Introducing

ARTS
Advanced Real-Time Switch

I had offered Cypress to name it Cy-PRESS, but they refused to take the acronym of
Cypress PSoC Real-time Embedded Scheduler Service.

What is a task scheduler and how can it help you making programming easier and cheaper?

Well, that will take a bit of time to explain, so why not get a cup of coffee, relax, sit back in your chair and read.
Do you have one of these old-fashioned machines that just keep warm the coffee or do you have one of these modern ones which grind the coffee and brew it freshly and -if you're a lucky guy- put some hot milk-foam on top of your mug? Those machines are driven by a microprocessor of course, so let's think about to put all that logic into a PSoC 4.
How does such a machine work like? Well, there is a grinder and a sensor telling the machine that enough coffee is grinded. Some noises in my machine tell me that there is some mechanic at work, so there are a motor and probably another sensor working. Then a water-pump comes up for a short time, pauses and then pumps again a selectable amount of hot water through the coffee powder. Hot water? Oh, yes! So we need a heating and a temperature sensor. And because some other guys will have espresso or latte macchiato or just a simple milk-coffee (my favorite, a big one, please) we need a couple of buttons for selections. Ah, and some safety-sensors that keep watch that none of the parts of the machine is open while brewing, the waste-container is not full yet, there's water and there's coffee and there's milk. A handful of indicators (green or blue or even white LEDs) make our coffee-brewing-machine look interesting.
Complex enough, but not too complicated to fit into a PSoC 4, so let's think about, how it should be programmed:

We can Isolate a couple of independent modules:

- The grinder with its sensor(s)
- The heater with its sensor
- The keyboard with a few buttons
- The mechanic motor-driven parts with their sensors
- The water-pump with its volume sensor
- The LED indicators
- Some general sensors for doors (not windows!) and other resources.
- The keypad / button interface

Now we can easily define the functions for the modules one by one:

Grinder:

Start grinding
When Sensor for coffee comes: Stop grinding.

That easy??
I'm afraid not!
Let's try to translate that into C

```
void GrindCoffee(void)
{
     StartGrindMotor();
     while( ! CoffeeGrindSensor()); // Wait
     StopGrindMotor();
}
```

Now, what happens, if the sensor doesn't come up (Not enough coffee)?
So we ought to change the logic a bit:

```
void GrindCoffee(void)
{
unsigned int GrindWait;
#define GRINDTIMEOUT 5000          // should be 5 seconds
     StartGrindMotor();
     GrindWait = GRINDTIMEOUT;     //
     while(!(CoffeeGrindSensor() && GrindWait--)); // Wait
     StopGrindMotor();
     if (!GrindWait) {             // Counted down to zero?
          // ERROR, we have a timeout!
          ErrorStopMachine();
     }
}
```

Looks better, doesn't it? But what happens during the execution of the loop

while(! (CoffeeGrindSensor() && GrindWait--)); // Wait

To be honest, nothing happens at all, all CPU-time is used up checking the sensor, regardless what is happening in any other part of the machine, the keyboard, the heat-sensor and so on, **THE PROGRAM WILL NOT SEE IT!**

Obviously it would be much better and safer if, while waiting for the sensor to signal the grinding is ready, the program could look for the keyboard, the heating and the waste-basket (and for the milk, if you prefer).
So what can we do to get out of this?
The answer is: RTOS, the acronym for "Real-Time Operating System". A class of programs which ARTS is a member of. When using ARTS the grinding module could look like this:

```
Task (GrindCoffee)
{
#define GRINDTIMEOUT (TICKSPERSECOND * 5)  // 5 seconds
EventID Delay;
     while(forever)
     {
          WaitForMonitorByte(&StartGrinding);     // Wait start
          Delay = SetupDelay(GRINDTIMEOUT);       // Timeout setup
          SetupMonitorByte(&GrindSensor);         // Sensor setup
          StartGrindMotor();                      // Grind
          if(Wait() == Delay) // returns EventID that ended Wait
          {         // ERROR, we have a timeout!
               ErrorStopMachine();                // Handle error
          }
          else CoffeeGrinded = TRUE;

          StopGrindMotor();                       // Stop grinding
          StartGrinding = FALSE;                  // Finished
     }
}
```

The difference is, that when using ARTS the line

```
          if(Wait() == Delay)
```

gives control to a ARTS function named Wait() which in turn gives control to other defined modules as the heater, the keyboard the waste-basket (and the milk, if you prefer). At a frequency of up to 1000 times a second these modules, simple sub-routines as that one for the grinder, are executed and their functions performed.

The controlling program could look like

```
#define STACKDEPTH 30
     PumpTask  = CreateTask(&PumpWater, STACKDEPTH);
     GrindTask = CreateTask(&GrindCoffee, STACKDEPTH);
     HeaterTask= CreateTask(&HeatWater, STACKDEPTH);
     … and so on
     // make coffee
     WaterHeated = FALSE;
     StartTask(HeaterTask);// We can heat the water while grinding
     CoffeeGrinded = FALSE;
     StartGrinding = TRUE);   // GrindCoffee task sees that
     WaitForMonitorByte(&WaterHeated);  // wait until water is hot
     WaitForMonitorByte(&CoffeeGrinded);// wait until grinded
     WaterPumped = FALSE;
     WaitForMonitorByte(&WaterPumped);// water hot is a noble
thing
     …
```

You see the pattern? Every time we call a wait-function the ARTS takes over and schedules the execution to all the tasks still active and running.
But what, if one (or more than one) task uses the CPU for a longer period of time without calling Wait()? Do the other tasks stall? No, they don't, every 1/1000th of a second the running tasks are interrupted and the scheduler looks for another task to run. The scheduler takes a Task-Priority into account when it selects the next task to run. And if there is no task to run? There is a system's Idle-Task that may send the PSoC into low power mode until the next time-slice is over.

What does it cost? In Cent?? ARTS takes 3-8k from your program memory. It uses some Cortex M0 internal resources as the TickTimer. That's all.
What you get? Flexibility! Should be written in bold: **Flexibility!!!** You want to warm the cups? You've got a steam-valve? The integration of any new or altered components or changed designs is much easier with ARTS. Changing task-priorities? In the first non-ARTS program design example probably a nightmare, with ARTS a one-liner:

```
     RaisePriority(MyTask);
```

Anything more? Yeah, you get access to the NMI and the HardFault interrupt that informs your program of something seriously going on and give a chance to take appropriate action.

Now for the bad thing: I do not have a coffee-brewing-machine. Still worse: I have got one, but I am not willing to cannibalize it for the sake of an interesting PSoC demo. So what have I got? And what have you got??
Fortunately there is the PSoC 4-M Pioneer board CY8CKIT-044 that has got several hardware components and comes with example projects.

The PSoC4-M Pioneer Kit

I have set up a demonstration project that shows some of the capabilities of ARTS. I used the provided examples from Cypress and let them all run in parallel. You will need PuTTY or a similar terminal emulation to display the results on your PC screen.

A brief overview of the project:

```
int main()
{
      InitializeSystem();
      CreateAllTasks();
      ARTS_Run();                    //    Will never return
      return 0;                      //    Just to avoid a warning
}
```

The tasks running in parallel:

```
void CreateAllTasks(void)
{
      ARTS_CreateTask(MyTask,70,1);        //Create a task
      ARTS_CreateTask(MyTask,70,2);        //Create another instance of same task
      ARTS_CreateTask(CountTask,70,0);     //Create one more task
      ARTS_CreateTask(MemTask,80,0);       //Task using local memory on stack
      ARTS_CreateTask(ShortTask,30,0);     //Task that will be aborted by ARTS
      ARTS_CreateTask(OneSecTask,70,0);    //Task which waits for one second
      ARTS_CreateTask(ByteCheckingTask,70,0);   //Task which waits for a byte
                                                //becoming non-zero
      ARTS_CreateTask(FunctionCheckingTask,70,0);//Task which waits for a
                                                //function returning non-zero
      ARTS_CreateTask(ADCTask,70,0);       //Reading analog values
      ARTS_CreateTask(ProxiTask,64,0);     //Proximity sensors on Kit-044
      ARTS_CreateTask(TestUART,0x80,0);    //UART
      ARTS_CreateTask(LightSensorTask,0x80,0);//Ambient light sensor
      ARTS_CreateTask(AccelTask,0x100,0); //Accelerometer
      ARTS_CreateTask(Producer,0x80,0);    //Message producer task
      ARTS_CreateTask(Consumer,0x80,0);    //Message consumer task
      ARTS_CreateTask(ScreenHandler,0x100,0);// Screen output organizer
}
```

The second parameter of ARTS_CreateTask() is the number of 32-bit stack entries to reserve, the third is an optional parameter given to the task.

When you look into the ARTS.h file you will see a brief description of all APIs.

Get your Evaluation-board and let the show begin!

Happy coding,
Bob Marlowe