real-time systems
implementation techniques

Prof. J.-D. Decotignie
CSEM Centre Suisse d’Electronique et de Microtechnique SA
Jaquet-Droz 1, 2007 Neuchâtel
jean-dominique.decotignie@csem.ch

Outline
- Introduction
- Synchronous implementations
- Asynchronous implementations
- Mixed Implementations
- Kernel Based Implementations

Taxonomy of Implementations
- With or without kernel support
- Synchronous or asynchronous
- Time-triggered (TT) or event-triggered (ET)

Synchronous Implementations
- introduction
- polling loops
- example - terminal emulator
- cyclic executives
- coroutines
- state based systems
- synchronous languages

Styles may obviously be mixed
What’s good about synchronous implementations

- Low overhead between all task switches are known in advance
- No interrupt, hence no context save
- No concurrent access nor data integrity problems (except in multiprocessors)
- Jitter may be kept low
- Behavior is predictable

Example: terminal emulator

- Microprocessor board with
  - Display interface
    - Position registers: RegX et RegY
    - RegChar for display
    - Fixed font (24 lines x 80 char.)
  - Serial interface to the host computer
    - Speed: 9600 bits/s
    - OutReg for emission
    - InReg for reception
    - Status register
  - Serial interface to keyboard
    - Speed: 300 bits/s
    - RegKeyB for punched key
    - Status register

Example: terminal emulator (2)

- Each character received from the host must be displayed (maximum 1 character / 1ms)
- Each character received from keyboard must be sent to host (max. 1 char. / 33 ms)
- All characters are displayed
  - Starting from top left corner
  - When 80 characters are displayed on a line, the display position to next line, 1st position
  - When bottom right of screen reached, start again at top left

Polling Loop

Loop

if character_received_from_host then
  PositionX := PositionX + 1;
  if PositionX = 80 then
    PositionX := 0; PositionY += 1;
    if PositionY = 24 then PositionY := 0; endif;
  endif;
  RegY := PositionY;
  RegX := PositionX;
  RegChar := InReg;
endif;
if character_received_from_keyboard then
  OutReg := RegKeyB;
endif;
end loop;
Polling loop – general structure

```plaintext
LOOP
  if Event_1 then Action_1;
  if Event_2 then Action_2;
  ....
  ....
  if Event_N then Action_N;
ENDLOOP;
```

- condition: $\forall j \in [1..N] \sum_{i=1}^{N} C_i \leq \min(T_j)$
- beware: "false" events

Periodic Actions

```plaintext
Loop
  if Counter > LCM(Ti) then
    Counter = Counter - LCM(Ti);
  for i=1..N do Count_i = 0; endfor
  endif;
  for i=1..N do
    if Counter \geq Count_i.Ti then
      action_i;
      Count_i = Count_i + 1;
    endif;
  endfor;
EndLoop;
```

Constraints for proper operation

- Condition $\forall j \in [1..N] \sum_{i=1}^{N} C_i \leq \min(T_j)$
- No preemptive scheduling [Jeffray]
  $\sum_{i=1}^{n} C_i \leq 1 \text{ and } \forall i \cdot 1 < i \leq n; \forall L \cdot T_1 < L < T_i; L \geq C_i + \sum_{k=1}^{i-1} \left| \frac{L-1}{T_k} \right| C_k$
- More detailed condition $\sum_{i=1}^{N} C_i + C_1 \leq 2T_i$
- Necessary condition $\forall i, j \in [1..N]; C_i \leq T_j$

Terminal Emulator

- 24 lines, 80 character each / 2048 memory loc.
- 1 register with the address of the first displayed line
  - used to scroll up screen / need to clean line appearing

```plaintext
PROCEDURE ScrollUp;
  (* First prepare a blank line ‘under’ *)
  (* the bottom line of the display. *)
  EraseChar(HDisplayed, TotalDisplayed);
  (* ‘Then scroll up *)
  StartDisplayOffset += HDisplayed MOD MemorySize;
  CRTSelectionReg := BYTE(13);
  CRTSelectedReg := BYTE(StartDisplayOffset);
  CRTSelectionReg := BYTE(12);
  CRTSelectedReg := BYTE(StartDisplayOffset DIV Bits8);
  SetCursor(CursorPos);
```
Terminal Emulator (2)

- Characters are displayed are return of beam sweep
  - Done every 60 µs

```pascal
PROCEDURE DisplayChar(Ch:CHAR);
VAR MemPos : CARDINAL;
BEGIN
  MemPos := (CursorSPos + StartDisplayOffset) MOD MemorySize;
  CRTSelectionReg := BYTE(19);
  CRTSelectedReg  := BYTE(MemPos);
  CRTSelectionReg := BYTE(18);
  CRTSelectedReg  := BYTE(MemPos DIV Bits8);
  CRTSelectionReg := BYTE(31);
  WHILE NOT(7 IN ByteSet(CRTStatus)) DO  END;
  CRTRegLatch     := Ch;
  CRTSelectedReg  :=BYTE(Ch);
  Right;
END DisplayChar;
```

Terminal Emulator (3)

- Cleaning up may take up to 5 ms
- Characters may be received every 1 ms
- Within 5 ms, up to 5 characters may be received

- How to avoid loosing characters?
  - Split into sub-actions

Cyclic executives

- May be considered as generalized polling loops
- Principle:
  - Create a schedule for the task set assuming they are non preemptive
  - Implement this order as consecutive procedure calls (one each time a task appears in the schedule) in a loop
- Execution order never changes
  - Jitter may be controlled
  - Precedence constraints may be easily fulfilled

Cyclic Executives: definitions

- Macrocycle (hyperperiod): repetition period of the whole sequence
  - Duration = LCM(Ti); LCM= Least Common Multiple
- There may be different sequence for different modes
- Macrocycle is divided into frames (enforce timeliness)
- Frames are often of the same duration -> microcycle
**Cyclic Executives: Example**

<table>
<thead>
<tr>
<th>Task</th>
<th>T</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>15</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>30</td>
<td>10</td>
</tr>
</tbody>
</table>

- Macrocycle=30
- Microcycle=10
- C divided into C1, C2 and C3 of duration 3, 3 and 4

**Microcycle Duration Calculation**

- 4 necessary conditions
  - 1: \( m \leq D_i \)
  - 2: \( m \geq C_i \)
  - 3: \( m + \{m - \text{GCD}(m, T_i)\} \leq D_i \)
  - 4: \( \exists k \in \mathbb{N} \cdot M = km \)

**Example – Chemical Reactor**

<table>
<thead>
<tr>
<th>Task</th>
<th>T</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>100</td>
<td>Max.</td>
</tr>
<tr>
<td>B</td>
<td>75</td>
<td>75</td>
<td>Max.</td>
</tr>
</tbody>
</table>

- M = 300
- \( m = 75, 60, 50, 42.85, 37.5, 33.3, \ldots \) (cond.1&4)
- \( m = 75, 60, 42.85 \) impossible (cond.3)
- \( m = 50 \) OK according to cond. 3 but not good
- \( m = 37.5 \Rightarrow C_A \) and \( C_B \leq 37.5 \)

**Example – Chemical Reactor (2)**

- Processor is used at 87.5% max.
- Reaction time is 75ms for A and 37.5ms for B
- Task A is not strictly periodic (by chance it is for B)
- Preemptive scheduling would have allowed tasks with longer duration
Example – Chemical Reactor (3)

- GCD approach \( \Rightarrow m=25 \)
- Modifying the periods
  - \( T_A = 75 \Rightarrow M=m=75 \)

Exercise

- Implement this task set as a cyclic executive

<table>
<thead>
<tr>
<th>Task</th>
<th>T</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>14</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>30</td>
<td>6</td>
</tr>
</tbody>
</table>

Cyclic Executives - Evaluation

- Advantages
  - Predefined sequence of tasks
  - Constraints are guaranteed
  - No need to have a real-time kernel
  - Non preemptive
    - No overload due to task switch
    - No need to protect shared resources
  - Free choice of schedule creation technique

- Traps
  - High sensitivity to frame overtime
  - Condition 2 \( \Rightarrow \) long tasks may be split = hidden preemption
    - Risks with data consistency
    - Risks that precedence constraints may not be filled
  - Requirements may be unduly changed (i.e. changing \( T_A \) from 100 ms to 75 ms)
Frame Overtime

- Nearly impossible to avoid so what to do?
  - Nothing
    - Risk of avalanche effect
    - A task that is not the cause of the overtime may miss its deadline
  - Stop the running task
    - May leave data structures (shared or not) in non-consistent state
  - Stop the task but give processor to exception code
    - Similar to exceptions in Java
    - The user may program the desired policy

Cyclic Executives - Evaluation

- Drawbacks
  - Any modification on a task implies that
    - Sequence should be checked again
    - Task split may have to be revised
  - Same case when a task is added
  - Loss of efficiency by using periodic tasks to handle sporadic events

Terminal Emulator as a cyclic executive

- Erase line that appears (instead of the line that disappears)
- Split this action into sub-actions

Terminal Emulator Pseudo-code

```
loop
  if char_receive_from_host then put_into_buffer endif
  if mode=1
    if char_received_from_KeyB then OutReg=RegKeyB endif
    NbCar = 0;
    while (char_in_buffer & char≠LF & NbCar≤16)
      increment(NbChar); display(char);
      increment(NbCharLine[CurrentLine])
    endwhile
    if car = LF then
      mode = NbCharLine[1] DIV 16 + 2;
      NbCar = NbCharLine[1] MOD Y6;
      endif;
    elseif mode=2
      while (NbChar>0) NbChar-=1; display(space); endwhile
      scroll_up; mode = 1;
    elseif mode>2
      for i=1 à 23 do NbCharLine[i] = NbCharLine[i+1]; endfor
      mode = mode - 1;
      endif;
  endif
endloop;
```
Coroutines

- Invented by Conway

"an autonomous program that communicates with adjacent modules as if they were input and output procedures. Coroutines are thus procedures that are at the same hierarchical level, each one acting as if was the main program when, in fact there is no main program"

Coroutine A

```plaintext
begin
  Call B
  Call A
  Call B
  Call A
end
```

Coroutine B

```plaintext
begin
  Call B
  Call A
end
```

Call B

Call A

Call B

Call A

Coroutines

Coroutines in Modula-2

- Creation

```
NEWPROCESS ( Procedure_Name : PROC;
  Work_Space : ADDRESS;
  Work_Space_Size: CARDINAL;
  VAR Coroutine_Handle : ADDRESS);
```

- Transfer of control

```
TRANSFER (VAR Handle Coroutines 1, Handle Coroutines 2: ADDRESS);
```

Coroutine features

- local data values persist between the instant at which a coroutine looses control of processor and the next instant at which it gains control
- execution of a coroutine is suspended when it looses control of the processor and will not resume before it will be given the control back at a later time
- control is explicitly transferred from one coroutine to another one. This results in the currently executing coroutine to be suspended and to resume the execution of the target coroutine. This is what is often called quasi-parallelism.

Example of coroutines

```
MODULE Coroutine1;
FROM SYSTEM IMPORT ADDRESS,
  NEWPROCESS, TRANSFER;
VAR PP, C1, C2, C3 : ADDRESS;
PROCEDURE P1;
BEGIN
  LOOP
    ...
    TRANSFER (C1, C2);
    ...
  END;
END P1;

PROCEDURE P2;
BEGIN
  LOOP
    ...
    TRANSFER (C2, C3);
    ...
  END;
END P2;

PROCEDURE P3;
BEGIN
  LOOP
    TRANSFER (C3, C1);
    ...
  END;
END P3;
BEGIN (* main program *)
  NEWPROCESS(P1, C1);
  NEWPROCESS(P2, C2);
  NEWPROCESS(P3, C3);
  TRANSFER (PP, C1);
END Coroutine1.
```
Coroutine Implementation

PROCEDURE TRANSFER (VAR Source, Destination: ADDRESS);
BEGIN
Source := REG(A6);
SETREG(A6, Destination);
END TRANSFER;

Example of coroutines (2)

MODULE Coroutine2;
FROM SYSTEM IMPORT ADDRESS, NEWPROCESS, TRANSFER;
CONST   Nb = 20; (* number of coroutines *)
VAR   Curr       : CARDINAL;
Coroutines : ARRAY [0..Nb] OF ADDRESS;
PROCEDURE DoSomeThing;
VAR   EnCours : CARDINAL;
BEGIN
  LOOP
    Curr := (Curr+1) MOD Nb;
    TRANSFER (Coroutines[EnCours], Coroutines[Curr]);
  END;
END DoSomeThing;
BEGIN (* programme principal *)
  FOR I:=0 TO Nb-1 DO
    NEWPROCESS (DoSomeThing, ..., ..., Coroutines[I]);
  END;
END;

Coroutines - Evaluation

- Advantages
  - simple and efficient
  - predetermined sequence => constraints are guaranteed to be met
  - real-time kernel useless
  - no preemption =>
    - additional load due to switch is reduced
    - no need to protect shared resources
  - the manner to obtain the sequence is free
Coroutines – Evaluation (2)

- drawbacks
  - a coroutine that terminates causes the application to terminate
  - software structure ???
  - no verification of timing constraints at run time
  - not strictly periodic

State based programming

- A number of formalisms
  - For instance: GRAFCET

Graph Interpretation Algorithm

Loop
- Read input states
- Determine transition that can be fired
- Simultaneous firing of all transition that can be
- State update
- Action update
End Loop.

Graph Interpretation Algorithm (2)

Loop
- read inputs
  loop
    for each branch, determine next state
  endloop
loop
  for each branch, update state
endloop
loop
  for each branch, update actions
endloop
Endloop
Terminal Emulator Graph

- actions (tasks) split in "atoms"
- all events handled at same frequency
- only relevant events are tested

Observations

- The worst case loop execution time depends on:
  - the duration of the atomic actions \( AA \)
  - the number \( Na \) of these that can be executed in parallel
  - the duration of:
    - Input readings \( T_i \)
    - Updating the outputs \( T_o \)
    - Executing evolution tests \( T_{tests} \)

\[
\text{loop duration} = \sum_{i=1}^{Na} AA_j + N_i T_i + N_o T_o + T_{tests}
\]

Observations (2)

- Actions are normally supposed to be subcontracted to the operative part (of the control part)
  - Multitasking or separate processors
- The response time (loop time) should be lower than the shortest constraint
- If this is the case (without subcontracting), the application may be implemented in a sequential manner

Possible Translation

```
MODULE Terminal;
FROM Usart IMPORT ReadUSART, WriteUSART;
FROM Screen IMPORT Display, NbChar, CharDisplayed, First, SavePos, RestorePos, ScrollUp;
FROM KeyBoard IMPORT ReadKeyBoard;
FROM Buffer IMPORT ReadBuffer, WriteBuffer;
CONST LF = 12C; Space = ' ';
VAR
  StepKeyB, StepReception, StepPrinc: CARDINAL;
  NextStepKeyB, NextStepPrinc, NextStepReception: CARDINAL;
  Char: CHAR;
  Count, Nb: CARDINAL;
PROCEDURE InitStep;
BEGIN
  StepPrinc := 21; StepReception := 1; StepKeyB := 11;
END InitStep;

PROCEDURE Loop;
BEGIN
  LOOP
    NextStepKeyB := StepKeyB;
    CASE StepKeyB OF
      11, 13: ReadKeyBoard(Char, Nb);
        IF Nb = 1 THEN
          NextStepKeyB := 12;
          WriteUSART(Char);
        END;
      12: NextStepKeyB := 13;
    | ELSE
    NextStepReception := StepReception;
    CASE StepReception OF
      1: ReadUSART(Char, Nb);
        IF Nb = 1 THEN
          NextStepReception := 2;
          WriteBuffer(Char);
        END;
      2: NextStepReception := 3;
    | ELSE
    NextStepReception := StepReception;
    END;
  END;
END;
```

```
Possible Translation (cont.)

Possible Translation continued...

Special Cases

- Actions (beware cancellation)
  - Continuous
  - Pulse shaped
  - Conditional
- Transitions
- Timers

Special Cases (2)

- AND junction
- AND distribution

Special Cases (3)

- OR distribution
- OR junction
Other Synchronous Approaches

- Synchronous languages
  - ESTEREL
  - LUSTRE
  - SIGNAL
  - STATE CHARTS

Asynchronous implementations

- events are detected by peripherals
- signaled by interrupts
- observed condition is often programmable
  - interface circuit (parallel, serial, communication, ...)
- no kernel support
- sporadic tasks

Asynchronous implementations (2)

- one interrupt level for each task and peripheral
- several peripheral share the same level
  - polling
  - daisy chain

One Interrupt per peripheral

- priorities assigned using DM, RM or another
- response time \( R_i = C_i + \sum_{j \in hp(i)} \frac{R_j}{T_j} C_j \)
- enforcement of minimum event inter arrival time
  - measure elapsed time
    - too short, just note it
    - too short, only acknowledge interrupt
- implementation: 1 task = 1 interrupt routine
- save and restore context
An interrupt for more than one peripheral

- daisy chain
- priorities assigned using DM, RM or another
- response time: \( R_i = C_i + \sum_{\forall j \in hp(i)} \left[ \frac{R_j}{T_j} \right] C_j + \max \{ C_j \} \)
- implementation: 1 task = 1 interrupt routine
- enforcement of minimum event inter arrival time

An interrupt for more than one peripheral (2)

- polling
  - if fair: \( R_i = C_i + \sum_{\forall j \in hp(i) \, j \neq i} \left[ \frac{R_j}{T_j} \right] C_j + \sum_{j \in pool_i \, j \neq i} \{ C_j \} \)
  - priorities assigned using DM, RM or another
  - implementation:
    - 1 interrupt routine by interrupt level
      - find source
      - calls task (sub-program)
      - saves and restores context

Periodic tasks

- one timer per task
  - as sporadic tasks
  - difficult to control jitter
- one timer for more than one task
  - period = GCD\{T_i\} or cf. cyclic executives
  - at each interrupt occurrence, find sequence of tasks to execute
  - one interrupt handler
  - sequences codes for each instant
  - tasks called as procedures

Asynchronous implementation - evaluation

- advantages
  - no need for a RT kernel
  - no need to split tasks
  - simple to implement
  - short response times
- drawbacks
  - preemption =>
    - additional load for context save/restore
    - need to protect access to shared resource
    - difficult to control jitter
    - sequence based upon fixed priority
    - non preemptible is some cases
Asynchronous implementation – mutual exclusion

- only conventional method: mask interrupts
- not necessary when several tasks share the same interrupt level
- data consistency may be solved by other means
  - "atomic" pointers
  - data banks

Hybrid systems

- use the processor idle time to run a useful task
- have interrupt triggered tasks as foreground and a number of tasks as a cyclic executive in background
  => calculate according to Joseph and Pandya
- extension of coroutines to handle asynchrony

Coroutines in Modula-2

- creation
  - NEWPROCESS (ProcedureName : PROC;
                Workspace : ADDRESS;
                WorkspaceSize : CARDINAL;
                VAR Coroutine_Handle : ADDRESS;
                Priority : CARDINAL);

- control transfer
  - TRANSFER (VAR Handle_Coroutine1,
               Handle_Coroutines 2: ADDRESS);

- asynchronous events
  - IOTRANSFER (VAR Handle_Coroutine1,
                 Handle_Coroutines 2: ADDRESS;
                 Vector : ADDRESS);

Event handling in Modula-2

- IOTRANSFER better named as WaitForInterrupt
- second parameter has two roles

\[ P2 := \text{FirstDestination}; \]
\[ \text{LOOP} \]
\[ \text{WHILE again DO} \]
\[ \ldots \text{(* configure interface *)} \]
\[ \text{IOTRANSFER (P1, P2, Vector);} \]
\[ \ldots \text{(* handle interrupt *)} \]
\[ \text{END; (* WHILE *)} \]
\[ \text{(* no more input or output operation *)} \]
\[ \ldots \]
\[ \text{TRANSFER (P1, P2);} \]
\[ \text{END; (* LOOP *)} \]
IOTRANSFER Implementation

- Nothing in A0, A1, D0, D1 at procedure call time
- A5 will be used as the running coroutine handle (not used by compiler)
- in case of interrupt, the coroutine that invoked IOTRANSFER should be given the control

=> one interrupt routine for each IOTRANSFER invocation

IOTRANSFER Implementation - interrupt

- A single generic procedure that is copied in each coroutine workspace (68K)

```
(*$P- SUPPRESS ENTRY/EXIT CODE GENERATED BY COMPILER *)
PROCEDURE InterruptCode;
(* this code is copied at the top of the coroutine workspace. Its *)
(* start address in copied in the vector when IOTRANSFER is invoked *)
BEGIN
ASSEMBLER
ORI #0700H, SR         ; no interrupt        ; 2 Words
MOVEM.L D0-D7/A0-A7,20(A5) ; save registers      ; 3 Words
MOVE.L A5, D0       ; 1 Word
LEA -98(PC), A5        ; descriptor address  ; 2 Words
MOVEA.L (A5), A1           ; Destination^.ThisPD ; 1 Word
MOVE.L D0, (A1)       ; Dest^.ThisPD^ = current ; 1 Word
MOVEM.L 28(A5),D2-D7 ; restore registers that ; 3 Words
MOVEM.L 60(A5),A2-A7 ; hold information     ; 3 Words
RTE                                               ; 1 Word
END (* ASSEMBLER *);
```

Coroutine descriptor

Towards bottom of stack

<table>
<thead>
<tr>
<th>Top of stack</th>
<th>Workspace + size of</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7</td>
<td>34 bytes reserved for interrupt code</td>
</tr>
<tr>
<td>A6</td>
<td>+84</td>
</tr>
<tr>
<td>D1</td>
<td>+80</td>
</tr>
<tr>
<td>D0</td>
<td>+20</td>
</tr>
<tr>
<td>reserved</td>
<td>+16</td>
</tr>
<tr>
<td>Free heap size</td>
<td>+12</td>
</tr>
<tr>
<td>Stack limit</td>
<td>+8</td>
</tr>
<tr>
<td>Handle</td>
<td>+4</td>
</tr>
<tr>
<td>Ending procedure</td>
<td>+4</td>
</tr>
<tr>
<td>Exception identifier</td>
<td>+6</td>
</tr>
<tr>
<td>Initial PC</td>
<td>+10</td>
</tr>
<tr>
<td>Initial SR</td>
<td>+12</td>
</tr>
<tr>
<td>Towards bottom of stack</td>
<td>workspace</td>
</tr>
</tbody>
</table>

Code for TRANSFER

```
PROCEDURE TRANSFER (VAR Source, Destination : ADDRESS);
BEGIN
ASSEMBLER
MOVEA.L Source, A0
MOVEA.L Destination, A1
TRAP      #TfrTrap
END (* ASSEMBLER *);
```

PROCEDURE TransferTrap;
(* Paramètres: A0 = Source; A1 = Destination. *)
```
BEGIN
ASSEMBLER
ORI #0700H,SR           ; disable interrupts
MOVEM.L D2-D7,DSOffset(A5) ; save registers
MOVEM.L A2-A7,DSOffset(A5) ; DOffset=28, AOffset=60
MOVE.L A5,D0       ; save current handle
MOVEA.L (A1),A5       ; update current process
MOVE.L D0,(A0)       ; source = current (RegOffset=20)
MOVEM.L R2Offset(A5),D0-D7/A0-A7 ; restore registers
RTE ; resume the destination coroutine
END (* ASSEMBLER *);
```

PROCEDURE TransferTrap;
(*)
### IOTRANSFER Implementation

```plaintext
PROCEDURE IOTRANSFER (VAR Src, Dest: ADDRESS; Vector: ADDRESS);
BEGIN
  ...
END IOTRANSFER;

(*$P- Suppress entry/exit code generated by compiler *)

PROCEDURE IOTransferTrap;
(* Parameters: A0 = Source; A1 = Destination; D0 = vector *)
BEGIN
  ASSEMBLER
  ORI      #0700H, SR           ; disable interrupts
  MOVEM.L  D2-D7, D2Offset(A5)  ; save registers
  MOVEM.L  A2-A7, A2Offset(A5) ...
  SETREG (A0, New); Handle := New;
  END (* ASSEMBLER *);
END IOTransferTrap;
```

### Remarks

- An interrupt should not occur before IOTRANSFER has been invoked
  - Use module priorities
  - Define a priority for each coroutine
- After interrupt processing, control may be or may not be returned to the interrupted coroutine.

### NEWPROCESS implementation

```plaintext
PROCEDURE NEWPROCESS (Name: PROC; Workspace: ADDRESS;
Size: CARDINAL; VAR Handle: ADDRESS;
Priority: CARDINAL);
VAR New: ADDRESS;
BEGIN
  IF (ODD (Workspace)) OR (ODD (Size))
  OR (Size < MinimumSize) THEN
    HALT (NEWPROCERR);
  END (* IF *);
  IF TrapsNotInitialized THEN
    TransferAddr := TransferTrap;
    IOTransferfrAddr := IOTransferTrap;
    TrapsNotInitialized := FALSE;
  END (* IF *);
  New := VAL(ADDRESS(VAL(LONGINT, Workspace) +
  VAL(LONGINT, Size) - VAL(LONGINT, InterruptCodeSize) -
  VAL(LONGINT, SIZE(Descriptor))));
  LOAD #(A0), A1
  MOVE.L Workspace, -(A1)     ; stack limit = workspace
  CLR.L     -(A1)                ; ThisPD := NIL;
  LEA.L TerminateException, A2 ; execution termination procedure
  MOVE.L A2, -(A1)             ; ...
  MOVE.W Priority, D1
  BEQ Store                ; IF (Priority # 0) THEN
  AND.W #0F8FFH, D0            ; get priority bits
  AND.W #0F8PP, D0            ; update SR priority bits
  OR.W D1, D0            ; END (* IF *)
  Store: MOVE.W D0, -(A1)     ; initial SR
  MOVE.L A0, A5Offset(A0)     ; Initial process
  LEA A0Offset(A0), A0
  MOVE.L (A0)+
  MOVE.L A1, (A0)+
  MOVE.W InterruptCode, A1
  MOVE.W #InterruptCodeSize, D0 ; =33
  Copy: MOVE.W (A1)+, (A0)+
  MOVE.W #Copy, D0, D0, Copy
  JMP... Code
  END (* ASSEMBLER *);
END NEWPROCESS;
```
Conclusion

- it is possible
  - to have simple implementations (no kernel)
  - with a predictable result
  - with low overhead
- concurrent programming does not imply
  - tasks
  - mutual exclusion
  - synchronization