OSEK realtime operating system osCAN Tutorial

Version 1.0

Authors:	Prof. Horst Wettstein, Snezana Radakovic
Version:	1.0
Status:	released
No of Pages:	18

1 Why operating systems?

Software systems are complex systems and they become more complex with every new version or every additional function. That is also a concern of embedded systems, even when they are based on 8 or 16 bit micro controllers. It is very important that those systems posses a well defined structure. Furthermore, real time applications require the possibility to exploit concurrency and to employ elementary services for interaction and resource management. The combination of both these targets can best be achieved on the basis of an appropriate operating system.

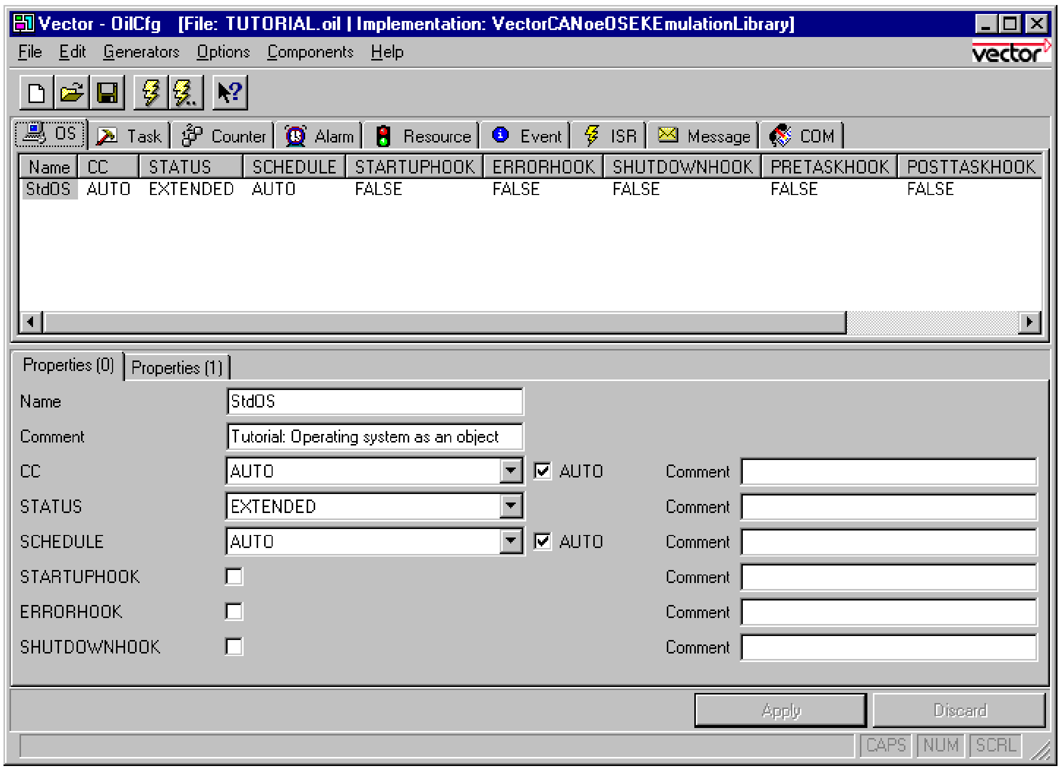
Most of these services can be realised in many variants depending on semantic details. According to that technical applications deal with so-called dedicated systems, which means that all objects involved are known in advance, relations among such objects are of static nature. So operating systems used for real time applications should and can supply services very efficiently. There is no case against their usage in contemporary embedded situations.

OSEK is an operating system developed especially for automotive applications. It offers a rich set of application program interfaces (API functions) to the application programmer. Nevertheless, programmers should make an effort to switch from a conventional monolithic style of programming to become familiar with concepts such as tasks, scheduling policy, message passing, or time management. The following tutorial helps the programmer to advance quickly to an appropriate level of competence for applying those concepts.

An operating system (more precisely an operating system kernel) involves a certain number of system objects. These objects consist of a data structure for their attributes and of a set of operations (called system services) to accomplish change of state. System services are a constant part of operating systems while the number of objects and their attributes depend on a particular application. The OSEK environment supplies the OIL configuration tool (in the case of osCAN it is a window oriented implementation) for very comfortable creation of applications. That means for any system object there is a sheet to set up the object's attributes. The following tour should demonstrate in a simple manner how to use this tool and make new application.

2 Operating system as an object

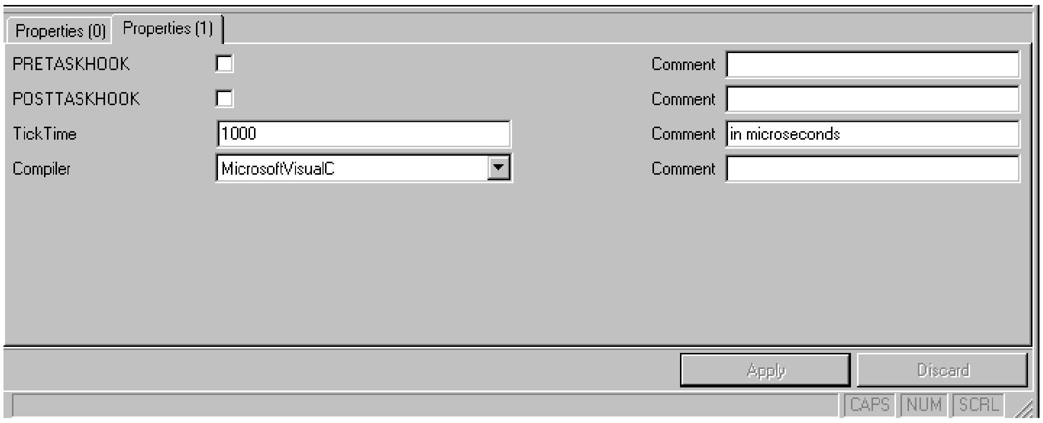
The operating system can be considered as an object, too. Naturally, it can be declared only once. The operating system parameters (and their attributes) can be treated such as parameters (and their attributes) of the other application oriented objects (tasks, counters, alarms, events, etc.). The main system parameters are: choice of functionality (conformance classes), degree of error checking (standard or extended), scheduling strategy (non preemptive scheduling, full preemptive scheduling, mix or AUTO scheduling), usage (or not) of hook routines, fundamental tick period (in µs) and stack management. There are also some specific realisation variants (refer to the chapter 6.2.). Some parameters are always present (like scheduling strategy that allows to select the task switching mechanism, choice of tick period, etc.) while others depend on implementation. Example in Figure 2-1 and Figure 2-2 point out properties (and chosen attributes) of the operating system.


Vector - OilCfg [File: TUTORIAL.oil | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties (0) Properties (1)
Apply Discard
CAPS NUM SCRL

Name	CC	STATUS	SCHEDULE	STARTUPHOOK	ERRORHOOK	SHUTDOWNHOOK	PRETASKHOOK	POSTTASKHOOK
StdOS	AUTO	EXTENDED	AUTO	FALSE	FALSE	FALSE	FALSE	FALSE

Name	StdOS		
Comment	Tutorial: Operating system as an object		
CC	AUTO	☑ AUTO	Comment
STATUS	EXTENDED		Comment
SCHEDULE	AUTO	☑ AUTO	Comment
STARTUPHOOK	☐		Comment
ERRORHOOK	☐		Comment
SHUTDOWNHOOK	☐		Comment

Figure 2-1 Declaration of operating system parameters



PRETASKHOOK	☐	Comment
POSTTASKHOOK	☐	Comment
TickTime	1000	Comment: in microseconds
Compiler	MicrosoftVisualC	Comment

Figure 2-2 Declaration of operating system parameters

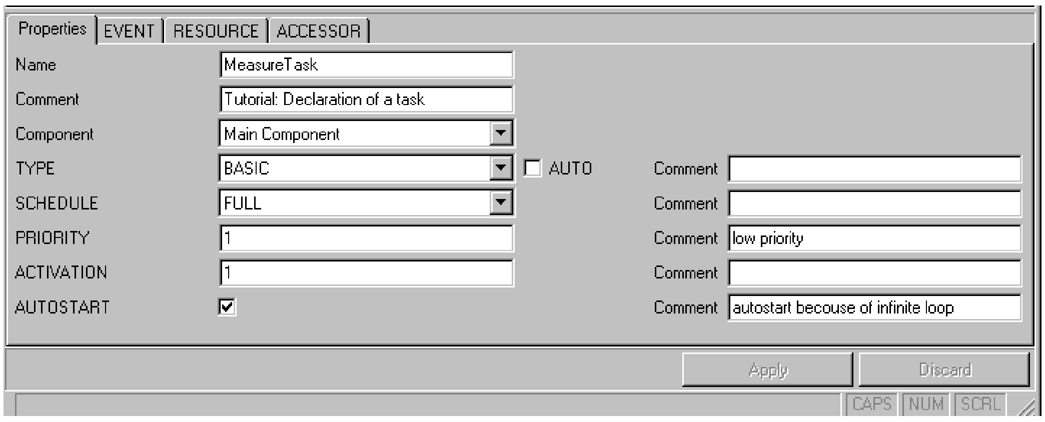
3 Tasks

A usual part of a number of user applications is a demand for receiving measurements from peripherals, treating them according to the applications' requirements and directing them to some other places. So, the user can organise this measuring as an OSEK task called "MeasureTask". Furthermore, the measuring should execute concurrently with other activities (other tasks) in the user system. The general parts of the task are shown in Code 3-1.

```
TASK(MeasureTask)
{
      while (1)
      {
            // Receive data from peripherals
            // Do some computation
            // Direct the result to somewhere else
            //    (e.g. to some other tasks)
      }
}
```

Code 3-1 General parts of task entity

Obviously, this task executes in the background according to the organisation as an infinite loop. Therefore, it can be assigned as a low priority task. Also, there are no wait elements related to the task and therefore the user can declare it as a basic task. The task can be activated automatically at system start up since it is supposed to run infinitely. So the user can declare this specific task in the OIL configuration tool as shown in Figure 3-1.



Name	MeasureTask			
Comment	Tutorial: Declaration of a task			
Component	Main Component			
TYPE	BASIC	☐ AUTO	Comment	
SCHEDULE	FULL		Comment	
PRIORITY	1		Comment	low priority
ACTIVATION	1		Comment	
AUTOSTART	☑		Comment	autostart becouse of infinite loop

Figure 3-1 Declaration of a task

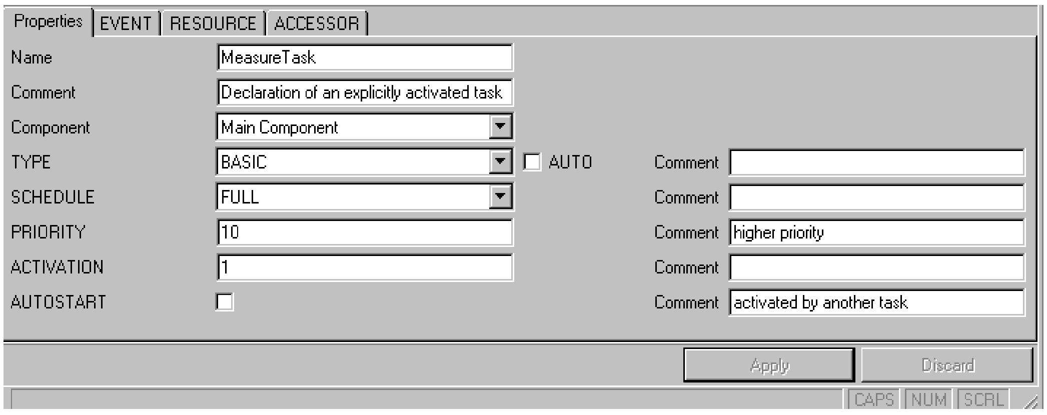
4 Activation of tasks

Afterwards, the development of an application may lead to the conclusion that the permanently running background activity no longer will be appropriate for the processing in some other task. Moreover, measured values are needed in a specific moment (depending on some system state). Therefore, the user should build another task (call it "ControlTask"). If "ControlTask" needs peripheral values, it can activate the measuring task by a special system service (*ActivateTask*). After execution the task for measurements has to terminate itself and then it is waiting for another activation. The termination of the task is mandatory. So, "MeasureTask" should stay passive most of the time. Code 4-1 shows the new application structure.

```
TASK(ControlTask)
{
     usrControlTaskStarted();
     for ( ; ; )
     {
          GetResource(Basic_Resource);
          if(time_for_newmeasurement)
             ActivateTask(MeasureTask);
          // else: do something else
          ReleaseResource(Basic_Resource);
     }
     TerminateTask();
} // END OF basic ControlTask

TASK(MeasureTask)
{
     // Receive input from peripherals
     // Calculate something
     // Send results
     usrNewMeasurement();

      TerminateTask();
} // END OF basic MeasureTask
```

Code 4-1 Activation and (self)termination of a task

The function calls with the prefix "usr" are user defined functions which are implemented separately. In this application context "MeasureTask" is not automatically activated. The priority should be higher. It should be allowed to execute without interruption by another task in case of the scheduling strategy has finally decided to begin its execution. Figure 4-1 shows the new declaration parameters.



Name	MeasureTask			
Comment	Declaration of an explicitly activated task			
Component	Main Component			
TYPE	BASIC	☐ AUTO	Comment	
SCHEDULE	FULL		Comment	
PRIORITY	10		Comment	higher priority
ACTIVATION	1		Comment	
AUTOSTART	☐		Comment	activated by another task

Figure 4-1 Declaration of an explicitly activated task

5 Events

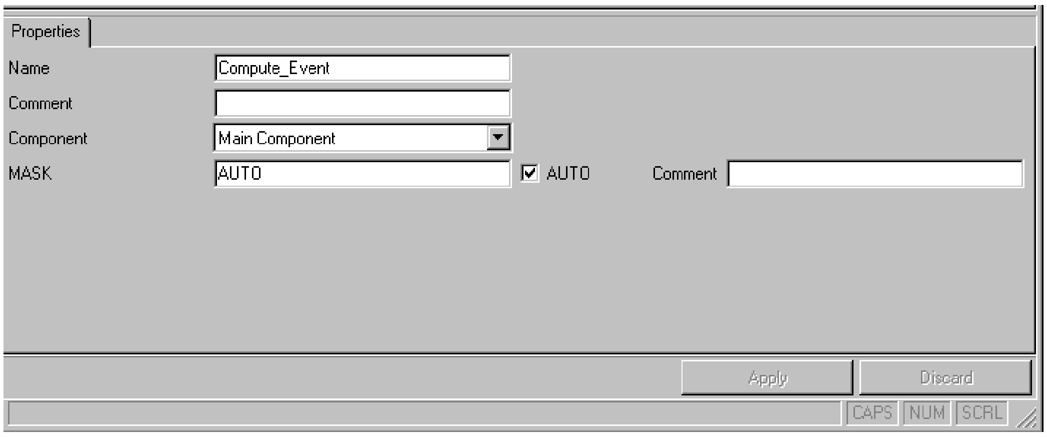
Suppose that the user needs data from the peripheral device which is a node of a net (e.g. peripheral device is part of a distributed system connected by the CAN- bus). The user has to send a request to obtain next peripheral's values and has to wait for an answer. In some cases it is very important to synchronize obtaining and processing mechanism of measured values. The user can split the task for measuring into two tasks ("MeasureTask" for measuring and "ComputeTask" for calculations) and synchronize them by using an event object. The new system structure is presented in Code 5-1.

```
TASK(MeasureTask)
{
     // Receive input from peripherals
     usrNewMeasurement();
     SetEvent(ComputeTask, Compute_Event);
     TerminateTask();
}   // END OF basic MeasureTask
```

```
TASK(ComputeTask)
{
      for( ; ; )
      {
            WaitEvent(Compute_Event);
            ClearEvent(Compute_Event);

            // Calculate something
            // Send results
            usrNewCalculation();
      }
      TerminateTask();
}   // END OF extended ComputeTask
```

Code 5-1 Usage of an event object

All tasks can send events to any not suspended extended tasks. Only the owner is able to clear its events and to wait for the reception of its events. In this example, the basic task "MeasureTask" sets the event of the extended task "ComputeTask" and the extended "ComputeTask" waits for this event and when it occurs - clears it. The event is set according to the event mask (AUTO MASK is chosen). An OSEK event mask consists of several bits. Generally, it is allowed to specify which bit the task is waiting for or which bit to clear. In this simple example the OIL configuration tool decide which bit is used. Figure 5-1 shows the event specification.


Properties

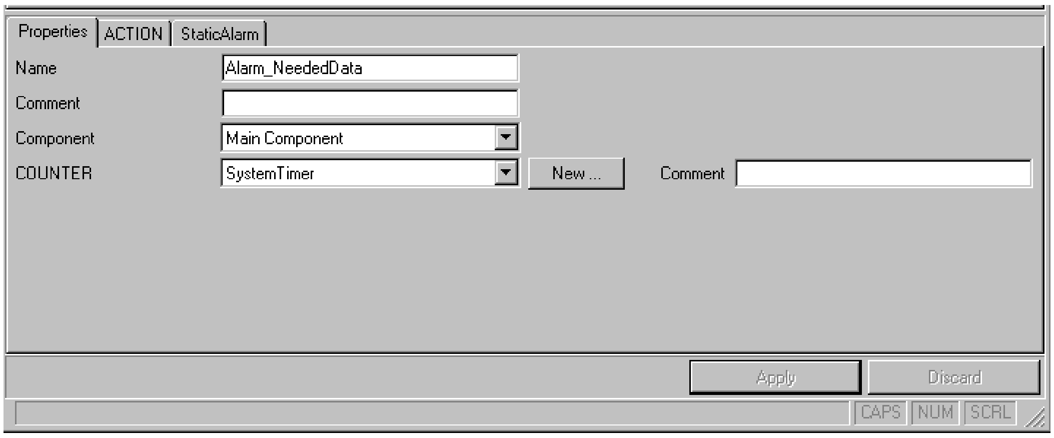
Name	Compute_Event		
Comment			
Component	Main Component		
MASK	AUTO	☑ AUTO	Comment

Figure 5-1 Declaration of events

6 Alarms

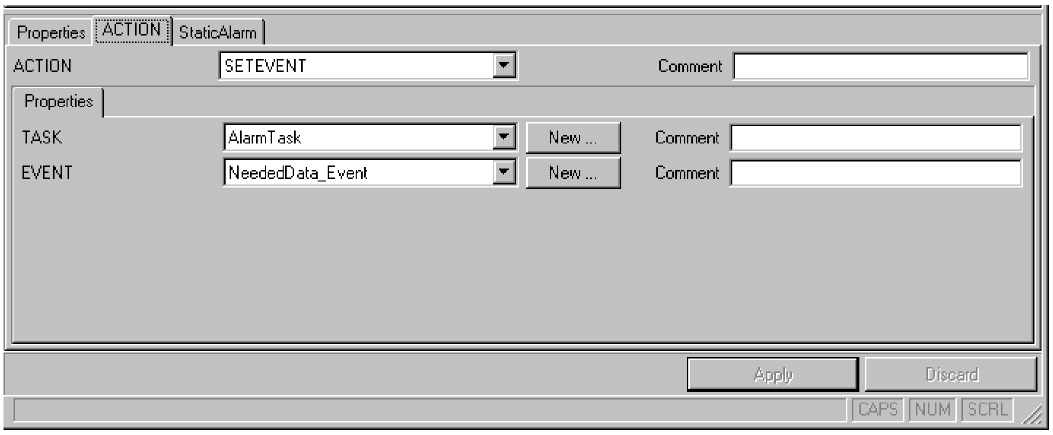
The user usually needs the ability for periodical measurements. The OSEK operating system concept makes it possible by using counters and alarm objects. The time base of the operating system is configurable and can be set in the OIL configuration tool (TickTime in OS object). The osCAN implementation realises a predefined counter called the system timer. The operating system provides services to activate tasks, set events or call alarm "callback" routine when an alarm expires. The operating system also provides services to cancel alarms. The alarm mechanism involves the basic tick rate, the start time, the period time and the desired operation (*ActivateTask*, *SetEvent* and "callback" are supplied). It is important to specify a task or an event related to the alarm.

For instance, the "AlarmTask" is responsible for the time when a next measurement will occur. The alarm object and some parameters (the tick base, the task and the event related to the alarm) must be declared at system generation time (refer to Figure 6-1 and Figure 6-2). In the application program only a start time and a period time have to be set (Code 6-1).



Name	Alarm_NeededData
Comment	
Component	Main Component
COUNTER	SystemTimer

Figure 6-1 Alarm declaration



ACTION	SETEVENT
TASK	AlarmTask
EVENT	NeededData_Event

Figure 6-2 Alarm declaration

```
TASK(AlarmTask)
{
      SetRelAlarm(Alarm_NeededData, MSEC(100), MSEC(500));

      for( ; ; )
      {
           WaitEvent(NeededData_Event);
           ClearEvent(NeededData_Event);
           UsrTimeForNewMeasurement();

      }

} // END OF extended AlarmTask
```

Code 6-1 Set up alarm parameters for periodical measurements

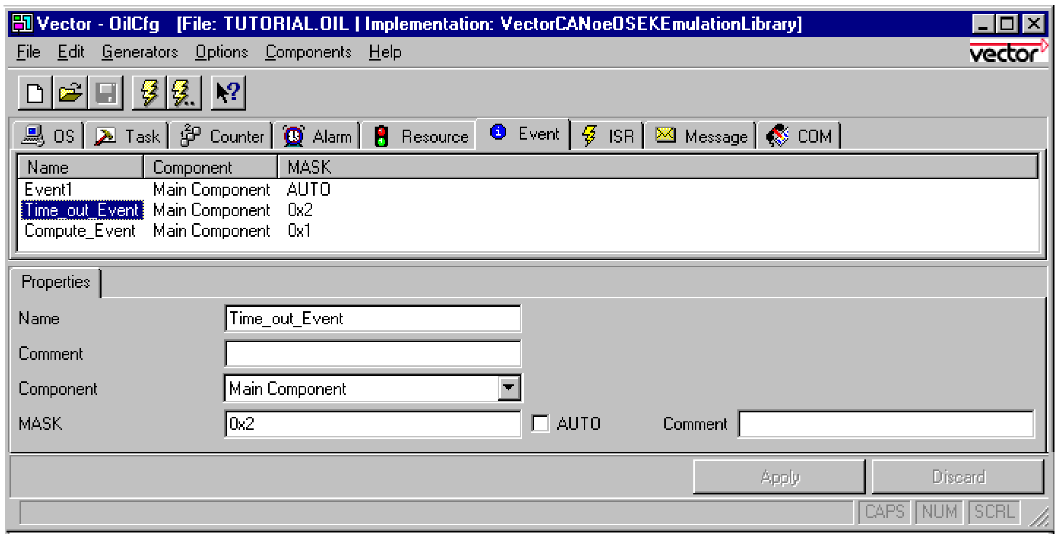
7 More powerful events

Suppose that the result of a computation from measured values is necessary to supply a part of a technical process as an essential parameter. For example, this parameter has to be available at certain time instances. It depends on an external speed. Because of the congestion in some part of the system the next valid value will not arrive to the compute task at the appropriate moment. So the user can decide that the task for calculations should take the last available value, eventually extrapolate it according to an appropriate formula and send it to the external device. To realise this, the wait-event operation has to be triggered by two different sources, the data source for the regular case (first bit) and an alarm source for the timeout case (second bit).
This situation shows a typical application of multiple bit events. If any of the event bits in the event mask changes to the "set up" state the task leaves the "wait" state and begins with execution (refer to Code 7-1 and Figure 7-1). In this example the event masks are set manually. Usually the masks are defined symbolically and set up automatically by the OIL configuration tool.

```
TASK(MeasureTask)
{
      //  Receive input from peripherals
      usrNewMeasurement();

      SetEvent(ComputeTask, 0x01);

      TerminateTask();
}   // END OF basic MeasureTask
```

```
TASK(ComputeTask)
{
      EventMaskType Event;
      SetRelAlarm(Time_out, MSEC(100), 0);

      for(;;)
      {
            WaitEvent(0x01 | 0x02);
            GetEvent(ComputeTask, &Event);

            if(Event & 0x01)
            {
                  ClearEvent(0x01);
                  CancelAlarm(Time_out);
                  //  Calculate something
                  //  Send results

                  usrNewCalculation();

                  SetRelAlarm(Time_out, MSEC(100), 0);
            }
            else if (Event & 0x02)
            {
                  ClearEvent(0x02);
                  //  Do time_out handling
                  usrTimeoutHandling();
            }
      }
      TerminateTask();
}    // END OF extended ComputeTask
```

Code 7-1 Usage of multiple bit events


Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties
Apply Discard
CAPS NUM SCRL

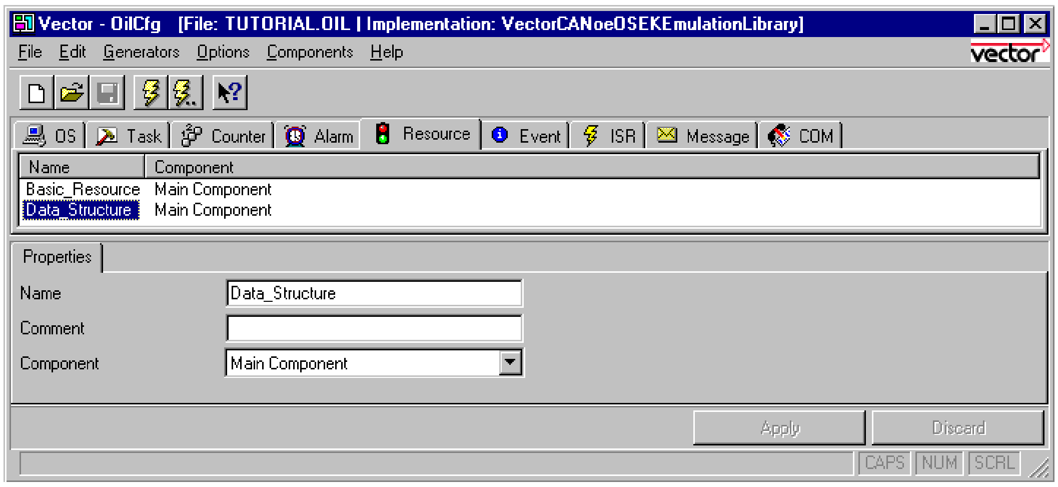
Name	Component	MASK
Event1	Main Component	AUTO
Time_out_Event	Main Component	0x2
Compute_Event	Main Component	0x1

Name	Time_out_Event
Comment	
Component	Main Component
MASK	0x2
AUTO	☐
Comment	

Figure 7-1 Declaration of a two bits event

8 Resources

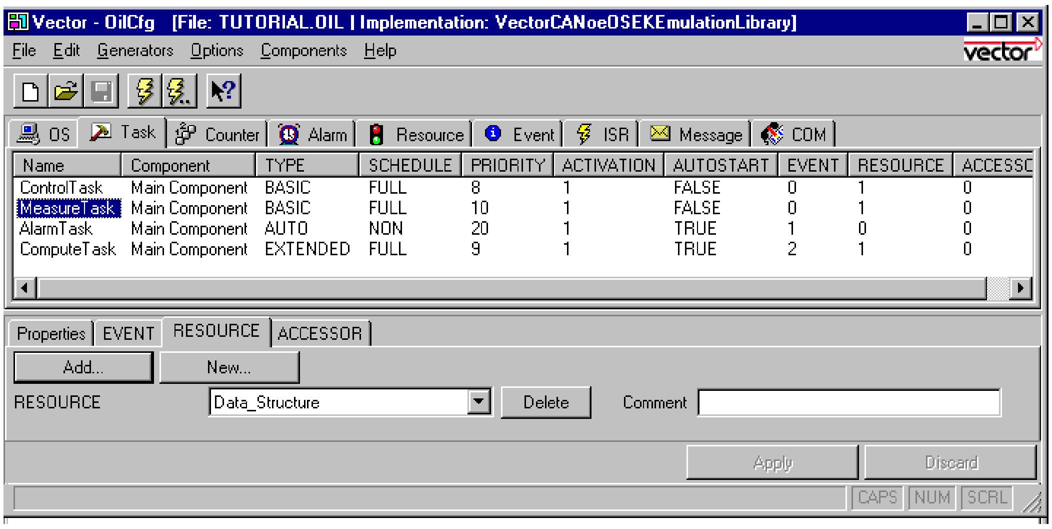
Generally, there are two concepts for data exchange among tasks or among the others software units: *access to shared data structures* and *messages transfer*. The shared data structures are more efficiently than passing of messages and that is the reason they are illustrated at first.
The user has to declare a shared data structure outside the tasks that intend to access it. The task can be interrupted in the middle of an access to the data structure, if two or more tasks use the shared data structure. It can lead to incorrect data when the interrupted task comes back into the running state. To avoid this problem, most operating systems offer specific lock operations called semaphore. The main disadvantage of using of semaphores is the problem related to priority inversion and the deadlock problem. To avoid the problems of priority inversion and deadlocks the OSEK operating system involves priority ceiling protocol (refer to the specification of the OSEK operating system). The OSEK operating system treats shared variables as resources and improves synchronisation mechanisms by supplying the user with two operations: *GetResource* - to lock resources (i.e. critical sections) and *ReleaseResource* - to release resources (critical sections). The OSEK strictly forbids nested access to the same resource.
Figure 8-1 and Figure 8-2 present declaration of a resource object in the OIL configuration tool and Code 8-1 presents the usage of that resource object in the application.


Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties
Apply Discard
CAPS NUM SCRL

Name	Component
Basic_Resource	Main Component
Data_Structure	Main Component

Name	Data_Structure
Comment	
Component	Main Component

Figure 8-1 Declaration of a resource object


Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties EVENT RESOURCE ACCESSOR
Add... New...
Delete
Apply Discard
CAPS NUM SCRL

Name	Component	TYPE	SCHEDULE	PRIORITY	ACTIVATION	AUTOSTART	EVENT	RESOURCE	ACCESSO
ControlTask	Main Component	BASIC	FULL	8	1	FALSE	0	1	0
MeasureTask	Main Component	BASIC	FULL	10	1	FALSE	0	1	0
AlarmTask	Main Component	AUTO	NON	20	1	TRUE	1	0	0
ComputeTask	Main Component	EXTENDED	FULL	9	1	TRUE	2	1	0

RESOURCE: Data_Structure
Comment:

Figure 8-2 The task with its related resource

```
TASK(MeasureTask)
{
      GetResource(Data_Structure);
      // Receive input from peripherals
      usrNewMeasurement();
      ReleaseResource(Data_Structure);

      SetEvent(ComputeTask, Compute_Event);

      TerminateTask();
}    // END OF basic MeasureTask
```

```
TASK(ComputeTask)
{
      EventMaskType Event;
      SetRelAlarm(Time_out, MSEC(100), 0);

      for(;;)
      {
            WaitEvent(Compute_Event | Time_out_Event);
            GetEvent(ComputeTask, &Event);
            if(Event & 0x1)
            {
                  ClearEvent(Compute_Event);
                  CancelAlarm(Time_out);
                  GetResource(Data_Structure);

                  // Calculate something
                  usrNewCalculation();

                  ReleaseResource(Data_Structure);
                  // Send results
                  SetRelAlarm(Time_out, MSEC(100), 0);
            }
            if (Event & Time_out_Event)
            {
                  ClearEvent(Time_out_Event);
                  // Do time_out handling

                  usrTimeoutHandling();
            }
      }

      TerminateTask();
}   // END OF extended ComputeTask
```

Code 8-1 Mutually exclusive access to shared data

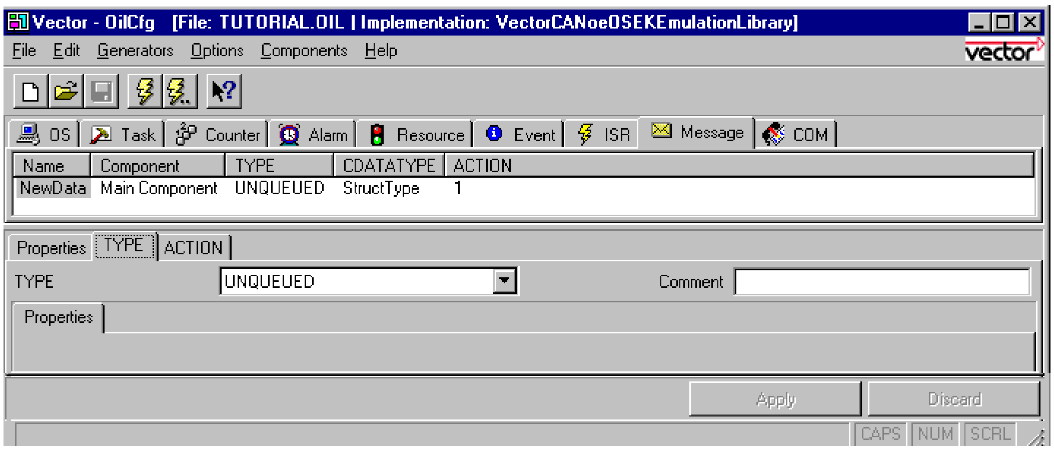
9 Communication

The second concept for data exchange is the transmission of messages. The messages can be transported by value, by reference or by using a buffer. The source task ("ComputeTask" in the example) sends a message from the source buffer to the target buffer of the target task ("ReceiveTask" in the example). The synchronization mechanism among the sending and the receiving task is assured by using an event object (Send_Event in the example). Sending and receiving are realised by OSEK operating systems services *SendMessage* and *ReceiveMessage*. The StartCOM service is called in the first running task in an application. It is important to provide the message initialisation function (*MessageInit*) in the user application although the user can let this function empty. Code 9-1 presents intertask communication for transmitting the computed data from the sending task ("ComputeTask") to the task waiting for the message ("ReceiveTask").

```
TASK(ComputeTask)
{
      StructType SourceBuffer;
      EventMaskType Event;

      SetRelAlarm(Time_out, MSEC(100), 0);

      for( ; ; )
      {
            WaitEvent(Compute_Event | Time_out_Event);
            GetEvent(ComputeTask, &Event);
            if(Event & Compute_Event)
            {
                  ClearEvent(Compute_Event);
                  CancelAlarm(Time_out);
                  GetResource(Data_Structure);
                  // Calculate something
                  usrNewCalculation();
                  ReleaseResource(Data_Structure);

                  // Send results
                  SendMessage(NewData, &SourceBuffer);
                  SetRelAlarm(Time_out, MSEC(100), 0);
            }
            else if (Event & Time_out_Event)
            {
                  ClearEvent(Time_out_Event);
                  // Do time_out handling

                  usrTimeoutHandling();
            }
      }

            TerminateTask();
}    // END OF extended ComputeTask

TASK(ReceiveTask)
{
      StructType TargetBuffer;

      for(;;)
      {
            WaitEvent(Send_Event);
            ClearEvent(Send_Event);
            ReceiveMessage(NewData,&TargetBuffer);
            // process the received data
            usrReceivedMessage();

      }

      TerminateTask();
} // END OF extended ReceiveTask
```

Code 9-1 Message Transfer

The communication object has to be declared via the OIL configuration tool. The user can set up the type of the message (queued or ungueued; refer to Figure 9-1), the event object for synchronization including the action of the message (Figure 9-2) and buffering of the message (WithCopy or WithoutCopy; refer to Figure 9-3). Sometimes messages can be queued for the case when the target process needs some previous values.

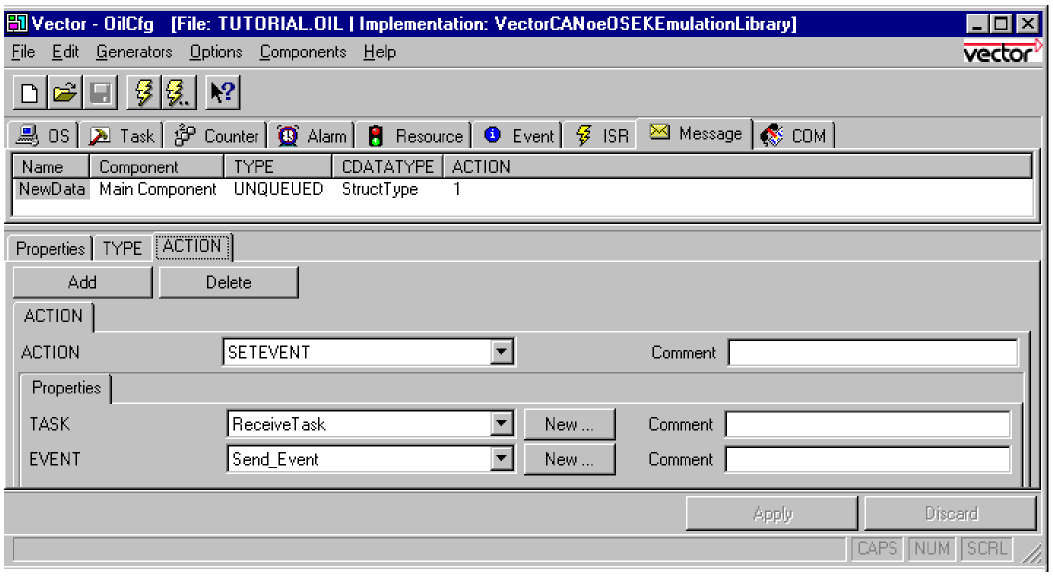

Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties TYPE ACTION
Apply Discard
CAPS NUM SCRL

Name	Component	TYPE	CDATATYPE	ACTION
NewData	Main Component	UNQUEUED	StructType	1

TYPE: UNQUEUED

Comment:

Figure 9-1 The type of the message


Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties TYPE ACTION
Add Delete
New ...
Apply Discard
CAPS NUM SCRL

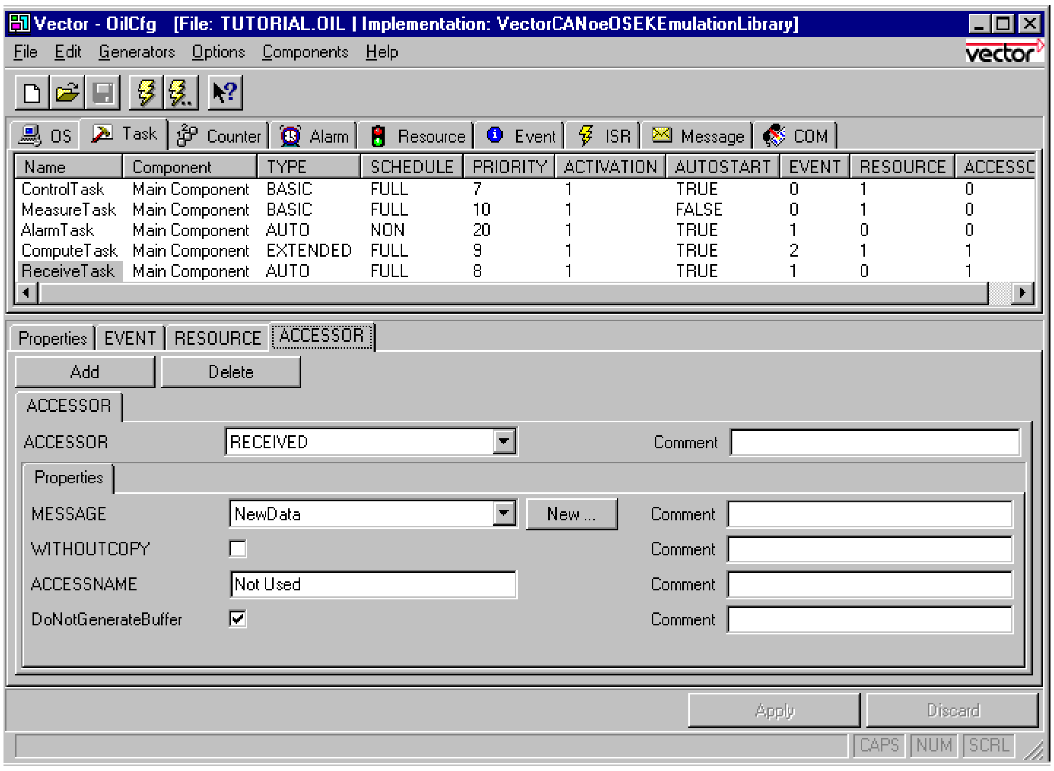
Name	Component	TYPE	CDATATYPE	ACTION
NewData	Main Component	UNQUEUED	StructType	1

ACTION: SETEVENT

TASK: ReceiveTask

EVENT: Send_Event

Figure 9-2 The action of the message


Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties EVENT RESOURCE ACCESSOR
Add Delete
Apply Discard
CAPS NUM SCRL

Name	Component	TYPE	SCHEDULE	PRIORITY	ACTIVATION	AUTOSTART	EVENT	RESOURCE	ACCESSC
ControlTask	Main Component	BASIC	FULL	7	1	TRUE	0	1	0
MeasureTask	Main Component	BASIC	FULL	10	1	FALSE	0	1	0
AlarmTask	Main Component	AUTO	NON	20	1	TRUE	1	0	0
ComputeTask	Main Component	EXTENDED	FULL	9	1	TRUE	2	1	1
ReceiveTask	Main Component	AUTO	FULL	8	1	TRUE	1	0	1

ACCESSOR

ACCESSOR	RECEIVED	Comment	

Properties

MESSAGE	NewData (New ...)	Comment	
WITHOUTCOPY	☐	Comment	
ACCESSNAME	Not Used	Comment	
DoNotGenerateBuffer	☑	Comment	

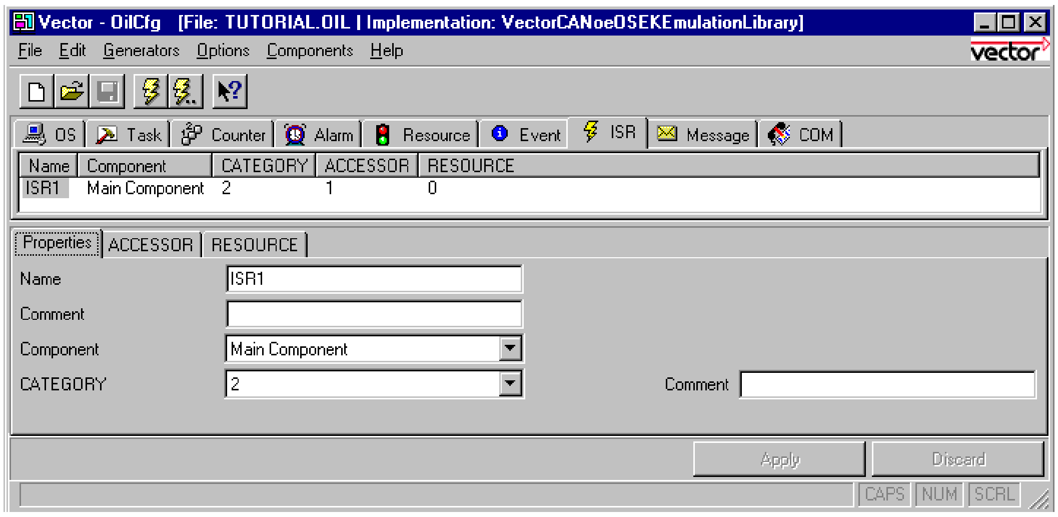
Figure 9-3 Buffering of messages

10 Interrupt Service Routines

The real time operating system is able to offer the opportunity to an application programmer to implement his/her own interrupt service routines like reaction to interrupts from the technical environment. The interrupt routine should be executed as soon as possible when the interrupt occurs. The programmer should take care when employing these routines. The OSEK operating system distinguishes two different categories of ISRs. There are interrupts of category1 which are not allowed to use API functions (therefore they are faster) and interrupts of category 2 which may use, certain restricted, API functions (they are able to initiate taskswitches).
The example may be the case when the measured values cause an interrupt i.e. the time of emergence of peripherals values is unpredictable. Code 10-1 presents the usage of this interrupt service routine. Figure 10-1 presents the declaration of an interrupt object.

```
ISR (ISR1)
{
      // Eventually clear interrupt signal
      // Take input from peripheral device
}
```

Code 10-1 Interrupt service routine


Vector - OilCfg [File: TUTORIAL.OIL | Implementation: VectorCANoeOSEKEmulationLibrary]
File Edit Generators Options Components Help
OS Task Counter Alarm Resource Event ISR Message COM
Properties ACCESSOR RESOURCE
Apply Discard
CAPS NUM SCRL

Name	Component	CATEGORY	ACCESSOR	RESOURCE
ISR1	Main Component	2	1	0

Name	ISR1
Comment	
Component	Main Component
CATEGORY	2
Comment	

Figure 10-1 Declaration of an interrupt object