating task) in order to enable fully-preemptive multitasking. Similarly, when a dynamic scheduling conflict is detected (because a task exceeded its requested duration), the system will have to allocate extra stack memory and preempt the guilty task, not to break the real-time schedule.

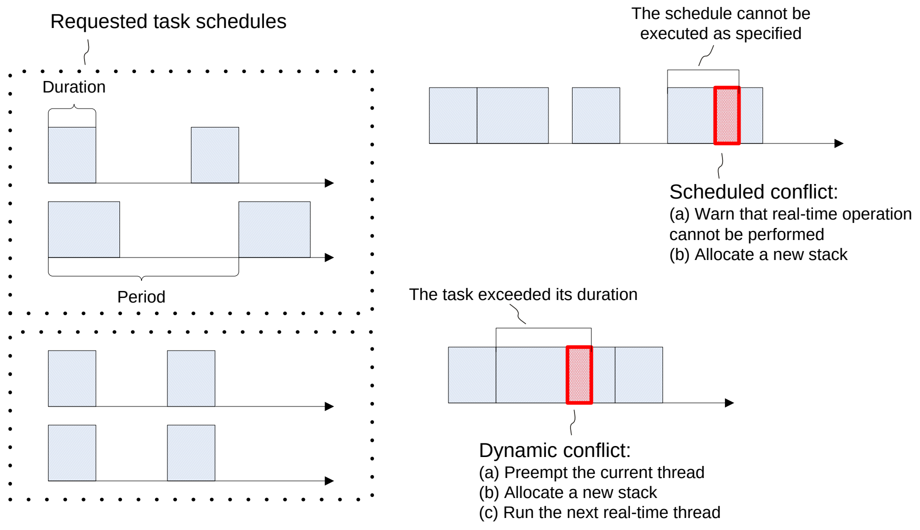

Requested task schedules
Duration
Period
The schedule cannot be executed as specified
Scheduled conflict:
(a) Warn that real-time operation cannot be performed
(b) Allocate a new stack
The task exceeded its duration
Dynamic conflict:
(a) Preempt the current thread
(b) Allocate a new stack
(c) Run the next real-time thread

**Figure 5: Requested task scheduling properties and detectable task scheduling conflicts**

To enable scheduled and preemptive task activation, the task dispatcher maintains a ready task list as shown in the figure below. The circular ready list comprises task descriptor indices (or pointers) and scheduled activation time (or zero for preempted tasks).

The time may be in absolute or relative units, up to the largest period, and is updated in accordance with the requested activation periods. When a task is preempted (or yields control), its descriptor is inserted into the first available slot on the circular list. Note that a task may be inserted in the list before other tasks, in case its activation deadline comes earlier, hence the ready list is time-ordered.

Ready tasks can be picked up from the list starting from the ready list's tail pointer or from task list pointers provided in processor control blocks (described further in this section).

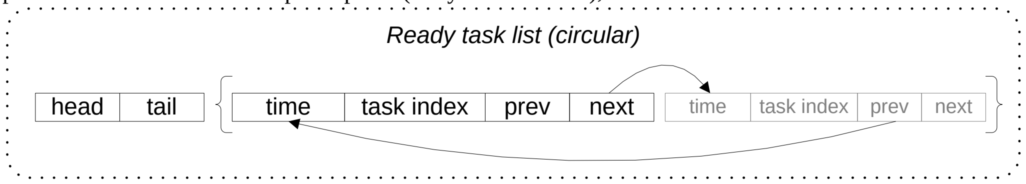


**Figure 6: Ready task list maintained by the task dispatcher**

The task dispatcher overhead can be theoretically decomposed into the static and dynamic parts. The former is a constant time of processor context initialization (note that the chansys design provides for faster context initialization due to cooperative preemption), while the latter is the overhead of the task dispatcher itself, proportional to the number of search operations over the ready list (both to find a current ready task and to insert a new ready task). There is no need to search for a task to be activated, since it is always located at the tail pointer. The task insertion overhead can be minimal in case the tasks have requested equal activation periods. Otherwise, the search can be performed in a logarithmic time, since the ready list comprises a set of contiguous chunks of mostly sorted data, and the unsorted elements can be reached by the chained links.

## 3.4 Channel Management

To manage channel operations, the system organizes allocated channels as shown in the figure below. According to the figure, each channel is associated with a channel descriptor. The descriptor stores information on the channel topology, identifier, and provides pointers to the channel body (in the common address space where all channels are allocated) and to descriptors of all tasks that are connected to the channel.

The backward channel-to-task links are necessary for task identification by means of channel pointer and in-channel index (for inter-task control passing). The in-channel index in this scheme is the position of the task descriptor on the task descriptor list (`chan2task`).

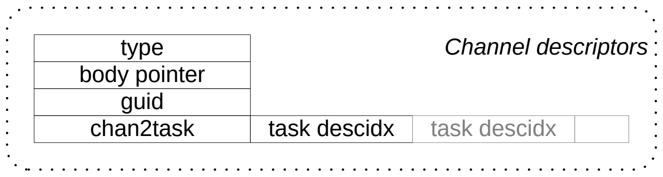


**Figure 7: Channel descriptor table**

## 3.5 Memory Management

Memory management schemes are not explicitly specified by the chansys design. Any traditional scheme will do, as there is no dependency of the kernel on a particular memory handling method.

Thus, Memory Control Block based allocation schemes may be applicable to shared address space environments; various Page Table based algorithms should be used for address space separation/virtualization (including linearly non-fragmented scheme for the cases when the amount of available physical memory is much less than the linear space).

No matter which memory control algorithm is chosen, the operating system kernel has to ensure all channels are allocated in a common address space in a manner that enables unambiguous mapping of the channels to the address spaces of their connected tasks.

## 3.6 System Calls and Context Management

System channels are complex means of task identification, task context preservation, getting system information, emitting system calls, and passing control to other tasks.

Figure 8 below sketches the structure of the system channel. Thus, arguments for syscall (break, interrupt, sysenter) instruction are copied to permanent places within the channel, rather than being transferred on the stack or in registers.

System information (comprising any information on the computer system properties and capabilities) is present in the system channel and hence is readily available without extra system call overhead. The combination of the four-pointer execution environment and the execution context in the same system channel may be convenient for remote task debugging/monitoring purposes.

Per-task timing information is updated by the kernel in real time. Per-processor fields enable parallel task activation (the tasks are identified by their channel pointers and in-channel indices).

The chansys system implementations may dedicate one register (available for reading to not privileged tasks) to serve as a permanent pointer to the current task system channel (a segment register, for example, given hardware support), so that the tasks are not required to preserve the system channel pointer and may reference it via a macro definition instead.

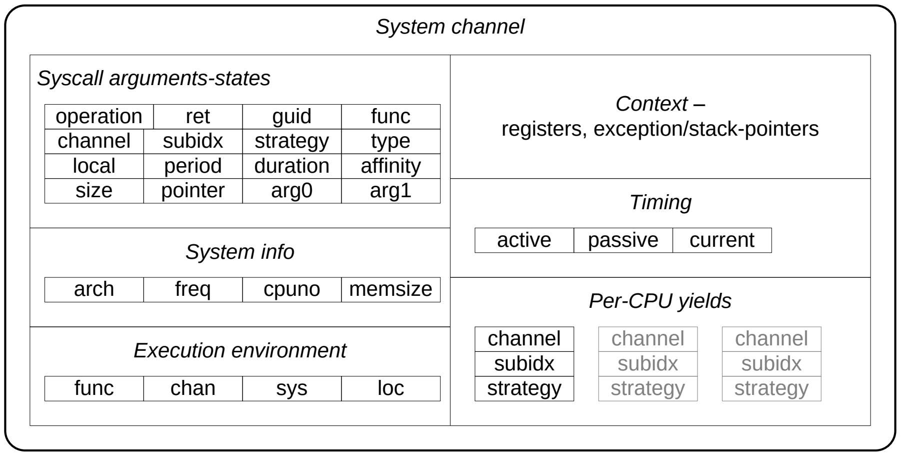


**Figure 8: System channel: a combination of task context, system information, performance data, per-processor task scheduling configuration, and system call parameters**

At least the following system calls should be supported:

**(1) `int exec(func, size, type, arg0, arg1);`**
- creates a new task from the specified function of the specified size; returns status{ok, err};

**(2) `int yield(chan, subidx, strategy);`**
**`int yield();`** for per-processor specification;
- passes control to the task connected to the specified channel at the specified index; returns status{ok, err};

**(3) `void exit();`**
- terminates the calling task;

**(4) `void* malloc(size, type);`**

- allocates a memory buffer of the specified size and type; returns a pointer to the allocated buffer;

**(5) int free(pointer);**

- releases memory of the specified buffer; returns status{ok, err};

**(6) void* export(pointer, guid, type);**

- exports a channel of the specified ID and topology (type) at the specified address; returns a new address of the channel;

**(7) void* import(pointer, guid, type);**

- imports a channel of the specified ID and topology (type) at the specified address; returns a new address of the channel;

**(8) int disconnect(pointer);**

- disconnects from the specified channel; returns status{ok, err, channel_destroyed};

**(9) int self(pointer);**

- returns the in-channel index of the specified channel.

## 3.7 Processor Management

Each processor within the system is assigned a Processor Control Block which contains some properties of the currently executed task (see the diagram below). Other processor control tables (interrupt and system descriptor tables), including inter-processor interrupt support structures, are not listed here.

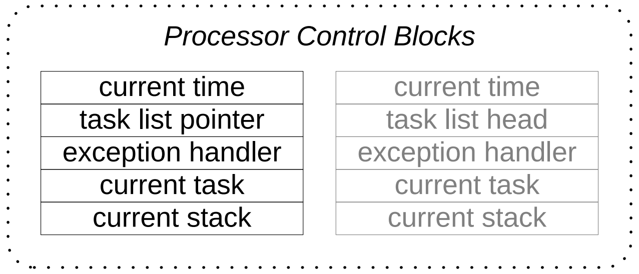


**Figure 9: Processor control structures providing task execution environment**

## 3.8 Debugging Support

Debugging support is not covered in detail in this article. The chansys design provides for variations in the debugging area.

The debugging capabilities may be provided by either the kernel itself or by a dedicated privileged task by means of establishing a debug channel to place commands and receive data. The commands may comprise requests for read/write access to a task's memory, including channels and the system channel. The debugger task may also query memory region status of the debuggee and receive notifications on system calls, debug exceptions, etc. Note that access to the debuggee's system channel means access to its register context.

A task may debug any other task it creates; a privileged (or even an ordinary, if the system is so configured) task may debug any other task whose channels it connects; a privileged task (having access to internal operating system resources) may debug any task within the system.

## 4. CORE COMPLEXITY LEVELS

The complexity of particular chansys implementations may vary depending on available memory resources, processor capabilities and performance. The following subsections describe the options for adjusting the complexity of the operating system kernel.

### 4.1 Channel Management Complexity

Channels being the central part of the chansys design provide a good deal of variation. Thus, particular systems may not support all of the proposed channel types (topologies) and implement different schemes of channel identification.

For example, the channels may be identified using globally unique 128-bit identifiers, or by assigned certain indices whose validity may be controlled by some sort of local (system-wide, network-wide) or global (inter-network, inter-organizational) arbitration authority. Another option may be named channels, whose names are constructed in the form of hierarchical paths (similar to file system paths).

As to different channel types (topologies), the system developers may choose to support dual-channels only, which will eliminate the need for dynamic allocation of channel member lists. In addition, the system may lack support for code channels (or at least mixed code and data channels), thus enabling the use of traditional compilers and development environments (as there will be no need for aggregation of code and data).

### 4.2 Memory Management Complexity

The chansys design may be equally implemented using shared or separated address spaces, whichever fits particular system requirements better.

Similarly, and independently from the chosen memory virtualization scheme, particular chansys implementations may or may not support memory protection, overlaying, or swap-out.

### 4.3 Task Management Complexity

As in some state-of-the-art operating system, the chansys design provides an option of implementing tasks as either processes or threads, that is, the tasks are either supposed to be isolated (logically and physically, if processor hardware permits), or share the same address space and thus are "encouraged" to pass data via shared variables.

Another task differentiation criterion is a task privilege level, that is, the availability of certain system and processor resources (e.g., inter-processor interrupts, processor descriptor tables, input-output ports) to the task. It is possible to implement privileged tasks only, and in that manner save efforts otherwise spent on task isolation, memory and processor resource protection, and so forth.

In case both privileged and non-privileged tasks are supported, the chansys design provides the following options for implementing privileged operations.

Firstly, all privileged operations may be performed by the kernel only; in other words, all privileged programs become part of the system kernel and execute in the kernel's (or requestor's) context (joint scheme).

On the positive side, servicing privileged operations may not require switching task contexts (the operations are performed in the requestor's context) in case the system kernel's address space is mapped to each task's address space. Besides, since the system kernel is always threaded by the number of processors, parallel tasks requesting privileged operations will not be stalled even if it is not possible to execute in the context of the requesting task.

On the negative side, the requesting task will not be able to continue execution until the request is processed.

Secondly, all privileged tasks may be separated from the kernel (separate scheme). This solves the problem of parallel execution of the requesting and servicing tasks, but introduces an extra context switch upon each request and upon each interrupt reception (as interrupts owned by privileged tasks may occur during execution of any other task).

A combination of the joint and separate schemes of privileged operations may be a beneficial solution. By default, all privileged operations may be performed on behalf of the requesting tasks (as part of the kernel), and, if needed, may create other privileged tasks that run separately and transfer control and data by general means of channeled communications.

The following extension to the above described core functionality may be required: an extra system call to connect to an interrupt vector, and a channel service function to be called by the system kernel when the control is transferred to a privileged channel whose owner is part of the system kernel rather than a separate task.

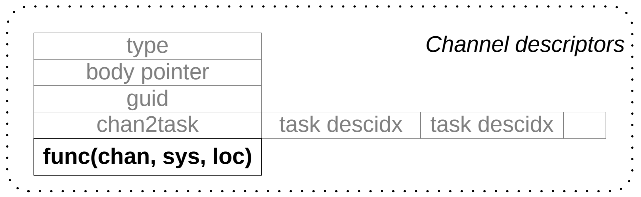


**Figure 10: Four-pointer service function associated with channels serviced without a task switch**

```
(10) int connect(int vector, void*
    handler);
```
- the system checks the requestor privileges, and the current task index; it is the system kernel's responsibility to ensure the handler's context is switched to the privileged requestor task upon reception of each interrupt at the specified vector; zero `handler` parameter removes the previously installed interrupt handler; returns status{ok, err}

## 5. CHANNELED COMMUNICATION EXAMPLES

This section furnishes several communication models which illustrate the use of dual- and multi- channel topologies, as well as advanced methods of task synchronization, request servicing and data transfer.

### 5.1 Dual Channels

The simplest task communication models (but not the least efficient) are dual-channels. Such channels are guaranteed to serve at most two communicating agents, though each of the agents may connect to more than one channel of the same ID.

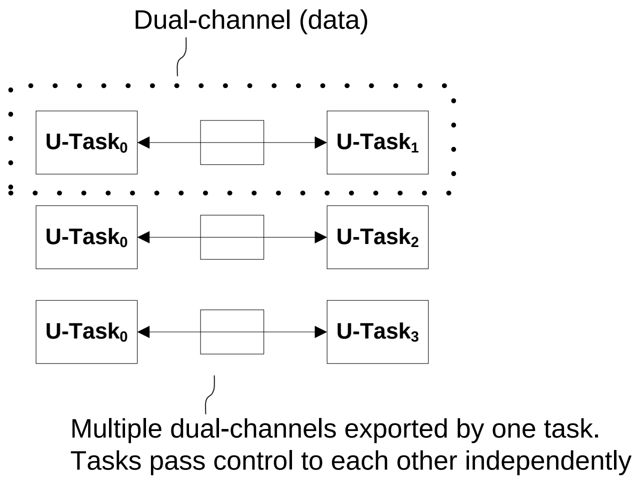

Dual-channel (data)
U-Task$_0$
U-Task$_1$
U-Task$_0$
U-Task$_2$
U-Task$_0$
U-Task$_3$
Multiple dual-channels exported by one task.
Tasks pass control to each other independently

**Figure 11: Communication through dual data channels**

Dual data channels are efficient data transfer means, wherein the connected agents copy their data to the shared in-channel location and notify each other on the completion of the data copying/processing operation (so that the counterpart doesn't waste processor resources).

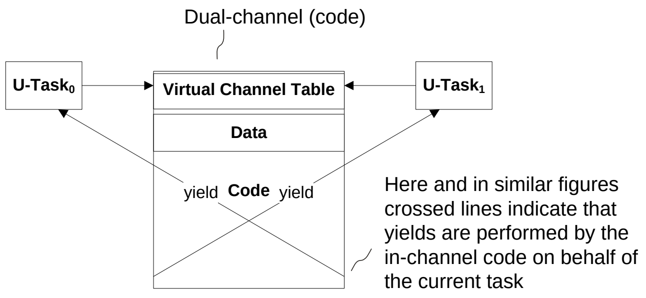

Dual-channel (code)
U-Task$_0$
Virtual Channel Table
U-Task$_1$
Data
yield
Code
yield
Here and in similar figures crossed lines indicate that yields are performed by the in-channel code on behalf of the current task

**Figure 12: Communication through dual code channels**

Dual code channels are almost equivalent to the data channels, the only difference being they provide a shared interface that encapsulates data transfer and processing operations, which may be more convenient in some cases of object-oriented design.

### 5.2 Multi-Channels

Multi-channels may be employed for providing computational service: that is, sharing code between multiple tasks (similar to dynamic load libraries in many traditional operating systems).

The shared code does not necessarily need to be backed by its exporter task.

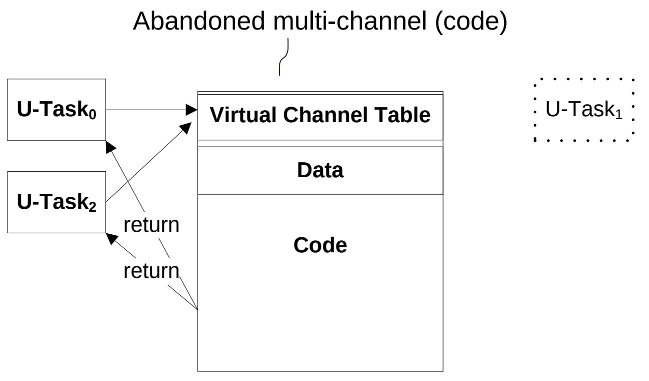


**Figure 13: Abandoned multi code channel analogous to a dynamic load library**

Indeed, the exporter task may initially export the channel and terminate, and the in-channel code, when imported, will allocate local data in the address space of the importing task, and thus will become fully integrated with the importer and, at the same time, isolated from other importers (in case the particular system implementation supports address space isolation).

More advanced multi-channel configurations are described in the subsections below.

## 5.3 Task Pools

Task pools are intended for supporting SMP-like synchronization models. The data channel connected to by multiple tasks may both contain the shared data being processed in parallel, and serve as a means of synchronization by holding the number of participating tasks and, optionally, their channel-related indices.

The idea behind task pools is to employ operating system's task scheduling capabilities and establish efficient synchronization of parallel tasks (as opposed to the most primitive channel-polling synchronization scheme).

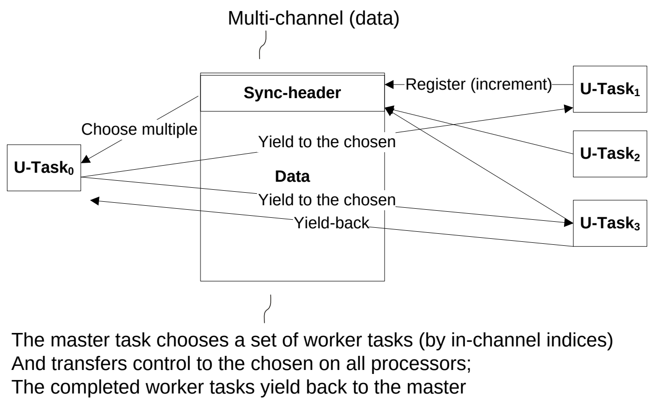

Multi-channel (data)
Sync-header
Register (increment)
U-Task1
Choose multiple
U-Task2
U-Task0
Yield to the chosen
Data
Yield to the chosen
Yield-back
U-Task3
The master task chooses a set of worker tasks (by in-channel indices)
And transfers control to the chosen on all processors;
The completed worker tasks yield back to the master

**Figure 14: Implementation of thread pools on multi data channels**

The exporter of the channel may assume work distributor's responsibility and, since the number of processors and the number of participating tasks are known, the exporter may yield execution to certain tasks at certain processors. When the tasks finish processing their data, they may return control to the distributor task.

## 5.4 Request Pools

The request pool model may be efficient when sharing large memory buffers or servicing a big number of clients, when establishing a separate channel per client would have wasted the memory space.

The pools provide arbitration interfaces, so that an arbitration winner can place the request (transfer data), and other requestors can be switched to an inactive state (in a transparent manner – by the in-channel code). When the servicing of the winner completes, the winner gets notified, copies its output data, and then renews the arbitration.

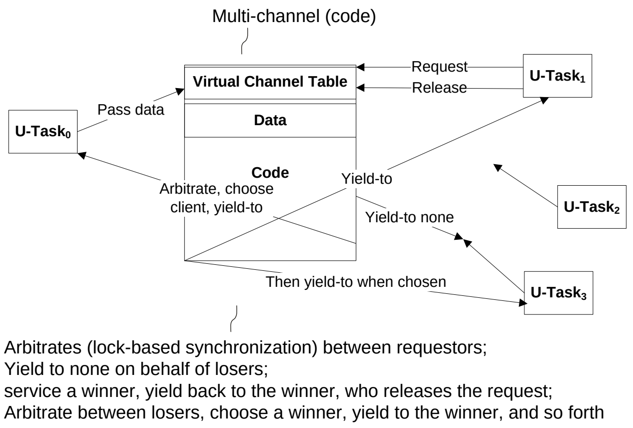

Multi-channel (code)
Virtual Channel Table
Request
Release
U-Task1
Pass data
Data
U-Task0
Code
Yield-to
Arbitrate, choose client, yield-to
Yield-to none
U-Task2
Then yield-to when chosen
U-Task3
Arbitrates (lock-based synchronization) between requestors;
Yield to none on behalf of losers;
service a winner, yield back to the winner, who releases the request;
Arbitrate between losers, choose a winner, yield to the winner, and so forth

**Figure 15: Request pool communication model**

Further on, a new winner is selected from the other requestors, is woken up by a yield-to operation, and the above procedure repeats.

## 5.5 Access Tokens

Access tokens enable a model of sharing data between (and/or receiving the same type/quality of servicing by) multiple tasks/agents without establishing the actual data transfer channels. Instead, a single (primary) service provider that manages the resources of/provides service to all the agents may be active in the system.

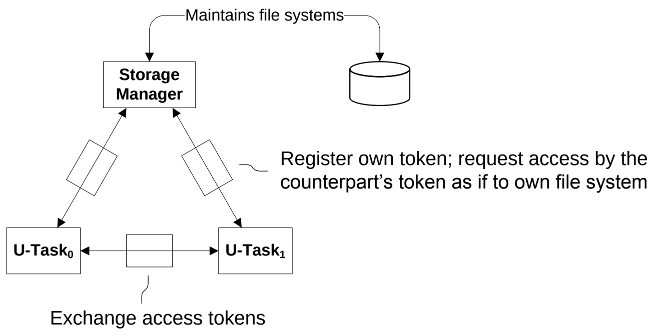

Maintains file systems
Storage Manager
Register own token; request access by the counterpart's token as if to own file system
U-Task0
U-Task1
Exchange access tokens

**Figure 16: Communication model based on access tokens**

Each agent may request an access token from the service provider, and the access token will become a key to the data of the requesting agent. Then, the agents may share the token with other trusted agents (via a special channel of exchange), and the trusted agents may receive the same type of service or access the same data by providing the shared token to the primary service provider.

## 5.6 Parallel Device Data Handling

In many cases, it is necessary to provide multiple tasks with simultaneous access to data streams produced (consumed) by a hardware device. The figure below illustrates two typical examples of parallel device communications.

The first one may be efficient if the system kernel differentiates between processes and threads. In this case, a dedicated task may handle all hardware device operations and buffer incoming/outgoing data streams, while the actual service to consumer tasks may be provided by threads, which have full access to internal buffers and thus may check for data availability and avoid unnecessary stalls.

The other example may be applicable to packet-wise data processing. Here, the hardware-interfacing task may allocate necessary buffers inside the channels it exports (since the size of a packet is known and limited, so is the size of the buffers). Both sides connected to the channels are supposed to indicate data availability to each other, so that the servicing task may process all connected channels upon reception of a single request, without regard to where the request came from, thus facilitating parallel operations.

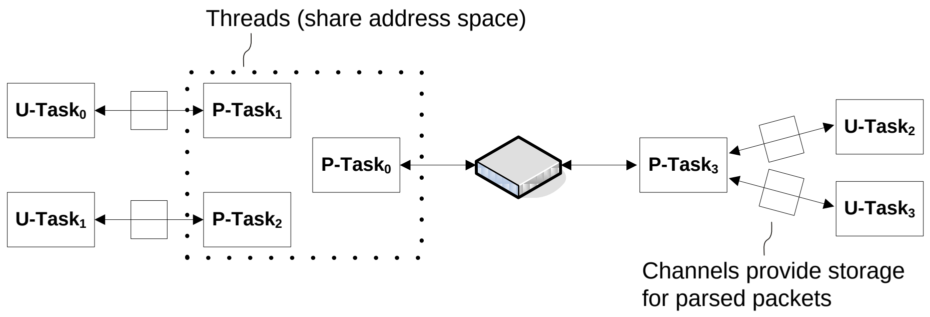
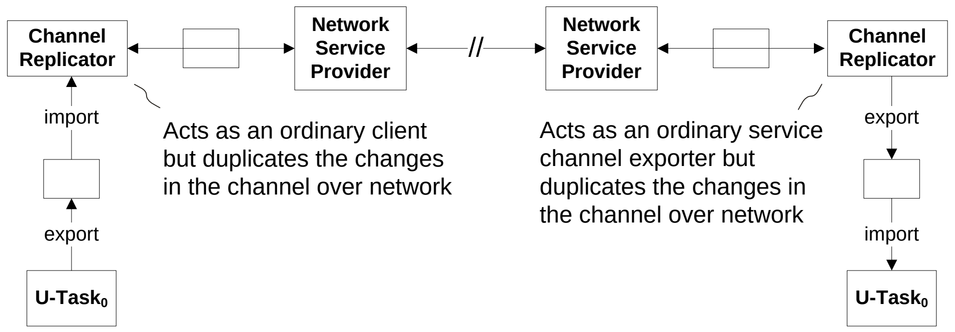


**Figure 17: Thread-based (left) and in-channel buffering (right) parallel communication models**

## 5.7 Remote Channeling

According to the location-independent nature of the ROS model, each agent in a local system may be connected to a channel exported remotely. For that purpose, so-called channel replicators may query their local systems for exported/imported channels, check on the network if those channels are requested, and maintain a pair (a set) of remotely synchronized channels in a manner transparent to their in-channel counterparts, as shown in the figure below.



Channel Replicator
Network Service Provider
Network Service Provider
Channel Replicator
import
export
U-Task$_0$
Acts as an ordinary client but duplicates the changes in the channel over network
Acts as an ordinary service channel exporter but duplicates the changes in the channel over network
export
import
U-Task$_0$

**Figure 18: Remote channeled communication**

Note that in order to efficiently support the remote channel replication, a new system call may be required:

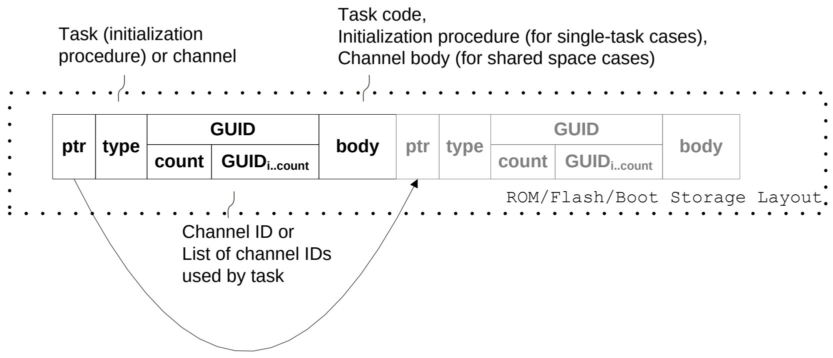
```
(11)   int query(guid[], size[], type[]);
```

- wherein `guid[]` is an array of channel IDs, `size[]` provides respective channel sizes, and `type[]` contains the channel request type/topology (imported, exported, or multi); returns the array length or error.

## 6. SYSTEM DESIGN EXAMPLES

This section furnishes several illustrational designs of resource-constrained micro-systems. The section also lists the system booting procedure and shows how a general, not necessarily resource constrained system may be organized from the perspective of the chansys architecture.

Note that all of those (micro-)system designs may be based on the following format of a boot media image.



Task (initialization procedure) or channel
Task code, Initialization procedure (for single-task cases), Channel body (for shared space cases)
ptr
type
GUID
count
GUIDi..count
body
ptr
type
GUID
count
GUIDi..count
body
ROM/Flash/Boot Storage Layout
Channel ID or List of channel IDs used by task

**Figure 19: Exemplary system boot image layout**

The general idea is to let the operating system detect the availability of hardware resources and load up support programs as needed. The system may query hardware device IDs (in case device identification is supported by hardware), match them against the list of channel IDs, and run the tasks that claim dependency on the selected channel IDs.

## 6.1 Single Task Systems

In case a particular chansys implementation does not support multitasking, the operating system kernel may run the pieces of code of the boot image as procedures within the context of the single task.

It becomes a programmer's responsibility to ensure there is a primary procedure that will communicate to the kernel and organize the inter-procedural communication through the channels.

## 6.2 Multi-Task Systems

Naturally, in multitask environments, each piece of code becomes a separate task, exports and imports channels as reflected in the boot image, communicates directly to the kernel and is responsible for setting up its activation schedule and synchronization scheme.

## 6.3 Shared Memory Space

In micro-systems with limited amounts of memory and non-virtualized address space, the boot image (in case the boot media is writable) may contain pre-initialized channel bodies in order to avoid extra memory allocations and minimize the system boot time.

## 6.4 Differentiation of Privileges

Optionally, if the processor supports memory protection, the system may differentiate between user-level tasks and privileged tasks, which are granted access to processor input-output resources and system memory. The level of task privilege is then provided in the boot image.

## 6.5 General Booting Procedure

The system boot procedure in accordance with the chansys design can be described as follows.

(a) The boot image may be copied from read-only to random-access memory, if necessary; (b) control is transferred to the main initialization procedure; (c) the initialization procedure creates (privileged) tasks (runs channel initialization procedures) for the channels whose identification matches that of available hardware; (d) the main initialization procedure also creates tasks for (initializes) the default channels, e.g., memory swap-out channel if the system supports memory virtualization. (e) The default channel tasks establish communication with initialized hardware service providers (e.g., hard drive channels). (f) After all default and hardware-managing tasks (and the corresponding channels) have been successfully initialized, the main initialization procedure loads the first non-privileged task (from either boot ROM or via the initialized channels), and that completes the system boot process.

## 6.6 Channeled System Organization

From the chansys architecture perspective, a typical personal computer system may be organized as shown in the figure below. A personal computer here serves merely illustrative purposes to let a reader quickly grasp the difference between traditional system designs and the chansys architecture employing ROS programming model.

The major difference introduced by the chansys design is that the raw data are never exposed to the end user. Instead, all operations are performed in a purpose-(or service-) oriented manner.

For example, the user may firstly select a desired operation (let it be insertion of a picture into a text document). Then, connect the selected operation with data sources. Note that there's no need to specify the location of the pictures and documents any more: the user is responsible for object identification, that is, to specify what an object is rather than where it is located, hence no files, folders, and strict hierarchical data organization.

The introduction of specialized data acquisition units enables more efficient search and indexing operations (including elements of artificial intelligence) performed over the stored data. And the presence of the topology configurator allows the user to change program communication pattern dynamically (and transparently to the communicating programs, as the channel communication semantics remain unchanged). The latter is especially convenient when abstracting remote communications.

The semantic storage manager (that organizes the incoming data in accordance with its meaning – as reported by data providers) logically complements the channeled system design.

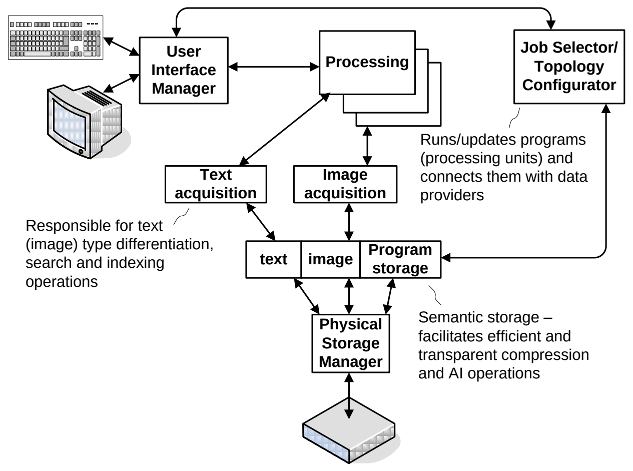

User Interface Manager
Processing
Job Selector/ Topology Configurator
Runs/updates programs (processing units) and connects them with data providers
Text acquisition
Image acquisition
Responsible for text (image) type differentiation, search and indexing operations
text
image
Program storage
Physical Storage Manager
Semantic storage – facilitates efficient and transparent compression and AI operations

**Figure 20: Typical personal computer system from the CHANSYS architecture perspective**

## 7. PROGRAMMING LANGUAGE SUPPORT

This section discusses possible changes to C/C++ language syntax and code generation requirements in order to efficiently support the ROS programming model.

### 7.1 Syntax Extension

Existing programming languages may need to be extended in order to support the Resource-Owner-Service programming model and facilitate program development in/for the chansys environment.

In case of C++ language, the extension that may affect the language semantics may be the `channel` type modifier to indicate the type contents should be aggregated into a form suitable for inter-task communication. The rest of communication channel-related problems may be solved using special support functions as illustrated in the figure below.

According to the figure, a channel type has first to be declared (using channel type modifier), and memory regions for both exporter and importer sides of the channel need to be allocated (with standard memory allocation operators). Then, the channel can be connected to by means of exporting the channel on one side and importing the channel on the other side. That can be accomplished by calling the provided `export()` and `import()` functions and specifying channel pointers, the system channel pointer, and the channel type (multi vs. dual channel). The pointer checks after the function calls are given here for shared address space environments, where the operating system kernel is likely to move the channel body to some memory location other than the initially allocated.

Each agent connected to a channel has its in-channel index (retrieved by `self()`), which remains the same unless the agent disconnects (calling `disconnect()` function) and then reconnects (by means of `import()` and `export()` functions) to the channel.

The rest of channel-related functions are recommended to accept a pointer to a channel in question, a pointer to the current system channel (task pointer), and a pointer to the local storage or an optional parameter. This allows prototyping of the majority of channel functions, such as the initial task function, the channel initialization function (which is supposed to allocate and return a pointer to the allocated local storage), and any other function that may be contained in a channel.

```
/// declarations
channel class A     /// a new type modifier
{
  int x;
  virtual void f();
  void g();
} *a, *b, *c, *d;

/// system channel – to make system calls
syschan_t* sys;
bool multi = false; /// or true for multi-channels

/// allocations
a = new A;          /// combines x, f, g, and vft
b = new A;          /// import space allocation

/// support functions
c = export(a, sys, multi);     /// export a channel of type A
if(c != a){ delete a; a = c; } /// the system moved the channel
d = import(b, sys, multi);     /// import a channel of type A
if(d != b){ delete b; b = d; } /// the system moved the channel

int i = self(a, sys);  /// get in-channel client's index
disconnect(a, sys);    /// disconnect the exported channel
disconnect(b, sys);    /// disconnect the imported channel
/// OS may free channel descriptors now
```

**Figure 21: Exemplary C++ language support for Resource-Owner-Service programming model**

```
/// in-channel functions are advised to have
/// the following prototype, wherein:
/// chanfunc may be:
///  task start function with (arg0, sys, arg1) semantics
///  channel initialization returning a pointer to local data,
///  or any service function contained in a channel
/// chan is the self-pointer to the channel
/// sys is the pointer to the system channel
/// loc is the pointer to local data or an optional parameter

void* chanfunc(chan, sys, loc);
```

**Figure 22: In-channel and channel-related function prototypes**

### 7.2 Code Generation

Care should be taken when writing programs for the chansys environment using compilers intended for other operating systems. It is a programmer's responsibility not to use the static storage class variables and operate only on automatic (stack) and dynamically allocated data. Ideally, a chansys-specific compiler should be able to detect static data references and generate memory access address computation relatively to the accessing function address.

Another problem that cannot be solved by traditional compilers (other than assembler) is the maintenance of combined code and data channels. The chansys-specific compiler should be able to aggregate channel function bodies and channel data in order to construct a solid memory object to be further mapped, moved, and processed by the operating system as appropriate.

For processor architectures that support data execution protection, the code and data aggregation should be performed on a different page basis not to compromise the system security.

## 8. CONCLUSION

Operating system is a key element that binds together various hardware and software components in a computer system, hence the efficiency of the operating system's design and the maturity of its programming models often play a crucial role in the success or failure of the entire product.

In this article we tried to address the challenges of modern parallel and distributed computer systems (focusing mainly on the needs of wireless sensor networks as the most extreme case of a heterogeneous system) by introducing a service-oriented approach to parallel program execution and communication.

Such service orientation infers new principles of control transfer, task scheduling, and resource management, as well as new logical topologies of program communication and synchronization, which, in their turn, provide for a very adaptive and scalable operating system design and unification of programming models for a wide range of hardware systems, from sensor motes to desktops.

Many of the disclosed operating system design principles were initially implemented by the authors of this article in 1999 as part of a light-weight operating system that served as a predictable and fully controllable testing environment for Windows NT executables.

Since then, we refined and generalized our operating system architecture and plan to complete its implementation for a custom wireless sensor network, both at the sensor and base station sides, and port the monitoring, control, and communications software to the new channel-based parallel programming schemes.

## 9. REFERENCES

[1] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel. O'Reilly, First Edition October 2000. ISBN: 0-596-00002-2

[2] Shucker at al. Embedded Operating Systems for Wireless Microsensor Nodes University of Colorado, 2005. www.knovel.com

[3] Lucero, S., Schatt, S. Wireless Sensor Networking (WSN) in Industrial Automation. ABI Research, 2007. abiresearch.com

[4] Zuberi et al. EMERALDS: a small-memory real-time microkernel. 17th ACM Symposium on Operating Systems Principles (SOSP '99), Published as Operating Systems Review, 34(5):277–291, Dec. 1999

[5] Shah Bhatti et al. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. ACMKluwer Mobile Networks & Applications (MONET) Journal, Special Issue on Wireless Sensor Networks, August 2005

[6] Rowe et al. FireFly: A Time Synchronized Real-Time Sensor Networking Platform. Department of Electrical and Computer Engineering Carnegie Mellon University. www.andrew.cmu.edu/~agr/pubpg/firefly-06.pdf

[7] PicOS. Olsonet Communications, 2006. http://www.olsonet.com

[8] Contiki 2.0 Reference Manual, 2007. http://www.sics.se/contiki/publications-and-documentation.html

[9] Han et al. A Dynamic Operating System for Sensor Nodes. University of California. http://nesl.ee.ucla.edu/projects/sos/

[10] Singularity Operating System. Microsoft Research, 2007. http://research.microsoft.com/os/singularity/

[11] ReactOS Operating System. http://www.reactos.org/en/index.html