



RGX Real Time Operating Systems

Copyright 1990-2010 R. D. Godsey and Associates, Inc.

RGX Systems Overview

RGX is a minimalist, event driven, prioritized, multitasking operating system. It provides management of computer time allowing several programs or tasks to share computer resources. RGX contains a fundamental set of services to support inter-task and background to foreground communications via message/exchange(mailbox) objects. These allow inter-task data transfers, data and task queuing, and formation of other behaviors.

Writing RGX systems requires an understanding of this environment and the system architecture. An RGX system consists of:

Tasks:

Tasks are programs that run in foreground time and perform the functions of the system. RGX tasks are a complete program loop that provides a system function like managing a display, interfacing to the user or controlling a set of solenoids and run in foreground time. Tasks are usually comprised of a collection of procedures and modules that have the same *scope* in that they share the same stack and memory space and are only referenced by procedures and source modules in that scope. Aside from references to the OS or public data commons, each task is pretty much a standalone function.

Tasks are treated as a program loop and is required to wait at a mailbox exchange by calling `rgx_wait` at least once in a loop. Tasks communicate with each other by sending and receiving messages and through data commons.

Rules for RGX tasks:

Tasks are written as closed loops as opposed to as 'threads'.

All tasks must call the executive to wait at a mailbox for an event at least once in a loop.

Tasks should not turn off the RGX time base interrupt.

For RGX51, all tasks must use register bank 0. The other 3 register banks are reserved for ISRs. Every task has a descriptor that contains the current task status and properties.

ISRs:

A number of **I**nterrupt **S**ervice **R**outines or ISRs and supporting hardware. Short and to the point procedures for fast, asynchronous response to real time hardware events. These events cause hardware interrupt signals from devices such as counter/timers, ADCs, limit switches, etc. Since interrupts suspend execution of foreground code and force a subroutine call, ISRs run in background time. ISRs can send messages to mailboxes. Usually an ISR and a task work together to form a device driver.

A typical ISR would buffer incoming serial data until an end of string character is received, then send a message to a foreground task through RGX that a string has been received and contains the relative data. The foreground task and background ISR together form part of a serial device driver.

A Time Base ISR that calls `rgx_clocktick` is always necessary to provide the system time base.

Exchanges/Mailboxes/:

These static data structures serve as the interface object for RGX scheduling and system communications. Tasks wait for a message event at mailbox. A message can be posted to a mailbox from another task, an ISR or from the RGX operating system. If a message is already at a mailbox the task receives the message address and execution continues. If not, the task is suspended until one is posted. A task can wait for a message for a number of RGX system clock ticks or forever.

A mailbox consists of 2 pointers. The first pointer is the address of a task descriptor waiting at the mailbox or 0 if there is none. The second is the address of a message posted at the mailbox or 0 if there is none.. Messages can be queued at a mailbox through the message link field chain.

Messages:

These memory structures are used to send data between tasks, from ISRs and RGX to tasks. Except for a mandatory header, they are user defined.

The message header defines a link field, a return mailbox address and message type field. The link field allows messages to be queued at an exchange by chaining link fields. A link field value marks of 0 marks the end of the chain and queue.

Once a message has been posted at a mailbox, it is out of the scope of the sending task and it loses possession of until it is received again from `rgx_wait` or `rgx_getmsg`. The field 'retex' contains a return mailbox address. Because messages can be chained, a message cannot be posted more than once at a mailbox until the sending task regains possession.

The only exception are system messages sent from RGX itself. These messages are never returned and contain static data.

The type field identifies the message type which is largely user defined. Message types 0 through 15 are reserved for RGX system messages.

RGX Code, Descriptors and RGX Configuration:

RGX uses descriptors and data structures defined in the `RGXCFG` configuration module. There are descriptors for task management and for mailboxes.

There are 2 types of task descriptors. The task creation descriptors are defined in the code segment and are used to generate runtime task descriptors in the data segment. Runtime task descriptors contain the dynamic status of a task and task stack information. (RGX51 runtime descriptors contain stack storage space.)

Mailbox/Exchanges are defined in the configuration and are initialized at startup.

The runtime descriptors and mailboxes are arranged in arrays to allow the RGX scheduler to run faster. To keep RGX as simple as possible, dynamic task and mailbox creation are not supported. These services are usually not required in embedded control systems.

The configuration module also contains an Idle task, and the public variable ActiveTask.

RGX System Functions:

start_rgx

This is the startup and entry point for RGX. This is usually called at the end of **Main** after any hardware setup or initializations that need to be done outside the RGX environment. The configuration tables are used to create and initialize the system data structures. Task execution starts with the highest priority task.

rgx_wait

```
void xdata * rgx_wait(struct rgx_exchange xdata *, unsigned int time);
```

A call to `rgx_wait` causes a task to wait for a message at a mailbox. If a message is present, `rgx_wait` returns the message address. If there is no message posted and 'time' is 0 then the task is suspended and placed on the waiting list until a message is sent to the mailbox. If 'time' is more than 0, then the task is placed on a delay list until 'time' clock ticks count down to 0 then RGX system posts a system message of timeout type and the task is placed on the active list. In either case that a task is made waiting context arbitration is started.

All tasks, except the idle task, must call `rgx_wait` at least once in a loop. If it does not, all tasks of lower priority will be blocked from executing.

rgx_recmsg

```
void xdata * rgx_recmsg(struct rgx_exchange xdata *);
```

If a message is present, a call to `rgx_recmsg` returns the message address. If not

0 is returned. A call to `rgx_recmsg` does not start context arbitration.

rgx_send

```
void rgx_send(void xdata *, struct rgx_exchange xdata *);
```

`rgx_send` posts a message to a mailbox and starts context arbitration. If a task of higher priority is waiting at that mailbox, the sending task is suspended and placed on the preempted list. The task will resume when higher priority tasks are waiting again.

rgx_postmsg

```
void rgx_postmsg(void xdata *, struct rgx_exchange xdata *);
```

`rgx_postmsg` posts a message to a mailbox, but does not start context arbitration.

rgx_int_ready

```
void rgx_int_ready(void xdata *, struct rgx_exchange xdata *);
```

`RGX_int_ready` posts a message to a mailbox from an ISR. This call makes a task waiting at the exchange ready. It does not start context arbitration in RGX51..

rgx_clock_tick

```
void rgx_clock_tick();
```

This call is made from the system time base ISR. The interrupt is provided by a timer, usually T1 for 8051s, and must be the last thing done before the ISR returns. The interrupt should be the lowest hardware priority. Because there is no hardware mechanism in the 8051 core to track interrupt processing and depth, and to avoid implementing a software tracking system, this call causes any scheduled task context switch. Common clock tick intervals are between 1 and 10 milliseconds.

