

Message Passing. Speed, Strength and Simplicity for Embedded Systems in Communications Applications

This article discusses the differences between message passing and mailboxes as used in inter-process communication in a real-time operating system. The advantages of message passing as a modular, easily handled, debugged and distributed system are highlighted.

Many of the real-time operating systems (RTOSs) that claim to be well-suited to communications applications are based on the "mailbox" model of interprocess communication (IPC). However, for the multi-tasking, fault-tolerant and high-availability distributed systems of today's communications applications, a message-passing RTOS provides a simple and powerful alternative solution to problems that used to require complex solutions or expensive custom-built proprietary technology.

Message passing, specifically "direct message passing" is a style of IPC in which processes containing data or simply an identification tag send discrete messages directly to one another. Message passing developed in response to specific problems and considerations in distributed and parallel processing systems. However, because the commercial message-passing RTOSs of today provide high performance and can be easily debugged, programmed and maintained, they are equally valuable for both single CPU systems and distributed systems. Often systems grow from a single CPU design to systems which require multiple CPUs or subsystems. A message-passing RTOS allows such systems to be expanded and distributed to achieve optimal performance and functionality.

RTOSs simplify and clarify the construction of real-time systems by allowing the system to be divided into stand alone programs called processes, and permitting those processes to communicate with each other (enabling top-down designs). A part of the RTOS, called the scheduler, enables the execution of the most important process at a given instant in time. The RTOS defines the criteria that establishes which process is "most important." These criteria can be simple or extremely complex. In general, processes execute in response to interrupts, the passage of time, or interprocess communication.

INTERPROCESS COMMUNICATION

Interprocess communication occurs whenever one process influences another. The most typical applications of IPC are passing data, mutual exclusion, synchronization and sharing of resources.

The methods of IPC available to the application pro-

grammer vary from RTOS to RTOS. Conventional RTOSs provide the traditional methods of IPC, such as mail boxes, semaphores, event flags, pipes, etc. These methods of IPC can lead to complex systems that are difficult to program, debug and maintain. Message-passing RTOSs allow messages to be sent directly from one process to another. This type of RTOS is more simple to use, offers high performance and works within a clear design that can be easily debugged.

Message Passing (process-to-process)

OSE is a good example of a general purpose, commercial RTOS based on message passing. In this model, a process allocates messages from the RTOS dynamic memory manager, with a message's size limited only by the physical memory available in the system.



Figure 1. Direct message passing results in a clear design.

The allocated message is owned by the allocating process which uses memory as needed.

When the process is ready to send the message, it calls the RTOS, submits the message address and

specifies the destination process. The address of the message—not the data contained in the message—is placed in the queue of the destination process. The ownership of the message is given to the destination process when it receives the message.

When a process is ready to receive a message, it calls the RTOS and specifies which message or set of messages to receive. It also indicates how long it will wait for the message, be it a specified amount of time or a simple poll to the queue and return. When a message is available, the address of the message is returned and the ownership of the message is transferred to the receiving process, which can use or modify the message contents as desired. The message can then be sent to yet another process or returned to the dynamic memory manager.

Mailboxes (process-to-mailbox-to-process)

In a conventional RTOS, processes send messages to a mailbox where another process can retrieve the messages. The receiving process has no choice as to which message to receive from the mailbox, but rather receives the first message available in the queue. If a process does not want to receive all of its messages in the sequence in which they were sent from the same mailbox, another mailbox must be created where certain messages should be sent and received separately. For example, if a process wants to receive 50 different messages at different times in its processing, 50 mailboxes must be administered.

This method of IPC, while common, does not compare favorably to direct message passing. The most significant failure of mailboxes is that the communicating processes must define their own rules and methods of communication. This spreads the coordination, which must be handled by the application, over multiple processes. The process which creates the mailbox determines the name, size and attributes of the mailbox: how receiving tasks wait, whether the mailbox is local or global, and if it should handle fixed or variable length messages. The sending processes must determine the correct mailbox, whether the message should be put at the beginning or end of a queue, and whether the mailbox handles fixed or variable length messages. The receiving processes must wait at the correct mailbox and know if the mailbox has fixed or variable length messages. The processes also must know the format of the messages they receive and handle errors if the mailboxes are full, do not exist or have been deleted.

A process that requires a prompt response from another process must either know at which mailbox to wait or it must send messages to all possible mailboxes. In the latter case, the unused messages must be removed once the process has received one copy of the message. This adds overhead to the transfer.

In the mailbox model, a minimum of three processes, and most likely many more, must coordinate actions. Such a model is inconsistent with modular, process-oriented programming, which should be the designer's primary concern and the reason for using an RTOS.

Message passing avoids the encumbrances associated with mailboxes. The receiving process, the one which is actually implementing the algorithm, decides what is important according to its own requirements. This is consistent with top-down methodologies and critical when dividing up an application between multiple developers.

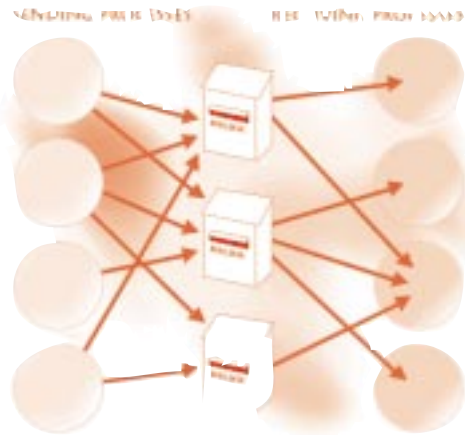


Figure 2. Mailbox communication is more complex than direct message passing.

Semaphores

Some RTOSs provide semaphores, which are often used to synchronize processes for a predetermined purpose. In a system where processes share a global memory area (shared memory), semaphores can be used to block one process from reading a memory area until another process has stored data in that area. This 'shared memory' method of IPC has been used extensively and continues to cause the same problems today as it did when it was first implemented:

- The semaphore must be created.
- The processes concerned must use the same semaphore.
- The designer must determine what happens if a semaphore is deleted or a process owning the semaphore is killed.
- Communicating processes must somehow agree on the structure of the data, data types and array sizes.

Finally, semaphores provide no queuing mechanism, which means that queuing must be handled by the application, adding code and overhead to the system, as well as time to market. Other synchronization methods, such as disabling interrupts or preemption, are similar to semaphores, but incompatible with each other.

In a large application, it can become very difficult to keep track of the different semaphores, data structures and processes involved in the system. When such an application grows into a fully distributed system, the semaphore technique is simply not feasible.

SIMPLIFIED PROGRAMMING

Today's real-time embedded communications systems are typically large and complex, employing thousands of different types and structures of data, and hundreds of processes. The RTOS should relieve the application of the responsibility for administering process-to-process transmission, allowing the application to concentrate on its primary functions. Not only should the RTOS provide IPC services, it also should integrate them into a single, easy-to-use method.

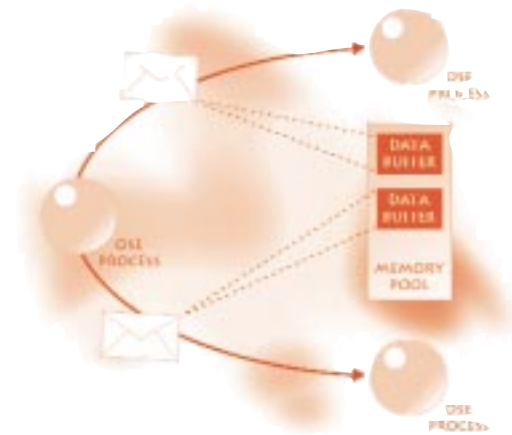


Figure 3. The message data is located in the memory pool. When sending a message to a process in the same address space, only the pointer to the buffer is moved to achieve high performance.

Because message-passing RTOSs work in this manner, they create a more effective programming platform for embedded communications applications: message-passing RTOSs

- Simplify programming by requiring only a few operating system calls
- Integrate IPC with memory allocation and usage
- Limit bug sources and integrate IPC with debug tools
- Address problems that are unique to distributed systems.

In a message-passing RTOS, the application is divided into logical blocks using processes. Each process receives input messages, performs algorithms on the messages and produces output messages. Data is passed directly from one process to another in the system. A process simply receives the messages it wants, processes them and sends the results onward as new messages. The input data needed by a process is always in the same place and can be received at any time. Results are sent directly to other processes, even if they are interrupt processes.

As only one optimized method of IPC is used, code becomes much more readable and maintainable. Fewer bugs are introduced because a few familiar system calls are repeatedly used. Maintenance is facilitated, as processes and code segments are much more likely to be reused in other projects because of the well-defined interfaces message passing provides.

Blocks-groups of processes that perform a specific function-provide an interface mechanism that allows changing the internal implementation of the block without affecting the interfaces to the rest of the system. This model allows parts of an application to be removed and replaced easily, adding flexibility, time-savings and cost-effectiveness in communications applications that require frequent maintenance.

There is no size limitation to queues and no limit to the number of queued messages. The only restriction is the amount of RAM available. In a multi-CPU application, messages are still sent directly from one process to another, and the RTOS handles the transfers, relieving the application of this task.

Memory Handling

When examining the transmission of data, it is imperative to consider how memory is handled. Dynamic memory (run-time) allocation is necessary, so that a memory area which is not being used by one process can be available to another. A system using static memory allocation requires significantly more memory than the same system using dynamic memory allocation, in which only the memory actively being used at a given point in time is needed with dynamic memory allocation.

In the OSE message-passing RTOS, the operating system relieves the application of memory management. Messages are allocated only when they are needed and freed when they are not, in contrast to the semaphore, which requires the two processes to agree on a memory area and how it should be used. OSE allows a process to use many different messages and send them to any other process, making for simpler code readability, maintenance and reuse.

Message passing is performed by reference, with only the address of the message passed, not the data itself. When data is stored in a message, it is never copied unless, as described in Figure 3, the destination process memory is located in another address space. Ownership of a message is gained by allocating a message, and ownership is transferred by sending the message to another process. Only the owner is allowed to read or write a message, send a message or free a message. This is the protection mechanism which all IPC methods must ensure in multitasking systems. Message passing relieves the application from the responsibility of ensuring that data is not changed by one process while being read by another. In the same address space only the pointer to the buffer is moved to achieve high performance.

If the destination process is located in another address space, OSE transparently transfers the message to the other address space as a new message (another address from another memory pool). After transfer, the original message is freed back to its memory pool and the destination process receives the new message. The application is not responsible for data transfer or memory handling. It only needs to specify a message buffer and a destination process. In such a system, incorporating a memory management unit (MMU) protects against memory corruption.

An effective RTOS should provide transfer by reference and should copy messages only when processes do not share or have access to the same address space. Most importantly, the RTOS should do this without requiring any participation from the application, which should send and receive messages in EXACTLY the same manner, regardless of the destination. This model allows processes to be moved freely from one CPU to another without change, which is critical for an effective fault-tolerant distributed system.

Debugging

In a message-passing RTOS, the passing of messages and scheduling of processes may require debugging. In following with the top-down design of a system, debugging needs to be easily performed on the interfaces of the logical blocks. An RTOS vendor should offer specialized debugging tools which focus only on these system events. These tools allow monitoring, tracing and stopping of the system based on the movement of messages and process scheduling. Additional tools will provide memory, CPU usage information and even enable system-level traces that time-tag system-level events.

In traditional RTOSs, system-level events may include mailboxes, semaphores, event flags, rendezvous and more. It is difficult to create effective RTOS-level debugging tools in these environments because each application, and even each developer, has a different mode of implementation. How can a semaphore-protected, shared memory area be traced by a debugger, when every developer handles the semaphores, addresses and memory areas differently? Most traditional RTOSs rely heavily on the use of source code debuggers which are not designed to debug multitasking applications and have no knowledge of RTOS-level events.

Distributed systems

Message-based RTOSs were developed for distributed systems and are ideally suited for communications systems. A communications application based on a distributed system should not know about or administer system distribution. Messages should still be passed directly from process-to-process. The communication links between the nodes of the system should be implemented on the RTOS level, independent of the application. In the OSE RTOS, this is accomplished with the concept of Link Handlers (LH), which provide logical channels to processes located in remote systems. An LH receives messages destined for remote processes, and the contents of the messages are transferred across the link to another LH, which forwards the message to the destination processes.

CONCLUSION

The direct message-passing RTOS simplifies IPC while enhancing functionality and performance, allowing applications to be generated faster and with fewer bugs. With message passing's simple structure, applications are inexpensive to maintain and larger portions can be reused in new projects. In addition, an application can grow to be distributed without requiring code changes, which is particularly important in the telecommunications industry where hardware lifecy-

cles can exceed 25 years.

Traditional IPC methods like mailboxes can be compared to the command-line interface, which was useful in the past because it was all that existed. Today, it exists solely because of its ties with history and old software that is still used. However, few people would choose to purchase a system with a command-line interface, with the current availability of the graphical user interface (GUI).

Traditional IPC methods are the command-line inter-

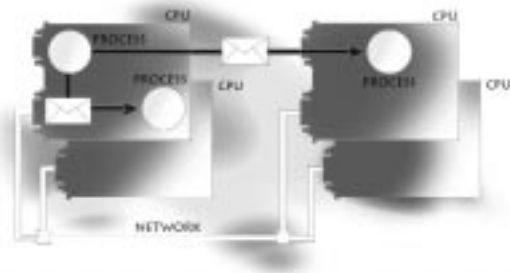


Figure 4. An OSE message may be sent transparently to a remote process.

faces of the RTOS industry. While they are familiar and comfortable to many designers, they are no longer able to handle the size and complexity of emerging applications. The direct message-passing RTOS will take real-time applications to the next level just as the GUI has for the user interface ■

Bill McCombie graduated from San Diego State University with a B.S.E.E in 1984. He has worked with real-time, primarily embedded, systems for 16 years. In the last 12 years, Bill has worked with RTOSs, the latest of which represents the state of the art in distributed and fault-tolerant systems. Bill is currently Technical Marketing Manager at Enea OSE Systems.

In the following examples, code was written for OSE and for pSOS for benchmarking purposes. Two tasks are created by a shell task, and then the shell task sends a message to start the application via task A. In a loop, A sends a message to B and then waits for a response. When task B gets this message from A, it sends a response back to A. When the loop is complete, the shell task deletes both A and B.

Example of code written for pSOS, using the conventional IPC method:

Definitions are excluded. Code won't compile!

Upon reaching any error condition, the intent here is to solely report it and not to recover, as this code was written for benchmarking purposes

```

void
cmd_ipcperf(int argc, char *argv[], char *env[])
{
    t_create("IPCA", 210, 4096, 512, 0, &tid_a); /* create tasks A & B */
    t_create("IPCB", 210, 4096, 512, 0, &tid_b); /* next create queues */
    rc = q_vcreate("A_Q", Q_GLOBAL | Q_FIFO, 10, MAX_MSG_LENGTH, &qid_a);
    if (rc != NOERR) k_fatal(0, 0);
    rc = q_vcreate("B_Q", Q_GLOBAL | Q_FIFO, 10, MAX_MSG_LENGTH, &qid_b);
    if (rc != NOERR) k_fatal(0, 0);
    rc = q_vcreate("SH_Q", Q_GLOBAL | Q_FIFO, 10, MAX_MSG_LENGTH, &qid_sh);
    if (rc != NOERR) k_fatal(0, 0);
    rc = pt_create("PTNM", /* create memory partition */
                 (void *) (((unsigned long) pbufm + 4) & 0xFFFFFFFF),
                 (void *) 0, 2048, 64, PT_NODEL, &ptid_msg, &nbuf_m);
    if (rc != NOERR) k_fatal(0, 0); /* next we start tasks A & B */
    t_start(tid_a, T_USER | T_PREEMPT | T_NOTSLICE, ipc_a, 0);
    t_start(tid_b, T_USER | T_PREEMPT | T_NOTSLICE, ipc_b, 0);
    ipc_init_a.msg_type = IPC_INIT_MT; /* build up start up message */
    ipc_init_a.num_msgs = atoi(argv[1]);
    ipc_init_a.size_msg = atoi(argv[2]); /* next we send start up message */
    rc = q_vsend(qid_a, (unsigned long *) &ipc_init_a, sizeof(ipc_init_a));
    if (rc != NOERR) printf("Error sending init msg: 0x%x\n", rc);
    rc = q_vreceive(qid_sh, Q_WAIT, 0, (unsigned long *) &sh_msg_buf[0],
                  MAX_MSG_LENGTH, &msg_length); /* here we wait for complete msg */
    if (rc != NOERR) printf("Error receiving done msg: 0x%x\n", rc);
    ipc_done = (IPC_DONE *) &sh_msg_buf[0];
    rc = pt_delete(ptid_msg); /* start deleting system resources for cleanup */
    if (rc != NOERR) printf("Error deleting partion: 0x%x\n", rc);
    rc = t_delete(tid_a);
    if (rc != NOERR) printf("Error deleting task a: 0x%x\n", rc);
    rc = t_delete(tid_b);
    if (rc != NOERR) printf("Error deleting task b: 0x%x\n", rc);
    rc = q_vdelete(qid_a);
    rc = q_vdelete(qid_b);
    rc = q_vdelete(qid_sh);

    return;
}

void
ipc_a(void)
{
    for(;;)
    {
        fclose(0); /* these close/free calls are done so we can delete this task later */
        close_f(0);
        close(0);
        free((void *) -1);
        rc = q_vident("A_Q", 0, &qid_a); /* prepare to wait on a queue to get started */
        if (rc != NOERR) printf("Error indentifying queue A: 0x%x\n", rc);
        rc = q_vreceive(qid_a, Q_WAIT, 0,
                      (unsigned long *) &ipc_a_msg_buf[0],
                      MAX_MSG_LENGTH, &msg_length);
        if (rc != NOERR) printf("Error receiving ipc_init_a msg: 0x%x\n", rc);
    }
}

```

SONY®

Do you want to develop the TV of the 21st century?

**Sony Digital Media Europe (DME)
is looking for you!**

We create architectures and develop software for interactive audio-visual services and participate in major projects for the introduction of interactive digital TV platforms (set-top boxes and digital TVs).

The software we develop ranges from device drivers, over Java virtual machines and middleware to end-user applications.

We want to work with people who want to write product quality software, and who are ready to stretch their professional skills to experience the thrill of making a Sony product.

Interested?

Give us a call. We expect a degree in computer science or an equivalent education and a background in C/C++ or Java in a UNIX or NT environment.

jobs-dme@sonycom.com or:

Keetje Tromp Meesters, Sony Service Center,
Digital Media Europe - Brussels, Sint-Stevens-Woluwestraat 55,
1130 Brussels. Tel.: 02/724.86.81 or fax: 02/724.86.88.

```

ipc_init_a = (IPC_INIT_A *) &ipc_a_msg_buf[0];
num_msgs = ipc_init_a->num_msgs; /* start getting parameters sent to us */
size_msg = ipc_init_a->size_msg;
rc = pt_ident("PTNM", 0, &ptid_msg); /* preparing to send data */
if (rc != NOERR) printf("Error indentifying msg partition: 0x%x\n", rc);
rc = q_vident("B_Q", 0, &qid_b);
if (rc != NOERR) printf("Error indentifying queue B: 0x%x\n", rc);
/* in loop that follows we send data to task B and wait for ACK back, then we return
buffer to partition
*/
for (ctr = 0; ctr < num_msgs; ctr++) {
    rc = pt_getbuf(ptid_msg, (void **) &b_msg);
    if (rc == NOERR){
        b_msg->msg_type = B_MSG_MT;
        rc = q_vsend(qid_a, (unsigned long *) b_msg,
                    size_msg);
        if (rc == NOERR){
            rc = q_vreceive(qid_a, Q_WAIT, 0,
                           (unsigned long *)
&ipc_a_msg_buf[0],
                           MAX_MSG_LENGTH,
&msg_length);
            if (rc != NOERR) printf("Error receiving B ack 0x%x\n", rc);
            b_ack = (B_ACK *) &ipc_a_msg_buf[0];
            rc = pt_retbuf(ptid_msg, b_msg);
            if (rc != NOERR) printf("Error returning b_msg to buffer\n");
        }
    }
} /* start preparing to send done message */
rc = q_vident("SH_Q", 0, &qid_sh);
if (rc != NOERR) printf("Error indentifying shell queue: 0x%x\n", rc);
ipc_done.msg_type = IPC_DONE_MT;
rc = q_vsend(qid_sh, (unsigned long *) &ipc_done,
            sizeof(ipc_done));
if (rc != NOERR) printf("Error sending done msg 0x%x\n", rc);
}
}

void
ipc_b(void)
{
    for(;;)
    {
        /* in loop that follows we receive a message and send back an ACK */
        rc = q_vident("B_Q", 0, &qid_b); /* getting ready to receive Tx */
        if (rc != NOERR) printf("Error indentifying queue B: 0x%x\n", rc);
        fclose(0); /* next 4 are done so we can delete this task later */
        close_f(0);
        close(0);
        free((void *) -1);
        rc = q_vreceive(qid_b, Q_WAIT, 0,
                      (unsigned long *) &ipc_b_msg_buf[0],
MAX_MSG_LENGTH, &msg_length);
        if (rc != NOERR) printf("Error receiving b_msg msg: 0x%x\n", rc);
        b_msg = (B_MSG *) &ipc_b_msg_buf[0];
        rc = pt_ident("PTNM", 0, &ptid_msg);
        if (rc != NOERR) printf("Error indentifying msg partition: 0x%x\n", rc);
        rc = pt_getbuf(ptid_msg, (void **) &b_ack);
        if (rc == NOERR){
            b_ack->msg_type = B_ACK_MT;
            rc = q_vident("A_Q", 0, &qid_a);
            if (rc != NOERR) printf("Error indentifying queue A: 0x%x\n", rc);

```

```

rc = q_vsend(qid_a, (unsigned long *) &b_ack, sizeof(B_ACK));
if (rc != NOERR) printf("Error sending B Ack msg 0x%x\n", rc);
rc = pt_retbuf(ptid_msg, b_ack);
    }
}
}

```

Example of code written for OSE, using direct message passing model:

Definitions are excluded. Code won't compile!

Upon reaching any error condition, the intent here is to solely report it and not to recover, as this code was written for benchmarking purposes.

```

int cmd_ipc_perf()
{
    /* create and start OSE processes */
    pid_a = create_process(OS_PRI_PROC, "ipc_a", ipc_a, 4096, 0,
                          0, 0, NULL, 0, 0);
    pid_b = create_process(OS_PRI_PROC, "ipc_b", ipc_b, 4096, 0,
                          0, 0, NULL, 0, 0);

    start(pid_a);
    start(pid_b); /* next we build message to be send to process A */
    ipc_init_a = (IPC_INIT_A *) alloc(sizeof(IPC_INIT_A, IPC_INIT));
    ipc_init_a->num_msgs = strtol(argv[1], NULL);
    ipc_init_a->size_msg = strtol(argv[2], NULL);
    /* in final section we send our start message, wait for completion and clean up resources */
    send((union SIGNAL **) &ipc_init_a, pid_a);
    signal = receive(&any_sig[0]);
    free_buf(&signal);
    kill_proc(pid_a);
    kill_proc(pid_b);
}

OS_PROCESS(ipc_a)
{
    for(;;) { /* initially we receive a start message
    ipc_init_a = (IPC_INIT_A *) receive(&select_init[0]);
    num_msgs = ipc_init_a->num_msgs;
    size_msg = ipc_init_a->size_msg;
    (void) hunt("ipc_b", 0, &pid_b, NULL); /* make sure destination process is alive */
    for (ctr = 0; ctr < num_msgs; ctr++) { /* in this loop we send a message to process B and receive ACK */
        b_msg = (B_MSG *) alloc(sizeof(B_MSG) + size_msg, B_MSG_MT);
        send((union SIGNAL **) &b_msg, pid_b);
        b_ack = (B_ACK *) receive(&any_signal[0]);
        free_buf((union SIGNAL **)&b_ack);
    }
}

OS_PROCESS(ipc_b)
{
    for(;;) { /* here we wait for msg from A and send back ACK */
    b_msg = (B_MSG *) receive(&select_b_msg[0]);
    b_ack = (B_ACK *) alloc(sizeof(B_ACK), 0);
    source_pid = sender((union SIGNAL **)&b_msg);
    send((union SIGNAL **)&b_ack, source_pid);
    free_buf((union SIGNAL **) &b_msg);
}
}

```