Designing Software Components for Real-Time Applications

Class 421/431
Embedded Systems Conference Chicago
June 2002

Dave Stewart
Chief Technology Officer
dstewart@embedded-zone.com
Designing Software Components for Real-Time Applications

Class 421
Introduction to Reconfigurable Software Components

Class 431
Details for Creating your own Framework
Class 421
Introduction to Reconfigurable Software

- What is reconfigurable software?
- Motivation
- Model of a component
- Configuring components
- Overview of framework
- Subsystem decomposition
- Summary of Class 421
What is Reconfigurable Software?

*Highest Degree of Modularity*

+ Replacement Independence

i.e. A Building Block
Reconfigurable Software:
Not a Far-Fetched Solution ...
Just good Software Engineering.

Reconfiguration is necessary so that non-experts can create their own applications, by configuring the infrastructure provided.

Difference in real-time systems is that there are also timing constraints.
Static vs. Dynamic

- **Static Configurability**
  - Off-line Integration of Components

- **Dynamic Reconfigurability**
  - Run-time Activation and Deactivation of Components
Reconfigurable Software Components for Embedded Systems

Does it make sense?

Yes, it DOES make sense!

- Rapid development through component-based design
- Hardware/software co-design
- Tele-configuration
- Changing software “on-the-fly”
- Evolutionary design
- Product lines
- Flexible fine-tuning
- Increased reliability through automated analysis
Reasons for NOT using Software Components

Nothing in this world except software is manufactured without a building-block approach!
Software Components for Embedded Systems

Software components have successfully been used on the smallest systems (e.g., intelligent MEMS sensors with 1 MHz processor, less than 1K RAM) to the largest systems (e.g., Airline Reservation Systems).

The decision is not whether components should be used. Rather, the choice is what approach to use.

The port-based-object approach presented here is suitable for small systems (8-bit processor, 1K RAM or more, 16K ROM or more) to medium-sized self-contained distributed systems (i.e., one organization oversees all the software in the distributed network).

While the approaches, including models, architecture, and implementation, may change, the fundamental methodology of using building blocks that plug into each other remains the same across all approaches.
Today ...

@ Embedded Systems Conference

How to design and implement your own building blocks

You don’t need expensive tools
You don’t need years of experience
You don’t need a graduate degree

You just need to know a few techniques you’ll learn today

● ● ● and the discipline to design before you implement

There exist many CASE tools that aid in creating software components. In this presentation, we focus on the techniques, not the tools.
Today ... Two Versions, One Interface

- **Non- and Limited Preemption**
  - Implement directly on any processor, in place of an RTOS
  - Non-preemptive dynamic EDF scheduling, or
  - Limited-preemption mixed-priority MUF scheduling
  - Best suited for 8/16-bit microcontrollers, and fixed and floating point digital signal processors

- **Fully Preemptive**
  - Implement as a layer above a commercial RTOS
  - Full preemption, RMA, EDF, or MUF scheduling
  - Best suited for 32-bit microcontrollers and non-real-time multitasking environments

*The Application Programmer Interface (API) is the same for all versions on all processors!*  
*Only the framework internals change, and those changes are transparent to the programmer.*
Model of a Software Component

We introduce the concept of a

Port-Based Object

which combines

- **Object-Based Design**
  - for addressing reusability and configurability issues
    - (Wegner, 1990)
    - data encapsulation with access via its own methods only
    - standard interface to each class of objects
    - Implement objects as Abstract Data Types (ADTs) in C

- **Port-Automaton Theory of Concurrent Processes**
  - for addressing real-time and control system issues
    - (Streenstrup, Arbib, and Manes, 1983)
    - all processes are independent; communication with other processes only through input/output ports.
    - process is unaware of source of input or destination of output.
Port-Based Object

Two Aspects

Data-Oriented Aspect

Port-Based Object

k₁ · · · kᵣ

r₁ · · · rᵣ

Code-Oriented Aspect

pboFrame(pbo) {
    pbo
}

PBO Framework
Data-Oriented Aspect

Configuration Constants
\[ k_1 \cdots k_p \]

Input Ports
\[ x_1 \cdots x_m \]

port-based object

Output Ports
\[ y_1 \cdots y_m \]

Resource (I/O) Ports
\[ r_1 \cdots r_p \]
Sample Library of Software Components for Robotic Control

Generic Components

- Generalized Forward Kinematics
  - Parameters: \( n_{dof} \), \( dh \), \( \theta_m \)
  - Function: \( \text{gfwdkin} \)
  - Output: \( x_m \)

- Generalized Inverse Kinematics
  - Parameters: \( n_{dof} \), \( dh \), \( x_r \)
  - Function: \( \text{ginvkin} \)
  - Output: \( \theta_m \)

- Cartesian Trajectory Interpolator
  - Parameters: \( x_m \), \( x_d \)
  - Functions: \( \text{cinterp} \)
  - Outputs: \( v_r \), \( x_r \)

Device Drivers

- Reconfigurable Modular Robot
  - Parameters: \( n_{dof} \), \( dh \), \( \theta_m \), \( \omega_m \)
  - Function: \( \text{rmms} \)

- Puma 560 Robot
  - Parameters: \( n_{dof} = 6 \), \( dh = K \)
  - Function: \( \text{puma} \)

- 6-dof Trackball
  - Function: \( \text{tball} \)
  - Output: \( x_d \)

Hardware-Dependent Computational Components

- Puma 560 Forward Kinematics
  - Parameters: \( \theta_m \)
  - Function: \( \text{pfwdkin} \)
  - Output: \( x_m \)

- Puma 560 Inverse Kinematics
  - Parameters: \( x_r \)
  - Function: \( \text{pinvkin} \)
  - Output: \( \theta_m \)
Sample Configuration
Teleoperated Cartesian Control of RMMS Robot

- Sample Configuration
- Teleoperated Cartesian Control of RMMS Robot
- Diagram showing the process of controlling the robot with inputs from sensors and outputs to actuators.
Sample Configuration
Teleoperated Cartesian Control of **Puma Robot**

Class 421/431
Embedded Systems Conference
June 2002

Dave Stewart
Embedded Research Solutions, LLC
www.embedded-zone.com

© 2002 Dr. David B. Stewart
Sample Configuration

Optimized Teleoperated Control of Puma Robot

- \( \theta_m \)
- \( n_{dof}=6 \)
- \( \omega_m \)
- \( \theta_r \)
- \( \omega_r \)
- \( \theta_m \)
- \( \omega_m \)

Diagram:

- From trackball
- From sensors to actuators
- \( x_m \)
- \( x_d \)
- \( v_r \)
- \( x_r \)
- \( pfwdkin \)
- \( pinvkin \)
- \( puma \)
Sample Configuration

Autonomous Cartesian Control of Puma Robot

Sample Configuration

Autonomous Cartesian Control of Puma Robot

Sample Configuration

Autonomous Cartesian Control of Puma Robot

Sample Configuration

Autonomous Cartesian Control of Puma Robot
Sample Configuration
Autonomous Cartesian Control of RMMS Robot

Class 421/431
Embedded Systems Conference
June 2002
Sample Configuration
Traditional Implementation of Device Drivers

Diagram showing the flow of data from sensors to actuators through device drivers and kinematics modules.
Traditional Process-Flow Model of a Device Driver

- Control Algorithm
  - sensor data
  - actuator output
  - data filter
    - raw data
    - read
      - xxread
    - cmd filter
      - raw data
      - write
      - xxwrite
  - Device driver
  - I/O filtering
  - RTOS driver interface

Designing Real-Time Software for Real-Time Applications

Class 421/431
Embedded Systems Conference
June 2002

© 2002 Dr. David B. Stewart
Instead: **Data-Driven Model of a Device Driver**

Fewer layers leads to better performance and lower memory utilization. Data-driven model provides more flexibility for creating a hardware-independent interface.
Sample Configuration
Device Drivers are simply PBOs or pbos

Since the device driver is itself a task, it can be scheduled to read or write the devices in a predictable manner.
Interfacing with I/O Processors

- Code executing on the main processor
  - Control Algorithm
  - Shared State Data

- I/O Processor Device Driver
  - sensor data → sensor
  - actuator output → actuator
  - I/O electronics
Sample Configuration
Hardware-in-the-loop Simulation of RMMS Robot

Class 421/431
Embedded Systems Conference
June 2002

Dave Stewart
Example (both diagrams on same page)

Dynamic Reconfiguration of Control Algorithm

Diagram 1:
- `ctraj` to `cinterp` to `vr`
- `x_m` to `x_d`
- `pfwdkin + Jacobian` to `pinvkin`
- `dh` to `n_dof`
- `n_dof=6`
- `dh=K`

Diagram 2:
- `ctraj` to `cinterp` to `vr`
- `x_m` to `x_d`
- `pfwdkin + Jacobian` to `damped least squares` to `integrator`
- `dh` to `n_dof`
- `n_dof=6`
Reconfigurable Software Enables Visual Programming
(Example)
Reconfigurable Software Enables User Monitoring
(Example)
Do-it-yourself framework vs. Commercial Tools

- Do the entire framework yourself if
  - You need to minimize or eliminate licensing costs
  - Operating system overhead is a critical concern
  - Lower-end (8/16-bit) microcontrollers or DSP (e.g. < 30 MHz, < 64 KBytes of memory)
  - Non-preemptive or limited-preemptive scheduling is okay
  - Simpler applications

- Use a commercial RTOS but create your own framework if
  - faster 32-bit processors with lots of memory (e.g. > 30 MHz, > 1 MByte Memory)
  - A good RTOS that meets your needs is available
  - Fully preemptive scheduling is necessary
  - Licensing costs (if any) are acceptable

- Use commercial CASE tool with RTOS if
  - Complex applications
  - You also want/need visual programming capabilities
  - The tools you select provide a complete framework, that includes patterns/templates for reconfigurable software and device driver support

Any of these methods can lead to quality reconfigurable software.
Overview of Code-Oriented Aspect

Traditional Real-Time Module

```c
main() {
    POSIX
    RT Thread
}
```

Timing, communication, and synchronization under control of user

⇒ *less predictable*

User Code

Port-Based Object Module

```c
pboFrame(pbo) {
    pbo
}
```

Timing, communication, and synchronization under control of RTOS or Framework

⇒ *more predictable*

RTOS Code
Reconfigurable Port-Based Objects Plug-In to Framework

pboFrame(*pbo) {

PBO Framework

}

pbo
Details of implementing both Code-Oriented and Data-Oriented Aspects in Part II of Class
Decomposing of a Subsystem into Components
A mini-tutorial
Step 1: Create one module for each I/O element

For example, cruise control.

Cruise control is a textbook example for many software engineering courses. Thus, for comparison with other proposed techniques, we use that same example to demonstrate our method for software decomposition.
Step 1: Create one module for each I/O element

Draw modules as follows:

- **Input module**
  - Sensor

- **Output module**
  - Actuator
Step 1: Create one module for each I/O element

- accel
  - accelerator pedal
- brake
  - brake pedal
- ccui
  - cruise control
  - user interface
- position
  - wheel position
  - resolver
- engine
  - force to
  - engine
- wheels
  - force to
  - wheels
- dashboard
  - speedometer
  - odometer
Step 2: Determine *input ports* of output components

\[ f_{de} = \text{desired acceleration force to apply to engine} \]

\[ f_{dw} = \text{desired deceleration force to apply to wheels} \]

\[ v_z = \text{measured velocity (for speedometer)} \]

\[ x_z = \text{measured position (for odometer)} \]
Step 3: Define Computational Modules

control maintains a single force value

split-pm is a generic module that splits input into separate +/- components on output.

\[ f_d = \text{desired force} \]
Step 4: Define Inputs to Computational Modules

$f_{ra}$ = Reference acceleration (e.g. gas pedal)
$f_{rb}$ = Reference deceleration (e.g. brake pedal)
$U_r$ = User command (e.g. cruise ‘on’)
$v_z$ = Measured velocity
$v_r$ = Reference velocity (i.e. cruise control velocity to maintain)
Diagram redrawn with input modules shown again

- accel (accelerator pedal)
- brake (brake pedal)
- ccui (cruise control user interface)
- position (wheel position resolver)

- engine
  - force to engine

- split-pm
  - +
  - -

- control
  - f_d
  - U_r
  - v_z
  - v_r

- wheels
  - force to wheels

- dashboard
  - speedometer
  - odometer

- fd
- f_{ra}
- f_{rb}
- f_{d_1}
- f_{d_2}
- f_{d_3}
- f_{d_4}
- f_{d_5}
- f_{d_6}
- f_{d_7}
- f_{d_8}
- f_{d_9}
- f_{d_{10}}
Step 5: Define source of computational module inputs

accel
- accelerator pedal
  f_{ra}

brake
- brake pedal
  f_{rb}

ccui
- cruise control user interface
  U_r

control

position
- wheel position resolver
  x_z

split-pm

engine
- force to engine
  f_{de}

wheels
- force to wheels
  f_{dw}

dashboard
- speedometer, odometer
  v_z

vr

??

v_z

??
Step 6: Add modules to generate variables if necessary

Measured velocity is time derivative of measured position
Sometimes extra thought is needed to determine the source of a specific variable.

When the set command arrives to control module from $U_r$ variable, remember the current measured velocity, $v_z$, and that becomes $v_r$. 
We now have a *first* draft.
Step 7: Review, and make corrections, adjustments, etc.

If brake pedal is pressed, immediately turn off cruise control.
Step 7: Review, and make corrections, adjustments, etc.

Other typical adjustments:

- Combine two modules into one
- Split one module into two
- Create a new module that can replace an existing one
- Add an input or output item
- Add a module to perform monitoring and/or data logging
- Minimizing number of modules that need interrupt handlers
- Create a simulation module to simulate an I/O element that is not currently available.
Step 8: Repeat decomposition for large modules.

For example

Using this decomposition method (in a larger application), the box in the middle can look like:

Repeat the decomposition from Step 1, assuming w, x, y, z were output ports, and a, b, c, d, e were input ports, and create a module to generate each.
Step 9: Preliminary Real-Time Analysis

- **Estimate the following:**
  - Periodic or aperiodic
  - Frequency or period
  - Worst-case execution time for a variety of available processors
  - If a module needs an interrupt handler, minimum interarrival time of interrupt

- **Use real-time analysis to determine following:**
  - Do you need pre-emptive system, or is non-preemptive or limited-preemptive okay?
  - Static scheduling algorithm or dynamic scheduling algorithm?
  - For which scheduling algorithm/processor(s) is the task set schedulable?
  - Assume 10% increase in execution time, is task set still schedulable? Repeat for 20%, 30%, etc., to determine range of flexibility in case estimated execution time is wrong.

Avoid the temptation to start implementing immediately. Do not start implementation until you know that the task set is schedulable on the processor you choose, preferably with some flexibility to increase execution time.
Class 421
Summary of Part I

- What is reconfigurable software?
- Motivation
- Model of a component
- Configuring components
- Overview of framework
- Subsystem decomposition
Designing Software Components for Real-Time Applications

Class 421/431
Embedded Systems Conference Chicago
June 2002

Dave Stewart
Chief Technology Officer
dstewart@embedded-zone.com
Designing Software Components for Real-Time Applications

Class 421
Introduction to Reconfigurable Software Components

Class 431
Details for Creating your own Framework
Class 431
Details for Creating your own Framework

- Brief Review of Class 421
- What you need to create your own framework
  - Preemptive vs. Non-preemptive frameworks
  - Code-Oriented Aspect
  - Data-Oriented Aspect
  - Real-Time Issues
- Summary of Classes 421/431
Review ... Port-Based Object

Two Aspects

Data-Oriented Aspect

Code-Oriented Aspect
Review: Data-Oriented Aspect

\[ \theta \]
\[ m \]
\[ x_d \]
\[ v_r \]
\[ \theta_r \]
\[ \omega_m \]
\[ n_{dof} \]
\[ dh \]
from sensors
to actuators

from trackball

\[ x_m \]
\[ dm \]
\[ gfwdkin \]
\[ cinterp \]
\[ ginvkin \]
\[ rmms \]
Review: Code-Oriented Aspect

Traditional Real-Time Module

```c
main() {
    POSIX RT Thread
}
```

Timing, communication, and synchronization under control of user ➞ less predictable

User Code

Port-Based Object Module

```c
pboFrame(pbo) {
    PBO Framework
}
```

Timing, communication, and synchronization under control of RTOS or Framework ➞ more predictable

RTOS Code
Today ... Two Versions, One Interface

- **Non- and Limited Preemption**
  - Implement directly on any processor, in place of an RTOS
  - Non-preemptive dynamic EDF scheduling, or
  - Limited-preemption mixed-priority MUF scheduling
  - Best suited for 8/16-bit microcontrollers, and fixed and floating point digital signal processors

- **Fully Preemptive**
  - Implement as a layer above a commercial RTOS
  - Full preemption, RMA, EDF, or MUF scheduling
  - Best suited for 32-bit microcontrollers and non-real-time multitasking environments

*The Application Programmer Interface (API) is the same for all versions on all processors!*

*Only the framework internals change, and those changes are transparent to the programmer.*
Reconfigurable Port-Based Objects Plug-In to Framework
Reconfigurable Port-Based Objects Plug-In to Framework (non-preemptive version)
Reconfigurable Port-Based Objects Plug-In to Framework (preemptive version)

Replicate PBO Framework once for each PBO
Implementing the Framework: Core Functions

For Preemptive Versions with RTOS

- **pbo-p**: Port-based object framework, preemptive environment
- **svar-p**: Two-level shared memory structure, for preemptive systems

For Non-preemptive Versions without RTOS

- **sched**: Non-preemptive or limited-preemption scheduling
- **pbo-n**: Port-based object framework, non-preemptive environment
- **svar-n**: One-level shared memory structure, for non-preemptive systems
- **etime**: For timing purposes

**Framework Process**

- **pbo-p**: Port-based object framework, preemptive environment
- **pbo-n**: Port-based object framework, non-preemptive environment

**State Variable Communication**

- **svar-p**: Two-level shared memory structure, for preemptive systems
- **svar-n**: One-level shared memory structure, for non-preemptive systems

**Timing and Scheduling Functions**

- **RTOS**: Real-time operating system with preemptive scheduler
- **sched + etime**: Non-preemptive or limited-preemption scheduling
Implementing the Code-Oriented Aspect

Framework Process
- pbo-p: Port-based object framework, preemptive environment
- pbo-n: Port-based object framework, non-preemptive environment

State Variable Communication
- svar-p: Two-level shared memory structure, for preemptive systems
- svar-n: One-level shared memory structure, for non-preemptive systems

Timing and Scheduling Functions
- RTOS: Real-time operating system with preemptive scheduler
- sched + etime: Non-preemptive or limited-preemption scheduling
PBO Framework
Preemptive Version

States of PBO
- OFF
- ON
- NOT CREATED

Synchronization
- spawn
- sync
- interrupt
- cycle
- external signal
- timer
- periodic
- aperiodic

User-defined methods
- init
- term
- off
- on
- >in-vars
- >in-consts
- >out-vars
- >out-consts
- >in-vars
- >out-vars

Communication (preemptive version only)
- wakeup
- term
- off
- on
Designing Real-Time Software for Real-Time Applications

PBO Framework
Preemptive Version

- States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)

Class 421/431
Embedded Systems Conference
June 2002

© 2002 Dr. David B. Stewart
States of PBO
- Not Created
- On
- Off

Synchronization
- Term
- Wakeup

User-defined methods
- Spawn
- Cycle

Process flow/transition
- Term
- Off

Communication (preemptive version only)
- External signal
PBO Framework
Non-Preemptive Version

- States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)
States of PBO

- Synchronization
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)

**PBO Framework**

**Both Versions**

Class 421/431
Embedded Systems Conference
June 2002

© 2002 Dr. David B. Stewart
PBO Framework
Preemptive Version Only

- States of PBO
- Synchronization
- User-defined methods
- Process flow/transition

Communication (preemptive version only)
The Main Loop: Periodic Process

States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)
The PBO Framework

States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)

The Main Loop: Aperiodic Servers

- PBO Framework
- States of PBO
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)
PBO Framework
Both Versions

States of PBO
- Synchronization
- User-defined methods
- Process flow/transition

Communication (preemptive version only)
Deactivation and Termination

States of PBO
- Synchronization
- User-defined methods
- Process flow/transition

Communication (preemptive version only)
Automatic Generation of API via C Macro

typedef int (* pboFunc_f)(void *);

typedef struct {
    pboFunc_f init;
    pboFunc_f on;
    pboFunc_f cycle;
    pboFunc_f sync;
    pboFunc_f off;
    pboFunc_f term;
} pboFunc_t;
Automatic Generation of API via C Macro

carFunc

pboFunc_t carFunc = {
    carInit,
    carOn,
    carCycle,
    carSync,
    carOff,
    carTerm,
};

typedef int (* pboFunc_f)(void *);

typedef struct {
    pboFunc_f init;
    pboFunc_f on;
    pboFunc_f cycle;
    pboFunc_f sync;
    pboFunc_f off;
    pboFunc_f term;
} pboFunc_t;
Automatic Generation of API via C Macro

typedef int (* xyz##Func_f)(xyz##_t *);

xyz##Func_f xyz##Init

define struct {
    pboFunc_f init;
    pboFunc_f on;
    pboFunc_f cycle;
    pboFunc_f sync;
    pboFunc_f off;
    pboFunc_f term;
} pboFunc_t;

xyz##Func_f xyz##Init

typedef int (* pboFunc_f)(void *);

xyz##Func_t xyz##Func = {
    xyz##Init,
    xyz##On,
    xyz##Cycle,
    xyz##Sync,
    xyz##Off,
    xyz##Term,
};
Automatic Generation of API via C Macro

PBO_MODULE(car)

int xyz##Cycle(xyz##_t *loc)
int carCycle(car_t *loc)

pboFunc_t

init  carInit
don  carOn
cycle  carCycle
sync  carSync
off  carOff
term  carTerm

xyz => car

xyz##Func_t xyz##Func = {
  xyz##Init,
  xyz##On,
  xyz##Cycle,
  xyz##Sync,
  xyz##Off,
  xyz##Term,
};

typedef int (* pboFunc_f)(void *);
typedef struct {
  pboFunc_f  init;
  pboFunc_f  on;
  pboFunc_f  cycle;
  pboFunc_f  sync;
  pboFunc_f  off;
  pboFunc_f  term;
}  pboFunc_t;
```c
typedef int (* pboFunc_f)(void *);

typedef struct _ pboFunc_t {
    pboFunc_f  init;
    pboFunc_f  reinit;
    pboFunc_f  on;
    pboFunc_f  cycle;
    pboFunc_f  sync;
    pboFunc_f  off;
    pboFunc_f  term;
} pboFunc_t;

#define   PBO_MODULE(xyz)\n    typedef int (* xyz##Func_f)\n            (xyz##_t *);
    int xyz##Init(xyz##_t *);
    int xyz##Reinit(xyz##_t *);
    int xyz##On(xyz##_t *);
    int xyz##Cycle(xyz##_t *);
    int xyz##Sync(xyz##_t *);
    int xyz##Off(xyz##_t *);
    int xyz##Term(xyz##_t *);
}\n
typedef struct _##xyz##Func_t {
    xyz##Func_f        init;
    xyz##Func_f        reinit;
    xyz##Func_f        on;
    xyz##Func_f        cycle;
    xyz##Func_f        sync;
    xyz##Func_f        off;
    xyz##Func_f        term;
} xyz##Func_t;

const xyz##Func_t xyz##Func = {
    xyz##Init,\n    xyz##Reinit,\n    xyz##On,\n    xyz##Cycle,\n    xyz##Sync,\n    xyz##Off,\n    xyz##Term\n};

// end   PBO_MODULE defintion
```
Sample User Code: Module car

```c
#include <etime.h>
#include <svar.h>
#include <pbo.h>

typedef struct {
  pbo_t *pbo; // pointer to pbo info
  // pointers to global SVARs go here
  bool_t *user_ref;
  int16_t *v_mez;
  int16_t *v_ref;
  uint16_t *f_direct;
  // variables for internal states go here
  etimeAbs_t time_new;
  etimeAbs_t time_old;
} car_t;

PBO_MODULE(car);

int carInit(pbo_t *pbo) {
  car_t *car = pbo->car;
  car->pbo = pbo; // remember pbo pointer
  // translate global state variable names
  // to local pointers
  car->user_ref = svarXlate(
    &pbo->svartable, "user_ref", bool_t);
  car->v_mez = svarXlate(
    &pbo->svartable, "v_mez", int16_t);
  car->v_ref = svarXlate(
    &pbo->svartable, "v_ref", int16_t);
  car->f_direct = svarXlate(
    &pbo->svartable, "f_direct", uint16_t);
  return 0;
}

int carOn(car_t *car) {
  *car->f_direct = 0;
  *car->v_ref = 0;
  etimeClock(&car->time_new);
  return 0;
}

int carCycle(car_t *car) {
  bool_t *user_ref = car->user_ref;
  int16_t *v_mez = car->v_mez;
  int16_t *v_ref = car->v_ref;
  uint16_t *f_direct = car->f_direct;
  etimeAbs_t *time_new = &car->time_new;
  etimeAbs_t *time_old = &car->time_old;
  *time_old = *time_new; // save old time
  etimeClock(time_new); // get new time
  // calculate force needed to maintain ref velocity if cruise is enabled
  if (*user_ref == FALSE) { // cruise disabled
    *f_direct = *v_mez /
      etimeDeltaT(*time_new, *time_old);
    *v_ref = *v_mez;
  } else { // cruise enabled
    *f_direct = (*v_ref - *v_mez)
      / etimeDeltaT(*time_new, *time_old);
  }
  return 0;
}

// rest of module not included on slide
```
Implementation of Main Loop for PBO-P
(Preemptive Version)

Execute each PBO as a thread, by binding every thread to the function `pboFrame()`, and passing the PBO as an argument.

Note: This is pseudocode. Some auxiliary functionality is not listed, such as function return values and checking for change of state.
Recall: Plugging `carCycle` into cycle socket of framework

```c
typedef int (* pboFunc_f)(void *);

typedef struct {
  pboFunc_f init;
  pboFunc_f on;
  pboFunc_f cycle;
  pboFunc_f sync;
  pboFunc_f off;
  pboFunc_f term;
} pboFunc_t;

xyz##Func_t xyz##Func = {
  xyz##Init,
  xyz##On,
  xyz##Cycle,
  xyz##Sync,
  xyz##Off,
  xyz##Term,
};
```
Recall: only a single framework for PBO-N version
Implementation of Main Loop for PBO-N (non-preemptive version)

while (1) {
    do {
        etimeClock(&now);
        // Move newly awoken tasks from PauseQ to ReadyQ
        while (!Empty(pauseQ) &&
                etimeCompare( pboSchedTime(pauseQ),now) <= 0) {
            pboDequeue(&pauseQ, & pbo);
            etimeAdd(& pbo->schedtime, pbo->period);
            pboEnqueue(&readyQ,  pbo);
        }
    } while (readyQ.next == NULL);

    // Head of readyQ has earliest-deadline task
    pboDequeue(&readyQ, & pbo);

    pbo->func->cycle( pbo);
    if ( pbo->tasktype ==  pbo_PERIODIC)
        pboEnqueue(&pauseQ, pbo);
    else
        pbo->func->sync( pbo);
}

Note: This is pseudocode. Some auxiliary functionality is not listed, such as function return values and checking for change of state.
Implementing the Data-Oriented Aspect

For Preemptive Versions with RTOS

- pbo-p
- svar-p
- RTOS

For Non-preemptive Versions without RTOS

- sched
- pbo-n
- svar-n
- etime

Framework Process
- pbo-p: Port-based object framework, preemptive environment
- pbo-n: Port-based object framework, non-preemptive environment

State Variable Communication
- svar-p: Two-level shared memory structure, for preemptive systems
- svar-n: One-level shared memory structure, for non-preemptive systems

Timing and Scheduling Functions
- RTOS: Real-time operating system with preemptive scheduler
- sched + etime: Non-preemptive or limited-preemption scheduling
State Variable (SVAR) Communication

Framework of a Port-Based Object

- States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication (preemptive version only)
SVAR Communication is Based on Shared Memory

Multiprocessor Version

Global State Variable Table

Single-processor

Preemptive

Non-

Global State Variable Table

PBO A_1

PBO A_j

PBO K_1

PBO K_j

local table

local table

local table

local table
SVAR for Multiprocessor and Preemptive Systems

Transfers between local/global table can be via `memcpy()`, DMA, or network. Appropriate non-blocking synchronization (e.g. spin-locks, test-and-set, cpu-locking, etc.) is needed.

**New hardware/software co-design results:** A pre-programmed DMA mechanism can eliminate most overhead of this mechanisms.
SVAR for Non-Preemptive Systems

No need for a local table, as without preemption, there is no contention for the global table.
Preemption for a limited number of tasks can be achieved in a non-preemptive system by using interrupts. Two cases:

One or two very fast tasks: The cycle routine is installed as an interrupt handler. Updates between local and global table occurs whenever the scheduler is called.

One or two very slow tasks: Only the slow tasks can be preempted. Additional overhead for data copying, but since these tasks are slow, overhead is not incurred very often. Use maximum-urgency-first scheduling to control which tasks can be preemted.
SVAR Communication vs. Message Passing

Why Not Message Passing?

- All processes are usually asynchronous and aperiodic, making real-time analysis nearly impossible.
- If messages are synchronous, then everything must execute at same frequency, reducing flexibility
- Undesired blocking due if receive buffer is empty
- Crucial messages can get lost due to buffer overflow
- High overhead of sending and receiving messages
- Possible deadlock in feedback systems
- Need to replicate messages if fan-out != 1.
- High overhead re-binding message queues during dynamic reconfiguration.
- Does not take advantage of performance gains available with shared memory in non-preemptive environments
Most messages can be converted to states

Using Messages, “Apply Brake”:

```
SEND ("apply brake")
```

Using States, “The Brakes Should be ON”:

```
IF brake == ON
activate brake
ELSE
brake 2
```

```
IF brake == ON
activate brake
ELSE
brake 1
```
SVAR Communication vs. Regular Shared Memory

Why Not Regular Shared Memory or Global Variables?

- Shared data must be accessed as critical sections to preserve the integrity of the data.
- Semaphores and similar guards against race conditions may cause blocking, that can lead to priority inversion.
- Solutions such as priority ceiling protocol which address priority inversion use significant overhead (as much or more than message passing).
- Multiprocessor extensions to priority ceiling protocol are extremely costly, and not suitable for most embedded systems.
- Not extensible to multiprocessor embedded systems that use networks (e.g. CANbus).
- Global variables cannot be re-bound to provide dynamic reconfiguration.
- SVAR abstraction allows transparent implementations for different environments (e.g. preemptive vs non-preemptive).
Hybrid Communication Frameworks

Data-Oriented Aspect

- State communication for small volumes of data
- Stream communication for large volume of data
  - Circular buffers for synchronous streams
  - Triple-buffer mechanism for asynchronous streams
Hybrid Communication Frameworks
Code-Oriented Aspect

```c
pboFrame(*pbo) {
    pco->func->read(pbo->invar);
    pbo->func->cycle(pbo->local);
    pco->func->write(pbo->outvar);
}
```
Implementing the Framework: Core Functions

For Preemptive Versions with RTOS

- **pbo-p**: Port-based object framework, preemptive environment
- **svar-p**: Two-level shared memory structure, for preemptive systems

For Non-preemptive Versions without RTOS

- **sched**: Real-time operating system with preemptive scheduler
- **pbo-n**: Port-based object framework, non-preemptive environment
- **svar-n**: One-level shared memory structure, for non-preemptive systems
- **etime**: Non-preemptive or limited-preemption scheduling

**Framework Process**
- **pbo-p**: Port-based object framework, preemptive environment
- **pbo-n**: Port-based object framework, non-preemptive environment

**State Variable Communication**
- **svar-p**: Two-level shared memory structure, for preemptive systems
- **svar-n**: One-level shared memory structure, for non-preemptive systems

**Timing and Scheduling Functions**
- **RTOS**: Real-time operating system with preemptive scheduler
- **sched + etime**: Non-preemptive or limited-preemption scheduling
Real-Time Scheduling, Periodic Tasks

To avoid clock skew, do not use sleep()
Timing Failure Detection and Handling (TFH)

Hard real-time systems are often implemented with the dangerous assumption that a timing error will never occur!

Start

INIT

tfhInstall(handler,priority)

set RESTART point

pause()

CYCLE

MDEADLINE

msgSend()

RESTART

MAXEXEC

handler

lower priority

CONTINUE
Implementing Aperiodic Servers using TFH

TFH handlers can be used to implement Deferrable or Sporadic Servers (Ref. Sprunt, 1990). Define WCET as capacity, and replenishment time as deadline.

Also available at http://www.embedded-zone.com
Timing Issues: Preemptive vs. Non-preemptive

- **Preemptive version:**
  - System clock at least at rate of fastest process
  - Too much interrupt handling overhead
  - At best, 5 to 10 msec periods on 16-bit processor or 1 msec on 32-bit processor
    - Cannot schedule a task at 150 or 400 Hz!
  - Round-off errors in period due to limited granularity of system clock

- **Non-preemptive version:**
  - Cannot schedule task set if there is one really fast task, or one really slow task that uses a significant percentage of CPU time.
  - Requires more analysis ahead of time to ensure good scheduling

- **Limited preemption:**
  - The best of both worlds from a scheduling perspective
  - More complicated to implement
Real-Time Scheduling

- **Preemptive: RM, EDF, or MUF**
  - Scheduling algorithm generally provided by RTOS
  - Can use any proven RT algorithm for preemptive systems; the PBO model doesn’t restrict the type of scheduling

- **Non-Preemptive: EDF or Cyclic**
  - Most timer chips let you read countdown value in timers.
    - Use that to increase resolution while slowing down the timer.
  - E.g. Set clock tick to between 1 and 100 usec, and interrupt only once every 50 to 200 msec;
  - Flexibility to trade-off time granularity for period range
    - minimal difference on interrupt overhead
  - Works in any non-preemptive environment, since interrupt from clock is not needed to preempt tasks.

- **Limited Preemption: MUF**
  - Fixed criticality distinguishes between tasks that can preempt others, tasks that can be preempted, and tasks that are non-preemptive among other non-preemptive tasks.

RM=Rate Monotonic, EDF=Earliest Deadline First, MUF=Maximum Urgency First
Cyclic=Fixed Priority Cyclic Real-Time Executive
Reconfigurable Software has definitive start and stop points for each process, making it easier to measure the actual execution time.
Measuring Execution time

Start - End

Instrumentation points to measure WCET.

Start - End

Instrumentation points to measure communication time.

Start - End

Instrumentation points to measure overhead.

- States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication

Class 421/431
Embedded Systems Conference
June 2002
Automated Data Logging

Instrumentation point to log data

log outvar

States of PBO
- Synchronization
- User-defined methods
- Process flow/transition
- Communication

Classes of PBO
- Periodic
- Aperiodic

Events
- Timer interrupt
- Sync
- External signal
- Wakeup
- Cycle
- On
- Off
- Term
- Init
- > in-vars
- > out-vars
- > in-consts
- > out-consts
- NOT CREATED
- Spawn
Summary

- **Class 421**
  Overview of Reconfigurable Software
  - Motivation for methodology
  - Configuring components
  - Overview of framework
  - Subsystem decomposition into components

- **Class 431**
  Implementing a Framework
  - Preemptive vs. non-preemptive frameworks
  - Code-oriented aspect (PBO_MODULE)
  - Data-oriented aspect (SVAR Mechanism)
  - Real-time issues