

Homogeneous Resource Configuration and Access for an Autonomous Robotic Vehicle

Steven D. Johnson, Bryce Himebaugh and Scott A. Dial
Indiana University

Copyright © 2008 SAE International

Abstract

ERTS is a computer controlled golf car whose primary functional goal is *autonomous real-world navigation*. Its mission is to serve as a platform for collaborative research in diverse areas of cognitive robotics, human-computer interaction, among others, as well as basic research in embedded and real-time systems. This purpose demands a highly flexible platform on which it is relatively easy to configure and access resources ranging from vehicle components to sensor hardware. One avenue we are exploring toward this end is a uniform *file-space interface*, based in the *Plan 9* operating system that originated at Bell Labs, and more recently, the work of Brown and Pisupati. They extended the approach for embedded and system-on-chip applications. We advocate *synchronous design methods*, where applicable, as this presents a more tractable model for students and researchers with higher experimental aims. The work described here establishes intrinsic support for globally synchronous, real-time interactions through the file-space abstraction. A small example of ERTS steering control is presented to illustrate the approach.

KEYWORDS: embedded systems, autonomous navigation, synchronous software, rapid prototyping.

INTRODUCTION

The *ERTS* robotic vehicle (Fig. 1) is developed by and for our introductory *Embedded & Real-Time Systems* course [1]. Its overall functional goal is autonomous real-world navigation, but the real mission of the ERTS project is to serve as a platform for collaborative research in areas ranging from cognitive robotics (e.g. human-robot interaction) to safety-critical systems (e.g. formal implementation synthesis), as well as other focus areas (e.g. programming languages, robotic vision) at Indiana University.

For a number of distinct reasons, the ERTS mission demands a great deal of flexibility, relative simplicity, and open visibility. In cognitive robotics, for example, we are deploying large existing experimental systems for human-robot interaction. Typically, these systems regard the vehicle as a black-box object with relatively high-level functionality, but often, they also need to have project-specific, even experiment-specific sensor arrays. We need to con-

figure ERTS and integrate the systems very quickly, in order encourage early engagement.

In contrast, embedded systems research may delve deeply into the underlying architecture, to the level of fundamental software and hardware components. ERTS must be configured “vertically” to instantiate an abstract design model, as dictated by the research objective. The topic of this article is an illustration: it develops middleware supporting a *synchronous* software design methodology, discussed later in this Introduction.

Most experimenters and students have minimal experience with embedded systems, but only a few months to reach a productive level of development. For this reason—and because we have a bottom-up teaching philosophy—we prefer to postpone the use of complex tool chains, specialized design environments, and even language-specific interfaces. Because the core ERTS development group is small, we must also keep the ERTS code base small and generic.

One avenue we are exploring to address these challenges is incorporating a homogeneous resource-access interface in the underlying file system (FS). The immediate benefit is exploiting the operating system’s directory services to bind resources in its global, hierarchical name space. The file data space also provides a shared, language-independent *image region* through which ERTS system components interact. It is hardly a new idea, but rather one originating in the earliest stages of Unix design.

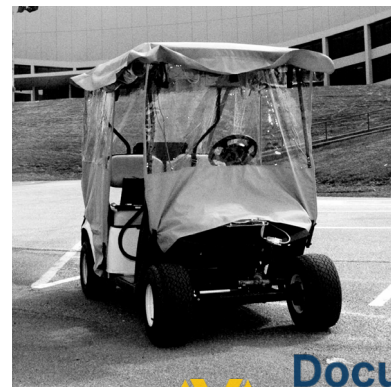


Figure 1: ERTS Vehicle

The file system interface (FS I/F) approach is only a basic framework for experimental prototyping. In order to facilitate rapid development, we must develop support for one or more *design methods*. This paper centers on that aspect. We present a method-specific extension for *synchronous* (or “time triggered” [2]) software design. More specifically, we discuss enhancements to the underlying FS I/F resulting in a framework called *SyncFS* for implementing synchronous design models, and illustrate its use in a navigation sub-system for steering ERTS toward a compass heading.

FILE-SYSTEM INTERFACE. Currently, ERTS provides language-specific software frameworks for application development. The primary function of this framework is sustaining a periodic exchange of *command* and *status* messages between a TCP socket server on ERTS and the external “navigator” (*NAV*) client application. The application is linked to the framework at load time as a call-back procedure, which must meet the timing constraints (typically 20 Hz) of the ERTS communication cycle.

Two problems with this approach are the need to create a suite of functionally equivalent frameworks for each application language and keeping the message conversion services up to date across all frameworks as ERTS capabilities evolve.

This paper is based on an alternative approach that embeds the ERTS bilateral communication protocols in the host file space. Brown and Pisupati propose using a lightweight file system as a uniform resource access interface in embedded and system-on-chip applications [3]. They extend concepts originating in the *Plan 9* operating system [4] to incorporate both software and hardware resources in a uniform way.

The tiny example below illustrates one attraction of the approach. It is a sensor-logging task implemented by a Unix shell script that sweeps through each sensor directory, appending the readings from the enabled sensors to a text file, *log*. The sensor readings and control words are “ghosted” by text files in the file directory.

```
foreach S (sensor*)
  if ((cat $$S/status | grep "ON" | wc -w) <> 0)
    cat $$S/reading >> log
  endif
end
```

The example in Figure 2 illustrates how a self-organizing sensor array is configured. It illustrates how auxiliary information such as executable binaries may be positioned, in the directory hierarchy to reflect the logical architecture of a system.

Any language with standard file I/O can be used in this way to access and manipulate system components. However, in early development stages, especially in system integration and component testing, scripting languages, shell scripts, and even command-line interaction predom-

inate. The Design Illustration presented in later in this article uses the *Python* scripting language, but some of its scripts might be converted to *C* at a later stage of development.

Embedding devices and other resources in the file name space is an old idea, frequently revisited. A recent example is the *ParticleFS* of Decker, Christian, et al., developed for ubiquitous computing applications [5]. It employs the traditional technique of incorporating file transactions in the device drivers. In contrast, we are leaving both the file system and the drivers in tact by introducing transaction processes in the background—in essence we are adding user-level drivers. This simplifies things on the device side because it does obligate the driver to use a particular file transport. And it is an important feature in cases where the system developer does not control the software embedded in the physical device.

SYNCHRONOUS DESIGN SUPPORT The file directory provides a homogeneous name space and homogeneous resource access, but it does not prescribe any means for process synchronization or access coordination. In this paper we present techniques and an example for extending the FS interface to reflect an ad hoc design methodology.

We advocate broader use of *synchronous* programming methods that are prevalent in domains like real-time embedded systems, safety-critical control, and other design-critical applications where the cost of design errors is high relative to overall design costs. Synchronous design languages like ESTEREL [6] have been around for decades, and frameworks like GIOTTO [7] reflect higher level abstractions of architecture and control needed for increasingly complex applications.

Figure 2: From [3]. A script that downloads binaries to configure individuals in a self organized sensor network. The directory *sensornet* has been created by a discovery process resulting in a sub-directory for each sensor. Each sub directory contains file entries *reading*, *status*, and *ctl*. Writing the command “program 1” to a sensor’s *ctl* file causes local binary file *image1* to be loaded into that sensor.

```
for(c in /sensornet/cluster*) {
  cd $c
  cat /sensorapps/RBlink/main.ihex > image1
  cat /sensorapps/BBlink/main.ihex > image2
  for (s in $c/RedSensors/s*) {
    echo 'program 1' > $$s/ctl
  }
  for (s in $c/BlueSensors/s*) {
    echo 'program 2' > $$s/ctl
  }
}
```

Since we are aiming for language independence in our work, we are more concerned with conceptual methods than their codification in a particular language. Support for methodology involves three interdependent aspects: design modeling, implementation synthesis, and infrastructure support. Although these are often packaged in a language syntax, compilation parameters, a custom design environment, and even specialized hardware, our goals of language and platform independence are better served by a collection of coherent design concepts, programming conventions (templates, invariants, obligations, etc.), and services.

Design Model The fundamental design abstraction is that of a task or function that takes all its inputs instantaneously at the beginning of its cycle, and stores all its results instantaneously at the end. In other words, it is the classical *process* model, $\langle ns: I \times S \rightarrow S; out: I \times S \rightarrow O \rangle$, consisting of *next-state* (*in*) and *output* (*out*) functions operating on global inputs (*I*), and internal state (*S*) to produce global outputs (*O*). In time, the sequential semantics are given by

$$\begin{aligned} O_{k+1} &= out(I_k, S_k) \\ S_{k+1} &= ns(I_k, S_k) \end{aligned}$$

with initial values O_0 and S_0 derived from process creation parameters. A *system* is the concurrent composition of *N* such processes in a connection network associating each input I_k with a subset of the O_k s. Local states, *S* are non-observable. For a process *component* $C = \langle ns, out \rangle$, let the recursive defining equation

$$[O, S] = C(I, S)$$

stand for the equations above. Then a *synchronous system* *S* is expressed as the simultaneously recursive system of defining equations

$$\begin{aligned} S(I_0) = \vec{O} \text{ where } [O_1, S_1] &= C_1(\vec{I}_1, S_1) \\ &\vdots \\ [O_n, S_n] &= C_n(\vec{I}_n, S_n) \end{aligned}$$

In which $\vec{O}$ and all $\vec{I}_k$ are composed from subsets of $\{I_0, O_1, \dots, O_k\}$. The design problem is to decompose a higher-level specification into a hierarchy of systems, that is, *S*, too, is a component with state $\vec{S}$ and output $\vec{O}$. The term *component*, introduced above, corresponds to the *component* entities developed in later sections.

Implementation Model A collection of conventions and techniques for *systematically* translating design models into programs. The implementation model reflects underlying mechanisms for process coordination and the target architecture, ERTS in this paper. In our work, these conventions often take the form of pseudo-code templates and global invariants such as the *SWMR* constraint defined in the next section.

Realization Support The term *realization* refers to actual design artifacts, such as binary program objects, and the system infrastructure over which they execute. Here, that infrastructure includes the file system interface already described, as well as extensions supporting synchronous execution. The sequel describes infrastructure that is inspired by synchronous-digital hardware. The synchronizing event is the *tick* of a global clock, at which moment all components simultaneously update their local state. Assuming the Implementer can assure that all tasks execute in the period between ticks, there are no race conditions and the system state is observably valid at known times.

Kopetz, et. al implement an *Interface File System* (IFS) over the TTP/A, their time-triggered protocols [8, 9]. Their motivation has much in common with our own.

IMPLICATIONS FOR PRACTICE As just discussed, ERTS/SyncFS targets academic research applications and its peculiar needs for fast start-up, mixed expertise, and small development teams. Although this profile does not conform to a large-scale production environment, we believe the approach is quite attractive for rapid prototyping, component-ware testing, and advanced design.

OUTLINE The remainder of this article is in three parts. In the next section we describe the *SyncFS* file system. We then present a Design Example to illustrate the ideas of this work. The example is a slice of the ERTS control system that manages vehicle steering. In the Conclusion we present some preliminary performance studies and discuss future directions of the work.

SYNCFs

SyncFS is a file system with synchronous, system-wide write commits. The synchronizing element is modeled on the functionality of globally clocked D-type flip-flop. A file's writer may make successive changes to a file without effecting its visible content. On the server's virtual clock tick, the writer's version is "latched," and only then does the new content become visible to the file's readers.

We begin with a brief sketch of the ERTS vehicle architecture in which SyncFS is embedded. After that, key design aspects of SyncFS are described. In the following section, we make these ideas more concrete by presenting a design example.

ERTS Architecture Figure 3 is a high level diagram of ERTS physical architecture. It typical of modern embedded architectures as described by Kopetz [2]. An on-board local-area network (LAN) of Linux compute nodes (CNODEs) hosts a higher-level control system configured to present the vehicle at the appropriate level of abstraction for the experimental "navigator" (NAV). One CNODE