

Autonomous Tag-Playing Robots

“Auto-Tag”
Nathan Blythe

EEL 5666 (Intelligent Machine Design Laboratory)
Dr. Arroyo, Dr. Schwartz

Abstract	3
Introduction.....	4
Background.....	4
Scope and objectives.....	4
In this document.....	4
Game description	5
Integrated System.....	6
Mobile Platform	7
Drive motor.....	9
Heading motor	9
Sensors	10
Audible signal transmitter/receiver.....	10
Collision bus	10
Behaviors	12
Offensive players	12
Defensive players.....	12
Scanning.....	13
Experimental Results	14
Demo Day	14
Media Day.....	14
Appendix 1: Board design	15
Appendix 2: Source code	21
Main.a51	21
ADC.a51	33
ADuC841.mcu	36
Loco.a51.....	40
RNG.a51	42
Tic.a51	46

Abstract

We present a proposal for the design of an autonomous robot that, produced in number, will play the competitive children's game "tag". We formally define the game, which consists of at least two offensive or defensive players attempting to become or remain defensive, where players exchange modes by colliding with one another. We describe the necessary facilities of a robotic player and present a proposal for the overall structure and behavior of one approach, entitled *Auto-Tag*.

We define the drive-train, sensors, and algorithms of *Auto-Tag* at a high, generalized level. We describe a special sensor system named the "collision bus" – a collection of connected, touch-sensitive, conductive panels used by robots to exchange state information upon collision with one another. Finally we discuss experimental results.

Introduction

Background

Competition is a classic implementation of automation and machine intelligence. Through the use of human-relatable games the relative intelligence of a machine can be tested. Deep Blue demonstrated the ability of a computer system to master a human's game; an autonomous robot extends this approach to the tangible, allowing the observer to recognize the machine as a participant in the human's world.

Any number of competitive games would be suitable for this demonstration. The children's game of tag is a very "primal" example. A collection of robots that play competitive tag relate themselves to the competitive and somewhat primitive techniques employed by children. The robots can model real-life behavior to some level of accuracy as the mechanics of the game do not rely on any human-specific action.

Scope and objectives

The robot design is referred to as *Auto-Tag*. A collection of identical instances of such a design will competitively play tag. Some number of robots will begin as offensive players and the remaining robots will begin as defensive players. Robots navigate the playing field avoiding obstacles. If contact is made with a robot of the other role, the two player switch roles and the chase begins again.

Auto-Tag will be fully-autonomous, with the only operator input being the initial state selection and calibration. There will be no deliberately cooperative play between robots. All robots will be built to the same design and specifications.

Success of the project will be measured by the following objectives.

1. *Auto-Tag* locates nearby players and correctly determines their states.
2. *Auto-Tag* maneuvers away from an obstacle after detecting its presence.
3. *Auto-Tag* detects collisions with other players and behaves accordingly.
4. *Auto-Tag* approaches or avoids other players as appropriate by state.
5. Each robot is robustly constructed and behaves consistently.

In this document

The remainder of this document will lay out the formal rules of the game, describe the "world" in which the robots will compete, describe mechanical and electrical systems that make up *Auto-Tag* and conclude with a summary of experimental results.

Game description

Tag is defined by the following rules.

1. Tag is played by at least two identical, autonomous, non-cooperating entities referred to as *players* in an enclosed space referred to as the *arena*.
2. The arena contains *obstacles*: stationary objects roughly the same size as players. The bounding wall of the arena is detectable as an obstacle.
3. A player has a dynamic *state*: either *offensive* or *defensive*.
4. Tag is divided into two segments: the *pregame* and the *game*.
5. During the pregame, a human *operator* initializes each player's state and places players randomly throughout the arena. Defensive players are placed first, followed by offensive players.
6. Players prefer to be defensive.
7. When two players collide they exchange states. The player becoming offensive halts play until the player becoming defensive moves away.
8. The game ends only when interrupted by the operator.

Integrated System

The functionality of *Auto-Tag* is demonstrated at a high level in figure 1.

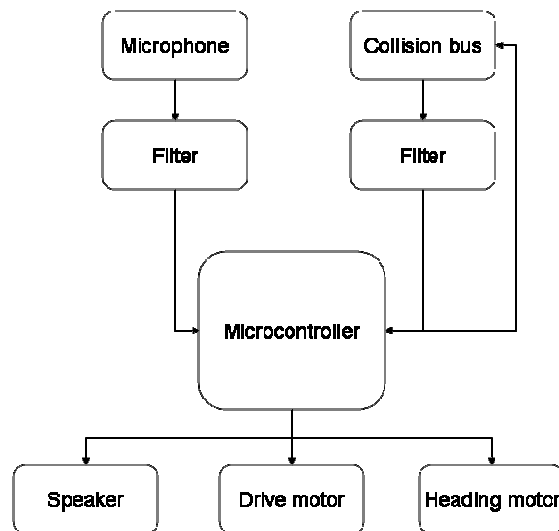


Figure 1: Functional layout of *Auto-Tag*

The robot is driven by two motors. The *drive motor* provides forward and reverse drive capabilities. The *heading motor* turns the robot about the drive motor's contact patch. Thus the heading motor is used to select a ray in the arena and the drive motor propels *Auto-Tag* along this ray.

Auto-Tag's audible signal is generated at the *speaker*. The *microphone* and *filter* is used to detect (unidirectionally) the presence of other players. Note that this feature was not included in final robot production due to time and financial constraints.

The *collision bus* is a contact-sensitive communications bus surrounding the perimeter of *Auto-Tag*. When two robots make contact they both detect the physical collision, and then share a common bus over which information (the state of each robot) can be exchanged. This allows them to adjust states as described in the rules.

Two buttons and two LEDs are used to allow operator-interaction and visual behavior confirmation.

Mobile Platform

The lower-layer of the two-layer platform is illustrated below.

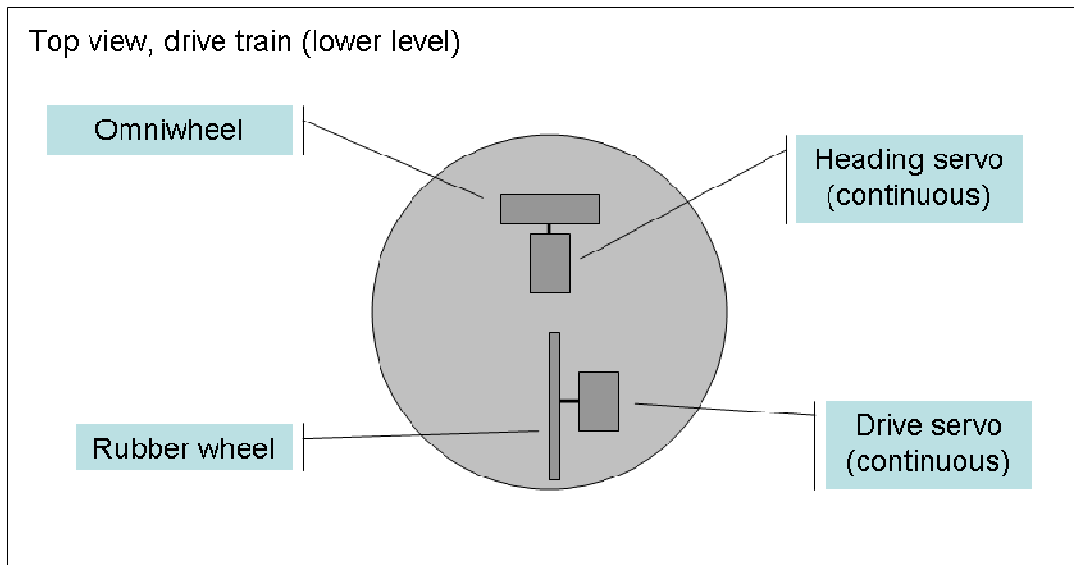


Figure 2: Top-view of the lower robot-platform drive-train

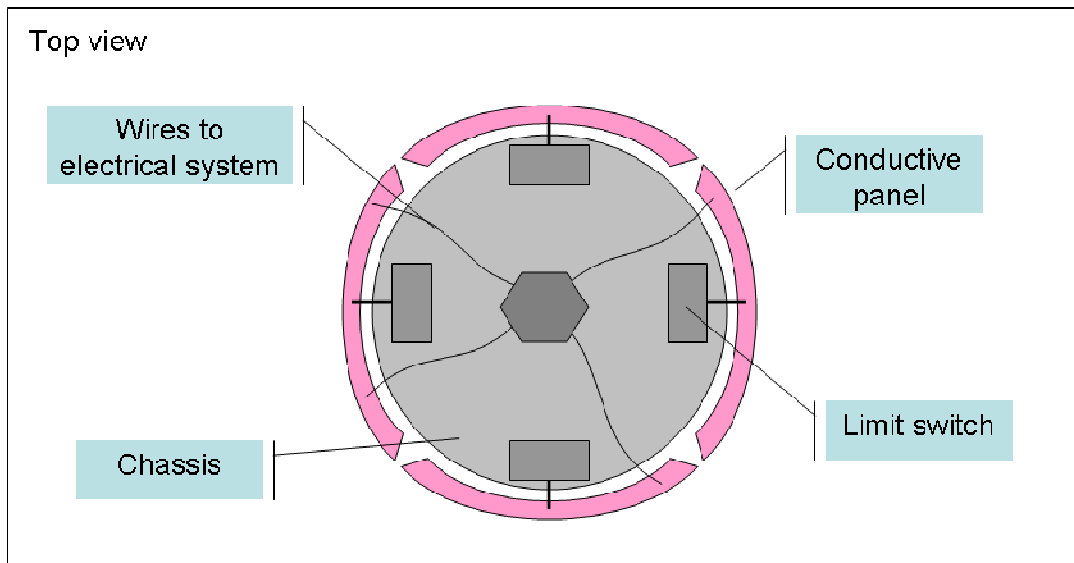


Figure 3: Top-view of the lower robot-platform collision bus

The lower chassis is a plastic disc available at Wal-Mart. The conductive panels are cardboard rectangles covered with two strips of aluminum foil (the upper to transmit the signal and the lower to provide the ground).

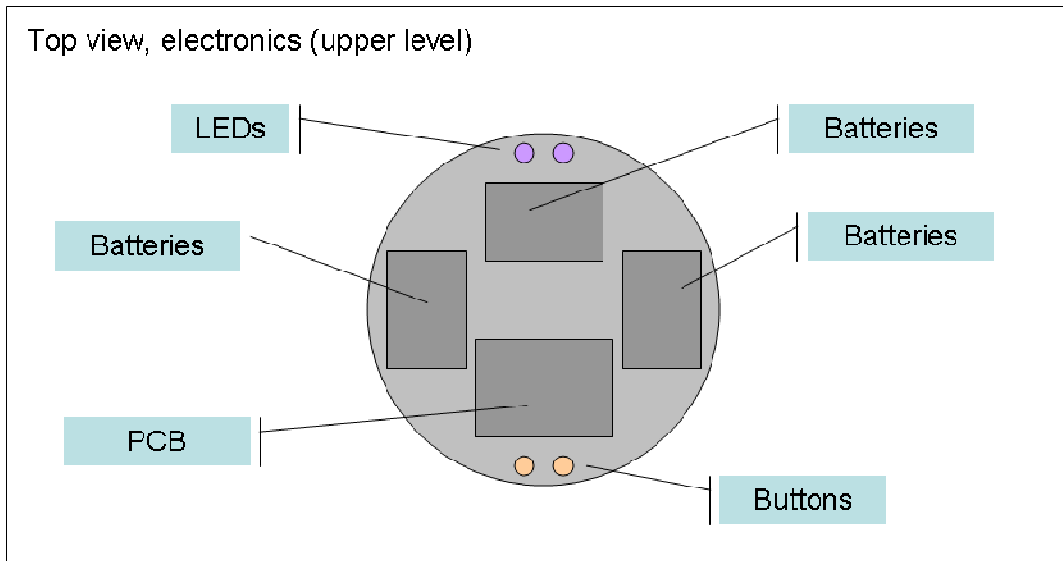


Figure 4: Top-view of the upper robot-platform electronics

The upper chassis is a duplicate of the lower chassis. The PCB and three battery containers are mounted around the center hole (see figure 3, demonstrating how the collision bus wires are fed to the upper layer).

Drive motor

The drive motor is a “hacked” servo attached to a thin rubber wheel. The drive wheel thus has a small contact patch. This gives *Auto-Tag* a small contact patch about which the robot rotates upon angular force from the heading motor.

Heading motor

The heading motor is a “hacked” servo turning an omni-directional wheel. The radius of the wheel is twice the distance to the contact patch of the drive motor. This allows for a 1:2 relation between angle of the servo and heading of the robot. A 1 degree clockwise adjustment of the servo yields a 2 degree clockwise adjustment of the robot. Thus there is a near-linear relation between “hacked” servo speed and rotational velocity of *Auto-Tag*. This design decision is intended to simplify calibration and adjustment of the speed at which the robot scans the surroundings (see “Behaviors”).

Sensors

Auto-Tag has two key sensor systems: the audible signal transmitter/receiver and the collision bus.

Audible signal transmitter/receiver

The audible signal system will be documented here but as previously discussed, was not included in the final production of the robots.

The purpose of the audible signal is to detect and identify the states of other players in the arena. A unidirectional microphone and filter is used to make an analog estimation of the players in the forward direction of *Auto-Tag*. By making continuous measurements as heading is adjusted, *Auto-Tag* can choose the optimal direction of travel.

Auto-Tag must also produce an audible signal. The signal is transmitted from a simple microphone. Self-interference is not be a concern as a player is only interested in detecting players of the other state, and the use of two non-harmonizing frequencies ensures that there is no interference between the two.

Collision bus

The collision bus is a unique contact-sensitive communications bus that surrounds *Auto-Tag*. Four conductive panels, each attached to a contact switch, surround the robot. When a collision occurs, one of the switches closes and that panel makes contact with the other player's panel.

Each panel has two conductive foil strips; the upper is a signal line and the lower is a ground line. When two robots collide they become commonly grounded for a short duration, during which both robots push their signal across the upper panels for reception by the other.

Each player pseudorandomly selects a pulse frequency. Each player alternates, at that frequency, between reading the bus and writing its own frequency (depending on its state) to the bus. When reading, the signal is filtered (in the same way that the audible signal received from the microphone is filtered). At some point there will be a time when one player is reading while the other is writing (and vice versa) and both players will detect the state of the other.

This seemingly complicated mechanism is in reality quite simple and has been used by the author of this proposal in unrelated (non-robotics) applications. By alternating between reading and writing the robots are (probabilistically) ensured of seeing each other's signal in a short amount of time, after which they can respond as per the rules.

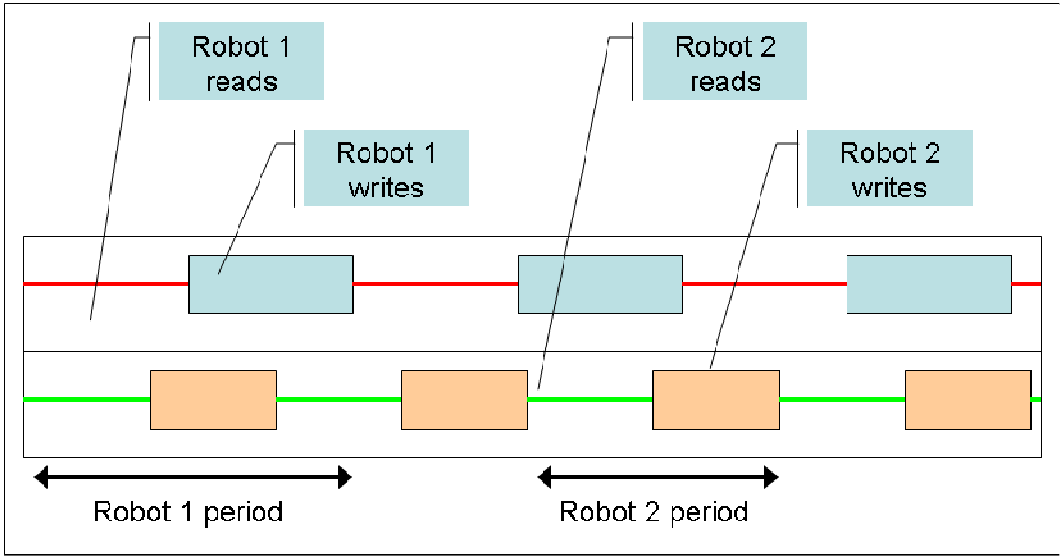


Figure 5: Example of bus communication

Figure 5 illustrates an example of bus communication, showing the offset between robot read/writes and the durations of bus access.

Behaviors

Offensive players

Offensive players will have one goal: to collide with a defensive player so as to change state to defensive.

1. Scan (see below). Rank all possible directions by a combination of the offensive and defensive indices that encourages directions with strong defensive indices and weak offensive indices. These directions are likely to have isolated defensive players. Note that the signal detection portion of this routine was not implemented in production robots.
2. Adjust heading to the chosen direction.
3. Drive forward for a fixed amount of time. Note: if time and resources permit, an advanced distance system can be implemented, in which distance is computed from the scan results and measured as it is traversed.
4. Go to (1).

On a collision:

1. Cycle the collision bus (see “Sensors” section).
2. Decide if the collision was with an obstacle (no signal on the bus), an offensive player (offensive signal on the bus) or a defensive player (defensive signal on the bus).
3. If the collision was with an obstacle, pseudorandomly choose a new heading centered 180 degrees from the current heading, and drive forwards a set amount of time to move away from the obstacle. Return to the ordinary control loop. Note: if time and resources permit, an advanced obstacle detection and mapping algorithm can be used to make use of remembered obstacles.
4. If the collision was with an offensive player, drive in reverse for a set amount of time and return to the ordinary control loop.
5. If the collision was with a defensive player, change state to defensive, drive in reverse for as set amount of time, and then begin following the defensive player behavior.

Defensive players

Defensive players attempt to avoid any offensive players they locate so as to remain defensive.

1. Scan (see below). Rank all possible directions by a combination of the offensive and defensive indices that discourages directions with strong defensive indices. These directions are likely to have offensive players and should be avoided. Note that the signal detection portion of this routine was not implemented in production robots.
2. Adjust heading to the chosen direction.
3. Drive forward for a fixed amount of time. Note: if time and resources permit, an advanced distance system can be implemented, in which distance is computed from the scan results and measured as it is traversed.
4. Go to (1).

On a collision:

1. Cycle the collision bus (see “Sensors” section).
2. Decide if the collision was with an obstacle (no signal on the bus), an offensive player (offensive signal on the bus) or a defensive player (defensive signal on the bus).
3. If the collision was with an obstacle, pseudorandomly choose a new heading centered 180 degrees from the current heading, and drive forwards a set amount of time to move away from the obstacle. Return to the ordinary control loop. Note: if time and resources permit, an advanced obstacle detection and mapping algorithm can be used to make use of remembered obstacles.
4. If the collision was with an defensive player, drive in reverse for a set amount of time and return to the ordinary control loop.
5. If the collision was with an offensive player, change state to offensive, wait for the collision bus to indicate that contact is broken, and pause for a set amount of time, after which adhering to offensive player behavior.

Scanning

1. Turn 90 degrees to the left in small increments (to be specified). At each increment make a reading from the microphone and filter (see “Sensors” section). This yields two numbers: the offensive and defensive indices, analog values which indicate a relative strength of the signals emitted from that direction. Note that the signal detection portion of this routine was not implemented in production robots.
2. Turn 90 degrees to the right without pausing.
3. Repeat (1) to the right.

Experimental Results

The design of *Auto-Tag* will be analyzed according to the following criteria.

1. *Auto-Tag* should move about the arena autonomously.
2. *Auto-Tag* should move away from obstacles within a reasonable time period after collision occurs, without undue repetition of the collision.
3. *Auto-Tag* should properly change mode upon collision with an opposing robot.
4. Each robot should be robustly constructed and structurally sound.
5. Code and design should be sufficiently documented and described.

Demo Day

On demo day the robots performed obstacle avoidance and attempted to transmit data on collision. The robustness of the collision detection and panels was demonstrated. Unfortunately the robots were unable to properly detect other robot's signals. This meets criteria 1, 2, 4, and 5 but not 3.

Media Day

On media day the robots satisfied all criteria except that proper bus detection and mode exchanging was only functional during "static" tests, when the robots were controlled by hand. During "live" collisions the robots did not make contact long enough, with a solid enough collision between the signal panels to exchange mode information. Hand-controlled testing worked well, however, demonstrating the general concept of the collision bus.

Appendix 1: Board design

Board design schematics and layout follow.

See page 1 of the accompanying schematics (AutoTag_Schem.pdf)

See page 2 of the accompanying schematics (AutoTag_Schem.pdf)

See page 3 of the accompanying schematics (AutoTag_Schem.pdf)

See page 4 of the accompanying schematics (AutoTag_Schem.pdf)

See the accompanying board layout (AutoTag_Board.pdf)

Appendix 2: Source code

Main.a51

```
; Auto-Tag
; 2009 Nathan Blythe
;

$INCLUDE (ADuC841.mcu)

; Configuration constants.
;
DRIVE_TIME      EQU 1      ; Duration of drive state.
DRIVE_TIMEBASE  EQU 1      ; Timebase for DRIVE_TIME.
BLINK_TIME      EQU 10     ; Duration of LED blink.
BLINK_TIMEBASE  EQU 0      ; Timebase for LED blink.
SLEEP_TIME      EQU 2      ; Duration of post-init sleep.
SLEEP_TIMEBASE  EQU 1      ; Timebase for post-init sleep.

SCAN_TIME       EQU 5      ; Duration of a single scan loop.
SCAN_TIMEBASE   EQU 0      ; Timebase for SCAN_TIME.
SCAN_LOOP       EQU 5      ; Number of loops in a scan.

THRESH_LO       EQU 058H
THRESH_HI       EQU 020H

; Pin configurations.
;
PIN_CBACK       EQU P0.0
PIN_CRIGHT      EQU P0.1
PIN_CLEFT       EQU P0.2
PIN_CFRONT      EQU P0.3
PIN_MODE0       EQU P0.4
PIN_MODE1       EQU P0.5
PIN_DEB0        EQU P0.6
PIN_DEB1        EQU P0.7

PIN_BUS         EQU P1.0
PIN_CYCLE       EQU P0.7

; State variables.
;
BSEG
    flagMode:    dbit 1
    flagCol:     dbit 1
    flagPrevF:   dbit 1
    flagPrevB:   dbit 1
    flagPrevR:   dbit 1
    flagPrevL:   dbit 1

; Interrupt vector table.
;
CSEG at 00000H
    ljmp Reset
    ljmp Stub      ; External interrupt 0
    ds    5
```

```

    ljmp Stub          ; Timer 0
    ds 5
    ljmp Stub          ; External interrupt 1
    ds 5
    ljmp Stub          ; Timer 1
    ds 5
    ljmp Stub          ; UART
    ds 5
    ljmp Stub          ; Timer 2
    ds 5
    ljmp adcISR        ; ADC
    ds 5
    ljmp Stub          ; SPI, I2C
    ds 5
    ljmp Stub          ; Power supply monitor
    ds 5
    ljmp Stub          ; Reserved
    ds 5
    ljmp ticISR        ; Timer interval counter
    ds 5
    ljmp Stub          ; Watchdog timer
Stub:
    reti

; Firmware entry point at reset.
;
;
Reset:
; Enable TIC interrupt.
;
    mov IE,    #10000000b
    mov IEIP2, #00000100b

; Initialize firmware components.
;
    call locoInit
; call adcInit
    call rngInit

    clr PIN_DEB0
    clr PIN_MODE0
    call setModeDisabled

; And here we go!
;
    ljmp Startup

; Detect a signal on the collision bus.
;
; Takes:
;   Nothing
;
; Returns:
;   A: 0 = no signal
;       1 = LO signal
;       2 = HI signal
;
; Mangles:
;   Nothing
;
readBus:

```

```

    push B
    clr PIN_BUS

; Count number of high edges seen in 255 131-cycle
; loops. TODO count
;
    mov A, #0
    mov B, #0FFH
readBus_Loop:
    jnb PIN_BUS, readBus_Loop_Ok    ; 4 cycles
;
    inc A                            ; 1 cycle
    sjmp readBus_Loop_Delay         ; 3 cycles
;
readBus_Loop_Ok:
    nop                               ; 1 cycle
    sjmp readBus_Loop_Delay         ; 3 cycles
;
readBus_Loop_Delay:                  ; 131 cycles
    push B                            ; 2 cycles
    mov B, #30                        ; 3 cycles
readBus_Loop_Delay0:
    djnz B, readBus_Loop_Delay0      ; 4 cycles
    pop B                             ; 2 cycles
    djnz B, readBus_Loop             ; 4 cycles

; Compare to thresholds.
;
    clr C
    subb A, #THRESH_LO
    jc readBus_noSignal
;
    subb A, #THRESH_HI
    jc readBus_loSignal

; HI signal detected.
;
    mov A, #002H
    pop B
    ret

; LO signal detected.
;
readBus_loSignal:
    mov A, #001H
    pop B
    ret

; No signal detected.
;
readBus_noSignal:
    mov A, #000H
    pop B
    ret

; Scan the bus once.
;
; Takes:
;   Nothing
;
; Returns:
;   A: 0 = no signal detected

```

```

;      1 = LO signal detected
;      2 = HI signal detected
;
; Mangles:
;   Nothing
;
BusDetect:
    push B
    setb PIN_DEB0

; Loop for a scan period (driving the bus with our signal).
;
    call setModeEnabled
    mov A, #SCAN_TIME
    mov B, #SCAN_TIMEBASE
    call ticStart
BusDetect_loop0Top:
    jnb ticTock, BusDetect_loop0Top

; Get off the bus and loop again, this time reading the bus
; as we go.
;
    call setModeDisabled
    mov A, #SCAN_TIME
    mov B, #SCAN_TIMEBASE
    call ticStart
;
BusDetect_loop1Top:
    call readBus
    jnz BusDetect_Done
    jnb ticTock, BusDetect_loop1Top

; All done.
;
BusDetect_Done:
    clr PIN_DEB0
    pop B
    ret

; Poll the collision bus.
;
; Takes:
;   Nothing
;
; Returns:
;   A:  0 (no collision)
;       'F' (front panel)
;       'B' (back panel)
;       'R' (right panel)
;       'L' (left panel)
;
; Mangles:
;   Nothing
;
pollCol:
    jb PIN_CFRONT, pollCol_notFront
    mov A, #'F'
    ret
pollCol_notFront:

    jb PIN_CBACK, pollCol_notBack
    mov A, #'B'

```

```

    ret
pollCol_notBack:

    jb PIN_CLEFT, pollCol_notLeft
    mov A, #'L'
    ret
pollCol_notLeft:

    jb PIN_CRIGHT, pollCol_notRight
    mov A, #'R'
    ret
pollCol_notRight:

    mov A, #0
    ret

; Disable the mode output.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
setModeDisabled:
    setb PIN_CYCLE
    ret

; Enable the mode output.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
setModeEnabled:
    clr PIN_CYCLE
    ret

; Set mode to offensive.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
setModeOffensive:
    clr PIN_CYCLE
    clr flagMode

```

```

    setb PIN_MODE0
    clr  PIN_MODE1
    ret

; Set mode to defensive.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
setModeDefensive:
    clr PIN_CYCLE
    setb flagMode
    clr  PIN_MODE0
    setb PIN_MODE1
    ret

; Toggle the mode.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
toggleMode:
    jnb flagMode, toggleMode_is0

    clr flagMode
    setb PIN_MODE0
    clr  PIN_MODE1
    ret

toggleMode_is0:
    setb flagMode
    clr  PIN_MODE0
    setb PIN_MODE1
    ret

; "Startup" state
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
Startup:
; Stop the motors.

```

```

;
mov A, #LOCO_STOP
call locoState

; Start timer for mode toggling.
;
mov A, #BLINK_TIME
mov B, #BLINK_TIMEBASE
call ticStart

Startup_Loop:
; Panel pressed: exit the loop.
;
call pollCol
jnz Startup_LoopDone

; Time elapsed: toggle the mode.
;
jnb ticTick, Startup_Loop
mov A, #BLINK_TIME
mov B, #BLINK_TIMEBASE
call ticStart
call toggleMode
sjmp Startup_Loop
Startup_LoopDone:

; Front panel pressed: defensive.
;
cjne A, #'F', Startup_Loop_notFront
call setModeDefensive
call setModeDisabled
;
mov A, #SLEEP_TIME
mov B, #SLEEP_TIMEBASE
call ticStart
Startup_Loop_Front:
jnb ticTick, Startup_Loop_Front
;
ljmp BugBot ; Drive
Startup_Loop_notFront:

; Back panel pressed: offensive.
;
cjne A, #'B', Startup_Loop_notBack
call setModeOffensive
call setModeDisabled
;
mov A, #SLEEP_TIME
mov B, #SLEEP_TIMEBASE
call ticStart
Startup_Loop_Back:
jnb ticTick, Startup_Loop_Back
;
ljmp BugBot ; Drive
Startup_Loop_notBack:

; Neither panel pressed: keep waiting.
;
sjmp Startup

; "Drive" state
;

```

```

Drive:
    call setModeDisabled

; Defensive: stand stopped.
; Offensive: drive forwards.
;
    mov A, #LOCO_DRIVE_F
    jnb flagMode, Drive_Start
    mov A, #LOCO_STOP
Drive_Start:
    call locoState

; Loop until a collision occurs.
;
Drive_Loop:
    call pollCol
    jz Drive_Loop

; Collision occurred: leap over to "Collide".
;
    ljmp Collision

; "Bounce" state
;
Bounce:
    call pollCol

; Front panel collision: drive backwards.
;
    cjne A, #'F', Bounce_notFront
    mov A, #LOCO_DRIVE_R
    call locoState
    mov A, #DRIVE_TIME
    mov B, #DRIVE_TIMEBASE
    call ticStart
    sjmp Bounce_Loop
Bounce_notFront:

; Back panel collision: drive forwards.
;
    cjne A, #'B', Bounce_notBack
    mov A, #LOCO_DRIVE_F
    call locoState
    mov A, #DRIVE_TIME
    mov B, #DRIVE_TIMEBASE
    call ticStart
    sjmp Bounce_Loop
Bounce_notBack:

; Right panel collision: turn left.
;
    cjne A, #'R', Bounce_notRight
    mov A, #LOCO_SPIN_L
    call locoState
    mov A, #DRIVE_TIME
    mov B, #DRIVE_TIMEBASE
    call ticStart
    sjmp Bounce_Loop
Bounce_notRight:

; Left panel collision: turn right.
;

```

```

    cjne A, #'L', Bounce_notLeft
    mov A, #LOCO_SPIN_R
    call locoState
    mov A, #DRIVE_TIME
    mov B, #DRIVE_TIMEBASE
    call ticStart
    sjmp Bounce_Loop
Bounce_notLeft:

; No collision: how did we get here?!
;
    ljmp Drive

; Loop until timer runs out or another collision
; occurs.
;
Bounce_Loop:
    jb ticTock, Bounce_Timeout
    call pollCol
    jz Bounce_Loop

; Collision occurred.
;
    ljmp Collision

; Timer ran out: back to regular operation.
;
Bounce_Timeout:
    ljmp Drive

; Handle a collision.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   R0
;
Collision:
    push ACC
    push B

; Scan loop.
;
    mov B, #SCAN_LOOP
    mov R0, #0
Collision_loopTop:
    call BusDetect
    jnz Collision_foundRobot
    djnz B, Collision_loopTop
    sjmp Collision_Done

; If we detected somebody, store the conclusion and
; then loop, unless we're done.
;
Collision_foundRobot:
    mov R0, A
    djnz B, Collision_loopTop
    sjmp Collision_Done

```

```

; Scan loop is done. Adjust the mode accordingly.
;
Collision_Done:
    mov A, R0
    cjne A, #001H, Collision_notLO
    call setModeOffensive
    call setModeDisabled
    pop B
    pop ACC
    ret
Collision_notLO:
    cjne A, #002H, Collision_notHI
    call setModeDefensive
    call setModeDisabled
    pop B
    pop ACC
    ret
Collision_notHI:
    pop B
    pop ACC
    ret

; "Escape" state.
;
; If offensive, do nothing for a few seconds.
; If defensive, run away from the direction of collision.
;
Escape:
    ljmp Bounce ; TODO

    jb flagMode, Escape_Defensive

    mov A, #LOCO_STOP
    call locoState
    mov A, #DRIVE_TIME
    mov B, #DRIVE_TIMEBASE
    call ticStart
Escape_Offensive:
    jnb ticTock, Escape_Offensive
    ljmp Drive

Escape_Defensive:
    ljmp Bounce

; "Bug bot" state
;
; Drive forward until a collision is detected. When
; there's a collision, drive backwards a bit, turn
; randomly and drive forwards again.
;
BugBot:
; Start out stopped.
;
    mov A, #LOCO_STOP
    call locoState

BugBot_Loop:

; Front switch hit. Drive backward, turn randomly,
; drive backward.

```

```

;
; jb PIN_CFRONT, BugBot_Loop_notFront
;
; call Collision
;
; mov A, #LOCO_DRIVE_R
; call locoState
;
; mov A, #3
; mov B, #1
; call ticStart
BugBot_Loop_frontWait0:
; jnb ticTock, BugBot_Loop_frontWait0
;
; call rngGet
; mov B, A
; mov A, #LOCO_SPIN_R
; jnb B.0, BugBot_Loop_frontTurn
; mov A, #LOCO_SPIN_L
BugBot_Loop_frontTurn:
; call locoState
;
; mov A, #2
; mov B, #1
; call ticStart
BugBot_Loop_frontWait1:
; jnb ticTock, BugBot_Loop_frontWait1
;
; mov A, #LOCO_DRIVE_R
; call locoState
; ljmp BugBot_Loop
BugBot_Loop_notFront:

; Right switch hit. Turn left, drive forward
; or reverse randomly.
;
; jb PIN_CRIGHT, BugBot_Loop_notRight
;
; call Collision
;
; mov A, #LOCO_SPIN_L
; call locoState
;
; mov A, #2
; mov B, #1
; call ticStart
BugBot_Loop_rightWait0:
; jnb ticTock, BugBot_Loop_rightWait0
;
; call rngGet
; mov B, A
; mov A, #LOCO_DRIVE_F
; jnb B.0, BugBot_Loop_rightDrive
; mov A, #LOCO_DRIVE_R
BugBot_Loop_rightDrive:
; call locoState
; ljmp BugBot_Loop
BugBot_Loop_notRight:

; Left switch hit. Turn right, drive forward or
; reverse randomly.

```

```

;
;  jb PIN_CLEFT, BugBot_Loop_notLeft
;
;  call Collision
;
;  mov A, #LOCO_SPIN_R
;  call locoState
;
;  mov A, #2
;  mov B, #1
;  call ticStart
BugBot_Loop_leftWait0:
;  jnb ticTock, BugBot_Loop_leftWait0
;
;  call rngGet
;  mov B, A
;  mov A, #LOCO_DRIVE_F
;  jnb B.0, BugBot_Loop_leftDrive
;  mov A, #LOCO_DRIVE_R
BugBot_Loop_leftDrive:
;  call locoState
;  ljmp BugBot_Loop
BugBot_Loop_notLeft:

; Rear switch hit. Drive forward, turn randomly,
; drive forward.
;
;  jb PIN_CBACK, BugBot_Loop_notRear
;
;  call Collision
;
;  mov A, #LOCO_DRIVE_F
;  call locoState
;
;  mov A, #3
;  mov B, #1
;  call ticStart
BugBot_Loop_rearWait0:
;  jnb ticTock, BugBot_Loop_rearWait0
;
;  call rngGet
;  mov B, A
;  mov A, #LOCO_SPIN_R
;  jnb B.0, BugBot_Loop_rearTurn
;  mov A, #LOCO_SPIN_L
BugBot_Loop_rearTurn:
;  call locoState
;
;  mov A, #2
;  mov B, #1
;  call ticStart
BugBot_Loop_rearWait1:
;  jnb ticTock, BugBot_Loop_rearWait1
;
;  mov A, #LOCO_DRIVE_F
;  call locoState
;  ljmp BugBot_Loop
BugBot_Loop_notRear:

; Nothing: keep doing whatever it is we're doing.
;

```

```

        ljmp BugBot_Loop

; Additional source files.
;
$INCLUDE(loco.a51)
$INCLUDE(tic.a51)
$INCLUDE(adc.a51)
$INCLUDE(rng.a51)

END

ADC.a51
; ADC support.
; 2009 Nathan Blythe
;
; Public routines:
;   adcInit:   Initialize and calibrate the ADC.
;   adcStart:  Start continuous ADC measurements on a channel.
;   adcStop:   Stop continuous ADC measurements.
;   adcSingle: Start a single ADC measurement on a channel.
;   adcISR:    ADC interrupt service routine.
;
; Public variables:
;   adcValid:  Flag indicating that the ADC value is now valid.
;   adcValue:  Last read value of the ADC.
;

; ADC channels.
;
ADC_MIC_OFFENSIVE EQU 0
ADC_MIC_DEFENSIVE EQU 1
ADC_COL_OFFENSIVE EQU 2
ADC_COL_DEFENSIVE EQU 3

; ADC bits.
;
BSEG
    adcValid: dbit 1

; ADC bytes.
;
DSEG
    adcValue: ds 1

; Routines follow.
;
CSEG

; Initialize and calibrate the ADC.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing

```

```

;
; Mangles:
;   Nothing
;
; Notes:
;   Internal 2.5V reference voltage.
;   ADC clock divider of 8.
;   4 clocks of track-and-hold.
;   No DMA.
;
;   See pages 24, 30 of the documentation.
;
adcInit:
    push ACC

; Initialize SFRs.
;
    mov ADCCON1, #10101100b
    mov ADCCON2, #00000000b
    mov ADCCON3, #00000000b

; Select channel AGND and perform offset calibration.
;
    mov ADCCON2, #00001011b
    mov ADCCON3, #00000101b
ADC_Calibrate_waitOffset:
    mov A, ADCCON3
    jb ACC.7, ADC_Calibrate_waitOffset

; Select channel VREF and perform gain calibration.
;
    mov ADCCON2, #00001100b
    mov ADCCON3, #00000101b
ADC_Calibrate_waitGain:
    mov A, ADCCON3
    jb ACC.7, ADC_Calibrate_waitGain

; All done.
;
    pop ACC
    ret

; Start continuous ADC measurements on a channel.
;
; Takes:
;   A: channel on which to begin measurements.
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
adcStart:
    push ACC

    clr adcValid

    orl A, #00100000b
    mov ADCCON2, A

    pop ACC

```

```

    ret

; Stop continuous ADC measurements.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
adcStop:
    mov ADCCON2, #00000000b
    ret

; Start a single ADC measurement on a channel.
;
; Takes:
;   A: channel on which to begin a measurement
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
adcSingle:
    push ACC

    clr adcValid

    orl A, #00010000b
    mov ADCCON2, A

    pop ACC
    ret

; ADC interrupt service routine.
;
adcISR:
    push ACC

; Upper nybble.
;
    mov A, ADCDATAH
    swap A
    anl A, #11110000b
    mov adcValue, A

; Lower nybble.
;
    mov A, ADCDATAH
    swap A
    anl A, #00001111b
    orl A, adcValue
    mov adcValue, A

; Value is now valid.

```

```

;
; setb adcValid

; All done.
;
; pop ACC
; reti

```

ADuC841.mcu

```

; Processor definitions for the Analog Devices ADuC841
; 2009 Nathan Blythe
;

```

```

$NOMOD51

```

```

; Disable listings (will re-enable at the end of this file).
;
$NOLIST

```

```

; Block data allocation around registers.
;

```

```

DSEG AT 00H
        ds 32

```

```

; Special function registers.
;

```

```

P0          DATA      080H ; Port 0. Default value 0xFF. Bit addressable.
SP          DATA      081H ; Stack pointer. Default value 0x07.
DPL         DATA      082H ; Data pointer low address byte. Default value
0x00.
DPH         DATA      083H ; Data pointer high address byte. Default value
0x00.
DPP         DATA      084H ; Data pointer access.
PCON        DATA      087H ; Power control and general purpose status flags.
Default value 0x00.
TCON        DATA      088H ; Timer 0, 1 control. Default value 0x00. Bit
addressable (see below).
        ITO          BIT      088H
        IE0          BIT      089H
        IT1          BIT      08AH
        IE1          BIT      08BH
        TR0          BIT      08CH
        TF0          BIT      08DH
        TR1          BIT      08EH
        TF1          BIT      08FH
TMOD        DATA      089H ; Timer 0, 1 modes. Default value 0x00.
TL0         DATA      08AH ; Low byte for timer/counter 0. Default value
0x00.
TL1         DATA      08BH ; Low byte for timer/counter 1. Default value
0x00.
TH0         DATA      08CH ; High byte for timer/counter 0. Default value
0x00.
TH1         DATA      08DH ; High byte for timer/counter 1. Default value
0x00.
P1          DATA      090H ; Port 1. Default value 0xFF. Bit addressable
(see below).
        T2          BIT      090H
        ADC0        BIT      090H
        T2EX        BIT      091H
        ADC1        BIT      091H

```

ADC2	BIT	092H	
ADC3	BIT	093H	
ADC4	BIT	094H	
ADC5	BIT	095H	
SS	BIT	095H	
ADC6	BIT	096H	
ADC7	BIT	097H	
I2CADD1	DATA	091H	; Second I2C peripheral address. Default value 0x7F.
I2CADD2	DATA	092H	; Third I2C peripheral address. Default value 0x7F.
I2CADD3	DATA	093H	; Fourth I2C peripheral address. Default value 0x7F.
SCON	DATA	098H	; Serial port control. Default value 0x00. Bit addressable (see below).
RI	BIT	098H	
TI	BIT	099H	
RB8	BIT	09AH	
TB8	BIT	09BH	
REN	BIT	09CH	
SM2	BIT	09DH	
SM1	BIT	09EH	
SM0	BIT	09FH	
SBUF	DATA	099H	; Serial port buffer access.
I2CDAT	DATA	09AH	; I2C bus access.
I2CADD	DATA	09BH	; First I2C peripheral address. Default value 0x55.
T3FD	DATA	09DH	; Timer 3 fractional divide ratio. Default value 0x00.
T3CON	DATA	09EH	; Timer 3 control. Default value 0x00.
P2	DATA	0A0H	; Port 2. Default value 0xFF. Bit addressable (see below).
PWM0	BIT	0A6H	
PWM1	BIT	0A7H	
TIMECON	DATA	0A1H	; Timer Interval Counter (TIC) control. Default value 0x00.
HTHSEC	DATA	0A2H	; 1/128 second counter. Default value 0x00.
SEC	DATA	0A3H	; 1 second counter. Default value 0x00.
MIN	DATA	0A4H	; 1 minute counter. Default value 0x00.
HOURL	DATA	0A5H	; 1 hour counter. Default value 0x00.
INTVAL	DATA	0A6H	; Interval after which to TIC interrupt. Default value 0x00.
DPCON	DATA	0A7H	; Data pointer control. Default value 0x00.
IE	DATA	0A8H	; Interrupt enables. Default value 0x00. Bit addressable (see below).
EX0	BIT	0A8H	
ET0	BIT	0A9H	
EX1	BIT	0AAH	
ET1	BIT	0ABH	
ES	BIT	0ACH	
ET2	BIT	0ADH	
EADC	BIT	0AEH	
EA	BIT	0AFH	
IEIP2	DATA	0A9H	; Secondary interrupt enables and priorities. Default value 0xA0.
PWMCON	DATA	0AEH	; PWM control. Default value 0x00.
CFG841	DATA	0AFH	; ADuC841 special configuration. Default value 0x10.
P3	DATA	0B0H	; Port 3. Default value 0xFF. Bit addressable.
RXD	BIT	0B0H	
TXD	BIT	0B1H	
INT0	BIT	0B2H	
INT1	BIT	0B3H	

MISO	BIT	0B3H	; Also used as PWM1.
T0	BIT	0B4H	
PWMC	BIT	0B4H	; Also used as PWM0.
T1	BIT	0B5H	
CONVST	BIT	0B5H	
WR	BIT	0B6H	
RD	BIT	0B7H	
PWM0L	DATA	0B1H	; PWM 0 low time. Default value 0x00.
PWM0H	DATA	0B2H	; PWM 0 high time. Default value 0x00.
PWM1L	DATA	0B3H	; PWM 1 low time. Default value 0x00.
PWM1H	DATA	0B4H	; PWM 1 high time. Default value 0x00.
SPH	DATA	0B7H	; Extend stack pointer to 11 bits. Default value 0x00.
IP	DATA	0B8H	; Interrupt priorities. Default value 0x00. Bit addressable (see below).
PX0	BIT	0B8H	
PT0	BIT	0B9H	
PX1	BIT	0BAH	
PT1	BIT	0BBH	
PS	BIT	0BCH	
PT2	BIT	0BDH	
PADC	BIT	0BEH	
ECON	DATA	0B9H	; Flash memory control. Default value 0x00.
EDATA1	DATA	0BCH	; Flash memory page byte 0.
EDATA2	DATA	0BDH	; Flash memory page byte 1.
EDATA3	DATA	0BEH	; Flash memory page byte 2.
EDATA4	DATA	0BFH	; Flash memory page byte 3.
WDCON	DATA	0C0H	; Watchdog timer control. Default value 0x10. Bit addressable (see below).
WDWR	BIT	0C0H	
WDE	BIT	0C1H	
WDS	BIT	0C2H	
WDIR	BIT	0C3H	
PRE0	BIT	0C4H	
PRE1	BIT	0C5H	
PRE2	BIT	0C6H	
PRE3	BIT	0C7H	
CHIPID	DATA	0C2H	; Chip ID.
EADRL	DATA	0C6H	; Flash memory address low byte.
EADRH	DATA	0C7H	; Flash memory address high byte.
T2CON	DATA	0C8H	; Timer 2 control. Default value 0x00. Bit addressable (see below).
CAP2	BIT	0C8H	
CNT2	BIT	0C9H	
TR2	BIT	0CAH	
EXEN2	BIT	0CBH	
TCLK	BIT	0CCH	
RCLK	BIT	0CDH	
EXF2	BIT	0CEH	
TF2	BIT	0CFH	
RCAP2L	DATA	0CAH	; Timer 2 capture/reload time low byte. Default value 0x00.
RCAP2H	DATA	0CBH	; Timer 2 capture/reload time high byte. Default value 0x00.
TL2	DATA	0CCH	; Timer 2 value low byte.
TH2	DATA	0CDH	; Timer 2 value high byte.
PSW	DATA	0D0H	; Program status word. Default value 0x00. Bit addressable (see below).
P	BIT	0D0H	
F1	BIT	0D1H	
OV	BIT	0D2H	
RS0	BIT	0D3H	
RS1	BIT	0D4H	

F0	BIT	0D5H	
AC	BIT	0D6H	
CY	BIT	0D7H	
DMAL	DATA	0D2H	; Direct memory access address low byte. Default value 0x00.
DMAH	DATA	0D3H	; Direct memory access address high byte. Default value 0x00.
DMAP	DATA	0D4H	; Direct memory access pointer.
PLLCON	DATA	0D7H	; Phase lock loop control. Default value 0x53.
ADCCON2	DATA	0D8H	; ADC control (part 2). Default value 0x00. Bit addressable (see below).
CS0	BIT	0D8H	
CS1	BIT	0D9H	
CS2	BIT	0DAH	
CS3	BIT	0DBH	
SCONV	BIT	0DCH	
CCONV	BIT	0DDH	
DMA	BIT	0DEH	
ADCI	BIT	0DFH	
ADCDATAL	DATA	0D9H	; ADC data low byte.
ADCDATAH	DATA	0DAH	; ADC data high byte.
PSMCON	DATA	0DFH	; Power supply monitor control. Default value 0x0E.
ACC	DATA	0E0H	; Accumulator. Default value 0x00. Bit addressable.
I2CCON	DATA	0E8H	; I2C control. Default value 0x00. Bit addressable (see below).
I2CI	BIT	0E8H	
I2CTX	BIT	0E9H	
I2CRS	BIT	0EAH	
I2CM	BIT	0EBH	
MDI	BIT	0ECH	
I2CID0	BIT	0ECH	
MCO	BIT	0EDH	
I2CID1	BIT	0EDH	
MDE	BIT	0EEH	
I2CGC	BIT	0EEH	
MDO	BIT	0EFH	
I2CSI	BIT	0EFH	
ADCCON1	DATA	0EFH	; ADC control (part 1). Default value 0x40.
B	DATA	0F0H	; B register. Default value 0x00. Bit addressable.
ADCOFSL	DATA	0F1H	; ADC offset calibration component low byte. Default value 0x00.
ADCOFSH	DATA	0F2H	; ADC offset calibration component high byte. Default value 0x00.
ADCGAINL	DATA	0F3H	; ADC gain low byte. Default value 0x00.
ADCGAINH	DATA	0F4H	; ADC gain high byte. Default value 0x00.
ADCCON3	DATA	0F5H	; ADC control (part 3). Default value 0x00.
SPIDAT	DATA	0F7H	; SPI bus access.
SPICON	DATA	0F8H	; SPI control. Default value 0x04. Bit addressable (see below)
SPR0	BIT	0F8H	
SPR1	BIT	0F9H	
CPHA	BIT	0FAH	
CPOL	BIT	0FBH	
SPIM	BIT	0FCH	
SPE	BIT	0FDH	
WCOL	BIT	0FEH	
ISPI	BIT	0FFH	
DAC0L	DATA	0F9H	; DAC 0 value low byte. Default value 0x00.
DAC0H	DATA	0FAH	; DAC 0 value high byte. Default value 0x00.
DAC1L	DATA	0FBH	; DAC 1 value low byte. Default value 0x00.

```
DAC1H      DATA      0FCH ; DAC 1 value high byte. Default value 0x00.
DACCON     DATA      0FDH ; DAC control. Default value 0x04.
```

```
; Enable listings.
;
$LIST
```

Loco.a51

```
; Auto-Tag locomotion.
; 2009 Nathan Blythe
;
; Public routines:
; locoInit: Initialize the locomotion system.
; locoState: Set the state of the locomotion system.
;

; Locomotion states.
;
LOCO_STOP      EQU 0
LOCO_DRIVE_F   EQU 1
LOCO_DRIVE_R   EQU 2
LOCO_SPIN_L    EQU 4
LOCO_SPIN_R    EQU 3

; PWM values for servo speeds.
;
SPEED_R        EQU 012H
SPEED_STOP     EQU 015H
SPEED_F        EQU 018H

; Initialize the locomotion system.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
; Notes:
;   PWM mode 2 is used (8 bit twin PWMs). PWM0 and PWM1
;   are synchronized to start at the same time. Both PWMs
;   have a resolution of 1 / 255.
;
;   See page 43 of the documentation.
;
locoInit:
    ;mov CFG841,
    mov PWM1L, #255          ; Period of both PWMs.
    mov PWM1H, #000H        ; Offset of PWM1's rising edge (PWM0 rises at 0).
    mov PWM0L, #000H        ; PWM0 has initial duty cycle 0.
    mov PWM0H, #000H        ; PWM1 has initial duty cycle 0.
    mov PWMCON, #00100001b ; Mode 2, clock is fosc / 64.
    ret
```

```

; Set the state of the locomotion system.
;
; Takes:
;   A: state
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
locoState:
    push ACC
    push DPH
    push DPL
    clr C
    rlc A
    clr C
    rlc A
    clr C
    rlc A
    mov DPTR, #locoState_JT
    jmp @A+DPTR

; Top of jump table (8 byte entries).
;
locoState_JT:

; Both motors stopped.
;
    mov PWM0L, #SPEED_STOP
    mov PWM0H, #SPEED_STOP
    sjmp locoState_Done

; Heading motor stopped, drive motor forwards.
;
    mov PWM0L, #SPEED_STOP
    mov PWM0H, #SPEED_F
    sjmp locoState_Done

; Heading motor stopped, drive motor reverse.
;
    mov PWM0L, #SPEED_STOP
    mov PWM0H, #SPEED_R
    sjmp locoState_Done

; Heading motor forwards, drive motor stopped.
;
    mov PWM0L, #SPEED_F
    mov PWM0H, #SPEED_STOP
    sjmp locoState_Done

; Heading motor reverse, drive motor stopped.
;
    mov PWM0L, #SPEED_R
    mov PWM0H, #SPEED_STOP
    sjmp locoState_Done

; All done.
;
locoState_Done:
    pop DPL
    pop DPH

```

```
pop ACC
ret
```

RNG.a51

```
; 8-bit random number generator.
; 2009 Nathan Blythe
;
; Public routines:
;   rngInit:   Initialize and seed the RNG.
;   rngGet:    Retrieve the next RNG value.
;
```

```
; LCG parameters.
```

```
;
RNG_A_3 EQU 000H
RNG_A_2 EQU 019H
RNG_A_1 EQU 066H
RNG_A_0 EQU 00DH
RNG_C_3 EQU 03CH
RNG_C_2 EQU 06EH
RNG_C_1 EQU 0F3H
RNG_C_0 EQU 05FH
```

```
; State.
```

```
;
DSEG
  rngState: ds 4
  rngTemp0: ds 4
  rngTemp1: ds 4
  rngTemp2: ds 4
  rngTemp3: ds 4
```

```
; Routines follow.
```

```
;
CSEG
```

```
; Initialize the RNG.
```

```
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   Nothing
;
  rngInit:
    mov rngState + 0, #0
    mov rngState + 1, #0
    mov rngState + 2, #0
    mov rngState + 3, #0
    ret
```

```
; Add argument C to the state.
```

```
;
; Takes:
```

```

; Nothing
;
; Returns:
; Nothing
;
; Mangles:
; A
;
rngAdd:
    mov A, rngState + 0
    add A, #RNG_A_0
    mov rngState + 0, A

    mov A, rngState + 1
    addc A, #RNG_A_1
    mov rngState + 1, A

    mov A, rngState + 2
    addc A, #RNG_A_2
    mov rngState + 2, A

    mov A, rngState + 3
    addc A, #RNG_A_3
    mov rngState + 3, A

    ret

; Multiply register A against state.
;
; Takes:
; A
; R0: points to storage space for result
;
; Returns:
; Result at R0
;
; Mangles:
; Nothing
;
rngMulB:
    push ACC
    push B
    mov A, R0
    push ACC
    mov A, R1
    push ACC
    mov A, R2
    push ACC

; Point R1 to the state.
; Carry byte initially 0.
; Count initially at 4.
;
    mov R1, #rngState
    mov R2, #0
    mov B, #4

; A times byte n + carry byte (n - 1).
; Generates byte n and carry byte (n).
;
rngMulB_Loop:
    push B

```

```

    push ACC
;
    mov A, @R1
    inc R1
    mov B, A
    pop ACC
    push ACC
;
    mul AB
    add A, R2
    jnc rngMulB_nc
    inc B
rngMulB_nc:
;
    mov @R0, A
    inc R0
    mov R2, B
;
    pop ACC
    pop B
    djnz B, rngMulB_Loop

; All done, unroll and return.
;
    pop ACC
    mov R2, A
    pop ACC
    mov R1, A
    pop ACC
    mov R0, A
    pop B
    pop ACC
    ret

; Multiply argument A against the state.
;
; Takes:
;   Nothing
;
; Returns:
;   Nothing
;
; Mangles:
;   A
;
rngMul:
    push B
    mov A, R0
    push ACC

; A.0 * State -> Temp0
;
    mov R0, #rngTemp0
    mov A, RNG_A_0
    call rngMulB

; A.1 * State -> Temp1
;
    mov R0, #rngTemp1
    mov A, RNG_A_1
    call rngMulB

```

```

; A.2 * State -> Temp2
;
mov R0, #rngTemp2
mov A, RNG_A_2
call rngMulB

; A.3 * State -> Temp3
;
mov R0, #rngTemp3
mov A, RNG_A_3
call rngMulB

; Temp0.0 -> State.0
;
mov A, rngTemp0 + 0
mov rngState + 0, A

; Temp0.1 + Temp1.0 -> State.1
;
mov A, rngTemp0 + 1
mov B, rngTemp1 + 0
add A, B
mov rngState + 1, A

; C + Temp0.2 + Temp1.1 + Temp2.0 -> State.2
;
mov A, rngTemp0 + 2
mov B, rngTemp1 + 1
addc A, B
mov B, rngTemp2 + 0
addc A, B
mov rngState + 2, A

; C + Temp0.3 + Temp1.2 + Temp2.1 + Temp3.0 -> State.3
;
mov A, rngTemp0 + 3
mov B, rngTemp1 + 2
addc A, B
mov B, rngTemp2 + 1
addc A, B
mov B, rngTemp3 + 0
addc A, B
mov rngState + 3, A

; All done.
;
pop ACC
mov R0, A
pop B
ret

; Retrieve the next RNG value.
;
; Takes:
;   Nothing
;
; Returns:
;   A: pseudo-random value
;
; Mangles:
;   Nothing
;

```

```
rngGet:
    call rngMul
    call rngAdd
    mov A, rngState + 3
    ret
```

Tic.a51

```
; TIC support.
; 2009 Nathan Blythe
;
; Public routines:
; ticStart: Start the TIC counting for a certain number of ticks.
; ticStop: Stop the TIC counter, whatever it's doing.
; ticISR: TIC interrupt service routine.
;
; Public variables:
; ticTock: Flag indicating that the TIC has tocked.
;

; TIC bits.
;
BSEG
    ticTock: dbit 1

; Routines follow.
;
CSEG

; Start the TIC counting for a certain number of ticks.
;
; Takes:
; A: number of ticks the TIC should count.
; B: 0 = tick in 128ths of seconds; 1 = tick in seconds.
;
; Returns:
; Nothing
;
; Mangles:
; Nothing
;
; Notes:
; Single-shot timer (no auto-restart).
;
; See page 55 in the documentation.
;
```

```

ticStart:
  push ACC

; Disable the TIC and reset current time interval.
;
  mov TIMECON, #00000000b
  call ticSafety
  mov HTHSEC, #00H
  call ticSafety
  mov SEC, #00H
  call ticSafety
  mov MIN, #00H
  call ticSafety
  mov HOUR, #00H
  call ticSafety

; Set the counter limit.
;
  mov INTVAL, A
  call ticSafety

; Reset the event flag.
;
  clr ticTock

; Enable the TIC with the proper timebase.
;
  mov A, #00000011b
  jnb B.0, ticStart_Ready
  mov A, #00010011b
ticStart_Ready:
  mov TIMECON, A
  call ticSafety

; All done.
;
  pop ACC
  ret

; Stop the TIC counter, whatever it's doing.
;
; Takes:
; Nothing
;
; Returns:

```

```

; Nothing
;
; Mangles:
; Nothing
;
ticStop:
    mov TIMECON, #00000000b
    call ticSafety
    ret

; Delay between TIC-accessing instructions.
;
; Takes:
; Nothing
;
; Returns:
; Nothing
;
; Mangles:
; Nothing
;
; Notes:
; See page 44 in the documentation.
;
ticSafety:
    push ACC
    push B
    mov A, #002H
ticSafety_0:
    mov B, #0FFH
ticSafety_1:
    djnz B, ticSafety_1
    djnz ACC, ticSafety_0
    pop B
    pop ACC
    ret

; TIC interrupt service routine.
;
ticISR:
    setb ticTock
    reti

```