FULLY ARMED NXT

ARM Assembly Language Programming
Using the LEGO® MINDSTORMS® NXT

Tat-Chee Wan
FULLY ARMED NXT

ARM ASSEMBLY LANGUAGE PROGRAMMING USING THE LEGO® MINDSTORMS® NXT

TAT-CHEE WAN
For dearest Adele,
to whom I owe much,
and whose forbearance
made this book possible.
4.2 ARM Instruction Set

4.2.1 Standard ARM Instructions
4.2.2 Addressing Mode 1 (Operand2) Syntax
4.2.3 Addressing Mode 2 (Indirect) Syntax
4.2.4 Addressing Mode 3 (Halfword/Signed Byte) Syntax
4.2.5 Addressing Mode 4 (LDM/STM) Syntax

4.2.5.1 LDM/STM Instruction Addressing Submodes
4.2.5.2 Equivalent Syntax for Stack Manipulation
4.2.6 Addressing Mode 5 (Co-Processor)
4.2.7 Pseudo Instructions
4.2.8 Instruction Suffixes

4.2.8.1 Condition Flags and Status Flag Updating
4.2.8.2 Conditional Execution
4.2.9 ARM Instruction Encoding
4.2.10 Using ARM Assembly
4.2.11 Accessing Variables

4.3 Instruction Synthesis

4.3.1 Macros and Instruction Synthesis
4.3.2 GNU Assembler Macro Format
4.3.3 Advantages and Disadvantages of Macros

4.4 Your First Assembly Language Program

4.4.1 Building the First Program
4.4.2 Components of armskel
4.4.3 Debugging the First Program

4.4.3.1 Console based Debugging
4.4.3.2 Eclipse™ CDT Debugging
4.4.4 Modifying the First Program

4.5 Chapter Summary
4.6 Review Questions and Problems

5 The First Step

5.1 Data Movement Instructions

5.1.1 Register Access Instructions
5.1.2 Memory Access Instructions

5.1.2.1 Single Address Location Access
5.1.2.2 Multiple Address Location Access

5.2 Integer Arithmetic Instructions

5.2.1 Addition Instructions
5.2.2 Subtraction Instructions
5.2.3 Multiplication Instructions

5.2.3.1 Dedicated Multiplication Instructions
5.2.3.2 Equivalent Multiplication Instructions

5.2.4 Division Instructions

5.2.4.1 Dedicated Division Instructions
5.2.4.2 Equivalent Division Instructions

5.3 Logic and Bit Manipulation Instructions

5.3.1 Logic Instructions
5.3.2 Shift and Rotate Instructions

5.4 Comparison and Conditional Execution

5.4.1 Comparison Instructions
5.4.2 Condition Flags and their meaning
5.4.2.1 Negative (N) Flag
5.4.2.2 Zero (Z) Flag
5.4.2.3 Carry (C) Flag
5.4.2.4 Overflow (V) Flag
5.4.3 Conditional Execution

5.5 Flow Control Instructions
5.5.1 Gotos using Unconditional Branches
5.5.2 Decision Making Using Conditional Branches
5.5.3 Subroutine Calls using the Link Register
5.5.4 Kernel Mode Access Using Supervisor Exceptions

5.6 Coprocessor Instructions

5.7 Line Following Basics
5.7.1 Introduction to Line Following
5.7.2 Searching for Min and Max values

5.8 Line Follower Robot

5.9 Chapter Summary
5.10 Review Questions and Problems

6 Divide and Conquer

6.1 Introduction to Behavior-Based Robotics
6.1.1 Definition of Behavior-Based Robotics
6.1.2 Decomposing Tasks into Software Modules

6.2 Stacks and Their Use
6.2.1 Idealized Stacks
6.2.2 Memory-based Stack Implementations
6.2.3 Sequence for Using PUSH and POP Instructions

6.3 Calling Routines
6.3.1 Leaf Routine Linking and Returns
6.3.2 Non-leaf Routine Calls and Returns
6.3.3 Recursion

6.4 Parameter Passing Convention
6.4.1 Memory-based Parameter Passing
6.4.2 Register-based Parameter Passing
6.4.3 Stack-based Parameter Passing
6.4.4 Frame-based Parameter Passing / Referencing

6.5 C and Assembly Module Interfacing
6.5.1 Inline Assembly in C
6.5.2 Calling Assembly from C
6.5.3 Calling C from Assembly

6.6 Converting Algorithms into Routines
6.7 Modules and Software Libraries

6.8 Decision Making Basics
6.8.1 Implementing If...Then...Else Statements
6.8.2 Optimizing Simple If...Then...Else Clauses

6.9 Grasper Robot
6.9.1 Behavior-Based Robotics Framework for Grasper
6.9.2 Implementing the Grasper Arbiter
6.9.2.1 Preemptive Multitasking Arbiter
6.9.2.2 Cooperative Multitasking Arbiter
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.9.3</td>
<td>Implementing the Grasper Controller</td>
<td>140</td>
</tr>
<tr>
<td>6.9.3.1</td>
<td>Staggered Processing Example in Grasper</td>
<td>140</td>
</tr>
<tr>
<td>6.9.3.2</td>
<td>Data Structures for Grasper Controller</td>
<td>142</td>
</tr>
<tr>
<td>6.9.4</td>
<td>Software Modules for Grasper</td>
<td>144</td>
</tr>
<tr>
<td>6.9.4.1</td>
<td>General Algorithms Module</td>
<td>144</td>
</tr>
<tr>
<td>6.9.4.2</td>
<td>System Housekeeping Module</td>
<td>145</td>
</tr>
<tr>
<td>6.9.4.3</td>
<td>Common Robot Operation Module</td>
<td>145</td>
</tr>
<tr>
<td>6.9.4.4</td>
<td>Behavior Based Robotics Framework Module</td>
<td>145</td>
</tr>
<tr>
<td>6.9.4.5</td>
<td>Worker Routines Module</td>
<td>145</td>
</tr>
<tr>
<td>6.10</td>
<td>Chapter Summary</td>
<td>146</td>
</tr>
<tr>
<td>6.11</td>
<td>Review Questions and Problems</td>
<td>146</td>
</tr>
<tr>
<td>7</td>
<td>Decision, Decisions</td>
<td>147</td>
</tr>
<tr>
<td>7.1</td>
<td>Arbitration Schemes</td>
<td>147</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Subsumption Architecture</td>
<td>148</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Fixed Priority Hierarchical Arbitration</td>
<td>148</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Managing Data for Behavior Arbitration</td>
<td>149</td>
</tr>
<tr>
<td>7.2</td>
<td>Pointers and Data Structure Access</td>
<td>149</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Memory Access and One-dimensional Arrays</td>
<td>150</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Accessing Elements in Multi-byte Structures</td>
<td>151</td>
</tr>
<tr>
<td>7.2.2.1</td>
<td>Declaring Multi-byte Structures in C</td>
<td>151</td>
</tr>
<tr>
<td>7.2.2.2</td>
<td>Multi-byte Structure Declaration in Assembly</td>
<td>154</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Accessing Elements in Multi-dimensional Arrays</td>
<td>155</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Initializing and Copying Blocks of Memory</td>
<td>157</td>
</tr>
<tr>
<td>7.3</td>
<td>Pattern Matching and Data Manipulation using Pointers</td>
<td>160</td>
</tr>
<tr>
<td>7.3.1</td>
<td>String Initialization and Copying</td>
<td>160</td>
</tr>
<tr>
<td>7.3.2</td>
<td>String Manipulation</td>
<td>161</td>
</tr>
<tr>
<td>7.4</td>
<td>Look Up Tables (LUT)</td>
<td>162</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Simple LUTs Via Direct Lookup</td>
<td>163</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Advanced LUTs</td>
<td>164</td>
</tr>
<tr>
<td>7.5</td>
<td>Decision Making Revisited</td>
<td>166</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Switch Statements Using Jump Tables</td>
<td>167</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Jump Tables with Key Lookup</td>
<td>169</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Advanced Jump Tables</td>
<td>170</td>
</tr>
<tr>
<td>7.5.4</td>
<td>Advantages and Disadvantages of Jump Tables</td>
<td>172</td>
</tr>
<tr>
<td>7.6</td>
<td>Mover Robot</td>
<td>173</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Subsumption Support via Staggered Processing</td>
<td>174</td>
</tr>
<tr>
<td>7.6.2</td>
<td>Suppression and Inhibition Representation</td>
<td>176</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Interleaving Multi-sample Sensor Inputs</td>
<td>177</td>
</tr>
<tr>
<td>7.6.4</td>
<td>Implementing the Dispatcher using Jump Tables</td>
<td>177</td>
</tr>
<tr>
<td>7.6.5</td>
<td>Implementing the Mover Arbiter</td>
<td>177</td>
</tr>
<tr>
<td>7.6.6</td>
<td>Minimizing Actuator Updates in the Controller</td>
<td>178</td>
</tr>
<tr>
<td>7.6.7</td>
<td>Reimplementing the Follow-Line Behavior</td>
<td>178</td>
</tr>
<tr>
<td>7.6.8</td>
<td>Implementing the Move-Object Behavior</td>
<td>178</td>
</tr>
<tr>
<td>7.6.9</td>
<td>Software Modules for Mover</td>
<td>180</td>
</tr>
<tr>
<td>7.7</td>
<td>Chapter Summary</td>
<td>180</td>
</tr>
<tr>
<td>7.8</td>
<td>Review Questions and Problems</td>
<td>181</td>
</tr>
</tbody>
</table>
# 8 Split Personality

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Thumb State</td>
<td>184</td>
</tr>
<tr>
<td>8.2 Standard Thumb Instructions</td>
<td>184</td>
</tr>
<tr>
<td>8.2.1 Data Movement Instructions</td>
<td>186</td>
</tr>
<tr>
<td>8.2.2 Integer Arithmetic Instructions</td>
<td>186</td>
</tr>
<tr>
<td>8.2.3 Logic and Bit Manipulation Instructions</td>
<td>186</td>
</tr>
<tr>
<td>8.2.4 Shift and Rotate Instructions</td>
<td>186</td>
</tr>
<tr>
<td>8.2.5 Comparison Instructions</td>
<td>189</td>
</tr>
<tr>
<td>8.2.6 Flow Control Instructions</td>
<td>189</td>
</tr>
<tr>
<td>8.2.7 Thumb Instruction Encoding</td>
<td>190</td>
</tr>
<tr>
<td>8.3 Calling Between ARM and Thumb Modes (Interworking)</td>
<td>190</td>
</tr>
<tr>
<td>8.3.1 Direct Thumb to ARM Interworking</td>
<td>192</td>
</tr>
<tr>
<td>8.3.2 Direct ARM to Thumb Interworking</td>
<td>193</td>
</tr>
<tr>
<td>8.3.3 Indirect Thumb to ARM Interworking</td>
<td>194</td>
</tr>
<tr>
<td>8.3.4 Indirect Thumb to Thumb Interworking</td>
<td>195</td>
</tr>
<tr>
<td>8.3.5 Indirect ARM to Thumb Interworking</td>
<td>195</td>
</tr>
<tr>
<td>8.3.6 Interworking Macros</td>
<td>195</td>
</tr>
<tr>
<td>8.3.7 The Evolution of Thumb State Support</td>
<td>196</td>
</tr>
<tr>
<td>8.4 Movement Control Basics</td>
<td>197</td>
</tr>
<tr>
<td>8.4.1 Control and Feedback Loops</td>
<td>197</td>
</tr>
<tr>
<td>8.4.2 PID Parameters</td>
<td>199</td>
</tr>
<tr>
<td>8.4.3 Discrete PID Controller</td>
<td>199</td>
</tr>
<tr>
<td>8.4.4 PID Controller Implementation</td>
<td>200</td>
</tr>
<tr>
<td>8.5 Tribot Robot</td>
<td>205</td>
</tr>
<tr>
<td>8.5.1 Changes to Existing Behaviors</td>
<td>207</td>
</tr>
<tr>
<td>8.5.2 Obstacle Avoidance using PID Controls</td>
<td>208</td>
</tr>
<tr>
<td>8.5.2.1 Distance Input Pre-processing</td>
<td>208</td>
</tr>
<tr>
<td>8.5.2.2 PID Controller Operation</td>
<td>209</td>
</tr>
<tr>
<td>8.5.2.3 Motor Actuator Output Clamping</td>
<td>209</td>
</tr>
<tr>
<td>8.5.3 Code Size Reduction via Thumb Interworking</td>
<td>211</td>
</tr>
<tr>
<td>8.5.3.1 Changes To Existing Source Code</td>
<td>211</td>
</tr>
<tr>
<td>8.5.3.2 Code Size Comparison</td>
<td>211</td>
</tr>
<tr>
<td>8.6 Chapter Summary</td>
<td>212</td>
</tr>
<tr>
<td>8.7 Review Questions and Problems</td>
<td>213</td>
</tr>
</tbody>
</table>

# 9 Sensing the Environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 MINDSTORMS Central Nervous System</td>
<td>216</td>
</tr>
<tr>
<td>9.2 Common Microcontroller Features</td>
<td>216</td>
</tr>
<tr>
<td>9.3 I/O Peripheral Logic Modules</td>
<td>216</td>
</tr>
<tr>
<td>9.3.1 Liquid Crystal Display (LCD) / External Display</td>
<td>217</td>
</tr>
<tr>
<td>9.3.2 Keyboard Scanner</td>
<td>218</td>
</tr>
<tr>
<td>9.3.3 Resistive/Capacitive Touchpad Controller</td>
<td>218</td>
</tr>
<tr>
<td>9.3.4 Parallel I/O ports, General Purpose I/Os (GPIO)</td>
<td>218</td>
</tr>
<tr>
<td>9.3.5 External Serial Communications</td>
<td>218</td>
</tr>
<tr>
<td>9.3.6 Internal Serial Communications</td>
<td>218</td>
</tr>
<tr>
<td>9.3.7 Event Measurement and Control</td>
<td>219</td>
</tr>
<tr>
<td>9.3.8 Analog and Digital Conversion</td>
<td>219</td>
</tr>
<tr>
<td>9.3.9 Universal Serial Bus (USB)</td>
<td>219</td>
</tr>
<tr>
<td>9.3.10 Real Time Clock (RTC)</td>
<td>219</td>
</tr>
<tr>
<td>9.3.11 Coprocessors</td>
<td>219</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>9.3.12</td>
<td>Dedicated Hardware-based Codecs</td>
</tr>
<tr>
<td>9.3.13</td>
<td>Programmable Glue Logic</td>
</tr>
<tr>
<td>9.4</td>
<td>Atmel AT91SAM7S I/O Blocks</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Debug &amp; Test Interfaces (JTAG/ICE &amp; DBGU)</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Reset Controller (RSTC)</td>
</tr>
<tr>
<td>9.4.3</td>
<td>Real Time Timer (RTT)</td>
</tr>
<tr>
<td>9.4.4</td>
<td>Periodic Interval Timer (PIT)</td>
</tr>
<tr>
<td>9.4.5</td>
<td>Watch Dog Timer (WDT)</td>
</tr>
<tr>
<td>9.4.6</td>
<td>Voltage Regulator Mode Controller (VREG)</td>
</tr>
<tr>
<td>9.4.7</td>
<td>Memory Controller (MC)</td>
</tr>
<tr>
<td>9.4.8</td>
<td>Embedded Flash Controller (EFC)</td>
</tr>
<tr>
<td>9.4.9</td>
<td>Fast Flash Programming Interface (FFPI)</td>
</tr>
<tr>
<td>9.4.10</td>
<td>Peripheral DMA Controller (PDC)</td>
</tr>
<tr>
<td>9.4.11</td>
<td>Advanced Interrupt Controller (AIC)</td>
</tr>
<tr>
<td>9.4.12</td>
<td>Clock Generator</td>
</tr>
<tr>
<td>9.4.13</td>
<td>Power Management Controller (PMC)</td>
</tr>
<tr>
<td>9.4.14</td>
<td>Parallel Input/Output (PIO) Controller</td>
</tr>
<tr>
<td>9.4.15</td>
<td>Serial Peripheral Interface (SPI)</td>
</tr>
<tr>
<td>9.4.16</td>
<td>Two Wire Interface (TWI)</td>
</tr>
<tr>
<td>9.4.17</td>
<td>Univ. Synch. Asynch. Receiver Transceiver (USART)</td>
</tr>
<tr>
<td>9.4.18</td>
<td>Synchronous Serial Controller (SSC)</td>
</tr>
<tr>
<td>9.4.19</td>
<td>Timer Counter (TC)</td>
</tr>
<tr>
<td>9.4.20</td>
<td>Pulse Width Modulator (PWM) Controller</td>
</tr>
<tr>
<td>9.4.21</td>
<td>USB Device Port (UDP)</td>
</tr>
<tr>
<td>9.4.22</td>
<td>Analog to Digital Converter (ADC)</td>
</tr>
<tr>
<td>9.5</td>
<td>NXT Hardware Architecture</td>
</tr>
<tr>
<td>9.5.1</td>
<td>NXT Built-In Peripherals</td>
</tr>
<tr>
<td>9.5.1.1</td>
<td>AVR Coprocessor</td>
</tr>
<tr>
<td>9.5.1.2</td>
<td>Motor Output Ports</td>
</tr>
<tr>
<td>9.5.1.3</td>
<td>Sensor Input Ports</td>
</tr>
<tr>
<td>9.5.1.4</td>
<td>Liquid Crystal Display</td>
</tr>
<tr>
<td>9.5.1.5</td>
<td>Bluetooth Interface</td>
</tr>
<tr>
<td>9.5.1.6</td>
<td>Universal Serial Bus Interface</td>
</tr>
<tr>
<td>9.5.1.7</td>
<td>Speaker Control</td>
</tr>
<tr>
<td>9.5.2</td>
<td>NXT External Peripherals</td>
</tr>
<tr>
<td>9.6</td>
<td>Chapter Summary</td>
</tr>
<tr>
<td>9.7</td>
<td>Review Questions and Problems</td>
</tr>
<tr>
<td>10</td>
<td>Handle With Care</td>
</tr>
<tr>
<td>10.1</td>
<td>I/O Categories</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Non Real Time I/O</td>
</tr>
<tr>
<td>10.1.2</td>
<td>Real Time I/O</td>
</tr>
<tr>
<td>10.2</td>
<td>I/O Processing Techniques</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Polling I/O</td>
</tr>
<tr>
<td>10.2.1.1</td>
<td>Kernel Level Polling</td>
</tr>
<tr>
<td>10.2.1.2</td>
<td>Application Level Polling</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Interrupt-driven I/O</td>
</tr>
<tr>
<td>10.2.2.1</td>
<td>Handling Interrupts</td>
</tr>
<tr>
<td>10.2.2.2</td>
<td>Minimizing Interrupt Latencies</td>
</tr>
<tr>
<td>10.2.2.3</td>
<td>Interrupt Handler Initialization</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Popular Microprocessors in the Market [1, 2] . . . . . . . . . . 4  
1.2 Differences Between Microprocessors and Microcontrollers . . 5  
1.3 Popular Microcontrollers in the Market [1, 2] . . . . . . . . . . 7  
1.4 Word Sizes for Different Microprocessor Bit-widths . . . . . . 9  
1.5 Bit-Significance and Bit Positions for Binary Values . . . . . 10  
1.6 Bit-Significance and Bit Positions for a 16-bit Binary Value . . 11  
1.7 Division Technique to Convert from Decimal to Binary . . . . 11  
1.8 Sign-Magnitude Format for Representing Negative Values . . 12  
1.9 2’s Complement Numbers from its Unsigned Magnitude . . . 13  
1.10 Calculating the 2’s Complement Value for −7510 . . . . . . . 14  
1.11 Endian-ness of Microprocessor Architectures . . . . . . . . . . 16  
2.1 LEGO MINDSTORMS Tribot . . . . . . . . . . . . . . . . . . 20  
2.2 32-bit Intel ia32 Registers, courtesy of Intel Corporation. [3] . 25  
2.3 32-bit PowerPC Registers, courtesy of Freescale Semi., Inc. [4] 25  
2.5 Memory-mapped vs. I/O-mapped I/O Access [6] . . . . . . . . 27  
2.6 Pipelining Improves a Processor’s Execution Throughput . . . 27  
2.7 Architecture of a Generic Microprocessor . . . . . . . . . . . . 29  
2.8 Architecture of ARM7TDMI Core, courtesy of ARM Ltd. [7] . 34  
2.9 Architecture of an Advanced Microprocessor . . . . . . . . . . 35  
3.1 Accumulator Architecture . . . . . . . . . . . . . . . . . . . . . 40  
3.2 Register-Memory Architecture . . . . . . . . . . . . . . . . . . 40  
3.3 Register-Register Architecture . . . . . . . . . . . . . . . . . . 41  
3.4 Stack-based Architecture . . . . . . . . . . . . . . . . . . . . . 42  
3.5 32-bit ARM Register File, courtesy of ARM Ltd. [7] . . . . . 45  
3.6 Program Status Register (xPSR), courtesy of ARM Ltd. [7] . 46  
4.1 Development Workflow for Assembly Language Programming 54  
4.2 32-bit ARM Instruction Encoding, courtesy of ARM Ltd. [7] . 67  
5.1 Operation of ARM Shift and Rotate Instructions . . . . . . . . 90  
5.2 Layout of xPSR register . . . . . . . . . . . . . . . . . . . . . . 91  
5.3 Line Following Using Single Light Sensor . . . . . . . . . . . . 97  
5.4 Line Follower Robot . . . . . . . . . . . . . . . . . . . . . . . . 104  
6.1 Behavior-Based Robotics Framework [5] . . . . . . . . . . . . 114
# List of Tables

1.1 Temperature Classifications for Electronics Products .......................... 8  
1.2 Decimal, Binary and Hexadecimal Numbers ........................................ 12  
1.3 Bit-patterns for 8-bit 2's Complement Numbers .................................. 14  
2.1 Comparison of Biological Senses vs. NXT Sensors ................................. 21  
2.2 ARM Instruction Set vs. Hardware Generation ........................................ 32  
3.1 AAPCS Register Usage Convention .................................................. 47  
3.2 Nomenclature Mapping for ARM Addressing Modes ............................... 49  
4.1 Logical vs. Hardware Behavior Descriptions of Instruction Sets ................. 53  
4.2 ARMv4 Data Movement Instructions .................................................. 58  
4.3 ARMv4 Data Movement Instructions .................................................. 59  
4.4 ARMv4 Integer Arithmetic Instructions .............................................. 60  
4.5 ARMv4 Logic and Bit Manipulation Instructions .................................... 61  
4.6 ARMv4 Comparison Instructions ....................................................... 61  
4.7 ARMv4 Flow Control and Exception Instructions ................................... 61  
4.8 ARMv4 Coprocessor Instructions ...................................................... 62  
4.9 Stack Operation Equivalent Mnemonics .............................................. 63  
4.10 ARM Pseudo Instructions ............................................................... 64  
4.11 Signed and Unsigned Conditional Execution Suffixes ............................ 66  
5.1 Register Access Instructions using Addressing Mode 1 ........................... 81  
5.2 Logic Instructions using Addressing Mode 1 ......................................... 88  
5.3 Range of Light Sensor Readings for Line Following ................................ 97  
6.1 PUSH order for different stack usage conventions ................................. 125  
6.2 Controller Outputs for Individual Grasper Behaviors ............................... 136  
6.3 Activation Triggers for Grasper Behaviors .......................................... 137  
7.1 256 Entry LUT for sqrt() function using Direct Lookup ........................ 163  
7.2 Advanced LUT for sqrt() using Lower Bound Input Bins ....................... 165  
7.3 Actuator Outputs for Individual Mover Behaviors .................................. 173  
7.4 Activation Triggers for Mover Behaviors ............................................ 174  
8.1 Thumbv4T Data Movement Instructions ............................................. 186  
8.2 Thumbv4T Data Movement Instructions ............................................. 187  
8.3 Thumbv4T Integer Arithmetic Instructions ......................................... 188  
8.4 Thumbv4T Logic and Bit Manipulation Instructions ................................ 188
8.5 Thumbv4T Shift and Rotate Instructions .......................... 189
8.6 Thumbv4T Comparison Instructions ............................... 189
8.7 Thumbv4T Flow Control and Exception Instructions .......... 190
8.8 Actuator Outputs for Individual Tribot Behaviors ......... 206
8.9 Activation Triggers for Tribot Behaviors ....................... 207
8.10 Code Size Comparison for Interworking Applications ...... 212
8.11 ARM vs. Thumb Object File Sizes .............................. 213

9.1 Common Actuators and Sensors for NXT ....................... 228

10.1 Comparison of Various I/O Processing Approaches .......... 245

B.1 Cross Development Tools for various Host Platforms ....... 262
B.2 GNU Cross Compiler Toolchains Contents .................... 280
B.3 Comparison of Register Names ................................. 281
B.4 Comparison of Assembler Directives ........................... 282
B.5 Comparison of Linker Directives ............................... 283
B.6 Macros and Conditional Assembly Directives ................. 284
B.7 Common Command Line Options for GAs ...................... 284
B.8 GLd Directives .................................................. 287
B.9 Common Command Line Options for GLd ..................... 288

C.1 Organization of nxos modules .................................. 299

D.1 Web Resources .................................................. 301
Preface

In the future, we won't have computers like what we have today. Instead, they would just exist as part of everyday gadgets, whether they are our multimedia communicator, microwave oven, or even part of our clothing. All these computer-based devices are known as “Embedded Systems.”

The next big thing in microprocessors is not in how fast the Central Processing Unit (CPU) is compared to the previous generation, or how many transistors can be put into a single Integrated Circuit (IC). Instead, the next wave of microprocessors will emphasize small sizes, low power consumption, and just become part of the surroundings (or “embedded”). This is made possible through the development of “System on a Chip” ICs, which are also known as microcontrollers, that hold a processor core, memory, and peripheral control logic for Input and Output (I/O). Embedded systems perform specific tasks, for example, “fuzzy logic” washing machines wash clothes using artificial intelligence algorithms, digital video cameras record and compress video images, etc.

In order to achieve small sizes and lower power consumption, microcontrollers were typically much less powerful compared with microprocessors. However, the emergence of power efficient 32-bit microcontrollers has changed the landscape. Today, microcontroller designs typically consist of a modern microprocessor core coupled with associated I/O logic and RAM in a single package. Nonetheless, most microcontrollers have limited system resources (RAM, I/O ports) compared with what is found in a desktop-based computing system. Application development should be carried out in a much more conscientious manner. C++ and C applications generating 100MB executable images are unaffordable luxuries in a microcontroller-based platform.

Consequently, microcontroller and embedded systems programming are often done in Assembly Language. This book introduces the world of microprocessors, microcontrollers and assembly language programming to students in their final year of Computer Science or Computer Engineering. Assembly Language describes the basic instructions that can be executed atomically by the microprocessor or microcontroller. As the range of available processors in the market is huge, learning the architecture, instruction set and assembly language for all those processors is impossible. In addition, new architectures come to the market constantly. Furthermore, Embedded Systems often favor processor architectures that are different from desktop computer processors due to a variety of reasons (cost, robust-
ness, I/O peripheral support, market share). This makes learning specific processor architectures and assembly languages an \textit{ad hoc} process. The approach taken in this book is to first describe general principles in microprocessors and embedded systems programming, and then illustrate it using the ARM\textsuperscript{®} processor architecture.

The ARM microprocessor core, which is found in the heart of the LEGO\textsuperscript{®} MINDSTORMS\textsuperscript{®} NXT, is used as the basis of this course as it is one of the most widely used CPU architecture in embedded systems, from cellular phones to PDAs as well as SOHO wireless access points and routers. It is a 32-bit RISC processor, with a native instruction set of 32-bit word lengths, and the ability to switch state to use 16-bit instructions to further reduce the object code size. It has a very flexible and powerful instruction set, leading to very good code density despite the fact that it is a RISC processor.

\section*{Acknowledgements}

This book would not be possible without the help and feedback from many people, especially those in the LEGO MINDSTORMS community who have demonstrated their camaraderie and willingness to help someone who’s new to the world of robotics and NXT programming. Although it would be impossible to name every one here, I would like to recognize the following key people who have helped to make the development of the NxOS-Armdebug software possible:

- Sivan Toledo, whose \textit{Bare Metal on the NXT} \cite{13} was the catalyst for the NxOS-Armdebug platform used in this book
- John Hansen, who developed the NXT Enhanced Firmware and cross-platform BrixCC Toolset \cite{14} which supports the download and execution of bare metal programs files
- David Anderson and collaborators, who developed NxOS which was the basis for NxOS-Armdebug platform
- Nicolas Schodet, with his Python-fu, who helped to make the NXT GDB Server a reality with minimal coding on my part
- Linus Atorf and Andy Shaw, who helped me understand various NXT interfacing and general robotic concepts

Tat-Chee Wan
February 2013
In this book, several icons are used to highlight sections in the text that may provide important information, as well as indicate coding examples, exercises, and review information. The icons should help readers locate the relevant information quickly and easily.

### Icon Keys

<table>
<thead>
<tr>
<th>Icon</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>🍀</td>
<td>Valuable Information</td>
</tr>
<tr>
<td>📖</td>
<td>Coding Examples</td>
</tr>
<tr>
<td>🌚</td>
<td>Test Your Knowledge</td>
</tr>
<tr>
<td>🌞</td>
<td>Chapter Summary</td>
</tr>
<tr>
<td>🗂</td>
<td>Programming Exercise</td>
</tr>
</tbody>
</table>

In this book, several icons are used to highlight sections in the text that may provide important information, as well as indicate coding examples, exercises, and review information. The icons should help readers locate the relevant information quickly and easily.
Chapter 1

Fully ARMed (and Dangerous)

Tweedle-dum and Tweedle-dee
Resolved to have a battle,
For Tweedle-dum said Tweedle-dee
Had spoiled his nice new rattle.
Just then flew by a monstrous crow,
As big as a tar-barrel,
Which frightened both the heroes so,
They quite forgot their quarrel.

from “Through the Looking Glass,” Lewis Carroll, 1832-1898

1.1 Learning Embedded Systems using Robots

Embedded Systems are devices or systems which incorporate microprocessors or microcontrollers for performing various functions, such as calculations or executing tasks. Examples of embedded systems include modern microwave ovens, digital music players (commonly known as MP3 players), Internet enabled cellular telephony devices, as well as robots. Robots are machines which perform various tasks without close human supervision or control. In addition, they often have to handle inputs in real-time, meaning that actions or responses need to be carried out within a given time limit when a certain input is detected. Consequently, robots are excellent examples of Embedded Real-Time Systems.

Creating robots with LEGO® MINDSTORMS® is an exciting process. Unlike other robotic kits, the MINDSTORMS parts are designed for easy assembly and do not require any sawing, cutting, or soldering. In addition, due to the modular nature of the kit, many different types and forms of robots can be created with the same components. This flexibility makes robotics accessible to a wide variety of people, from students to adults who are interested in robotics but not keen on dealing with low level electronics or mechanical components.

The process of robot creation involves several steps, namely:
Robot Design and Construction
NXT Program Development
Interfacing with NXT Sensors and Actuators
Robot Debugging and Testing

Robot Design and Construction involves creating the physical form of the robot using the LEGO Technic™ beams, struts, joints, and axles that is the basis for the MINDSTORMS kit. While this is an integral and challenging part of the robot creation process, it is not the focus of this book. Various books on how to construct interesting robots are available, and you are encouraged to investigate them further if you are new to robot construction using MINDSTORMS. Instead, this book focuses on the other three aspects, namely in Developing NXT Programs using Assembly Language Programming, interfacing with NXT Sensors which detect environmental inputs, and issuing commands to actuators such as Motors, LCD Displays, and the Speaker, as a response. The process of program development also involves Debugging and Testing, which is essential to make sure that the NXT operates correctly. While Robot Debugging includes hardware debugging to ensure that the physical construction of the robot is robust and is able to move in a certain way, the focus of this book again is on software debugging and testing, to ensure that the behavior of the robot is according to specifications and requirements. Consequently we will focus mainly on the NXT Brick which contains the programs that controls the robot.

1.2 Why Bother with Assembly?

Perhaps the first question that crosses someone’s mind when assembly language programming is mentioned is, “Why bother?” There are various high level programming tools that can be used to develop software more quickly and with less effort compared to learning all the details and intricacies of assembly language programming. For the LEGO MINDSTORMS NXT there are already several programming options such as the NXT-G graphical programming environment from LEGO Group, LeJOS for Java programming, and NXC (Not eXactly C) for C-like programming.

However, each of these options assume a virtualized computing device that executes the given application much less efficiently than if the NXT brick were programmed directly using bare metal programming (otherwise called Native Programming, the term used by Sivan Toledo[13]). Other than the proverbial answer of “Because it’s there!” bare metal programming using C and especially Assembly Language allows the programmer to squeeze the highest possible degree of performance out of the NXT brick, something that may determine whether a given MINDSTORMS creation is able to outperform other competitors. Another reason for programming in assembly language instead of using abstract high level languages arises when attempting to optimize algorithms that hit a performance wall. In order to meet the required response time or processing throughput, bare metal programming in a low level system programming language such as
C, or directly in Assembly, is often the only option short of upgrading the microcontroller to a more capable device.

For those interested in learning how the ARM® processor works, as the first step towards working with more advanced embedded systems, using MINDSTORMS with its associated motors and sensor devices as the learning platform is inherently more interesting (to this author, in any case) than making LEDs blink on a typical ARM Developer's Evaluation Board. Lastly, despite the advances in compilers and high level language tools, embedded microcontrollers such as the Atmel® AT91SAM7S256 found in the NXT still have severely limited amounts of on-chip RAM and ROM/Flash. Consequently, the use of high level languages for program development would constrain the feature set that can be incorporated due to the large memory footprints required. In comparison, the use of Assembly Language for bare metal program development would often result in software having smaller memory footprints.

1.3 A Bit of History

Microprocessor generations over the past 40 over years of computing history typically have seen a regular doubling of performance and capabilities based on Moore's Law. The classification of processor generations is typically done using the natural data bit-width of each processor. Therefore, we have 4-bit, 8-bit, 16-bit, 32-bit and now, 64-bit processors. Microcontrollers usually lag behind microprocessors in terms of capabilities and performance, primarily due to the slower pace of change in embedded systems requirements. 4-bit microcontrollers are more than sufficient to control a simple microwave oven. Since users do not purchase a microwave oven based on the processor capabilities but the functions that it provides, there is no need to have the “latest-and-greatest” processor in embedded systems. In addition, embedded systems are designed to be sold in the millions and millions as consumer products. Any unnecessary change in processor design would increase their cost and make the product unattractive to buyers.

However, all this is changing with the emergence of “intelligent devices” that power the digital lifestyles we have today. Tablet computers, Internet connected cellular phones, and other consumer gadgets require processing power that exceeds that found in desktop computers 15-20 years ago. Such devices emphasize portability and low battery power consumption. Consequently, microprocessors that were originally designed to be used with a constantly available power source are ill suited for such portable mobile devices. In this arena, the ARM Architecture has taken significant lead in terms of mindshare and market share, and can be found in almost all Tablet and advanced cellular phones in use today.
1.4 Evolution of Microprocessors

As can be seen from Figure 1.1, microprocessor data-width and capabilities have increased tremendously. Microprocessors have evolved from simple 4-bit processor designs to 64-bit advanced processor designs. In addition, the market has moved from CISC-based designs to RISC-based designs, with advanced processors in the market today incorporating elements of both (e.g., Intel Core, Itanium, Power). Line widths have shrunk from 10 microns to 20 nanometers over the span of 40 years, resulting in an increase in transistor counts (2K to > 1G), achievable clock speeds (< 1 MHz to > 3 GHz), and power consumption (100 mW to > 100W). While this has resulted in the equivalent of supercomputer processing power on the desktop, much of the advances have yet to filter down into the microcontroller-based embedded systems market.

---

1. MC68000 is considered a 32 bit processor due to its use of 32-bit internal data registers.
2. Itanium is a hybrid Very Long Instruction Word (VLIW) architecture.
Microprocessors were found to be very effective in instrumentation and equipment control as well as to automate the functions of electro-mechanical devices. Eventually, as electronic devices become common, the use of microprocessors meant that cumbersome mechanical knobs and levers could be replaced with electronic switches and knobs. Such computer-enhanced devices are known as Embedded Systems. However, microprocessors are not well suited for use within embedded systems, since the devices are often powered only with batteries or else are subject to much harsher environmental conditions. For example, a car with an electronic timing system could be parked outdoors on a hot sunny day, or else be driven up mountains to temperate or freezing temperatures. In addition, microprocessors cannot function by themselves. They require a set of supporting devices in order to have memory and peripheral I/O control lines. These add to the space required to implement a microprocessor based device controller and is not feasible for incorporation into small or portable devices.

1.5 Development of Microcontrollers

As semiconductor miniaturization progressed, more and more circuitry could be placed on the same die as the microprocessor. Microcontrollers were created specifically for the role of embedded systems control. They differ from microprocessors in that memory and peripheral I/O support are integrated onto a single device that also contains the microprocessor core. A typical microcontroller has the following features depicted in Figure 1.2.

In general, the development trend in microcontrollers lags behind that for microprocessors. 4-bit microcontrollers are still in use today in simple devices such as tape-based answering machines, whereas advanced 32-bit microcontrollers are becoming more common due to the introduction of smart cellular-phones, PDAs, as well as digital convergence products such as digital cameras, video recorders and music players.
Figure 1.3 indicates the spectrum of popular microcontrollers in the market. Essentially all popular microcontrollers, regardless of whether they are 4-bit, 8-bit, 16-bit or 32-bit, are still available in one form or another in the market. The need to support the large quantities of consumer good that were designed and continue to be designed using those processors, in addition to the long life cycle of many consumer product categories (e.g., fridges, washing machines, cars), meant that a specific microcontroller remains useful for a much longer duration compared with microprocessors. An example is the United States NASA Space Shuttle, which used computers with Intel 8086 processors to operate its flight control systems.

Currently, most of the advanced microcontrollers are 32-bit designs. Although 64-bit microcontrollers exist, the use of such devices is limited to specialized applications such as high speed routers and high performance computing clusters. In addition, the memory capacity, clock speed and processor complexity of microcontrollers are generally less powerful compared with equivalent microprocessors.

Recent high-end microcontrollers may include advanced features such as built-in Digital Signal Processing (DSP) cores to provide high-speed processing capabilities for multimedia data. Other features that may be present in advanced microcontrollers include embedded DRAM and reconfigurable logic arrays.

Another characteristic of the microcontroller market is that a processor design is often licensed to many semiconductor vendors. The availability of multiple sources results in the creation of many variants of the same processor core, targeting different niche markets (consumer, industrial, automotive, military, etc.). In addition, embedded systems manufacturer prefer to have more than one supplier in order to ensure stability of component supplies and some leverage in component pricing.

1.6 Characteristics of Microcontrollers

The important characteristics of microcontrollers are:

- All-in-one design
- Robustness
- Low power consumption

1.6.1 All-In-One Design

The microcontroller is designed to be self-contained. It should be able to perform most of its functions with only the addition of a few discrete logic parts, capacitors and resistors in the controller board. This helps to save on space, power consumption, as well as improve reliability. Since all computer-based controllers perform their tasks based on some inputs (knobs, buttons, switches, etc.) which cause specific actions to be started and stopped (energize a magnet, switch on a motor, increase the voltage, etc.) and provide feedback to the user via status indicators (lamps, LEDs,
Figure 1.3: Popular Microcontrollers in the Market
Table 1.1: Temperature Classifications for Electronics Products

<table>
<thead>
<tr>
<th>Classification</th>
<th>SD-18 Nomenclature</th>
<th>Usage Environment</th>
<th>Temperature Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>Protected</td>
<td>Home/Office/Indoor</td>
<td>0°C to +70°C</td>
</tr>
<tr>
<td>Industrial</td>
<td>Normal</td>
<td>Industrial/Outdoor</td>
<td>−40°C to +85°C</td>
</tr>
<tr>
<td>Military (MIL-Spec)</td>
<td>Severe</td>
<td>Military/Extreme</td>
<td>−55°C to +125°C</td>
</tr>
</tbody>
</table>

LCD displays, etc.), the microcontroller must contain a CPU core, permanent storage (ROM, EEPROM, Flash), read-write memory (SRAM), and I/O controls (Interrupts, timers, I/O lines, etc.). Microcontrollers are the predecessor to “System-on-a-chip” designs that seek to shrink a desktop PC onto a single IC die. 32-bit microcontroller designs are often created based on this goal, since having a desktop processor core in a microcontroller device means that development and debugging of software could be performed on desktop PCs and then finally placed on the microcontroller. However, not many microprocessor cores are able to meet this requirement due to power consumption constraints.

1.6.2 Robustness

Robustness is of vital importance in embedded systems. Robustness refers to the ability of the embedded system to tolerate variations in temperatures, mechanical vibrations, as well as variations in I/O voltages and currents.

Generally desktop computers and microprocessors do not control critical equipment that may cause injury or death if they crash. However, embedded systems used for control, instrumentation and medical purposes have to operate in a reliable manner to avoid mishaps.

The de-facto classification for temperature robustness is based on Table 1.1, which is documented in the US Department of Defense’s SD-18 “Part Requirement and Application Guide” [15]: Microprocessors are generally fabricated using Consumer specifications, since they are normally used in desktop computers or servers located in air-conditioned buildings. However, embedded systems are used both indoors and outdoors, often in unpredictable weather conditions. Consequently industrial (and similarly, automotive) specifications require a greater temperature tolerance in the components used, especially for the microcontroller that determines the correct operation of a given system. Finally, military specifications used in designing equipment for extreme environmental conditions, including military use.

1.6.3 Low Power Consumption

Low power consumption is critical for embedded systems that are portable or run on batteries. Microcontrollers often have several power-saving
modes that shut off on-chip components (such as I/O interfaces) that are not in use, to further reduce power drain. A microcontroller in deep sleep mode, for example, may consume < 1 mW of electricity. The capability of microcontrollers to control power consumption far exceed that available in microprocessors, which usually operate at higher clock frequencies, leading to higher power drain.

1.7 Some Essential Definitions

In order to properly understand and categorize microprocessors and microcontrollers, we need to define the following terms.

1.7.1 Bits, Nibbles, Bytes, and Words

The most basic unit of data stored in a computer that can be processed by a microprocessor is a bit (binary digit). Bits are grouped together for processing, 4 bits form a nibble, while 8 bits form a byte. Bytes are the minimum sized chunk of data that can be stored to and accessed from memory. Multiple bytes grouped together form a word. Microprocessors are categorized based on the natural word size of its registers. For example, a 16-bit processor has a natural word size of 16 bits (2 bytes), while a 64-bit processor has a natural word size of 64-bits (8 bytes).

Since the word-size is processor dependent, processors often define additional terms such as half word, double word, and quad word to describe different multiples of bytes. For example, a 16-bit processor defines a double word as 32 bits (4 bytes), while a 32-bit processor defines a half word as 16 bits (2 bytes). Nonetheless, 8-bit processors always define words to have 16-bits (2 bytes) (Figure 1.4).
1.7.2 Number Systems and Bit-significance

Number systems in arithmetic represent the way in which we count. For humans, a decimal (base-10) number system is natural since we often use fingers on our two hands to perform counting. However, in computers and microprocessors, the most basic unit is a binary digit (bit) that can be either ‘1’ or ‘0’. This leads to the use of a binary (base-2) number system.

Each group of bits in a processor is a numerical value that represents data stored in the system. The significance or weight of a bit refers to the implicit value assigned to the bit within a particular group. For example, an 8-bit group (one byte) defines a Most Significant Bit (MSB) and a Least Significant Bit (LSB) that represents different values in binary. We usually number bit-positions starting from the LSB as bit-0, which is followed by bit-1, bit-2 and so on, until the highest numbered bit or MSB (Figure 1.5).

Therefore, a 16-bit word would have bit positions b0 to b15, where b0 has significance (weight) of 1, while b15 has significance (weight) of 32768. Similarly, a 32-bit word has bit positions b0 to b31, where b31 has significance (weight) of 2147483648 (Figure 1.6).

To illustrate the binary to decimal conversion process, an 8-bit binary number with the value $10110010_2$ is equivalent to:

$$10110010_2 = (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

$$= 128 + 32 + 16 + 2$$

$$= 178_{10}$$

Conversely, we can use integer division to obtain the binary equivalent of a decimal number as shown in Figure 1.7. The remainder from each step of the integer division process forms the digits of the binary number. Therefore, $178_{10} = 10110010_2$ by reading the value of the remainders from top (MSB) to bottom (LSB).

An n-bit binary number has a range of $[0, 2^n - 1]$. For example, an 8-bit binary number can have a value from $0_{10}$ ($00000000_2$) to $255_{10}$ ($11111111_2$). Nonetheless, binary numbers are difficult for humans to work with. If we group 4-bit values together, forming nibbles, we create hexadecimal numbers. Hexadecimal (base-16) numbers are typically used to describe data values used by computers. Therefore, instead of $10110010_2$ we can write...
Figure 1.6: Bit-Significance and Bit Positions for a 16-bit Binary Value

Figure 1.7: Division Technique to Convert from Decimal to Binary
Table 1.2: Decimal, Binary and Hexadecimal Numbers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

For hexadecimal values beginning with an alphabet (A-F) instead of a digit, we usually prefix the value with a 0 digit in Assembly Language programs to distinguish it from a variable name or label. Therefore, B2\textsubscript{16} would be written as 0B2\textsubscript{h} or 0xB2 (C-style notation). The conversion among decimal, binary and hexadecimal numbers is given in Table 1.2.

1.7.3 Signed & Unsigned Numbers, 1’s & 2’s Complement

Microprocessors perform arithmetic operations on binary numbers using their bit-significance to differentiate between larger and smaller numbers. However, bit-significance only caters for non-negative integer values. Therefore, a mechanism must be devised in order to represent negative numbers. We can reserve one bit in an n-bit data register or memory location to capture the sign (positive, negative) of the number, and use the rest to represent the magnitude based on bit-significance. In general, the MSB is used as the Sign (S) bit. S=1 represents a negative value while S=0 represents a positive value, to maintain compatibility with positive (unsigned) numbers that are able to use all available bits for representing the magnitude. Therefore, this Sign-Magnitude format for binary numbers shown in Figure 1.8 allows us to represent positive and negative numbers.

For hexdecimal values beginning with an alphabet (A-F) instead of a digit, we usually prefix the value with a 0 digit in Assembly Language programs to distinguish it from a variable name or label. Therefore, B2\textsubscript{16} would be written as 0B2\textsubscript{h}, while 37\textsubscript{16} would be written as 37\textsubscript{h}. 
The range of values is now \([-2^{n-1} - 1, 0] \cup [0, +2^{n-1} - 1]\). Note that we have -0 and +0 represented as 10...0000₂ and 00...0000₂ respectively. For an 8-bit number, the possible values would be -127 to +127.

Negative values can also be represented using 1’s Complement format. The 1’s Complement of a number is obtained by inverting all bits of the binary number. The MSB can still be used to determine the Sign of the number. In this case, +0 = 00000000₂, while −0 = 11111111₂.

While the Sign-magnitude or 1’s Complement format is able to represent positive and negative values, it poses several difficulties. First, it is wasteful to have two numbers representing 0 value. Second, -0 and +0 are mathematically identical, but the binary values differ. This means that the microprocessor must have additional logic circuits to handle these mathematical similarities when performing comparisons, arithmetic and other functions. Fortunately, an elegant solution to this problem is available through the use of 2’s Complement numbers.

2’s Complement numbers use the MSB as the Sign bit in a similar way to indicate whether the value is positive or negative. However, instead of storing the magnitude of the value in the normal manner (either sign-magnitude or 1’s complement format), it stores it in 2’s Complement format. 2’s Complement format for a negative value is obtained by first taking the value as an (n-1)-bit unsigned number, where the Sign bit S=0, then inverting all n-bits of the number (resulting in a 1’s Complement number), and lastly, adding one to the 1’s Complement number to give the 2’s Complement number, as shown in Figure 1.9.

For example, \(-75\)₁₀ in 8-bit 2’s Complement format is calculated as shown in Figure 1.10.

---

3Early computer systems actually use 1’s Complement (all the bits in the magnitude part are inverted to give the magnitude for negative numbers) [16][17]. In any case, both Sign-Magnitude and 1’s Complement formats have similar characteristics.
Figure 1.10: Calculating the 2’s Complement Value for $-75_{10}$

Table 1.3: Bit-patterns for 8-bit 2’s Complement Numbers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-128</td>
<td>1000 0000</td>
</tr>
<tr>
<td>-127</td>
<td>1000 0001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-2</td>
<td>1111 1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111</td>
</tr>
<tr>
<td>0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001</td>
</tr>
<tr>
<td>2</td>
<td>0000 0010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>127</td>
<td>0111 1111</td>
</tr>
</tbody>
</table>

Based on the formula in Figure 1.9, we find that $-1 = 11111111_2$, while $-128 = 10000000_2$. We can also determine the magnitude of a negative 2’s Complement number by using the same formula: Take the 2’s Complement number, invert all the bits, and add 1 to the value to get the unsigned magnitude of the negative number.

An $n$-bit 2’s Complement binary number has a range of $[-2^n, +2^n - 1]$. For an 8-bit number, the possible values would be -128 to +127 (Table 1.3). The elimination of −0 enables the definition of an extra negative value.

1.7.4 Binary Addition and Subtraction

Binary addition and subtraction follow the same logic as decimal addition and subtraction. However, since there are only two digits in binary numbers, the addition of two ‘1’s result in a carry to the next digit position, i.e., $1_2 + 1_2 = 10_2$.

Binary addition for 2’s complement numbers has an interesting property in that addition is done in exactly the same way whether the numbers are 2’s Complement or unsigned. Therefore, the ALU does not need to distinguish between unsigned and 2’s Complement numbers when performing addition, since the result will be consistent. It is the programmer’s responsibility to ensure that the operands and results are interpreted correctly, i.e., operands and results must all be of the same format. For example, addition of two unsigned numbers gives an unsigned result, while addition of two 2’s Complement numbers gives a 2’s Complement result.\(^4\)

\(^4\)If the addition of two 2’s Complement numbers with identical signs (both positive or both negative) gives a result with the opposite sign (result negative when operands are positive, or vice-versa), it means that an arithmetic overflow has occurred, where the result exceeded the bit-width of the register or memory location.

14
Subtraction in microprocessors also takes advantage of the property of 2’s Complement numbers. Instead of a dedicated subtractor to perform the subtraction of two numbers x and y, the processor generates the 2’s Complement of y and then perform an addition: i.e., \( x - y \Rightarrow x + (-y) \).

### 1.7.5 Endian-ness

The endian-ness of a processor architecture refers to the way in which data words are stored in memory. Each memory location is identified by a unique memory address, which typically stores a single byte (8 bits) of data. Since a word consists of multiple bytes, the processor architecture must first determine the order in which bytes in a given word is stored in memory. Bytes in a word can also be classified based on their significance. The Most Significant Byte (MSByte) contains the 8 highest order bits of the word (bits (n-8) to (n-1)), while the Least Significant Byte (LSByte) contains the 8 lowest order bits of the word (bits 0-7).

Microprocessor architectures are divided into two categories, based on whether they are big-endian or little-endian. In a big-endian architecture, the MSByte of a word is stored in the lowest memory address, while the LSByte is stored in the highest consecutive memory address according to the number of bytes in the word. On the other hand, little-endian architectures store the LSByte in the lowest memory address, while the MSByte is stored in the highest consecutive memory address (Figure 1.11).

The endian-ness of a processor does not affect its normal operation when data is accessed as words from memory. However, it is important for programmers to pay attention to how data values are stored in memory, especially when accessing data stored in words one byte at a time. This ensures that the byte-significance of the value is preserved. Otherwise, incorrect computations will occur if the MSByte of a big-endian word is used as the LSByte of a little-endian word.

Another area in which endian-ness requires special attention is in the transfer of binary data between systems with different processor architectures. This is especially critical in network-based communications, where systems need to interoperate correctly. Typically, a standard known as network byte ordering is specified to ensure the correct transfer of binary data through computer networks.

### 1.8 Chapter Summary

- Microprocessors and Microcontrollers are closely related technologies.

- Microprocessor development has resulted in increasing clock speeds, complexity and capabilities over the past 40 years. However, these advances have not been adopted by most Microcontrollers due to the differing design requirements: All in One, Robustness, Low Power

- Binary and Hexadecimal number systems are used extensively in Computer and Processor Programming. Negative values can be repre-
Figure 1.11: Endian-ness of Microprocessor Architectures
sented in several ways, but computers typically use 2’s complement format.

- The Endian-ness of a processor determines how multi-byte values are stored in memory. It is important to note the endian-ness of a processor when processing multi-byte values one byte at a time.

1.9 Review Questions and Problems

1. Search the Internet for datasheets for examples of each of the following microcontrollers: 4-bit, 8-bit, 16-bit and 32-bit, and determine what processor core is contained within the microcontroller.

2. Convert the following unsigned decimal numbers to (i) binary (ii) hexadecimal, and determine whether they fit in 8-bit or 16-bit registers:
   - (a) 100  (b) 259  (c) 1015
   - (d) 1    (e) 2501  (f) 200

3. Convert the following signed decimal numbers to (i) binary (ii) hexadecimal, and determine whether they fit in 8-bit or 16-bit registers:
   - (a) -1   (b) -129  (c) -2048
   - (d) -10  (e) -65535 (f) -128

4. Convert the following unsigned 8-bit and 16-bit binary or hexadecimal values to decimal:
   - (a) 1011 0011b  (b) 0F7h  (c) 0001 1111b
   - (d) 1010 0111b  (e) 3615h (f) 0FAB3h

5. Determine which value in each pair of number is larger, given that they are signed 8 bit numbers:
   - (a) 51, 01011100b  (b) 0EAh, -100  (c) 11000101b, 10h
   - (d) -1, 80h        (e) 126, 7Fh  (f) 00110100b, 10010010b

6. Determine which value in each pair of number is larger, given that they are unsigned 8 bit numbers:
   - (a) 51, 00101100b  (b) 215, 08Bh  (c) 23h, 00110101b
   - (d) 127, 82h       (e) 254, FEh  (f) 01110001b, 11010000b
Chapter 2

Dissecting the Beast

‘If you only had brains in your head you would be as good a man as any of them, and a better man than some of them. Brains are the only things worth having in this world, no matter whether one is a crow or a man.’

said by an old crow to the Scarecrow, from “The Wonderful Wizard of Oz,” L. Frank Baum, 1856-1919.

2.1 Anatomy of a LEGO MINDSTORMS Robot

LEGO MINDSTORMS Robots come in all shapes in sizes, from humanoid to insectoid shapes. However, since wheels are typically used for movement in many robots, it would be interesting to examine one particular robot from the Retail 1.0 Kit, the Tribot (Figure 2.1), in greater detail. It is possible to construct a robot which is similar to the Tribot using the LEGO MINDSTORMS Education Kit, though some changes are needed due to the different parts found in that set. In addition, an updated Retail 2.0 Kit has been released in August of 2009 with parts for building different robots. Nonetheless, the NXT brick, which is the core of the LEGO MINDSTORMS kit, is identical in terms of its hardware, with only a firmware update (a brain transplant, if you will) as the only difference. In any case, we can dissect Tribot into the following components:

- Central & Peripheral Nervous System: NXT brick and Connector Cables
- Skeletal Framework: Technic beams, joints, and connectors
- Senses: MINDSTORMS Sensors
- Muscles: MINDSTORMS Servo Motors and gears
- Communications: Speaker, Bluetooth® and USB interfaces
2.1.1 The Central & Peripheral Nervous System

The NXT brick is the Central Nervous System (CNS) of any MINDSTORMS robot. Without it, the robot is just a collection of movable parts that cannot function independently. Similar to the CNS of an animal, there is a brain, which is based on an ARM Microcontroller, that issues all the relevant commands to the muscles (motors) which enable the robot to move, as well as process inputs from the various senses (sensors). While there is no literal spinal cord present in the NXT, it does have a Peripheral Nervous System (PNS) comprising of various Connector Cables which connect the CNS to the various senses and muscles. These cables enable the NXT brick to receive input data from the sensors, as well as generate output commands to the motors.

The ARM Microcontroller contains a Reduced Instruction Set Computer (RISC) processor that is widely used in portable electronic devices such as smartphones, Portable Digital Assistants, and even industrial electronic devices. We will learn more about the features of the ARM processor and microcontroller in the following chapters.

2.1.2 Skeletal Framework

The skeletal framework provides the support and structure to an animal. Similarly, various beams, joints and connectors enable us to create a robot that has a given shape and with the ability to pick, hold or carry objects in the environment. Since constructing robot structures is not the focus of this book, you should refer to the User Guide in the LEGO MINDSTORMS Education Kit, or other books and guides for more information on how to
Table 2.1: Comparison of Biological Senses vs. NXT Sensors

<table>
<thead>
<tr>
<th>Biological Sense</th>
<th>NXT Sensor</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch</td>
<td>Touch</td>
<td>Basic On/Off detection only</td>
</tr>
<tr>
<td>Hearing</td>
<td>Sound</td>
<td>Measures the intensity (loudness) of the sound only</td>
</tr>
<tr>
<td>Smell</td>
<td>N/A</td>
<td>No sensor available</td>
</tr>
<tr>
<td>Sight</td>
<td>Light, Color</td>
<td>Light Sensor detects only the intensity (brightness), Color Sensor detects the Intensity of three primary (R,G,B) colors. Also serves as LED light source.</td>
</tr>
<tr>
<td>Taste</td>
<td>N/A</td>
<td>No sensor available</td>
</tr>
<tr>
<td>Balance and Acceleration</td>
<td>Gyroscope^, Accelerometer^</td>
<td>Third Party Sensor</td>
</tr>
<tr>
<td>Temperature</td>
<td>Temperature^</td>
<td>Third Party Sensor</td>
</tr>
<tr>
<td>Pain</td>
<td>N/A</td>
<td>Pain can be inferred from Pressure</td>
</tr>
<tr>
<td>Kinesthesia</td>
<td>Servo</td>
<td>Motors have servos that report the position of the motor</td>
</tr>
<tr>
<td>Direction</td>
<td>Magnetometer^</td>
<td>Third Party Sensor. Operates similarly to a compass, accuracy affected by magnetic materials in the environment</td>
</tr>
<tr>
<td>Echolocation</td>
<td>Ultrasound</td>
<td>Measure the distance to a solid (reflective) object</td>
</tr>
</tbody>
</table>

create robots with the appropriate features such as limbs, wheels, grabbers, and so on. A list of resources is provided in Appendix D of the book.

2.1.3 Senses

There are five basic senses that we are familiar with: touch, hearing, smell, sight, and taste. In addition, senses such as balance and acceleration, temperature, pain, kinesthesia, direction and echolocation have also been identified though not all these senses are present in humans. While the LEGO MINDSTORMS robot tries to mimic many of the available senses, it is typically not to the degree of sensitivity or sophistication that is achieved in a biological organ. Some senses are functionally complex, for example, hearing, sight and taste generate a lot of data that require sophisticated processing mechanisms in order to make sense of the sensory inputs. A comparison of biological senses with NXT sensors is given in Table 2.1

2.1.4 Locomotion

The LEGO MINDSTORMS NXT brick provides three motor control ports for interfacing with a servo-enabled motor. Unlike biological organisms, movement for LEGO MINDSTORMS robots depends primarily on rotating wheels. Ambulatory motion used by real animals prove to be very complex and difficult to implement, since it requires coordination of many motors
(muscles) and also a sense of balance to prevent the animal from tipping over unexpectedly. Bipedal motion (on two legs) is extremely difficult to implement successfully, and the most recognizable bipedal robot, the HONDA ASIMO [13] requires multiple advanced computers/controllers to coordinate its movements, especially on non-flat surfaces. AlphaRex, one of the MINDSTORMS robots that can be built using the Retail kit, is only able to perform very limited bipedal motion on unobstructed flat surfaces, as well as performing a shuffling/sliding motion instead of a real walking gait. Nonetheless, insect like movements may be achievable since multiple legs provide various points of contact with the ground, but the limited number of servo motors supported by a single NXT brick makes it a challenging design task.

2.1.5 Communications

The LEGO MINDSTORMS NXT has a built-in speaker that can be used to play back recorded sound founds and/or simple tones generated by the microcontroller. In addition, command and control of the NXT robot can be done using Universal Serial Bus (USB) or Bluetooth interfaces. USB is the typical means for programs and other files to be downloaded to the NXT brick. However, since it is a wired connection, it is not convenient for issuing interactive commands to a moving robot since the range of the robot would be limited by the length of the USB cable, as well as potentially getting in the way. Alternatively, the Bluetooth interface can be used to remotely communicate with the robot using suitable Bluetooth enabled devices such as a Personal Digital Assistant (PDA) or Smartphone.

2.2 Key Microprocessor Concepts

As mentioned previously, the NXT brick contains an ARM Microcontroller. Microcontrollers contain a microprocessor core and peripheral control logic (similar to the brain and the central nervous system), which need to be programmed. Without programming, the microcontroller is just an inert piece of electronic circuitry. Since we’re interested in assembly language programming, we need to have a detailed understanding of specific features found in the processor of interest (in our case, this is the ARM processor). This is because the instruction set and available registers for programmer use is specific to a given processor architecture. It is generally not possible to transfer the assembly language program or generated application code of one processor architecture to another for execution without significant effort in rewriting and reassembling the code.

While it is possible to virtualize the processor architecture, and implement a processor emulator application that executes instructions written for the virtual processor on another platform, this approach often require a platform with significantly more computing power and resources compared to the targeted virtual processor itself. In the LEGO MINDSTORMS NXT, user applications developed using NXT-G are also compiled to a byte code based virtual machine instruction set, termed NXT Byte Codes (NBC). This is then executed on the NXT Virtual Machine running as the firmware
application on the NXT brick. Consequently, the achievable performance of
NXT-G and NBC based applications would be an order of magnitude less
than what is possible using native code, whether written in a high level
language or assembly language.

Java byte-codes is another example of a virtualized instruction set exe-
cuting on a virtual processor using a Java Virtual Machine (JVM) processor
emulator. In fact, the original design for Java was to rely on JVM emula-
tors in order to achieve processor independence for applications developed
using that platform. Nonetheless, it is completely feasible to design and cre-
ate a Java processor in hardware (examples include the picoJava processor
logic specification from Sun Microsystems). Such a processor would imple-
ment all the byte-code instructions and registers defined for the JVM, mak-
ing it extremely efficient in executing Java applications. Whether such a
Java processor is a commercial success depends on the usage requirements
and devices that can make use of it. Most processors choose to provide in-
struction set extensions to support new features such as Java byte-codes.
These extensions may be macros (in software) or microcode (in hardware)
that implement the functionality of executing the Java byte-codes using
the available processor instruction set and registers, such as implemented
in the Jazelle (J-variant) ARM cores. While this approach is less efficient, it
protects the investment of existing device designs and assembly language
software that has already been developed for a given processor platform.

2.2.1 CISC vs. RISC

In order to understand the ARM microcontroller used in the NXT brick,
a brief background on microprocessor architectures would be helpful. As
microprocessor architectures evolved, more and more complex instructions
were introduced to simplify the generation of program code for execution on
the processor by high-level language compilers. This led to increased com-
plexity in the microprocessor, in order to handle different instruction types,
data types, and operand sizes. In addition, instructions can take different
number of clock cycles to complete [1]. Consequently, such issues make it
difficult to increasing the achievable throughput of the microprocessor.

Such microprocessor architectures, later known as Complex Instruction
Set Computer (CISC), have the following important characteristics:

- Large and varied Instruction Set
- Small number of CPU registers
- Multiple supported Data Types
- Multiple Memory Access mechanisms
- Variable Length Instructions and Operands
- Variable instruction execution timing
- High Transistor Count
Reduced Instruction Set Computers (RISC) were developed to overcome some of the limitations faced by CISC processors. System designers discovered that in a typical program, a majority of the instructions used come from a few common instructions within the instruction set. This meant that many complex instructions in a CISC processor were underutilized, and could be implemented as an algorithm using the more common instructions instead. RISC architectures have the following common characteristics [1]:

- Uniform Length Instructions and Operands, typically based on the natural word length of the processor
- Standardized execution time, preferably single-cycle instructions
- Limited (reduced) Instruction Set
- Large number of CPU registers
- Limited Data Types
- Limited Memory Access mechanisms
- Lower transistor count, excess transistors used as on-chip caches/RAM

By reducing the complexity of the processor, RISC architectures aim to lower the power consumption, improve reliability, and achieve higher data/instruction throughput compared with CISC designs.

Although modern CISC processors have caught up with RISC processors in terms of their processing power, this is achieved at the expense of very high transistor counts and clock speeds, leading to high power consumption and significant increase in design and validation time to ensure the correct operation of the processor. In addition, advanced microprocessor designs are often hybrid designs, incorporating elements of both RISC and CISC features in the same architecture.

The differences between CISC and RISC architectures are most evident in the Register file design (Figures 2.2, 2.3). When RISC architectures were initially proposed, CISC processors often have limited internal general-purpose registers, since the CPU can access operands stored in main memory using various memory addressing mechanisms, including using the contents of memory locations as a pointer to another memory location. In comparison, RISC architectures have limited memory-addressing mechanisms, typically confined to retrieval of data contents from external memory into internal registers and storage of register values into external memory. Consequently, RISC architectures maintain a large number of internal general-purpose registers in order to minimize the number of external memory accesses, and keep as many variables on-chip as possible to speed up the processing of a given algorithm.

### 2.2.2 Memory Architectures

The classic von Neumann computer architecture uses a single memory address space to store both instruction code and data. This architecture is adopted by many early microprocessors. The Harvard architecture separates the memory space into independent storage areas for code and data.
Figure 2.2: 32-bit Intel ia32 Registers, courtesy of Intel Corporation. [3]

User-Level Registers

<table>
<thead>
<tr>
<th>User Instruction Set Architecture (UISA)</th>
<th>Virtual Environment Architecture (VEA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point registers</td>
<td>Count and Link Registers</td>
</tr>
<tr>
<td>Condition register integer exception</td>
<td>Count register</td>
</tr>
<tr>
<td>Floating-point status and control register</td>
<td>Link register</td>
</tr>
<tr>
<td>Control register</td>
<td>Time-Base Registers (Read-Only)</td>
</tr>
<tr>
<td>Status register</td>
<td>apr 2018 TLB</td>
</tr>
<tr>
<td>Tag register</td>
<td>apr 2018 TBE</td>
</tr>
<tr>
<td>Opcode Register (11-bits)</td>
<td>Time base counter</td>
</tr>
<tr>
<td>Instruction-Accessible Registers</td>
<td>apr 2019 TBE</td>
</tr>
</tbody>
</table>

Figure 2.3: 32-bit PowerPC Registers, courtesy of Freescale Semi., Inc. [4]
This enables the processor to access code and data simultaneously, enabling better memory bandwidth compared with von Neumann designs [5].

Harvard Architectures are typically used in DSPs where high memory bandwidth is needed for real-time data processing. However, advanced microprocessors today often make use of a modified Harvard architecture, where separate Instruction and Data caches (L1 cache) are found internally, to optimize access to a single external memory space (Figure 2.4).

In addition to the distinction between instruction and data memory spaces, processor architectures also differ in the way they access I/O devices. This is termed as memory-mapped and I/O-mapped (also known as port-mapped) [6]. In memory-mapped architectures, I/O device registers used to control and interface with external devices appear as part of the memory address space. I/O devices are accessed in the same way as normal memory locations. On the other hand, I/O-mapped architectures refer to I/O device registers in their own special I/O address space. Consequently special instructions must be used to access I/O device registers for I/O-mapped architectures (Figure 2.5). For example, Motorola® (now Freescale®) processors are Memory-mapped, while Intel® processors are I/O-mapped.

2.2.3 Instruction Pipelining

The execution of microprocessor instructions involves several common stages: Fetch, Decode, and Execute. The number of instruction stages depend on many factors, depending on the design of the processor. Four stage designs with Fetch, Decode, Execute and Store stages are also common [1]. If each of these stages takes one clock cycle to execute, a simple microprocessor
Figure 2.5: Memory-mapped vs. I/O-mapped I/O Access

Figure 2.6: Pipelining Improves a Processor's Execution Throughput

would take three clock cycles to perform the task given in one instruction (assuming a three stage instruction design). Pipelining is a technique used to speed up the process, where these steps are interleaved, so that in the ideal case, once we've started execution of a sequence of instructions, we would be able to complete the task of one instruction every clock cycle (Figure 2.6). In the example, the throughput of the processor increased from 2 instructions in 6 cycles, to 4 instructions in 6 cycles. However, the improvement in throughput obtained via pipelining is limited by the need to implement flow control, such as the execution of alternative sequence of instructions depending on the value of a given variable. If the alternative sequence is required, then a branch penalty occurs, and the pipeline must be flushed, and execution started afresh from the new sequence.

Pipelining throughput in an actual processor depends on the instruction and operand size as well as timing requirements. A processor would
not be able to maintain a full pipeline if some instructions require additional stages to perform their task, while the number of execution cycles may also differ from instruction to instruction. For example, multiply or divide instructions may require several execution cycles to complete its task.

2.3 Core Microprocessor Features

Since each microcontroller contains a microprocessor core, the following discussion is applicable to both. Microprocessors have the following general features (Figure 2.7):

- Instruction Pipeline
- Instruction Decoder
- Instruction Control Unit
- Program Counter
- Arithmetic Logic Unit (ALU)
- Flag Register
- Register File
- Address Control Logic
- Data Bus Access Logic
- Interrupt & Control Logic
- Power & Clock Distribution

2.3.1 Instruction Pipeline

The Instruction Pipeline is responsible for interleaving the execution of sequential instructions, to achieve higher instruction throughput. Most 16-bit or better microprocessors have this feature.

2.3.2 Instruction Decoder

The Instruction Decoder carries out the Decode stage of the processor. It takes the binary data from a given address location and determine which type of instruction the binary value represents, as well as which registers or memory locations will be referenced by the given operands of the instruction. It ensures that the correct data values and contents of memory locations will be retrieved for use by the Instruction Control Unit by the Data Bus and Address Control Logic.
2.3.3 Instruction Control Unit

The Instruction Control Unit (ICU) is responsible for the correct execution of each given instruction. The ICU is a Finite State Machine, which can be implemented as hardwired logic or microprogrammed. Microprogramming refers to the use of microcode to implement the different steps that need to be performed on the ALU in the process of executing a given instruction. Microcode is used solely by the ICU and is not visible to the programmer. Instructions requiring multiple clock cycles (such as multiply or divide instructions) are usually implemented as microcode. Most RISC processors implement hardwired single clock-cycle instructions in order to improve the throughput and performance of the ICU.

2.3.4 Program Counter

The Program Counter (PC) references the instruction to be executed next. It is initialized to the starting address of the memory location containing the application's code, and is modified by the execution of instructions. Instructions that are executed sequentially (one following another) cause the PC to be incremented to the address of the next instruction. Branching instructions cause the PC to be set to the target of the branch location. Branching is important for implementing different logical sequences in the application that is executed depending on the value of specific variables. Nonetheless, branching also causes the CPU to flush the instruction
pipeline and suffer a delay before the next instruction can be retrieved from memory and decoded for execution by the ICU. This is termed the branch penalty.

2.3.5 Arithmetic Logic Unit
The Arithmetic Logic Unit (ALU) is the brain of the CPU. It performs the actual manipulation of data values, whether using arithmetic or logical operations, and generates the result. ALUs operate on integer values, taking one or two operands as input and generating the result as output. It is closely linked to the Flag Register and Register File.

In more advanced processors, hardware multipliers, barrel-shifters, and Floating Point Units (FPU) may be available to provide single clock-cycle performance for complex instructions.

2.3.6 Flag Register
The Flag Register is modified as a result of operations performed by the ALU. For example, if the value of the result exceeds the size of the register, an Overflow is indicated by the Flag register. Other flags contain the results of comparison instructions (greater than, equal to, less than), and sign (positive, negative) of the result.

2.3.7 Register File
The register file is the set of general-purpose registers within the CPU. Registers are used as the source operands of arithmetic and logic instructions, as well as store the results of such instructions. In addition, the ALU usually can only access data that is stored in the register file. Consequently, data in external memory must first be retrieved into the register file, and the results stored back to external memory for use by the application.

2.3.8 Address Control Logic
The Address Control Logic generates the correct physical address on the external address bus in order to activate the corresponding memory location in external memory. Some translation from virtual or logical addresses used by the application program to physical addresses may be necessary if the CPU supports virtual memory.

2.3.9 Data Bus Access Logic
The Data Bus Access Logic interfaces with the external data bus in order to send and retrieve information between external memory and the CPU. It is triggered when external memory access is required, and works closely with the Address Control Logic. The width of the data bus may differ from the natural data width of the CPU (internal bus width). This means that multiple read/write cycles may be required for the retrieval and storage of data words. For example, a 16-bit CPU with an 8-bit external data bus
(e.g., Intel 8088) requires two data access cycles in order to access a word in memory.

The width of the external data bus also affects the memory bandwidth and memory throughput of the CPU. A wide external data bus would provide a higher memory bandwidth compared to a narrow bus.

2.3.10 Interrupt & Control Logic

The Interrupt & Control Logic deals with access to external I/O devices. Devices use interrupt signals connected to the CPU to indicate that they need attention. Interrupts can be prioritized such that some device interrupts may be ignored if the CPU is busy executing important code. The control signals determine which external peripheral device is selected for bus operations.

2.3.11 Power & Clock Distribution

While not strictly a core microprocessor feature, the Power & Clock Distribution circuitry is responsible for delivering required voltage, current and clock signals to each functional block within the processor. Processors with power control capabilities may also enable and disable specific functional blocks within the processor to reduce power consumption.

2.4 ARM Processor Architecture

The 32-bit ARM processor architecture was initially developed by Acorn Computers in the United Kingdom in the mid-80’s. The company subsequently became Advanced RISC Machines (ARM) Ltd. after successfully evolving the ARM architecture to create the ARM6 microprocessor core. ARM Ltd. does not manufacture the processors themselves. Instead, various licensees use the ARM architecture as building blocks to create their own customized version of the processor, mostly for the embedded (microcontroller) market. The ARM architecture has undergone several more iterations, of which ARM7 is the first ARM microcontroller to be widely used in many embedded systems. Subsequent hardware generations (ARM8 to ARM11) were released, with the latest being multiprocessor-capable Cortex cores.

Although the ARM hardware architecture has evolved through multiple generations, the instruction set (assembly language instructions) that is seen by programmers has evolved more slowly. The instruction set that is supported by a given hardware generation is specified by the version number. Consequently, an ARM7 processor uses the ARMv4T (ARM Version 4T) Instruction Set. The 'T' stands for THUMB mode which is a 16-bit instruction subset operating on a fixed subset of the ARM registers. The ARM processor can switch modes easily between 32-bit ARM mode and 16-bit Thumb mode, providing higher code density for embedded applications. Latter versions of the ARM Instruction Set added instructions for Floating Point processing and Vector processing. The newer instruction sets are
Table 2.2: ARM Instruction Set vs. Hardware Generation

<table>
<thead>
<tr>
<th>Instruction Set Version</th>
<th>Hardware Generation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv4T</td>
<td>ARM7TDMI, StrongARM, ARM720T</td>
<td>Thumb mode</td>
</tr>
<tr>
<td>ARMv5</td>
<td>ARM9x6E, ARM926EJ-S, ARM102xE, XScale</td>
<td>Coprocessor extensions Jazelle (Java) mode</td>
</tr>
<tr>
<td>ARMv6T2</td>
<td>ARM1156T2F-S</td>
<td>Thumb-2 mode Vector &amp; Math coprocessors</td>
</tr>
<tr>
<td>ARMv7-AR</td>
<td>Cortex-R Family, Cortex-A Family</td>
<td>Thumb-2EE mode Multi-core processor blocks</td>
</tr>
<tr>
<td>ARMv6-M</td>
<td>Cortex M Family</td>
<td>Thumb2-only mode Embedded applications</td>
</tr>
<tr>
<td>ARMv7-M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>Future</td>
<td>64-bit support Hypervisor mode Legacy 32-bit ARM mode</td>
</tr>
</tbody>
</table>

backwards-compatible, which means that code developed for the ARMv4T would still run unmodified on newer generations of processors.

A brief table mapping the hardware generation to the instruction set version is given in Table 2.2.

The ARM processor is a 32-bit RISC processor which can operate in either big-endian or little-endian mode, controlled via a hardware input signal to the CPU (Figure 2.3). This makes it flexible in embedded systems environment which may be tailored towards different types of applications. There are 15 general purpose registers named R0 to R14 visible to the programmer, along with several processor modes (differentiated as Unprivileged, Privileged and Exception modes), enabling a programmer to separate application code from kernel code used for controlling the operating environment of the processor. The ARM processor implements a 3-stage instruction pipeline, where instructions are Fetched, Decoded and Executed in successive stages, enabling single cycle execution of instructions when the pipeline is filled.

The ARM CPU includes a hardware multiplier and a barrel shifter, which makes it possible to execute integer multiplication, bit shifts, and bit rotates in a single cycle. This is an extremely powerful and useful function in embedded systems which often perform bit manipulation as part of the data processing, making ARM based microcontrollers high performance devices for such environments.

Two of the general purpose registers, R13 and R14, have additional func-

---

1The ARMv6-M and ARMv7-M instruction sets in the Cortex-M Family (meant for cost sensitive embedded applications) have deviated from this, by supporting only the Thumb2 (which is a superset of Thumb) instruction set, dropping ARM instruction support. In addition, the 64-bit ARMv8 instruction set will have streamlined backward compatibility with 32-bit ARM architectures and instructions sets, restricting conditional execution of ARM instructions.
tions. R13 is used as the Stack Pointer by convention, though it is not enforced by the hardware. R14 is the Link Register which contains the return address for a subroutine call, and should not be modified without saving its value during normal processing unless no subroutine calls are performed in the code at all (which is quite unlikely). In addition, these two registers are shadowed in various operating modes (the official term is 'banked registers'), where Exception modes maintain their own private instance of the two registers. This enables the microprocessor to quickly switch from one operating mode to another without having to update the contents of these two registers to mode-specific values. Furthermore, registers R8-R12 are also shadowed in Fast Interrupt (FIQ) Exception Mode to avoid the need to save register contents during Fast Interrupt servicing. Finally, the Current Processor Status Register (CPSR) is shadowed in the Exception modes to enable the processor to determine the cause of the invocation of the Exception mode.

2.5 Advanced Microprocessor Features

As device fabrication technologies improve, the same die area is now able to accommodate additional circuitry due to the use of finer line-widths. Consequently, more and more advanced features are being incorporated into processor designs, to enhance their performance and ability to process new data types. These advanced features were often implemented externally in supporting devices in earlier processor generations. However, newer generations of processors tend to incorporate more and more of these supporting devices into the same core, reducing the power requirement of the system as well as the number of external interconnections between devices. This serves to improve reliability and throughput for the entire system since on-chip communications can occur at the internal processor bus speed instead of external data or I/O bus speed.

Some features of advanced microprocessors (Figure 2.9) include:

- Instruction/Data Cache (L1-Cache)
- Memory Management Unit (MMU)
- Floating Point Unit (FPU)
- Branch Prediction Unit (BPU)
- Vector Processor
- Super-Pipelined/ Super-Scalar Design

2.5.1 Instruction/Data (I/D) Cache

The Instruction/Data Cache is used to store recently used instruction and/or data contents retrieved from external memory. This reduces the need of the CPU to access the external address and data bus, improving the access speed. The Cache may be common (instructions and data share the
Figure 2.8: Architecture of ARM7TDMI Core, courtesy of ARM Ltd. [7]
Figure 2.9: Architecture of an Advanced Microprocessor
same cache), or separate (with individual caches). While older processors do not have caches, most modern processors provide caches to improve CPU performance.

2.5.2 Memory Management Unit (MMU)

The MMU enables processes running on the microprocessor to use logical or virtual addresses to refer to memory locations used within the process. The logical or virtual address space is typically much larger than the available physical address space. In this way, each process can maintain its own set address space that is not accessible from other processes, without worrying about where it is in memory physically. For example, the operating system can be mapped to the same addresses in each logical address space, to simplify the calling of system functions. In addition, the MMU also performs address range locking and enforce access permissions (read, modify) between user and kernel processes. Virtual Memory using secondary (hard disk) storage requires a MMU in order to detect page faults and execute storage access routines to retrieve and store pages from main memory.

2.5.3 Floating Point Unit (FPU)

The FPU (otherwise known as a math coprocessor) implements hardware accelerated floating-point instructions. Instructions that would require significant processing time if implemented as algorithms are executed in a single cycle (assuming a pipelined architecture). For example, multiplication and division of floating point numbers used in Digital Signal Processing can be implemented effectively using FPUs.

2.5.4 Branch Prediction Unit (BPU)

The BPU seeks to reduce branch penalties in the instruction pipeline by pre-fetching the initial instructions from each segment of the branch. In this way, the instruction to be executed next regardless of whether the branch is taken or not, would be available in the pipeline for immediate decoding.

2.5.5 Vector Processor

The Vector Processor provides high speed processing of multiple operands used in multimedia data streams. It is also known as a Single Instruction Multiple Data (SIMD) processor since the same instruction is executed on several operands at the same time. ARMv5 added VFP coprocessor support in the instruction set. In some architectures (e.g., 80x86), Vector Processors are implemented on top of the FPU, while in others (e.g., PowerPC), they are implemented as dedicated processing units. Vector Processors are also increasingly used for processing graphics data, resulting in microprocessors which perform both general purpose computing and graphics data manipulation in the same package.
### 2.5.6 Super Pipelined/Super Scalar Designs

Super-pipelining and Super-scalar microprocessor designs have multiple instruction pipelines and ALUs, FPUs or other computational units. This allows the execution of several different instructions in the same clock cycle, as long as a computational unit is available. Multiple pipelines are needed in order to ensure that a given computational unit is able to proceed with subsequent instructions even though other computational units are busy.

### 2.6 Convergence of CISC and RISC

The traditional division of processor architectures into CISC and RISC started to blur in the 1990’s as advanced fabrication techniques enabled the creation of processors with more and more transistors on die. Consequently, many of the RISC concepts such as regular instruction decoding, simple instructions, and single cycle execution have been adopted even for CISC processors such as the 80x86 microprocessor families. Although the programmer visible processor architecture of the 80x86 microprocessor families remains relatively the same, the internal processor functions have been replaced with micro-operations and other RISC-like processing steps. This improved their performance significantly and in some ways superseded the performance achieved by the early generations of RISC architectures. Consequently, the success of a given microprocessor in the market is no longer dependent on the type of processor architecture but more dependent on marketing, availability from multiple vendors, software compatibility (both binary and source compatibility) and suitability to task.

Other developments such as multiple computing cores per processor die, integration of arithmetic and graphics processing units on the same processor, and asymmetric processor elements which implement processors of different capabilities or instruction sets on the same die, makes the analysis of microprocessors today a complex task. In addition, the emphasis has shifted from purely a performance measurement, typically given in terms of Millions of Instructions Per Second (MIPS), to an efficiency-based measurement, given in terms of performance per watt, given as MIPS/watt. With the widespread use of battery operated portable devices, the microprocessor efficiency becomes even more important.

### 2.7 Chapter Summary

- CISC and RISC represent two different design philosophies for addressing the issue of processor complexity vs. capability.
- The basic components of a microprocessor are presented, and the ARM 32-bit RISC processor architecture is described.
- Some advanced features of modern microprocessors are discussed.
- Modern processors are typically evaluated based on their computational efficiency, and measured using performance per watt as the metric.
2.8 Review Questions and Problems

1. How does a Harvard Architecture processor support the functionality of an Assembler or Compiler, since these programs generate output which is Code and not Data?

2. Describe the evolution of the ARM instruction set features from version 4 (v4) to the current generation.

3. Some RISC architectures implement Windowed Registers consisting of over 100 registers. What is the impact of such a design?

4. What happens if we access a 16-bit halfword using an odd-numbered address, or 32-bit word on an address that is not divisible by four? Why can't this be done easily on most processors?

5. Calculate the Memory Bandwidth needed to support real-time processing of Full High Definition (Full HD) format full color video signal. The data in an entire frame needs to be processed before the next frame arrives. To do this, you would need to determine: bits per pixel, frame dimensions, and frame rate; so that the memory bandwidth (in bits per second) can be determined. What does the effect of different data bus width have on the speed of the data bus?
Chapter 3

NXT Anthropology

‘It’s time for you to answer now,’ the Queen said, looking at her watch: ‘open your mouth a little wider when you speak, and always say “your Majesty.”’

from “Through the Looking Glass,” Lewis Carroll, 1832-1898

The original LEGO Robotic Invention System (RIS) RCX contained an 8-bit Hitachi microcontroller [14]. This has been significantly upgraded in the NXT brick to incorporate a 32-bit Atmel ARM microcontroller as well as an Atmel AVR peripheral controller, which is responsible for coordinating all the input and output processing for the robot. The focus of this book is on programming of the Atmel AT91SAM7S256 ARM microcontroller; the AVR peripheral controller will be treated as an intelligent peripheral attached to the ARM CPU.

In order to understand how the ARM CPU functions, it is important to understand the register and memory architecture of the CPU, as well as how that impacts the way that information is accessed in memory storage locations.

3.1 Registers, Memory and Addressing

The register and memory architecture of a processor is critical in determining the type of operations can be performed on data. If the number of available registers is limited, then algorithms must be written to access memory locations for storage and retrieval of intermediate results. On the other hand, a large number of registers simplify the creation of algorithms since intermediate variables can be kept on-chip in different registers. Various register architectures have been implemented in the past [20], the most common are highlighted in the following section.

3.1.1 Accumulator Architectures

The register file of a microprocessor determines the type of operations and number of temporary variables that are accessible at any given time. Early
microprocessors have very few registers, and are often described as Accumulator Architectures (Figure 3.1). At minimum, an Accumulator register is provided for storing intermediate results of arithmetic and logic calculations. Instructions can access memory locations using fixed addresses as part of the operands. In addition, programs can access any memory location in the address space by using the Index register as an address pointer. The Index register, which has the same size as the Program Counter (PC), provides flexibility in memory access, and can also be used as a counter to control the execution of control loops.

However, such architectures are Memory I/O bound, since most operations that require two operands (e.g., arithmetic instructions: add, subtract, multiply and divide) must refer to contents stored in memory in order perform its task. Early 8-bit microprocessors generally have this limited register file design.

3.1.2 Register-Memory Architectures

In order to enhance the performance of 8- and 16-bit microprocessors, additional data and index registers were added to later designs, resulting in a Register-Memory Architecture (Figure 3.2). E.g., for the Intel 8086 microprocessor, 4 data registers and 4 Index registers are available for program use. This means that the operands and results could all be stored in internal registers to speed up the processing of compute-intensive algorithms. In addition, registers could be combined together to provide access to extended data types such as word- and double-word-sized (16 & 32 bit) operands. For example, the multiplication of two 8-bit numbers gives a 16-bit result, which is stored in two 8-bit registers.
While Register-Memory Architectures refer to the ability of the instruction set to use a register and a memory address location as operands, it is also capable of using two registers as operands. Advanced Register-Memory Architectures can even allow both operands to be memory address locations. However, the important characteristic for Register-Memory Architectures is its ability to mix the use of registers and memory location operands in the same instruction.

Early Register-Memory architectures still differentiate among the various registers. For example, the results of an arithmetic instruction must be placed in a specific register (e.g., the Accumulator), while accessing memory locations via pointer addresses must be done using an index register and not a data register.

However, more advanced Register-Memory architectures such as the Intel 80x86 made the various registers interchangeable. Therefore, the operands of any instruction can take any of the data registers as its source and destination, blurring the distinction between Register-Memory and Register-Register architectures.

### 3.1.3 Register-Register (Load-Store) Architectures

Register-Register or Load-Store Architectures (Figure 3.3) allow access to external memory addresses via specific instructions, namely Load and Store, hence its name. All other instructions use registers as their source and destination operands. This is a major characteristic of RISC processor designs. Since access to external memory is limited, RISC processors often provide a large number of general-purpose registers in order to reduce the number of required memory accesses during the execution of a given algorithm. The ALU can use any general-purpose register as its source and destination operands.

Register-Register Architectures do not distinguish between the different registers. They can contain data values, or address locations to be used as pointers for the Load and Store operations. Register allocation is the responsibility of the programmer or Operating System. Typically a convention is implemented where certain registers are used as pointers, for example, while others are used as operands, to simplify the development of routines and the exchange of data between them.
3.1.4 Stack-based Architectures

Stack-based Architectures (Figure 3.4) \cite{17} are interesting in that no memory addresses are used to reference operands for arithmetic operations. Instead, a Stack Pointer (SP), which is automatically adjusted whenever operands are stored to or retrieved from the stack, is used to access all operands. This means that only the top-most operands are available to the ALU at any given time. Stack-based Architectures are normally used to implement Forth and Postscript based systems, and are no longer found in general microprocessors designs.

3.2 Instruction Operands and Addressing Modes

Microprocessors use Assembly Language instructions to perform computations using its ALU, registers and memory locations. Consequently, instructions have operands that indicate the source and destinations of each instruction. Instructions can have 0, 1, 2, 3, or even up to 4 operands depending on the type of instructions and the processor architecture \cite{17}. Four-operand architectures are not commonly used anymore. The Addressing Mode refer to the type of operand that is used with the instruction:

1. Inherent (Implicit) Addressing
2. Register Locations
3. Immediate (Constant) Values
4. Direct (Fixed) Addressing
5. Indirect (Pointer) Addressing

Typically CISC designs have up to 2 operands per instruction. This is because most arithmetic operations require two source operands to generate a result (e.g., \(A+B \Rightarrow \text{result}\)). The destination operand is usually assumed to be one of the two source operands, therefore the source operand location is modified after the instruction execution to contain the result of the operation (e.g., \(A+B \Rightarrow A\)). The order of the operands for the instruction is therefore important as it affects the way that they are used by the instruction. If the order were reversed, the result would be stored in the other
location instead (i.e., B+A ⇒ B), leading to unexpected consequences if that operand should not change its value.

RISC designs usually have up to 3 operands per instruction, where the destination of an operation can be different from the source operands (e.g., A+B ⇒ C). This provides greater flexibility for RISC architectures since operands can only access registers and not external memory locations.

3.2.1 Inherent (Implicit) Addressing

Inherent Addressing is used when instructions do not require any addresses or register locations to be specified as operands. These zero-operand instructions either do not access register or memory locations; or else the locations are implicit.

Examples (80x86 assembly language):
- NOP (No Operation): A NOP instruction just advances the PC to the next instruction address.
- RET (Return from Subroutine): A RET instruction resumes operation from the point immediately after a subroutine call. The PC is updated automatically with values from the stack.

3.2.2 Register Locations

The instruction's operand refers to a register location. The register's name is specific to a given processor architecture.

Examples (80x86 assembly language, operands given in bold italics):
- INC AX (Increment value in register AX): The instruction operand refers to a register location. The value stored in that register location is incremented (1 added to existing value, and stored in the same register).
- MOV AX, BX (Copy value from register BX to register AX): The order of operands is important, and depends on the convention adopted by the given assembly language. In this example for the 80x86 Assembly Language, the first operand is the destination, while the second operand is the source.

3.2.3 Immediate (Constant) Values

The instruction's operand refers to a constant value that is used as an operand. Consequently, immediate values can only be source operands since a constant value cannot be changed.

Example (80x86 assembly language, operands given in bold italics):
- MOV AX, 013Eh (Store constant value 13E\textsubscript{16} into register AX): The register AX is initialized to the value 13E\textsubscript{16} after the instruction has been executed. This example also illustrates the use of multiple addressing modes in the same instruction, a feature typically found in CISC architectures.

3.2.4 Direct (Fixed) Addressing

For instructions that access external memory locations, Direct Addressing specifies the memory location address to be accessed. This type of addressing mode is not always available in a Register-Register architecture as access to external memory is done using Load and Store instructions.
Example (80x86 assembly language, operands given in **bold italics**):

MOV [0200h], AX (Store value in register AX into memory location 20016): The memory location refers to a specific address in main memory. Square brackets ('[ ' and ']') are used to denote access to external memory in 80x86 assembly language syntax. This address is typically a *logical* address since virtual memory and other addressing schemes may result in a different *physical* address. The 80x86 architecture uses Segment-based addressing to convert logical addresses into physical addresses.

### 3.2.5 Indirect (Pointer) Addressing

For instructions that access external memory locations, Indirect Addressing uses an Index or Data register to contain the memory location address to be accessed. The processor obtains the required value from the register, then configure the address control logic accordingly.

There can be several variants of Indirect Addressing, depending on the complexity of the instruction set. Constant offsets could be added to the pointer value before accessing the external memory location, or two or more pointer values can be used together to specify the required external memory location.

Examples (80x86 assembly language, operands given in **bold italics**):

MOV AX, [BX] (Load value in external memory location pointed to by address value in BX into register AX): Here, register BX is used as an address pointer. It is initialized to the value of the external memory address that we would like to access. The contents of the external memory location with address given by BX is loaded into the destination register AX.

MOV [BX]+5, AX (Store value in AX to external memory location pointed to by address value in BX, with offset of 5): Again, register BX is used as an address pointer. It is initialized to the value of the external memory address that we would like to access. However, the use of constant offsets allows us to use the address pointer in BX as a pointer to a structure or array. The contents of the register AX is therefore stored into the 6th byte location pointed to by the address pointer BX (first location is at offset 0).

### 3.3 ARM Register File and Addressing Modes

At any given time, the ARM processor provides sixteen programmer visible 32-bit wide registers (Figure 3.5), of which 15 registers (R0 - R14) are general purpose register, while R15 is used as the Program Counter (PC). In addition, a flag register, called the Current Program Status Register (CPSR), stores the results of arithmetic and logic operations, as well as hardware interrupt status of the CPU (Figure 3.6). The key feature of the ARM Register File design is the use of Shadow or Banked Registers which are visible only in Exception operating modes. In the ARM Technical Reference manuals, the number of provided registers is given as 31 registers due to this reason (although only sixteen of them are accessible at any given time). Furthermore, the CPSR is copied to a mode-specific Saved Program Status Register (SPSR) every time the CPU switches from the User or System mode to an Exception mode; while the value in the mode-specific SPSR is copied back
to the CPSR when exiting an Exception mode to either the User or System modes.

The use of banked registers provides an advantage over CPU designs that do not have such features. Since Exception modes are invoked when the normal flow of execution is interrupted, it is often necessary for the exception handling routines to preserve the contents of the executing program in order to resume processing from where it was interrupted. The banked registers simplify this process by providing either a private storage location for preserving information from one invocation of a specific exception mode to another, or else they can be used as temporary scratch registers to save the contents of the interrupted program without the need to store those register values into memory, typically into a stack. This is especially useful for the Fast Interrupt (FIQ) Exception Mode that has access to seven banked registers, which allows for the creation of lightweight FIQ handlers that do not need to spend time storing and restoring register values in order to process the interrupt request.

### 3.3.1 Function of ARM Registers

Although all the registers R0-14 are interchangeable and can be used as the source or destination operands for assembly instructions, some registers have predefined usage, which is specified in the Procedure Call Standard for the ARM Architecture (AAPCS) [21]. Notably, R12 is used as the Intra-Procedure call Scratch Register, R13 is used as the Stack Pointer, while R14 is the Link Register that is used to store the return address for a subroutine.
call. Consequently R13 and R14 cannot be used as a data register unless the software does not utilize the stack or make subroutine calls. Although R15 is often used as the source or destination operand in instructions, the modification of R15, which is the Program Counter (PC), has to be done very carefully since modifying the value of the PC will probably cause the program to crash if incorrect values were stored into R15 (PC). The PC should only be modified by program instructions when performing branches or subroutine calls.

### 3.3.2 Procedure Call Standard for the ARM Architecture

The Procedure Call Standard for the ARM Architecture (AAPCS), and ELF for the ARM Architecture (AAELF), defined as part of the ARM Application Binary Interface (ABI) standard [22, 23, 24, 21], specifies the usage of various registers in the CPU so that routines and modules can be written independently and linked together to form the final software application (Table 3.1). In addition, a Full Descending stack is used. The Procedure Call Standard dictates how different high level language (e.g., C) function calls interface with each other and also with assembly language code, whereas the ABIs specify how the executable code is packaged for different execution environments, whether “Bare Metal” systems running custom code or operating systems, or Linux-based Operating Systems using the ELF executable file format.

We will return to this topic when we discuss Subroutine calls in Chapter 6.3.

### 3.3.3 ARM Addressing Modes

The ARM CPU supports the following addressing modes:

- Register Locations
<table>
<thead>
<tr>
<th>Register</th>
<th>AAPCS Usage</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>Argument (a1)</td>
<td>Parameter passing</td>
</tr>
<tr>
<td>R1</td>
<td>Argument (a2)</td>
<td>from subroutine caller</td>
</tr>
<tr>
<td>R2</td>
<td>Argument (a3)</td>
<td>to callee and</td>
</tr>
<tr>
<td>R3</td>
<td>Argument (a4)</td>
<td>for return value</td>
</tr>
<tr>
<td>R4</td>
<td>General / THUMB (v1)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R5</td>
<td>General / THUMB (v2)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R6</td>
<td>General / THUMB (v3)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R7</td>
<td>General / THUMB (v4)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R8</td>
<td>General (v5)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R9</td>
<td>Platform Register (v6/SB/TR)</td>
<td>Must be preserved (if v6 defined)</td>
</tr>
<tr>
<td>R10</td>
<td>General (v7)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R11</td>
<td>General (v8)</td>
<td>Must be preserved Note: Used as frame pointer (fp) by gcc unless -fomit-frame-pointer is specified</td>
</tr>
<tr>
<td>R12</td>
<td>Intra-Procedure-call Scratch Register (IP)</td>
<td>Corruptible</td>
</tr>
<tr>
<td>R13</td>
<td>Stack Pointer (SP)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R14</td>
<td>Link Register (LR)</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>R15</td>
<td>Program Counter (PC)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: AAPCS Register Usage Convention
• Immediate (Constant) Values

• Indirect (Pointer) Addressing

There are no implicit instructions for ARM, all instructions specify register operands as part of the instruction. This reduces the number of instruction encodings.

An important distinction for ARM CPUs is that immediate (constant) values can be encoded as part of the instruction, but with restrictions on allowed values. 32-bit ARM instructions have a 12-bit Immediate field. The 12-bit Immediate field is encoded as an 8-bit value and a 4-bit rotate value. Since the constant value will be stored into a 32-bit register, this means that only a subset of all possible 32-bit constant values could be directly encoded as an instruction operand. Nonetheless, arbitrary 32-bit constant values can still be loaded into ARM registers, by means of Literal Pools, which are memory locations initialized to the required value, and loaded into registers using a load instruction.

As mentioned previously, RISC processors do not have Direct Addressing Modes. Instead all access to the memory storage locations are done through load and store instructions. The load instruction in ARM is called LDR, while the store instruction is STR. The summary of the various address modes is given in Table 3.2.

The Indirect Addressing modes has additional options, for pre-indexing and post-indexing. Pre-indexing is the default addressing mode, which first calculates the effective memory address based on the operands before accessing the actual memory location. In addition, pre-indexing will update the pointer register with the new offset value if writeback (‘!’) is specified, otherwise it keeps the original pointer register value. Post-indexing access the actual memory location based on the contents of the pointer register, then updates the pointer register with the offset value after accessing memory.

In addition, there is also a Mode 5 addressing mode, which is used to interface with coprocessors.

3.4 Chapter Summary

• Different Register Architecture designs result in different memory addressing modes that can support them.

• CISC architectures often support multiple addressing modes, while RISC architectures support only a limited number of addressing modes. However, the available addressing modes for the ARM architecture is more advanced than is typically found in RISC processors, making it highly versatile.

1Allowable constant values are limited to a subset of possible values due to instruction encoding requirements.

2In THUMB state (which uses 16-bit instructions), an instruction can only hold an 8-bit Immediate field.
<table>
<thead>
<tr>
<th>Address Mode (General)</th>
<th>ARM Mode 1</th>
<th>ARM Mode 2</th>
<th>ARM Mode 3</th>
<th>ARM Mode 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Scaled Immediate (restricted range)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Register only (default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with Immediate Shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with Variable (Register) Shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with RRX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word &amp; Unsigned Byte Indirect</td>
<td>Register only (default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word &amp; Unsigned Byte Indirect with Offset</td>
<td>12-bit Immediate Offset</td>
<td>Register Offset</td>
<td>Scaled Register Offset</td>
<td></td>
</tr>
</tbody>
</table>
3.5 Review Questions and Problems

1. Given that the following pseudocode needs to be implemented: Add contents of Register A to contents of Register B, and store the result in Register C, determine which approach is more efficient: two-operand instructions or three-operand instructions?

2. List all the registers present in the ARM processor, the associated processing modes when they can be accessed, and give their respective functions.

3. Explain the difference between Direct and Indirect Addressing modes, and how memory access is performed if Direct Addressing is not available for a given processor.

4. For each of the following processors, determine how many operands are needed for common instructions (such as Addition and Subtraction), as well as what type of addressing modes are provided by the processor for memory access:
   (a) Intel 80x86 processor family
   (b) Motorola (Freescale) 680x0 processor family
   (c) ARM processor family
   (d) MIPS processor family
   (e) Intel 8051 processor family
   (f) Motorola (Freescale) 68HC11 processor family
Chapter 4

Lost in Translation (Not!)

‘My Japanese is getting better. We started speaking English.’
Bob Harris, from the movie “Lost In Translation.”

GETTING the NXT to do what we wish is the main challenge of any MINDSTORMS project. Obviously we’d have to speak the language understood by the NXT itself before that can happen. As mentioned in the beginning of this book, the NXT language most commonly encountered is NXT-G which is the default programming environment provided by LEGO. Nonetheless, NXT-G is only a high level programming interface, the actual language used to specify commands for execution on the NXT is known as NXT Byte Code (NBC). Nonetheless, NBC does not correspond to any specific microprocessor instruction set, notably the ARM processor found in the NXT brick. Instead, it is a virtual instruction set which is executed by the virtual machine software running on the ARM processor inside the NXT brick.

The above example is similar to the situation where someone who only speaks English wishes to talk to someone who speaks only Japanese. This requires the use of interpreters. If NXT-G is English, then NBC is (for illustration), Mandarin. The NBC virtual machine is the interpreter converting Mandarin to Japanese. If someone can speak Mandarin, they would be able to convey messages much more quickly than someone who does not. Nonetheless, since not everyone can converse in Mandarin, additional interpreters would be needed (given that the Mandarin to Japanese interpreter does not understand other languages, and no other interpreter understands Japanese). Fortunately for us, in programming languages, the actual meaning of a command does not get lost when converting from one language to another. Of course, the more layers of interpretation required, the slower messages are conveyed and commands executed.

Hence, programming in NXT-G would require two layers of command conversions to the format understood by the ARM processor, resulting in much slower execution of commands. Other high level languages such as Not eXactly C (NXC), supported using the NXC enhanced firmware [14], still generates commands in NBC but since C is able to specify commands more precisely and directly compared with NXT-G, it is able to generate a shorter list of commands to perform the same action, which would then
execute much faster compared to NXT-G.

The goal of this book is to teach you how to speak Japanese, or more accurately, to use the native language understood by the ARM processor, known as ARM Assembly Language. This allows us to bypass all interpreters and issue commands directly. However, there is a price to be paid. Assembly language, though it is able to execute at the highest speed on the processor, requires us to be very precise and careful in the way commands are issued. There is no 'interpreter' to help deal with the nuances of meaning, we also need to keep track of where each piece of information is stored in memory and in the processor registers ourselves. However, if we make the effort, we can make the NXT perform tasks that would otherwise take too long to complete if programmed in a high-level language. To program in Assembly Language we need to learn the syntax and grammar of the native language, otherwise known as the Processor Instruction Set.

4.1 Instruction Set Characteristics

Instruction sets exposes the capability of the microprocessor architecture to the programmer. Consequently, in order to write programs in Assembly Language, which provide the lowest level of device access to programmers, it is important to understand the given processor instruction set. The type of instructions in a typical Instruction Set can be generalized into:

- Data Movement Instructions
- Flow Control Instructions
- Arithmetic Instructions
- Logic and Bit Manipulation Instructions
- Comparison Instructions

Since Instruction Sets are closely tied to the architecture and design of the microprocessor or microcontroller, each processor may use its own syntax and notation to describe the behavior of operational codes (opcodes) that represent the basic commands that can be executed by that given processor.

Instruction Sets can be understood at two levels: first is at the programming (logical) level, which defines the opcodes and allowable operands and how they can be used to implement the required algorithm or program; while second is at the hardware (device) level, which describes the effect of executing opcodes on the state of the processor. The two levels are closely related, and to achieve a good understanding of the operation of the processor at the logical level often requires examining what happens at the hardware level.

The distinction between the two arise due to the focus of different disciplines in computing, namely computer science vs. computer engineering. Consequently, the terminology used to describe the behavior of the processor differs accordingly.

In this chapter, we will use the term Syntax to refer to logical (programming) descriptions, while Notation will be used to refer to hardware descriptions.
### Table 4.1: Logical vs. Hardware Behavior Descriptions of Instruction Sets

<table>
<thead>
<tr>
<th>Domain</th>
<th>Behavior Description</th>
<th>Focus On</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science</td>
<td>Logical (Programming)</td>
<td>Mnemonics, Opcodes and Operands</td>
<td>Assembly Language Syntax</td>
</tr>
<tr>
<td>Computer Engineering</td>
<td>Hardware (Device)</td>
<td>Machine Code, Register and Processor States</td>
<td>Register Transfer Logic (RTL) Notation</td>
</tr>
</tbody>
</table>

#### 4.1.1 Development Workflow

Programming often starts with an abstract representation of the logic, which can be captured using a Flow Chart. Assembly Language is the text-based (human readable) method of describing the program flow that uses the various opcodes for a given processor architecture (also known as its instruction set) to implement an algorithm. Opcodes refer to the basic instructions that are provided by the processor. This is known as machine code at the hardware level, where opcodes are represented by specific bit patterns stored in a memory location, and interpreted as instructions to be executed by the processor.

Since memorizing machine code is very difficult for most programmers, the opcodes are represented using text strings called mnemonics that describe the function of a specific opcode. The use of mnemonics in specifying program statements allows programmers to create Assembly Language Programs. Consequently, Assembly Language Syntax is very much dependent on the Assembler for the given processor architecture. The Assembler takes an Assembly Language program and converts it into Object Code, which is the binary representation of the opcodes and operands used in a given assembly language program. Even if a programmer uses a High Level Language (such as C), the Compiler or Cross Compiler (which runs on a different CPU architecture than the target system, typically required for embedded systems development) would process the high level language source file and generate a corresponding Assembly Language program that would be used to generate the Object Code.

The object code would then be converted into an executable program using a Linker, which has the main task of resolving references to external variables and generating a binary file suitable for use by the Runtime Loader. For an embedded system such as the NXT, a Downloader application would then be used to transfer the executable program from the development host to the device. When the user wishes to invoke or execute a given executable program, a Runtime Loader is used to setup the program in main memory, and start the processing of the opcodes found in the executable starting from an initial location. This can be summarized in the following way in Figure 4.1.
4.1.2 Assembly Language Syntax

Assembly Language syntax specifies variable and flow control label naming conventions, starting locations in memory for the application, format for opcode mnemonics and associated operands in a given program statement, as well as how macros (used to automatically generate a sequence of statements) can be specified.

A program statement is the atomic unit of executable instruction that can be specified for a given processor in its assembly language. The program statement can consist of an optional flow control label, an opcode, associated operands, and possibly comments documenting the algorithmic task that the statement accomplishes. The ARM Assembly Language syntax follows the convention where the instruction opcode (operation code) is followed by the destination operand, and then, the source operand(s). However, the specific syntax for ARM Assembly Language depends on the assembler used. The examples in this book are based on GNU Assembler (GAs) syntax [25]. The format for an Assembly Language Statement for GAs is as follows:

```
(label:) opcode (dest)(,src1)(,src2)(,src3) (@ comment)
```

‘{’ and ‘}’ refer to optional parts of the statement. In addition, the number of required operands (1, 2, 3 or 4) depends on the particular instruction opcode. Asterisks (*) are used to separate in-line comments from the rest
of the program statement\(^1\), though GAs accepts Shell-style comments prefixed by Hash (‘#’) as well as C-style comments (enclosed between ‘/’ and ‘*/’). The destination operand is typically required except for instructions with implicit addressing mode.

### 4.1.3 Label definition

Labels are alphanumeric names used to define the starting location of a block of statements, and are case sensitive in GAs. They have the same format as Variable names, which are used to identify data storage locations in memory. Labels, like variable names, must begin with an alphabet or one of the following three symbols; period (‘.’), dollar (‘$’) or underscore (‘_’). The rest of the symbol can consist of alphanumeric and period (‘.’), dollar (‘$’) and underscore (‘_’) characters. A label must be followed by a colon (‘:’) suffix when declared, but is referenced in operands using the label name only. Labels must be unique in the executable file. Another identical label encountered by the Assembler will generate an error.

There is another category of labels called Local Labels. These have limited scope and are often used in a routine for references that will not be included in the symbol table (unless overridden via assembler directives). They are useful in macro definitions, to allow branching to labels within macros. Local Labels are defined as single numeric digits in the range \([0 \ldots 9]\), followed by a colon (‘:’). Each local label will be converted into a unique label by the Assembler \(^2\). The scope of a given Local Label is valid until another Local Label with the same digit is encountered, hence they can be non-unique. Local Labels are referenced using the numeric digit only, followed by either a ‘f’ or a ‘b’ to indicate either forward or backward references. Consequently, up to 9 different Local Labels can be referenced in either the forward or backward direction at any given time.

### 4.1.4 Using Immediate (Constants) Values

In GAs, numeric constants or Immediate values are defined according to C syntax. Immediate (also called literal) decimal (Base 10) constants are specified as is (without any leading zeros), binary (Base 2) constants are specified with a ‘0b’ prefix, octal (Base 8) constants are specified with a leading zero ‘0’, whereas hexadecimal (Base 16) constants uses ‘0x’ as the prefix. Immediate values used as operands are prefixed by a Hash (‘#’) symbol, except when the pseudo-instruction version of Load (LDR) is used to load constant values into registers. Literal constants used as operands for LDR are prefixed by an equal (‘=’) symbol.

The code fragment in Listing 4.1 illustrates the syntax used for referencing literal constants; the code increments (add one) the value stored in an external memory location (address 200000\(^\text{16}\)). Note: For the ARM processor, memory access is via Load (LDR) and Store (STR) instructions only.

\(^1\)In other ARM assemblers, the semicolon (‘;’) is more commonly used as a comment separator.

\(^2\)GAs accepts the Dollar (‘$’) prefix for constants as well, though it is not recognized by other ARM assemblers.
Indirect addressing memory references use ‘[’ and ‘]’ as delimiters, while register locations are referenced using their given names (e.g., R0, R1, etc.). The last instruction, Branch (B), causes the program to loop indefinitely.

Listing 4.1: ARM Assembly to increment memory variable (Alg-4.1.S)

START: MOV R1, #0x00200000 @ Setup address pointer
LDR R0, [R1] @ Retrieve contents to R0
ADD R0, R0, #1 @ Increment value by 1
STR R0, [R1] @ Store to addr. 200000h
STOP: B STOP @ Loop forever

4.1.5 Using Labels (Symbolic Address Locations)

The pseudo-instruction LDR is often used with Label (symbolic address location) operands in the code. However, care must be taken when using labels as operands. If a label were used to reference the contents of the location indicated by the label, it is specified as is for the operand (e.g., ‘LDR R0, label’ @ Load Contents stored in location label). However, if the address of the location were required, then the label is prefixed by an equal (‘=’) symbol (e.g., ‘LDR R0, =label’ @ Load Address location of label). Note that the Assembler would only accept labels for referencing contents of locations (i.e., no ‘=’ prefix) as operands only if they were defined in the same code segment. There are no restrictions on the use of address references (i.e., with ‘=’ prefix) which will be stored in Literal Pools.

4.1.6 Register Transfer Logic (RTL) Notation

The RTL notation is used to describe the changes to the processor state (register, flags, etc.) that occurs when an instruction (opcode) is executed. Typically RTL notation is tailored to the processor architecture of interest, since the registers and flags have unique names that are processor-specific. The RTL notation used in the Instruction Set Quick Reference Guides for ARM, THUMB2 and THUMB [26, 27] has the following general format:

\[
\text{result} := \text{operand1 operator operand2}
\]

The valid types for result, operand1, and operand2 depend on the available addressing modes (immediate, register, indirect, etc.). For example, the ADD instruction has the following RTL [26]:

\[
\text{Rd} := \text{Rn} + \text{Operand2} \quad \text{(from ADD Rd, Rn, <Operand2>)}
\]

See Section 4.2.3.

This is illustrated with the following RTL describing change in processor state caused by execution of the ADD instruction:

\[
\text{R1} := \text{R0} + \#2 \quad \text{(from ADD R1, R0, \#2)}
\]

This RTL equation means that the value in register R0 (operand1) is ADDed (operator) to the constant value \(2^8\) (operand2), and the result stored into register R1 (result). To differentiate between different addressing modes

\footnote{Note: The equal symbol is NOT used for constants or labels used as operands for Flow Control and Exception instructions.}
and subsets of operands (e.g., the LSByte of a multi-byte word), the follow-
ing notation can be used on the operands and result:

- \text{operand}[b] \quad \text{Bit number } b \text{ in a multi-bit operand}
- \text{operand}[b_1:b_2] \quad \text{Bit range } (b_1 \text{ to } b_2) \text{ of the operand}
- \text{[register]} \quad \text{Contents of external memory at address}
  \text{referenced by the register (address pointer).}
  \text{Register value contains the address}
  \text{location (indirect address)}
- \text{[address]} \quad \text{Contents of external memory at address}
  \text{referenced by the constant value}

The following are examples of typical addressing modes in RTL notation:

- \text{R4[31]} \quad \text{The MSB of 32-bit register R4}
- \text{R9[7:0]} \quad \text{The LSByte of register R9}
- \text{[R14]} \quad \text{Contents of external memory at address}
  \text{referenced by register R14.}
  \text{Register R14 contains the address location.}
- \text{[0x1000]} \quad \text{Contents of memory at address 1000_{16}}

### 4.2 ARM Instruction Set

#### 4.2.1 Standard ARM Instructions

The ARMv4T Instruction Set consists of 40 general register and 4 Coproces-
sor related 32-bit instructions, which is supported by all subsequent ARM
Instruction Set versions. Consequently, it is the basis for all ARM micro-
controllers in common use today and hence the focus of this book. ARM ar-
chitectures having a ‘T’ suffix also support the 16-bit THUMB instruction
set which operates on a subset of the 16 registers. THUMB instructions
will not be discussed in this chapter.

New instructions introduced with latter ARM Architectures deal mostly
with support for coprocessor and vector processors. Most ARM instructions
that involve register operands (other than multiply and co-processor in-
structions) incur a single clock cycle, while instructions having operands
with register shifts incur an additional cycle\(^4\) and Flow Control instruc-
tions which causes a pipeline flush and require a minimum of three clock
cycles\(^7\). Memory access instructions incur a minimum of two clock cycles
depending on the operands and number of registers transferred. Instruc-
tions are divided into the following categories: Data Movement\(^6\) (Table 4.2
and 4.3), Integer Arithmetic (Table 4.4), Logic and Bit Manipulation (Ta-
ble 4.5), Comparison (Table 4.6), Flow-Control and Exception (Table 4.7),
and Coprocessor (Table 4.8) instructions. Each instruction category will be
covered in greater detail in Chapter 5. See also Sec. 4.2.8 for information
regarding conditional execution of instructions.

\(^4\)Instructions modifying PC (R15) will incur an additional cycle beyond those already stated.
\(^5\)The Move Negative (1’s complement) instruction, although it has a Data Movement
mnemonic prefix, is more correctly a Logic instruction and hence is covered in Section 4.5.
Table 4.2: ARMv4 Data Movement Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
</table>
| Move (Copy)                       | MOV{S}          | Rd, <Operand2>                    | S: Flags Updated; Rd := Operand2  
New UAL Mnemonics for Bit-shifted Operand2                                          |
| Move to ARM Register from Status Register | MRS             | Rd, <PSR>                         | PSR = {CPSR, SPSR}; Only in Exception modes                              |
| Move to Status Register from ARM Register | MSR             | <PSR>, <fields>, Rm               | PSR = {CPSR, SPSR}; fields = {c,x,s,f}; Only in Exception modes            |

### 4.2.2 Addressing Mode 1 (Operand2) Syntax

In the ARM Architecture, Operand2 for Addressing Mode 1 involves Immediate and Register values, and can be specified in the following ways:

- **Scaled Immediate (#<imm8m>), with 4-bit scaling factor**
- **Register Only (Rm), equivalent to (Rm, LSL #0)**
- **Register with Immediate Shift (Rm, <BitShiftOp> #<shift_imm>)**
- **Register with Variable Shift (Rm, <BitShiftOp> Rs)**
- **Register with Rotate Right with eXtend (Rm, RRX)**

where BitShiftOp consists of one of the following: LSL, LSR, ASR, ROR.

MOV instructions having Operand2 involving BitShiftOp are now defined using alternative syntax given in Table 4.10.

### 4.2.3 Addressing Mode 2 (Indirect) Syntax

Addressing Mode 2 is used for Indirect Addressing to Load and Store data from/to memory. The memory address is determined as follows:

- **Immediate Offset ([Rn, #+/-<imm12m>])**
- **Register Offset ([Rn, +/-Rm])**
- **Scaled Register Offset ([Rn, +/-Rm, <BitShiftOp> #<shift_imm>])**
- **Immediate Pre Indexed ([Rn, #+/-<imm12m>])!**
- **Register Pre-Indexed ([Rn, +/-Rm]!)**
- **Scaled Register Pre-Indexed ([Rn, +/-Rm, <BitShiftOp> #<shift_imm>]!)**
- **Immediate Post-Indexed ([Rn, #+/-<imm12m>])**
- **Register Post-Indexed ([Rn], +/- Rm)**
- **Scaled Register Post-Indexed ([Rn], +/- Rm, <BitShiftOp> #<shift_imm>)**

The ‘T’ (Translation) suffix is used for Load and Stores in Exception modes, to force User Mode Restricted Access to memory locations.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
</table>
| Load Register from Memory   | LDR(T)          | Rd, <addr_mode2> | T: Translation - See Sec 4.2.3  
Rd[31:0] := [effective addr] (Word)  
Non Word Aligned access rotates byte at Address to LSByte |
| Load Register Byte from Memory | LDRB(T)      | Rd, <addr_mode2> | T: Translation - See Sec 4.2.3  
Zero Extend Rd[7:0] := [effective addr] (Byte) |
| Load Register Halfword from Memory | LDRH         | Rd, <addr_mode3> | Zero Extend Rd[15:0] := [effective addr] (Halfword)  
Address must be Halfword Aligned |
| Load Register Signed Byte from Memory | LDRSB       | Rd, <addr_mode3> | Sign Extend Rd[7:0] := [effective addr] (Byte) |
| Load Register Signed Halfword from Memory | LDRSH     | Rd, <addr_mode3> | Sign Extend Rd[15:0] := [effective addr] (Halfword)  
Address must be Halfword Aligned |
| Store Register to Memory    | STR(T)          | Rd, <addr_mode2> | PC Relative Addr Not Available  
T: Translation - See Sec 4.2.3  
[effective addr] (Word) := Rd |
| Store Register Byte to Memory | STRB(T)      | Rd, <addr_mode2> | PC Relative Addr Not Available  
T: Translation - See Sec 4.2.3  
[effective addr] (Byte) := Rd[7:0] |
| Store Register Halfword to Memory | STRH        | Rd, <addr_mode3> | PC Relative Addr Not Available  
[effective addr] (Halfword) := Rd[15:0]  
Address must be Halfword Aligned |
| Swap Register with Memory   | SWP            | Rd, Rm, [Rn] | Used for Locking Primitives  
temp := [Rn]; [Rn] := Rm; Rd := temp |
| Swap Register Byte with Memory | SWPB        | Rd, Rm, [Rn] | Used for Locking Primitives  
temp := Zero Extend [Rn] (Byte);  
[Rn] (Byte) := Rm[7:0]; Rd := temp |
| Load Multiple Registers from Memory | LDM<mode> | Rn[!], <creglist-PC> <  
Rn, <creglist-PC>^ | Addressing Mode 4  
mode = [IA|IB|DA|DB].  
Rn cannot be PC nor in reglist.  
^ updates Rn. If PC specified, BX assumed.  
^ loads user mode registers in Exception mode only.  
(“!” + “^”) restores CPSR returning from Exception.  
Use NOP as next instr to avoid race conditions |
| Store Multiple Registers to Memory | STM<mode> | Rn[!], <creglist> | Addressing Mode 4  
mode = [IA|IB|DA|DB].  
^ updates Rn.  
Rn cannot be PC nor in reglist if writeback (“!”) used.  
^ stores user mode registers in Exception mode only.  
Rn should not be updated (“!” + “^”) in Exception mode.  
Use NOP as next instr to avoid race conditions |

59
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>ADD(S)</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn + Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADC(S)</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn + Operand2 + Carry</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB(S)</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn - Operand2</td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SBC(S)</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: Carry = NOT(Borrow)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn - Operand2 - NOT(Carry)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Reverse Subtract</td>
<td>RSB(S)</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Operand2 - Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Reverse Subtract with Carry</td>
<td>RSC(S)</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: Carry = NOT(Borrow)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Operand2 - Rn - NOT(Carry)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Multiply and Accumulate</td>
<td>MLA(S)</td>
<td>Rd, Rm, Rs, Rn</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := ((Rm × Rs) + Rn)[31:0]</td>
</tr>
<tr>
<td>Multiply</td>
<td>MUL(S)</td>
<td>Rd, Rm, Rs</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := (Rm × Rs)[31:0]</td>
</tr>
<tr>
<td>Signed Multiply</td>
<td>SMLAL(S)</td>
<td>RdLo, RdHi, Rm, Rs</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td>Accumulate Long</td>
<td></td>
<td></td>
<td>RdLo := signed((Rm × Rs) + RdHi,RdLo)[31:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RdHi := signed((Rm × Rs) + RdHi,RdLo)[63:32]</td>
</tr>
<tr>
<td>Unsigned Multiply</td>
<td>UMLAL(S)</td>
<td>RdLo, RdHi, Rm, Rs</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td>Accumulate Long</td>
<td></td>
<td></td>
<td>RdLo := unsigned((Rm × Rs) + RdHi,RdLo)[31:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RdHi := unsigned((Rm × Rs) + RdHi,RdLo)[63:32]</td>
</tr>
<tr>
<td>Signed Multiply</td>
<td>SMULL(S)</td>
<td>RdLo, RdHi, Rm, Rs</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td>Long</td>
<td></td>
<td></td>
<td>RdLo := signed(Rm × Rs)[31:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RdHi := signed(Rm × Rs)[63:32]</td>
</tr>
<tr>
<td>Unsigned Multiply</td>
<td>UMULL(S)</td>
<td>RdLo, RdHi, Rm, Rs</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td>Long</td>
<td></td>
<td></td>
<td>RdLo := unsigned(Rm × Rs)[31:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RdHi := unsigned(Rm × Rs)[63:32]</td>
</tr>
</tbody>
</table>
### Table 4.5: ARMv4 Logic and Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Clear</td>
<td>BIC[S]</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn AND NOT Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Bit AND</td>
<td>AND[S]</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn AND Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Bit Exclusive OR</td>
<td>EOR[S]</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn EOR Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Bit OR</td>
<td>ORR[S]</td>
<td>Rd, Rn, &lt;Operand2&gt;</td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := Rn OR Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td>Bit NOT</td>
<td>MVN[S]</td>
<td>Rd, &lt;Operand2&gt;</td>
<td>Move Negative (Bitwise 1's Complement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S: Flags Updated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd := NOT Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressing Mode 1</td>
</tr>
</tbody>
</table>

### Table 4.6: ARMv4 Comparison Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Compare</td>
<td>CMP</td>
<td>Rn, &lt;Operand2&gt;</td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Updates flags in CPSR for Rn - Operand2</td>
</tr>
<tr>
<td>Arithmetic Compare Negative</td>
<td>CMN</td>
<td>Rn, &lt;Operand2&gt;</td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Updates flags in CPSR for Rn + Operand2</td>
</tr>
<tr>
<td>Test Equivalence</td>
<td>TEQ</td>
<td>Rn, &lt;Operand2&gt;</td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Updates flags in CPSR for Rn EOR Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Does not affect the V flag</td>
</tr>
<tr>
<td>Bit Test</td>
<td>TST</td>
<td>Rn, &lt;Operand2&gt;</td>
<td>Addressing Mode 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Updates flags in CPSR for Rn AND Operand2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Does not affect the V flag</td>
</tr>
</tbody>
</table>

### Table 4.7: ARMv4 Flow Control and Exception Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>B</td>
<td>&lt;label&gt;</td>
<td>PC := label</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Label must be ±32MB</td>
</tr>
<tr>
<td>Branch &amp; Link</td>
<td>BL</td>
<td>&lt;label&gt;</td>
<td>PC := label; LR := PC - 4 (ARM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Label must be ±32MB</td>
</tr>
<tr>
<td>Branch &amp; Exchange</td>
<td>BX</td>
<td>Rn</td>
<td>Switch between ARM and THUMB modes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PC := Rn; Rm[0] = {1: THUMB, 0: ARM}</td>
</tr>
<tr>
<td>Supervisor (Software Interrupt)</td>
<td>SVC / SWI</td>
<td>&lt;imm24&gt;</td>
<td>SVC is the new UAL mnemonic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SWI Deprecated</td>
</tr>
</tbody>
</table>
Table 4.8: ARMv4 Coprocessor Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessor Data Processing</td>
<td>CDP</td>
<td>&lt;copr&gt;, &lt;op1&gt;, CRd, CRn, CRm, &lt;op2&gt;</td>
<td>Addressing Mode 5</td>
<td></td>
</tr>
<tr>
<td>Load Coprocessor from Memory</td>
<td>LDC</td>
<td>&lt;copr&gt;, CRd, &lt;operand3&gt;</td>
<td>Addressing Mode 5</td>
<td></td>
</tr>
<tr>
<td>Store Coprocessor to Memory</td>
<td>STC</td>
<td>&lt;copr&gt;, CRd, &lt;operand3&gt;</td>
<td>Addressing Mode 5</td>
<td></td>
</tr>
<tr>
<td>Move to Coprocessor from ARM Register</td>
<td>MCR</td>
<td>&lt;copr&gt;, &lt;op1&gt;, Rd, CRn, CRm, &lt;op2&gt;</td>
<td>Addressing Mode 5</td>
<td></td>
</tr>
<tr>
<td>Move to ARM Register from Coprocessor</td>
<td>MRC</td>
<td>&lt;copr&gt;, &lt;op1&gt;, Rd, CRn, CRm, &lt;op2&gt;</td>
<td>Addressing Mode 5</td>
<td></td>
</tr>
</tbody>
</table>

4.2.4 Addressing Mode 3 (Halfword/Signed Byte) Syntax

Addressing Mode 3 is used for Indirect Addressing to Load and Store half-words and bytes from/to memory. It is similar to Addressing Mode 2 except that the range for the immediate offset is less, and there is no scaled register offset mode. The memory address is determined as follows:

- Immediate Offset ([Rn, #+-<imm8m>])
- Register Offset ([Rn, +/-Rm])
- Immediate Pre Indexed ([Rn, #+-<imm8m>]!)
- Register Pre-Indexed ([Rn, +/-Rm]!)
- Immediate Post-Indexed ([Rn], #+-<imm8m>)
- Register Post-Indexed ([Rn], +/- Rm)

4.2.5 Addressing Mode 4 (LDM/STM) Syntax

4.2.5.1 LDM/STM Instruction Addressing Submodes

Addressing Mode 4 is used to Load and Store multiple registers from/to memory in a single instruction. There are four submodes:

- Increment After (IA) (Rn[]), <registers>[^]
- Increment Before (IB) (Rn[]), <registers>[^]
- Decrement After (DA) (Rn[]), <registers>[^]
- Decrement Before (DB) (Rn[]), <registers>[^]

The register Rn is the Base Register which provides the starting address for all subsequent memory access. The list of registers (R0-R15) will be stored in consecutive memory locations where the lowest numbered register is stored at the lowest memory address, and vice-versa, regardless of the addressing mode (direction of base register movement).
Table 4.9: Stack Operation Equivalent Mnemonics

<table>
<thead>
<tr>
<th>Stack Type</th>
<th>Load Instruction (Equiv.)</th>
<th>Load Instruction (Actual)</th>
<th>Store Instruction (Equiv.)</th>
<th>Store Instruction (Actual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Descending (FD)</td>
<td>LDMFD</td>
<td>LDMIA</td>
<td>STMFD</td>
<td>STMDB</td>
</tr>
<tr>
<td>Full Ascending (FA)</td>
<td>LDMFA</td>
<td>LDMDA</td>
<td>STMFA</td>
<td>STMIB</td>
</tr>
<tr>
<td>Empty Descending (ED)</td>
<td>LDMED</td>
<td>LDMIB</td>
<td>STMED</td>
<td>STMDA</td>
</tr>
<tr>
<td>Empty Ascending (EA)</td>
<td>LDMDA</td>
<td>LDMDB</td>
<td>STMEA</td>
<td>STMDA</td>
</tr>
</tbody>
</table>

If "!' is specified, the Base Register will be updated after the instruction has been executed.
If "!' is specified, it means that when the CPU is in a privileged mode, user mode banked registers would be transferred instead of the current privileged mode registers (this affects R13 and R14). However, if the instruction is a LDM instruction that involves R15 (PC), then "!' means that CPSR would be loaded from the current mode's SPSR.

4.2.5.2 Equivalent Syntax for Stack Manipulation

Since stack manipulation maps naturally to the LDM/STM instructions, and given that stacks can be implemented in one of four ways: a combination of Full or Empty with Ascending or Descending, the ARM syntax has been given alternative names to make it easy to refer to the correct stack operation (Table 4.9). Note: In AAPCS [21], a Full Descending stack, aligned to an 8-byte boundary has been specified.

4.2.6 Addressing Mode 5 (Co-Processor)

Addressing Mode 5 is used for co-processor interfacing. However, it will not be covered in this book since the NXT does not utilize these instructions. Interested readers should refer to [19, 11] for more information.

4.2.7 Pseudo Instructions

The ARM Unified Assembly Language (UAL) specifies the use of Bit Shift instructions which were previously specified as the Shifter Operand (Source Operand 2) for the MOV instructions. Basically the UAL mnemonic replaces the shifter operand format for easier comprehension. The reason these instructions are called pseudo instructions is because they are converted by the Assembler into the most appropriate commands for code generation, depending on the requirement.

4.2.8 Instruction Suffixes

The ARM Instruction Set supports additional optional Suffixes that control the conditional execution of Instructions based on the values of the Status Flags, as well as the updating of the Status Flags (specified using the 'S' suffix) after the execution of instruction that affect the flags. The general
### Table 4.10: ARM Pseudo Instructions

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Opcode</th>
<th>Operands</th>
<th>Equivalent Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Constants</td>
<td>ADR</td>
<td>Rd, &lt;label&gt;</td>
<td></td>
<td>Form PC relative Address using one ADD or SUB, error if not within range.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Literal Pool not used</td>
</tr>
<tr>
<td></td>
<td>ADRL</td>
<td>Rd, &lt;label&gt;</td>
<td></td>
<td>Form PC relative Address using two ADD or SUB, error if not within range.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOP generated if second instruction is not needed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Literal Pool not used</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td>Rd, ={label}</td>
<td>MOV or MVN used if possible.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Otherwise constant for PC relative LDR instr. stored in Literal Pool</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP</td>
<td></td>
<td>MOV R0, R0</td>
<td>Expend one execution cycle</td>
</tr>
<tr>
<td>Bit Shift</td>
<td>ASR</td>
<td>Rd, Rm, &lt;Rs</td>
<td>sh&gt;</td>
<td>MOV Rd, Rm, ASR &lt;Rs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Arithmetic Shift Right: &lt;Rs</td>
</tr>
<tr>
<td></td>
<td>LSL</td>
<td>Rd, Rm, &lt;Rs</td>
<td>sh&gt;</td>
<td>MOV Rd, Rm, LSL &lt;Rs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logical Shift Left: &lt;Rs</td>
</tr>
<tr>
<td></td>
<td>LSR</td>
<td>Rd, Rm, &lt;Rs</td>
<td>sh&gt;</td>
<td>MOV Rd, Rm, LSR &lt;Rs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logical Shift Right: &lt;Rs</td>
</tr>
<tr>
<td></td>
<td>ROR</td>
<td>Rd, Rm, &lt;Rs</td>
<td>sh&gt;</td>
<td>MOV Rd, Rm, ROR &lt;Rs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rotate Right: &lt;Rs</td>
</tr>
<tr>
<td></td>
<td>RRX</td>
<td>Rd, Rm</td>
<td>MOV Rd, Rm, RRX</td>
<td>Rotate Right with Extend Register only</td>
</tr>
<tr>
<td>Stack</td>
<td>POP</td>
<td>{reglist}</td>
<td>LDMIA SP!, {reglist}</td>
<td>Full Descending (FD) Stack</td>
</tr>
<tr>
<td></td>
<td>PUSH</td>
<td>{reglist}</td>
<td>STMDB SP!, {reglist}</td>
<td>Full Descending (FD) Stack</td>
</tr>
</tbody>
</table>
syntax for conditional execution and status flag updating (where applicable) is as follows:

\[ \text{opcode\{<cond_exec_suffix>\}\{S\}} \]

However, Load and Store instructions do not affect the Status Flags, and hence do not use the ‘S’ suffix. Nonetheless, the Load and Store instructions have byte, halfword, and signed/unsigned variants which are specified after the conditional execution suffix. The syntax for the Load and Store instructions are as follows:

\[ \text{LDR\{<cond_exec_suffix>\}\{B|SB|H|SH\}} \]
\[ \text{STR\{<cond_exec_suffix>\}\{B|SB|H|SH\}} \]

### 4.2.8.1 Condition Flags and Status Flag Updating

There are four condition flags found in the Status Register

- Negative (N)
- Zero (Z)
- Carry (C)
- Overflow (V)

These four flags are affected by arithmetic and register-to-register copy operations, if the Status Flag Update (‘S’) is specified for the given instruction. Otherwise the flags are not affected by the instruction execution. Instructions that support the Status Flag Update are listed in the Opcode table with the \{S\} suffix.

### 4.2.8.2 Conditional Execution

Normally, branch instructions specifies the usage of conditional execution in order to implement various flow control and expression evaluation logic. However, conditional execution of instructions is not limited only to branch instructions for the ARM processor architecture. Instead, all ARM instructions can potentially be executed conditionally. Instead of relying only on flow control instructions to determine whether a block of code should be executed or otherwise, the use of Conditional Execution Suffixes enables execution flow control on a per-instruction basis. If the condition specified were not met, the instruction becomes a NOP. The use of conditional execution instructions results in a fixed number of execution cycles for such a block of code regardless of the logic path taken. For most instructions, the conditional execution suffixes are placed just before the ‘S’ suffix (if present).

Table 4.11 lists the various conditional tests and the associated boolean logic equation used to check for the given condition\(^6\). It should be noted that the comparison conditions and suffixes are different for signed and unsigned comparisons. Examples of the conditional execution format for some instructions are given below:

\(^6\)Carry flag = inverted Borrow flag for Subtraction and Comparison Instructions.
Table 4.11: Signed and Unsigned Conditional Execution Suffixes

<table>
<thead>
<tr>
<th>Unsigned Condition</th>
<th>Suffix</th>
<th>Boolean Equation</th>
<th>Mathematical Equivalent</th>
<th>Signed Condition</th>
<th>Suffix</th>
<th>Boolean Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>EQ</td>
<td>Z</td>
<td>( A = B )</td>
<td>Equal</td>
<td>EQ</td>
<td>Z</td>
</tr>
<tr>
<td>Not Equal</td>
<td>NE</td>
<td>( \neg Z )</td>
<td>( A \neq B )</td>
<td>Not Equal</td>
<td>NE</td>
<td>( \neg Z )</td>
</tr>
<tr>
<td>Higher</td>
<td>HI</td>
<td>( C \cdot Z )</td>
<td>( A &gt; B )</td>
<td>Greater Than</td>
<td>GT</td>
<td>( Z \cdot (N = V) )</td>
</tr>
<tr>
<td>Higher or Same</td>
<td>HS/CS</td>
<td>C</td>
<td>( A \geq B )</td>
<td>Greater Than or Equal</td>
<td>GE</td>
<td>( N = V )</td>
</tr>
<tr>
<td>Lower</td>
<td>LO/CC</td>
<td>( \neg C )</td>
<td>( A &lt; B )</td>
<td>Less Than</td>
<td>LT</td>
<td>( N \neq V )</td>
</tr>
<tr>
<td>Lower or Same</td>
<td>LS</td>
<td>( \neg C \cdot Z )</td>
<td>( A \leq B )</td>
<td>Less Than or Equal</td>
<td>LE</td>
<td>( Z + (N \neq V) )</td>
</tr>
<tr>
<td>Always</td>
<td>AL</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

MOVGE : Copy to specified register if greater or equal (signed)
ADDEQUS : Add values in registers if equal, and update status flag
LDRLE : Load word from memory into register if less than (signed)
LDRNEB : Load byte from memory into register if not equal
STRLSH : Store half-word into memory if lower or same (unsigned)
STRGTSB : Store signed byte into memory if greater than (signed)

4.2.9 ARM Instruction Encoding

The ARM Instruction Encoding format is given in Figure 4.2 for reference. The conditional execution control bits are defined as the Most Significant Nibble of the instruction word, while various instruction prefixes are used to group similar instructions together. While it is not critical for Assembly Language programmers to know the encoding format of a given instruction, it is a useful debugging skill to be able to examining object file dumps to determine if a given instruction in the object code was assembled correctly from the specified source line, or when trying to catch logic errors in instruction operand specification which assembles correctly but performs an unintended action.

4.2.10 Using ARM Assembly

Listing 4.2 is a short example illustrating how the ARM opcodes can be used to implement an algorithm to add two 32-bit constant values, 313A₁₆ and A0BB₁₆, and store the result in external memory in a variable called SUM which at address 20103C₁₆. The source code, when assembled, will generate an object code file. The object code file can be disassembled using the objdump application, to give the output shown in Listing 4.3. It should also be noted that the LDR pseudo-instruction used PC-relative Addressing mode to access constant values stored in memory locations immediately following the block of code.

In the Disassembly output, comments are prefixed by semicolon (‘;’) which is not the symbol accepted by GAs.
<table>
<thead>
<tr>
<th>Data processing and FSR transfer</th>
<th>Cond</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>Cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>S</td>
<td>Rd</td>
</tr>
<tr>
<td>Multiply long</td>
<td>Cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>A</td>
<td>S</td>
<td>RdHi</td>
</tr>
<tr>
<td>Single data swap</td>
<td>Cond</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
</tr>
<tr>
<td>Branch and exchange</td>
<td>Cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Halfword data transfer, register offset</td>
<td>Cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>O</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Halfword data transfer, immediate offset</td>
<td>Cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>1</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Single data transfer</td>
<td>Cond</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>P</td>
<td>U</td>
<td>O</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Undefined</td>
<td>Cond</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Block data transfer</td>
<td>Cond</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Branch</td>
<td>Cond</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Coprocessor data transfer</td>
<td>Cond</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>N</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Coprocessor data operation</td>
<td>Cond</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CP</td>
<td>Opc</td>
<td>CRd</td>
<td>CP#</td>
<td>CP</td>
</tr>
<tr>
<td>Coprocessor register transfer</td>
<td>Cond</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CP</td>
<td>Opc</td>
<td>L</td>
<td>CRn</td>
<td>Rd</td>
</tr>
<tr>
<td>Software interrupt</td>
<td>Cond</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.2: 32-bit ARM Instruction Encoding, courtesy of ARM Ltd. [7]
Listing 4.2: ARM Assembly to add two constant values (Alg-4.2.S)

.equ SUM, 0x0020103C
START: LDR R0, =0x313A @ Load 313Ah
LDR R1, =0xA0BB @ Load A0BBh
ADD R0, R1, R0 @ Perform Addition
LDR R2, =SUM @ Init R2 as Addr. Ptr.
STR R0, [R2] @ Store to addr. 20103Ch
STOP: B STOP @ Loop Forever

Listing 4.3: Object Code Output for Alg-4.2.S

Alg-4.2.o: file format elf32-littlearm

Disassembly of section .text:

00000000 <START>:
 0: e59f0010 ldr r0, [pc, #16] ; 18 <STOP+0x4>
 4: e59f1010 ldr r1, [pc, #16] ; 1c <STOP+0x8>
 8: e0810000 add r0, r1, r0
 c: e59f200c ldr r2, [pc, #12] ; 20 <STOP+0xc>
10: e5820000 str r0, [r2]

00000014 <STOP>:
 14: eaafffffe b 14 <STOP>
 18: 0000313a .word 0x0000313a
 1c: 0000a0bb .word 0x0000a0bb
 20: 0020103c .word 0x0020103c

4.2.11 Accessing Variables

Unlike High Level Languages, variables (stored in external memory) used in Assembly Language programming must be managed by the programmer. Assignment of a value to a variable (e.g., the result of an arithmetic operation such as SUM in Listing 4.2) means that the value in a register must be stored to the proper memory address location (20103Ch). Temporary variables such as the intermediate results of arithmetic operations can be stored in Registers if there are only a few of them. R0 and R1 are temporary variables in Listing 4.2. These variables are known as register variables and do not have any significance outside the processor. Temporary registers can be used for other calculations once the algorithm is completed. It is the responsibility of the programmer to ensure that register variables are preserved if they are needed by other parts of the program. For example, if R13 were used as a stack pointer, then the value in R13 should not be modified in any way by other parts of the program unless it has to do with stack manipulation.
4.3 Instruction Synthesis

For RISC architectures, there is often a need for specific instructions that may be available as a standard part of the CISC instruction set, but not implemented in the RISC architecture. This is addressed by means of Instruction Synthesis, typically through the use of Macro definitions. Macros define a sequence of instructions that implement a frequently used function which can be used like other built-in instructions. For example, there is no No Operation (NOP) instruction in the ARMv4T instruction set. However, NOPs are useful in situations where the processor state requires time to stabilize, for example, when modifying the CPSR or loading or storing multiple registers involving the Stack Pointer (R13). Consequently, NOP is assembled as MOV R0, R0 by the GNU Assembler. In this way, a given RISC instruction set can be expanded to provide needed functionality.

4.3.1 Macros and Instruction Synthesis

Macros are Assembler directives used to define and access a sequence of frequently used instructions. Macros can be used as building-blocks to simplify assembly language programming, to make writing and understanding a given algorithm much easier. Instead of constantly writing a sequence of instructions to Clear all Contents of Registers, we can define a Macro called CLEARREG that contain instructions to perform that task. We use the macro by using its name as the opcode in an instruction, in a similar way to using built-in instruction opcodes. By using macros, we eliminate typing and logical mistakes that may arise if we entered the same set of statements each time. The use of macros also helps in our understanding of the algorithm in question. The purpose of a sequence of instructions is captured in the name of a macro. This allows the programmer to keep track of the logical flow in the algorithm instead of trying to determine which instructions each logical step in the flowchart maps to.

Macros map well to the need for Instruction Synthesis in RISC architectures. As described previously, we needed instructions such as NOP, COM (2’s Complement) and INC (increment) that are used frequently. By defining these synthesized instructions as macros, we can keep them in header files that are included into any programs that we write. This leads to code reuse and help in creating a standard library of code fragments that allow the program to tackle more advanced programming tasks.

4.3.2 GNU Assembler Macro Format

Since Macros are functionality provided by the Assembler, the way in which macros are defined and used is heavily dependent on the capabilities of the given Assembler, and may vary from assembler to assembler. At minimum, using macros cause the same sequence of instruction statements to be automatically inserted into the source code when it is assembled. The GNU Assembler defines macros according to the following syntax:

---

*A NOP instruction was finally added to the ARMv6K architecture.*
The arguments for a given macro are optional, and can have as many arguments as necessary. Arguments are defined as variable names, and referenced in the body of the macro using the \( \backslash \) prefix. Since the GNU Assembler basically performs string substitutions for each argument in the body of the macro, arguments can be used to specify instructions, operands, labels, and variable names. For example, the COM macro can be defined as follows:

```
.macro COM dest, src
   MVN \dest, \src @ dest := NOT src (1’s compl)
   ADD \dest, \dest, #1 @ dest := dest + 1
.endm
```

When invoking the macro, it is written in the source file as if the macro is a normal instruction, and the arguments supplied to the macro are actual register or variable references as necessary. For example, to store \(-100\) in 2’s Complement format in R1, we can do the following:

```
... MOV R0, #100  @ load constant value +100
   COM R1, R0  @ convert to 2’s complement format (-100)
...```

This will then be expanded by the GNU Assembler into:

```
... MOV R0, #100  @ Original Source Line
   MVN R1, R0  @ Expanded from COM Macro
   ADD R1, R1, #1 @ Expanded from COM Macro
...```

Consequently, each time a macro is invoked, it increases the size of the object file by the number of instructions contained within the macro. Lengthy macros may not be a good idea if it were to be invoked frequently in a program. For example, to calculate the square root of a number involves the use of many instructions. In such cases, Routines (Chapter 5) should be used to implement the functionality instead.

In addition to basic instruction insertion into our code, we can also create macros that perform conditional substitutions and variable-length expansions. These features are documented in Appendix B.5.4.

### 4.3.3 Advantages and Disadvantages of Macros

By using previously defined macros, we can ensure that:

- The readability and semantics (meaning) of the algorithm is easily understood (e.g., ‘COM R0,R2’ instead of ‘MVN R0,R2; ADD R0,R0,#1’).
• Implementation of a well-defined operation is not accidentally modified due to mistyping or other logical errors.

• Underlying macro implementation may be changed depending on requirements. For example, to enable error checking in a debugging version, and without any error checking in the release version.

• Execution speed is much faster compared with routine calls since parameters are accessed directly when the macro is created, without the need to pass parameters to the routine.

However, the use of macros also have disadvantages:

• Macros may increase the size of the application significantly compared with routines. Each use of a macro results in the Assembler inserting the same sequence of instructions into the object code, while a routine call uses just one instruction (Call routine) if no parameters are passed via external memory addresses.

• The use of different macro implementations for debugging vs. release versions of the program results in very different code sizes. The debugging version may exceed the maximum branch offset distance for flow control instructions due to the addition of extra instructions.

4.4 Your First Assembly Language Program

Writing Assembly Language programs for the NXT can be done at many levels. The lowest level involves bare-metal programming which requires the programmer to write code which operates from the moment the ARM processor comes out of Reset mode, to initialize all the hardware peripheral modules found in the Atmel AT91SAM7S256 microcontroller, and finally to execute the instruction sequences that carry out tasks that the robot is supposed to perform. Since this is rather daunting for someone who’s getting started in Assembly Language programming, we’ll write the first Assembly Language program for an Operating System which performs all the basic hardware initialization tasks for us instead. The NxOS operating system [28] is a ground-up implementation of an open source operating system for the MINDSTORMS NXT, providing a comprehensive base system for controlling the NXT hardware, sensors, display and motors. Although the bulk of NxOS has been developed in C, the ARM AEBI standard allows for easy linking of existing C code and code written in ARM Assembly Language.

Since developing programs in Assembly Language can sound complex at first, good Debugger support is important. NxOS includes the Armdebug module, which provides debugging via the GNU Debugger (GDB) Remote Serial Protocol, to allow us to connect remotely from the PC to the NXT for examining values of registers and memory locations, as well as step through instructions as they are executed on the microcontroller. Further details of how to setup the Debugger link between the PC and the NXT is found in Appendix B.3.
4.4.1 Building the First Program

To build our first program, we will start with the skeleton program found in nxos/systems/armskel subdirectory. First, copy the armskel subdirectory to a new subdirectory under nxos/systems. For example, the new subdirectory would be named first, which is also the name of the NXT executable. The SConscript file in the first subdirectory is used by the scons build system to create the first.rxe NXT executable file. The contents of SConscript (example shown in Listing 4.4) needs to be updated by changing the ‘armskel’ parameter to the name of the new executable (in our case, ‘first’). This enables the build system to recognize the name of the executable to build during the build process. Once you have updated the SConscript file with the correct subdirectory name, you can proceed to build the first.rxe executable. The build process is performed using scons, and initiated from inside the nxos directory under the top level nxos-armdebug project directory.

$ cd ~/path/to/nxos-armdebug/nxos; scons appkernels=first

Listing 4.4: SConscript file for armskel program

from glob import glob
Import('env')
env.AppKernel('armskel', glob('*.cS'), kernelsize='50k')

4.4.2 Components of armskel

The armskel subdirectory consists of the following files:

- SConscript: used for building the executable files, including the RXE format executable <subdirname>.rxe, and the firmware-based <subdirname>_*.bin
- armskel.S: Assembly source file containing the main routine

The names of the individual files within the subdirectory is not critical, as long as the Assembly source files have the .S suffix, C source files have the .c suffix, and Header files have the .h suffix, scons would be able to compile or assemble them into the correct object files. These object files would then be linked with the Nxos base modules to generate the executable for the NXT. It should be noted that the executable is built from both C-based as well as Assembly Language based modules. Nxos base directory contains drivers and supporting libraries written in C. Our focus is on the Assembly language source file, armskel.S. The interfacing of C-based modules with Assembly language modules is covered in greater detail in Chapter 6.5.

Three versions of the armskel executable are created:

- <subdirname>.rxe, meant to be downloaded as a NXT Program File in Flash, and executed from RAM. The RXE file is a Native ARM executable in the NXTBINARY format, recognized by both the NXT Enhanced Firmware by John Hansen (29) and NXT Improved Firmware (NIF) (30).

*The standard MINDSTORMS NXT Firmware from LEGO needs to be replaced with either the NXT Enhanced Firmware or NIF for it to recognize the NXTBINARY RXE file format.
• `<subdirname>_rom.bin`, meant to be flashed to the NXT and executed from Flash. WARNING: This effectively replaces the built-in firmware with the executable.

• `<subdirname>_samba.bin`, meant to be downloaded to RAM in the NXT and executed from RAM. WARNING: This effectively erases the built-in firmware, and upon reboot, the NXT cannot execute any program without first flashing new firmware or downloading a new program to RAM using SAMBA mode.

Since the ROM and SAMBA versions of the executable erases and/or overwrites the entire Flash contents, it would not be very easy or convenient to use these formats for iterative development and debugging. On the other hand, the NXTBINARY RXE files can be managed using John Hansen’s NeXT Tool ([31]) in the same way as other NXT-G or NXC executables. In addition, it does not affect the NXT Firmware in Flash, so existing NXT-G or NXT applications can co-exist with NxOS-based native ARM executables. Consequently, the NXTBINARY RXE executable format is suitable for iterative program development and debugging.

### 4.4.3 Debugging the First Program

When the build process is completed without any errors, the executable file will be found inside `~/path/to/nxos-armdebug/nxos/systems/first`. The executable file, `first.rxe` is then downloaded to the NXT using a suitable Downloading Tool such as NeXT Tool via a USB connection. To run the program, the NXT Firmware program execution menu is used to invoke the `first.rxe` program. At that point in time, the execution will encounter a manually inserted (hardcoded) Breakpoint in the executable code, which causes the NXT to enter the Debugger and wait for remote GDB commands to be issued by the Host to inspect variables and control execution of the program instructions. The manual breakpoint is important since it ensures that the program execution does not continue before we are ready to issue the relevant debugging commands via GDB.

#### 4.4.3.1 Console based Debugging

Invoking the debugger involves first starting the GDB server process in a terminal window.

```
$ cd ~/path/to/nxos-armdebug/scripts/armnxtgdbserver
$ ./armnxtgdbserver
```

The `armnxtgdbserver` script checks the environment, then invokes the GDB server in `~/nxos/armdebug/Host/nxt-gdb-server.py`. For Mac OS X, the command prefix:

```
$ cd ~/path/to/nxos-armdebug/scripts/armnxtgdbserver
```

While Flash has a finite lifetime, typically over 100,000 erase/write cycles, it is not likely for us to cause the Flash to fail due to repeated erase and download cycles in the course of developing a program.
export VERSIONER_PYTHON_PREFER_32_BIT=yes; arch -i386

is used to invoke the GDB server since the Mac OS X Fantom drivers are 32-bit only. The variable export is the Mac OS X preferred way of specifying 32-bit python invocation. This command prefix will work for both the default Apple supplied Python as well as Python installed via MacPorts. Note that the configuration of the Mac OS X python version is already included as part of the armnxtgdbserver script.

This GDB server process is used to communicate with the GDB stub built into the NXT executable file. It will start, then prompt for an <ENTER> press after the NXT program has started execution, and then wait for a GDB client connection on Port 2828. If you’re using the terminal-based GDB client, a new terminal window should be started, and the GDB client invoked as follows:

```bash
$ cd ~/path/to/nxos-armdebug/nxos/systems/first
$ arm-none-eabi-gdb first_rxe.elf
```

When the GDB client is invoked, it does not know that the executable is actually running on the NXT. To tell the GDB client to connect to the GDB server on port 2828, the `target remote` command should be used.

```gdb
(gdb) target remote localhost:2828
```

If the NXT program Debugger stub is working correctly, it will exchange a series of messages between the GDB client and the NXT, after which the gdb prompt returns and we are now ready to interact with the NXT.

```gdb
(gdb) info registers
```

An example of the output of `info registers` is given in Listing 4.5.

### Listing 4.5: Sample Output of `info registers`

```
r0 0x0 0
r1 0x201a21 2103841
r2 0x0 0
r3 0x37 55
r4 0x20fefe 2162428
r5 0x20a9f5 2140661
r6 0x20aa39 2140729
r7 0x11 17
r8 0x0 0
r9 0x0 0
r10 0x0 0
r11 0x40000 262144
r12 0xffffb0054 4294967296
sp 0x20fefe 0x20fefe
lr 0x10179b 1054619
pc 0x1017a2 0x1017a2
fps 0x0 0
cpsr 0x80000033 2147483699
```
Other commands that can be used in the GDB client to disassemble the object code, specify breakpoints, and step through the CPU instructions can be found in Appendix B.3.5.

### 4.4.3.2 Eclipse CDT Debugging

First, switch Perspectives to the Debug Perspective. Invoke the GDB server process via the External Tools Menu (see Appendix B.3.3.1). You can create a new Console instance to view the output of the armmxtgdbserver process. Then, the Debugger is started using the Run->Debug Configurations... menu (see Appendix B.3.3.2), which will start the Graphical GDB front end, and prompt for your next action.

If you're using the Graphical GDB front end from inside Eclipse, the break: label provides the Eclipse GDB front end a breakpoint location which it can use to pause the execution of the program. Otherwise when the Eclipse GDB front end first attaches to the remote GDB process, it issues a 'Continue' command which causes the execution to proceed with the rest of the program after the manual breakpoint.

### 4.4.4 Modifying the First Program

The original code in armskel.S (Listing 4.6) just generates a startup tone sequence, initializes the LCD Display, display a title string, and then triggers the hardcoded (manual) Breakpoint.

Listing 4.6: armskel.S

```c
/** @file armskel.S
 * @brief ARM Skeleton Program
 * [What does this component do?]
 * *
 * [Optionally, more stuff about the component in general.]
 * This file must be processed using cpp before assembly.
 */

/* Copyright (C) 2007-2011 the NxOS developers
 * See AUTHORS for a full list of the developers.
 * Redistribution of this file is permitted under
 * the terms of the GNU Public License (GPL) version 2.
 */
#define __ASSEMBLY__
#include "base/interwork.h"
#include "base/lib/scaffolding/scaffolding.h"
#include "armdebug/Debugger/debug_stub.h"

/* To disable Manual Breakpoints, change the following to #undef ENABLE_BREAKPOINTS */
#define ENABLE_BREAKPOINTS
extern nx_systick_wait_ms
.equ SYSTICK_1000MS, 1000
.equ SYSTICK_500MS, 500
.equ LOOPCNT, 3
.data
.align 4
title: .asciz "Armskel Prog"
prompt: .asciz "Welcome to NxOS"
```
As the first modification, uncomment the subtraction and the branch statement just before the label exit_main:

```
subs r7, r7, #1
bne loop
```

This will cause the execution of the program to repeat the display of the messages on the LCD screen three times before exiting. Build the modified program using the steps outlined in Section 4.4.3, then debug the program in GDB. Note the initial value of register R7 when the program encounters the manual breakpoint when you first start the debugger. Then, Continue execution of the program, and hit <CTRL-C> after the display has cleared and displayed the title string. You should observe that the value of R7 has
changed. You can also step through the instructions one by one, though it will be very tedious to step through the `nx_systick_wait_ms()` routine since it is a loop which only exits when the timer interrupt (systick) is updated.

### 4.5 Chapter Summary

- The different notations (Assembly Language Syntax and Register Transfer Notation) for representing processor execution behavior were presented.

- Program code generation is a process of converting abstract logic into concrete machine language instructions to be executed by the processor.

- Instruction Synthesis is a powerful methodology that is used in RISC architectures to implement less commonly used features.

### 4.6 Review Questions and Problems

1. Virtual Machines are software applications that emulate the instruction set of another microprocessor using software routines. One popular example is the Java Virtual Machines (JVM) for executing Java byte-codes on different processors. Discuss the advantages and disadvantages of virtual machines in software development and application execution.

2. CISC instruction sets often have more instructions for different functions and of different lengths (in bytes). Determine whether the following statements are TRUE or FALSE, and give your reasoning:
   
   (a) Fewer number of instructions are needed to implement a given algorithm using a CISC instruction set.

   (b) The variable length instructions in CISC architectures are easier to decode compared with the fixed length instructions in RISC.

   (c) The larger number of registers available to programmers in RISC instruction sets leads to better algorithm optimization opportunities.

   (d) The lack of string manipulation instructions in the ARM instruction set means that it cannot execute text editing software.

   (e) CISC instructions usually take multiple cycles to execute compared with RISC instructions which are often single-cycle instructions, therefore CISC processors are always slower than RISC processors.
Chapter 5

The First Step

‘Can you do Addition?’ the White Queen asked. ‘What is one and one and one and one and one and one and one and one and one and one?’
‘I don’t know,’ said Alice, ‘I lost count.’

from “Through the Looking Glass,” Lewis Carroll, 1832-1898

We want our MINDSTORMS robot to perform useful tasks. All that depends on the software program running on the NXT brick, in order to process inputs from sensors, act on the inputs by either displaying some information on the NXT LCD screen, or activate output devices such as LEDs, motors, and play sounds through the speaker. The software program, can be divided into fundamental pieces such as frequently used algorithms to implement a particular action that are written by us as routines and functions, and existing routines and functions provided by the firmware or operating system, typically used to control inputs and outputs of the NXT.

Furthermore, each routine or function consists of a sequence of steps, or instructions to be executed by the ARM processor. These instructions can be classified as data movement, arithmetic, logic and flow control instructions, and are fundamental to the creation of routines and programs, by implementing the logic contained in flowcharts or pseudocode description of the program function.

5.1 Data Movement Instructions

Data Movement Instructions were described in Tables 4.2 and 4.3 as well as Chapter 4.2.2. Basically data movement instructions are used to transfer contents of one register to another. Data movement instructions are also used to initialize the value of registers, retrieve data contents from external memory, and store results to external memory.
5.1.1 Register Access Instructions

Since data inside the processor is stored in general registers, it is important to have instructions which manipulate the contents of these registers. This includes initializing them to known (constant) values, as well as copying them from one register to another in the course of executing a given algorithm. The basic instruction for doing so is MOV (MOV), where Rd is the destination register that will be modified by the instruction.

\[
\text{MOV}\{S\} \text{ Rd, } \langle\text{Operand2}\rangle
\]

While the MVN (Move Negative) instruction has “Move” in its name, it is actually a logic instruction. MVN copies the inverse value (1’s complement or NOT) in the source operand into the destination register. Logic instructions are covered in Section 5.3.

\[
\text{MVN}\{S\} \text{ Rd, } \langle\text{Operand2}\rangle
\]

Most ARM instructions have an optional variant (S: Set Flags) which modifies the Condition Flags in the Program Status Register (PSR) depending on the value stored in Rd. For example if the value stored into Rd is zero, then the Z flag would be set, and so on.

\langle\text{Operand2}\rangle specifies the type of arguments that will be used to update the contents of destination register Rd, and is classified as Addressing Mode 1. For the ARM architecture, \langle\text{Operand2}\rangle is very flexible, as it can be used to specify 32-bit constants (though it can only be from a specific range of values), as well as copy the value from a source register that is optionally adjusted by shifting or rotating it. This is one of the unique features of the ARM architecture. The flexibility is very useful when the register is used to store an index for accessing array elements in memory, which we will look at when dealing with pointers.

Constant (Immediate) source values are encoded using 12-bits in the instruction field. This value consists of an 8-bit constant value and a 4-bit rotate value. The 4-bit rotate value is multiplied by two in order to obtain the rotate count. This means that only a subset of 32-bit integers can be loaded into a register using the MOV instruction. The value must have an even-rotate count contiguous bit-span that does not exceed 8 bits in length. The bit-span is defined as the number of bits between the leftmost ‘1’ bit and the rightmost ‘1’ bit. In addition, the rotate count must be even. If the register needs to be initialized with immediate values that do not meet this criteria, then the LDR instruction would be used instead.

For MOV involving registers, the source register value can be optionally bit-shifted using one of LSL, LSR, ASR, ROR or RRX modes before the resulting value is stored into the destination register. Instructions involving bit-shifted source registers can be written directly as a Bit-shift instruction instead of as a MOV instruction.

Special instructions are provided to manipulate the Program Status Registers (PSR), which refers to the CPSR and SPSR registers used in privileged modes. MRS copies the value from a PSR into a general register, while MSR copies the value from a general register or a constant value into a PSR. These two instructions can only be executed in one of the exception modes since the PSR is not accessible in user mode.
Table 5.1: Register Access Instructions using Addressing Mode 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, #0xFF</td>
<td>Load 255d into R0. Valid since bit-span of constant fits in 8-bits, with even-rotate count</td>
</tr>
<tr>
<td>MOV R7, #0x00800000</td>
<td>Load 8388608d into R7. Valid since bit-span of constant fits in 8-bits with even-rotate count</td>
</tr>
<tr>
<td>MOV R0, #0x0F800300</td>
<td>Can’t load 260047616d into R0. Not valid since bit-span of constant exceeds 8-bits</td>
</tr>
<tr>
<td>MOV R0, #0x1FE</td>
<td>Can’t load 510d into R0. Not valid, since rotate count is odd, although bit-span fits in 8-bits</td>
</tr>
<tr>
<td>MOV R3, R0, LSL #2</td>
<td>Copy value in R0 multiplied by 4 (LSL #2), to register R3; equivalent to LSL R3, R0, #2</td>
</tr>
<tr>
<td>MOV R9, R1, ROR R2</td>
<td>Copy value in R1, rotated right by valued in R2, into R9; equivalent to ROR R9, R1, R2</td>
</tr>
<tr>
<td>MOV R0, R2, RRX</td>
<td>Copy value in R2, rotated right by 1 into Carry, existing Carry is rotated into bit 31, into R0; equivalent to RRX R0, R2</td>
</tr>
</tbody>
</table>

```
MRS Rd, <PSR>
MSR <PSR>_<fields>, Rm
MSR <PSR>_<fields>, #<imm8m>
```

The <fields> suffix can be used to specify a subrange of bits in the PSR that will be modified. Valid values for #<imm8m> has to be used by referring to the section on CPSR bit definitions. This command allows the CPU to switch modes (from exception modes to user mode, or from one exception mode to another), as well as enable/disable Interrupt signals.

Examples of valid and invalid MOV instructions for each Addressing Mode 1 Category is given in Table 5.1. Listing 5.1 is provided to illustrate the use of the MOV instruction.

Listing 5.1: Usage of MOV instructions

```
/* Code to load values into various registers: 
 * R0: 0x30
 * R1: 0x180
 * R2: -1
 * R3: 65535 
 */
.global START
START:  MOV R0, #0x30      @ .... 0000 0011 0000 b
        MOV R1, R0, LSL #3  @ .... 0001 1000 0000 b
        MOVS R2, #-1        @ 'N' flag set (negative)
        MVN R3, #0          @ ls compl. is 0xFFFFFFF
        LSR R3, R3, #16     @ right shift, so R3 = 0x0000FFFF
STOP:    B STOP
```

5.1.2 Memory Access Instructions

Since ARM is a RISC-based architecture, it adopts the Load/Store paradigm for memory access. The memory map of the ARM microprocessor is used to
access both storage locations (RAM, ROM, Flash), as well as I/O peripheral control and status registers, according to the Memory-Mapped I/O architecture design. All memory access instructions use Addressing Mode 2 to specify the memory operands.

5.1.2.1 Single Address Location Access

The basic instruction is LDR (Load Register from Memory), and STR (Store Register to Memory).

\[
\begin{align*}
\text{LDR } & \text{ Rd, } <\text{addr_mode2}> \\
\text{STR } & \text{ Rd, } <\text{addr_mode2}>
\end{align*}
\]

Variants:

- LDRT
- LDRB, LDRBT, LDRSB
- LDRH, LDRSH
- STRT
- STRB, STRBT
- STRH

Variants of LDR and STR are used to access 16-bit values (H: Halfwords), 8-bit values (B: Bytes), Signed or Unsigned values (S: Signed) and also to emulate memory access in user mode (T: Translated) to memory when operating in an exception mode.

If we wish to load arbitrary 32-bit values into registers, we use an ‘=' symbol to prefix the constant value that we want to load. This causes the assembler to allocate data space in memory to store the arbitrary 32-bit value, and then reference the memory location to load the value into the register. It should be noted that using ‘=' to load constants creates a Literal Pool which is a block of memory within the code space used to store such constants, so each load instruction will use up 8 bytes (4 bytes for the actual instruction and 4 bytes for the constant value storage). We can control where in program memory the Assembler places the Literal Pool within a given assembly object file by using the ‘.ltorg’ assembler directive. By default, it is placed at the end of the last instruction statement in an object file.

Addressing Mode 2 is very flexible. Basically it is an indirect addressing mode which allows for Register only, Register with constant offsets, Register with variable offsets, as well as scaled variable offsets to be added to the base register value, to access elements within an array or structure easily. In addition, the base register value can be updated automatically either before (pre-indexed) or after (post-indexed) the memory location access, to enable quick iteration through consecutive items in an array. The base register is indicated using ‘[‘ and ‘]’ (which denotes memory access), and the pre-indexed pointer update is indicated using ‘!’, whereas the post-indexed pointer update is indicated by a third operand after the ‘[...’ address specification.

A special instruction SWP (Swap) can be used to implement atomic operations involving updating of a memory variable that cannot be interrupted.
by any other instruction. This is typically used to implement Mutual Exclusion (Mutex) and Semaphores.

\[
\text{SWP Rd, Rm, [Rn]}
\]
\[
\text{SWPB Rd, Rm, [Rn]}
\]

When swapping values between a CPU register and memory, the contents of Rm is saved to memory at address given by pointer Rn, while the contents of memory is retrieved to Rd. A variant of SWP allows the use of byte-sized (B: byte) variables.

### 5.1.2.2 Multiple Address Location Access

ARM provides a multi-register load store instruction which executes in multiple instruction cycles. LDM (Load Multiple) and STM (Store Multiple) can be used to load and store register banks quickly when entering and leaving subroutine calls. Technically they are not RISC instructions since they take a variable number of cycles to execute, depending on the number of registers specified in the register list, but they are very useful instructions since they simplify access to memory arrays and stacks.

\[
\text{LDM<mode> Rn(!), <reglist>{^}}
\]
\[
\text{STM<mode> Rn(!), <reglist>{^}}
\]

The <reglist> is the list of registers from R0-R15. Normally, R15 should not be included in the register list since the value stored into memory for R15 is not the address of the current instruction, as the ARM architecture has a pipeline which is executing the fetch of an instruction at a higher memory address at that particular moment (the actual value of R15 stored is dependent on the ARM architecture that the instruction is running on). The <mode> suffix is used to specify the direction and sequence for updating the base pointer Rn. Valid modes are Full Descending (FD), Full Ascending (FA), Empty Descending (ED), and Empty Ascending (EA). The LDM and STM instructions can be used to implement stack Push and Pop operations easily, with support for stacks which utilize descending or ascending memory addresses for storing subsequent items, as well as whether the base pointer refers to an occupied slot or an empty slot. The ARM AAPCS specifies the use of a full descending stack, so the <mode> is usually FD.

The base pointer register can be updated after the LDM or STM instruction completes, this is useful since a stack would require the stack pointer to be updated after each Push or Pop operation. If the ‘^’ suffix is added after the register list, then the STM instruction will store the user mode registers instead of the banked R13 and R14 registers if the instruction is executed in an exception mode. In contrast, the LDM instruction will restore the SPSR to the CPSR when the ‘^’ suffix is used. This can only be specified in an exception mode since it is a privileged operation. This function is useful when an exception handler wishes to return execution to a user program. Subroutine calls and returns will be covered in the chapter on Subroutines.

Some usage of memory access instructions is explained in Listing 5.2.
Listing 5.2: Usage of Memory Access Instructions

/* Code to illustrate memory content manipulation */

a. Initialise Stack Pointer to 0x5200
b. Store 0x83 to memory @ 0x4000
c. Push registers R0-R3 into the stack
d. Load word @ 0x1000 to R0
e. Load word @ 0x1004 into R1
f. Store R0 to memory @ 0x1004
g. Store R1 to memory @ 0x1000 (swap 2 words in memory)
h. Repeat d-g using scaled register offsets
i. Repeat d-g using Swap
j. Repeat d-g using Swap and one register only
k. Pop registers R0-R3 from the stack */

.global START
START:
    MOV SP, #0x5200 ; Set SP = R14, Initialize Stack Pointer
    MOV R0, #0x83 ; Use to access memory @ 0x4000
    MOV R1, #0x4000 ; Load memory address into R1
    STR R0, [R1] ; Store value 0x83 into memory @0x4000
    ; Push registers R0-R3 to stack (Use Full Descending Stack)
    STMFD SP!, {R0-R3} ; Push using SP, Update SP after
    ; Swap 2 words in memory
    LDR R3, =0x1000 ; R3 is array base pointer
    LDR R0, [R3], #4 ; Load R0 from 0x1000, R3 = 0x1004 after
    LDR R1, [R3] ; Load R1 from 0x1004, do not increment R3
    STR R0, [R3], #4 ; Store R0 to 0x1004, R3 = 0x1000 after
    STR R1, [R3] ; Store R1 to 0x1000, do not decrement R3
    ; Repeat instructions using scaled register offset
    MOV R2, #1 ; Use to access memory @ 0x1004
    LDR R0, [R3] ; Keep R3 unchanged (0x1000)
    LDR R1, [R3, R2, LSL #2] ; R2 LSL #2 => 1 x 4
    STR R0, [R3, R2, LSL #2]
    STR R1, [R3]
    ; Repeat instructions using Swap
    ; val(memory addr) indicates contents of memory address location
    ; (? ?) indicates unknown value
    SWP R1, R0, [R3] ; R1 has val(0x1000), 0x1000 has R0 (??)
    SWP R0, R1, [R3, #4] ; @ 0x1004 has R1 (val(0x1000)), R0 has val(0x1004)
    SWP R1, R0, [R3] ; @ 0x1000 has R0 (val(0x1004)), R1 has val(0x1000) (??)
    ; Repeat instructions using Swap and one register only
    SWP R0, R0, [R3] ; @ 0x1000 has R0 (val(0x1000)), 0x1000 has R0 (??)
    SWP R0, R0, [R3, #4] ; 0x1000 has R0 (val(0x1000)), R0 has val(0x1004)
    SWP R0, R0, [R3] ; 0x1000 has R0 (val(0x1004)), R0 has val(0x1000) (??)
    ; Pop registers R0-R3 from stack (Use Full Descending Stack)
    LDMFD SP!, {R0-R3} ; Pop using SP, Update SP after

STOP:
    B STOP

5.2 Integer Arithmetic Instructions

The next group of instructions are used to manipulate values stored in registers. These instructions implement the most common data processing tasks since they manipulate data in registers using typical arithmetic operators. Available arithmetic operators are Add, Subtract, and Multiply. Divide is not provided since its implementation requires more complex logic. Consequently division has to be performed using algorithms. All the arithmetic operations can optionally modify the Flag bits in the Status register using the ‘S’ instruction variant. The Negative, Carry, Zero, and Overflow (N, C, Z, V) flags capture the state after an arithmetic operation has been performed. Often the flags are used to perform conditional operations that depend on the outcome of a given calculation. Flags will be covered in more detail in Chapter 5.4 and Chapter 5.5.
5.2.1 Addition Instructions

The addition instructions uses Addressing Mode 1, which take the first operand as a register Rn, and the second operand as \(<\text{Operand2}>\). The result of the addition will be stored in the destination register Rd. Flags are updated if the ‘S’ variant is specified.

\[
\text{ADD(S)} \ Rd, \ Rn, \ <\text{Operand2}> \ @ \ Rd := Rn + <\text{Operand2}>
\]
\[
\text{ADC(S)} \ Rd, \ Rn, \ <\text{Operand2}> \ @ \ Rd := Rn + <\text{Operand2}> + C
\]

The use of \(<\text{Operand2}>\) as one of the source operands means that we can either add the contents of a register with a constant value, or else the contents of another register that is optionally scaled or rotated. ADD performs the addition of the two source operands without consideration of the Carry (C) flag, whereas ADC adds the two source operands and the value of the Carry flag to obtain the result. Since the Carry flag is often used to indicate result values that exceed the size of a single register (i.e., results which exceed 32 bits in size), we can use two registers to store a double word (64-bit value), and handle the carry from the addition of the least significant word to the addition of the most significant word when that happens, using the ADC (Add With Carry) instruction.

5.2.2 Subtraction Instructions

Similarly, subtraction has analogous formats:

\[
\text{SUB(S)} \ Rd, \ Rn, \ <\text{Operand2}> \ @ \ Rd := Rn - <\text{Operand2}>
\]
\[
\text{SBC(S)} \ Rd, \ Rn, \ <\text{Operand2}> \ @ \ Rd := Rn - <\text{Operand2}> - \neg C
\]

SUB performs subtraction of \(<\text{Operand2}>\) from Rn and stores the result into Rd, whereas SBC (Subtract with Carry) performs the subtraction while taking into account the Borrow (inverse function to Carry) that shares storage in the C Flag (The C flag can hold either the Carry or the Borrow depending on the arithmetic instruction executed).

In addition, there are also Reverse Subtract instructions, RSB (Reverse Subtract) and RSC (Reverse Subtract with Carry) which swaps the position of \(<\text{Operand2}>\) with the first operand:

\[
\text{RSB(S)} \ Rd, \ Rn, \ <\text{Operand2}> \ @ \ Rd := <\text{Operand2}> - Rn
\]
\[
\text{RSC(S)} \ Rd, \ Rn, \ <\text{Operand2}> \ @ \ Rd := <\text{Operand2}> - Rn - \neg C
\]

5.2.3 Multiplication Instructions

5.2.3.1 Dedicated Multiplication Instructions

ARM provides two types of multiplication support, Multiplication of two numbers (either with 32-bit or 64-bit results), and Multiply and Accumulate where the results of the multiplication is added to a third number (either 32-bit or 64-bit). The MUL instruction providing 32-bit results works for either signed or unsigned numbers since only the least significant word of the result is stored, whereas the MULL (Multiply Long) instruction providing 64-bit results has two variants, Signed (S) and Unsigned (U). The
64-bit results is stored in two registers. If the 'S' suffix is provided, the
Condition Flags are updated appropriately. The format of the Multiplica-
tion instructions is as follows:

\[
\text{MUL}(S) \quad \text{Rd, Rm, Rs @ Rd := (Rm x Rs)}[31:0] \\
\text{xMULL}(S) \quad \text{RdLo, RdHi, Rm, Rs} \\
\quad \text{RdLo := (Rm x Rs)}[31:0]; \\
\quad \text{RdHi := (Rm x Rs)}[63:32]; \quad x = (S, U)
\]

Multiply and Accumulate is often associated with digital signal processing
tasks. Most DSPs implement single cycle MAC (Multiply and Accumulate)
instructions to simplify the implementation of algorithms. If the 'S' suffix
is provided, the Condition Flags are updated appropriately. The format of
the Multiply and Accumulate instruction in ARM is as follows:

\[
\text{MLA}(S) \quad \text{Rd, Rm, Rs, Rn @ Rd := ((Rm x Rs) + Rn)}[31:0] \\
\text{xMLAL}(S) \quad \text{RdLo, RdHi, Rm, Rs} \\
\quad \text{RdLo := ((Rm x Rs) + RdHi,RdLo)}[31:0]; \\
\quad \text{RdHi := ((Rm x Rs) + RdHi,RdLo)}[63:32]; \quad x = (S, U)
\]

Multiplication and Multiply and Accumulate takes several cycles to execute
on the ARM7TDMI, depending on the value of the source operand Rs. In
addition, it is different from most other ARM instructions in that it takes
four operands. Again, it is not technically a RISC instruction, but its in-
clusion as a standard part of the instruction set makes the ARM processor
much more powerful and flexible. It is possible to achieve single cycle mul-
tiplication instruction throughput at the expense of a significant increase
in logic gates to implement the feature.

### 5.2.3.2 Equivalent Multiplication Instructions

For most common algorithms, multiplication instructions are very costly
and unnecessary. Any multiplication with powers of two can be imple-
mented quickly using Logical Shift Left (LSL). The relationship between
the Multiplier and LSL is given by: \( \text{Multiplier} = 2^l \), where \( l \) is the number
of bits to left shift. For example, multiplication by 8 is equivalent to LSL
with \#3, and multiplication by 64 is equivalent to LSL with \#6.

\[
\text{MOV R0, R1, LSL \#3 @ R0 := R1 x 2^3}; \quad \text{multiply R1 by 8} \\
\quad \text{LSL R0, R1, \#3 (UAL Syntax)} \\
\text{MOV R3, R2, LSL \#6 @ R3 := R2 x 2^6}; \quad \text{multiply R2 by 64} \\
\quad \text{LSL R3, R2, \#6 (UAL Syntax)}
\]

Since the ADD instruction accepts \(<\text{Operand2}>\) arguments, we can imple-
ment single cycle MAC when the multiplier is a constant power of two (variable
shifts using register specified values take two cycles for ARM7TDMI
[32]). Consequently we can implement limited signal processing algorithms
quickly in ARM using the built-in instructions.

\[
\text{ADD R0, R1, R2, LSL \#2 @ R0 := R1 + R2 x 2^2}; \\
\quad \text{multiply R2 by 4 and add to R1}
\]

86
The use of the Add and Subtract instructions to implement multiplication by multipliers not in powers of two can also be done easily for certain classes of numbers. For the MAC example given previously, we can implement multiply by five by changing the first source operand:

```
ADD R0, R2, R2, LSL #2 @ R0 := R2 + R2 \times 2^2;
```

@ multiply R2 by 2 and add with R2 == multiply by 5

Consequently, single cycle multiplication by \(2^i, 2^i+1,\) and \(2^i-1\) can be achieved easily [19].

```
MOV{S} Rd, Rn, LSL #i @ Rd := Rs \times 2^i; multiply Rs by 2^i
ADD{S} Rd, Rn, Rn, LSL #i @ Rd := Rs + Rs \times 2^i = Rs \times (2^i + 1)
RSB{S} Rd, Rn, Rn, LSL #i @ Rd := Rs \times 2^i - Rs = Rs \times (2^i - 1)
```

### 5.2.4 Division Instructions

#### 5.2.4.1 Dedicated Division Instructions

There are no instructions specifically designed for division. Typically division is not commonly done in most programs, and the logic to handle division is very complex. Consequently, division has to be implemented using suitable routines.

#### 5.2.4.2 Equivalent Division Instructions

Similar to multiplication with powers of two, division by powers of two can be implemented quickly using Arithmetic Shift Right (ASR). It is important to use ASR and not LSR (Logical Shift Right) since the most significant bit (the Sign bit) must be preserved for Signed numbers (There is no Arithmetic Shift Left instruction as the two left shift operations are equivalent). The relationship between the Denominator and ASR is given by: \(Denominator = 2^i,\) where \(i\) is the number of bits to right shift. The result (Quotient) of the division is stored in the destination register. For example, division by 4 is equivalent to ASR with #2, and division by 256 is equivalent to ASR with #8.

```
MOV R0, R1, ASR #2 @ R0 := R1 / 2^2; divide by 4
    @ ASR R0, R1, #2 (UAL Syntax)
MOV R3, R2, ASR #8 @ R3 := R2 / 2^8; divide by 256
    @ ASR R3, R2, #8 (UAL Syntax)
```

The general form for single cycle division by \(2^i\) is as follows:

```
MOV{S} Rd, Rn, ASR #i @ Rd := Quotient (Rs / 2^i); Divide Rs by 2^i
```

### 5.3 Logic and Bit Manipulation Instructions

Bits in a variable are often used as status or Boolean flags. This saves storage space since to represent each flag as a separate byte or word in memory wastes a lot of space when we are only interested in keeping track of two
Table 5.2: Logic Instructions using Addressing Mode 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORR R0, R1, #0xFF</td>
<td>Set least significant 8 bits in R1 to all ‘1’s, and store result into R0. Value in R1 is unchanged. Constant is valid since bit-span of constant fits in 8-bits, with even-rotate count.</td>
</tr>
<tr>
<td>MVN R0, #0x00</td>
<td>Load 4294967295d (unsigned) or -1 (signed) into R0 (1’s complement of 0x00 is 0xFFFFFFFF).</td>
</tr>
<tr>
<td>ORR R7, R7, #0x00800000</td>
<td>Set bit 23 of R7 to ‘1’. Constant is valid since bit-span of constant fits in 8-bits with even-rotate count.</td>
</tr>
<tr>
<td>MVN R3, R0</td>
<td>Copy the 1’s complement value of the contents in R0 to register R3</td>
</tr>
<tr>
<td>AND R3, R2, R0, LSL #3</td>
<td>Using value in R0 multiplied by 8 (LSL #3) as a mask, perform Bitwise AND on the value in R2, and store result to R3; R0 and R2 is unchanged.</td>
</tr>
</tbody>
</table>

values (typically represented by ‘1’ and ‘0’). Consequently logic instructions can be used to manipulate up to 32 flags stored in a single register, which can be used represent the status of a peripheral such as the touch sensor, LED on the light sensor, and whether the motor is on or off.

### 5.3.1 Logic Instructions

Logic instructions treat the values in registers as a collection of bits and not as numbers. Consequently logic operations operate in bit-wise fashion, where the result of the logic operation is derived by using the bit values of the two input operands at a given bit position, and storing the result in the same bit position of the destination register. The common Logic operations are: Or, And, Xor, and Not. Each of them operates on two input operands, except for Not which bit-inverts a single input operand. These logic operators are supported by the ARM using the following mnemonics:

- **ORR(S)** Rd, Rn, <Operand2> @ Bitwise Or
- **AND(S)** Rd, Rn, <Operand2> @ Bitwise And
- **EOR(S)** Rd, Rn, <Operand2> @ Bitwise Xor
- **MVN(S)** Rd, <Operand2> @ Equivalent to ‘Not’

Table 5.2 shows some example of logic instructions in use.

In addition, it is often useful to clear (change bit value to 0), set (change bit value to 1) and toggle (change 1 to 0, and 0 to 1) flags in a variable, in response to changes in some external inputs, WITHOUT affecting the values of the other flags. This can be easily accomplished by means of a Bitmask and using the following logic instructions:

- **BIC(S)** Rd, Rn, <Operand2> @ Clear Rn bits using <Operand2> Bitmask
The Bitmask is used to specify the bit positions that we want to manipulate (clear, set, or toggle). By setting Rd == Rn, the value in the register Rn will be updated accordingly.

### 5.3.2 Shift and Rotate Instructions

Shift and Rotate instructions are used to change the position of bit values in a register. This is useful when processing bitmaps or bit-vectors that represent the status of certain I/O controls. For example, in order to generate a sequence of square waves, a bit-vector of ‘1010101...’ can be used to control the output of an I/O port, where the MSB (sign bit) indicates the current output value. Therefore, by rotating the register to the left one bit at a time, and reading the value of the sign bit, we can set the I/O port output accordingly to give a square wave.

Examples of shift and rotate instructions are: logical and arithmetic bit-shifts, as well as bit-rotates. Bit-shifts modify the bit-position of the bits within a given register, either to the right or left depending on the shift instruction. Bits that are shifted out are discarded, while the resulting empty bit-positions are filled with ‘0’ or ‘1’ bit values depending on the type of shift and the original value of the register. Bit-rotates do not discard any bits from the register. Instead, the bit values are removed from one end of the register and inserted into the other end.

Since the shift and rotate operations are built-in to the ALU, no specific instructions were defined initially. Instead the MOV instruction with appropriate Operand2 arguments were used. However, in the UAL syntax, they can be written explicitly as Shift and Rotate instructions for better clarity. The syntax of the shift and rotate instructions are given below, and their operation illustrated in Figure 5.1.

```
ORR{S} Rd, Rn, <Operand2> @ Set Rn bits using <Operand2> Bitmask
EOR{S} Rd, Rn, <Operand2> @ Toggle Rn bits using <Operand2> Bitmask
```

Each of the Shift and Rotate instructions, except for RRX (Rotate Right Extended) can shift an arbitrary number of bits in one cycle for a constant argument (<imm_shift>) as Operand2. The ASR (Arithmetic Shift Right) instruction preserves the sign bit of a signed number when shifting,
whereas the LSR (Logical Shift Right) instruction does not. As mentioned previously, there is no ASL (Arithmetic Shift Left) instruction since the number maintains its sign bit unless it overflows the value range of the register. In addition, Rotate Left by i bits can be achieved using Rotate Right by (32-i) bits, so there is no specific Rotate Left instruction.

5.4 Comparison and Conditional Execution

5.4.1 Comparison Instructions

Comparison instructions are used to check the value of given registers against constant values or other register values. Two types of compare instructions are available, Arithmetic Compare and Bitwise Compare.

The syntax of the various compare instructions are as follows:

```
CMP Rn, <Operand2> @ Set flags in CPSR for Rn - Operand2
CMN Rn, <Operand2> @ Set flags in CPSR for Rn - (-Operand2)
@ = Rn + Operand2
TEQ Rn, <Operand2> @ Set flags for Rn EOR Operand2;
@ does not affect V flag
TST Rn, <Operand2> @ Set flags for Rn AND Operand2;
@ does not affect V flag
```

The Comparison instructions perform the same operation as their arithmetic and logic counterparts, except that the resultant value is not saved to a register. This makes it possible to check for the outcome of a given logic expression without destroying the contents of the registers being examined. Typical logical expressions are checks for zero values, and checks for limits (lower or upper bounds). These checks are often associated with conditional branches in algorithms, where the execution of a given logic path is

---

1Technically, the two instructions are not equivalent since the Carry flag is updated differently for Rotate Left (C = bit 32-i) and Rotate Right (C = bit 32-i-1)
performed only when the associated logical expression is satisfied. Consequently, comparison instructions are most often paired with Flow Control Instructions (Chapter 5.5) to implement such execution paths in programs.

5.4.2 Condition Flags and their meaning

Condition Flags are used to record the effect of arithmetic, logic and comparison instructions. There are four flags defined for the ARM architecture, Negative (N), Carry (C), Zero (Z), and Overflow (V). These flags are found in the Process Status Register (PSR). The Current PSR (CPSR) is updated each time instructions that affect the flags are executed (whether Comparison instructions or instructions that have the ‘s’ suffix); when the processor enters an exception state, the CPSR is copied into the Exception state Saved PSR (SPSR). The layout of the CPSR and the respective exception mode SPSRs is given in Figure 5.2. Only the Condition Code Flags would be explained in further detail in this section.

5.4.2.1 Negative (N) Flag

The Negative Flag is set based on the value of bit 31. This provides a quick check for performing conditional execution of flow control instructions as well as conditional execution of a given instruction (See Chapter 5.4.3).

5.4.2.2 Zero (Z) Flag

The Zero flag is set when the result in the destination register is all zero, or if two compared values were identical. This provides a quick way to check if the register contents is zero or otherwise, or if a comparison succeeded or not.

5.4.2.3 Carry (C) Flag

The Carry flag is set when a the results of a given addition operation exceeds the size of the destination register, or the results of a given subtraction results in a Borrow due to the first operand being smaller than the second operand. In addition, the Carry flag is also affected by Shift and Rotate Logic Instructions.
5.4.2.4 Overflow (V) Flag

Overflow occurs due to arithmetic operations on two signed numbers that resulted in a change in the sign of the result due to the value of the result exceeding the size of the register. Consequently, if the addition of two large positive signed numbers resulted in a negative result, or the addition of two large negative signed numbers resulted in a positive result, they will cause an Overflow. Note that the Overflow flag is Unpredictable after Multiply Long operations.

5.4.3 Conditional Execution

Normally, conditional execution is associated with Flow Control Instructions (Chapter 5.5). As mentioned previously, the ARM architecture is unusual in that all the 32-bit ARM instructions all have conditional execution capability. This means that an instruction is executed only if the current Condition Flags (N, C, Z, V) match the predefined conditions specified in the instruction encoding. Consequently, each instruction in the ARM instruction set has an optional <cond> suffix, which is added to the instruction mnemonic before the sign ‘s’ suffix, if the instruction were to be executed conditionally.

The various conditions that control the execution of a given instruction were listed in Table 4.11. Briefly they are organized into three groups, Equality Conditions, Unsigned Conditions, and Signed Conditions.

In the Equality Conditions, the tests are for Equal (EQ) or Not Equal (NE). For Unsigned Conditions, the tests are for Higher (HI), Higher or Same (HS), Lower (LO), and Lower or Same (LS). Signed Conditions include Greater Than (GT), Greater Than or Equal (GE), Less Than (LT), and Less Than or Equal (LE). Note that signed and unsigned conditions have different naming convention to make their meaning unambiguous. It is important to use the correct signed or unsigned conditional checks so as not to introduce subtle logic errors into your algorithm. There is also an Always Condition, which does not need to be specified explicitly. The default state of an instruction is unconditional execution (i.e., always execute).

5.5 Flow Control Instructions

Branch instructions are typically categorized into Unconditional and Conditional branch instructions. Furthermore, most processors provide different types of branch instructions, namely short branches relative to the current position of the PC (typically referred to simply as a Branch instruction), and long branches that use the address of the destination instruction (typically referred to as a Jump instruction). Since the PC in the ARM architecture is treated almost like any other general purpose register where the value of the PC can be modified using the MOV instruction, it is not necessary to provide any dedicated long branch instructions to update the PC contents.

There are four flow control instructions in the ARM architecture. They are Branch (B), Branch & Link (BL), Branch & Exchange (BX), and Super-
visor (SVC), also known as Software Interrupt (SWI). Branch instructions changes the sequence of instructions from one consecutive processing sequence to some other location to continue execution. This is important for implementing subroutines or functions, which are frequently used sequence of code performing a given function, as well as for making decisions regarding which block of code to execute depending on the state of the robot or the value of some variable.

The syntax of the four flow control operations are:

- `B{<cond>} <label> @` Label must be +/- 32 MB
- `BL{<cond>} <label> @` Label must be +/- 32 MB
- `BX{<cond>} Rm @` Only BX accepts a pointer as an argument
- `SVC{<cond>} <imm24> @` UAL mnemonic, SWI was the old syntax

5.5.1 Gotos using Unconditional Branches

Unconditional branches are essentially Goto instructions. The program flow will always proceed to the new instruction address. The Branch (B) instruction accepts a label (a named position in the assembly source file) as the target. This label must be within ±32MB of the current PC.

```
B <label> @ Target Address within +/- 32 MB
```

In addition, the Branch and eXchange (BX) instruction is provided to enable switching between ARM and Thumb processing states.

```
BX Rm @ Branch and Exchange (between ARM and Thumb states)
```

Note that the target address for the BX instruction must be loaded into a register first. Rm is the Target Address Pointer. This will be covered in greater detail in Chapter 8 when dealing with Thumb state.

It is frequently useful to refer to a target address using a register variable. This allows us to modify the destination address to branch to without requiring the use of multiple Branch instructions with all different target addresses. The equivalent code to Branch using a Target Address Pointer is given by:

```
MOV PC, Rm @ Equivalent to ‘Branch Rm’
```

This is also needed for performing long branches in the ARM architecture where the target address can be anywhere in addressable memory, since the target address can access the full 32-bit address space.

5.5.2 Decision Making Using Conditional Branches

Conditional branches are like decision diamonds in a flow chart. The branch will be taken or otherwise based on the result of a Boolean test, where the condition evaluates to either TRUE or FALSE. Conditional branches are often used in conjunction with Comparison instructions since the logic described in the comparison statements provides the needed Boolean output
to give the branch/don’t branch decision. The condition for the branch de-
cision is included in the flow control instruction as a suffix <cond>, where
the conditions defined in Table 4.11 are used to determine if the branch is
taken or otherwise.

Conditional branches are important for implementing most of the com-
mon program flow control operations. Examples from the C language are:

```plaintext
if...then...else
switch
while
do...while
for
```

The result of a successful conditional branch is to transfer processing to the
address operand given in the conditional branch statement. In contrast, an
unsuccessful conditional branch means that program execution continues
with the next instruction in sequence after that conditional branch state-
ment. For example, a simple algorithm to illustrate conditional branching
given in pseudocode format in Listing 5.3. In this algorithm, the MIND-
STORMS NXT checks for a button press value that was read from the Touch
sensor previously, and stored into a memory variable (We won’t worry about
how to access inputs from the sensors yet). First, we need to retrieve that
value into register R0, since the ARM processor can only perform compar-
isons on values stored in registers. Then if the Button were pressed, mean-
ing that the register value R0 is non-zero (True), it will perform some ac-
tion, otherwise it will read the variable stored in memory location 0x4000
and load it into register R1, and continue execution from ‘(*).’

**Listing 5.3: Conditional Branching Example Pseudocode**

```plaintext
Load Buttonpress Status (Boolean value) into R0
If value in R0 is True (non-zero), then
Branch to statement location PRESSED
else
Load value in memory M(0x4000) into R1
endif
... (*)
PRESSED: ...
```

The equivalent ARM Assembly instructions are given in Listing 5.4.

**Listing 5.4: ARM Assembly version of Conditional Branching Example**

```plaintext
@ ARM Assembly version of Button Pressed Checking Algorithm
@
@ Load Buttonpress Status (Boolean value) into R0
@ If value in R0 is True (non-zero), then
@ Branch to statement location PRESSED
@ else
@ Load value in memory M(0x4000) into R1
@ endif
@ ... (*)
@
@ R0: Button Press Value
@ R1: Address Pointer
.data /* Data Storage Area */
```
.align 4    /* Important for Word sized access */
Buttonpress: .word 0x1 /* Boolean: 0 = FALSE, Non-0 = TRUE */

.code 32   /* ARM Mode */
.text      /* Code Storage Area */
.align 4   /* ARM code needs to be word aligned */

.global START
START:
LDR R1, =Buttonpress /* Load address of Buttonpress into R1 */
LDR R0, [R1]    /* Load Buttonpress into R0 */
CMP R0, #0     /* Compare against 0, Flags updated */
BNE PRESSED    /* Non-zero value in R0, so Branch */

/* Else */
MOV R1, #0x4000 /* Setup pointer to location 0x4000 */
LDR R1, [R1]   /* Retrieve value from memory into R1 */
/* Endif */
...
(**)          /* Continue execution */
PRESSED:      /* Alternative action */
...
.end

5.5.3 Subroutine Calls using the Link Register

The Link Register (LR), which is stored in R14, is used to perform function call return to the calling functions. This will be covered in greater detail in Chapter 6.3 when dealing with subroutines and functions.

Branch & Link is the process used to call a subroutine from the current location in the code. We can either call the subroutine directly via its address as an operand for the BL instruction, or else, call it indirectly using the address stored in a register (represented here as Rm). The equivalent code in ARM state for performing an indirect Branch & Link using a Function Address Pointer is:

```
MOV LR, PC
MOV PC, Rm
...
```

When the subroutine is a Thumb function (see Chapter 8.3), we would need to perform a Branch, Link & Exchange to the given subroutine, since we need to switch operation state upon entering the subroutine. A direct call to a Thumb subroutine involves loading the Thumb subroutine address into a temporary register, since the BX instruction does not allow us to specify the target address as an immediate operand. This is explained in detail in Chapter 8.3. The indirect call to a Thumb subroutine is much simpler. The equivalent code in ARM state for performing an indirect Branch, Link & Exchange to a Thumb subroutine using a Function Address Pointer in register Rm is:

```
MOV LR, PC
BX Rm
...
```

The reason that both the indirect Branch & Link, and indirect Branch, Link & Exchange code sequences work, is because when the 'MOV LR, PC'

---

2ARMv5T provides a dedicated BLX instruction for this purpose, which unfortunately is not available for our use.
instruction was executed, the PC is actually pointing to the instruction following the ‘MOV PC, Rm’ and the ‘BX Rm’ instructions, due to the use of the three stage pipeline in the ARM7TDMI architecture. Consequently, when the subroutine returns, it will return using the value in LR which points to the instruction following the function call.

5.5.4 Kernel Mode Access Using Supervisor Exceptions

Technically, SVC (Supervisor Call) is not a Branch instruction, in the sense that program execution continues normally from the target address. Instead, it switches operation mode from the current mode into Supervisor (SVC) mode, where the instruction triggers the SVC Exception Handler to determine what Kernel service is needed by the program. Normally the SVC instruction has a code attached (<imm24>), that is used to identify the required service. Typical kernel services may be to Read Inputs from the user, or Generate Outputs to the peripheral devices.

The mapping between service codes and kernel functions is operating system or firmware specific; you would need to refer to the particular operating system or firmware to find out what features are provided via the SVC interface.

5.6 Coprocessor Instructions

ARM has four coprocessor instructions:

LDC @ Load Coprocessor
STC @ Store Coprocessor
MCR @ Move to Coprocessor from ARM Register
MRC @ Move to ARM Register from Coprocessor

ARM Coprocessor instructions will not be discussed further in this book since the AT91SAM7S does not have any coprocessor functions.

5.7 Line Following Basics

5.7.1 Introduction to Line Following

Line Following is a basic robot navigation technique. Many robots use markings on the floor to navigate correctly from one place to another. As the robot moves along a predefined path given by the floor markings, variation in wheel speeds and friction will cause the robot to veer off course. In addition, changes in the direction of the path to turn a corner, will require the robot to track the markings correctly to determine when it should move to the right or left to keep itself on the path.

For the MINDSTORMS robot, the Light Sensor is often used to detect the presence or absence of these markings. The markings are often in the form of a line of a certain width (2-3 cm wide). We can determine the most intense and least intense inputs from the Light Sensor, and deduce whether it is on top of a line, at the edge, or not on a line. The Light Sensor readings
can be retrieved either as normalized (0-100%) or raw readings (unsigned 10-bit values). While it is good that the Light Sensor can provide us with a way to detect the line markings, these inputs are analog in nature, and can vary depending on the reflectivity of the surface, as well as changes in the position of the Tribot due to movement, especially when we are on the edge of the line (Figure 5.3). Consequently it would be good to collect a series of input values and either average them or determine the minimum and maximum values that they fall between before deciding whether we are correctly following the line or not. 

This can be summarized as follows in Table 5.3.

### Table 5.3: Range of Light Sensor Readings for Line Following

<table>
<thead>
<tr>
<th>Case</th>
<th>Color</th>
<th>Readings</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Inside Line</td>
<td>Black</td>
<td>High</td>
</tr>
<tr>
<td>II: Edge</td>
<td>Black/White/Gray</td>
<td>Intermediate</td>
</tr>
<tr>
<td>III: Outside Line</td>
<td>White</td>
<td>Low</td>
</tr>
</tbody>
</table>

5.7.2 Searching for Min and Max values

We will soon examine how Tribot implements a line-following function, to better understand how the various Assembly Language instructions work together. Before we get into the details of how inputs are retrieved from the sensors in the MINDSTORMS, we will first study the implementation of an abstract Min-Max algorithm in Assembly Language. For this example,
we assume that some function has already collected a series of normalized readings (i.e., values are from 0-100) and stored them in an array in memory. Since these are normalized readings, we can use a byte sized array to store the readings. Subsequently, we can search for the minimum and maximum values in the list using flow control and comparison instructions. The minimum and maximum values will be stored into min and max variables in memory at the end of the program. For simplicity, we use a linear search which does not require the list to be sorted beforehand. In addition, we assume that the numbers are unsigned 8-bit values and are stored in an array in memory with starting index = 0 (C language convention). This is illustrated in the following pseudocode listings and ARM assembly program.

First we write down the algorithm based on our understanding of the problem (searching an array of light sensor readings to find the min and max values). The pseudocode is given in Listing 5.5.

Listing 5.5: Pseudocode for Min-Max Search Algorithm using For Loop

Given n 8-bit unsigned numbers in array readings[]
max = 0
min = 255
For counter = 0 to (n-1)
  if (readings[counter] < min) then
    min = readings[counter]
  if (readings[counter] > max) then
    max = readings[counter]
endfor

Here, we use a for loop that counts from 0 to (n-1), a total of n numbers. In a for loop, there is an implicit comparison against a non-zero number (counter < n) required to determine when to exit the loop. In reality, Compilers convert for loops into while loops as most instruction sets do not have built-in for-endfor instructions. *(This is reflected in the C language syntax: for (counter = 0; counter < n; counter++).*

Converting a for loop into a while loop results in Listing 5.6, where we perform the comparison at the start of the loop. Note the value of the comparison limit and comparison expression used.

Listing 5.6: Min-Max Search Algorithm using While loop

Given n 8-bit unsigned numbers in array readings[]
max = 0
min = 255
counter = 0
while (counter < n)
  if (readings[counter] < min) then
    min = readings[counter]
  if (readings[counter] > max) then
    max = readings[counter]
  counter = counter + 1
endwhile

We can now implement this in ARM assembly language (Listing 5.7).
Listing 5.7: ARM Assembly Min-Max Search Using While Loop (Alg-5.7.S)

```assembly
; ARM Assembly version of Min-Max Algorithm
; (c) 2011 TC Wan, USM

; Given n 8-bit unsigned numbers in array readings[]
; max = 0
; min = 255
; counter = 0
; while (counter < n)
;   if (readings[counter] < min) then
;     min = readings[counter]
;   end if
;   if (readings[counter] > max) then
;     max = readings[counter]
;   end if
;   counter = counter + 1
; endwhile

.equ n, 5 ; Number of items in array */
.data
.align 4
readings: .byte 70, 3, 95, 0, 23
min: .byte 0
max: .byte 0
.code 32
.text
.align 4

/*
R0: Temporary value storage
R1: counter
R2: Address Pointer
R3: min value
R4: max value */

.global START
START:
    MOV R4, #0 ; Initialize max to smallest 8-bit value */
    MOV R3, #255 ; Initialize min to largest 8-bit value */
    MOV R1, #0 ; Initialize counter */
    LDR R2, =readings ; Array Address Pointer, setup outside loop */
loop:
    CMP R1, #n ; Flags updated using (R1-#5) */
    BHS exitloop ; HS: unsigned compare */
    /* If */
    LDRB R0, [R2, R1] ; Retrieve readings[counter] into R0 */
    check_min:
      CMP R0, R3 ; check readings[counter] (in R0) < min */
      BHS check_max ; */ Min = readings[counter] */
    check_max:
      CMP R0, R4 ; check readings[counter] (in R0) > max */
      BLS incr_counter ; */ Max = readings[counter] */
    incr_counter:
      ADD R1, R1, #1 ; */ counter = counter + 1 */
    loop ; */ Next iteration */
exitloop:
    LDR R2, =min ; */ Store min into memory location */
    STRB R3, [R2]
    LDR R2, =max ; */ Store max into memory location */
    STRB R4, [R2]
    done:
    B done ; */ Done, loop forever */
.end
```

After assembly, an object file is generated. The corresponding listing file output is given in Listing 5.8.
Listing 5.8: Listing of While Loop Min-Max Search (Alg-5.7.S)

ARM GAS Alg-5.7.S page 1

1 /* ARM Assembly version of Min-Max Algorithm */
2 * (c) 2011 TC Wan, USM
3 *
4 * Given n 8-bit unsigned numbers in array readings[]
5 * max = 0
6 * min = 255
7 * counter = 0
8 * while (counter < n)
9 * if (readings[counter] < min) then
10 * min = readings[counter]
11 * if (readings[counter] > max) then
12 * max = readings[counter]
13 * counter = counter + 1
14 * endwhile
15 */
16
17 .equ n,5 /* Number of items in array */
18
19 .data
20 .align 4
21 0000 46035F00 readings: .byte 70, 3, 95, 0, 23
22 0005 00 min: .byte 0
23 0006 00 max: .byte 0
24
25 .code
26 0007 00000000 .text
27 00
28 .align 4
29
30 /*
31 * R0: Temporary value storage
32 * R1: counter
33 * R2: Address Pointer
34 * R3: min value
35 * R4: max value
36 */
37
38 .global START
39 START:
40 0000 0040A0E3 MOV R4, #0 /* Initialize max to smallest 8-bit value */
41 0004 FF30A0E3 MOV R3, #255 /* Initialize min to largest 8-bit value */
42 0008 0010A0E3 MOV R1, #0 /* Initialize counter */
43 loop:
44 0010 050051E3 CMP R1, #n /* Flags updated using (R1-#5) */
45 0014 0800002A BHS exitloop /* HS: unsigned compare */
46 47 /* If */
48 0018 0100D2E7 LDRB R0, [R2, R1] /* Retrieve readings[counter] into R0 */
49 check_min:
50 001c 030050E1 CMP R0, R3 /* check readings[counter] (in R0) < min */
51 0020 0000002A BHS incr_counter /* R0: unsigned compare */
52 incr_counter:
53 0024 011081E2 ADD R1, R1, #1 /* counter = counter + 1 */
54 0028 F4FFFFEA B loop /* Next iteration */
55 exitloop:
56 002C 0000000A LSL incr_counter /* R0: signed compare */
57 0030 0040A0E3 MOV R4, R0 /* Max = readings[counter] */
58 incr_counter:
59 0034 011081E2 ADD R1, R1, #1 /* counter = counter + 1 */
60 0038 F4FFFFEA S Loop /* Next iteration */
61 exitloop:
62 003C 0000000A LSL R2, #min /* R2: signed compare */
63 0040 003000E3 STRB R3, [R2] /* Store min into memory location */
64 0044 004000E3 STRB R4, [R2] /* Store max into memory location */
65 done:
66 0048 004000E3 S done /* Done, loop forever */
67 0050 00000000 .end
68 0050 00000000
69 0050 00000000
70 0050 00000000
71
ARM GAS Alg-5.7.S page 2

ARM GAS Alg-5.7.S page 3

DEFINED SYMBOLS

Alg-5.7.5:17 .equ n,5
Alg-5.7.5:20 .data:readings: 0x70, 0x03, 0x95, 0x0, 0x23
Alg-5.7.5:21 .data:0x00000000 min
Alg-5.7.5:22 .data:0x00000000 max
Alg-5.7.5:23 .data:0x00000000 START

100
However, since we’ve discussed algorithm optimization previously, we find that a for or while loop can be rewritten as a Do-While loop, which eliminates a comparison against a non-zero number n. Instead, we take advantage of the Z flag to determine when to exit the loop (Listing 5.9).

Listing 5.9: Min-Max Search Algorithm Optimization using Do-While loop

Given n 8-bit **unsigned** numbers in array readings[]

- \( \text{max} = 0 \)
- \( \text{min} = 255 \)
- \( \text{counter} = n \)

**do**

- \( \text{counter} = \text{counter} - 1 \)
- **if** \( \text{readings[counter]} < \text{min} \) then
- \( \quad \text{min} = \text{readings[counter]} \)
- **if** \( \text{readings[counter]} > \text{max} \) then
- \( \quad \text{max} = \text{readings[counter]} \)

**while** \( \text{counter} <> 0 \)

Note: Care must be taken when converting the original algorithm to maintain its equivalence. This is especially important when it comes to array indices and number of iterations through a loop. In Listing 5.10 we maintain n passes through the loop, with array indices from \( (n-1) \) to 0. Although the processing order is from back to front, it does not affect the actual behavior of the algorithm.

Listing 5.10: Optimized ARM Assembly Min-Max Algorithm

```assembly
/* ARM Assembly version of Min-Max Algorithm */
(c) 2011 TC Wan, USM

- Given n 8-bit unsigned numbers in array readings[]
- \( \text{max} = 0 \)
- \( \text{min} = 255 \)
- \( \text{counter} = n \)

**do**

- \( \text{counter} = \text{counter} - 1 \)
- **if** \( \text{readings[counter]} < \text{min} \) then
- \( \quad \text{min} = \text{readings[counter]} \)
- **if** \( \text{readings[counter]} > \text{max} \) then
- \( \quad \text{max} = \text{readings[counter]} \)

**while** \( \text{counter} <> 0 \)

..equ n, 5  /* Number of items in array */

.data
.align 4
readings: .byte 70, 3, 95, 0, 23
min: .byte 0
max: .byte 0

.code 32
.text
```

101
The corresponding Listing file output is given in Listing 5.11. As we can see, four instructions were saved (for a total of 16 bytes) from the code, due to the use of the do...while loop structure, the use of Conditional Execution instructions (MOVHI, MOVLO) for updating the registers containing min and max values, which eliminated the conditional branch instructions (note that the conditions are different between the Branch case and the Conditional Execution case); as well as by using the Immediate Post-Indexed Addressing mode when storing the final min and max values to memory.
5.8 Line Follower Robot

Having seen the abstract Min-Max algorithm we will now implement it using the Tribot in order to understand how it works in an actual example (Figure 5.4). The source code for the Line Follower is given in Listing 5.13.

Some constants and defines that will be used by the tribot applications are kept in a shared include file tribot.h given in Listing 5.12.

Listing 5.12: ARM Assembly Header tribot.h

```c
/** @file tribot.h
 * @brief Definitions of common Tribot constants, etc.
 */
```

5.8 Line Follower Robot

Having seen the abstract Min-Max algorithm we will now implement it using the Tribot in order to understand how it works in an actual example (Figure 5.4). The source code for the Line Follower is given in Listing 5.13. Some constants and defines that will be used by the tribot applications are kept in a shared include file tribot.h given in Listing 5.12.
Figure 5.4: Line Follower Robot

/*
 * Copyright (C) 2007-2012 the NxOS developers
 * See AUTHORS for a full list of the developers.
 * Redistribution of this file is permitted under
 * the terms of the GNU Public License (GPL) version 2.
 */

#ifndef __NXOS_SYSTEMS_TRIBOT_H__
#define __NXOS_SYSTEMS_TRIBOT_H__

/** @addtogroup tribot */
/*@{*/

/* Timer Constants */
.extern nx_systick_wait_ms
.equ SYSTICK_1000MS, 1000
.equ SYSTICK_500MS, 500
.equ SYSTICK_100MS, 100
.equ SYSTICK_50MS, 50
.equ SYSTICK_1MS, 1

/* Sensor Port Assignments */
.equ TOUCH_PORT, 0 /* Sensor 1 */
.equ SOUND_PORT, 1 /* Sensor 2 */
.equ LIGHT_PORT, 2 /* Sensor 3 */
.equ ULSND_PORT, 3 /* Sensor 4 */
.equ DIGIO, 0
.equ DIG11, 1

/* Actuator Port Assignments */
.extern nx_motors_rotate
.extern nx_motors_stop
.equ CLAW_PORT, 0 /* Motor A */
.equ RWHEEL_PORT, 1 /* Motor B */
.equ LWHEEL_PORT, 2 /* Motor C */

/* Min-Max constant */
.equ n, 5 /* Number of items in array */

/** Name Robot State Enums */
 * Robot State.
 * The enums must be consecutive, starting from 0
 */
/*@}*/
We see that the Min-Max code from Listing 5.11 was used pretty much as given in Listing 5.13, except that there is a pair of push and pop instructions surrounding the block of code, in order to make the min-max algorithm a subroutine. The program also utilizes various supporting routines from NxOS (these are prefixed with 'nx_'), which are documented in the NxOS source repository. The conversion of algorithms into subroutines and how to call routines that expect one or more parameters will be described in greater details in Chapter 6. We will just assume that the routines are called correctly for now.

The Line Follower Program also illustrates some issues related to reading sensor inputs. To understand the behavior of the Light Sensor, we will enable a Debugging Version of the program which only displays the Min and Max values of the normalized Light Sensor readings, by enabling #define DEBUG_MIN_MAX (Listing 5.13). Since inputs are analog in nature, they tend to have fluctuations in values even when the robot is not moving. This is due to slight variations in the sensor position and light intensity, and is made even more pronounced when the robot is actually in motion. Consequently, it is important that decisions for the line following algorithm be based on a range of values, and these values should be calibrated using the DEBUG_MIN_MAX mode beforehand in order to cater for the type of surface the robot operates on. In addition, we must also work within the limitations of the sensor value retrieval process. The NXT requires a 3 ms interval between successive reads of analog sensor values due to hardware limitations. Consequently, a 3 ms delay was inserted in the get_light_readings code block to ensure that the values read via the nx_sensors_analog_get_normalized() routine are valid. Finally, the reading of sensor values is interleaved with the movement logic, so that the robot continues to operate using the previous Min and Max thresholds while gathering a new set of readings from the Light Sensor (Listing 5.13).
* Redistribution of this file is permitted under
  * the terms of the GNU Public License (GPL) version 2.

#define __ASSEMBLY__
#include "base/interwork.h"
#include "base/lib/scaffolding/scaffolding.h"
#include "armdebug/Debugger/debug_stub.h"
#include "systems/tribot/include/tribot.h"

/* To disable Manual Breakpoints, change the following to #undef ENABLE_BREAKPOINTS */
#undef ENABLE_BREAKPOINTS
/* To disable Tones, change the following to #undef ENABLE_TONES */
#undef ENABLE_TONES
/* To disable Motor Debugging, change the following to #undef DEBUG_MOTOR */
#undef DEBUG_MOTOR
/* To disable Min-Max Debugging, change the following to #undef DEBUG_MIN_MAX */
#undef DEBUG_MIN_MAX

#if defined DEBUG_MIN_MAX
  .equ MAX_ITERATIONS, 100
#else
  .equ MAX_ITERATIONS, 0x0FFFF
#endif

/* Light Sensor Color Detection Intensity (0-100) ranges */
/* 8547 TestPad White: 49-50; Edge: 54; Black: 66-67; Blue: 60-64 */
/* Changed the range so that it will be a smooth transition from White to Black */
  .equ WHITE_MIN, 45  
  .equ WHITE_MAX, 53  
  .equ EDGE_MIN, 54  
  .equ EDGE_MAX, 64  
  .equ BLACK_MIN, 65  
  .equ BLACK_MAX, 70

/* Motor Control Constants */
  .equ FWD_SPEED, 50  
  .equ FASTROT_SPEED, 40  
  .equ SLOWROT_SPEED, 20

.data
.align 4
.title: .asciz "Linefollower Bot"
white: .asciz "White!"
black: .asciz "Black!"
edge: .asciz "Edge!"
unknown: .asciz "Unknown!"

#if defined DEBUG_MIN_MAX
minstring: .asciz "min: 
maxstring: .asciz "max: 
#endif

.readings: .space n, 0x0
.min: .byte 0
.max: .byte 0
.robot_state: .byte 0

.code 32
.text
.align 4

/* Min-Max Routine 
 * R0: Temporary value storage 
 * R1: counter 
 * R2: readings Array Pointer 
 * R3: min value 
 * R4: max value */
.global min_max
min_max:
push {r4, lr}
MOV R4, #0 /* Initialize max to smallest 8-bit value */
MOV R3, #255 /* Initialize min to largest 8-bit value */
MOV R1, #n /* Initialize counter */
LDR R2, =readings /* Array Address Pointer, setup outside loop */
min_max_loop:
SUB R1, R1, #1 /* counter = counter - 1 */
check_min:
LDRB R0, [R2, R1] /* Retrieve readings[counter] into R0 */
CMP R0, R3 /* check readings[counter] (in R0) < min */
MOVLO R3, R0 /* Min = readings[counter] */
check_max:
CMP R0, R4 /* check readings[counter] (in R0) > max */
MOVHI R4, R0 /* Max = readings[counter] */
next_iter:
TEQ R1, #0 /* Compare R1 against zero */
BNE min_max_loop /* NE: R1 is non-zero */
exit_min_max_loop:
LDR R2, =min /* Store min into memory location */
LDRA R3, =min /* Store max into memory location */
done_min_max:
pop {r4, pc}

/* Actuator Primitives */
#ifdef ENABLE_TONES
plip:
push {lr}
ldr r0, =3000
mov r1, #100
bl nx_sound_freq
pop {pc}

plop:
push {lr}
ldr r0, =500
mov r1, #100
bl nx_sound_freq
pop {pc}

plap:
push {lr}
ldr r0, =1500
mov r1, #100
bl nx_sound_freq
pop {pc}
#endif
light_led_enable:
push {lr}
mov r0, #LIGHT_PORT
mov r1, #DIGI0
bl nx_sensors_analog_digi_set
pop {pc}
light_led_disable:
push {lr}
mov r0, #LIGHT_PORT
mov r1, #DIGI0
bl nx_sensors_analog_digi_clear
pop {pc}
black_detected:
push {lr}
#ifdef ENABLE_TONES
bl plip
#endif
ldr r0, =black
ldr r2, =robot_state
mov r1, #ROBOT_FWD
ldrb r0, [r2]
teq r0, r1
b eq 1f /* Already in given state */
strb r1, [r2] /* State Change: Update new state */

mov r0, #RWHEEL_PORT
mov r1, #FWD_SPEED
bl nx_motors_rotate
mov r0, #LWHEEL_PORT
mov r1, #FWD_SPEED
bl nx_motors_rotate

1:
pop {pc}

white_detected:
push {lr}

#ifdef ENABLE_TONES
bl plop
#endif

ldr r0, =white
bl nx_progcontent

ldr r2, =robot_state
mov r1, #ROBOT_CW
ldrb r0, [r2]
teq r0, r1
b eq 1f /* Already in given state */
strb r1, [r2] /* State Change: Update new state */

mov r0, #RWHEEL_PORT
mov r1, #FALSE /* Don't brake */
bl nx_motors_stop
mov r0, #LWHEEL_PORT
mov r1, #FASTROT_SPEED
bl nx_motors_rotate

1:
pop {pc}

edge_detected:
push {lr}

#ifdef ENABLE_TONES
bl plap
#endif

ldr r0, =edge
bl nx_progcontent

ldr r2, =robot_state
mov r1, #ROBOT_CWM
ldrb r0, [r2]
teq r0, r1
b eq 1f /* Already in given state */
strb r1, [r2] /* State Change: Update new state */

mov r0, #RWHEEL_PORT
mov r1, #FALSE /* Don't brake */
bl nx_motors_stop
mov r0, #LWHEEL_PORT
mov r1, #FASTROT_SPEED
bl nx_motors_rotate

1:
pop {pc}

unknown_detected:
push {lr}

ldr r0, =unknown
bl nx_progcontent

ldr r2, =robot_state
mov r1, #ROBOT_STOP
ldrb r0, [r2]

108
teq r0, r1
beq 1f     /* Already in given state */
strb r1, [r2]     /* State Change: Update new state */

mov r0, #RWHEEL_PORT
mov r1, #FALSE     /* Don’t brake */
bl nx_motors_stop
mov r0, #LWHEEL_PORT
mov r1, #FALSE     /* Don’t brake */
bl nx_motors_stop

1:
pop (pc)

/** init_robot
* Robot Specific Initialization tasks
* Customize this routine for your needs
* Note: Modifies various register contents on exit
*/
init_robot:
push {lr}
    /* Configure Sensors */
mov r0, #LIGHT_PORT
bl nx_sensors_analog_enable
mov r0, #TOUCH_PORT
bl nx_sensors_analog_enable
ldr r5, =readings
mov r7, #FALSE
ldr r8, =MAX_ITERATIONS
pop {pc}

/** shutdown_robot
* Robot Specific Shutdown tasks
* Customize this routine for your needs
*/
shutdown_robot:
push {lr}
    unknown_detected        /* Stop motors */
pop {pc}

/** main
* Main Program
* R5: Address Pointer to Light Sensor Readings Array
* R6: Index of Light Sensor Readings
* R8: Iteration Counter (termination)
*/
.global main
main:
push {lr}
    nx_proginit
    ldr r0, =title
    bl nx_progtitle

#ifdef ENABLE_BREAKPOINTS
    dbg__bkpt_arm
#else
    mov r0, #SYSTICK_1000MS
    bl nx_systick_wait_ms
#endif

break:
    nop     /* Provide GDB with a safe instruction to breakpoint at */

没有想到------------------------- Begin Customization Here -------------------------
    bl init_robot
#ifdef DEBUG_MOTOR
    /* Left and Right Wheel Motor Debugging Code */
mov r0, #LWHEEL_PORT
mov r1, #FWD_SPEED
bl nx_motors_rotate
mov r0, #SYSTICK_1000MS
bl nx_systick_wait_ms
mov r0, #RHEEL_PORT
mov r1, #FALSE  /* Don't brake */
bl nx_motors_stop
mov r0, #RHEEL_PORT
mov r1, #FWD_SPEED
bl nx_motors_rotate
mov r0, #SYSTICK_1000MS
bl nx_systick_wait_ms
mov r0, #RHEEL_PORT
mov r1, #FALSE  /* Don't brake */
bl nx_motors_stop
#endif

/* Line Follower Algorithm:
  1. Collect Samples
  2. Find Min-Max (range)
  3. Determine if we're:
      * -- inside line
      * -- on edge
      * -- outside line
  4. If Inside Line, move straight ahead
  5. If On Edge, rotate slowly
  6. If Outside Line, rotate quickly
*/
bl light_led_enable
mov r6, #n  /* Number of readings to collect */
main_loop:
#ifdef DEBUG_MIN_MAX
mov r0, #SYSTICK_1000MS
bl nx_systick_wait_ms
#endif
subs r8, r8, #1
beq robot_stop
get_light_readings:
sub r6, r6, #1  /* Convert to Index */
mov r0, #LIGHT_PORT
bl nx_sensors_analog_get_normalized  /* returned value in range 0-100% */
strb r0, [r5, r6]
mov r0, #SYSTICK_3MS
bl nx_systick_wait_ms  /* Delay 3 ms before next reading (Required by AVR A/D Converter) */
cmp r6, #0
bhi check_light_readings  /* Skip new Min-Max calculation until we have enough samples */
calc_min_max:
bl min_max
mov r6, #n  /* Reset Number of readings to collect */
#ifdef ENABLE_BREAKPOINTS
dbg__bkpt_arm
#endif
check_light_readings:
ldr r2, =min
ldrb r0, [r2]  /* min */
ldr r1, [r2, #1]  /* max */
#ifdef DEBUG_MIN_MAX
mov r9, r0  /* keep min in R9 */
mov r10, r1  /* keep max in R10 */
#endif
5.9 Chapter Summary

- The various instruction categories for ARM were presented and discussed.

- A complete ARM assembly language program was presented and discussed to highlight usage of various instructions.
A Line-Following Program was developed to illustrate various programming concepts.

5.10 Review Questions and Problems

1. Write a short program to calculate the 8-bit checksum of a byte array of length m in memory. The checksum is calculated as the Exclusive OR (EOR) of each byte in the array, and the result truncated to 8-bits for storage at the end of the array. i.e., the checksum is stored in array index m (for index starting from 0).

2. Rewrite the Min-Max algorithm to handle raw light intensity values which are unsigned 10-bit numbers. Note that the 10-bit values are right justified in 32-bit sized registers (i.e., R0[9:0]).

3. Starting from the code for the Line Follower Robot, modify it to enable the Tribot to successfully navigate a course which has straight runs, rounded and sharp corners, and zig-zag line patterns.

4. Enhance the Line Follower Robot code to handle the case where the line along the course has a 15 cm gap between the first part of the course and the second part of the course.
Chapter 6

Divide and Conquer

I gave her one, they give him two,
You gave us three or more;
They all returned from him to you,
Though they were mine before.

from “Alice’s Adventures in Wonderland,” Lewis Carroll, 1832-1898

Complex problems are often solved by first decomposing them into smaller chunks where the solution can be worked out much more easily compared with tackling it head on. Consequently, when we are trying to get Tribot to move around a room without bumping into obstacles, we would need to process inputs from various sensors (e.g., Touch, Ultrasound, Light) determine if we have encountered any obstacles, and then make appropriate changes to the direction of movement.

6.1 Introduction to Behavior-Based Robotics

This task of unhindered movement can be more easily tackled if we divide them into smaller tasks, where each task is handled in individual modules. Often these tasks are parallel in nature. For example, while Tribot is moving toward an object by means of following a line on the ground, it would also have to detect if the actual object has been reached, either by means of the touch bumper, or if the object is large (such as a wall or other obstacle), the use of the ultrasound sensor. These tasks are inherently parallel in nature. In addition, Tribot would often have to deal with unexpected situations while attempting to accomplish its goals. An obstacle may block the planned path, causing it to be unable to proceed by following the line on the ground. The object may be slippery and escape from Tribot’s grasp as it tries to grab it. Hence the Tribot should be able to handle such unexpected situations in its programming, to have a high level plan or objective, as well as lower level tasks needed to accomplish the objectives.

Behavior-Based Robotics is a robotic development paradigm which seeks to address these issues in a parallel yet simple to specify and develop manner, while being able to accomplish the objectives in an opportunistic
6.1.1 Definition of Behavior-Based Robotics

The goal of this book is on programming, so it is only possible to provide a general idea of the concepts behind Behavior-Based Robotics. More comprehensive treatment of the subject can be found in [33, 8, 34], with practical application of Behavior-Based Robotics in [9]. The premise of Behavior-Based Robotics is that all the tasks that need to be performed by the robot can be consolidated into a set of Behaviors which govern the operation of the robot. Behaviors obtain inputs via Sensing the environment, and acts on the environment via Actuations. Multiple competing Behaviors operate concurrently; higher level behaviors subsume or suppress (overrides) lower level behaviors. Consequently, a set of behaviors may each generate different competing actuations (output signals) for a given actuator device. However, a given behavior may not generate an output signal for a specific actuator, for example to turn a motor clockwise or counterclockwise. If a particular output signal is not generated by a behavior, that signal is inhibited. This is similar to an electronic bus circuit with multiple connected modules, where a given module output is placed in High Impedance (Z) state if it is not activated. This is summarized in Figure 6.1.

While behavioral modules can be combined in complex ways [8], a more structured fixed priority control scheme would be easier to implement and debug [9]. This can be realized using one or more Arbiters (Figure 6.2).
which separates the specification of Behaviors from the task of prioritizing the behaviors. The task of the Arbiter is to decide which Behavior should be active at any given time. Generally, a higher level behavior always has priority over a lower level behavior. Using the electronic bus circuitry analogy, the Arbiter controls output signal for which module has access to the bus, since only one signal can be active at any given time. In Behavior Based Robotics terminology, the higher level behavior with an active output signal will suppress (override) the lower level behavior output signal and gain access to the bus. The design of the Arbiter therefore governs the actual response of the given robot.

Consequently, a practical implementation of the Behavior-based Robotics Framework would look like Figure 6.2. In this Behavior-Based Robot Control Software example, the robot actuators control motors which determine the movement of the robot, whether to go straight, turn left or right, reverse, or stop. Sensors such as Photocells, Infra Red (IR) detectors, and Bump Detectors provide inputs that are processed concurrently by different behaviors. The behaviors which activate based on the received inputs will send their movement commands to the Arbiter. The Arbiter selects the winning Behavior (typically the one with the highest priority, which is shown in a module higher up in the list of Behaviors in Figure 6.2), and sends the appropriate command to the Motor Controller. We will explore how to implement the various components which make up the Behavior-Based Robot Control Architecture in greater detail starting from this Chapter.

6.1.2 Decomposing Tasks into Software Modules

From Figure 6.2, we can decompose the entire robot control software into Blocks which implement specific functions. Each of the blocks would then be implemented as a routine that performs a given action. Some blocks (e.g., Motor Controller) may consist of further subblocks which together implement the complete function shown in the figure. For example, individual behaviors may be implemented as routines such as Escape(), DarkPush(), and so on, while the Arbiter() and MotorController() are also routines (or a group of routines) which perform specific tasks. This decomposition process allows us to develop the functionality of the robot step-by-step, as well as enable us to test and debug small components of the robot software piece-by-piece.
In addition, various related routines are often grouped into its own source file as a module. Hence the routines that implement the Motor Controller functions could be one module, kept in its own source file. Modules may be written in Assembly or C, depending on the needs. In NxBOS, the OS modules are mostly written in C for portability and speed of development. In contrast, since we’re interested in Assembly Language programming, the user modules such as the Behavior modules that we’re developing in this book are written in Assembly. Consequently, it would be useful to understand how to call between C and Assembly language modules. The GNU Linker allows us to link the various modules (compiled or assembled to object files) into the final program regardless of the source language, as long as each module obeys the same AAPCS procedure call conventions [21].

Procedure calls require the passing of parameters from the calling routine to the called routine. Although up to four simple parameters such as 32-bit integer values may be placed inside designated parameter passing registers based on the AAPCS procedure call convention, additional parameters, as well as local variables used only within the called routine would need to be allocated storage locations in memory. This is often accomplished with the help of Stacks, where parameters and local variables are placed on the stack for access from within the called routine, while avoiding the use of fixed memory locations.

6.2 Stacks and Their Use

Stacks are important concepts in Computer programming. Most processor architectures provided hardware support for a memory-based stack, for storing temporary variables, for supporting nested routine calls, and for handling processor interrupts. The stack has a Last In First Out (LIFO) behavior. Any item that is stored in the stack can only be retrieved after all other items that were stored in the stack later were removed. Stacks use Inherent Addressing mode, i.e., the location of the top-most item does not need to be given as an operand.

6.2.1 Idealized Stacks

Typically, PUSH and POP instructions are provided for accessing the Stack, shown in Figure [6.3] to place an item on the stack and to remove the top-most item. However, sometimes it may be useful or necessary to access the contents of the stack without POPping the items one-by-one from the stack. While this violates the idealized design for a stack, it may be necessary if the number of registers is limited. For example, if a calling routine has to pass several parameters to a called routine via the stack, the called routine may have to access a parameter which is now an item somewhere in the middle of the stack for its use.

6.2.2 Memory-based Stack Implementations

The stack is actually implemented as an array of contiguous memory address locations defined somewhere in external memory. Stacks are refer-
enced using a special address pointer, called the Stack Pointer. The Stack Pointer can be implemented as either pointing to a Full Stack or Empty Stack. In addition, since items in the stack are stored in sequential memory locations, they can either be stored in Ascending Order or Descending Order. The AAPCS for the ARM architecture specifies the use of the Full Descending model (similar to most other processor architectures), although the CPU is capable of supporting all four combinations (See Chapter 4.2.5).

The Full Descending Stack adopts the convention where the stack starts from a high memory address (known as the Bottom of the Stack) and grows towards a lower memory address (where the Top of the Stack is located). The Stack Pointer will always point to the Topmost Item, or if there were no items in the stack, the Bottom of the Stack Address. The Bottom of the Stack address location is initialized at the beginning of the program and is not used to store any data.

Diagrammatically, this appears as if the stack has been inverted and anti-gravity forces are keeping the items in place. However, since actual stack implementations are only a realization of a logical concept, laws of gravity do not apply. The processor keeps track of access to the stack using the Stack Pointer. The Stack Pointer (SP) is automatically decremented when an item is added (pushed) to the stack, while it is automatically incremented when an item is removed from the stack. Programmers do not need to access the Stack Pointer directly except when it is initialized at the beginning of the application, and when manipulating items in the stack directly.

The Full Descending Stack is illustrated in Figure 6.4. The Bottom of the Stack was initialized to address 0x100000. Items are stored on the stack according to the endian-ness of the processor. For example, a 32-bit value stored into the stack would observe the processor endian-ness by storing the MSByte and LSByte appropriately based on the address location that
the item uses. There is a 32-bit word item previously PUSHed onto the stack during program execution, located at address 0xFFFFC (LSByte) to 0xFFFFF (MSByte), following the little-endian processor format.

In order to insert (push) a new item onto the stack, SP is first decremented, and then the item stored in the correct endian format starting at that address. Therefore, if SP = 0x100000 before a PUSH, PUSHing a 32-bit word onto the stack is accomplished via decrementing SP is by 4 (each address location stores one byte) to 0xFFFFC, and then storing the new value at the given address location. Hence, PUSH 0x31AF2BCD onto the stack would store CD\textsubscript{16} at the lower address 0xFFFFC, 2B\textsubscript{16} at the next higher address (0xFFFFD), AF\textsubscript{16} at the subsequent address (0xFFFFE), and 31\textsubscript{16} (MSByte) at the highest address (0xFFFFF).

Conversely, POP-ping the topmost item from the stack involves Loading the value into a register using the address in SP, and then incrementing SP by 4 to point to the next topmost item. The memory locations are not actually cleared until a subsequent PUSH that replaces the old contents with new values. POPs can continue until the Stack is empty. If another POP is performed on an empty stack, we get a Stack Underflow error. A Stack Overflow error occurs when the Top of the Stack exceeds the memory allocation for the stack array and is usually catastrophic since it overwrites potentially important data used by the operating system or application.

Accessing the topmost item in the stack without POP-ping it out of the stack is very easy, since SP already points to the appropriate address. However, we often need to access other items further down in the stack, namely, parameters passed from a calling routine to a subroutine. This can be accomplished by means of a pointer that is initialized to point to the contents in the stack. Typically a Frame Pointer (FP) is used, although the Stack Pointer can also be used as a pointer with some difficulty\footnote{Since the SP will change if new items are PUSHed into or POPped from the stack, we would need to keep careful track of the current SP value and update the required offset to access existing items in the stack.}. Other items deeper inside the stack can similarly be accessed by first using FP as the starting address pointer, and then adding the correct offset into the stack to obtain the item address value. This can then be used to Copy the given item.

Figure 6.4: Full Descending Stack for Little-endian Processor
from the stack into a register. The use of the Frame Pointer is discussed in greater detail in Section 6.4.4.

6.2.3 Sequence for Using PUSH and POP Instructions

One important note regarding stack usage is that the number of registers PUSHed into the stack and the number of registers POPped from the stack within a routine must be balanced. For example, if we PUSHed R0, R1, R2 onto the stack, then they must be removed in reverse order of R2, R1, R0 if we intend to return the correct values to their original registers. If this were not done, then stack items would be retrieved to a different register, changing its value to that in another register. Obviously, if we POPped something from the stack before PUSHing an item in, it would corrupt the stack. Fortunately this can be accomplished easily on the ARM architecture by means of the LDM and STM instructions, first discussed in Chapter 5.1.2.2. A single STMFD / LDMFD pair can save and restore all affected registers in the correct sequence. The equivalent PUSH and POP pseudo-instructions are provided as a convenience as well. This is illustrated in Listing 6.1 where Subroutine1 is balanced while Subroutine2 and Subroutine3 would result in unexpected behavior or program crashes.

Listing 6.1: Number of PUSHed and POPped Registers Must Balance

```c
; Balanced Stack Usage
Subroutine1:
PUSH {r4-r8}  ; Equivalent to STMFD sp!, {r4-r8}
...
PPOP {r4-r8}  ; Equivalent to LDMFD sp!, {r4-r8}

; Unbalanced Stack Usage
Subroutine2:
PUSH {r1}     ; Store one register in the stack
...
PPOP {r1,r2}  ; Retrieve two registers from the stack (!)

; Unbalanced Stack Usage
Subroutine3:
PUSH {r2,r3}  ; Store two registers in the stack
...
PPOP {r2}     ; Retrieve one register from the stack (!)

; ! Stack Underflow may occur if the last item in the stack was popped off
; !! Garbage left on the stack, prevents access to subroutine return address
```

6.3 Calling Routines

The use of routines for performing common tasks is the key to software development. This saves time development and testing, since a routine needs to be written and tested once, and then used as often as needed. Macros provide some of the advantages of routines, but as observed previously when we studied Macros in Chapter 4.3 routines are more code-efficient if the algorithm is complex and used often, for example, to calculate the checksum for an input value. The Standard C Library is an example of a well known collection of routines used in every C language program.
6.3.1 Leaf Routine Linking and Returns

Algorithms that are self-contained (i.e., does not call other routines), are known as leaf routines. Such routines can be called using a technique known as linking. Instead of storing the return address in the stack, we keep the address of the next instruction in a designated register, and use Unconditional Branch or Jump instructions to access the routine. At the end of the routine, we return to the next instruction in the calling routine via an Unconditional Indirect Branch or Jump using the address stored in the designated (link) register.

Some processor architectures (including the ARM) provide link instructions that would automatically store the return address in a designated register before jumping to the routine. Routine linking is especially useful for very fast interrupt servicing (such as reading data from the USB subsystem) since access to the stack is not required [20]. It is the programmer’s responsibility to ensure that the contents of the link register is preserved (if it is needed) before invoking the link instruction, since its value would be overwritten.

For the ARM processor, the Link Register is R14, while the Branch and Link instruction is BL. Therefore, when:

\[
\text{BL <label>}
\]

is executed, the address of the next instruction following the Branch and Link instruction is stored into the Link Register R14 (commonly referred to as LR).

Link Return is accomplished copying the contents of LR to the PC, this can be achieved via:

\[
\text{MOV PC, LR}
\]

in ARM mode (where PC is R15 and LR is R14). If Interworking is required, the Branch and Exchange instruction,

\[
\text{BX LR}
\]

which allows switching between ARM and Thumb states on return to the calling routine should be used instead. Interworking is covered in greater detail in Chapter 8.

Figure 6.5 illustrates the use of the Link Register in conjunction with the Branch and Link instruction to perform single level routine linking. As shown in the figure, linking to another routine from within the first routine will result in the loss of the original return address, making it impossible to return to the main routine. The use of routine linking must be done carefully to avoid such a scenario.

6.3.2 Non-leaf Routine Calls and Returns

In general, we may need to support multiple levels of routine calls, since a complex routine may depend on the services of other simpler, commonly used routines. Such routines are known as non-leaf routines. In order to
Figure 6.5: Leaf Routine Linking
support an arbitrary nesting of routine calls, we need to keep the return address for each routine call somewhere. The stack is the natural candidate for this role, since the return address would remain in the stack without being overwritten as long as we observe the correct PUSH-POP order.

Many architectures providing hardware support for stacks (e.g., via a dedicated Stack Pointer) implements Stack-based routine calls. In most processors with built-in stack support, Calling a routine will automatically PUSH the return address to the stack, while Returning from a routine will automatically POP the next instruction address from the stack. Therefore, we must make sure that any items placed onto the stack within the routine be removed prior to returning to the calling routine. Otherwise, the return address would not be at the top of the stack for the Return.

 Nonetheless, the ARM processor does not implement a hardware stack. Instead, the STM and LDM instructions are provided for performing appropriate return address manipulation on entry to, and on exit from the given routine. Since the most recent return address is stored in the Link Register (R14), the first step in a Non-leaf Routine is most frequently to PUSH the LR onto the stack, along with other registers that need to be preserved (which would be used for temporary storage within the given routine). At the end of the routine, the affected registers will be restored to their original values, along with the LR. This is illustrated in Listing 6.2.

Listing 6.2: Non-Leaf Routine Calls and Returns

```
@ Non-Leaf Routine calls
@ Subroutines
Subroutine1:
    PUSH (R4-R6, LR) @ Store registers that need preserving, LR
    ...
    BL Subroutine2 @ Call subsequent routine
    ...
    POP (R4-R6, LR) @ Retrieve registers, LR
    MOV PC, LR @ Return to calling routine via address in LR

Subroutine2:
    PUSH (R4-R6, LR) @ Store registers that need preserving, LR
    ...
    POP (R4-R6, PC) @ Retrieve registers, restore LR directly to PC

@ Main routine
Main:
    BL Subroutine1
    BL Subroutine2
Cont:
    ...
```

In the algorithm, Subroutine1 preserves R4-R6, and LR on the stack, and restores them just before returning to the calling routine by retrieving it from the stack, and finally, copying the return address found in LR to the PC to return execution to the main program. This ensures that registers modified within Subroutine1 would be restored to their original values expected by the main program, and allows for subsequent routine calls from within Subroutine1 (e.g., to Subroutine2), which updates the LR with the new return address of the called routine. It should be noted that the behavior of Subroutine2 on the return is slightly different and maybe a

---

2There is also the possibility of Stack Overflow, but it can be avoided by choosing a suitable maximum size for the stack if RAM is not limited.
little bit unexpected. Instead of restoring register LR with the original contents, the POP sequence retrieves the value in LR and places it directly into PC. This saves one instruction in terms of execution, while LR is left in a indeterminate state after the return to the calling routine. Generally it is not critical to restore LR to the original value since the calling routine Subroutine1 would have preserved LR on the stack before invoking Subroutine2. However, it would not be possible to do this when switching between ARM and Thumb modes on the return (See Chapter 8).

6.3.3 Recursion

Some algorithms are easier to implement if the Recursion technique were used. Recursion is the calling of a routine from within the routine itself, and is done repeatedly until a certain condition (termed the exit condition) is satisfied. A famous example of a recursive routine is the factorial() function written in C (Listing 6.3).

Listing 6.3: C code for Factorial routine

```c
unsigned int factorial(unsigned int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

The factorial() function can be rewritten in ARM Assembly language as shown in Listing 6.4. Note that the factorial() recursive function had to keep the temporary variable (n) on the stack to prevent it being overwritten in the subsequent routine call. Consequently, recursion can use up a lot of stack storage and may not be suitable if the number of temporary variables is large, or if the depth of recursion (number of times the function is called before it terminates) is great. Imagine trying to calculate 50!, assuming that we can represent it properly as a multi-word number. Hence, it is the programmer’s responsibility to ensure that Stack Overflow does not occur.

Listing 6.4: ARM Assembly Factorial routine

```
@ Factorial Routine in ARM Assembly @
@ @ unsigned int factorial(unsigned int n) { @
@     if (n == 0) @
@         return 1; @
@     else @
@         return n * factorial (n-1); @
@ } @
```

3 An optimization to the factorial routine could be performed if we note that 1! = 0! = 1. The algorithm could be rewritten such that the initial test becomes “n <= 1” instead of “n == 0” to eliminate one recursion step.

4 In Embedded Systems, where RAM sizes are limited and there are constraints in available Stack space, recursive algorithms may not be a good idea, especially if the depth of recursion cannot be controlled. It is often better to convert a recursive algorithm into an iterative one (using loops and temporary variables) where possible.
@ On Entry:
@ R0: integer n
@ On Exit:
@ R0: integer result

factorial:
PUSH {R4, LR} @ Use R4 to keep n
TEQ R0, #0 @ if (n == 0)
MOVEQ R0, #1 @ Return 1
BEQ exit_factorial
@ else
MOV R4, R0 @ R4 := n
SUB R0, R4, #1 @ R0 := (n-1)
BL factorial @ Call factorial (n-1)
MUL R0, R4, R0 @ R0 := n x factorial (n-1)
@ Result of multiplication in R0 (return value)
exit_factorial:
POP {R4, PC} @ Restore R4, retrieve LR to PC, exit

6.4 Parameter Passing Convention

When creating routines, it is important to observe the parameter passing convention. The AAPCS [21] specifies the register usage norms when calling routines and returning values to the caller. Generally simple parameters such as bytes, half-words, words and pointers are passed as values in registers R0-R3. The return value is stored in R0 before resuming execution of the calling routine. Further details are given in the AAPCS document.

Generally, parameter passing can be categorized into the following approaches:

- Memory-based Parameter Passing
- Register-based Parameter Passing
- Stack-based Parameter Passing
- Frame-based Parameter Passing Technique

6.4.1 Memory-based Parameter Passing

The Memory-based parameter passing technique use predefined memory address locations for storing global parameters that are needed by particular routines. This technique may be useful if the parameters do not fit in available registers, and a memory usage convention is adopted where all programs know about the existence of the relevant memory address locations and do not accidentally overwrite it. The biggest drawback of such memory-based parameter passing techniques is that it is very difficult to write recursive routines that use just the given memory locations to keep track of the different routine call levels. Worse, if the routine can be called by User as well as Supervisor programs (invoked via Traps or Interrupts), parameters will be overwritten. The use of Memory-based Parameter Passing technique is strongly discouraged since it is not easy to extend the routine for reuse in future. Such routines are termed non-reentrant.
Table 6.1: PUSH order for different stack usage conventions

<table>
<thead>
<tr>
<th>First-to-Last Order</th>
<th>Last-to-First Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH Arg1</td>
<td>PUSH Arg3</td>
</tr>
<tr>
<td>PUSH Arg2</td>
<td>PUSH Arg2</td>
</tr>
<tr>
<td>PUSH Arg3</td>
<td>PUSH Arg1</td>
</tr>
<tr>
<td>CALL routine</td>
<td>CALL routine</td>
</tr>
</tbody>
</table>

6.4.2 Register-based Parameter Passing

If the number of parameters are less than five, such as for the factorial() routine example, the register-based parameter passing technique is ideal. This allows routines that are called frequently to be accessed without additional memory access to store and retrieve parameters. However, this technique is only suitable if the parameter type is simple (e.g., integers), and registers are available for such use. For parameters that are too large to fit in a register, pointers to the data would be used instead of the actual values (this is termed Pass via Pointer or Pass via Reference). This enables us to pass complex data items such as structures and objects by referring to the actual data via a pointer value placed in the register.

In the AAPCS, registers R0-R3 are generally used for passing parameters to called routines, and their values do not need to be preserved before returning to the calling routine. However, if other registers were also used for parameter passing (not according to the AAPCS guidelines), then the values upon entry to the routine should be preserved by convention. This would avoid unnecessary problems with calling routines from other modules that were written in a higher level language in future.

6.4.3 Stack-based Parameter Passing

Stack-based parameter passing techniques provide generalized solution to the issue of parameter passing. Most high level languages use Stack-based techniques since the number of parameters and the type of parameters may differ significantly from routine to routine, and a Stack can accommodate the differences easily. In this technique, parameters to be passed to the routine are first pushed onto the stack, and the routine is called. From within the routine, parameters on the stack are accessed directly using a pointer into the stack. The stack usage convention determines how parameters are placed on the stack. Two types of convention exist: first-to-last order, and last-to-first order (Table 6.1).

For example, if a routine take three parameters Arg1, Arg2 and Arg3, first-to-last order would mean that we PUSH Arg1 onto the stack, then Arg2, and lastly Arg3. Last-to-first order does the opposite, where Arg3 is first PUSHed onto the stack, followed by Arg2 and then Arg1.

---

5C uses last-to-first order while Pascal which uses first-to-last order for parameter passing. What is the impact of this ordering on routines that accept variable number of parameters (e.g., C's printf() function)?
Prior to calling the routine, three parameters (Arg1, Arg2, and Arg3) were PUSHed onto the stack using the C calling convention. The Stack Pointer SP now points to 0x0FFFF0. On entry to the routine, the Return Address in LR is PUSHed onto the stack, updating SP to 0x0FFFFEC. Nonetheless, we can still use the Stack Pointer to access the parameters from within the routine. We can access the respective parameters since we know their offsets from the Stack Pointer. In the figure, Arg1 is at [SP, #4], Arg2 is at [SP, #8], while Arg3 is at [SP, #12]. This is necessary since we cannot POP the parameters off the stack, due to the presence of the Return Address.

It should be noted that it is generally not possible to pass the Return Value (RetVal) from the called routine back to the calling routine via the stack due to the presence of the Return Address at the top of the Stack. Return parameters in high level languages are usually simple types or pointers that are returned to the calling routine via a designated register. Therefore, stack-based parameter passing techniques are not symmetrical.

A related issue to the stack usage convention is the stack cleanup convention. Parameters that are placed on the stack before the routine call need to be cleaned up after the routine returns to the calling routine. By default, the calling routine is responsible for cleaning up the parameters that it placed on the stack. Therefore, the steps for Stack-based Parameter Passing are:

Place parameters on the Stack based on stack usage convention
Call Subroutine
Clean up parameters from the Stack
Return from Routine

In the AAPCS, the stack is used only when there are five or more param-
eters that need to be passed to the routine. The first four parameters are placed in R0-R3 according to the Register-based Parameter Passing Technique, while the fifth and/or higher parameter values are placed on the stack according to the Stack-based Parameter Passing Technique. This avoids the overhead of PUSHing parameters onto the stack if the number of parameters were small. In addition, the Return Value is put in R0 for return to the calling routine, so that the topmost item on the stack is the LR just before the routine returns.6

6.4.4 Frame-based Parameter Passing / Referencing

Since the ARM supports both leaf and non-leaf routines, where the LR is not PUSHed onto the stack in leaf routines, the Stack Pointer is not a consistent means of accessing the passed parameters since the offsets may change from one routine call to another. For Leaf Routines, the topmost parameter in the Stack, Arg4 would be at [SP], whereas for non-leaf routines, it is at [SP, #4] (Figure 6.6). While compilers can keep track of this discrepancy easily, human programmers might get easily confused. Here, a frame-based parameter passing technique, using an explicit Frame Pointer can be used to overcome this problem and ensure that parameters are accessed consistently regardless of how they were called.

By copying the Stack Pointer value to a register, we can use Indirect Addressing with offset modes to retrieve the required parameters from the stack. This pointer is called the Frame Pointer (FP). The GNU Assembler uses R11 as the FP. The offsets are fixed relative to the FP for the entire duration of execution of the routine, even if we subsequently place additional items onto the stack (which will move SP, but FP remains unchanged). This is illustrated in Figure 6.7 for Non-leaf routines, which PUSHes LR and previous FP onto the stack upon entry into the routine. The difference for Leaf routines is that only the previous FP would be PUSHed into the stack. The FP would the be adjusted to always point to the bottom-most item PUSHed into the stack within the routine, either LR for Non-Leaf routines, or Previous FP for Leaf Routines.

This convention is used by the GNU C compiler when the usage of the Frame Pointer is specified during code generation8. In the figure, the first four arguments, Arg1 to Arg4 are stored in R0-R3. The fifth argument is stored on the stack, and accessed via [FP+4], and subsequent arguments are located further down in the stack ([FP, #8], etc.) in Last-to-First order. The SP will be updated to point to other values during the course of execution of the routine, but FP remains unchanged. Local and temporary variables stored on the stack can also be referenced using FP and consistent offsets for the duration of the routine, enabling the programmer to follow the local variable references easily.

The setup for enabling FP for non-leaf routines is as follows:

---

6We can still use the stack to store local and temporary variables inside the body of the routine, but they must be removed from the stack prior to the return.

7In Figure 6.6, the topmost parameter in the stack is shown as Arg1 since the illustration is not AAPCS specific.

8By default, FP (R11) is used for code generation, unless '-fomit-frame-pointer' or '-Ox' (x is the optimization level) is specified.
Figure 6.7: Frame-based Parameter Passing for Non-Leaf C Routines

routine1:
    PUSH (FP, LR)
    ADD FP, SP, #4 @ Point to LR in Stack
    ...

For leaf routines, only FP is pushed onto the stack, so:

routine2:
    PUSH (FP)
    ADD FP, SP, #0 @ Point to FP in Stack
    ...

Actually, the ‘ADD FP, SP, #0’ can be written as ‘MOV FP, SP’ but the compiler generates the ADD instruction to be consistent. The constant added to FP should be adjusted if additional registers were pushed onto the stack to preserve their values.

6.5 C and Assembly Module Interfacing

Since the GNU Link allows us to combine modules written in different languages, it is often convenient to leverage previously developed libraries in our code. Our main interest here is in Assembly Language and C Interfacing. There are three ways in which code in C and Assembly can be mixed.

6.5.1 Inline Assembly in C

Assembly statements can be inserted into C programs by means of the `asm()` directive. This is described in the Assembly Language HOWTO
and further documented in the GCC Inline-Assembly HOWTO [36]. However, most of the online documentation provides examples only for x86 Assembly, so it requires some trial-and-error to get it working for ARM Assembly Language.

6.5.2 Calling Assembly from C

Calling Assembly Language routines from C requires that the Assembly Language routine be written according to AAPCS requirements, i.e., in terms of the parameter passing convention [21]. It is not important that the Assembly Language routine follows exactly the C register and stack usage convention within the routine, as long as it preserves the necessary registers (R4-R11, SP, LR), and places the return value in R0 before returning from the routine.

Nonetheless, it is important that a suitable routine prototype be declared in a header (.h) file and made available to the C program, and the routine entry point be declared as a `.global` label.

An example is given below:

```c
/* memcpy.h
 * Function Header for memcpy_arm()
 */
char *memcpy_arm(char *dest, char *src, int n);
```

The actual routine would then be written in its own .S file, shown in Listing 6.5. Note that in this example, `memcpy_arm()` does not perform input validation and bounds checking for simplicity. In addition, we need to use the `.type <func_name>, %function` directive to inform the linker that `memcpy_arm` is a function entry point.

```
Listing 6.5: ARM Assembly version of memcpy()

/* memcpy.S
 * ARM Assembly version of memcpy()
 * On Entry:
 * R0: dest Pointer
 * R1: src Pointer
 * R2: n (Assume n > 0)
 * On Exit:
 * R0: dest Pointer
 */

.type memcpy_arm, %function @ Needed by binutils to recognize function
.global memcpy_arm @ Needed for Linker to find the entry point
memcpy_arm:
PUSH {R0, LR} @ R0 must be returned to caller
1: LDRB R3, [R1], #1 @ Load char from src
   STRB R3, [R0], #1 @ Store char to dest
   SUBS R2, R2, #1 @ Decrement n
   BNE 1b
POP (R0, PC) @ Return to caller with original dest Pointer
```

To call the routine, we would invoke it as usual in our C program:


```c
#include "memcpy.h"
main() { 
    ...
    memcpy_arm(outbuf, inbuf, n);
    ...
}
```

Care should be taken that the Compiler Flags used would be compatible with the programming style used in the Assembly Language routine. For example, if the Assembly Language routine relies on FP to access stack parameters, then `-fomit-frame-pointer` must not be specified (explicitly or implicitly) for the C compilation.

### 6.5.3 Calling C from Assembly

Calling C routines from Assembly is even easier. All that is needed is for the C routine name to be referenced via an `extern` directive. The Assembly Language routine should observe the AAPCS and Parameter Passing conventions (Section 6.4.4) used for compiling the C module. One of the AAPCS requirements is to have 8-byte aligned stack where the Stack Pointer address is divisible by 8 upon entry to the called publicly accessible routine. This requires a bit more care when setting up arguments on the stack before calling the C function. Dummy values are PUSHed into the stack if necessary before PUSHing the actual arguments, to meet the 8-byte alignment requirement for SP, before invoking the routine via BL. This is illustrated in Listing 6.6.

#### Listing 6.6: Calling C Routine `sento()` from Assembly

```assembly
.data
.align 4
socket: .word 0x0
buffer: .space 256, 0x0
length: .word 256
flags: .word 0x0
dest_addr_ptr_ptr: .word 0x0
dest_len: .word 0x0

.code 32
.align 4
.extern sendto @ C socket library routine
% sendto(int socket, const void*buffer, size_t length, int flags,
% const struct sockaddr *dest_addr, socklen_t dest_len);

start:
% Assume SP already initialized and word aligned
% We assume that all arguments to sendto() are stored in
% memory variables; and flags, dest_addr_ptr_ptr and dest_len
% were initialized before coming to the code fragment shown.
% We use a variable to the store the dest_addr pointer for
% simplicity (details of struct sockaddr not required for this
% example).
%
% IP: Intra-Procedure-call Scratch Register (R12)
...

% Put Arg5 & Arg6 onto the stack
MOV IP, SP @ Use IP as scratch register for original SP
```

Typically, all C routines would be publicly accessible if they were declared in a header file.
In the routine, the original value of SP must be preserved before placing arguments onto the stack. The *Intra Procedure-call Scratch Register* IP (R12) is used as a temporary register for keeping the original SP value. However, since IP is not guaranteed to be kept intact after calling another routine, we must place it in the stack for safekeeping. Then, to make sure that the calling routine receives an 8-byte aligned stack, we reserve space in the stack for the arguments, and mask the value in SP to the nearest 8-byte boundary. This is accomplished using:

```
SUB SP, SP, #((<number of args more than 4> + 1) * 4)
BIC SP, SP, #0x07
```

The amount of space to reserve is given by the number of arguments more than 4, plus one for the original value of SP. For the given example in Listing 6.6 which calls `sendto()`, 3 slots were reserved, where `Arg5`, `Arg6`, and the original SP value were placed on the stack by storing it to [SP], [SP+4] and [SP+8] respectively. This ensures that the parameters are always at the top of the stack. Any unused stack space (up to 4 bytes) below these items is wasted, depending on the original stack alignment.

After the routine call, SP must be restored to its original value. This is accomplished by loading the original value of SP directly from the stack:

```
LDR SP, [SP, #(<number of args more than 4> * 4)]
```

since the original SP was the first item PUSHed into the stack.
6.6 Converting Algorithms into Routines

We have seen how a routine is written and how parameters are passed to the routine. The task of writing routines is slightly more involved than just adding a ‘MOV PC, LR’ at the end of an existing algorithm. The programmer needs to consider issues such as parameters and how they are accessed via the routine. For example, an algorithm written as a program may access memory locations directly. This is identical to the use of global variables in high level languages. Global variables in routines are not encouraged in any programming languages since it exposes a routine to unintended changes if the global variable is accessed by the program in other places. Consequently, all global memory location references in an algorithm must be converted to either register references (variables stored in registers) or stack based references (variables stored on the stack). These references are then initialized by the calling routine as necessary (Section 6.4).

Another issue in writing routines is the use of temporary registers. Registers may be used by the calling routine to keep intermediate values that are needed after the routine call. It is therefore important to preserve those register values that will be modified during the execution of the routine but are not used for returning parameters to the calling routine. Usually, this is accomplished by PUSHing the registers onto the Stack, and restoring them at the end of the algorithm just before the routine Return. This was illustrated in Listing 6.5. These two steps go a long way towards making routines relocatable and reentrant, two important criteria in modules and library development.

Finally, on the ARM Architecture, there is an additional feature to be considered: the use of ARM and Thumb routines in the same program, known as Interworking. The issue of making routines Interworking-safe will be covered in greater detail in Chapter 8.

6.7 Modules and Software Libraries

The development of routines that perform specific functions is the first step towards creating modules that contain groups of commonly used routines. Such modules can be written and tested to verify their correctness, and once completed, can be reused in other programs. This is the basis for creating software libraries that are used to build larger systems. The Standard C Library (libc) is an example of reusable software modules that are commonly used by all C programmers.

Assembly language-based modules which contain commonly used routines can also be developed in a similar manner. In fact, if a routine interface (parameter passing convention) is well defined, it is even possible to provide different versions of the same routine for supporting different hardware devices (e.g., send display output to a LCD panel vs. to a video screen).

Several characteristics of good modules or software libraries are:

- Cohesion: The routines in a module or library should address a common functional areas, such as I/O processing, string manipulation, etc.
It is not a good idea to mix unrelated routines together in a module. A high level of cohesion makes a module useful and complete.

- **Coupling**: A module should not depend on other modules if possible, to avoid having to include several modules just to access a given function. A low level of coupling between modules is ideal as it allows the programmer to reduce the size of the program code.

- **Consistency**: All routines in a module (or a library) should use the same Parameter Passing Convention. E.g., standardized registers to pass and return parameters. A high level of consistency is necessary to avoid programming errors.

- **Relocatability**: Modules or routines should not hard code address locations such that it is not possible to reassemble the code at another memory address. Modules that are totally relocatable are preferred over those that can only assemble at specific memory address locations.

- **Reentrancy**: If the routine is called by a user program as well as an interrupt routine, the variables used by the routine should not be accidentally overwritten. Reentrancy is vital for modules that are used in pre-emptive scheduling systems or interrupt service routines.

### 6.8 Decision Making Basics

In Chapter [5.8](#), the Line Follower Robot had to determine whether it was detecting a line, edge, or something else, depending on the values of the Light Sensor input. Different actions were taken based on the values of the inputs, where different sequence of instructions would be executed depending on whether the robot had to travel forward, or turn on the spot to reacquire the line to follow. Conditional Statements (also known as Flow Control Statements) are provided in high level languages to support such decision making. In C, two distinct approaches for implementing conditional statements are provided:

- **If...Then...Else statements**

- **Switch / Case statements**

These conditional statements often have to be implemented efficiently, especially if the series of conditional checks and actions to be taken are complex. Here we will examine the first construct using If...Then...Else statements to see how it can be implemented effectively in ARM Assembly Language.
6.8.1 Implementing If...Then...Else Statements

Implementing If...Then...Else statements involves evaluating conditional expressions, and based on the outcome, perform one series of action instead of another. The general form of the If...Then...Else statement is as follows:

```
If (operand1 <condition> operand2) Then
  If-clause(s)
Else
  Else-clause(s)
Endif
...```

This can be implemented directly in ARM Assembly as follows using pseudo-labels IF, ELSE, THEN, ENDIF for illustration:

```
IF:
  CMP operand1, operand2
  B<condition> THEN
ELSE:
  Else-clause(s)
  B ENDIF
THEN:
  If-clause(s)
ENDIF:
...```

The evaluation of the If expression 'operand1 <condition> operand2' is achieved using suitable Comparison instructions (e.g., CMP, CMN, TEQ, TST). This will update the Flags in the Status Register and enable the conditional branch to be taken (to Branch to THEN) if the expression is True, or else to fall through to the ELSE block to execute the Else-clause(s). There is an unconditional branch statement 'B ENDIF' at the end of the ELSE block that is needed to bypass the THEN block statements that would not be relevant if the expression were False. More complex expressions can similarly be evaluated with appropriate combinations of Comparison instructions and Conditional Branch instructions.

6.8.2 Optimizing Simple If...Then...Else Clauses

Nonetheless, simple If-clause(s) and Else-clause(s) could be implemented in a more efficient manner using Conditional Execution (Chapter 5.4.3) in the ARM processor. In C, we often encounter the following single line If-Then-Else assignment statement written using the ternary operator ‘?’:

```
result = (operand1 <condition> operand2) ? If-val : Else-val
```

This could be easily written using ARM Assembly as follows:

```
result = (operand1 <condition> operand2) ? If-val : Else-val
```

\[10\] Label naming requirements (Chapter 4.1.3) must be observed in normal programs.
where Not-condition is the complementary state of the specified Condition, and result is stored in a register. For example, if Condition were Equal, then Not-condition is Not-Equal. By means of the conditional execution capability of the ARM instruction set, two branch instructions are eliminated, while only the instruction matching the result of the comparison would be executed. This technique avoids pipeline flushing, and takes constant time to evaluate any expression (i.e., the number of cycles to process the IF block is the same as the number of cycles to process the ELSE block since the non-matching conditional execution instructions would be treated as NOPs.

The disadvantage is that Conditional Execution takes longer than executing only the relevant instruction block if the number of instructions in each block is large. In addition, Conditional Execution would be very difficult to implement if there are nested conditional statements (additional If...Then...Else embedded inside a given block), since any Comparison conditional execution instructions would alter the Flags, affecting the execution of subsequent conditional execution instructions. Finally, conditional execution for most instructions is not available in Thumb state for the ARMv4T architecture (Chapter 8), so it is not a universal solution.

6.9 Grasper Robot

We return to Tribot, to understand more of how Behavior-Based Robotics work in a simple example. We have already discuss a robot which implements Follow-Line Behavior in Chapter 5.8, although at that time we didn’t know about Behavior-Based Robotics yet. Tribot is equipped with a set of claws which can be used to grasp an object, and move it from one location to another. There is a Bumper-bar attached to a Touch Sensor which is located at the back of the claw cavity that would be activated when the object to be grasped comes into contact with it. Consequently, when the Bumper-bar gets pushed against an object placed along its path, the Touch Sensor is activated, and we can then direct the motor controlling the Claws to close around the object and grasp it. Let’s call this version of Tribot the Grasper.

6.9.1 Behavior-Based Robotics Framework for Grasper

Grasper will need to perform the following task: follow a line on the ground until it detects an object to be grasped, at which point it would stop, grasp the object in its claws, and stay in the given location. We already have the Follow-Line Behavior implemented previously for the Line Follower. To implement Grasper, we will add three more behaviors, Idle, Open-Claws and Grasp-Object, to the robot. Idle is the lowest priority behavior, which

11Assuming that all affected instructions are single cycle execution instructions.
12Thumb-2 introduces the IT instruction to support C ternary operator style coding.
13The 9797-Edu kit based Tribot has the bumper-bar design. The Retail version NXT Tribot has only one touch sensor.
causes the Tribot to remain in the current location. *Open-Claws* will open the Tribot’s claws when no object is sensed by the Touch Sensor but the claws are in the closed position (which can happen if the object slips out of the claws), while *Grasp-Object* will cause the Tribot’s claws to close around the object when the Touch Sensor is activated.

A simple *Behavior Arbiter* is used by Grasper to determine which of the behaviors has precedence. The *Behavior Arbiter* will determine which behavior is activated and has the highest priority. The activated behavior would then determine the appropriate actuator outputs to be forwarded to the *Controller*, which is responsible for programming the motor output values. There may be additional behavioral states (e.g., Claw Movement, Robot State) which are kept by various behaviors to indicate currently active behaviors which should not be interrupted. The Behavior-Based Control Architecture for Grasper is shown in Figure 6.8. In the figure, the Controller is show as separate submodules to indicate that they control actuators with different functions, although it is implemented as one module in the program. The *Controller* outputs is summarized in Table 6.2.

To determine whether a behavior is activated or otherwise, the *trigger condition* for a given behavior is used. In the BBR paradigm, the lowest priority behavior, *Idle* is always active. All *Idle* does is to stay in the given location by stopping movement and braking the motors. *Follow-Line* will

---

**Figure 6.8: Behavior Diagram for Grasper**

**Table 6.2: Controller Outputs for Individual Grasper Behaviors**

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Claw Controller</th>
<th>Motor Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasp-Object</td>
<td>Close Claws</td>
<td>Stop Movement</td>
</tr>
<tr>
<td>Open-Claws</td>
<td>Open Claws</td>
<td>Stop Movement</td>
</tr>
<tr>
<td>Follow-Line</td>
<td>(Inhibited)</td>
<td>Line-following</td>
</tr>
<tr>
<td>Idle</td>
<td>(Inhibited)</td>
<td>Stop Movement</td>
</tr>
</tbody>
</table>
Table 6.3: Activation Triggers for Grasper Behaviors

<table>
<thead>
<tr>
<th>Activation \ Trigger</th>
<th>Touch Inactive</th>
<th>Touch Active</th>
<th>Claws Open</th>
<th>Claws Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasp-Object</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>&amp;&amp;</td>
<td>-</td>
</tr>
<tr>
<td>Open-Claws</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>-</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Follow-Line</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Idle</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Legend: '-' = N/A, '&&' = AND, '||' = OR

be activated and move the wheels only when the Touch Sensor is Inactive. Follow-Line does not consider the state of the claws (i.e., it will be activated whether the claws are open or closed). When the robot touches something, the trigger conditions for Follow-Line will become invalid and Follow Line will no longer be activated. If the Touch Sensor is Inactive, then Open-Claws is activated if the claws were closed. However, if the Touch Sensor were Active, then Grasp-Object is activated if the claw were open. Grasp-Object is given the highest priority so that it can suppress the Idle, Follow-Line and Open-Claws behaviors when an object is grasped. This is summarized in Table 6.3

### 6.9.2 Implementing the Grasper Arbiter

In the Behavior Based Robotic Architecture, the output of a behavior module is a signal sent from the behavior module to the Arbiter (Figure 6.8). While it is possible to send these signals as messages, it is not practical since the NXT has limited RAM for storing variables and data. Consequently, signals typically consist only of the values needed by the actuators to perform a given action. In the original Behavior Based Robotics Architecture, the behavior modules operate concurrently, and their signal outputs are also concurrent, due to the electronic circuitry-based paradigm used in formulating the architecture. When implementing this on a single core microcontroller, some adaptation of the architecture is needed. If concurrent processes are supported by the operating system, then multiple concurrent processes or threads can be used to approximate the concurrent behavior of the BBR behavior modules. Nonetheless, such an approach requires significant computation power and memory in the microcontroller, which constraints the type of robots that can implement the BBR architecture. Since the BBR Architecture can be implemented even on 8-bit microcontrollers [9], a different approach would be necessary.

First, we assume that the Arbiter implements a Fixed Priority Hierarchical arbitration scheme, where only one defined behavior is active at any given time, and an active higher priority behavior always overrides lower priority behaviors. While this is not the only possible arbitration scheme (see Chapter 7.1), it simplifies the Arbiter algorithm tremendously.

Second, we need to determine the type of concurrency (also known as multitasking) support provided by the operating system. The two common process concurrency models are Pre-emptive Multitasking and Cooperative Multitasking. The difference lie in the way that the operating system man-
ages the switch from executing one process to another process that is active simultaneously. Pre-emptive Multitasking support is found in most modern operating systems for desktop computers, whereas Cooperative Multitasking, which used to be common in desktop computer operating systems, is now predominantly found in resource constrained systems (such as those used for embedded applications), and perhaps less intuitively, in systems which require real-time response to events.

The reason that real-time systems generally implement Cooperative Multitasking is because Pre-emptive Multitasking uses a time quantum to schedule process execution duration. This can cause a high priority process to be switched out of the schedule even though the processing has not completed yet. In a real-time system, this introduces jitter into the response time, and can jeopardize the correct function of a hard real-time system. In contrast, Cooperative Multitasking relies on each process to yield control of the processor once it has completed the processing steps needed. Processes will only yield the processor after it has completed the critical steps required to respond to a given event. Consequently, the response time to events is fully under the control of each process, but require the adoption of process cooperation specific programming paradigms in order to develop such systems. To illustrate this, we will consider two alternative algorithms for the Fixed Priority Hierarchical Arbiter used by Grasper.

6.9.2.1 Preemptive Multitasking Arbiter

The Preemptive Multitasking Arbiter algorithm pseudocode is as follows:14

```plaintext
Preemptive Arbiter:
Co-Begin

Behavior Grasp-Object:
    If (Touch Active && Claws Open), then
        Activate Close Claws
    Suppress lower priority behaviors

Behavior Open-Claws:
    If (Touch Inactive && Claws Closed), then
    Activate Open Claws
    Suppress lower priority behaviors

Behavior Follow-Line:
    If (Touch Inactive), then
    Check Light Input Level
    If (Black_MIN <= Level <= Black_MAX) (i.e., Line), then
        Activate Go straight (Left and Right Wheel Rotate Fast)
    ElseIf (Edge_MIN <= Level <= Edge_MAX) (i.e., Edge), then
        Activate Rotate CCW slow (Left Coast, Right Rotate Slow)
    Elseif (White_MIN <= Level <= White_MAX) (i.e., Outside), then
        Activate Rotate CW fast (Left Rotate Fast, Right Coast)
    Else (i.e., Unknown)
        Inhibit Motors
    Activate Follow Line
    Suppress lower priority behaviors
```

14The algorithm presented here is simplified; additional logic to check in-progress behavior actions (e.g., Claw Movement, Robot State) is not shown.
Behavior Idle:
Stop Motors and Brake
Co-End

For the Preemptive Arbiter, although the behaviors are listed based on priority, each behavior is executed in parallel; each behavior needs to suppress the lower priority behaviors explicitly.

### 6.9.2.2 Cooperative Multitasking Arbiter

In contrast, the **Cooperative Multitasking Arbiter** algorithm pseudocode looks like this:

```plaintext
Cooperative Arbiter:
If (Touch Active && Claws Open) (i.e., Grasp-Object), then
    Activate Close Claws
Elseif (Touch Inactive && Claws Closed) (i.e., Open-Claws), then
    Activate Open Claws
Elseif (Touch Inactive) (i.e., Follow-Line), then
    Check Light Input Level
    If (Black_MIN <= Level <= Black_MAX) (i.e., Line), then
        Activate Go straight (Left and Right Wheel Rotate Fast)
    Elseif (Edge_MIN <= Level <= Edge_MAX) (i.e., Edge), then
        Activate Rotate CCW slow (Left Coast, Right Rotate Slow)
    Elseif (White_MIN <= Level <= White_MAX) (i.e., Outside), then
        Activate Rotate CW fast (Left Rotate Fast, Right Coast)
    Else (i.e., Unknown)
        Inhibit Motors
        Activate Follow Line
Else (i.e., Idle)
    Stop Motors and Brake
```

This algorithm has implicit suppression of lower priority behavior, where lower priority behaviors will not be activated if a higher priority behavior is triggered. In addition, the algorithm adopts what is known as a lazy evaluation approach. This means that once an active behavior is found, the lower priority behaviors will not be evaluated due to the 'If...Then...Elseif...Else' construct used by the Fixed Priority Cooperative Multitasking Arbiter.

It should also be noted that behavior evaluation should be done quickly, to reduce the overhead of arbitration and allow for quick response to changes in inputs and trigger conditions. Consequently, it is not advisable to access the actuators and controllers directly in the Arbiter, but to perform the actuation in a separate Controller module.

139
6.9.3 Implementing the Grasper Controller

Generally the Controller managed the various actuators, motors and other output devices for the robot. Typically actuator control is a continuous operation that takes an extended amount of time, such as moving the wheels in a line, closing the claw until it fully grasps an object, as well as opening the claw until it is fully open. If a behavior requires a particular operation, executing that operation to completion in an uninterrupted manner would block the evaluation and execution of other behaviors and operations in a cooperative multitasking operating system. Consequently we need to adopt a staggered processing approach.

In the staggered processing approach, the system typically executes a main loop which iterates through the respective tasks needed by the system within a predefined time interval for each pass. For a behavior-based robot, this would include Sensor Readings, Arbitration, and Controller programming of actuators. If there is excess computation time available for a given pass, the microcontroller may enter a sleep mode where it tries to conserve energy by entering a lower power state. It then reiterates the various steps in a new loop until the robot is commanded to shut down. The NxOS-Armdebug Robotic Framework (NARF) supports staggered processing in its main program loop as shown in Figure 6.9.

To support staggered processing, which is a form of cooperative multitasking, each behavior that is invoked periodically must maintain its own state variables, and save the current state of the behavior, before yielding control to the system. Consequently, behavioral tasks which executes over a long duration with respect to the interval for a given pass through the main loop, such as Follow-Line, Open-Claws, and Grasp-Object, would need to keep track of appropriate state variables (Robot State and Claw Movement), in order for it to continue execution of an active behavioral task during the next pass of the main loop. Work is done in chunks, where each chunk would lead progressively towards completing the given task.

6.9.3.1 Staggered Processing Example in Grasper

As an example, assume that the Open-Claws behavior has been activated and would cause the claw actuator motor to start turning such that the claws open. However, the process of opening the claw cannot be completed in one pass of the main loop. Hence the Open-Claws behavior would need to store the current direction of movement determined by the currently active behavior in the Claw Movement state variable, as well as the position of the claw actuator motor provided by the motor tachometer, and instruct the Controller to unlock the claw motor (by disabling motor braking, but the motor remains stopped) by updating the desired actuator settings for the claw motor. The Open-Claws behavior then yields the processor to the system to perform other tasks.

The Motor Controller is then invoked by the System, which retrieves the latest actuator settings set by the Open-Claws behavior. The Motor

\[\text{For the AT91SAM7S256 microcontroller, low power states are not available except by reprogramming the system clock to a lower clock frequency. We shall just wait for the end of the main loop duration instead.}\]
Figure 6.9: NxOS-Armdebug Robotic Framework (NARF) Flowchart
Controller compares the latest actuator settings with the previous actuator settings for the claw motor, and determines if a new actuator command need to be issued to the motor. If there were no changes, the motor does not need to be reprogrammed.

Once the claw motors have been serviced, the Motor Controller yields control to the System. Eventually control will be passed to the Open-Claws behavior again. In the new pass, the behavior determined that the Claw Movement state variable still indicates that the robot is opening the claws, hence the behavior remains active. However, since the motor has not started moving, Open-Claws will instruct the Controller to start opening the claw motor using a non-zero motor speed and yields control to the System again.

On subsequent passes of the behavior arbitration loop, the Open-Claws behavior remains active, and will monitor the position of the claw actuator motor to check that movement is progressing. Once it detects that the claw motor position has not changed from the previously stored position, the Open-Claws behavior will issue a new actuator command to stop and lock the claw actuator motor in the current position. The Claw Movement state variable is then updated to indicate that the movement has stopped, and hence, during the next pass through the behavior arbitration loop, the Open-Claws behavior will no longer be activated. It then yields control, and eventually the Motor Controller routine is executed. The new claw actuator settings differ from the previously stored actuator settings, hence the Motor Controller programs the claw actuator to stop and brake, locking the claws in the current position.

The Open-Claws Behavior undergoes the following state transitions:

\[ \text{Claw Unlocked} \rightarrow \text{Claw Opening} \rightarrow \text{Claw Stopped} \rightarrow \text{Claw Locked} \]

These correspond to the staggered processing steps (or chunks) referred to previously. The staggered processing and state variable evaluation process is illustrated in Figure 6.10.

### 6.9.3.2 Data Structures for Grasper Controller

The actuators used by Grasper are the Motor Actuators for the Left and Right Wheels and the Claw. Since there are three Motor Control Ports on the NXT, labeled Port A, Port B, and Port C, we will give each motor control variable in the `actuator_state` struct generic names so that we can reuse it for other robots.

The motor actuation speed values are in the range of \([-100,100]\), where negative values indicate reverse motion, while positive values indicate forward motion. Essentially this translates into clockwise and counterclockwise motion of the motor. A zero value means that the motor is stopped. In addition, we define `Inhibited` signals as those outputs that are not relevant to a given behavior. Since the motor actuation values are stored in a signed-byte variable, we can use the signed value of -128 (0x80) to indicate an Inhibited Motor Actuation value. However, when the signed byte value is loaded into a 32-bit register, we will need to sign-extend the value to keep it as a negative value. Hence when checking whether the variable indicates an Inhibited value or otherwise, we would compare it against 0xFFFFFFFF80.

Associated with the motor actuation is whether we want to apply braking when the motor is stopped. This is also recorded in the `actuator_state`
Figure 6.10: Example of Staggered Processing Steps in Grasper
This can be represented in ARM Assembly as follows:

```assembly
/* Signal Inhibition Values */
#define MOTOR_INHIBITED 0xFFFFFF80
#define MOTOR_INHIBITED_BYTE (MOTOR_INHIBITED & 0xFF)
/* For data initialization use */
/* Current Actuator Settings */
actuator_speedA: .byte MOTOR_INHIBITED_BYTE
actuator_speedB: .byte MOTOR_INHIBITED_BYTE
actuator_speedC: .byte MOTOR_INHIBITED_BYTE
actuator_brakeA: .byte FALSE
actuator_brakeB: .byte FALSE
actuator_brakeC: .byte FALSE
/* Previous Actuator Settings */
actuator_oldspeedA: .byte MOTOR_INHIBITED_BYTE
actuator_oldspeedB: .byte MOTOR_INHIBITED_BYTE
actuator_oldspeedC: .byte MOTOR_INHIBITED_BYTE
actuator_oldbrakeA: .byte FALSE
actuator_oldbrakeB: .byte FALSE
actuator_oldbrakeC: .byte FALSE
```

6.9.4 Software Modules for Grasper

The project directory for Grasper contains the various modules and routines which control the robot’s behavior. Generally, the modules can be divided into the following categories:

- General Algorithms (min-max routine)
- System Housekeeping (program initialization and shutdown)
- Common Robot Operation (initialization, shutdown, sleep, main loop)
- Behavior Based Robotics Framework (Arbiter, Controller)
- Worker routines (actuator primitives, sensor primitives, sensor input routines, state display routines, specific behavior routines)

These modules are organized into a basic robotic software structure based on Figure 6.9, which can be applied to any of the NxOS controlled robots used in this book. The source code for Grasper can be found in the NxOS-Armdebug repository, under the system subdirectory, as tribot/grasper.

6.9.4.1 General Algorithms Module

The modules referred to as General Algorithms Modules are routine which implement algorithms that can be used in various processing tasks. They are generally written in a way to enable reuse in various software applications. One example is the min-max routine used to determine the minimum and maximum values found in an array, used by the line following algorithm. The version of the mix-max routine for Grasper has been rewritten to support the processing of user defined arrays by specifying appropriate routine input parameters.
6.9.4.2 System Housekeeping Module

Program Initialization and Shutdown is an essential part of any NxOS program. The task of nx_proginit() includes clearing the LCD Display, playing tone sequences, and installing a Watchdog routine which allows us to shut off the robot even if the actual program is misbehaving. The nx_progshutdown() plays a different tone sequence to signal the termination of the program, and turns the power off to the robot. These routines are defined in the NxOS base library folder. Since these routines will be used by all the NxOS examples in this book, they are incorporated into the default program template armskel.S.

6.9.4.3 Common Robot Operation Module

The Common Robot Operation module includes the init_robot() routine for specific initialization of the sensor and devices attached to the robot, as well as turning off the motors and sensors when the robot is being shut down via shutdown_robot(). An essential part of the robot software is the Main Loop, found in the main() routine of the source file, which is responsible for handling all relevant inputs and outputs while the robot is operational. The main loop is executed repeatedly, until either the robot is shut down manually, or the execution duration exceeds the number of loops, which then triggers an automatic shutdown of the robot. The main loop is paced using the sleep_robot() routine, which limits the execution of the main loop to execute once every ROBOT_SCHED_DURATION. This is a constant defined at the beginning of the main source file, which controls the frequency of execution of the robot main loop. By choosing a smaller or larger number for ROBOT_SCHED_DURATION, the response time of the robot will be changed. However, it should be noted that there is a limit to how fast each sensor or motor can accept new commands, hence the scheduling duration should not be reduced arbitrarily. In addition, a certain amount of time would be needed for the various worker routines to be executed, hence the ROBOT_SCHED_DURATION must accommodate the worst case processing time for a single main loop.

6.9.4.4 Behavior Based Robotics Framework Module

The Behavior Based Robotics Framework implements grasper_arbiter() which is used to dispatch the various defined behaviors for the robot, and the grasper_controller() which is responsible for programming the various actuators used by the robot. Since the Arbiter has a fixed algorithm structure, it should be easy to plug in new behaviors where required. Similarly, the controller should be able to handle all the relevant actuators of the robot.

6.9.4.5 Worker Routines Module

The modules defined as worker routines are all the implementation specific routines for handling sensors, actuators, inputs, outputs, and behaviors. Obviously these routines are specific to the robot and need to be customized to meet the specified requirements of the robot.
6.10 Chapter Summary

- The Behavior-Based Robotics Framework approach maps well to Software Development using Routines and Modules.
- Routines are fundamental building blocks for complex program development.
- Parameter passing from a calling routine to a routine must be done using a standardized convention to avoid confusion. The AAPCS is the defined standard for ARM processors.
- A group of related routines can be put together to create a standard library.
- Cooperative Multitasking is often used in real-time systems to ensure deterministic response times.
- A Staggered Processing approach is necessary in a Cooperative Multitasking system.
- Grasper is an example of a robot based on the Behavior-Based Robotics Framework.

6.11 Review Questions and Problems

1. The most common stack design specifies the use of Full Descending Stacks. Investigate the issues encountered when other stack designs such as Empty Ascending Stacks are used, and which of these variants, if any, can be implemented effectively for software running on the ARM processor architecture.

2. What is the impact on stack usage using the recursive factorial() routine if we want to obtain the largest factorial result that can fit in a 32-bit word?

3. Rewrite the factorial() function using an iterative loop instead of the recursive implementation.

4. The behavior routines in Grasper implement basic functionality, but is not very robust when faced with unexpected situations. For example, the Follow-Line behavior will be stuck if no line markings were found within its radius of rotation. Implement a more advanced Follow-Line behavior which will attempt to break out of this situation.
Chapter 7

Decision, Decisions

“We must go back to Oz, and claim his promise.”
“Yes,” said the Woodman, “at last I shall get my heart.”
“And I shall get my brains,” added the Scarecrow joyfully.
“And I shall get my courage,” said the Lion thoughtfully.
“And I shall go back to Kansas,” cried Dorothy,


TRIBOT would need to process many inputs in order to achieve a goal such as locating an object to be picked up, and then moving it towards a user. One source of input was the light sensor, which was used to follow a line drawn on the ground to get to the object to be picked up. After it has successfully picked up the object, it would then need to move towards the user, who will indicate his or her location by means of sound (for example, by clapping). Finally, something that is instinctively easy to humans as avoiding obstacles is actually quite challenging to Tribot, since the robot can only respond to limited range and accuracy of the ultrasound sensor to detect objects or walls blocking its path. We will look again at the problem of obstacle avoidance in the Chapter 8. In order to successfully achieve all its goals, it would have to make many decisions based on the inputs it received. Decision making in Assembly language involves the use of suitable programming constructions, as well as suitable data structures to encode the data for making decisions. Consequently we need to how to access the relevant behavior states, arbitrate among active behaviors of different priorities, as well as determine the action to be taken based on the combination of inputs received. This will require an understanding of Pointers, Data Access, Data Initialization, Pattern Matching, Data Manipulation and Look Up Tables (LUT), in order to implement the required programming constructs, as well as to apply the programming constructs in the decision making process.

7.1 Arbitration Schemes

The discussion of Behavior-Based Robotics so far has focused on Fixed Priority Arbitration schemes [9]. This means that if enabled, a higher level
behavior will always completely override a lower level behavior. We will examine in greater detail how we can refine the implementation of the Fixed Priority Arbitration scheme later on in this chapter. Meanwhile, we will look at another approach that has been used in Behavior-Based Robotics.

7.1.1 Subsumption Architecture

The original Subsumption architecture [34, 8] defined Behavioral Modules that are controlled using Inhibition and Suppression Mechanisms, as seen in Figure 7.1. The Inhibitor blocks the output of a behavioral module from being sent to the Actuators, whereas the Suppressor overrides an existing active message and replaces it with a higher priority message [8]. Behavior modules can also generate messages to affect the operation of another Behavior module. A high level Behavior can affect the operation of a low level Behavior by inhibiting the output of the low level Behavior. For example, the Grasper Robot’s Grasp Object Behavior can inhibit the Follow Line Behavior, which would then cause the Controller to not to receive any output values from the Follow Line Behavior. By default, this would stop the robot wheels from turning. The high level Behavior output message is called the Inhibitor, while the mechanism which overrides the low level Behavior output message is handled by the Inhibitor Node.

Conversely, Inputs are Messages sent by Sensors or Behaviors, which acts upon a given Behavior to change its outputs. In the Subsumption architecture, a Suppressor input is an input message which will replace the original Input message. This is mediated by means of Suppressor Nodes. For example, instead of using the Arbiter to suppress the actuator outputs from a lower priority behavior, another way for the Grasper Robot’s Grasp Object Behavior to override the normal Follow Line Behavior is to replace the Light Sensor Input readings with new readings which would not be classified as being on a line, edge, or off the line (i.e., Invalid readings). This will cause the Follow Line Behavior to generate outputs for the Controller based on readings different from the actual sensor values, causing the wheels to stop moving. The diagrammatic representation of Suppressor and Inhibitor nodes is given in Figure 7.1.

7.1.2 Fixed Priority Hierarchical Arbitration

In the Behavior-Based Robotics Architecture presented in [9], a simplified form of the Subsumption Architecture with fixed priority controls was presented instead (Figure 6.2). Conceptually, the behaviors are always active; it is the role of the Arbiter to mediate among the various outputs of each behavior and determine the actuator outputs to be sent to the respective controllers. However, since there is effectively a fixed priority for the hierarchy of behaviors, lazy evaluation of behaviors can be performed, where

---

1So far we have only studied strictly parallel behaviors where the output of all behaviors feed directly into the Arbiter. Note that Suppressor and Inhibitor nodes can be cascaded where the output of one behavior becomes the input of another behavior. In the general case, Suppressor node are not restricted to suppressing input signals from sensors only, nor are Inhibitor nodes restricted to inhibiting output signals of behaviors only. Both types of nodes can be used to affect various inputs or outputs depending on the actual robot architecture requirements.
the highest priority active behavior would be determined and the rest ignored. The *Fixed Priority Hierarchical Arbiter* can be represented using the Suppression and Inhibition notation as shown in Figure 7.2. For the case of the *Fixed Priority Hierarchical Arbiter*, the use of Suppression is sufficient.

### 7.1.3 Managing Data for Behavior Arbitration

Since the behavioral states of each behavior consists of several related but non-uniform items of data, the most suitable data structure for managing the behavior actuations is a structure (known as a *struct* in C), while the actuator state outputs of all the behaviors of the robot are identical for a given actuator such as the motor, and hence can be most efficiently kept in an array data structure. Consequently, we need to study how data structures such as arrays and structs are accessed in ARM Assembly Language. This requires an understanding of how variables are kept in memory, and how pointers are used to access respective data locations.

### 7.2 Pointers and Data Structure Access

Pointers are important for data access and manipulation. External memory is often viewed as a *one-dimensional array* containing byte-sized items.
Consequently, the memory address location is simply the index into the memory array, assuming that the processor does not have a MMU and there is a direct one-to-one mapping between logical and physical memory. However, often we work with array of word-sized elements, or array of structures containing many bytes. High-level languages even provide multi-dimensional arrays, such as density[x][y][z], that may represent the concentration of some item in a geographical terrain. However, accessing these high-level language constructs in Assembly poses some additional challenges, since the processor does not have the concept of structures, let alone multi-dimensional arrays. Therefore, understanding Pointer Arithmetic and memory access is vital for algorithm development.

Pointer Arithmetic makes extensive use of the Indirect Addressing Modes of the processor. The variety of indirect addressing modes provided by the processor determines how easily we can access a data structure defined in a program. Nonetheless, access to any arbitrarily complex data structure can still be performed using pointer arithmetic.

### 7.2.1 Memory Access and One-dimensional Arrays

As described previously, the memory address space of a processor can be viewed as a one-dimensional array of bytes. Consequently, the address of the memory location can be considered as an index into the memory array (which starts at the lowest memory address, and the first item has index 0).

We access consecutive memory locations by incrementing or decrementing the index value and reading or writing to the new array location in each step. Accessing one-dimensional arrays uses basic Indirect Addressing.

Pointer Arithmetic in high-level languages (we will use the C language for comparison) is defined based on the operand type. For example, char is 1 byte (8-bit), short is 2 bytes (16-bits), and long is 4 bytes (32-bits). Therefore, when we increment the index into an array of longs, we point to the next item, while decrementing the index points to the previous item. However, since Address Pointers in a processor refer only to memory address and not array items, and the array may start at some arbitrary memory address location, we must map the array indices into the correct address. The Compiler translates this into the correct memory address pointer based on Equation (7.1).

$$\text{Item Address} = \text{ArrayStart Address} + \text{Index} \times \text{Size of operand}$$  \hspace{1cm} (7.1)

Therefore, to access consecutive items, we must increment or decrement the Item Address Pointer by the size of the operand type. For example, to access the next item in an array of longs, the Item Address Pointer must be incremented by 4 (Figure 7.3). It is a common mistake for beginning Assembly language programmers to increment the address pointer by 1 and not 4 when accessing multi-byte operands.

A related issue to accessing multi-byte values in memory locations is the issue of byte alignment. The ARM processor requires that all word access be word (4-byte) aligned, meaning that the address used to reference the given word value starts on a 4-byte address boundary. Non-aligned access generally results in an Exception on most ARM processors.
7.2.2 Accessing Elements in Multi-byte Structures

High-level languages often provide advanced operand types such as structures (and object classes) that consist of a collection of different basic operand types (elements) grouped together as an item. Array of structures therefore poses additional challenges when we want to access a particular element in each of the array items. In order to access a given data element, we would need to identify the correct item in the array of structures, then the element offset to the specific element within the given item. This would allow us to calculate the element address for the data variable we need using Equation 7.2 or Equation 7.3.

Element Address = ArrayStart Address + Index \times Size\_operand + Element Offset  \hspace{1cm} (7.2)
⇒ Element Address = Item Address + Element Offset \hspace{1cm} (7.3)

7.2.2.1 Declaring Multi-byte Structures in C

In order to better understand how multi-byte structures are used in program, we will first investigate the use of a structure for storing the various actuator values for the robot. This structure will be used for passing the output actuations for the robot from one software module to another. By using a common structure for parameter passing, we avoid the issue of having lengthy lists of input parameters for routines, as well as maintain consistency in the way we access the individual data elements needed to make the actuator controllers work.

Therefore, we define a structure \texttt{actuator\_state} to store the actuation outputs of each Behavior for the \texttt{Mover} robot, to be discussed in further
When defining structures and other complex operand types, high-level languages often use padding to make elements within the structure start at word boundaries to simplify processor access to a particular element (some processors, such as ARM, generate a memory fault when attempting to access half-word or word values which were not stored at appropriate address alignment boundaries). In the `actuator_state` struct given above, one byte of padding is needed to ensure that `tone_freq` and `tone_dur` have the correct word alignment.

Each `actuator_state` struct occupies 16 bytes (15 bytes of actual data and 1 byte of padding), which can be considered as an *item* which stores the output values for a given *Behavior*. We can access the `actuator_state` struct *item* for a particular behavior using Equation 7.1 where *Index* is the index of the given behavior, and *Size* operand = 16. Each `actuator_state` item contains various *elements* such as motor speed, motor brake status, and tone frequency. To determine the tone frequency for a given behavior, we would need to traverse the `behavior_actuations` array, and check the element `tone_freq` in each `actuator_state` structure in our array. We already know how to access a multi-byte item in an array (in this case, the multi-byte item is the `actuator_state` struct) using Equation 7.1. To access a particular element, e.g., `tone_freq`, we need to know the *offset* (index position from the beginning of the structure) of the element, given by Equation 7.2 which can be rewritten as Equation 7.3.

In comparison, the `actuator_state2` structure shown in the following declaration places the `tone_freq` and `tone_dur` elements in between each pair of motor control parameters. This causes the compiler to insert two bytes of padding after each pair of motor control parameters, resulting in

---

2It does not matter whether `tone_freq` and `tone_dur` is declared at the beginning of the structure or at the end, we still need one padding byte per struct to ensure that the next item in the `behavior_actuations` array is word aligned.
Figure 7.4: Accessing an Element in an Array of struct objects
a total of five padding bytes. The resulting structure has a size of 20 bytes compared to 16 bytes in the actuator_state struct.

```c
/* Actuator State Data Structure */
typedef struct {
    SBYTE speedA;
    UBYTE brakeA;  /* 2-byte padding inserted by Compiler */
    ULONG tone_freq;
    SBYTE speedB;
    UBYTE brakeB;  /* 2-byte padding inserted by Compiler */
    ULONG tone_dur;
    SBYTE speedC;
    UBYTE brakeC;
    UBYTE curr_state;  /* 1-byte padding inserted by Compiler */
} actuator_state2;

/* Array Variable for storing Actuator States for each Behavior */
actuator_state2 behavior_actuations2[NUM_BEHAVIOR];
/* sizeof (actuator_state2) = 20 bytes */
/* Non-ideal struct definition with excessive padding */
```

The padding behavior needs to be known when interfacing Assembly Language code to structures created by Compilers. One way to avoid data alignment issues while minimizing the padding overhead within structures is to organize the declaration order of the elements such that minimal or no padding is needed, e.g., in the actuator_state struct definition. In addition, the GNU C Compiler also allows for the generation of packed structures, using:

```c
__attribute__((__packed__))
```

after the struct declaration. Packed structures will not add padding to the structure; this is normally required when defining network communication headers. However, it is not safe to access multi-byte elements in packed structures directly (e.g., accessed using LDR), they must be treated as byte-sized data when reading (i.e., accessed using LDRB) and writing to memory to avoid misaligned data access exceptions.

### 7.2.2.2 Multi-byte Structure Declaration in Assembly

The struct declared in C can be similarly declared in Assembly. The Assembler is not as intelligent as the C compiler however, and we have to be careful regarding data alignment issues in the declaration of variables in Assembly. We must add the assembler directive, `.align` whenever we declare multi-byte data having sizes different from earlier data declarations (e.g., from declaring bytes to declaring words), in order to ensure that word alignment is correct, otherwise data alignment access errors will occur. Access to specific elements in a given struct is done using `.equ` to define the element offsets. Note that the following declaration shows not just type

---

7The `.align` directive has an optional argument. If not specified, it defaults to 2 which corresponds to word alignment for the ARM Architecture. See Appendix B.3 for more information.
definitions, but actual memory storage allocation for the structure as well. This differs from the C language example previously, which only defines the type of the struct variable using `typedef`, but the variables are not declared until later. We can allocate additional `actuator_state` struct variables using the `.space` directive to reserve required data storage based on the size of the structure (given by the `SIZEOF_ACTUATOR_STATE` constant), as follows:

```assembly
.equ NUM_BEHAVIORS, 5
.macro init_actuator_state state_label, \ speedA, brakeA, speedB, brakeB, \ speedC, brakeC, curr_state, tone_freq, tone_dur
  @ Common Structure used to keep latest actuation values
.align 0 @ Start structure on an aligned address
.state_label:
  .byte \speedA
  .byte \brakeA
  .byte \speedB
  .byte \brakeB
  .byte \speedC
  .byte \brakeC
  .byte \curr_state
.align 0 @ Needed to force word alignment padding
 .word \tone_freq
 .word \tone_dur
.endm
.data
.align 0 @ Start structure on an aligned address
actuator_state:
speedA: .byte MOTOR_INHIBITED_BYTE
brakeA: .byte FALSE
speedB: .byte MOTOR_INHIBITED_BYTE
brakeB: .byte FALSE
speedC: .byte MOTOR_INHIBITED_BYTE
brakeC: .byte FALSE
curr_state: .byte ROBOT_STOP
.align 0 @ Needed to force word alignment padding
tone_freq: .word TONE_INHIBITED
tone_dur: .word TONE_INHIBITED
 .equ SIZEOF_ACTUATOR_STATE, (. - actuator_state)
 .equ SPEEDA, (speedA - actuator_state)
 .equ BRAKEA, (brakeA - actuator_state)
 .equ SPEEDB, (speedB - actuator_state)
 .equ BRAKEB, (brakeB - actuator_state)
 .equ SPEEDC, (speedC - actuator_state)
 .equ BRAKEC, (brakeC - actuator_state)
 .equ CURR_STATE, (curr_state - actuator_state)
 .equ TONE_FREQ, (tone_freq - actuator_state)
 .equ TONE_DUR, (tone_dur - actuator_state)
.align
prev_actuator_state:
 .space SIZEOF_ACTUATOR_STATE, 0x0
.align
behavior_actuations:
 .space (SIZEOF_ACTUATOR_STATE * NUM_BEHAVIORS), 0x0
```

7.2.3 Accessing Elements in Multi-dimensional Arrays

Multi-dimensional Arrays are extensions of Multi-byte structures. We can consider them an array of structures (where each item is a structure con-
Figure 7.5: Linearization Sequence for Row and Column Major Formats

We can use this approach even as the number of dimensions increases, though the complexity of Pointer Arithmetic increases accordingly. Storage of each structure is done in consecutive memory locations. This is known as the Linearization sequence for Array storage.

For two-dimensional arrays (where the first dimension is called the row, and the second dimension the column), some high-level languages store the items in row-major format (e.g., C), while others store items in column-major format (e.g., FORTRAN). Row-major format means that items of the inner dimension (the columns) are stored sequentially row by row (all items within a row stored sequentially), while Column-major format stores the items column by column (first item of each row sequentially, then the second item of each row sequentially, and so on). The Linearization order for multi-dimensional storage is important for determining the correct way to calculate item addresses.

We assume that arrays are stored in row-major format in the following discussion. In addition, we assume that the inner dimensions of multi-dimensional arrays have static sizes (i.e., defined in the program) and will not grow or shrink. For example, to access a two-dimensional array Array2D[y][x] (y is the row, and x is the column), we use Equation 7.4, which can be rewritten as Equation 7.5, where the Row and Column terms are given by Equations 7.6 and 7.7 respectively.

\[
\text{Item Address} = \text{Array Start Address} + \text{Index}_y \times (\text{Num Items}_x \times \text{Size operand}) + \text{Index}_x \times \text{Size operand}
\]  
(7.4)

\[
\Rightarrow \text{Item Address} = \text{Row Address} + \text{Column Offset Address}
\]  
(7.5)

Where:

\[
\text{Row Address} = \text{Array Start Address} + \text{Index}_y \times (\text{Num Items}_x \times \text{Size operand})
\]  
(7.6)

\[
\text{Column Offset Address} = \text{Index}_x \times \text{Size operand}
\]  
(7.7)

Accessing elements in a particular item is done in a similar way to that described previously using the element offset. We can set the base pointer...
to the Row Address, and the offset pointer to the Column Offset Address. To access consecutive items within a given row, we only need to increment the column offset address pointer to the next item. To access the next row, the row address pointer needs to be updated with the size of a given row. For byte sized items, column access within a row involves an increment, while row access involves an addition if the size of a row has been pre-calculated.

A three-dimensional array accessed via Array3D[z][y][x] uses Equation 7.8 to determine the Item Address.

\[
\text{Item Address} = \text{Array Start Address} + \text{Index}_z \times (\text{NumItems}_y \times \text{NumItems}_x \times \text{Size}_\text{operand}) + \text{Index}_y \times (\text{NumItems}_x \times \text{Size}_\text{operand}) + \text{Index}_x \times \text{Size}_\text{operand}
\] (7.8)

Most processors do not have built-in support for three or higher dimensions. Consequently, we need to reduce the pointer arithmetic to two dimensions or less.

### 7.2.4 Initializing and Copying Blocks of Memory

Since memory addresses can be considered as a one-dimensional array, we can initialize a block of memory to a particular value by using array traversal. For example, the memset() function in C can be implemented as shown in Listing 7.1.

Listing 7.1: C implementation of byte-sized memset() function

```c
void *memset(void *s, int c, size_t n)
{
    /* Byte-sized location initialization */
    counter = n;
    do {
        *s = (char) c;
        s++;
        counter--;
    } while (counter > 0);
}
```

In this example, we initialize the memory locations one byte at a time. We can improve the throughput by modifying the algorithm to initialize one word at a time, assuming the number of locations to be initialized is divisible by 4, and the starting address is at a 4-byte boundary. In that case, the word-sized memset loop can be implemented in ARM Assembly Language (Listing 7.2).

Listing 7.2: ARM implementation of word-aligned word-sized memset()

```asm
@ Memset Routine in ARM Assembly
@
@ void *memset(void *s, int c, size_t n)
@
```
Figure 7.6: Three Source and Destination Memory Block configurations

(i) Non Overlapping

(ii) Dest Overlaps Source End

(iii) Dest Overlaps Source Beginning

Similarly, we often need to copy blocks of memory from one address location to another. We need to be careful when dealing with memory copying since the source address, memory block range, and destination addresses may overlap. There are three cases that need to be considered by a `memmove()` function\(^4\), shown in Figure 7.6.

In the first case, non-overlapping memory blocks are present. Copying from the source block to the destination block can be performed without any issues. In the second case, when the destination block starts before the end of the source block, we cannot start copying from the beginning of the block since it would overwrite data that is at the end of the source block. We need to start copying from the end of the block towards the beginning instead. In the third case, we cannot start copying from the end towards the beginning since that would overwrite the beginning of the source block. Instead, we

\(^4\) `memcpy()` in the C library does not handle overlapping memory regions
must copy starting from the beginning of the block. This is summarized in
the pseudocode given in Listing 7.3 and implemented in Listing 7.4.

Listing 7.3: Pseudocode for memmove() function

Given Source Begin, Dest Begin, Block Size
 Calculate Source End (Source Begin + Block Size)
 If (Source Begin < Dest Begin < Source End)
   Source Pointer = Source End - 1
   Calculate Dest End = Dest Begin + Block Size
   Dest Pointer = Dest End - 1
   Pointer Increment = Decreasing (negative)
 Else // If (Dest Begin < Source Begin < Dest End) or Non overlap
   Source Pointer = Source Begin
   Dest Pointer = Dest Begin
   Pointer Increment = Increasing (positive)
 Endif
 Counter = Block Size
 Repeat
   *Dest Pointer = *Source Pointer
   Source Pointer = Source Pointer + Pointer Increment
   Dest Pointer = Dest Pointer + Pointer Increment
   Counter = Counter - 1
 Until (Counter == 0)

Listing 7.4: ARM implementation of memmove()

@ void *memmove(void *s1, const void *s2, size_t n);
@ Given Source Begin, Dest Begin, Block Size
@ Calculate Source End = Source Begin + Block Size
@ If (Source Begin < Dest Begin < Source End)
@   Source Pointer = Source End - 1
@   Calculate Dest End = Dest Begin + Block Size
@   Dest Pointer = Dest End - 1
@   Pointer Increment = Decreasing (negative)
@ Else // If (Dest Begin < Source Begin < Dest End) or Non overlap
@   Source Pointer = Source Begin
@   Dest Pointer = Dest Begin
@   Pointer Increment = Increasing (positive)
@ Endif
@ Counter = Block Size
@ Repeat
@   *Dest Pointer = *Source Pointer
@   Source Pointer = Source Pointer + Pointer Increment
@   Dest Pointer = Dest Pointer + Pointer Increment
@   Counter = Counter - 1
@ Until (Counter == 0)
@ On Entry:
@   R0: dest memory pointer s1
@   R1: source memory pointer s2
@   R2: number of bytes n
@ On Exit:
@   R0: memory pointer s1
@   R1, R2, R3: destroyed

memmove:

159
PUSH (R0, R4, LR)  @ Keep s1, R4
ADD R3, R1, R2  @ R3: Source End := Source Begin (s2) + n
CMP R0, R1  @ Compare Dest Begin (s1) vs. Source Begin (s2)
BLS memmove_else  @ Source Begin >= Dest Begin, goto else
CMP R0, R3  @ Compare Dest Begin (s1) vs. Source End
BHS memmove_else  @ Dest Begin >= Source End, goto else

memmove_then:
  SUB R1, R3, #1  @ Source Pointer := Source End - 1
  ADD R0, R0, R2  @ R0 := Dest End := Dest Begin (s1) + n
  SUB R0, R0, #1  @ Dest Pointer := Dest End - 1
  MOV R3, #-1  @ R3: Negative Pointer Increment
  B memmove_loop

memmove_else:
  MOV R3, #1  @ R3: Positive Pointer Increment
  @ R0, R1 already contain Dest Begin and Source Begin Pointers

memmove_loop:
  LDRB R4, [R1], R3  @ R4 = *Source Pointer, update Source Pointer
  STRB R4, [R0], R3  @ *Dest Pointer = R4, update Dest Pointer
  SUBS R2, R2, #1
  BNE memmove_loop

memmove_exit:
  POP (R0, R4, PC)  @ Restore R0, R4, retrieve LR to PC, exit

7.3 Pattern Matching and Data Manipulation using Pointers

Pattern matching deals with the general problem of how to determine the location of a particular value or sequence of values in a data structure. One common use of pattern matching is in String Manipulation. Strings are simply array of characters. Typically, C-style strings consist of a sequence of non-null byte-sized data, and terminated by a null (0x00) character. Therefore, a string of length n can have up to (n-1) characters, followed by a null. If it exceeds (n-1) characters, it would overwrite storage space for other variables.

7.3.1 String Initialization and Copying

String Initialization is identical to array initialization using memset(), except that it must be terminated by a null character. Therefore we can call memset() with counter = (n-1), and then store a null to index n. However, if all we are interested in is to clear a string, it may be sufficient to just store a null character at the beginning of the string at index 0.

Similarly, String Copying is identical to array copying using memcpy() except that it terminates when a null character has been copied. While it is not incorrect to copy the entire allocated memory space to the destination string, it represents additional processing overhead that may not be required.
7.3.2 String Manipulation

String manipulation is a special case of data manipulation. The difference lies in the terminating conditions; data manipulation typically needs to specify the starting and ending range of memory locations to examine. However, in String manipulation, only the starting location needs to be specified, since the end of the string is inferred by the presence of the null character value (0x00).

Often we need to search for the occurrence of a character within a string. For example, in a text file parser, we may want to search for the occurrence of ’newline’ characters to separate lines from each other. Other string manipulation functions include string concatenation (adding a string to the end of another string), string comparison (to determine whether one string is identical to another), substring matching (to determine whether a given string is found in another), and determining the length of a string.

For example, the C standard library provides the `strchr()` function for the purpose of searching for the occurrence of a character. A simple implementation of `strchr()` using C is given in Listing 7.5.

Listing 7.5: C implementation of strchr()

```c
char *strchr(void *s, int c)
{
    while (*s) {
        if (*s == c)
            return s;
        else
            s++;
    }
    return NULL;
}
```

The important thing to note is that the exit condition for the while{} loop is either the end of string, or if a match occurs. If a match occurs, the pointer to the first occurrence of the character in the string is returned. Otherwise, a NULL pointer (0x00000000) is returned if no match is detected. The ARM Assembly program is given in Listing 7.6.

Listing 7.6: ARM implementation of strchr()

```assembly
@char *strchr(void *s, int c)
@{
    @while (*s) {
        @if (*s == c)
            @return s;
        @else
            s++;
        @}
    @return NULL;
@};
@On Entry:
@    R0: string pointer s
```
Look Up Tables (LUTs) are often used to store data values that are needed frequently. For example, the prompts used in a menu can be stored in a LUT, to enable a menu display routine to show the available selection by looping through each item in the LUT and sending it to the display. By using LUTs we can change the language or description of the prompt without affecting the algorithm used to display it. In fact, the number of items can also be increased or decreased without changing the prompt display algorithm if it is written to use a variable to control the number of items to display.

LUTs also serve another important function in Assembly language programming. Programmers often turn to Assembly Language to speed up the execution of an algorithm that cannot be coded efficiently in a high level language. However, such algorithms tend to be processor intensive and the speedup may only be a factor of 2 or 3 when implemented in Assembly. LUTs offer the potential to speed up such computationally intensive algorithms by a factor of 10 to 100. Although the concept of using LUTs for speeding up algorithm implementation is not restricted to Assembly Language usage (it can be used just as successfully in high level language programming), it is often critical for Control and Embedded systems where the response time to some particular input must be constant and the calculations needed to achieve it takes too long to complete, or else is not able to be implemented in a given processor (e.g., square root function in a processor without an advanced math unit).

Control Algorithms in Embedded Systems are often implemented as LUTs since the results of complex calculations are already pre-calculated and stored in the LUT. The LUT acts as a black-box that takes inputs and gives required outputs (Figure 7.7).
7.4.1 Simple LUTs Via Direct Lookup

The simplest way to implement a Look Up Table for a complex algorithm is to pre-compute the results and store it into an array. For example, if we need to find the integer square root (represented mathematically as $y = \lfloor \sqrt{x} \rfloor$) for an 8-bit input value $x$ with a range of [1-255], we can create a LUT as shown in Table 7.1. Note that 256 entries were defined (for a range of [0-255]) in the table, one for each possible value of $x$, to simplify the Look Up process. This eliminates the need to validate that the $x$ input value is in range using additional instructions.

The LUT will be defined as a byte array of 256 entries in the Assembly source file, where each element in the array contains the values of $y$. There is no need to store the $x$ (index) value; it is implicit since the array index is used to retrieve the correct item from the LUT via Direct Lookup (Listing 7.7).

### Listing 7.7: ARM Implementation of Simple LUT

@ Code Fragment for Simple LUT for integer square root
.data
.align 0
@ x (index) 0 1 2 3 4 ...
sqroot: .byte 0x0, 0x1, 0x1, 0x2, 0x2, ...

.code 32
.align 4
...
@ R0: y (output value)
@ R1: Index
@ R2: Array Pointer

163
Generally, the array index is given as an input value to the routine which implements the square root LUT, and not a constant as shown in the example. In addition, the Item Address calculation must take care of the size of the output value $y$, as described in Section 7.2.

However, not all algorithms are suitable for implementation as Simple LUTs. Suitable algorithms have the following criteria:

- The input and output ranges must be finite.
- LUT storage space considerations vs. code size vs. speed of algorithm: It is not practical to implement a one-to-one LUT for 32-bit input values ($x$). This would require 4 GB of storage space just for a table containing 8-bit output values ($y$), more if each output value require multi-byte storage.
- If we want to use simple Lookup with one-to-one mapping between the input and output values, we must use discrete inputs (typically integers, or enums) as the LUT index.

The technique of using LUTs for program execution control, called *Jump Tables*, is given in Section 7.5.1.

### 7.4.2 Advanced LUTs

We can provide sparse-mapping and many-to-one mapping by using the concept of Keys or Input Bins (range of input values). Keys are discrete valued inputs belonging to a sparse (non-contiguous) range that require exact matches. Example of keys include subsets of ASCII characters, non-contiguous integer values, enums, Hash values, Strings, etc. Input Bins are used for continuous valued inputs, and generally a Lower Bound ($x_{\text{min}}$) and an Upper Bound ($x_{\text{max}}$) will be specified for each input value $x$, which maps to a single output value $y$ in a many-to-one mapping. For example, if the input value is Calendar Time, then Input Bins can be defined on the Days of the Week (where all Calendar Time values for Monday from ‘Monday 00:00:00’ ($x_{\text{min}}$) to ‘Monday 23:59:59’ ($x_{\text{max}}$) would match against the output value $y$ for Monday).

In such a LUT, the Key $x$ or Input Bin bounds ($x_{\text{min}}$, $x_{\text{max}}$) has to be stored in addition to the output value $y$. It may or may not be necessary to store both the Lower Bound and the Upper Bound for each Input Bin range, depending on whether the Input Bins have a contiguous range or occupy disjoint ranges. Disjoint ranges would require storing both the Lower and Upper Bounds. Contiguous range Input Bins can be represented by either the Lower Bound or Upper Bound only, since the other value is implicit. It should be noted that contiguous range Input Bins need to specify the range

---

5This result in an approximated output to a given input value. This may or may not be acceptable depending on the type of application.
Table 7.2: Advanced LUT for $\sqrt{}$ using Lower Bound Input Bins

<table>
<thead>
<tr>
<th>Input Bin $x_{min}$: ($x_{min} \leq x &lt; x_{min+1}$)</th>
<th>Output ($y = \lfloor \sqrt{x} \rfloor$)</th>
<th>Index (not stored)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65025</td>
<td>255</td>
<td>254</td>
</tr>
<tr>
<td>65536 (stored as 0)</td>
<td>0</td>
<td>255</td>
</tr>
</tbody>
</table>

criteria carefully to avoid overlapping ranges since only one of the bounds is stored. Typically the range would be specified as $[x_{min}, x_{min+1})$ where $x_{min+1}$ is the next Input Bin Lower Bound value; or $(x_{max}, x_{max+1}]$ where $x_{max+1}$ is the next Input Bin Upper Bound value.

Normally, the LUT would be sorted according to the Key or Input Bin values, so that a Binary Search algorithm can be used to locate the required Key or Input Bin quickly. Nonetheless, if the number of items in the LUT is small, a Linear Search may suffice. Given an input value $x$, the LUT would have to be searched to locate the matching Key or the Input Bin ($x_{min}, x_{max}$) that the value of $x$ falls into. Using the example of the integer square root LUT described in Section 7.4.1, we can extend it to support a 16-bit input value for $x$ (with a range of $[1-65535]$) by specifying suitable Input Bins, to give the required output $y$ (Table 7.2) using a 256-entry Advanced LUT6.

This can be implemented in ARM Assembly by defining an array containing structures with two elements, namely the Input Bin Lower Bound $x_{min}$ and $y$. However, this would lead to excessive use of storage space, since $x_{min}$ is 16-bits, while $y$ is 8-bits. The 3-byte structure will incur one byte of padding per LUT entry. The alternative is to store the Input Bin Lower Bound $x_{min}$ and Output Value $y$ in separate Arrays (Listing 7.8).

Listing 7.8: ARM Implementation of Advanced LUT

```assembly
@ Routine for Advanced LUT for integer square root
.data
.align
@ x (index) 0 1 2 3 ... 254 255
sqroot_xmin:
   .hword 0x1, 0x4, 0x9, 0x10, ..., 0xFE01, 0x0
sqroot_y:
   .byte 0x1, 0x2, 0x3, 0x4, ..., 0xFF, 0x0
.code 32
.align 4
@ On Entry:
   @ R0: input value
   @
@ On Exit:
   @ R0: return y (output value)
   @ R1, R2, R3: destroyed
```
The first step in the Lookup process is to determine the correct index from the Input Bin array containing \(x_{\text{min}}\), then retrieve \(y\) from the Output Value Array using the given index. In case \(x\) was not found in the LUT, the last entry (for \(x_{\text{min}} = 65536\)) which was used as a guard value for \(x_{\text{min}}+1\) contained the error value to be returned. Of course, if both \(x_{\text{min}}\) and \(y\) were both 16-bit values, they should be stored together as a structure to simplify the lookup process (since no padding is required). The decision whether to store the Key or Input Bin values together with the Output Values depend on the trade off between padding overheads and algorithm simplicity.

### 7.5 Decision Making Revisited

Decision making involves several levels. There is high level decision making for long term goals, which involves goal seeking, objective optimization, and other heuristics-based approaches. High level decision making is necessary for a robot to achieve a stated objective (e.g., find blue colored objects that are squares). While high level decision making is important, they would eventually need to be implemented as many short term goals which involve low level decision making. We will focus on low level decision making in this chapter.

---

7This value is actually stored as 0 in the LUT, and will be detected in the LUT algorithm and updated to 65536.
Low level decision making involve making choices among two or more available options, based on some input. These choices will then result in some response or action. We have already looked at If...Then...Else Conditional Statements in Chapter 6.8. Here, we will examine how Switch / Case Statements can be implemented effectively in ARM Assembly Language.

7.5.1 Switch Statements Using Jump Tables

The Follow-Line behavior has to determine whether the Light Sensor readings indicate that it is on a Line, Edge or Outside. The Grasper version of the Follow-Line behavior performs the line classification via a hard coded algorithm. We can replace the hard coded line classification algorithm with a Disjoint-Range Look Up Table (LUT) for Line Classification (Section 7.4.2), which returns line_type as an Enumerated Value (enum) comprising of the following possible values: Outside, Edge, Line and Unknown (where Outside = 0, Edge = 1 and Line = 2, and Unknown = 3). Subsequently, we can initiate the appropriate behavioral response by performing different actions based on the line-type. One way to do so is by means of nested If...Then...Else statements, which can be cumbersome. Fortunately, C provides the Switch statement to perform actions based on the value for a given variable. An example of a C Switch statement for the Follow-Line behavior is:

```c
enum line_type {OUTSIDE = 0, EDGE, LINE, UNKNOWN};
switch (line_type) {
  case OUTSIDE:
    fast_cwrotate_motion();
    break;
  case EDGE:
    slow_ccwrotate_motion();
    break;
  case LINE:
    forward_motion();
    break;
  case UNKNOWN:
    default:
      stop_motion();
      break;
}
```

This can be coded efficient as a Jump Table (Listing 7.9).

```
Listing 7.9: Jump Table Implementation of Switch

.data
.align 4
@ The execution clauses for each line_type enum are implemented
@ as routines which do not need any input parameters
line_type_jumptable:
  .word fast_cwrotate_motion @ OUTSIDE, Enum 0
  .word slow_ccwrotate_motion @ EDGE, Enum 1
  .word forward_motion @ LINE, Enum 2
```

167
Determining the Number of Non-Default Routine Enums Automatically

* Via Table Size Calculations
* Do not include the default routine since it does not have an enum
* Note: We also exclude UNKNOWN since it has the same action as Default

```assembly
.equ num_line_type, (. - line_type_jumptable)/4 - 1
```

```assembly
.code 32
.align 4

@ Line Type Action Routine Dispatcher
@ On Entry:
@   R0: Line Type Enum [0 to (num_line_type-1)]
@ On Exit:
@   R0, R1, R2: destroyed
@ Register Usage:
@   R1: Line Type JumpTable Array Pointer
@   R2: Action Routine Address Pointer

line_type_dispatcher:
PUSH {LR}
CMP R0, #num_line_type @ Check enum value
MOVHI R0, #num_line_type @ Range exceeded, force to default
@ #num_line_type == index for the default command (last entry in Jump Table)
LDR R1, =line_type_jumptable @ Jump Table Address Pointer
LDR R2, [R1, R0, LSL #2] @ R0 LSL #2 = Enum x 4
@ Item Address = R1 + R0 x 4
@ R2 now contains execution routine address (Jump Table Address + Item Offset)
MOV LR, PC @ LR = PC + 8 (pipeline)
MOV PC, R2 @ Call Execution Clause Routines
done_execution:
  ...
PPOP {PC}
.end
```

For simplicity, the **Follow-Line** behavior assumes that the various action clauses are written as routines which do not require any input arguments. From the example, there are four defined actions for the possible line type values. The default action routine as the last item in the jump table. The first thing that the **Switch** statement does is to check that the `line_type` input in R0 is in range; invalid values will be handled by the default action clause. After validating the input, the corresponding Table entry is retrieved from memory using:

```assembly
LDR R2, [R1, R0, LSL #2]
```

The enum in R0 is used as the index into the Jump Table, which is then converted into the item offset by multiplying it with the table entry size (4 bytes for each entry). The **Effective Address** is calculated by adding the item offset to the Jump Table starting address. Control is passed to the routine after setting up LR with the return address to `done_execution` using:

```assembly
MOV LR, PC
```

When we return from the routine call, we continue the program at the label `done_execution`. This is due to the fact that at that point in time...
when the PC was copied to LR, the PC contains the value of that instruction address + 8 (due to the 3-stage pipeline of the ARM processor), which corresponds to the address of the instruction at the label `done_execution`. Note that we cannot use the `Branch and Link` instruction to call the routine since BL expects the target address as an immediate offset value in the instruction operand. Instead, we perform an indirect jump to the action routine address by copying the address of the target routine to the PC using the `MOV` instruction.

### 7.5.2 Jump Tables with Key Lookup

For the general case, the variable used for the Switch statement may not be sequential or start from 0. For such cases, we can use store the switched value as a Key for searching the Jump Table, and the matching entry in the table contains the routine address to be called. A special key value should be defined to detect non-matching switch input values, for calling the default execution routine instead. Let’s take the case where various characters sent to the robot via USB or Bluetooth links are used as remote control inputs to cause the wheels of the robot to move in a certain way. Since the key is a single character which can be stored in one byte, it is more space efficient to keep the Key Table separate from the Routine Address Table. Otherwise we would need to use three additional padding bytes per Jump Table entry to keep the routine address word aligned. The input characters are used as Keys for searching the Key Table, to determine the appropriate entry index. This index is then used to retrieve the execution routine from the Routine Address Table. In this example, string Pattern Matching is combined with Jump Table dispatch to achieve the requirements of the algorithm. the Key Table is organized as a C-String, where each byte is a character corresponding to a remote command. We search the string until either a match is obtained or we reach the end of the string. The index value obtained by the Pattern Matching algorithm is then used to reference the Jump Table to access the appropriate action routine. A default routine is associated with the null character of the C-String to be triggered if no matching command were found. This is illustrated in Listing 7.10.

```assembly
Listing 7.10: Key Lookup Jump Tables

@ Advanced Jump Table Implementation of
@ Non-contiguous Switch statement
.data
.align
cmd_char_keytable:
    .byte 'H','S','F','L','R',0 @ ASCIIZ to indicate End of Key Table
.align
.cmd_state_jumptable:
    .word halt_routine @ HALT, 'H'
    .word move_forward_slow_routine @ MOVE_FORWARD_SLOW, 'S'
    .word move_forward_fast_routine @ MOVE_FORWARD_FAST, 'F'
    .word turn_left_routine @ TURN_LEFT, 'L'
    .word turn_right_routine @ TURN_RIGHT, 'R'
    .word default_routine @ Default, no enum

.align

@ On Entry:
```

169
7.5.3 Advanced Jump Tables

The example given in Listing 7.9 assumes that all the switch cases are implicit and sequential, starting from zero, and hence does not need to be stored in the Jump Table itself. We can use a Disjoint-Range Lookup Table (Section 7.4.2) to combine the process of Line Classification with the dispatch of the appropriate action routine. The lower and upper bounds corresponding to the respective line types are stored for a given entry in the Advanced Jump Table, together with the action routine address to be called. The last entry should use special bound values to catch non-matching input values, to trigger the default action routine instead. In this way, we can support non-contiguous input values which potentially occupy a very large range (e.g., a 32-bit input value can have up to 4 billion possible values).

The Advanced Jump Table incorporating Line Classification is illustrated in Listing 7.11.
@ Use macros to make line classification declaration consistent

.macro adv_jump_table_item entry_label, minval, maxval, action_routine
.align
.entry_label:
.byte \minval, \maxval
.align
.word \action_routine
.endm

@ Element Offsets for Table Lookup

.equ MINVAL_OFFSET, 0
.equ MAXVAL_OFFSET, 1
.equ ACTION_ROUTINE_OFFSET, 4
.equ SIZEOF_ADV_JUMP_TABLE_ITEM, 8

@ The execution clauses for each line_type enum are implemented
@ as routines which do not need any input parameters
@ Each row (item) in the Jump Table takes 8 bytes (with 2 bytes padding)

@ The Line Classification bounds is sorted from low to high range

line_classification_jumptable:
  adv_jump_table_item line_outside, WHITE_MIN, WHITE_MAX, fast_cwrotate_motion
  adv_jump_table_item line_edge, EDGE_MIN, EDGE_MAX, slow_ccwrotate_motion
  adv_jump_table_item line_line, BLACK_MIN, BLACK_MAX, forward_motion
  adv_jump_table_item line_unknown, 0, 255, stop_motion

@ End of Table address constant
.equ line_classification_jumptable_end, .

.code 32
.align 4

/ *
* Advanced Jump Table Action Routine Dispatcher
* *
* Find Jump Table Item such that
* ((MINVAL <= Input Min Value) &&
*  (Input Max Value <= MAXVAL))
* *
* On Entry:
*  R0: Input Min Value (8-bit unsigned)
*  R1: Input Max Value (8-bit unsigned)
*  R2: Jump Table Item Pointer
*  R3: Jump Table End Pointer
*  Assumes that R0 <= R1 (i.e., min <= max)
* *
* On Exit:
*  R0: Jump Table Entry Found (Boolean)
* *
* Register Usage:
*  R0: Input Min Value / Action Routine Address Pointer (if matched)
*  R1: Input Max Value
*  R2: Item Min Value (scratch register)
*  R3: Item Max Value (scratch register)
*  R4: Jump Table Item Pointer
*  R5: Jump Table End Pointer
* *
* Linear Search used, since number of cases are small
*/

advanced_jump_table_dispatcher:
PUSH {R4, R5, LR}

jump_table_loop:
  CMP R4, R5, LTR
  BNE exit_advanced_jump_table_dispatcher
  LDRB R2, [R4, #MINVAL_OFFSET] 

  MOV R4, R2
  MOV R5, R3

  LDRH R0, [R2, R4, #MAXVAL_OFFSET] 

  BNE exit_advanced_jump_table_dispatcher

  BNE jump_table_loop

exit_advanced_jump_table_dispatcher:
  BX LR

jump_table_loop:
  CMP R4, R5, LTR
  BNE exit_advanced_jump_table_dispatcher
  LDRB R2, [R4, #MINVAL_OFFSET] 

  MOV R4, R2
  MOV R5, R3

  LDRH R0, [R2, R4, #MAXVAL_OFFSET] 

  BNE exit_advanced_jump_table_dispatcher

  BNE jump_table_loop

exit_advanced_jump_table_dispatcher:
  BX LR
LDRB R3, [R4, #MAXVAL_OFFSET]

check_bounds:
  CMP R0, R2
  BLO next_jump_table_item @ R0 < MINVAL
  CMP R1, R3
  BHI next_jump_table_item @ R1 > MAXVAL

matched_classification_item:
  @ (MINVAL <= R0) && (R1 <= MAXVAL)
  LDR R0, [R4, #ACTION_ROUTINE_OFFSET] @ R0 contains action routine address
  MOV LR, PC @ LR = PC + 8 (pipeline) == done_execution
  MOV PC, R0 @ Call Action Routines

done_execution:
  MOV R0, #TRUE @ Return Line Classification Found (TRUE)
  B exit_advanced_jump_table_dispatcher

next_jump_table_item:
  ADD R4, R4, #SIZEOF_ADV_JUMP_TABLE_ITEM
  B jump_table_loop @ Repeat for next item

exit_advanced_jump_table_dispatcher:
  POP {R4, R5, PC}

.end

In this routine, each item in the Advanced Jump Table takes 8 bytes, where 2 bytes are used for the lower and upper bounds, and 4 bytes are used for the routine address pointer. We need to include 2 bytes for padding for each item, to ensure that the routine address value is word-aligned. This wastes 2 bytes per item. If the number of entries is large, it might be worthwhile to split the jump table into two sub-tables. In such a design, the first table only stores the bounds and will be used to search for a matching item entry, and calculate the index into the second table containing the routine address pointer. Nonetheless, for the Follow-Line Behavior there are only four items so the overheads are not significant enough to justify such a design.

7.5.4 Advantages and Disadvantages of Jump Tables

The use of Jump Tables involves some tradeoffs, similar to any design choice. Firstly, it trades off one-off complexity with ease of adding new action routines. The complexity of developing a Jump Table based solution comes from the Dispatcher routine. The logic must be debugged thoroughly to ensure that it is able to handle the lookup process correctly, especially the terminating cases (default action). This process would probably take more time than writing the respective action routines one by one. Nonetheless, once the Dispatcher is done, it is almost trivial to add additional cases to the Jump Table, since it only involves modifying the Jump Table entries instead of writing new code.

The second issue relates to code size. If there were only a few short action routines, then coding them as Jump Tables might result in a larger object code size since the Dispatcher code imposes a fixed overhead, in addition to the memory space needed for the Jump Table entries. However, increasing the number of action routines would result in slower growth in object code size compared to individually written routines for each action.

Lastly, the use of Jump Tables allow for easy modification of the bounds used for selecting a given action routine. Individually written routines typi-
Table 7.3: Actuator Outputs for Individual Mover Behaviors

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Claw Controller</th>
<th>Motor Controller</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move-Object</td>
<td>Close Claws</td>
<td>Sound-seeking</td>
<td>Silence</td>
</tr>
<tr>
<td>Grasp-Object</td>
<td>Close Claws</td>
<td>Stop Movement</td>
<td>Silence</td>
</tr>
<tr>
<td>Open-Claws</td>
<td>Open Claws</td>
<td>Stop Movement</td>
<td>Silence</td>
</tr>
<tr>
<td>Follow-Line</td>
<td>(Inhibited)</td>
<td>Line-following</td>
<td>Silence</td>
</tr>
<tr>
<td>Idle</td>
<td>(Inhibited)</td>
<td>Stop Movement</td>
<td>Idle Tone</td>
</tr>
</tbody>
</table>

...cally uses hard-coded bounds\(^{10}\), meaning that the bound values are defined as constants, whereas the entries of the Jump Table are stored in a Look Up Table and hence can even be updated during program execution if desired.

### 7.6 Mover Robot

*Mover* is an enhancement to the Grasper Robot developed in Chapter 6 with the ability to move the grasped object, triggered by sounds such as a user’s voice or hand clapping. This is implemented as a new behavior, *Move-Object*, shown in Figure 7.8. The Actuator outputs are summarized in Table 7.3.

Similar to *Grasper*, the upper layer behaviors of *Mover* will override any of the lower layer behaviors. *Idle* is always activated, and will play an *Idle Tone* through the Speaker while active to indicate that *Mover* is idle. *Follow-Line* is activated if the Touch Sensor was Inactive and the line type was recognized. If the line type were not recognized, the *Follow-Line*...
Table 7.4: Activation Triggers for Mover Behaviors

<table>
<thead>
<tr>
<th>Activation Trigger</th>
<th>Touch Inactive</th>
<th>Touch Active</th>
<th>Claw Open</th>
<th>Claw Closed</th>
<th>Sound Input Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move-Object</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>11</td>
</tr>
<tr>
<td>Grasp-Object</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Open-Claws</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>-</td>
</tr>
<tr>
<td>Follow-Line</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>11</td>
<td>11</td>
<td>*</td>
</tr>
<tr>
<td>Idle</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Legend: '-' N/A '&&' AND '||' OR

behavior would be inhibited. This allows the Idle behavior to control the Speaker via its Idle Tone actuator output.

If the Touch Sensor were inactive, Open-Claws will be activated when the claws are in the closed position. Grasp-Object and Move-Object behaviors are evaluated when the Touch Sensor is Active. If the claws were open, then Grasp-Object is activated. If the claws were closed, then Move-Object is activated, and will respond to inputs from the Sound Sensor. If Sound from the microphone is detected, the Mover robot will try to move towards the source of the sound (e.g., when someone is clapping), otherwise it will rotate in place until a sufficiently loud input is received. This is summarized in Table 7.4.

7.6.1 Subsumption Support via Staggered Processing

In the Subsumption architecture, behaviors are always active and operate in parallel. It is the outputs of behaviors which are subjected to arbitration (suppression and inhibition). To support the Subsumption architecture in NARF, we will need to split the original Arbiter into two parts, the first, a Dispatcher to evaluate the various behaviors; and the second, a revised Arbiter which will evaluate all potential output signals generated by the various behaviors to determine which should be forwarded to the Controller. The revised flowchart for the enhanced version of NARF, called NxOS Enhanced Robotic Framework (NERF), is shown in Figure 7.9.

In NERF, behavior evaluation, arbitration and actuation are separate steps. Each Behavior will generate the appropriate actuator output to the respective arbiters according to their requirements. Since arbitration is now more closely associated with the respective actuators, we define separate Arbiters to control Movement (wheel actuators), Grasping Action (claw actuator), and Tone Generation (speaker actuator).

The reason that Mover has to rotate in place is because we have only one sound sensor, which does not allow us to determine the direction of a sound directly. By continually rotating in place, this helps us to determine the sound direction by setting a high enough threshold so that only a loud sound input would be detected. This occurs when the sound source is directly in front of Mover. However, this is not foolproof since the sound sensor is not very directional, so loud sounds coming from the back of Mover can still trigger a response. Reflections from nearby walls and other surfaces can also affect the accuracy of the direction determination.

No relation to the popular foam-based toys.
Figure 7.9: NxOS Enhanced Robotic Framework (NERF)
For example, given two behaviors, Grasp-Object and Follow-Line, which affects the movement of the robot, the signals would be the Left and Right Wheel Motor Speeds. Let’s say that the signal for the robot to Follow-Line consists of non-zero motor speed values for the left and right wheels; whereas to Grasp-Object, both wheels should have identical zero motor speed values. The left and right motor speeds for each behavior can therefore be stored in the actuator_state data structure defined in Section 7.2.2.1. For the Mover robot, the third motor controls the Claw and would also have a motor speed value for the claw motor in the actuator_state data structure. Since behaviors are always active and called in turn by the Dispatcher to update their actuation values, a behavior_actuations array of actuator_state structs would be needed to store the outputs of each of these behaviors for further processing by the Arbiter. The Arbiter would examine the behavior_actuations array to determine the highest priority actuation values for each actuator, and store the finalized values in an actuator_state structure. This structure is passed to the Controller, to program the various actuators with the appropriate output values.

7.6.2 Suppression and Inhibition Representation

In NERF, we assume that the behaviors operate concurrently and will all generate their actuator outputs for arbitration. In the Arbiter, we still maintain a fixed priority hierarchical order among the various output actuations. Consequently, higher priority actuator outputs will Suppress any lower priority actuator outputs. Nonetheless, a given behavior may not be interested in controlling a given actuator. Form example, the Follow-Line behavior is not interested in changing the position of the claws. Consequently, it will Inhibit its claw motor output actuations. As a general rule, we will not consider the inhibition of the actuation outputs of other behaviors. This means that NERF implements Self-Inhibition of actuation outputs that a given behavior is not interested in.

The behavior outputs of each behavior are stored in an actuator_state data structure, within a behavior_actuations array sized according to the number of defined behaviors. The definition of the actuator_state struct was presented in Section 7.2.2.1. The actuators used by Mover are the Motor Actuators for the Left and Right Wheels, the Claw Actuator, as well as the Speaker Actuator for tone generation. As mentioned previously for Grasper (Chapter 6.9.3.2), we use generic names for the motor control variables in the actuator_state struct so that we can reuse it for other robots. In addition, we will make use of the previously defined representation for Inhibited motor outputs, which is equal to the signed byte value of -128 (0x80).

In addition to the motor actuators, the Tone generator (speaker) is also activated by at least one behavior, Idle. The tone generator is activated using a given 32-bit frequency (in Hz) for a given 32-bit duration (in ms). The hexadecimal value of 0x80000000 for frequency and duration will be used to indicate Inhibited tones. We will also use the enum macros defined in _c_arm_macros.h (in the NxOS-Armdebug base subdirectory) to help specify the behavior enums used to index the behavior_actuations array. This is represented in NxOS-Armdebug as follows:
/* Signal Inhibition Values */
#define MOTOR_INHIBITED 0xFFFFFF80
#define MOTOR_INHIBITED_BYTE (MOTOR_INHIBITED & 0xFF)
/* For data initialization use */
#define TONE_INHIBITED 0x80000000
/** @name Behavior Enums
* The enums must be consecutive, starting from 0
*/
/*@{*/
ENUM_BEGIN ENUM_VALASSIGN(BBR_IDLE, 0) /**< Initial State. */
ENUM_VAL(BBR_FOLLOW_LINE) /**< Follow Line. */
ENUM_VAL(BBR_OPEN_CLAWS) /**< Open Claws. */
ENUM_VAL(BBR_GRASP_OBJECT) /**< Grasp Object. */
ENUM_VAL(BBR_MOVE_OBJECT) /**< Move Object. */
ENUM_END(bbr_behavior_t)
/*@}*/
.equ BBR_LAST_BEHAVIOR, BBR_MOVE_OBJECT
/* Need to update this if behavior list changes */
.equ NUM_BEHAVIORS, (BBR_LAST_BEHAVIOR+1)

7.6.3 Interleaving Multi-sample Sensor Inputs

In Grasper, the multi-sample collection and the min-max determination process for the Light Sensor readings was performed in one routine. A delay of 3 ms needs to be observed between sample collections due to the limitation of the Analog to Digital Converter used by the sensor. For example, to collect 5 Light Sensor samples the total duration is 15 ms. Since there are now two multi-sample input sources in Mover, namely the Light Sensor and the Sound Sensor, sequential collection of the Light Sensor samples followed by the Sound Sensor samples would double the total sensor input processing time to 30 ms. Fortunately, each Analog sensor generates its reading independently of the other sensors. We can therefore interleave the reading of different sensors inputs without the need for any time delay between these readings. It is only after all the sensors have been read that the 3 ms delay need to be observed before the next set of readings can be carried out. By interleaving the sampling of different sensors, we can maintain a total sampling duration of approximately $(3 \times n)$ ms for $n$ samples from each sensor regardless of the number of sensors that need to be serviced.

7.6.4 Implementing the Dispatcher using Jump Tables

We have discussed the implementation of a Dispatcher using Jump Tables in Section 7.5.1. Since each behavior will be activated in turn, we will just iterate through consecutive jump table array entries instead of having to perform an index lookup for each behavior index unlike what was done in Listing 7.9.

7.6.5 Implementing the Mover Arbiter

The Mover Arbiter will utilize the outputs of each behavior stored in the behavior_actuations array to determine the final actuator_state out-
put to be sent to the Controller. Since the actuators are organized by function, the Mover Arbiter performs its task by dispatching function-specific arbiters to evaluate the given actuator elements for each behavior to determine which behavior has active outputs. If any behavior had inhibited outputs for a given actuator, then the lower priority behavior outputs for that actuator would be evaluated, until a non-inhibited output is found. This non-inhibited output therefore suppresses subsequent lower priority behavior outputs for the given actuator controller.

7.6.6 Minimizing Actuator Updates in the Controller

Each time NERF enters the Controller module, it will first check to see if there were any changes to the controller settings. To speed up the comparison, we can use a pattern matching algorithm\(^\text{13}\) to compare the contents of the actuator_state structure generated by the Arbiter against the contents of the prev_actuator_state structure. If there were no differences, then the actuator programming routines can be skipped in its entirety. Conversely, if the two structures have different contents, then the controller will call the respective actuator programming routines to check for new values to be programmed.

7.6.7 Reimplementing the Follow-Line Behavior

We have seen how we can implement Line Classification using Advanced Look Up Tables with Disjoint Ranges in Section 7.5.3, where the output of the Advanced LUT is the routine address for handling the given line type. In each line type handling routine, we will setup the wheel motors with the appropriate values for forward motion, clockwise, and counterclockwise rotation as well as display a diagnostic message on the LCD. However, since each line type has well defined actuation outputs affecting the wheel motors, we can store in a fixed data structure the actuator_state values needed by the Follow-Line behavior for each line type beforehand, and copy them to the actuator_state output accordingly using memmove(). This illustrates the tradeoff between time to execute instructions for configuring the actuation outputs for the given line type vs. storage space needed to keep the required output values in the actuator_state data structures for the respective line types for immediate use.

7.6.8 Implementing the Move-Object Behavior

The Move-Object behavior implements a Sound-Seeking algorithm which utilizes sound input levels to try to determine the direction of a sound. Since the sound level classification process is similar to the processed used for Line-Classification, we can reuse the Advanced Jump Table Dispatcher approach to implement the Move-Object behavior support routines by classifying sound levels into Quiet, Soft and Loud sounds, and then performing appropriate actions based on the sound level. This makes it easier to debug

\(^{13}\)i.e., the memcmp() routine in the C library.
and verify the correctness of individual behavior support routines for *Move-Object* since the Advanced Jump Table routine dispatch logic has already been debugged for the *Follow-Line* Behavior.

The function of the behavior support routines is to map the sound input levels to a notion of direction of the originating sound. However, sound sources do not necessarily provide a steady input signal to the sensors. For example, sound generated by a person saying ‘Aaahhhh’ continuously until breath runs out is relatively constant in pitch but the loudness may vary depending on how well the person can control the volume. In contrast, the act of clapping generates a sharp, loud transient sound followed by a period of silence. Consequently, it is not possible to just use the current sound input level to determine an appropriate behavior response since the level might indicate that there is no sound being received for the current sampling interval even though a very loud sharp sound was detected during the previous sampling interval.

The alternative approach is to use a *Leaky Accumulator* to determine the appropriate response to such irregular input signals. This approach assumes a periodic sampling process which receives signal input values at regular intervals, termed the sampling interval, which in our case is given by the `ROBOT_SCHED_DURATION` for each iteration of the NERF main loop. The *Leaky Accumulator* is basically a variable which ‘accumulates’ sound input events by being incremented whenever sound is detected during the sampling interval, and decremented when there is quiet during the sampling interval. The increment and decrement quanta can vary depending on the strength of the input signal. For example, a *Soft* sound will increment the *Leaky Accumulator* less than a *Loud* sound. The decrementing action causes the *Leaky Accumulator* to ‘leak’. This usually occurs more slowly compared with the incrementing action, and is necessary to ensure that the behavior does not remain stuck in an active state when no input is detected for a period of time. To maintain an active response, the rate at which incrementing actions increase the *Leaky Accumulator* value must be higher than the rate at which the decrementing actions decrease its value.

The *Leaky Accumulator* will trigger different states depending on its value. In the case of the *Move-Object* Behavior, it will start in *Idle*, then transition to *Waiting*, *Seeking*, and finally *Following* states as the value increases, and in the opposite sequence as the value decreases. An additional consideration is the rate at which state transitions occurs when the variable value is close to the boundary between two adjacent states. If the input signal is highly variable, it is possible that the input causes ‘chattering’ where state transitions occur constantly due to the *Leaky Accumulator* value crossing the state transition threshold repeatedly. Consequently, a *Hysteresis* response[^14] is used to smooth out the transition from one state to another for the *Move-Object* Behavior. The *Hysteresis* response accounts for the current value of the *Leaky Accumulator* as well as the direction of change for the value when determining when to switch states.

[^14]: Hysteresis is used in many control systems to change states based on the history of past control signal input values. Typically two thresholds are used to define a state transition. The first threshold (the higher threshold) is used with increasing control signal values, to cause a transition INTO a given state; while the second threshold (the lower threshold) is used with decreasing control signal values, to cause a transition OUT OF the given state.
For example, when sound is detected, the *Waiting* state could be entered when the *Leaky Accumulator* value increases past 10, but during the subsequent period of silence when the *Leaky Accumulator* value is decrementing, it will not return to the *Idle* state until it drops below 1. This minimizes ‘chattering’ and results in a much more stable response from the *Move-Object* Behavior. This is illustrated in Figure 7.10.

### 7.6.9 Software Modules for Mover

The organization of the *Mover* software modules is pretty similar to that for *Grasper* (Chapter 6.9.4). Significant refactoring of the code was done to support the use of structures for data passing and control of the *Mover* actuators. In addition, various worker routines have been rewritten to implement Look Up Tables for various algorithms, with the introduction of common routines such as the Advanced Jump Table Dispatcher to enable greater code reuse. While software size may have increased due to extensive use of Jump Tables and other Look Up Tables, it provides a foundation for structured enhancement of the robot software and for quick addition of new behaviors and action routines.

### 7.7 Chapter Summary

- The Subsumption Architecture can be realized using the staggered processing approach in a cooperative multi-tasking system.
- An understanding of Pointer Arithmetic is critical to implementing array traversal algorithms.
- String manipulation is a common application of pattern matching algorithms.
- Look Up Tables (LUTs) are frequently used in embedded systems programming to implement complex algorithm calculations.
- Continuous inputs (whether contiguous ranges or disjoint ranges) can be handled using *Input Bins*.  

![Figure 7.10: Move-Object Behavior Conceptual Diagram](image-url)
• Jump Tables are efficient techniques for handling Switch statements.

• Control Algorithms such as Hysteresis and Leaky Accumulators are used to process irregular input signals.

• Mover utilizes data structures, multi-byte arrays, and pattern matching algorithms to implement various modules efficiently.

7.8 Review Questions and Problems

1. For the `memmove()` function, what is the appropriate way to handle the case where (Source Begin == Dest Begin)?

2. Rewrite the word-sized `memset()` function to handle addresses starting at non-word aligned addresses, and arbitrary number of locations.

3. Implement a substring search routine for zero-byte terminated strings similar to `strstr(mainstring, searchstring)` in the Standard C Library which returns a pointer to the start of a matching substring within the main string if a match occurs, else NULL if searchstring is not found in mainstring.

4. The `memset()` and `memmove()` functions can be further optimized by processing multiple words in each loop, using the LDM and STM instructions. Modify the `memset()` and `memmove()` functions to process 8 words per loop. Assume that the source and destination addresses are word aligned, and the number of bytes to transfer is divisible by 4 (multiples of word-sized chunks). You will have to handle the case where the number of bytes to process is not divisible by 32 (i.e., not multiples of 8-words). Why is 8 words the largest chunk of memory that can be practically transferred in one loop?

5. What are the pros and cons associated with the use of Jump Tables for routine dispatch. Formulate some guidelines to specify when it is better to implement routine dispatch as a normal code sequence, and when Jump Tables should be used instead.

6. Determine whether Hysteresis can improve the performance of the other input tasks, namely Light Sensing and Touch Sensing, for Mover. Discuss why the approach should or should not be used in each case.

7. Write an ARM routine to implement the factorial algorithm for values from 0! to 12! using the Look Up Table (LUT) approach.

8. The Sound-Seeking Algorithm is not very robust. Investigate the use of two or more Sound Sensors to provide better detection of sound source direction for Mover. Note that this requires additional Sound Sensors which are not supplied with the standard kit.
Chapter 8

Split Personality

They were standing under a tree, each with an arm round the other’s neck, and Alice knew which was which in a moment, because one of them had “DUM” embroidered on his collar, and the other “DEE”.

from “Through the Looking Glass,” Lewis Carroll, 1832-1898

The earlier discussion on how to subdivide tasks into more manageable chunks focused on the use of subroutines and modules in developing software programs. Nonetheless, software development for microcontrollers such as the Atmel ARM processor involves a tradeoff between implementing features and required storage space, since the on-board Flash and RAM is limited to only 64 KB RAM and 256 KB Flash for the NXT brick. There is another dimension in software development, which involves the optimization of code size in terms of space utilization. One way to accomplish this is to use write routine calls that do not move register contents around too often, as well as the use of equivalent instruction sequences which perform equivalent tasks with less instructions. However, developing these skills involves gaining experience in writing ARM assembly language programs, which takes practice.

Fortunately, there is one feature of the ARM process which helps reduce the code storage requirement, possibly increasing the number of functions that can be supported within a given Flash size. This is accomplished by means of the ARM Processor’s Thumb state. However, Thumb state programming is not a panacea for all software size problems. Some routines such as Exception Handlers operate in ARM state by default (though it is possible to switch to Thumb mode via suitable instructions within the Exception Handler itself), while access to Status Registers is not possible via Thumb mode. Nonetheless, various User level routines could be efficiently coded for Thumb mode execution, thus improving code density [19].

One usage of Thumb and ARM state interworking is in the development of Thumb based user programs which utilize ARM based modules to perform various control and filtering tasks. ARM state is needed for the modules due to extensive calculations used in the algorithms, whereas user programs typically involve user inputs and output tasks that have much lower computational requirements.
8.1 Thumb State

Thumb State is a special feature of the ‘T’ variant ARM processors \[19, 37\]. It implements 16-bit instruction encoding for many of the ARM instructions, as well as a more compact default register file organization (Figure 8.1). This results in higher code density with some limitations on the accessible registers for various instructions. In addition, there is no Conditional Execution option, so instructions would be executed unconditionally except for conditional branch instructions. ARMv5T introduces Thumb-2 Instructions which are 32-bit encoding of additional instructions that is executed in Thumb mode. Thumb-2 will not be covered in this book.

Thumb State divides the Registers into two groups: Lo Registers (R0-R7) and Hi Registers (R8-R15). R13 is used as the Stack Pointer, R14 as the Link Register, and R15 as the PC implicitly (Figure 8.2). Lo Registers (R0-R7) are available in all Thumb instructions, whereas access to Hi Registers (R8-R15) is possible only with restricted instructions. The register contents are unchanged when switching between ARM State and Thumb State, and code execution can pass between the two states easily using variables stored in the R0-R7 registers.

In addition, many of the Thumb instructions will automatically update the Flags in the Program Status Register (CPSR). To avoid confusion, the UAL format for Thumb Instructions explicitly includes the ‘S’ suffix, so that the behavior of the instruction is easily known. The GNU Assembler by default does not use the UAL syntax for Thumb, although it can be enabled using the following directive:

```
.syntax unified
```

Consequently when writing Thumb instruction based Assembly, we should first decide if UAL syntax should be enabled. Otherwise, the Assembler will trigger errors due to unrecognized instruction syntax and arguments, since the old-style Thumb instruction syntax does not specify the ‘S’ suffix for the instructions. Nonetheless, to avoid confusion and maintain compatibility with ARM code, it is recommended that UAL syntax be enabled when developing Thumb code. This would allow us to revert to ARM code generation easily without changing the behavior of the program. It should also be noted that the GNU Objectdump program generates Disassembly output using UAL syntax, so the comparison of the Disassembly output with the original source file would be much easier if UAL syntax were adopted.

8.2 Standard Thumb Instructions

Thumb instructions are divided into the same categories as the ARM versions: Data Movement (Tables 8.1 and 8.2), Integer Arithmetic (Table 8.3), Logic and Bit Manipulation (Table 8.4), Shift and Rotate (Table 8.5), Comparison (Table 8.6), and Flow-Control and Exception (Table 8.7) instructions. Most instructions operate on the Lo Registers only unless noted otherwise.
Figure 8.1: Thumb State Registers, courtesy of ARM Ltd. [7]

Figure 8.2: Thumb to ARM Register Mapping, courtesy of ARM Ltd. [7]
8.2.1 Data Movement Instructions

The Thumb Data Movement Instructions are encoded differently depending on the arguments used, in spite of the same mnemonic provided to the programmer. The MOV instructions have restrictions on the source and destination registers (no Lo to Lo registers), whereas MOVS can only operate on Lo Registers (Table 8.1). Consequently, the operands of the Data Movement instruction have irregular ranges, for example, the LDR command has different offset values depending on whether SP is the Base Pointer Register or otherwise (Table 8.2).

When using R15 (PC) as an operand, the value will be equal to current instruction address + 4, which points to the location two Thumb instructions away from the current instruction. This means that the techniques used for performing Jump Table dispatch described in Section 7.5 will still work for Thumb code.

Dedicated PUSH and POP instructions are provided for Thumb mode, where SP (R13) is implied. The PUSH and POP instructions have restrictions for including LR and PC as register operands and cannot be used for Interworking routines (see Section 8.3).

8.2.2 Integer Arithmetic Instructions

Almost all the Arithmetic Instructions will update the Flags in the Status Register automatically. However, SP and PC related arithmetic instructions, as well as commands involving Hi Registers do not update the Flags (Table 8.3).

8.2.3 Logic and Bit Manipulation Instructions

Thumb Logic and Bit Manipulation Instructions are similar to their ARM versions, except that the Flags are updated automatically. The Logic and Bit Manipulation Instructions are defined in Table 8.4.

8.2.4 Shift and Rotate Instructions

Shift and Rotate Instructions are explicitly defined in Thumb Mode since we do not have the luxury of using ARM Addressing Mode 1 (Operand2) features. The Shift and Rotate Instructions are defined in Table 8.5.

Table 8.1: Thumbv4T Data Movement Instructions 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move (Copy)</td>
<td>MOVS</td>
<td>Rd, #&lt;imm8m&gt;</td>
<td>Lo to Lo. Actually LSLS Rd, Rm, #0</td>
<td>Range: 0-255</td>
</tr>
<tr>
<td>Move (Copy)</td>
<td>MOV</td>
<td>Rd, Rm</td>
<td>Hi to Lo, Lo to Hi, Hi to Hi.</td>
<td>Not Lo to Lo.</td>
</tr>
</tbody>
</table>

1The Move Negative (1’s complement) instruction, although it has a Data Movement mnemonic prefix, is more correctly a Logic instruction and hence is covered in Section 8.4.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Register from Memory</td>
<td>LDR</td>
<td>Rd, [Rn, #&lt;imm5s&gt;]; Rd, [Rn, Rm]; Rd, [SP, #&lt;imm8s&gt;], Rn!</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Store Register Byte from Memory</td>
<td>STR</td>
<td>Rd, [Rn, #&lt;imm5s&gt;]; Rd, [Rn, Rm], Rd, [SP, #imm8s]</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Load Register Byte from Memory</td>
<td>LDRB</td>
<td>Rd, [Rn, #&lt;imm5&gt;]; Rd, [Rn, Rm]</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Load Multiple Registers from Memory</td>
<td>LDMIA</td>
<td>Rn!, &lt;loreglist&gt;</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Store Multiple Registers to Memory</td>
<td>STMIA</td>
<td>Rn!, &lt;loreglist&gt;</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Push</td>
<td>PUSH</td>
<td>&lt;loreglist&gt;</td>
<td>&lt;loreglist&gt;, LR&gt;</td>
</tr>
<tr>
<td>Store Register to Memory</td>
<td>STRB</td>
<td>Rd, [Rn, #&lt;imm5&gt;]; Rd, [Rn, Rm]</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Load Register from Memory</td>
<td>LDR</td>
<td>Rd, [Rn, #&lt;imm5s&gt;]; Rd, [Rn, Rm]; Rd, [SP, #&lt;imm8s&gt;], Rn!</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Load Register Byte from Memory</td>
<td>LDRB</td>
<td>Rd, [Rn, #&lt;imm5&gt;]; Rd, [Rn, Rm]</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Load Register Halfword from Memory</td>
<td>LDRH</td>
<td>Rd, [Rn, #&lt;imm5s&gt;]; Rd, [Rn, Rm]</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Store Register to Memory</td>
<td>STR</td>
<td>Rd, [Rn, #&lt;imm5s&gt;]; Rd, [Rn, Rm], Rd, [SP, #imm8s]</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Load Multiple Registers from Memory</td>
<td>LDMIA</td>
<td>Rn!, &lt;loreglist&gt;</td>
<td>Rn!, &lt;loreglist&gt;</td>
</tr>
<tr>
<td>Push</td>
<td>PUSH</td>
<td>&lt;loreglist&gt;</td>
<td>&lt;loreglist&gt;, LR&gt;</td>
</tr>
</tbody>
</table>

Table 8.2: Thumbv4T Data Movement Instructions 2

Instruction | Opcode Mnemonics | Operands | Description |
---|---|---|---|
Load Register from Memory | LDR | Rd, [Rn, #<imm5s>]; Rd, [Rn, Rm]; Rd, [SP, #<imm8s>], Rn! | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Load Register Byte from Memory | LDRB | Rd, [Rn, #<imm5>]; Rd, [Rn, Rm] | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Load Register Halfword from Memory | LDRH | Rd, [Rn, #<imm5s>]; Rd, [Rn, Rm] | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Store Register to Memory | STR | Rd, [Rn, #<imm5s>]; Rd, [Rn, Rm], Rd, [SP, #imm8s] | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Store Register Byte from Memory | STRB | Rd, [Rn, #<imm5>]; Rd, [Rn, Rm] | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Store Register Halfword from Memory | STRH | Rd, [Rn, #<imm5s>]; Rd, [Rn, Rm] | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Load Multiple Registers from Memory | LDMIA | Rn!, <loreglist> | Rn!, <loreglist> | Incr. After (IA); Rn not in loreglist; Rn updated |
Push | PUSH | <loreglist> | <loreglist>, LR> | Full Descending (FD) Stack; SP updated Push with Link; FD Stack; SP updated |

187
### Table 8.3: Thumbv4T Integer Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>ADDS</td>
<td>Rd, Rn, #&lt;imm3&gt;, Rd, Rn, Rm, Rd, Rd, #&lt;imm8&gt;</td>
<td>Rd := Rn + #&lt;imm&gt;; Range: 0-7 Rd := Rn + Rm; Lo to Lo Rd := Rd + #&lt;imm&gt;; Range: 0-255</td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>Rd, Rd, Rm, Rd, SP, #&lt;imm8s&gt;, SP, SP #&lt;imm7s&gt;</td>
<td>Rd := Rd + Rm; Hi to Lo, Lo to Hi, Hi to Hi. Not Lo to Lo Rd := SP + #&lt;imm&gt;. Range: 0-1020 mod 4; imm8s: 10-bit val &gt;&gt; 2 SP := SP + #&lt;imm&gt;. Range: 0-508 mod 4; imm7s: 9-bit val &gt;&gt; 2</td>
</tr>
<tr>
<td>Address from PC</td>
<td>ADR</td>
<td>Rd, &lt;label&gt;</td>
<td>Rd := PC + #&lt;imm&gt;. Range: 0-1020 mod 4 imm8s: 10-bit val &gt;&gt; 2</td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADCS</td>
<td>Rd, Rd, Rm</td>
<td>Rd := Rd + Rm + Carry; Lo to Lo</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUBS</td>
<td>Rd, Rn, #&lt;imm3&gt;, Rd, Rn, Rm, Rd, Rd, #&lt;imm8&gt;</td>
<td>Rd := Rn - #&lt;imm&gt;; Range: 0-7 Rd := Rn - Rm; Lo to Lo Rd := Rd - #&lt;imm&gt;; Range: 0-255</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SP, SP #&lt;imm7s&gt;</td>
<td>SP := SP - #&lt;imm&gt;. Range: 0-508 mod 4; imm7s: 9-bit val &gt;&gt; 2</td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SBCS</td>
<td>Rd, Rd, Rm</td>
<td>Rd := Rd - Rm - NOT Carry; Lo to Lo</td>
</tr>
<tr>
<td>Negate</td>
<td>NEGS</td>
<td>Rd, Rn</td>
<td>Flags Updated. Rd := - Rn; Lo to Lo Equivalent to RSBS Rd, Rn, #0 (UAL)</td>
</tr>
<tr>
<td>Multiply</td>
<td>MULS</td>
<td>Rd, Rm, Rd</td>
<td>Rd := (Rm × Rd)[31:0]; Lo to Lo</td>
</tr>
</tbody>
</table>

### Table 8.4: Thumbv4T Logic and Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Clear</td>
<td>BICS</td>
<td>Rd, Rd, Rm</td>
<td>Rd := Rd AND NOT Rm</td>
</tr>
<tr>
<td>Bit AND</td>
<td>ANDS</td>
<td>Rd, Rd, Rm</td>
<td>Rd := Rd AND Rm</td>
</tr>
<tr>
<td>Bit Exclusive OR</td>
<td>EORS</td>
<td>Rd, Rd, Rm</td>
<td>Rd := Rd EUR Rm</td>
</tr>
<tr>
<td>Bit OR</td>
<td>ORRS</td>
<td>Rd, Rd, Rm</td>
<td>Rd := Rd OR Rm</td>
</tr>
<tr>
<td>Bit NOT</td>
<td>MVNS</td>
<td>Rd, Rm</td>
<td>Move Negative (bitwise 1’s complement) Rd := NOT Rm</td>
</tr>
</tbody>
</table>
### Table 8.5: Thumbv4T Shift and Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Shift</td>
<td>ASRS</td>
<td>Rd, Rm, #&lt;imm5&gt;</td>
<td>Rd := Rm ASR #&lt;imm&gt;; Range: 1-32&lt;br&gt;Carry unaffected if Rs[7:0] == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rd, Rd, Rs</td>
<td>Rd := Rd ASR Rs;</td>
</tr>
<tr>
<td>LSLS</td>
<td>Rd, Rm, #&lt;imm5&gt;</td>
<td>.</td>
<td>Rd := Rm LSLS #&lt;imm&gt;; Range: 0-31;&lt;br&gt;Carry unaffected if shift == 0&lt;br&gt;(Equiv. to MOVs Rd, Rm)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rd, Rd, Rs</td>
<td>Rd := Rd LSLS Rs;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Carry unaffected if Rs[7:0] == 0</td>
</tr>
<tr>
<td>LSRS</td>
<td>Rd, Rm, #&lt;imm5&gt;</td>
<td>.</td>
<td>Rd := Rm LSRS #&lt;imm&gt;; Range: 1-32&lt;br&gt;Carry unaffected if Rs[7:0] == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rd, Rd, Rs</td>
<td>Rd := Rd LSRS Rs;</td>
</tr>
<tr>
<td>RORS</td>
<td>Rd, Rd, Rs</td>
<td></td>
<td>Rd := Rd RORS Rs;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Carry unaffected if Rs[7:0] == 0</td>
</tr>
</tbody>
</table>

### Table 8.6: Thumbv4T Comparison Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td></td>
<td></td>
<td>Updates flags in CPSR for Rn - Rm&lt;br&gt;Lo to Lo, Hi to Hi, Lo to Hi, Hi to Lo&lt;br&gt;(no restrictions)</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP</td>
<td>Rn, Rm</td>
<td>Updates CPSR Flags for Rn - #&lt;imm&gt;;&lt;br&gt;Range 0-255</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rn, #&lt;imm8&gt;</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>CMN</td>
<td>Rn, Rm</td>
<td>Updates CPSR Flags for Rn + Rm&lt;br&gt;Lo to Lo only</td>
</tr>
<tr>
<td>Compare Negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Test</td>
<td>TST</td>
<td>Rn, Rm</td>
<td>Updates CPSR Flags for Rn AND Rm&lt;br&gt;Lo to Lo only; Does not affect the V flag</td>
</tr>
</tbody>
</table>

#### 8.2.5 Comparison Instructions

Thumb Comparison Instructions are similar to the ARM versions, except that that Immediate values have restricted ranges. Nonetheless, only CMP may be used with any register, the other instructions are limited to Lo registers only. The Comparison Instructions are defined in Table 8.6.

#### 8.2.6 Flow Control Instructions

In Thumb state, the BL instruction is a 32-bit instruction, encoded as two 16-bit closely related Thumb instructions. In addition, the Branch and Link instruction will configure LR to indicate that the return address is a Thumb instruction. The other instructions are similar to their ARM counterparts, except that the Branch instruction has much reduced range, and Explicit Conditional Branch instructions must be used for Thumb algorithms, in place of Conditional Execution of ARM instructions (Table 8.7). The Branch and Exchange instruction is used to switch between Thumb and ARM state,
Table 8.7: Thumbv4T Flow Control and Exception Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Mnemonics</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>B</td>
<td>&lt;label&gt;</td>
<td></td>
<td>PC := label; Label must be within ±4KB</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>B&lt;cond&gt;</td>
<td>&lt;label&gt;</td>
<td></td>
<td>Label must be within -252 to +258 bytes of current Instruction</td>
</tr>
<tr>
<td>Branch &amp; Link</td>
<td>BL</td>
<td>&lt;label&gt;</td>
<td></td>
<td>LR := (PC - 2)</td>
</tr>
<tr>
<td>Branch &amp; Exchange</td>
<td>BX</td>
<td>Rm</td>
<td>&lt;imm8&gt;</td>
<td>Switch between ARM and THUMB modes PC := Rm AND 0xFFFFFFFE Rm[0] = (1: THUMB, 0: ARM) Rm can be Hi or Lo registers</td>
</tr>
<tr>
<td>Supervisor (Software Interrupt)</td>
<td>SVC / SWI</td>
<td>&lt;imm8&gt;</td>
<td></td>
<td>SVC is the new UAL mnemonic SWI Deprecated</td>
</tr>
</tbody>
</table>

as explained in Section 8.3.

8.2.7 Thumb Instruction Encoding

The Thumb instruction encoding format is given in Figure 8.3. As can be seen from the figure, the encoding format is not very uniform. Here compactness of the representation is the primary motivation, and takes precedence over elegance and consistency.

8.3 Calling Between ARM and Thumb Modes (Interworking)

The way to switch from ARM to Thumb mode and back is via the BX instruction \[^2\]. Consequently, routine call and returns must be done carefully in order for Interworking (calling ARM from Thumb and vice-versa) to work. All routines which are to be called via Interworking must be defined as a function using the .type directive, whether they are ARM or Thumb routine. Thumb routines should also be declared using.thumb_func in order for the Linker to recognize that it is a Thumb function. In addition, Interworking typically accesses functions kept in a different source file and compiled separately, so such functions should also be declared .global in order for the Linker to resolve symbols in other object files when creating the executable. Of course, parameter passing between the ARM and Thumb routines must follow the AAPCS requirements.

\[^2\] `POP {..., PC}' or `LDMFD SP!, {..., PC}' can be used to return to a different operating mode in ARMv5T and above, but this feature is not available in ARMv4T.

\[^3\] the .type directive is required for binutils > 2.21 to identify ARM Interworking routines, and is declared as .type <func_name>, %function.

190
### Thumb Instruction Encoding

Figure 8.3: Thumb Instruction Encoding, courtesy of ARM Ltd. [7]
8.3.1 Direct Thumb to ARM Interworking

The typical way of calling an ARM routine from Thumb state is to specify the target routine name using the *Branch and Link* opcode. The Assembler and Linker then adds the necessary glue instructions (called a veneer) to interwork the two routines together. The sequence for calling a routine written for ARM from Thumb is therefore:

```
.align 2
.thumb_func
@ Thumb routine
thumb_routine:
...
@ Thumb to ARM call
BL arm_routine
return_from_arm:
....
```

where *arm_routine* is a 32-bit ARM routine in a different source file, declared as an externally accessible Interworking routine using `.global` and `.type` directives. This is important since calls to ARM Interworking routines from Thumb code will automatically cause the Linker to insert a veneer (i.e., stub code) to switch to ARM state before branching to the actual routine. Failure to declare ARM routines that will be called from Thumb as Interworking routines will result in a direct *Branch and Link* to the ARM routine, which will cause the program to crash since the processor is still in Thumb state.

For a properly declared ARM Interworking routine, the original *Branch and Link* target in the Thumb routine is redirected to point to a Linker generated veneer, which looks like the following (given as a Disassembly fragment):

```
@ Thumb object code
0xxxxx<arm_routine_from_thumb>:
4778 BX PC @ Switch to ARM state via ‘B arm_routine’
46C0 NOP @ 16-bit Thumb instruction (padding)
@ ARM object code
EAxxxx B arm_routine @ 32-bit ARM instruction
```

The *Branch and Exchange* instruction switches state using the address in the Program Counter (PC), equal to the address of the BX instruction + 4 in Thumb state, which is pointing to the 32-bit ARM *Branch* instruction. The processor then immediately executes the *Branch* to the actual ARM routine. To return from an ARM routine to the Thumb caller, the following code sequence is used:

```
.code 32
.align 4
.global arm_routine
.type arm_routine, %function
@ ARM routine
arm_routine:
STMFD SP!, {..., LR} @ PUSH {..., LR} can also be used (UAL)
....
```

According to [39], the BX PC instruction must be executed from a word align address, otherwise the behavior is unpredictable.
The Link Register (LR) contains the address of the Thumb instruction immediately following the original 'BL arm_routine', which is located at the label 'return_from_arm'. In addition, the BL instruction sets bit 0 in the LR in Thumb state, therefore the ‘BX LR’ instruction in the ARM routine will switch execution mode back to Thumb and resume execution at the correct address.

### 8.3.2 Direct ARM to Thumb Interworking

Thumb routines should always be declared using the `.thumb_func` directive, whether they are to be called from ARM or Thumb state. In addition, Thumb Interworking routines should be declared using `.global` and `.type` directives to identify them as externally accessible Interworking routines. It is very important for Thumb routines to be declared using `.thumb_func`, otherwise, the Linker will assume that the Thumb routine is an ARM routine and generate a veneer even when it is not required, such as for calling the Thumb routine from another Thumb routine. The sequence is very similar when calling Thumb from ARM:

```assembly
.code 32
.align 4
@ ARM routine
arm_routine:
    ...  
@ ARM to Thumb call
BL thumb_routine
return_from_thumb:
    ...
```

The original Branch and Link target in the ARM routine is redirected to point to a Linker generated veneer. The veneer switches to Thumb state and continues execution at the target Thumb routine address. The veneer looks something like the following Disassembly code:

```
0xxxxx <thumb_routine_from_arm>:
e59fc000 ldr ip, [pc, #0]
e12fff1c bx ip
0xxxxx .word 0x0xxxxy  @ y ends with bit 0 = 1
```

Since the Thumb routine address must have Bit 0 set to 1 to indicate Thumb state switching, the IP is initialized with the address of the Thumb routine + 1. The `Branch and Exchange` then switches mode and transfers execution to the actual Thumb routine. To return from a Thumb routine to an ARM routine, the following code is used:

```assembly
.align 2
.thumb_func
.global thumb_routine
.type thumb_routine, %function
@ Thumb routine
thumb_routine:
    PUSH (...), LR
```

193
When we enter the Thumb routine, the low registers to be preserved (R0-R7), in addition to the Link Register, are PUSHed onto the stack. The Link Register (LR) contains the address of the ARM instruction immediately following ‘BL thumb_routine’, at the label ‘return_from_thumb’, to be used as the return address. When the routine is ready to return to the caller, we cannot ‘POP LR’ directly due to restrictions in the POP instruction format\(^5\). Consequently, we would need to use one of the argument registers (R0-R3) in order to POP the return address, and perform the Branch and Exchange back to ARM state. For the case where no arguments are returned, R0 could be used for that purpose. Otherwise one of the R1-R3 registers would be used to retrieve the return address (and its previous contents destroyed\(^6\)). In addition, since the return address has bit 0 cleared to 0, therefore the ‘BX Rx’ instruction in the Thumb routine will switch execution mode back to ARM and resume execution at the correct address.

### 8.3.3 Indirect Thumb to ARM Interworking

Previously, we were learning about indirect routine calls, we would use the sequence ‘MOV LR, PC; MOV PC, Rx’ to setup the Link Register LR and then transfer control to the Interworked ARM subroutine. It seems natural to replace the ‘MOV PC, Rx’ instruction with a Branch and Exchange instruction (‘BX Rx’) which will switch state to ARM state for us. However, this does not work since MOV instruction does not update the LR with the Thumb address flag (bit 0) set to indicate a return to Thumb state upon returning from the subroutine via a ‘BX LR’ instruction (Section \[8.3.1\]). Attempting to do so will result in executing what would appear to the processor as invalid ARM instructions located at the Thumb routine addresses following the subroutine call.

Consequently, indirect (register-based) Thumb to ARM Interworking calls are more complex, and we will need to do it indirectly via a Branch and Link call \[38\]:

```
LDR Rx, =arm_routine
B 0f
9:
  BX  Rx  @ register setup before indirect call
0:
  BL  9b
  return_from_call:
  ...
```

\(^5\)The POP instruction supports ‘POP {..., PC}’ but this would not perform a switch back to ARM state in ARMv4T.

\(^6\)Consequently, the register used by BX to return to the ARM routine does not need to be preserved on the stack at the beginning of the Thumb routine since there is no way to restore its value from within the routine before returning.
where register Rx contains the address of the target ARM routine loaded into the register earlier. Three instructions occupying 8 bytes (BL is a double length instruction requiring 4 bytes) are needed to implement the indirect call to the ARM routine. First, the code branches to a Branch and Link instruction. The Branch and Link instruction will set register LR with the correct return address located at return_from_call, with bit 0 set to indicate Thumb state. Control is then transferred to the stub subroutine containing the BX instruction. The BX instruction will use the supplied target address in the register Rx to perform the state switch. The reason that the BX instruction is placed before the Branch and Link instruction is so that the return address set by BL previously will have the next instruction address located at return_from_call, simplifying debugging. The labels ‘0’ and ‘9’ are local labels which can occur multiple times in a source file, and hence are safe to be used in macro definitions.

8.3.4 Indirect Thumb to Thumb Interworking

It would seem strange that calling an indirect Thumb Interworking routine from an existing Thumb routine presents problems that need to be addressed, since there is no state change. It should be remembered that an Interworked Thumb routine exits from the routine via a ‘Bx Rx’ instruction to enable a transfer of control back to a caller which can be in a different execution state (Section 8.3.2). The problem arises with the treatment of the register LR which needs to have the Thumb address bit set before the Branch and Exchange to the target Thumb Interworking routine. With the normal ‘MOV LR, PC; BX Rx’ sequence, the same problem as seen in Section 8.3.3 will also be present. Consequently, indirect calls to Interworked Thumb routines also need to use the same technique described in Section 8.3.3. This means that from a Thumb routine, we need to perform all indirect calls to Interworked routines (whether ARM or Thumb) the same way.

8.3.5 Indirect ARM to Thumb Interworking

Indirect (register-based) ARM to Thumb Interworking calls have the form:

```
LDR Rx, =thumb_routine
MOV LR, PC
BX Rx
return_from_call:
...
```

where register Rx contains the address of the target Interworked Thumb routine loaded into the register earlier. Since the Thumb routine should have been marked as a Thumb function using the .thumb_func directive, the assembler knows to set bit 0 in the function address to indicate a switch to Thumb state.

8.3.6 Interworking Macros

The various Interworking mechanisms can be made into macros to ease the writing of routines that will be Interworked. The arm_interwork and
thumb_interwork macros are used to declare routines with the correct options, whereas direct calling using arm_dcall and thumb_dcall (which is basically a Branch and Link to the target function, that will be replaced by a Linker generated veneer), and indirect (register-based) calling using arm_rcall and thumb_rcall respectively are provided for consistency in performing Interworking calls. These macros are listed in Listing 8.1. The .arm directive is equivalent to .code 32, and was used in its place because it is more obvious what is intended.

Listing 8.1: Interworking Macros

```
.macro arm_dcall arm_routine
BL \arm_routine  @ Linker will generate veneer automatically
.endm

.macro arm_rcall register
MOV LR, PC
BX \register  @ register setup before indirect call
.endm

.macro arm_interwork arm_routine
.align 4
.arm
.type \arm_routine, %function  @ Needed by new binutils (>2.21)
global \arm_routine
\arm_routine:
.endm

.macro thumb_dcall thumb_routine
BL \thumb_routine  @ Linker will generate veneer automatically
.endm

.macro thumb_rcall register
B 0f
BX \register
0:
BL 9b  @ register setup before indirect call
.endm

.macro thumb_interwork thumb_routine
.align 2
.thumb_func
.type \thumb_routine, %function  @ Needed by new binutils (>2.21)
global \thumb_routine
\thumb_routine:
.endm
```

8.3.7 The Evolution of Thumb State Support

The ARMv6T2 architecture and beyond introduced additional 32-bit instructions called Thumb-2 instructions. These are similar to ARM instructions except that they do not have conditional execution capabilities and operate in Thumb state. The most recent embedded ARM processor architecture variants, ARMv6-M [40] and ARMv7-M [41], support only Thumb state, in contrast to the more powerful ARMv7-AR [42] architecture variants which retains ARM state support. Consequently, for deeply embed-
ded systems, Thumb state becomes the de-facto programming environment. The 16-bit Thumb and 32-bit Thumb-2 instructions provide a streamlined programming model which avoids the overheads and complexities of switching between the ARM and Thumb states, while retaining the large register file design central to the ARM architecture. Nonetheless, ARM and Thumb interworking is full supported for the ARMv7-AR architecture, and remains useful for supporting advanced applications as well as existing codebases.

8.4 Movement Control Basics

The ability for Tribot to move towards a given location involves close control of the motor actuators attached to the wheels. If the motors were activated at high speed for too long, the movement may overshoot the destination, whereas several slower, shorter activations of the motor would avoid overshooting at the expense of slow progress. The ideal situation is for Tribot to activate the motors long enough to reach the target location in a single smooth movement. Nonetheless, achieving that is difficult since it is not in full control of its environment. Friction between the wheels and the ground may affect the acceleration and travel of the wheels. In addition, wheel slippage and other inefficiencies in the motor can also cause the robot to veer off course. The most common way to solve such problems is to use a close-loop control algorithm, which receives feedback from servos regarding how close the output is to the target value, and adjust the output based on the difference. This is typically achieved using a Proportional Integral Derivative (PID) Controller which has been extensively used in control systems for solving such problems [10]. While it is not possible to explain in detail the function of the PID Control Algorithm, a brief introduction to the various terms and concepts used for PID sufficient for the discussion of the implementation of a PID controller will be presented.

8.4.1 Control and Feedback Loops

There are two types of basic control mechanisms: Open and Closed Loop Control. Open Loop Control executes the control function on the output device based on pre-defined output actions which are usually developed and tested in a controlled environment (off-line environment) until the required behavior is achieved. It is then deployed in the actual system for use, where the output device and required actions are well understood and the environment does not change. On the other hand, Closed Loop Control is used when the environment is variable, or when the output device requires fine adjustment to achieve the required output action. An example of such an output device is the MINDSTORMS servo-controlled motor used for motion. The application of voltage to the motor leads to movement, but the angle and distance moved depends on the gear ratios and loading applied to the motor. Friction and inertia results in slightly different results depending on the duration of activation of the motor (short activation times will have more variable results compared to long activation times which enables the motor output to stabilize). For example, when we are trying to move Tribot to a certain location, the actual movement can either overshoot the target
location when the output signal is *Underdamped*, or else takes a long time to reach the desired position, due to *Overdamped* response. It may even *undershoot* if the target location is never reached (Figure 8.4). The ideal case is to have *Critically Damped* response where the target location is reached in a minimal amount of time, without oscillating around the desired location. However, achieving *Critically Damped* response is practically impossible in most real systems. Instead, some residual *Ringing* response where the output oscillates with decreasing error occurs until it finally converges on the desired output value. The time required to reach the desired output value is called the *Settling Time*.

The Closed Loop Control mechanism is designed to adjust the position of the motor based on feedback so that the desired position is reached eventually. This is illustrated in Figure 8.5 [10, 43]. The system generates a *Command* / *Target* or *Reference* signal $y_0$ which is used to set the desired *Output* $y$ of the system. The *Output* of the system is fed back to the controller as the *Feedback* signal and used to calculate the *Error* signal $e$ (difference between the *Command* value and the *Output* value). The *Error* is then used by the *Proportional* (P) and *Integral* (I) modules to generate the output terms that will be combined with the *Derivative* output term to give the *Update* signal $u$, which adjusts the motor position or other system control mechanism closer to the *Reference* signal $y_0$. Note that there are two ways of generating the *Derivative* term, D and D', with different characteristic response behavior. Classic PID Control [43] calculates the *Derivative* (D) term from the *Error* signal computed previously, which is then *added* to the P and I terms. [10] recommends that the *Derivative* (D') term be calculated using the direct feedback from the output, which is then *subtracted* from the Proportional (P) and Integral (I) terms to generate the control signal. Using D' instead of D results in smoother transitions [10, 43], but the effectiveness of either approach is subject to proper tuning of the various PID parameters.
8.4.2 PID Parameters

The PID Controller is specified using the following parameters [43]:

- Proportional Term, used to handle Present condition, specified using the Proportional Gain $K_P$
- Integral Term, used to recover from Past (historical) behavior, specified using the Integral Gain $K_I$
- Derivative Term, used to anticipate the Future behavior, specified using the Derivative Gain $K_D$

The Ziegler-Nichols Tuning Rule has been defined for optimal tuning of the PID controller [43], based on the critical gain $K_C$ and oscillation period $P_C$. After the tuning is completed, the PID Controller is updated every Sample Period $T$, based on the Time constants $T_P$, $T_I$ and $T_D$, of the respective P, I, and D terms of the system.

8.4.3 Discrete PID Controller

The Atmel Discrete PID Controller Implementation is a simple Integer-based PID algorithm that can easily be applied to 8-bit or higher micro-controllers [43]. Once the Ziegler-Nichols Tuning has been completed, the gains for the various terms are initialized using:

$$K_P = 0.65K_C$$  \hspace{1cm} (8.1)

$$K_I = \frac{K_P T}{T_I} = \frac{K_P T}{0.5P_C}$$  \hspace{1cm} (8.2)
\[ K_D = \frac{K_P T_D}{T} = \frac{K_P (0.12P_C)}{T} \tag{8.3} \]

The real values are converted into integer values of \( \alpha K_P \), \( \alpha K_I \) and \( \alpha K_D \) by multiplying them with a scaling factor \( \alpha \), to increase the resolution of the output. The output of the PID Controller is divided by \( \alpha \) after the calculations are complete to scale them back to the original output range. The output value is calculated using the following formula:

\[
 u_n = \frac{1}{\alpha} \left[ \alpha K_P e_n + \alpha K_I \sum_{k=0}^{n} e_k + \alpha K_D (y_n - y_{n-1}) \right] \tag{8.4}
\]

where \( n \) is the sample number, \( u_n \) is the \( n \)th PID Controller Update, \( e_k = (y_0 - y_k) \) is the \( k \)th error term, and \( y_n \) is the \( n \)th Output term. This formula uses the D’ form of the Derivative control described previously.

### 8.4.4 PID Controller Implementation

In order to implement the PID Controller, various related variables are normally grouped in a data structure. Since Assembly language programming does not provide complex data type support, the data structure has to be defined as elements specified as offsets into a memory buffer (Listing 8.2).

#### Listing 8.2: PID Control Structure Definitions

```assembly
@ 2s Complement macro
.macro com reg
  neg reg, reg
  add reg, reg, #1
.endm

.equ MAX_LONG_PLUS1, 2147483648 @ 32 bit signed
.equ MAX_SHORT_PLUS1, 32768 @ 16 bit signed
.equ SCALING_SHIFT, 16 @ Scaling factor: 65536

@ PID Control data structure offsets
.equ KP, 0 @ Proportional Factor
.equ KI, 4 @ Integral Factor
.equ KD, 8 @ Derivative Factor
.equ ERR_Max, 12 @ Limit for ERR
.equ ERRSUM_Max, 16 @ Limit for ERR_Sum
.equ STEADY_STATE_Thresh, 20 @ Threshold for determining steady state
.equ ERR_Sum, 24 @ sum[e(k)](k=0..n)
.equ Y_Prev, 28 @ y(n-1)
.equ STEADY_STATE_Count, 32 @ Count of error within STEADY_STATE_Thresh
.equ REF_Val, 36 @ Reference (Target / Set Point) value y0
.equ SIZEOF_PID_CONTROL, 40
```

A version of the Integer-based Discrete PID Controller adapted for ARM is given in Listing 8.3. This Discrete PID Controller implements 16-bit resolution for the integer reference, feedback and PID output values. However, since the ARM processor is capable of 32-bit resolution, the initialization values the P, I, and D terms used for \texttt{InitPID()} are actually \( \alpha K_P \), \( \alpha K_I \) and \( \alpha K_D \), which improves the accuracy of the integer-based calculations internally.

200
Listing 8.3: ARM version of Integer-based Discrete PID Controller

@ Discrete PID Controller Implementation
@ Based on Atmel Application Note AVR221
@ and libnxter (http://libnxter.sourceforge.net/)
@ (c) 2011, TC Wan <tcwan@cs.usm.my>
@ Universiti Sains Malaysia
@ This code is licensed under the GNU GPL version 2
@ The inputs and outputs have a resolution of 16 bits.
@ The PID Controller output is defined as:
@ \[ u(n) = KP \times \text{Err}(n) + KI \times \sum_{k=0}^{n} e(k) + KD \times \left[ y(n) - y(n-1) \right] \]
@ \text{Err}(n) = \text{RefVal} - y(n)
@ This Discrete PID uses 32-bit Integers to store the
@ parameters. However, to improve the accuracy of the
@ calculations, each P,I,D term is scaled using the
@ Scaling factor internally.

.include "interwork.h"
.include "pid_equates.h"
.extern ulongdiv

.text
.align 4

InitPID

void InitPID(struct PID_Control *pid, ulong alphaKP, ulong alphaKI,
                      ulong alphaKD, ulong STEADY_STATE_Thresh);

ARM routine to initialize PID Controller
Call once at beginning of program
The P,I,D factors are multiplied
by the Scaling Factor to improve resolution
internally

On Entry:
R0: PID_Control Struct Pointer
R1: alphaKP
R2: alphaKI
R3: alphaKD
Stack: STEADY_STATE_Thresh

On Exit:
R0,R1,R2,R3: Destroyed

.arm_interwork InitPID
push {r4, fp, lr}
add fp, sp, #8
@ FP points to LR

The Scaling should be performed before calling InitPID
Otherwise the less significant bits would be lost

mov r4, #SCALING_SHIFT
lsr r1, r1, r4
lsr r2, r2, r4
lsr r3, r3, r4

mov r4, r0
@ Keep PID Control Struct Pointer in R4

str r1, [r4, #KP]
str r2, [r4, #KI]
str r3, [r4, #KD]
ldr r3, [fp, #4]
@ Retrieve Steady State Threshold
str r3, [r4, #STEADY_STATE_Thresh]

mov r3, #1
add r2, r2, r3
@ R2 = KI + 1 (keep for later)
add r1, r1, r3
@ R1 = KP + 1
rsb   r0, r3, #MAX_SHORT_PLUS1 @ Max Signed Short is 32767
bl    ulongdiv @ R0 = MAX_SHORT / (KP + 1)
str   r0, [r4, #ERR_Max] @ Limit for Error
rsb   r0, r3, #MAX_LONG_PLUS1 @ Max Signed Long is 2147483647
asr   r0, r0, #1 @ MAX_LONG/2
mov   r1, r2 @ Retrieve R1 = KI + 1
bl    ulongdiv @ R0 = (MAX_LONG/2) / (KI + 1)
str   r0, [r4, #ERRSUM_Max] @ Limit for Error Sum

@ Initialize State variables
mov   r0, #0
str   r0, [r4, #ERR_Sum]
str   r0, [r4, #Y_Prev]
str   r0, [r4, #STEADY_STATE_Count]
str   r0, [r4, #REF_Val]
pop   {r4, fp, lr}
bx    lr

@===========================================================
@ SetPIDReferenceVal
@ void SetPIDReferenceVal(struct PID_Control*pid, ulong Reference);
@ ARM routine to set PID Controller output (y) reference (target) value
@ Call to initiate PID control
@ On Entry:
@ R0: PID_Control Struct Pointer
@ R1: Reference (target) Value
@ On Exit:
@ R0: Destroyed
@ R1: Destroyed
@====================================================================
arm_interwork SetPIDReferenceVal
@ Initialize State variables
str   r1, [r0, #REF_Val]
mov   r1, #0
str   r1, [r0, #ERR_Sum]
str   r1, [r0, #Y_Prev]
str   r1, [r0, #STEADY_STATE_Count]
bx    lr

@====================================================================
@ CheckPIDEnd
@ Bool CheckPIDEnd(struct PID_Control*pid);
@ ARM routine to set PID Controller output (y) target value
@ Call to initiate PID control
@ On Entry:
@ R0: PID_Control Struct Pointer
@ On Exit:
@ R0: 0: False, !0: True
@ R1, R2: Destroyed
@====================================================================
arm_interwork CheckPIDEnd
ldr   r2, [r0, #STEADY_STATE_Thresh]
ldr   r1, [r0, #STEADY_STATE_Count]
mov   r0, #0 @ Setup Return Value = False
teq   r2, #0 @ Zero threshold
bneq  lr @ return False
cmp   r1, r2 @ Is STEADY_STATE_Count > STEADY_STATE_Thresh?
movh   r0, #0 @ Yes, Return True
bx    lr

@====================================================================
@ PIDController
@ ulong PIDController(struct PID_Control*pid, ulong SystemStatus);
@ 202
ARM routine to set PID Controller output \( u \) value
Call to initiate next step in PID control

On Entry:
- R0: PID_Control Struct Pointer
- R1: System Status (Feedback) Value \( y(n) \)

On Exit:
- R0: PID Controller Output: \( u(n) \) [16-bit resolution]
- 0 = Steady State reached, \( u(n) \) = \( y(n) \)
- R0,R1,R2,R3: Destroyed

R4: PID_Control Struct Pointer
R5: \( y(n) - y(n-1) \)

```
arm_interwork PIDController
push {r4,r5}
mov r3, #1 @ Used for calculating LIMITS
mov r4, r0 @ Use R4 for PID Control Struct Pointer
ldr r0, [r4, #REF_Val] @ R0: Reference Value
ldr r5, [r4, #Y_Prev] @ R5: \( y(n-1) \)
sub r5, r1, r5 @ R5: \( y(n) - y(n-1) \)
str r1, [r4, #Y_Prev] @ Update Y_Prev with \( y(n) \)
sub r0, r0, r1 @ R0: Error = RefVal - \( y(n) \)
check_steady:
  teq r0, #0 @ Check for Steady State
  bne calc_pid @ Non-zero error, so not steady state
  ldr r2, [r4, #STEADY_STATE_Thresh] @ Non-zero threshold, so increment count
  addne r1, r1, #1 @ Store updated Steady State Count
  strne r1, [r4, #STEADY_STATE_Count] @ Update STEADY_STATE_Count
  cmp r1, r2 @ Is STEADY_STATE_Count > STEADY_STATE_Thresh?
  movhi r0, #0 @ Zero PID output
  bhi done_pid @ Done

calc_pid:
  ldr r3, [r4, #ERR_Max] @ R2: Err_Max
  cmp r0, r3 @ Error > Err_Max?
  rsbgt r2, r3, #MAX_SHORT_PLUS1 @ Yes, Set R2: PTerm to MAX_SHORT
  bgt exit_pterm_calc @ No, so calculate PTerm
  cmp r0, r2 @ Error < -Err_Max?
  bge pterm_calc @ No, so calculate PTerm
  mov r2, #MAX_SHORT_PLUS1 @ R2: MAX_SHORT + 1
  com r2 @ R2: MIN_SHORT
  b exit_pterm_calc @ Else, Set R2: Pterm to MIN_SHORT
  pterm_calc:
    ldr r1, [r4, #KP] @ R1: KP
    mul r2, r0, r1 @ R2 = KP x Error
    exit_pterm_calc:
  @ R2 contains PTerm, Calculate ITerm
    ldr r1, [r4, #ERR_Sum] @ R0: ERR_Sum + Error
    add r0, r1, r0
    cmp r0, r1 @ (ERR_Sum + Error) > ERR_Sum_Max?
    strgt r1, [r4, #ERR_Sum] @ Yes, Set ERR_Sum = ERR_Sum_Max
    asrgt r1, [r4, #MAX_LONG_PLUS1] @ and Set R1: ITerm to MAX_LONG/2
    b exit_iterm_calc @ Set ITerm to MAX_LONG/2
  iterm_calc:
    ldr r1, [r4, #ERR_Sum] @ ERR_Sum = ERR_Sum + Error
    mov r1, #MAX_LONG_PLUS1/2 @ R1: (MAX_LONG + 1) / 2
    sub r1, r1, #0 @ R1: MIN_LONG/2
    b exit_iterm_calc @ Else, set R1: Iterm to MIN_LONG/2
```
mul r1, r0, r1  @ R1 = KI \times ERR_Sum (new)

exit_iterm_calc:
@ R2 contains PTerm, R1 contains ITerm, calculate DTerm
ldr r3, [r4, #KD]
mul r0, r3, r5  @ R0 = KD \times [y(n) - y(n-1)]
@ R2 contains PTerm, R1 contains ITerm, R0 contains DTerm
add r0, r0, r1  @ R0 = DTerm + ITerm
add r0, r2, r0  @ R0 = PTerm + DTerm + ITerm
asr r0, r0, #SCALING_SHIFT
@ Limit to [MIN_SHORT, MAX_SHORT], which is done automatically by the ASR
if we are using SCALING_SHIFT of 16

This version includes Convergence Checking found in the libnxter code [44], which stops the PID Controller if the Error \(e\) has reached zero for a given number of Sample Periods \(m\), for a duration of \(mT\) where the output has converged.

The calculated Error value \(e\) is limited to ERR\_Max, which is set as \([MIN\_SHORT, MAX\_SHORT]\), within the range of 16-bit signed values, since the Update value \(u\) also has 16-bits resolution.

In addition, as the PID Controller executes over several Sample Periods \(kT\) the Integral Error Sum can increase significantly if there is a large difference between the Target value \(y_{0}\) and the current Output value \(y_{n}\). This results in a phenomenon termed Integral Windup, where the PID Update value overshoots the target value. Consequently, the Integral Error Sum has to be range limited to avoid making the PID Controller unstable. This limit is controlled using ERRSUM\_Max, which is currently set as \([MIN\_LONG, MAX\_LONG]\), which is within half of the range of 32-bit signed values. Nonetheless, the value for ERRSUM\_Max should be selected based on the given problem [43].

To generate the update value \(u\), the summed 32-bit calculated P, I, and D Terms are scaled by \(\frac{1}{\alpha}\) to give the final Update value \(u_{n}\), which has 16-bit resolution.

To calculate the appropriate ERR\_Max and ERRSUM\_Max values, the PID Controller uses an Unsigned Long Integer Division algorithm [45], as shown in Listing 8.4.

Listing 8.4: Unsigned Long Division Routine

```assembly
; Division Routine
; Based on 'http://www.tofla.iconbar.com/tofla/arm/arm02/index.htm'
; for Archive Magazine, algorithm written by Harriet Bazley

@ On Entry:
@ R0: Numerator
@ R1: Divisor
@ R2, R3: Preserved

.include "interwork.h"

@ The algorithm uses divides R1 by R2, so instead of
@ rewriting everything, we just move the inputs to
@ the right places
```
The following pseudocode sequence illustrates the use the PID Controller routines given in Listing 8.3:

```
InitPID()
SetPIDReferenceVal(Target_Value)
Repeat each Sample Period T
    Get Feedback_Value
    Output = PIDController(Feedback_Value)
    Set Output
Until CheckPIDEnd()
```

### 8.5 Tribot Robot

This section presents the complete Tribot Robot, which adds the Ultrasound sensor that can be used to detect distances to obstacles or walls. The Behavior Diagram for Tribot is shown in Figure 8.6.

Similar to Mover, the upper layer behavior will override any of the lower layer behaviors (Table 8.8). Idle is always activated, and will play an Idle Tone through the Speaker while active to indicate that Mover is idle. Follow-Line is activated if the Touch Sensor was Inactive and the line type was recognized. If the line type were not recognized, the Follow-Line behavior would be inhibited. This allows the Idle behavior to control the Speaker via its Idle Tone actuator output.
Figure 8.6: Behavior Diagram for Tribot

Table 8.8: Actuator Outputs for Individual Tribot Behaviors

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Claw Controller</th>
<th>Motor Controller</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid-Obstacle</td>
<td>(Inhibited)</td>
<td>Obstacle-Avoiding</td>
<td>Warning Tone</td>
</tr>
<tr>
<td>Move-Object</td>
<td>Close Claws</td>
<td>Sound-Seeking</td>
<td>Silence</td>
</tr>
<tr>
<td>Grasp-Object</td>
<td>Close Claws</td>
<td>Stop Movement</td>
<td>Silence</td>
</tr>
<tr>
<td>Open-Claws</td>
<td>Open Claws</td>
<td>Stop Movement</td>
<td>Silence</td>
</tr>
<tr>
<td>Follow-Line</td>
<td>(Inhibited)</td>
<td>Line-following</td>
<td>Silence</td>
</tr>
<tr>
<td>Idle</td>
<td>(Inhibited)</td>
<td>Stop Movement</td>
<td>Idle Tone</td>
</tr>
</tbody>
</table>
Table 8.9: Activation Triggers for Tribot Behaviors

<table>
<thead>
<tr>
<th>Activation \ Trigger</th>
<th>Touch Inactive</th>
<th>Touch Active</th>
<th>Claw Open</th>
<th>Claw Closed</th>
<th>Sound Input Active</th>
<th>Obstacle Detected</th>
<th>Obstacle Not Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid-Obstacle</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Move-Object</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>-</td>
<td>&amp;</td>
<td>11</td>
<td>-</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Grasp-Object</td>
<td>-</td>
<td>&amp;&amp;</td>
<td>&amp;</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Open-Claws</td>
<td>&amp;</td>
<td>-</td>
<td>11</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Follow-Line</td>
<td>&amp;</td>
<td>-</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Idle</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Legend: ‘.’ N/A  ‘&&’ AND  ‘||’ OR

If no obstacles were detected and the Touch Sensor were inactive, Open-Claws will be activated when the claws are in the closed position. Grasp-Object and Move-Object behaviors are evaluated when no obstacles were detected and the Touch Sensor is Active. If the claws were open, then Grasp-Object is activated. If the claws were closed, then Move-Object is activated, and will perform its Sound-Seeking action to try to move towards the source of the sound as long as no obstacles were detected. Finally Avoid-Obstacle is activated when the Ultrasound Sensor detects any obstacles such as walls or obstructions closer than the trigger distance. If obstacles were detected, it will reverse motion until a specified distance to the obstacle is achieved. If a person was walking towards Tribot, it may even keep backing away from the approaching person.

This is summarized in Table 8.9.

8.5.1 Changes to Existing Behaviors

Avoid-Obstacle inhibits the Claw Motors control since it triggered whenever an obstacle is detected, and does not know whether an object is being moved by Tribot or not when the behavior is triggered. It is possible that an object being grasped by Move-Object will come loose when Tribot is reversing due to the Avoid-Obstacle behavior, causing the Touch input to become Inactive. In such a case, the Open-Claws claw opening action will be activated and Tribot will lose the object. Consequently, to ensure that the Move-Object behavior will continue to grasp any object that has been found, it first checks for the case where an obstacle was detected and just maintain the previously activated behavior outputs, ensuring that the claw actuators remain activated if an object were grasped. This would help to keep a grasped object from coming loose easily. However, the sound input level triggers need to be reset so that when Tribot has cleared the obstacle, the Move-Object behavior does not suddenly reenable the wheel motor actuators which were active previously.

In addition, it is necessary for Grasp-Object and Open-Claws to check for the case where no obstacle was detected before activating the given be-

---

7The ability of Tribot to do so depends on the accuracy of the Ultrasound sensor. Non-uniform obstacles such as a moving person may not provide consistent distance readings to the sensor, so the reverse motion may be erratic.
8.5.2 Obstacle Avoidance using PID Controls

As discussed in Section 8.4, the use of PID Controls is common for movement regulation. We will use this approach in Tribot for controlling the reverse speed of the wheel motors for the Avoid-Obstacle behavior. Nonetheless, we will not be using the tachometer readings from the motor as feedback to the PID Controller unlike what was shown in Figure 8.5. Instead, we will use the Ultrasound readings as the feedback values since that is the actual parameter that we are interested in. Consequently, there is an indirect relationship between the PID Controller output which drives the motor and the feedback signal used to drive the PID Controller (Figure 8.7).

8.5.2.1 Distance Input Pre-processing

The raw input from the Ultrasound Sensor consists of distance readings, with the default values given in centimeters. The 8-bit input values ranges from 0, which indicates no object found, to 255, which indicates an invalid or erroneous reading. There needs to be a delay between obtaining consecutive readings from the Ultrasound Sensor. Since NERF implements a duration of 100 ms per loop for the main loop, we can acquire one reading...
every loop safely. In addition, the raw input stream can have random noise where invalid readings are received from time to time. In order to prevent this random noise from affecting the PID Controller, pre-processing of the Ultrasound Sensor inputs is done. Readings of value 0 indicating no object detected is filtered out, while invalid readings of value 255 will cause the input value to become 255 only when a number of consecutive of 0 or 255 values were detected.

8.5.2.2 PID Controller Operation

The PID Controller is triggered when the distances returned by the Ultrasound Sensor is less than OBSTACLE_MINDIST. Once the minimum distance threshold is breached, the PID Controller is initialized to the target distance value of OBSTACLE_STOPDIST, which is set to be greater than OBSTACLE_MINDIST. Once initialized, the PID Controller will be activated each time through the NERF main loop, which programs the wheel motor actuators to move Tribot until the position of Tribot converges on the target distance from the obstacle.

Obviously the PID Controller needs to have its control parameters initialized appropriately. For Tribot, we will just choose some basic values to illustrate the behavior of the PID controller. Given:

\[ \alpha = 65536, \quad K_C = 3, \quad T = 100 \text{ ms}, \quad P_C = 1 \text{ s} \]  

We get:

\[ \alpha K_p = 65536 \times 0.65 K_C = 127795 \]  \hspace{1em} (8.6)

\[ \alpha K_i = \frac{\alpha K_p T}{0.5 P_C} = \frac{127795 \times 0.1}{0.5 \times 1} = 25559 \]  \hspace{1em} (8.7)

\[ \alpha K_D = \frac{\alpha K_p (0.12 P_C)}{T} = \frac{127795 \times 0.12 \times 1}{0.1} = 153354 \]  \hspace{1em} (8.8)

These values will be used in configuring the PID Controller for Tribot.

8.5.2.3 Motor Actuator Output Clamping

Since the PID Controller generates outputs based on its own algorithm, the output values may exceed the safe operating speeds for Tribot. Consequently, the PID Controller output needs to be converted into a suitable motor actuation value before being stored in the actuator_state entry for the Avoid-Obstacle behavior. The range of motor actuation values is clamped to [WHEEL_SPEED_MIN, WHEEL_SPEED_MAX] in order to avoid running the wheel motors at too high a speed. From prior experience with Tribot, safe values were found to be in the range of \([-30, 30]\), but this may be adjusted depending on the actual operating conditions.

The data processing flow for the Avoid-Obstacle behavior is illustrated in Figure 8.8.

\footnote{Unofficial information regarding the minimum interval for reading the Ultrasound Sensor is at least 50 ms between readings.}
Figure 8.8: Data Processing Flow for *Avoid-Obstacle* Behavior
8.5.3 Code Size Reduction via Thumb Interworking

Two versions of Tribot were developed, namely tribot-arm and tribot-thumb. The tribot-arm application code was completed and functionality tested first before the process of converting the code to Interworked Thumb code started. This is to ensure that the difference between the two approaches was due mostly to instruction to object code conversion and not differences in algorithms. Nonetheless this is only a basic conversion process, it does not reflect the maximum savings that can be achieved. In addition, the underlying NxOS-Armdebug kernel code, which forms a major part of the overall application, remains as Interworked ARM code.

8.5.3.1 Changes To Existing Source Code

The immediate impact of switching to Thumb object code generation is the type of instructions that are available. Complex instructions such as MLA need to be converted into two Thumb instructions, i.e., MULS followed by ADDS. In addition, normal instructions have to have the ‘s’ suffix enabled regardless of whether the Condition flags in the Status registers need to be updated or not. This may impact complex ARM coded logic sequence that involve modification of register contents without affecting previously determined Condition flags.

All ARM conditional execution code in the source file would need to be converted into the equivalent conditional branch and unconditional execution code blocks. While this is not difficult, it does allow errors to easily creep into the code when incorrect branch conditions are specified. Another source of errors due to the limited range of constants that can be specified as instruction operands. This may require replacing those instructions with one or more equivalent instructions.

If constants with large values are required as operands in the code, such as address of variables used as pointer initialization values and 32-bit negative values, these operands need to be placed in literal pools close to the instruction which uses them. Multiple .ltorg Assembler directives need to be inserted into the source file at appropriate places to ensure that they are within the offset range of the instruction. Nonetheless, it is important to not use the directives liberally since it takes up object code space and may consume padding bytes needed to maintain half-word and word alignment for instruction addresses of Thumb and ARM instructions.

8.5.3.2 Code Size Comparison

The data in Table 8.10 gives one example of the different code sizes between ARM only code vs. Thumb generated code. Both output files were generated by modifying the LUT definitions from .text to .data, to determine the differences due to actual executable machine code. Normally the LUT definitions would have been declared as .text to indicate that they are static.

---

9E.g., additional size savings could be obtained by eliminating word-aligned routine start addresses for Thumb routines, disabling interworking-safe subroutine return for Thumb routines called only from other Thumb routines, etc.
Table 8.10: Code Size Comparison for Interworking Applications

<table>
<thead>
<tr>
<th>ARM-only Code</th>
<th>Interworked Thumb Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tribot-arm_rxe.elf)</td>
<td>(tribot-thumb_rxe.elf)</td>
</tr>
<tr>
<td>section</td>
<td>size</td>
</tr>
<tr>
<td>.rxe_header</td>
<td>16</td>
</tr>
<tr>
<td>.rxe_init</td>
<td>80</td>
</tr>
<tr>
<td>.vectors</td>
<td>56</td>
</tr>
<tr>
<td>.data</td>
<td>1556</td>
</tr>
<tr>
<td>.text</td>
<td>27744</td>
</tr>
<tr>
<td>.bss</td>
<td>1840</td>
</tr>
<tr>
<td>.stack</td>
<td>1672</td>
</tr>
<tr>
<td>Difference: N/A</td>
<td>Difference: 472 bytes (1.7% Reduction)</td>
</tr>
</tbody>
</table>

values not meant to be modified. Nonetheless, this does not significantly affect the calculated code size difference.

From Table 8.10, the percentage reduction in code size by using Thumb code generation seems to be a paltry 1.7%, corresponding to 472 bytes saved. However, in embedded systems such as the NXT which has only 64KB of RAM, a savings of 0.5KB could mean the difference between being able to implement the required functionality vs. having problems due to stack overflows or being unable to meet all the specified system requirements. If the kernel code for NxOS-Armdebug were to be converted to be mostly Interworked Thumb generated code\footnote{It is not possible to have fully Thumb object code since Exception handlers such as Reset and Interrupt Handlers need to be invoked in ARM state. In addition, complex algorithms involving many variables may not benefit from Thumb object code generation due to the limited number of usable registers in Thumb state.}, then the reduction would be even more significant.

When we look more carefully in the individual application level object file sizes, we see the following results (Table 8.11):

Both division.o and pid.o contain purely Interworked ARM routines, so there are no savings in code size. However, helper-routines.o and tribot-thumb.o have significant code size savings due to the generation of Thumb object code. The total object code size reduction is about 20%, which results in an effective savings of about 11% due to the addition of Linker Veneers for direct interworking calls between ARM and Thumb code.

Nonetheless, the Lookup Table driven design for Tribot means that a lot of similarly structured code has already been replaced by LUT entries. This earlier code optimized design makes the impact of subsequent code size reduction less dramatic in consequence.

### 8.6 Chapter Summary

- Thumb state instructions were described and presented.

212
Table 8.11: ARM vs. Thumb Object File Sizes

<table>
<thead>
<tr>
<th>Filename</th>
<th>ARM-only Code Size (.text)</th>
<th>Interworked Thumb Code Size (.text)</th>
<th>Difference: Bytes (% Reduction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>division.o</td>
<td>76</td>
<td>76</td>
<td>N/A</td>
</tr>
<tr>
<td>helper-routines.o</td>
<td>176</td>
<td>112</td>
<td>64 (36.4%)</td>
</tr>
<tr>
<td>pid.o</td>
<td>416</td>
<td>416</td>
<td>N/A</td>
</tr>
<tr>
<td>tribot-(arm,thumb).o</td>
<td>3792</td>
<td>2944</td>
<td>848 (22.4%)</td>
</tr>
<tr>
<td>Total</td>
<td>4460</td>
<td>3548</td>
<td>912 (20.4%)</td>
</tr>
<tr>
<td>Effective Savings</td>
<td>4460</td>
<td>3988</td>
<td>472 (10.6%)</td>
</tr>
</tbody>
</table>

- Interworking involves declaring required routines as .global functions, where Thumb routines are also declared as .thumb_func. The linker will automatically generate a veneer to switch modes as appropriate.
- Interworked routines need to return to the calling routine via ‘BX LR’ or ‘BX Rx’ instead of loading LR into PC directly.
- A Basic PID Control Algorithm were presented for the ARM microcontroller.
- The Tribot implementation was discussed.
- Thumb code generation can reduce the footprint of object code, but this requires careful management of Interworked ARM library and kernel routines to realize significant savings.

8.7 Review Questions and Problems

1. Discuss the tradeoffs for using Thumb state vs. ARM state in writing programs.

2. Compare various approaches to Closed Loop Control and how easy it is to implement in a microcontroller system such as the Atmel AT91SAM7S series microcontroller used in the NXT.

3. Modify the PID Controller to use Fixed Point Arithmetic for better accuracy.

4. PID Controller tuning is considered an art. Try to improve the convergence properties of the PID Controller used in Mover by tweaking the initialization values used.
Chapter 9

Sensing the Environment

‘Well! I’ve often seen a cat without a grin,’ thought Alice;
‘but a grin without a cat! It’s the most curious thing I ever saw in
all my life!’

from “Alice’s Adventures in Wonderland,” Lewis Carroll, 1832-
1898

We have been focusing on the CPU and software architecture in
earlier Chapters. Nonetheless, Tribot consists of many addi-
tional components other than the NXT controller. Apart from
the basic sensors that were introduced previously, there are
also motors, servos and advanced sensors that can be used to enhance the
capabilities of Tribot. All these components are termed peripherals. While
microprocessors perform the basic processing functions on data, it is not
possible for the microprocessor to control the function of peripherals such
as the output power of the servo motor or receive input from the various
sensors without suitable electronic interfacing circuits.

Therefore, peripherals perform Input and Output (I/O), and require spe-
cific ways to exchange information between the peripheral device and the
ARM processor in the NXT. This involves mechanical, electrical and mes-
 sage exchange standards to be defined and followed. The process of inter-
connecting the peripheral device to the CPU is called Interfacing. While
mechanical and electrical interfacing are important issues, they have been
addressed by LEGO by means of standardized plugs and sockets (based on
the RJ-45 standard), as well as defined electrical connection for how various
peripherals should be connected to the NXT. Consequently we shall focus
on the software and programming issues related to I/O Interfacing.

This differs from pure data processing in that it requires a means to
accept input from the environment and to create output that can affect the
environment. For example, to follow a marked path on the ground, Tribot
must have light sensors that can capture the intensity of light from the
LED reflected by the ground, and then process those inputs using suitable
algorithms to determine whether it is on the path or not. Subsequently, it
would need to adjust the movement of the motors to shift the direction of
movement to keep Tribot on the right path.
9.1 MINDSTORMS Central Nervous System

The MINDSTORMS NXT is an example of an Embedded or Microcontroller-based System. As the Central Nervous System of the Robot, it has various built-in I/O devices, but it also depends on various external I/O peripherals (sensors and motors) connected to the NXT brick via connector cables. We will study them in greater detail in Chapter 9.5.

The most important component of the NXT is the ARM Microcontroller. Due to the limited space and size of the NXT, it is often not possible to put a processor, ROM, RAM, and separate device interfacing logic into the given dimensions. Microcontrollers were developed to address this problem. As described previously in Figure 1.2, Microcontrollers often have non-volatile memory (EPROM/EEPROM/Flash) for storing executable programs, volatile memory (SRAM) for program and algorithm execution, as well as peripheral I/O logic built into the same device. This saves on both package space as well as interconnection requirements, since logic blocks on the same device can be interconnected using conducting paths instead of physical wires. In order to better understand the operation of the NXT, we will examine typical features of microcontrollers in greater detail.

9.2 Common Microcontroller Features

As discussed in Chapter 1.5, all microcontrollers have the following features in a single device:

- Microprocessor Core
- Permanent (non-volatile) Memory: e.g., ROM, EPROM, EEPROM, Flash
- Non-permanent (volatile) Memory: e.g., SRAM, DRAM
- Interrupt Support Logic: This provides support for external interrupt signals from various I/O devices
- I/O Peripheral Logic: This is where the differences between different microcontrollers arise, even within the same processor family.

We have already studied the features and characteristics of Microprocessor cores in Chapter 2.3 and specifically the ARM CPU in Chapters 2.4 and 3.3. The specific combinations of the size and type of Permanent and Non-permanent memory, as well as what type of I/O Peripheral Logic is provided for a given microcontroller is specific to the device model for a given manufacturer. The datasheets of the microcontroller of interest will provide both the electrical interfacing details, as well as the programming interfaces necessary for controlling the respective I/O peripheral logic for interfacing to the various supported devices.

9.3 I/O Peripheral Logic Modules

Since there is a wide variety of I/O peripherals that are needed in different embedded systems applications and markets, it is not possible to create an
exhaustive list of I/O Peripherals that can be included. Nonetheless, the most commonly available peripheral logic modules include:

- Liquid Crystal Display (LCD) and external display controllers
- Keyboard Scanner
- Resistive/Capacitive Touchpad Controller
- Parallel I/O ports, General Purpose I/Os (GPIO)
- Serial Communications Interface (SCI), Universal Asynchronous Receiver Transmitter (UART)
- Serial Peripheral Interface bus (SPI), Inter-IC bus ($I^2C$)
- Timers, Pulse Measurement (Input Captures), Pulse Generation (Output Compares) and Pulse Width Modulators (PWM)
- Analog-Digital Convertor (ADC), Digital-Analog Convertor (DAC)
- Universal Serial Bus (USB)
- Real Time Clock (RTC)
- Coprocessors: Floating Point Unit (FPU), Digital Signal Processor (DSP), etc.
- Programmable Glue Logic: Programmable Gate Arrays (PGA), etc.

All the I/O Peripherals are packaged in specific microcontroller models depending on the expected target application of the given microcontroller. For example, microcontrollers used in digital cameras would probably come with built-in LCD display controllers, Real Time Clocks, USB, and even advanced DSPs for image compression and decompression; whereas microcontrollers used for air-conditioner controls would be expected to have timers, PWM, DACs, and less likely to have Keyboard Scanners and USB peripheral logic. This differentiation is mainly due to the cost sensitive nature of microcontrollers. Ideally a given microcontroller for a specific target application would have only those peripheral logic modules that are required to implement the needed functions, since unused I/O logic modules take up die area on the device, wastes energy and incurs a higher cost to the embedded system designer.

9.3.1 Liquid Crystal Display (LCD) / External Display

LCD Displays are increasingly becoming common output devices for portable equipment due to its lightweight and relatively low power consumption. Built-in support for driving LCD displays would enable a microcontroller to interface to 7-segment LCD panels without the need for an additional LCD controller module. This is often used in cost-sensitive equipment since driving a dot-matrix LCD display requires significant processing overhead. External Display Controllers to support digital display standards, such as DVI and HDMI, are also increasingly common in modern microcontrollers.
9.3.2 **Keyboard Scanner**

A keyboard scanner allows the microcontroller to accept input from resistive membrane keyboards and keypads (similar to that found on cellular phones and calculators), where pressing a key on the keyboard would close a circuit laid out on an x-y grid. The key closure is converted into a hexadecimal value (keycode) that is used by the processor to determine which key has been pressed.

9.3.3 **Resistive/Capacitive Touchpad Controller**

Modern devices such as smartphones and tablet are often touch driven. Two types of touch sensors are in common use. Resistive touchpad controllers detect a change in resistance when a stylus is pressed on the sensing surface, which is typically found on top of the LCD display screen. Capacitive touchpad controllers detect a change in resistance when an object such as a finger contacts the sensing surface. Advanced touchpad controllers can detect multiple points of contact to perform advanced multi-touch gesture decoding.

9.3.4 **Parallel I/O ports, General Purpose I/Os (GPIO)**

Parallel I/O ports are often used to provide parallel transfer of data between the microcontroller and external devices, as well as to support the control of various discrete input and output ICs. For example, one of the output lines from a Parallel port can be used to control a relay for a motor, while another is used by a external switch to indicate a user selection (on/off). Very often the I/O ports are used independently of each other and consequently are known as GPIO lines instead of parallel I/O lines.

9.3.5 **External Serial Communications**

There are various external serial communications interface standards. External Serial communications interfaces typically adopt the RS-232 standard. RS-232 connections are commonly used to connect portable devices to personal computers. Consequently, a built-in serial interface driver would support the full-duplex transmission and reception of data in RS-232 format. This typically requires the addition of a line-driver IC to convert TTL or CMOS voltage levels to RS-232 levels (-12V to +12V). The peripheral providing this interface is commonly termed as a Serial Communications Interface (SCI), or a Universal Asynchronous Receiver Transmitter (UART) interface.

9.3.6 **Internal Serial Communications**

A microcontroller is often used to control other ICs that implement specific functions in a device, such as a Radio Frequency Tuner circuit, external DSP processor, and other mixed-analog devices that cannot be placed on the same die as the microcontroller. Internal serial communications with these devices is performed using a synchronous serial interface to reduce
the number of data lines that are required for interfacing compared to interfacing via the system bus. Two signal lines (transmit and receive) are often used in Serial Peripheral Interface bus (SPI) and Inter-IC bus (I\(^2\)C) shared serial device bus to provide access to multiple devices using a simple device addressing scheme.

### 9.3.7 Event Measurement and Control

Timers are used to determine the frequency of digital input pulses using Input Captures, as well as generate variable duty-cycle digital output pulses using Output Comparates and Pulse Width Modulators (PWM). They are often used in timing critical applications such as engine valve timing and stepper motor control. Generally, these events cause Interrupts to be generated, to enable the CPU to service the events without using excessive computational time for checking the inputs and outputs continuously.

### 9.3.8 Analog and Digital Conversion

Analog-Digital Convertors (ADC) are used by a microcontroller to determine the level of an input signal and perform sampling. Converse, Digital-Analog Convertors (DAC) converts digital samples into an output signal. This can be used in telemetry and remote sensing applications, for example.

### 9.3.9 Universal Serial Bus (USB)

USB interfaces are used for external connectivity. USB interfaces are becoming very common in personal computers, often replacing legacy ports such as serial and parallel interfaces. Consequently, portable devices that interface to PCs such as digital cameras, scanners and printers now require a USB interface. Some microcontrollers have built-in USB support to cater to this requirement.

### 9.3.10 Real Time Clock (RTC)

A Real-Time Clock is often used by a portable device such as a Video Cassette Recorder (VCR) to perform actions based on a specific date and time. Built-in RTCs would enable such devices to support these requirements without the use of external RTC modules. However, RTC features require a backup battery in order to keep correct time and date information.

### 9.3.11 Coprocessors

Advanced microcontrollers often contain additional coprocessors such as a Floating Point Unit (FPU), a Digital Signal Processor (DSP), as well as Vector Processors on the same die in order to reduce the number of interface lines and reduce power consumption.
9.3.12 Dedicated Hardware-based Codecs

Various tasks such as the encoding and decoding of video and audio require the use of complex processing algorithms. While these tasks can be performed using the coprocessor, it uses up a significant amount of energy to do so. Dedicated hardware circuits for performing these tasks require much lower energy usage, and frees up the CPU for other tasks.

9.3.13 Programmable Glue Logic

More recent microcontrollers may come equipped with reprogrammable gate array logic to allow for dynamic reconfiguration of logic blocks to support different processing requirements. For example, checksum calculation that is performed using gate array logic can be reprogrammed when a new algorithm is implemented. Devices that provide this capability often started off as Programmable Gate Arrays (PGA) or Programmable Logic Devices (PLD) that incorporate a processor core on-chip, rather than as microcontrollers that included a PGA or PLD block.

9.4 Atmel AT91SAM7S I/O Blocks

The Atmel AT91SAM7S256 used in the NXT is an ARM microcontroller with 256 KB of Flash and 64 KB of RAM, running at 48 MHz [11]. The integral peripherals used by the ARM microcontroller to communicate with the major subsystems in the NXT are the Two Wire Interface (I²C) and SPI buses as well as the Universal Synchronous/Asynchronous Receiver Transmitter (USART) interface. In addition, the USB port is used to interface with external systems such as a PC in slave mode. The various peripheral I/O blocks are illustrated in Figure 9.1.

9.4.1 Debug & Test Interfaces (JTAG/ICE & DBGU)

There are two debugging interfaces available on the Atmel AT91SAM7S microcontroller. The built-in Joint Test Action Group/In Circuit Emulator (JTAG/ICE) interface is a hardware interface which provides hardware controller via a JTAG interface module, whereas the Debug Unit (DBGU) is a dedicated serial UART interface that can be used for debugging and software downloading. The DBGU UART interface is separate from the default USART and provides access to Chip Identifier data and other debugging capabilities.

9.4.2 Reset Controller (RSTC)

The Reset Controller manages all the hardware and software resets for the CPU. Some RESET sources can be masked (does not trigger a reset). The CPU can be reset under software control if the correct KEY has been programmed into the RSTC. In addition, the cause of a RESET is also stored in the RSTC for checking via software.
Figure 9.1: Atmel AT91SAM7S I/O Blocks, courtesy of Atmel Corp. [11]
9.4.3 Real Time Timer (RTT)

This is a 32-bit seconds counter that is controlled by the Slow clock input, which can be used for generating time triggered (alarm) interrupts and for calculating elapsed wall clock time.

9.4.4 Periodic Interval Timer (PIT)

The Periodic Interval Timer (PIT) is usually used by operating systems to provide a programmable periodic interrupt signal to drive the process scheduler. This is typically the highest priority timer for the entire system.

9.4.5 Watch Dog Timer (WDT)

The Watch Dog Timer (WDT) is a peripheral that can be programmed to reset the CPU if it is not updated within a specific time interval. This prevents the system from locking up due to some logic error which prevents the WDT from being updated.

9.4.6 Voltage Regulator Mode Controller (VREG)

The Voltage Regulator Mode Controller (VREG) sets the system to either Standby (Low Power) or Normal mode.

9.4.7 Memory Controller (MC)

The Memory Controller (MC) controls the address mapping of the Flash and SRAM blocks, to map either the beginning of Flash Memory to low memory addresses (Address 0x0, which is the Exception Vector Table address), or else the SRAM block to the low memory addresses. This allows for execution of RAM-based applications which override the Flash Exception Vectors.

9.4.8 Embedded Flash Controller (EFC)

The Embedded Flash Controller (EFC) is actually a part of the Memory Controller (MC). It manages access to Flash Memory, especially for Thumb mode. In addition, it controls the writing and erasing of the Flash memory blocks.

9.4.9 Fast Flash Programming Interface (FFPI)

The Fast Flash Programming Interface (FFPI) is a separate hardware interface for high speed programming of the on-board Flash using an external Flash programming, bypassing the software programming done via the EFC. This is typically used only during the embedded system manufacturing process for in-factory firmware programming.
9.4.10 Peripheral DMA Controller (PDC)

The Peripheral DMA Controller (PDC) manages direct access to memory by the respective I/O peripheral. This avoids the ARM processor intervention, reducing the load on the CPU. Each I/O peripheral that has PDC support has its individual send and receive channels (PDC Channel 0 and PDC Channel 1) for bi-directional DMA.

9.4.11 Advanced Interrupt Controller (AIC)

The Advanced Interrupt Controller (AIC) provides 8-level prioritized, maskable, and vectored interrupt support to up to 32 interrupt sources. Typically these sources are I/O interrupts from either internal or external peripheral devices. The vectored interrupt support in the AIC speeds up interrupt processing since the CPU’s default interrupt vector is not able to distinguish which of the interrupt sources among the candidate devices triggered an interrupt. the AIC stores the interrupt vector belonging to a particular interrupt source to enable the CPU to access the correct interrupt handling routine quickly.

9.4.12 Clock Generator

The Clock Generator contains the hardware circuitry for generating the Main Oscillator and Slow Clocks used to drive the rest of the clock signals for the microcontroller.

9.4.13 Power Management Controller (PMC)

The Power Management Controller (PMC) and Clock Generator provides the Processor Clock (PCK), Master Clock (MCK) and USB Clock (UDPCK), peripheral clocks for all internal peripherals equipped with independent controls, as well as three programmable clock outputs. The PMC is driven by the Clock Generator block which is a non-programmable hardware block providing the various clock signals needed by the PMC.

9.4.14 Parallel Input/Output (PIO) Controller

The Parallel Input/Output (PIO) Controller manages miscellaneous input and output lines of the microcontroller. These input and output lines can be configured independently, and controlled either in groups (for a parallel interface), or as individual I/O lines.

9.4.15 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) provides a synchronous high speed four-wire interface bus for various external peripheral devices such as a LCD panel or a communications module. SPI was popularized by Motorola (now Freescale) for their microcontrollers.
9.4.16 Two Wire Interface (TWI)

The Two Wire Interface (TWI) enables the microcontroller to control external peripherals using the Inter-IC bus (I²C) communications protocol using only two signal wires. I²C was popularized by Intel for their microcontrollers.

9.4.17 Univ. Synch. Asynch. Receiver Transceiver (USART)

The Universal Synchronous Asynchronous Receiver Transceiver (USART) implements bi-directional serial communications for either synchronous or asynchronous communications. Typically it is used to interface to EIA-232 standard serial interfaces.

9.4.18 Synchronous Serial Controller (SSC)

The Synchronous Serial Controller (SSC) is usually used to interface to telecommunications equipment via synchronous audio or telecoms protocols such as I2S.

9.4.19 Timer Counter (TC)

The Timer Counter (TC) provides three timers for use in generating time-based interrupts, event counting, or simulate a Pulse With Modulator (PWM) output.

9.4.20 Pulse Width Modulator (PWM) Controller

The Pulse Width Modulator (PWM) Controller is used to generate waveforms with a given duty cycle.

9.4.21 USB Device Port (UDP)

The USB Device Port (UDP) implements the USB 2.0 Full Speed Device Specification to enable interfacing to a USB 2.0 Host.

9.4.22 Analog to Digital Converter (ADC)

The Analog to Digital Converter (ADC) accepts analog input signals and converts it to a 10-bit binary value based on the magnitude (level) of the analog input. It can support up to 8 Analog Input Lines.

9.5 NXT Hardware Architecture

We will use our human brain as an analogy to the organization of the NXT Hardware Architecture. The Main Processor (ARM7 microcontroller), is the Cerebrum (cortex), responsible for the higher order brain functions such as thinking and decision making. The Co-processor (AVR microcontroller), is
the Cerebellum (little brain), which is responsible for coordination of movement via the motors, among other tasks. Both the main processor and co-processor may be involved with receiving inputs from sensors. The division of task is based on whether the sensor is digital or analog. Digital sensors are controlled directly by the main processor, whereas analog sensors are handled by the co-processor.

The NXT has a set of four input buttons managed by the AVR co-processor. Two actuators (output devices) are also found in the NXT. These are the LCD Display and the Speaker, and are controlled by the ARM microcontroller. Finally, two communications interfaces, USB and Bluetooth, are also included in the NXT and controlled by the ARM microcontroller. The Hardware Architecture LEGO MINDSTORMS NXT is given in Figure 9.2 [12].

9.5.1 NXT Built-In Peripherals
The Atmel AT91SAM7S256 microcontroller which forms the main CPU for the NXT [11], is interconnected to various other peripheral devices in order to perform its functions. The major peripheral blocks found in the NXT are as follows [12]:

- AVR Coprocessor
- Motor Output Ports
- Sensor Input Ports
• Liquid Crystal Display
• Bluetooth Interface
• Universal Serial Bus Interface
• Speaker Control

9.5.1.1 AVR Coprocessor

The main peripheral device is the Atmel AVR I/O coprocessor, connected via the ARM \textit{I^2C} bus, which is used to detect pressed on the four buttons in front, drive the Motor Actuator Outputs (Ports A, B, C), perform Analog to Digital Input Conversion for Analog Sensor Inputs (Ports 1, 2, 3, 4), as well as provide power for the NXT Motors and Sensors (Ports 1, 2, 3, 4 and A, B, C). The point-to-point ARM-AVR \textit{I^2C} bus connection operates at 380 kbps which provides near to the maximum available throughput defined for \textit{I^2C} connected devices.

The ARM and AVR coprocessor exchange messages every 2 ms. This message exchange has the highest priority in terms of external I/O processing, and is set to interrupt the CPU with the second highest priority via the AIC, while the highest priority is assigned to the internal PIT Interrupt which controls the system tick. In order to keep the NXT system alive, the ARM must transmit a specific startup message sequence to the AVR within a predefined time limit, otherwise the AVR coprocessor will shut down the power to the NXT board.

Since the update interval for ARM-AVR messages is 2 ms, this means that the shortest possible interval for updating Motor Control outputs is also 2 ms.

In addition, the AVR coprocessor controls access to the Analog Sensor A/D conversion results. These results are updated every 3 ms, due to the limitation of legacy RIS Sensors which require the given sensor update timing. Consequently, polling of Analog Sensor input values should not exceed the 3 ms minimum duration requirement.

9.5.1.2 Motor Output Ports

The AVR Coprocessor controls the power output to the three Motor Output Ports. The sustained current output is 700 mA, while the peak current output is 1A. The output signal is controlled by the Pulse Width Modulator (PWM), with the ability to brake the motors or let them coast in between active signal output levels. Two tachometer pulses from each of the motors connected to Output Ports A, B and C are connected directly to the ARM microcontroller. These tachometer readings are used to determine direction of rotation and angle of rotation of the motor.

9.5.1.3 Sensor Input Ports

The four Sensor Input ports, Ports 1, 2, 3, and 4, are able to accept both Analog and Digital sensors. The 10-bit A/D Converters in the AVR coprocessor are used to sample the input level on the Analog input pin of the sensor
Two digital I/O ports are provided for use either as GPIO controls for Analog sensors, or as a software driven 9.6 kbps \( I^2C \) bus for communicating with Digital sensors\(^1\). Since the Sensor Port \( I^2C \) protocol is implemented using software-controlled GPIO signals on the ARM microcontroller, it normally operates at 9.6 kbps, though it is possible to operate sensors at higher \( I^2C \) speeds using custom firmware.

In addition, Port 4 is multiplexed with the RS-485 Serial Interface. This may cause problems with some Digital sensors due to the different implementation of Port 4 \( I^2C \) in terms of the circuitry used. The NxOS-Armdebug uses Open-Drain configuration on the Data line for two way communications. Once the ARM processor pulls the Data line High, the sensor will toggle the line High and Low in order to transmit a message back to the ARM processor. On the other hand, the standard NXT firmware configures the Data line as Push-Pull, where the Data line switches from being an Output port to an Input port for data transmission from the sensor to send messages to the ARM processor. Consequently, the use of Port 4 for Digital sensors may not be 100% reliable.

### 9.5.1.4 Liquid Crystal Display

The 100 (H) x 64 (W) pixel graphical LCD Display is connected to the ARM microcontroller via the SPI bus. The LCD is completely refreshed every 17 ms. Each byte in the LCD display bitmap corresponds to 8 pixels vertically. The NxOS-Armdebug implements 8 x 5 pixel sized character fonts. Consequently we can display text characters by storing each character bitmap in a 5-byte data structure, where each byte represents a column of the character bitmap. To display a character on screen, the LCD driver will use the character as an array index to determine the item address, and copy the 5 bytes to the graphical display buffer at the correct coordinate address.

### 9.5.1.5 Bluetooth Interface

The Bluetooth Interface is implemented using the CSR Bluecore\(^{TM}\) module. The Bluetooth Interface supports the Serial Protocol Profile (SPP). The SPI bus is used to interface with the module for firmware updates, but this is a reserved operation not normally accessible to user programs. Up to three slave NXT devices can be controlled via the Bluetooth Interface. In normal operation, the UART running at 460.8 kbps is used for data exchange between the microcontroller and the Bluetooth Interface.

### 9.5.1.6 Universal Serial Bus Interface

The USB Interface is directly connected to the ARM microcontroller. This is a USB 2.0 Full Speed interface running at 12 Mbps. Four End Points (EP) are available. EP0 is the bidirectional control channel. EP1 and EP2 are unidirectional bulk data pipes used to provide two-way communications between the PC and the NXT. EP3 is not used in the NXT.

\(^1\)This software driven Sensor Port \( I^2C \) bus is separate from the ARM hardware-based \( I^2C \) bus which is a dedicated link to the AVR I/O coprocessor.
Table 9.1: Common Actuators and Sensors for NXT

<table>
<thead>
<tr>
<th>Actuator/ Sensor</th>
<th>Vendor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servo Motor</td>
<td>LEGO</td>
<td>Motor with Servo Control/Feedback</td>
</tr>
<tr>
<td>Touch</td>
<td>LEGO</td>
<td>Analog, activated when pressed</td>
</tr>
<tr>
<td>Light</td>
<td>LEGO</td>
<td>Analog, measures reflected light intensity</td>
</tr>
<tr>
<td>Sound</td>
<td>LEGO</td>
<td>Analog, measures volume</td>
</tr>
<tr>
<td>Ultrasound</td>
<td>LEGO</td>
<td>Digital, measures distance</td>
</tr>
<tr>
<td>Color</td>
<td>LEGO</td>
<td>Available in NXT 2.0 Retail Kit</td>
</tr>
<tr>
<td>Color</td>
<td>HiTechnic</td>
<td>Different hardware from LEGO Color Sensor</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>HiTechnic</td>
<td>Determines acceleration</td>
</tr>
<tr>
<td>Magnetometer</td>
<td>HiTechnic</td>
<td>Compass</td>
</tr>
<tr>
<td>Gyroscope</td>
<td>HiTechnic</td>
<td>Determines direction and motion</td>
</tr>
<tr>
<td>Zigbee</td>
<td>Dexter Industries</td>
<td>Zigbee 802.15.4 WPAN Module</td>
</tr>
</tbody>
</table>

9.5.1.7 Speaker Control

The Speaker is interfaced to the PWM output of the ARM microcontroller via an audio driver circuit. The standard NXT firmware uses the PWM output to control the frequency of the generated tone. In NxOS-Armdebug the SSC is used to control the speaker output signal instead. Sampled audio can also be converted into a PWM output waveform for playback through the speaker.

9.5.2 NXT External Peripherals

External Peripherals refer to any device, sensor or actuator / motor that can be attached to the Input Sensor Ports 1, 2, 3, and 4 as well as Output Actuator Ports A, B and C. Sensors can be Analog, which returns an analog A/D value to the NXT via the AVR I/O coprocessor, or Digital, which are controlled directly by the ARM microcontroller in the NXT, using the software driven Sensor Port I^2C protocol for communications.

Other peripheral devices may also be connected to the RS-485 high speed serial interface on the ARM microcontroller via Sensor Port 4. A list of common sensors available from LEGO and popular third party vendors are given in Table 9.1.

9.6 Chapter Summary

- Common peripheral controls are placed on the same die as the processor to form single chip microcontrollers for portable devices.
- The various types of microcontroller peripherals are described.
- An Embedded System typically uses synchronous serial connections for interfacing with external peripheral devices.
- The hardware architecture of the NXT brick is highlighted.
9.7 Review Questions and Problems

1. Determine the characteristics of the SPI and \( I^2C \) serial buses. Compare their bus speed, maximum number of devices on a bus, and signaling protocols.

2. Find out more information regarding the RS-232 protocol. Describe the maximum distance that a RS-232 connection can support, the bit rate, and available frame formats.

3. What function do Input Captures and Output Compares perform? Describe how Input Captures and Output Compares can be used to regenerate a variable frequency binary digital waveform (you do not need to consider phase synchronization).

4. What is meant by the resolution of an ADC/DAC? What types of issues arise when an analog signal is sampled using an ADC, and later recreated using a DAC?
Chapter 10

Handle With Care

*Here the Doormouse shook itself, and began singing in its sleep
‘Twinkle, twinkle, twinkle, twinkle’—
and went on so long that they had to pinch it to make it stop.*

from “Alice’s Adventures in Wonderland,” Lewis Carroll, 1832-1898

Since sensors and other peripherals enable Tribot to interact with its surroundings via I/O interfaces, the NXT must adopt suitable strategies to alert the ARM CPU to new inputs, or else inform the CPU that a particular output process has been completed. These tasks are typically handled via Interrupts which alert the CPU to events that trigger based on changes in I/O peripherals. We shall now look at how I/O and Interrupts should be handled by the CPU and software routines.

Input/Output (I/O) to peripheral devices is closely related to Interrupt Processing and Peripheral Interfacing for microprocessors. The advantage for microcontrollers is that several commonly used peripherals are already built into the package, standardizing the steps needed for processing peripheral I/O whereas the external peripherals used by a microprocessor may differ from one design to another.

Nonetheless, I/O peripherals enable the microprocessor or microcontroller to interface with the environment to receive inputs and generate outputs that may change the conditions in the environment (e.g., a stepper motor controlling a machine). I/O processing falls into the category of asynchronous events; they are not fully under the control of the processor. This means that an input event may occur while the processor is busy performing a calculation. Depending on the type of application and the type of input, it may or may not be possible to delay responding to the input. Inputs that need immediate response are termed real-time events, while those that can be processed when the processor is not busy are termed non real-time events.
10.1 I/O Categories

In general when classifying I/O events, they can be categorized into two categories, namely:

- Non Real Time I/O
- Real Time I/O

None Real Time I/O operations do not have a specific deadline associated with them, while Real Time I/O operations need to be completed before a given deadline. The type of I/O event being serviced influence the strategy that we need to use in order to fulfill its time related requirements.

10.1.1 Non Real Time I/O

This has been the traditional I/O model used in batch computing and data processing. The processing of information is performed in a linear, predictable fashion. E.g., a flow chart describes when the program will accept inputs, and how those inputs will be processed, and finally, what outputs are produced. Consequently, I/O processing is program initiated. The bottleneck in I/O response is the speed of the processor. The I/O events that need to be processed typically consist of user inputs (via keyboard, files or other data storage media), and program outputs (via display screen, LCD, etc.) Such applications are not time critical since the inputs and outputs are merely data and are not used directly for device control.

10.1.2 Real Time I/O

Real Time I/O covers a wide spectrum of applications. The most common use of real time I/O is to provide interactive response to input, such as the movement of a mouse pointer on a computer screen when a mouse is moved. The response time to such interactive events is in the range of tens to hundreds of milliseconds, tuned to the psychological response time of human users. Typically, response times of 100 - 250 ms is acceptable for user input events. However, for microcontrollers, real time I/O is often associated with device control, where external devices such as stepper motors need to be started and stopped within a specified time duration (usually measured in microseconds or milliseconds), termed soft real time I/O. Failure to respond to real time inputs and implement suitable output steps in some applications may cause catastrophic results (e.g., the destruction of a machine or other dire consequences). These applications are termed hard real time I/O. In any case, real time I/O processing is an important component of microcontroller programming and interfacing.

10.2 I/O Processing Techniques

When an I/O operation is desired, the processor will send a command to the I/O peripheral by modifying the Control Command Register (CCR), upon which some action (input or output) will be initiated. I/O operations take
a certain amount of time to complete. Upon completion, the I/O peripheral often indicate this via a Control Status Register (CSR) that contains the status of the I/O operation.

The CCR and CSR are found in the I/O peripheral and not in the processor. As described previously in the chapter on Microprocessor Architectures, processors that implement Memory Mapped I/O architectures access the CCR and CSR via reads and writes to specific memory address locations given by the logic design, where the I/O peripheral is connected to the Memory Data and Address bus of the processor. I/O Mapped I/O architectures access the CCR and CSR via special I/O instructions that activates a separate I/O Data and Address bus where the I/O peripheral is connected.

Two common approaches are available for I/O processing:

- Polling I/O
- Interrupt-driven I/O

10.2.1 Polling I/O

Polling I/O refers to the checking of the CSR using a program loop in software. This occurs under program control, and is mostly suited for non real-time, interactive and soft real-time applications. In pseudocode, this is:

```plaintext
Send I/O Command to CCR of peripheral
While not done
  Read CSR of peripheral
  If CSR == done, exit
Endwhile
```

The advantage of Polling I/O is that it is very simple to implement for keeping track of repeated I/O operations, e.g., to output a string consisting of many characters to the display. Once a character I/O is complete, the program can increment the pointer to the next character and perform the I/O operation again until every character is processed. However, this blocks the execution of any other process, and is termed Blocking I/O.

A processor implementing Polling for I/O would be busy looping in the While-Endwhile part of the Pseudocode. This may waste a lot of processing cycles if the processor speed is much greater compared to the time required to complete the I/O operation.

10.2.1.1 Kernel Level Polling

In the kernel context, polling may be needed for time-critical I/O sequences that need to be invoked in device drivers to deal with hardware specific timing requirements. The advantage of Polling I/O is that the response time can be calculated and the worst case behavior analyzed beforehand. Nonetheless, such polling activity should be minimized in Interrupt Handlers since it would block the servicing of other interrupts while an interrupt is active. An example is given in Listing 10.1 where the LCD Driver needs to send a particular command sequence via the SPI bus uses polling to check on the status of the SPI bus.
Listing 10.1: LCD Driver Code Snippet

```c
/* Code Snippet from $NXOS_ARMDEBUG_PATH/nxos/base/drivers/_lcd.c */

/* Set the data transmission mode. */
static void spi_set_tx_mode(spi_mode mode) {
    if (spi_state.mode == mode)
        /* Mode hasn't changed, no-op. */
        return;
    else
        /* If there is a mode switch, we need to let the SPI controller
         * drain all data first, to avoid spurious writes of the wrong
         * type. */
        while(!(AT91C_SPI_SR & AT91C_SPI_TXEMPTY));
    spi_state.mode = mode;

    if (mode == COMMAND) {
        AT91C_PIOA_CODR = AT91C_PA12_MISO;
    } else {
        AT91C_PIOA_SODR = AT91C_PA12_MISO;
    }
}

/* Send a command byte to the LCD controller. */
static void spi_write_command_byte(U8 command) {
    spi_set_tx_mode(COMMAND);
    /* Wait for the transmit register to empty. */
    while(!(AT91C_SPI_SR & AT91C_SPI_TXEMPTY));
    /* Send the command byte and wait for a reply. */
    AT91C_SPI_TDR = command;
}
```

In the `spi_set_tx_mode()` routine, `AT91C_SPI_SR` is a CSR for the SPI module. The `while()` statement will keep checking on the Transmit Empty (TXEMPTY) flag in `AT91C_SPI_SR` until it is True. This is accomplished by checking the value of the TXEMPTY flag in `AT91C_SPI_SR` using the `AT91C_SPI_TXEMPTY` bitmask. This statement illustrates the use of Logical Instructions such as AND, NOT, and EOR for performing flag status checking. If the address of `AT91C_SPI_SR` was a normal memory address location, this `while()` statement will either execute once if the condition were True, or else it would loop forever if the condition were False. However, since `AT91C_SPI_SR` is a memory mapped Control Status Register, it will change its value based on the hardware status of the SPI module. It should be noted that `AT91C_SPI_SR` needs to be declared `Volatile` to avoid attempts by the C Compiler to optimize the code.

The `spi_write_command_byte()` routine further illustrates polling used in a time-critical sequence of I/O commands issued to the SPI module. The second polling I/O is performed on the `AT91C_SPI_TDRE` flag. The Transmit Register Empty (TDRE) flag must be set before we can issue the next command to the SPI module. By using Polling I/O we can minimize the overall command sequence time compared to the use of different Interrupt Handlers for each of these events.
10.2.1.2 Application Level Polling

Input processing in the application context tend to use polling. The reason for this is because applications are not normally allowed to install Interrupt Handlers easily. This is to prevent buggy applications from causing kernel panics or malicious applications causing security breaches in the system. Nonetheless, many applications use timers for triggering timeouts and performing other sequential tasks, so many operating systems provide a standard API for defining timer interrupt events and associating a callback routine with the timer. This application supplied callback routine will be executed once the timer expires so that the application can perform the appropriate action. Since the callback routine is invoked in the normal application context, security breaches or kernel panics can be avoided.

Listing 10.2: Tribot-Thumb Code Snippet

```c
/* get_light_reading
 * Light Sensor Input Routine
 * Collect one light sensor reading
 * On Exit:
 *  R0: Collected Sample (unsigned byte)
 */
thumb_interwork get_light_reading
push {lr}
movs r0, #LIGHT_PORT
bl nx_sensors_analog_get_normalized /* returned value in range 0-100% */
thumb_iret r1 /* Thumb Interworking Return */

/* collect_samples
 * Sensor Sample Collection Routine
 * This routine calls the respective sensor input collection routine n times
 * >>>Add additional sensor input routines here<<<
 * Note: This routine blocks until n samples have been collected
 * (Max. delay is ~ 3 * n ms)
 * Register Usage:
 *  R4: Sensor Readings Array Index
 *  R5: Light Readings Array Pointer
 *  R6: Sound Levels Array Pointer
 */
thumb_interwork collect_samples
push {r4, r5, r6, lr}
movs r4, #n /* Number of readings to collect */
ldr r5, =light_readings /* Sensor Readings Array Pointer */
ldr r6, =sound_levels /* Sound Levels Array Pointer */
sample_loop:
subs r4, r4, #1 /* Convert Number of readings to collect into Array Index */
blt done_collect_samples /* We’re done when index is negative */
/* Sensor Input routines */
bl get_light_reading
strb r0, [r5, r4]
bl get_sound_level
strb r0, [r6, r4]
/* A/D Converter Delay */
/* Delay 3 ms before next reading (Required by AVR A/D Converter) */
movs r0, #SYSTICK_3MS
bl nx_systick_wait_ms
b sample_loop /* Next Iteration */
done_collect_samples:
pop {r4, r5, r6}
thumb_iret r1 /* Thumb Interworking Return */
```
Listing 10.2 illustrates the use of application level polling. Polling is done in the `collect_sample` routine which calls `get_light_reading` and `get_sound_level` subroutines to perform a sensor input action. Both routines interface to Analog sensors, so they operate pretty much identically except for the sensor port parameter value. The `get_light_reading` routine in Listing 10.2 calls `nx_sensors_analog_get_normalized` which is a polling input routine that returns the most recent A/D converter output value to the caller.

### 10.2.2 Interrupt-driven I/O

Interrupt-driven I/O processing is commonly used to service interrupts since the processor is not burdened with busy looping inherent in I/O Polling. Interrupt-driven I/O must be supported by the logic design of the embedded system, since the I/O peripheral must be interfaced to the Interrupt handling logic in order to generate interrupts when CSR change occurs.

Interrupt-driven I/O can be used for both input and output. For example, the peripheral may generate an interrupt when an input is available, or else when an output command is completed. Interrupt-driven algorithms use a similar approach to Event-driven programs, where the program performs an action in response to an external input or output event. Internally generated events such as timers generally trigger interrupts due to the expiry of a counter that is decremented periodically by the system.

#### 10.2.2.1 Handling Interrupts

Interrupt processing is asynchronous and occurs in a separate context from the main program. Interrupt Handlers must preserve register context and maintain persistent global variables to keep track of the state of any I/O operation spanning multiple interrupts (e.g., the output of a string to a display device), since the Interrupt Handler may be called while the processor is performing some other task. For single event outputs, an I/O Interrupt handler is setup to detect the interrupt generated by the peripheral when the I/O operation is completed. Once the interrupt occurs, the Interrupt handler retrieves the CSR value and updates a main memory variable used to store the I/O completion status value. This I/O completion status value may not be required by the main program if it is a one-off Output operation.

If a series of Output operations needs to be performed, the I/O completion status value may be used to pace the normal processing and I/O operations within the program to avoid overrunning the peripheral device. This is termed a Rendezvous point, where the program and I/O operations are synchronized to each other. While checking of the I/O completion status value may be done via polling, an operating system running multiple programs may choose to preempt the current program until the interrupt occurs, thus enabling other processor-based programs to execute. Interrupt driven I/O enables overlapping of processor-based and I/O based operations, where the active task is switched from an I/O-pending task to a processing-intensive task while an I/O operation is being performed. Overlapped processing utilizes the processor more efficiently.
; Main program
Clear I/O completion status variable
Setup Output I/O Handler
Send I/O Command to CCR of peripheral
Continue Processing of other data
Check I/O completion status variable (if required)
to wait for I/O completion event (Rendezvous point)
...
; Output Interrupt Handler
Read CSR of peripheral
Update I/O completion status variable

For inputs, it may not be possible to predict when the next input will be received by the processor. Therefore the I/O Input Handler must be configured beforehand to respond when the Input interrupt is received. The input values are then stored to a Input variable buffer that is known to the main program. The I/O completion status variable is then used to signal to the main program that input has been received.

; Main program
Setup Input variable and Input I/O Handler
Clear I/O completion status variable
Continue Processing of other data
Check I/O completion status variable (if required)
to wait for I/O completion event (Rendezvous Point)
...
; Input Interrupt Handler
Read CCR of peripheral
Update Input variable & I/O completion status variable

Event-driven programming is a slightly different approach compared to Polling or using Rendezvous Points, where program execution will block until I/O operations have completed. An event-driven program will check for inputs (events) once every event loop, but will not wait for them to be received before continuing with the rest of the processing. This is termed non-blocking I/O since processing is done only when inputs are received.

; Main program
Setup Input variables and I/O Handlers
Clear I/O completion status variables
Mainloop:
Case (I/O completion status variable) of:
  Event A: process event A
  Event B: process event B
...
End case
Goto Mainloop
; Interrupt Handler (Input / Output)
Read CCR/CSR of peripheral
Update Input variable (if input)
Update I/O completion status variable
10.2.2.2 Minimizing Interrupt Latencies

If an Interrupt Handler which is triggered to handle an event which requires extensive processing, it would cause significant delay to the servicing of other interrupts. This is because the processor inhibits or masks the servicing of other interrupts by default when we invoke an Interrupt Handler. To minimize this latency, the typical Interrupt Handler should execute its task quickly and return control to the currently executing process. In order to deal with lengthy processing requirements, the processing would be split into two parts. The essential task performed in the Interrupt Handler is just to copy the peripheral input values into memory status variables without any further processing. A background task will then be invoked after the Interrupt Handler returns control to the currently executing task in order to perform the required processing steps on the input. Since the background task executes in the normal system context, other interrupts may be serviced during the processing.

10.2.2.3 Interrupt Handler Initialization

Listing [10.3] contains code snippets from the systick driver.

Listing 10.3: Systick Code Snippet

```c
/* Code Snippet from $NXOS_ARMDEBUG_PATH/nxos/base/drivers/systick.c */
/* High priority handler, called 1000 times a second */
static void systick_isr(void) {
    volatile U32 status __attribute__ ((unused));
    /* The PIT’s value register must be read to acknowledge the */
    /* interrupt. */
    /* Keeping up with the AVR link is a crucial task in the system, and */
    /* must absolutely be kept up with at all costs. Thus, handling it */
    /* in the low-level dispatcher is not enough, and we promote it to */
    /* being handled directly here. */
    /* As a result, this handler must be *very* fast. */
    nx__avr_fast_update();
    /* The LCD dirty display routine can be done here too, since it is */
    /* very short. */
    nx__lcd_fast_update();
}

void nx__systick_init(void) {
    nx_interrupts_disable();
    /* Install both the low and high priority interrupt handlers, ready */
    /* to handle periodic updates. */
}
```
In the nx__systick_init() function, we use the nx_aic_install_isr() routine to install the systick_isr() Interrupt Handler into the AIC interrupt vector list. The vector entry and interrupt priority need to be specified so that the AIC will trigger the relevant interrupt for the correct event, and the priority determines whether it will interrupt an ongoing interrupt request (if any) that is being serviced. This critical driver controls the NxOS-Armdebug system counter, systick, used to determine the temporal order of all events. Consequently, it is assigned the highest priority interrupt for the ARM microcontroller. The frequency of the interrupt is then programmed into the PIT which will generate the interrupts at the appropriate rate. A slower PIT would lead to less interrupt processing overheads, but at the expense of the granularity of the system timestamps. The smallest duration that can be tracked by NxOS is the duration of one tick of the systick timer.

### 10.2.2.4 Rendezvous Points

The nx_systick_wait_ms() routine in Listing 10.3 is an illustration of the use of rendezvous points in I/O synchronization. The while{} statement is basically a polling statement that depends on systick_time reaching the desired value. The variable will change value as it is updated by the systick_isr() interrupt handler despite appearing like an endless loop if the expression were True. Hence the application will keep check-
ing systick_time until the given duration or rendezvous point is reached before continuing processing of subsequent program statements.

10.2.2.5 Interrupt Handler

The systick_isr() Interrupt Service Routine (ISR) is the actual interrupt handler that is called when the PIT interrupt triggers. One entry to the ISR, we read the PIT register to indicate that the interrupt has been serviced. Then the systick_time value is incremented. Since the ISR is the only routine which modifies the value of systick_time, it is safe to do so even if interrupt processing have not been disabled due to the invocation of the interrupt handler. Nonetheless, if the systick_time variable were modified in more than one place in the code, it is important that a mutex be used to prevent incorrect updates to its value. The other processing tasks performed in the ISR must also be limited to short instruction sequences to avoid missing other interrupts.

10.3 Interrupt Nesting in NxOS-Armdebug

Normally, interrupts are masked when we are inside a Interrupt Handler. However, there are some device Interrupt Handlers which are not time critical but may take a longer duration to execute than the 1 ms interval of the PIT (systick) ISR. Consequently Interrupt Nesting is used to allow higher priority interrupts to take over while we are still servicing a lower priority interrupt [46]. The ARM AIC supports 8 levels of interrupt priorities. For NxOS-Armdebug, the PIT (systick) interrupt has the highest priority. The second highest priority interrupt is assigned to the ARM-AVR I²C Link. Device drivers are then assigned lower priority levels in the AIC.

Although Interrupt Nesting allows higher priority interrupts to be serviced regardless of the processing duration for the lower priority interrupts, this ability comes with a cost. First, the stack usage requirements increase. It is possible to stack several interrupt frames in the stack if new interrupt handlers with higher priorities are invoked. In addition, the complexity introduced by the ability to interrupt an existing interrupt routine can potentially lead to deadlocks and other system instability issues. Consequently Interrupt Nesting needs to be carefully implemented.

Listing 10.4 shows a snippet containing the AIC Interrupt Request (IRQ) Dispatch Handler for NxOS-Armdebug. This code snippet illustrates both the interrupt nesting logic as well as how the Interrupt Dispatch Handler nx__irq_handler() calls the actual interrupt handler routines.

Listing 10.4: NxOS-Armdebug Nested IRQ Handler Snippet

```c
/* Code Snippet from $NXOS_ARMDEBUG_PATH/nxos/base/interrupts.S */
.data
.align
.global irq_state

irq_state:
interrupts_count: .long 1
.equ IRQ_INTR_CNT, (interrupts_count - irq_state)
```
#if defined (__DBGENABLE__) 
.global irq_stack_frame_address

irq_nesting_level: .long -1
/* -1: Not in Interrupt; 0: Top Level Interrupt; 1+: Interrupt Nesting Level */
irq_stack_frame_address: .long 0
irq_spurious_count: .long 0

.equ IRQ_NEST_LVL, (irq_nesting_level - irq_state)
.equ IRQ_STK_FRAME, (irq_stack_frame_address - irq_state)
.equ IRQ_SPURIOUS, (irq_spurious_count - irq_state)
#endif

.text
.align
/* Enhanced NxOS Nested Interrupt Handler.
   Based on notes from:
   "Building Bare Metal ARM Systems with GNU," Miro Samek,
   
   No context information is stored in IRQ mode.
   Instead, the interrupt context will be stored
   in the following privileged modes:
   
   * User/System Task (USR/SYS) -> System (SYS) Mode
   * NxOS Task (SVC) -> Supervisor (SVC) Mode
   * Debugger Task (ABT/UNDEF) -> Abort (ABT) Mode
   * all other reserved modes -> Abort (ABT) Mode
   
   * The interrupt stack frame consists of the following
   * Registers: SPSR_irq, PC_irq, LH, R12, R3, R2, R1, R0
   * which is identical to the ARM v7-M hardware
   * interrupt stack frame (see ref: Miro Samek).
   
   * This version also keeps track of the interrupt nesting
   * level and stores the top level interrupted instruction
   * address in order to let the Debugger know which instruction
   * should be breakpointed when invoking the Debugger from
   * Platform Operation Mode.
   */
/global nx__irq_handler

nx__irq_handler:
    /* In IRQ Mode (IRQ Disabled, FIQ Enabled) */
    /* The IRQ stack is not used at all. Hence, SP used as temp. var. R13 */
    mrs r13, spsr /* Retrieve SPSR_irq */
    /* Compare the SPSR mode bits to both 0000 and 1111 (usr/sys mode). */
    ands r13, r13, #0x0F /* eq (0) == User Mode */
    eornes r13, r13, #0x0F /* ne (!0) && eor #0x0F => eq (0) == SYS mode */
    msreq cpsr_c, #(MODE_SYS | IRQ_MASK) /* Match, so switch to SYS Mode */
    beq _irq_save_stack_frame
    mrs r13, spsr /* Reload SPSR_irq */
    and r13, r13, #MODE_MASK /* check for other modes */
    teq r13, #MODE_SVC /* Supervisor Mode? */
    msreq cpsr_c, #(MODE_SVC | IRQ_MASK) /* Match, so switch to SVC Mode */
    beq _irq_save_stack_frame
    msr cpsr_c, #(MODE_ABT | IRQ_MASK) /* Else, goto Abort Mode (catch all) */

_irq_save_stack_frame:
    /* In target privileged mode (IRQ Disabled, FIQ Enabled) */
    sub sp, sp, #2*4 /* Reserve stack space for SPSR and PC of Interrupted instruction */
    stmd sp!, {r0-r3, r12, lr} /* Save AAPCS clobbered registers to interrupt stack frame */
    add r0, sp, #2*4 /* Pass top of current interrupt stack frame to IRQ mode */
    mrs r1, cpsr /* retrieve current privileged mode (for mode switchback */
    msr cpsr_c, #(MODE_IRQ | IRQ_MASK) /* Switch to IRQ mode to save SPSR_irq and PC_irq */
    /* In IRQ Mode (IRQ Disabled, FIQ Enabled) */
sub    r2, lr, #4
  /* Adjust return addr (interrupted instr addr), copy to R2 for stacking */
mrs    r3, spsr  /* Retrieve SPSR_irq to R3 for stacking */
movfd  r0, {r2, r3}  /* completed saving interrupt stack frame (SPSR_irq, PC_irq, LR, R12, R3-R0) */

/* Interrupt Handler Housekeeping */
ldr    r3, =irq_state
ldr    r2, [r3, #IRQ_NEST_LVL]  /* Raise Nesting Level (0: Top Level Interrupt) */
str    r2, [r3, #IRQ_NEST_LVL]
adc    r2, r2, #1  /* Raise Nesting Level (0: Top Level Interrupt) */
/* Else Save Top Level Interrupt Stack Frame Address (for Debugger) */

/* Get the IVR value. */
ldr    r2, =AT91C_BASE_AIC
ldr    r3, [r2, #AIC_IVR]

/* If we're in Protected mode (usually for JTAG debugging), we
  need to write back to the IVR register to tell the AIC it
  can dispatch other higher priority interrupts again.
  In normal mode, this has no effect, so we can safely do it. */
str    r2, [r2, #AIC_IVR]

and    r1, r1, #MODE_MASK  /* Enable IRQ (& FIQ) */
msr    cpsr_c, r1  /* switch back to previous privileged mode */

_irq_dispatch_aic_ivr:
  /* In target privileged mode (IRQ & FIQ Enabled) */
  /* Dispatch the IRQ to the registered handler. */
  mov    lr, pc
  bx     r3  /* Use R3 to dispatch Interrupt Handler, so that it is not clobbered by
                 * default exception handlers */

_irq_exit_handler:
  /* Clean up and exit */
mrs    r1, cpsr
orr    r1, #IRQ_FIQ_MASK  /* Mask IRQ and FIQ */
msr    cpsr_c, r1  /* Disable interrupts to restore context */

  /* In target privileged mode (IRQ & FIQ Enabled) */
  mov    r0, sp  /* Pass privileged mode SF to IRQ Mode */
  ldr    lr, [r0, #(5*4)]  /* Restore LR to privileged mode */
  add    sp, sp, #(8*4)  /* unstack interrupt stack frame from current privileged mode */
  /* Switch back to IRQ mode and tell the AIC that the interrupt has been
     * handled. */
  msr    cpsr_c, #MODE_IRQ | IRQ_FIQ_MASK
  /* In IRQ Mode (IRQ & FIQ Disabled) */
  ldr    lr, =AT91C_BASE_AIC
  str    lr, [lr, #AIC_EOICR]

  /* Interrupt Handler Housekeeping */
  ldr    r3, =irq_state
  ldr    r2, [r3, #IRQ_NEST_LVL]
  sub    r2, r2, #1  /* Decrease nesting level */
  str    r2, [r3, #IRQ_NEST_LVL]
  ldr    r1, [r0, #(7*4)]  /* Load SPSR to R1 */
  msr    spsr_cxsr, r1  /* Restore SPSR_irq */

ldmfd  r0, {r0-r3, r12, lr, pc}
/* Return execution to interrupted instruction, restore CPSR from SPSR_irq */
/* Note: SP_irq does not contain an actual IRQ stack pointer, whereas
  * LR_irq is not used, so it does not matter what value they have on exit */
The data definition section contains several variables that can be used by a JTAG-connected Remote Debugger to keep track of the nesting level and active stack frame pointer of an active interrupt handler. It is not safe to use the Armdebug Remote Debugger to perform Interrupt debugging, since the IRQ Handler will disable all lower priority interrupts upon entry, including the USB driver used by the Armdebug stub to communicate with the PC. Consequently interrupt level coding and debugging requires a more invasive approach in terms of a JTAG connection to the NXT circuit board, as well as the specialized JTAG Debugging module and software drivers. Nonetheless, we shall examine the AIC IRQ Dispatch Handler to understand a little bit more regarding how interrupts are processed in NxOS-Armdebug.

10.3.1 Interrupt Context Preservation

Typically when IRQ handlers are invoked, the first thing that needs to be done is to preserve all registers that are modified within the IRQ handler, so that we can restore them to their original state when we have finished processing the interrupt. These register values would be placed in the stack. Since the ARM Architecture allows for independent Stack Pointers in each privileged mode, it seems natural to define an IRQ stack for this purpose. Unfortunately the limited RAM in the AT91SAM7S256 microcontroller introduces a difficult compromise. RAM reserved for an IRQ stack would not be available for other use. If we are not able to size the IRQ stack correctly, we would either waste a lot of RAM storage, or else encounter sporadic data corruption as the IRQ stack overflows. Consequently, the IRQ Handler utilizes the existing privileged execution mode stack for interrupt context preservation. The NxOS-Armdebug IRQ Handler recognizes the following active privileged execution modes: System (SYS), Supervisor (SVC), and Abort (ABT). The SYS mode is used for handling both System and User context interrupt events while SVC mode is used for NxOS tasks, and would be the default context when most interrupts are encountered. The Abort mode is used by the Debugger tasks, and is used to catch interrupt events triggered in any unexpected (reserved) execution modes as well.

In order to speed up the Interrupt Context Preservation process and reduce the stack frame size, only essential registers would be preserved. Other registers not modified by the IRQ Handler will not be put on the stack. This introduces some constraints on how certain tasks are performed within the IRQ Handler. The first constraint is that any given register would be used (modified) only when absolutely necessary. The second constraint is that access to register locations is less costly than access to stack contents in memory, so data transferred between the two should be minimized. The approach used in Nxos-Armdebug is based on the algorithm presented in [46]. Only the $SPSR_{irq}$, $PC_{irq}$, $LR$, $R12$, $R3$, $R2$, $R1$, and $R0$ registers would be preserved. In addition, since we do not use an IRQ Stack, we use $R13$ ($SP_{irq}$) as a scratch register that will not be preserved.

Upon entry into the IRQ Dispatch Handler, IRQ Interrupts are dis-

\footnote{Currently all Nxos-Armdebug applications run in Supervisor (SVC) Context since the Task Scheduler is not enabled.}
The first step is to determine the execution context that was active. This information is stored in the $SPSR_{irq}$ register. Since we cannot manipulate the $SPSR_{irq}$ register directly, the MRS command is used to copy its value into the $R13$ scratch register. After determining the privileged execution context that triggered the interrupt, we will switch back to the previous privileged execution context (USR context interrupts will be processed in SYS context), and the registers $R0 - R3, R12$ and $LR$, which are non-banked registers except for $LR$ which is a context specific banked register, would be saved onto the stack of the interrupted context. However, before saving the registers for the given context, two additional slots are reserved for the IRQ mode $SPSR$ and $PC$ register values.

Once the non-banked registers have been saved to the stack, we can use them for performing the actual IRQ Dispatch Handler operations. $R0$ is used to point to the two reserved slots in the stack. We then return to the IRQ mode by configuration the $CPSR$ mode bits appropriately. The value of $SPSR_{irq}$ is copied to $R2$, and the address if the interrupted instruction is calculated from the value found in the $LR_{irq}$ register, and stored as $PC_{irq}$. Both these values need to be preserved in the interrupted context stack, which we access using the address passed via $R0$ earlier.

### 10.3.2 Interrupt Handler Dispatch

The next step is to perform some housekeeping tasks such as updating the debugging information related to the Interrupt Nesting Level, and updating the Interrupt Stack Frame Pointer as necessary. The actual Interrupt Handler for the given interrupt type is retrieved from the AIC. We then reenable AIC interrupts to allow for higher priority interrupts to trigger, since the previous context has already been preserved. We return to the interrupted context with IRQ enabled to allow for nested interrupts, and then dispatch the Interrupt Handler via an indirect interworking subroutine call using the BX instruction. This allows us the flexibility of writing the actual Interrupt Handler either as an ARM routine or as a Thumb routine. It should be noted that the Interrupt Handler is written as a normal ARM or Thumb interworked routine. Of course, it is responsible for saving any register values that need to be preserved based on the AAPCS requirements (Table 3.1). There are no special considerations for returning from the Interrupt Handler since the IRQ Dispatch Handler takes care of all the housekeeping tasks of preserving interrupt context registers and restoring them.

### 10.3.3 Interrupt Context Restoration

Upon returning from the Interrupt Handler, all interrupts (IRQ and FIQ) are disabled to enable the IRQ Dispatch Handler to restore registers to their previous values. First we must save the interrupt stack pointer value in $R0$, after which we will restore the original interrupted context $LR$ and $SP$ registers. We can safely restore the $SP$ register value in the interrupted context, although we still need to access the previously stacked values later.

---

3FIQ Interrupts may still be triggered but FIQ interrupts are not used in the NXT.
Table 10.1: Comparison of Various I/O Processing Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polling</td>
<td>Easy to Implement</td>
<td>Waste Processing Cycles</td>
<td>Batch Processing, Timing Critical Sequences (within I/O Handlers)</td>
</tr>
</tbody>
</table>

since no other interrupts can intervene and affect the SP register value or memory contents of the stack at this point. We then switch back to IRQ mode, inform the AIC that the interrupt has been serviced, and update the debugging variables. Finally we restore $SPSR_{irq}$ from the stack, to enable us to return execution to the interrupted instruction with the original context intact, and return control to the interrupted instruction via a LDMFD instruction using the interrupt stack frame contents referenced via $R0$. The caret (‘^’) at the end of the operand list indicates to the processor that it should restore the contents in $SPSR_{irq}$ to the $CPSR$ as a result of executing the LDMFD instruction.

10.4 Summary of I/O Processing Approaches

The advantages and disadvantages of each approach is summarized in Table 10.1. Kernel level device polling is typically not encouraged since they waste processing cycles. Nonetheless this approach may be employed in specific timing critical sequences within I/O Interrupt Handlers. Interrupts are generally used in most embedded systems but they incur additional overheads in terms of the need to preserve the interrupted context when servicing interrupts. Generally interrupt processing requires the use of I/O Completion routines to detect that a particular long duration I/O operation has completed and subsequent I/O processing can continue.

10.5 Chapter Summary

- I/O Processing involves understanding the time requirements and types of I/O needed by a system.

- Two main approaches for I/O processing are: Polling I/O and Interrupt-driven I/O.

- Multi-level interrupt priority is supported by the ARM AIC. This allows us to perform Interrupt nesting which allows higher priority interrupts to be triggered while serving a lower priority interrupt.
10.6 Review Questions and Problems

1. What is the impact of enabling Interrupts during Exception Processing? Give some examples why this would be necessary? What steps need to be taken in order to ensure that Interrupt handling is safe during Exception Processing?

2. Some microcontrollers has only one Interrupt Request input. Explain how multiple external interrupt sources could be handled using external logic devices, and the tradeoffs that are incurred using this approach? How does the ARM architecture handle multiple external interrupt sources?
Chapter 11

Moving On

'Dear me,' said the Voice, 'how sudden!
Well, come to me tomorrow, for I must have time to think it over.'
said by the Wizard of Oz to Dorothy and friends,

The MINDSTORMS kit is only the beginning for exploring Embedded Systems design and ARM Assembly Language programming. Having learnt the programming concepts and peripheral interfacing intricacies, it is now time to look at how a larger system is organized. Typically larger systems have a Real Time Operating System (RTOS) to coordinate all the low level details needed to build complex software and controls. A RTOS provides standardized interfaces for performing common resource acquisition and utilization tasks, at the expense of added memory and processing overheads.

Examples of RTOSes for the NXT include:

- NXTMote [47]
- nxtOSEK [48]
- NxOS [28]

NXTMote is based on TinyOS [49], which was designed for use in wireless sensor network nodes from several manufacturers, with a strong focus on inter-device communications primitives. In addition, it uses a custom C-like language (nesC) for specifying tasks that are executed by the TinyOS scheduler. nxtOSEK, on the other hand, was developed as a TOPPERS/ATK (Automotive Kernel) compliant embedded operating system for the NXT [48]. It utilizes the LeJOS NXJ [50] device drivers as the basis for supporting the NXT hardware platform for the TOPPERS operating system. nxtOSEK applications are developed in C/C++. On the other hand, NxOS, of which NxOS-Armdebug is the version we have used for this book, was developed as an open source, written from the ground up, C-based operating system for the NXT hardware platform. The objective of NxOS was to implement a more streamlined native OS for NXT which has a cleaner API. While architectural similarities exists between LeJOS and nxtOSEK, the drivers
have mostly been rewritten [28]. In NOS-Armdebug, applications can be developed either in C or ARM Assembly language.

In order to understand better the role of the RTOses, we should understand what are Real Time Embedded Systems, and how a general Real Time Embedded System is designed. Real Time Embedded Systems have been traditionally used in event-driven control systems, such as automatic transmission systems in automobiles, pick-and-place machines for manufacturing, and other industrial control applications. More recent real time embedded systems include multimedia encoding and decoding systems such as digital cameras, MP3 players, and cellular telephones. The common criteria in all real time embedded systems is that they must have a finite response time to user and other external (sensor) inputs that the system is designed to process.

11.1 Real Time System Concepts

Real-time systems have been in use for industrial and robotic controls since the time that reprogrammable machine controllers were invented. Various strategies for developing of Real Time Systems (RTS) are covered extensively in [51, 5, 52, 53]. For example, a modified System Analysis and Design (SAD) approach to systems specification and development was use in [51], whereas an enhanced Object-Oriented approach was used in [52]. Other issues such as the advantages of cooperative multitasking schedulers compared with pre-emptive schedulers in real-time systems design were highlighted in [53], which advocated using a Time-Triggered approach. The objective of this chapter is not to introduce or discuss in depth any specific design methodology for real time systems, since methodologies evolve over time, but to highlight several common issues which need to be addressed in any real time systems design:

- **Response Time**
- **Deadlines**
- **Instantaneous vs. Periodic Events**

Properly specified and designed real time systems have bounded event response time and task deadlines, while being able to process all specified instantaneous and period events without overloading the processor. These issues are what distinguishes real time systems from normal interactive systems such as those found on desktop computers. We often see desktop operating systems get overloaded performing some tasks and become non-responsive. Such behavior cannot be tolerated in real time systems such as robotic control software.

11.1.1 Response Time

The biggest difference between normal computing system design and real time system design is in the area of response time. Response time is measured from the moment that an input (usually triggered externally) event
occurs to the moment that the system performs the necessary output functions in response to the event. The maximum allowable response time (deadline) for each event handled by the system makes up the design criteria for the real time embedded system. The type of real time system affects the ability of a given embedded system to meet its deadline requirements.

11.1.2 Deadlines

The inability to meet deadlines occurs due to the system having to handle multiple tasks or processes. If a particular task takes too long to complete, subsequent input events may not be processed in time. In addition, if a preemptive scheduler is used in the Embedded System to process tasks, a particular task may be preempted causing it to miss a given deadline.

Missing deadlines will cause the system response to fall behind in processing an input stream (for example, in encoding a sequence of images on a video camera), which eventually forces the system to drop inputs or output actions in order to keep up with continuous input events. If there are intermittent pause in input events, the system may be able to ‘catch-up’ with its processing requirements and not drop any inputs or outputs.

11.1.3 Instantaneous vs. Periodic Events

Instantaneous events are one-off input events which occur asynchronously (at any time), whereas periodic events recur at a specified time interval. Examples of instantaneous events are starting and stopping an internal combustion engine by the user, while periodic events would be the timing sequence needed to control the fuel injectors and ignition to ensure the smooth running of the engine.

The embedded system must be able to service periodic events effectively and within deadlines, since missing such deadlines would cause the system to fall behind and eventually have to skip the processing of some periodic inputs. This indicates a poorly designed system as it cannot service its sustained load. The sustained load is the sum of the processing time required to service all periodic events. In addition, there must be sufficient spare processing power to handle the occurrence of instantaneous events which has to be processed on top of the sustained load. The total processing time required for servicing all periodic and instantaneous events is the peak load of the system.

11.2 Soft and Hard Real Time Systems

11.2.1 Soft Real Time Systems

Soft real time systems have less stringent deadline requirements. A soft real-time system attempts to meet given deadline requirements, but is not able to do so 100% of the time.

The percentage of events that are processed within its deadline requirements determine how suitable a given soft real time system is for particular tasks. If the system is only able to meet 50% of its deadline requirements, it
would not be suitable for most interactive or multimedia streaming applications. However, a system that is able to meet 98% of its deadline requirements may well be suitable for less demanding consumer applications such as TV program recording and playback at the expense of lowered picture quality.

11.2.2 Hard Real Time Systems

Hard real time systems are systems that are designed to meet 100% of their deadline requirements. Industrial control systems must invariably be hard real time systems since the inability to respond to an event may lead to catastrophic consequences, such as the failure of a nuclear reactor.

Designing hard real time systems is obviously more difficult and ‘harder’ than designing soft real time systems. A detailed analysis of all possible events and responses must be performed beforehand, forming the design criteria for the system. In addition, the execution time of all routines must be calculated and worst-case completion times must be provided for routines with multiple execution paths (caused by logic branches within the routine). Finally, events that trigger a complex sequence of responses must be analyzed to ensure that all pertinent actions are within the specified deadlines while the system is still responsive to new inputs that may occur asynchronously.

11.3 Real Time System Design Considerations

Real Time System Design issues typically fall into the following areas:

- Input and Output Processing
- Task Scheduling
- Processor Dimensioning

11.3.1 Input and Output Processing

The concept of Polling I/O vs. Interrupt Driven I/O was introduced in Chapter 10. For most real time systems, Interrupt Driven I/O is a necessity since input events occur asynchronously to each other, while some events cannot wait until they are polled but must be processed immediately (Chapter 10.2.2). Consequently Real Time Embedded Systems have many of Interrupt Handlers, one for each input event task the system is responsible for.

When an Interrupt occurs, normal user task execution is suspended while the Interrupt Handler is called to service the input event. However, events that require extensive computation cannot be handled exclusively using Interrupt Handlers, since other interrupts are disabled during the execution of a particular Interrupt Handler. Interrupt Handlers for such events often act as signals, to initiate a user task that is executed when the signal is set. Thus real time systems often have at least two levels of processing. The lowest level interrupt handling routines for immediate event processing, and a higher level of computationally intensive user tasks that are executed when the system is not processing interrupts.
11.3.2 Task Scheduling

Since the real time system has user level computational tasks that execute in addition to interrupt-based event handling tasks, the task scheduler must be designed with great care to ensure that it is able to meet required deadlines. In general, two categories of task schedulers are available [53]:

- preemptive
- non-preemptive

Non-preemptive schedulers execute each user task to completion before executing the next task. Real time systems prefer non-preemptive schedulers, as the completion time of each task can be determined accurately. This is not possible for a preemptive scheduler since a task may be suspended if its time quantum is exceeded.

In fact, for hard real time systems, the non-preemptive scheduler is often a static scheduler, where all user tasks are known beforehand and are executed in a specific order. In this way, the system can be made deterministic.

Task prioritization is another important issue to be addressed by the task scheduler. Peripheral input events often have much higher priority compared with user input events. This is in contrast to Interactive Computer applications where user inputs often have higher priority. The rationale for making user inputs a lower priority compared to peripheral inputs is to ensure that control algorithms are able to respond to external input events within the specified deadlines. User Input processing would only be performed when the system is not busy with its event handling tasks. However, since user input response times are much longer in comparison with event deadlines, the user may not be even notice it.

While preemptive schedulers can be used in real time embedded systems, its operation are relegated to soft real time systems where missing some deadlines are not as critical. Otherwise, multi-level priority queues must be used to ensure that critical system tasks are executed with sufficiently high priority. The priority determination process for preemptive schedulers is a very tricky design issue and often leads to unexpected problems in real time systems employing such schedulers [53].

11.3.3 Processor Dimensioning

Another issue that often arises in Real Time System design is the question of “How powerful a processor to use?” For most desktop personal computer systems, the question is often answered as “The fastest available for the budget.” However, for embedded systems, cost consideration is one of several aspects. Power consumption, robustness and type of application are also important criteria.

Consequently for consumer products, this question is often answered as “The cheapest that can do the task;” while for industrial control applications, the answer is often “As powerful and reliable as the design dictates.” In answering this question, a metric that is often used is the Processor Utilization Ratio (PUR).
PUR is measured based on processor utilization time, which is defined as the time the system is performing tasks (which include interrupt processing, computations and user inputs), given in Equation 11.1.

\[ \text{PUR} = \frac{\text{Processor Utilization Time}}{\text{Available Processor Time}} = \frac{\text{Available Processor Time} - \text{Processor Idle Time}}{\text{Available Processor Time}} \] (11.1)

A PUR of 0% implies that the system is not doing anything, while a PUR of 100% implies that the system is fully loaded or overloaded (where deadlines will be missed).

A soft real time system may have a high average PUR (> 95%) while a hard real time system might need to provide a guarantee that the system is able to respond to unexpected events by requiring a low average PUR. In any case, a hard real time system must have a worst-case or peak PUR of less than 100% to avoid missing deadlines. Portable systems often implement power saving modes that disable certain portions of the system to conserve battery power. The drawback of having power saving modes is that the system may take a longer time to respond to inputs (processing lag) compared to a fully powered system. Power saving modes would be implemented as long as the processing lag vs. power savings gain makes it a worthwhile tradeoff.

### 11.4 Three-tiered RTS Software Architecture

The various design considerations therefore lead to a three-tiered software architecture for supporting the input processing requirements of real time...
systems, as shown in Figure 11.1.

The lowest levels of this software architecture (closest to the hardware interface) must have sufficient processor cycles to complete their tasks in as short a time as possible. Interrupt Handlers must not perform lengthy calculations which tie up the processor, since other interrupts are disabled during interrupt processing. For example, a High Definition (HD) video capture device would just transfer the information to a memory buffer and not attempt to perform image processing on the input.

Computational Tasks are responsible for the bulk of the processing needs of the system. In the case of the HD video capture device, the image data obtained from the Interrupt Handler would be processed by one or more Computational Tasks that are responsible for image enhancement, compression and storage. Although Computational Tasks do not need to complete their tasks in as short a time as possible, they are typically used to handle periodic events (such as the encoding of a video stream) which require that the respective tasks be completed within a specific time duration.

User Interface tasks are instantaneous (one-time) events that occur asynchronously to normal computational task execution. For example, the user presses a Start/Stop Recording button. Since such events are considered rare in terms of the processing load of the processor, it may be relegated to the lowest priority task and only executed at specific intervals by the process scheduler. This would not normally be noticed by the user if the low priority UI task is able to respond to an input within 100-250 ms.

11.5 Linux, RTS and the Future of NXT

As microcontrollers improve in both capabilities and power, the trend has been to move towards standardized operating system platforms on embedded systems. For example one popular embedded operating system platform is Android, which is widely used in smartphones and tablet devices. Android is based on the open source Linux kernel [54]. Variants of the Linux kernel have been developed to support real time systems requirements [55]. Nonetheless, the resource requirements for Linux is much higher than that available on the AT91SAM7S microcontrollers, with a minimum RAM requirement of 2 MB for uCLinux [56] which is customized to run on small footprint embedded systems. Consequently it is not possible to run Linux on the current NXT hardware. However, the next generation of the MINDSTORMS platform, the EV3, will feature a more capable ARM-based microcontroller which supports Linux [57]. While most of what we’ve covered in this chapter is still relevant for Linux-based systems, they would also exhibit much greater complexity in terms of the overall system architecture. I hope that what you’ve learnt will serve as a compass to guide you as you navigate this brave new world of advanced embedded systems, whether in robotics or in other areas. In any case, it will be an exciting journey!
### 11.6 Chapter Summary

- Real Time System designers have to define accurately the *response time requirements, deadline compliance* and *Periodic & Instantaneous system events*, in order to create a viable real time system.

- Design considerations include *I/O processing, task scheduling*, and *processor dimensioning* issues.

- A Hard Real Time System must have peak PUR < 100% in order to function correctly.

- Embedded Operating Systems are moving towards standardized platforms; an example is the open source Linux-based Android OS.

### 11.7 Review Questions and Problems

1. Analyze a Digital Media Capturing Device (e.g. Video Camera) to determine relevant tasks at each level of the RTS Software Architecture hierarchy.

2. Determine what kind of processor (in terms of the bit width) and processor frequency (MHz) are typically used for the following devices/applications:

   - Microwave Oven
   - Washing Machine with Intelligent Wash Cycles (e.g., Fuzzy Logic)
   - Digital High Definition (HD) Video Recorder
   - Touch screen tablet computers
   - Digital Camera (16 Megapixels)
   - Car Engine Fuel Injection Controller

3. Calculate PUR for *Tribot* by modifying the `sleep_robot()` routine.
Appendix A

LEGO MINDSTORMS Components

‘The whole is greater than the sum of its parts.’
attributed to Aristotle, 384-322 BC

The LEGO MINDSTORMS kit consists of various parts, sensors and the NXT brick used to build a robot. The objective of this Chapter is to provide a quick introduction to some of the commonly used components, and not as a comprehensive reference to each and every part since different kits have their own bill of materials. These components can be categorized into the following:

- Controller (NXT)
- Motors
- Sensors
- Structural Beams (Technic)
- Brush, Axles, and Connectors (Technic)
- Gears (Technic)
- Miscellaneous parts

A.1 Controller (NXT)

The NXT Controller (Figure A.1) contains the ARM based Atmel microcontroller that is studied in detail in this book. The controller also houses several buttons, a speaker and a LCD screen. Further information is given in Figure 9.2 and 112.
Figure A.1: NXT Controller Brick

Figure A.2: Servo Motor and Connection Cables
A.2 Motors and Cables

The NXT Servo Motor (Figure A.2) is equipped with a tachometer interface to return the position of the motor. This allows the NXT to determine how far the motor has rotated for more precise control of the movement of the motor.

A.3 Sensors

Analog and Digital sensors (Figure A.3) are available for the NXT. Analog sensors provide their inputs via the A/D converters, while Digital inputs need to be accessed via the I2C bus in order to control and retrieve input data.

A.4 Structural Beams

Structure Beams (Figure A.4) are LEGO Technic parts that can be used to create static and dynamic structures controlled by the motors.
Figure A.4: Various Structural Beams

Figure A.5: Various Brushes, Axles, and Connectors
A.5 Brush, Axles, and Connectors

Beams, axles, connectors and brushes (Figure A.5) are used to connect the various beams to each other and to create joints and movable connections.

A.6 Gears

Gears (Figure A.6) allow the creation of drive trains which provide different torque ratios based on the difference in the number of cogs of connected gears.
Appendix B

Cross Development Toolchain

'GNU is as GNU does.'

The use of a Cross Development Toolchain is an essential component of ARM Assembly Language development. Various commercial and free alternatives are available, including the ARM Development Suite (ADS) syntax\[58\] supported by the Microcontroller Development Kit (MDK-ARM) from Keil - An ARM Company, the EWARM toolchain from IAR Systems used for developing the official NXT Firmware, to the GPL'ed GNU ARM Cross Development Toolchain (which includes the GNU C Compiler, gcc, and GNU Assembler, as) from the Free Software Foundation (FSF). One of the difficulties in using a particular toolchain for Assembly Language programming is the differences in Assembler Directives and even Assembly Mnemonic syntax.

B.1 Obtaining the GNU ARM Cross Development Toolchain

Although it is possible to build the GNU ARM Cross Development Toolchain yourself (a build script is provided in the NxOS distribution), it is easier to obtain a pre-built binary distribution of the toolchain for your operating system when first starting up. There are many pre-built options, among them, CodeSourcery, MICROCROSS, GNUARM and YAGARTO. YAGARTO may be the easiest for beginners since it is free and they provide up-to-date pre-built binaries for Windows, along with helpful tutorials on how to integrate it with the Eclipse Integrated Development Environment (IDE). The Eclipse C Development Toolkit (CDT) IDE is meant for C and C++ development, but it is useful as an editor for assembler language programs as well.

- Sourcery CodeBench\[59\]
- Microcross\[60\]
• GNUARM [61] (older gcc versions)
• YAGARTO [62]
• Eclipse CDT IDE [63]
• SConsolidator [64]
• NXTGCC [65]
• NeXT Tool [63]

A brief comparison of tools for various Operating Systems is provided in Table B.1. From experience, the YAGARTO cross-compilation toolchain is the most actively maintained on the Windows and Mac OS platforms. However, on the Mac OS X platform, it may be easier to stick to a single source for all the required tools (cross-compiler, project build tool (scons), and project documentation tool (doxygen)), by using the cross-compilation toolchain from MacPorts [66]. Generally the cross-compilation toolchain version available from MacPorts trails the version available from YAGARTO. On Linux, CodeSourcery appears to be the most actively maintained toolchain.

Table B.1: Cross Development Tools for various Host Platforms

<table>
<thead>
<tr>
<th>Cross Development Toolchain</th>
<th>Windows</th>
<th>Mac OS X</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sourcery CodeBench</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Microcross</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GNU ARM</td>
<td>ELF Only</td>
<td>No</td>
<td>ELF Only</td>
</tr>
<tr>
<td>YAGARTO</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NXTGCC</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MacPorts</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Eclipse CDT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NeXT Tool</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

B.1.1 A Note About ABIs

The Application Binary Interface (ABI) is the defined register usage convention for parameters passing among routines and libraries written in various high level languages, to enable correct interoperability and linking of different routines and libraries into a given executable file. ARM covers it in the ABI for ARM Architecture: The Base Standard (BSABI) document [22]. The BSABI defines the latest Embedded ABI (EABI) standard for ARM microprocessors, which has been implemented by GNU ARM Cross Development Toolchain in the arm-none-eabi target used for native (bare metal) embedded applications. While other GNU ARM Cross Development Toolchain targets such as arm-elf [24] are also available, they use the older ARM Procedure Call Standard (APCS) ABI which has been deprecated by ARM Ltd.

The other advantage of the arm-none-eabi target is that it has better support of software and hardware FPU, allowing for mixing of code which uses the different conventions, better support for structure packing and
more efficient syscall implementations\cite{67}. In addition, code generation is potentially more efficient since it is able to support interworking of ARM and Thumb code better.

So to summarize, standalone embedded applications written entirely in Assembly Language would probably not be affected significantly by the choice of the GNU ARM Cross Development Toolchain target, but when Assembly code is mixed with other code written in high level languages, the \texttt{arm-none-eabi} target is preferred.

### B.1.2 Toolchain Components

- **Java Runtime Environment (for use with Eclipse)**
  - **Interpreter:** Python (2.7.x, NOT 3.x)
- **Software Drivers**
  - OS specific: Fantom Driver\cite{68} for Mac OS X / Windows, libusb for Linux
- **Software Libraries**
  - OS specific: pyfantom\cite{69} for Mac OS X / Windows, pyusb for Linux
  - Common: nxt-python\cite{70}
- **Editor/IDE**
  - Eclipse CDT\cite{63}
  - SConsolidator\cite{64}
- **Downloader:** NeXTTool\cite{31}
- **Build Tools**
  - Version Control: git
  - Project Documentation: doxygen
  - Project Build Tool: scons
- **Cross Compilation Tools**
  - C Compiler: GNU C/C++ Compiler
  - Assembler: GNU as
  - Linker: GNU linker
  - Disassembler: GNU objdump
  - Debugger: GNU gdb (requires USB-based Armdebug server for remote debugging)
Before we can use the NXT for NxOS development, we need to upgrade the firmware to support downloading of native ARM programs, also known as NXTBINARY format programs. First we need to download and install NeXTTool (Figure B.1) [31], which is part of the Bricx Command Center toolkit maintained by John Hansen. It will be used to download NXTBINARY programs to the NXT, as well as update the firmware.

The NXT Enhanced Firmware [29], developed by John Hansen, is backward compatible with the default LEGO Firmware. However, it provides the additional capability of running NXTBINARY format native ARM programs from the on-NXT graphical user interface, hence enabling execution of both NXT-G and NxOS-Armdebug software on the same device. To update the firmware on the NXT, the NXT brick must first be powered on and connected to the PC via the USB cable. Make sure that you have fresh batteries (fully charged) before performing the firmware update, as sudden loss of power during the firmware updating process will render the NXT unbootable (also known as ‘bricked.’) \(^1\) Figure B.2 shows the NXT Enhanced Firmware file being selected after clicking on the 3D Blue Down Arrow in the NeXTTool screen. The PC will then show a progress bar, at the end of which the NXT will reboot and show the startup animation sequence.\(^2\) Finally, verify that the firmware has been updated (Figure B.3).\(^3\)

---

1. If the NXT is ‘bricked’, there will not be any display on the LCD. Instead, a soft ‘clicking’ sound will be heard from the speaker. This means that the NXT is now in firmware download mode and need to have a new firmware applied before it can function. You can trigger the firmware download mode manually by using a LEGO antenna or a bent paper clip to press the RESET button which is located inside a hole located right underneath the USB port.

2. There will not be any sound as the sound files need to be downloaded to the NXT file system as a separate step.

3. The NXT Enhanced Firmware include the string “NBC/NXC” in the Firmware (FW) version field.
Figure B.1: NeXTTool Main Window

Figure B.2: NXT Enhanced Firmware Download Selection
B.3 Using the Cross Development Toolchain

One of the main components of a Cross Development Toolchain is the Integrated Development Environment (IDE). In order to use Eclipse CDT for developing NxOS programs, we need to perform some customization of the environment, by adding some additional plugins. Alternatively, most of the tasks in software development can be accomplished using a normal text editor and familiarity with issuing commands via the command line.

B.3.1 Customizing Eclipse

To support building NxOS within Eclipse, we need to add support for the scons project build tool into Eclipse. There are two options, either use a Custom Build Tool, or use the SConsolidator plugin [64]. The Custom Build Tool is project dependent, and is stored as per-project configuration settings. This is described in Section C.2.1.1. On the other hand, SConsolidator is a plugin which is available for use by all projects within Eclipse. Here the steps needed to install the SConsolidator plugin will be described.

SConsolidator [64] is an external Eclipse plugin. SConsolidator can be found at http://www.sconsolidator.com/update. To install, access Help->Install New Software... in Eclipse, and enter the SConsolidator URL, then select the relevant options. SConsolidator Dependency Visualization needs additional Eclipse plugins to work (Figure B.3). Here
we are just interested in the Base functionality. Follow the prompts (Next), and eventually it should be installed successfully.

![Figure B.4: Installing SConsolidator](image)

**B.3.2 Defining Build Targets**

Details regarding how to import NxOS into Eclipse is given in Section C.1.2. This section pertains to configuring Eclipse for building and running NxOS after it has been successfully imported. Build targets in NxOS are defined using the format `appkernels=<target>`, where the target is the name of the subdirectory inside the `$(projpath)/nxos/systems/` folder. An example of a target is `armskel`, which is a skeleton program which can be used to start a new project, and to verify that the cross-compilation toolchain works correctly.

**B.3.2.1 Defining Build Targets for Custom Build Tool**

If you are using the Custom Build Tool, there is no convenient way of specifying the build targets. One way to do so is to specify the build target as part of the argument list in Figure C.8. Hence the argument list for always building the `armskel` target will become:

```
-C nxos appkernels=armskel
```

This is not flexible, since changing the target involves editing the settings for the Custom Build Tool.
Alternatively, the specific target is not defined in the Custom Build Tool. Instead, scons is invoked once via the command line with the required target (using the same argument list given above), and subsequent invocation via the Custom Build Tool will rebuild the last explicitly specified target since the target is cached in the 
\texttt{${projpath}/nxos/scons.options}$ file.

**B.3.2.2 Defining Build Targets for SConsolidator**

SConsolidator provides a GUI interface for specifying different build targets for a given project. A new build target is defined by first accessing the \textit{Project->Properties} dialog, and then selecting \textit{SCons->Build Targets} (Figure B.5). The target can then be created (Figure B.6).

![Figure B.5: SConsolidator Build Target Dialog](image)

![Figure B.6: SConsolidator Target Specification](image)

Alternatively, the \textit{SCons Target View} panel provides another way of setting the target (Figure B.7).
B.3.3 Debugging NxOS via Eclipse

Since we will be using remote debugging to access the NXT (see Section B.3.5), we need to invoke the Remote Debugging Server before starting the debugging session. In this case, the Remote Debugging Server will interface with the GNU Debugger (GDB) Client built into Eclipse. Invoking the External GDB Server can be done from within Eclipse using External Tool Launchers.

External Launchers are stored in a common user location for Eclipse. The default path for stored Debugger Tool Launchers can be found in:

\[\text{${workspace}/.metadata/.plugins/org.eclipse.debug.core/.launches}\]

There are two tool launchers needed, one to invoke the GDB server, and the second to start the debugging process for the actual program under development.

B.3.3.1 Configuring the External GDB Server

In Eclipse, we can launch external tools using the External Tools submenu, under Run->External Tools->External Tools Configuration..... We will name the GDB server launcher Launch_NXT-GDB-Server. The tool launches the armnxtgdbserver script which invokes the GDB server. Note the arguments ‘-vn’ passed to the GDB server to cause it to start up without waiting for the NXT program to be downloaded and activated on the NXT. The steps are given in Figures B.8, B.9, and B.10. The contents of the External Tool configuration file is found in Listing B.1.

\footnote{Launching armnxtgdbserver from inside Eclipse may not work reliably since the console output has to be processed by Eclipse and may block the server's operation. If that happens, use a separate terminal window to invoke armnxtgdbserver instead.}

Figure B.7: SConsolidator Target View
Figure B.8: External Tools Configuration: Main

Figure B.9: External Tools Configuration: Environment
Figure B.10: External Tools Configuration: Common

Listing B.1: Configuration for GDB Server Launcher

```xml
<launchConfiguration type="org.eclipse.ui.externaltools.ProgramLaunchConfigurationType">
  <mapAttribute key="org.eclipse.debug.core.environmentVariables">
    <mapEntry key="PATH" value="/usr/local/yagarto/bin:/opt/local/bin:${env_var:PATH}"/>
  </mapAttribute>
  <listAttribute key="org.eclipse.debug.ui.favoriteGroups">
    <listEntry value="org.eclipse.ui.externaltools.launchGroup"/>
  </listAttribute>
  <stringAttribute key="org.eclipse.ui.externaltools.ATTR_LAUNCH_CONFIGURATION_BUILD_SCOPE" value="${none}"/>
  <stringAttribute key="org.eclipse.ui.externaltools.ATTR_LOCATION" value="${workspace_loc:/nxos-armdebug/scripts/armnxtgdbserver}"/>
  <stringAttribute key="org.eclipse.ui.externaltools.ATTR_TOOL_ARGUMENTS" value="-vn"/>
  <stringAttribute key="org.eclipse.ui.externaltools.ATTR_WORKING_DIRECTORY" value="${workspace_loc:/nxos-armdebug}"/>
</launchConfiguration>
```

B.3.3.2 Configuring the Remote Debugging Launcher

In order to run the actual program being developed, a Debug Configuration needs to be defined under Run->Debug Configurations.... This launcher specifies the executable file name, and path to be accessed for the source file display. The launcher is created by first selecting the C/C++ Remote Application Configuration Type, and clicking on the New Configuration button. The Preferred Launcher should be set to “GDB (DS) Manual Remote Debugging Launcher” Type (found via “Select Other...” at the bottom of the dialog screen), according to Figure B.11. Then, the name of the Remote Debugging Launcher configuration and Application path should be set accordingly (Figure B.12) via the Main tab. For NxOS applications downloaded via NeXT Tool and executed from RAM, we’d use the RXE formatted files. Note that there are two RXE format files among the available
binaries. The ELF file (*.rxe.elf) contains the object code and debugging symbols understood by GDB, while the NXTBINARY file (*.rxe) is the Native binary application used by the NXT. Note that we will need to define a different Debug Configuration for each Application we’re building and testing on the NXT, especially if the application binary has a different name and/or is located in a different directory. Otherwise the Application path will need to be modified each time we want to debug a different application.

After that is done, the program statement (label) to stop the program (‘break’ in the example) when starting the debugging session, and the cross-compiled GDB client executable path needs to be specified under the Debugger Main tab (Figure B.13). The NxOS application is set to stop at the label ‘break’ because the application has to be started manually on the NXT first before starting the Remote Debugging session since Eclipse does not know how to download and run applications on the NXT. The NXT starts execution from ‘main’, then encounters the first manually defined breakpoint in the code, upon which it will stop execution of the user program and enter Debug mode to wait for inputs from the remote GDB client. When the GDB Remote Debugging session is initiated, the remote GDB client queries the NXT for its status, after which it will set a new breakpoint for the label ‘break’ and continue execution until that label is reached in the code.

Since we’re using the GDB Remote Debugging protocol, we will also need to specify the protocol parameters via the Debugger Connection tab (Figure B.14). Finally miscellaneous settings are found under the Common tab, as shown in Figure B.15. The contents of a Debug Configuration file is found in Listing B.2.

Figure B.11: Debug Configuration: Select Preferred Launcher
Figure B.12: Debug Configuration: Main

Figure B.13: Debug Configuration: Debugger Main

273
Figure B.14: Debug Configuration: Debugger Connection

Figure B.15: Debug Configuration: Common
Listing B.2: Example Configuration for NxOS Application Launcher

```xml
<launchConfiguration type="org.eclipse.cdt.launch.remoteApplicationLaunchType">
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.AUTO_SOLIB" value="true"/>
  <listAttribute key="org.eclipse.cdt.dsf.gdb.AUTO_SOLIB_LIST"/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.DEBUG_NAME" value="/usr/local/yagarto/bin/arm-none-eabi-gdb"/>
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.DEBUG_ON_FORK" value="false"/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.DEV" value="/dev/ttyS0"/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.DEV_SPEED" value="115200"/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.GDB_INIT" value=""/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.HOST" value="localhost"/>
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.NON_STOP" value="false"/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.PORT" value="2828"/>
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.REMOTE_TCP" value="true"/>
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.REVERSE" value="false"/>
  <listAttribute key="org.eclipse.cdt.dsf.gdb.SOLIB_PATH"/>
  <stringAttribute key="org.eclipse.cdt.dsf.gdb.TRACEPOINT_MODE" value="TP_NORMAL_ONLY"/>
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.UPDATE_THREADLIST_ON_SUSPEND" value="true"/>
  <booleanAttribute key="org.eclipse.cdt.dsf.gdb.internal.ui.launching.RemoteApplicationCDebuggerTab.DEFAULTS_SET" value="true"/>
  <intAttribute key="org.eclipse.cdt.launch.ATTR_BUILD_BEFORE_LAUNCH_ATTR" value="0"/>
  <stringAttribute key="org.eclipse.cdt.launch.COREFILE_PATH" value=""/>
  <stringAttribute key="org.eclipse.cdt.launch.DEBUGGER_ID" value="gdbserver"/>
  <stringAttribute key="org.eclipse.cdt.launch.DEBUGGER_START_MODE" value="remote"/>
  <booleanAttribute key="org.eclipse.cdt.launch.DEBUGGER_STOP_AT_MAIN" value="true"/>
  <stringAttribute key="org.eclipse.cdt.launch.DEBUGGER_STOP_AT_MAIN_SYMBOL" value="break"/>
  <stringAttribute key="org.eclipse.cdt.launch.PROGRAM_NAME" value="/Users/tcmac/gitrepo/nxos-armdebug/nxos/systems/armskel/armskel_rxe.elf"/>
  <stringAttribute key="org.eclipse.cdt.launch.PROJECT_ATTR" value="nxos-armdebug"/>
  <booleanAttribute key="org.eclipse.cdt.launch.PROJECT_BUILD_CONFIG_AUTO_ATTR" value="true"/>
  <stringAttribute key="org.eclipse.cdt.launch.PROJECT_BUILD_CONFIG_ID_ATTR" value="0.1550615571"/>
  <booleanAttribute key="org.eclipse.cdt.launch.remote.RemoteCDSFDebuggerTab.DEFAULTS_SET" value="true"/>
  <stringAttribute key="org.eclipse.debug.core.ATTR_GDBSERVER_COMMAND" value="gdbserver"/>
  <stringAttribute key="org.eclipse.debug.core.ATTR_GDBSERVER_PORT" value="2345"/>
  <stringAttribute key="org.eclipse.debug.core.ATTR_PRERUN_CMDS" value=""/>
  <booleanAttribute key="org.eclipse.debug.core.ATTR_SKIP_DOWNLOAD_TO_TARGET" value="false"/>
  <stringAttribute key="org.eclipse.debug.core.ATTR_TARGET_PATH" value=""/>
  <listAttribute key="org.eclipse.debug.core.MAPPED_RESOURCE_PATHS">
    <listEntry value="/nxos-armdebug"/>
  </listAttribute>
  <listAttribute key="org.eclipse.debug.core.MAPPED_RESOURCE_TYPES">
    <listEntry value="4"/>
  </listAttribute>
  <mapAttribute key="org.eclipse.debug.core.preferred_launchers">
    <mapEntry key="[debug]" value="org.eclipse.cdt.dsf.gdb.launch.remoteCLaunch"/>
  </mapAttribute>
  <listAttribute key="org.eclipse.debug.ui.favoriteGroups">
    <listEntry value="org.eclipse.debug.ui.launchGroup.debug"/>
  </listAttribute>
  <stringAttribute key="org.eclipse.dsf.launch.MEMORY_BLOCKS" value="<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<memoryBlockExpressionList context="reserved-for-future-use"/>
"/>
  <stringAttribute key="process_factory_id" value="org.eclipse.cdt.dsf.gdb.GdbProcessFactory"/>
</launchConfiguration>
```

B.3.4 Git Version Control

Git [71] was developed for the Linux Kernel Project as a tool to keep track of software changes and revisions. It is also used for retrieving the NxOS project from the repository, so some basic knowledge of Git is helpful to understand how to retrieve and keep your NxOS software package up to date. Eclipse has EGit support built in starting from the Indigo Release, so it is possible to perform many of the functions via the Eclipse Context Menus.

B.3.4.1 Command Line Git

Basic Git Commands are as follows:

- **clone**: create a local copy of the repository
- **add**: add a new file to the local repository
- **commit**: check in a changed file to the local repository
- **fetch**: retrieve changes from remote repository only (does not merge with working copy)
• push: update changes from local repository to remote repository
• pull: retrieve changes from remote repository to local repository, merge with working copy
• checkout: specify version of local repository to make active

B.3.4.2 Eclipse EGit

Eclipse EGit support is accessible from the Project Context Menu (Figure B.16).

Figure B.16: Eclipse EGit Context Menu

B.3.5 Debugging using GDB

The GNU Debugger (GDB) is a command driven debugger client that can be used to debug the cross-compiled application running on the NXT. In a cross-development environment, the NXT is actually running a different instruction set and may be entirely different from the Host. Consequently, the GDB client we use must be a cross-development debugger (in our case, the GDB client should be named arm-none-eabi-gdb). Alternatively, Graphical Front-ends such as the Eclipse CDT Debugger integrates with GDB to control the execution of the application. The main difference between normal GDB debugging and NXT-based GDB debugging is that the application

---

5This command is only valid for repositories that you create and has read-write access to. It is not a valid command for the public read-only NxOS repository.
is not running on the same host as the GDB client. This is termed Remote Debugging. Traditionally, remote debugging allows the GDB client to debug an application running on a different computer, connected via a serial connection. There should also be a GDB server process which understands the debugging commands sent by the GDB client via the serial connection, and control the application accordingly, to start, stop, examine variables and step through the instructions. For many current embedded devices such as the NXT, serial connections are no longer used; this necessitates the use of a GDB server which interposes between the GDB Client and the embedded device, converting the debugging commands from the GDB client to a format which can be sent via the USB connection.

B.3.5.1 Hardware Based GDB Remote Debugging

The microcontroller in many embedded systems such as the NXT has hardware debugging support, implemented using a Joint Test Action Group (JTAG) interface. This interface is a separate hardware debugging interface which is not enabled by default in the NXT. A JTAG Header (hardware connection interface) would need to be soldered to the circuit board of the NXT in order to enable JTAG Debugging. Furthermore, JTAG Debugging requires the use of a JTAG Emulator or Adapter which can be expensive. Since the focus of this book is on learning software development, the JTAG based approach is not desirable.

B.3.5.2 Software Based GDB Remote Debugging

A software based solution for GDB Remote Debugging requires a GDB stub module to be running on the embedded system. This GDB stub module interprets the GDB commands sent from the GDB client and performs the appropriate actions. The disadvantage of using a software based solution is that some types of routines, such as Interrupt Service Routines, cannot be debugged reliably, since it would interfere with the driver providing the remote connection support. The greatest advantage is that there is no need for additional hardware components such as the JTAG Adapter; the GDB debugging session is done entirely via USB. Nonetheless, since the GDB Client does not understand how to access USB devices, we must go through a GDB server which accepts the debugging commands and then send it to the NXT via the USB link. A GDB stub module must also be running in the NXT which can understand the GDB commands forwarded by the GDB server, and perform the necessary Debugging actions. Consequently, the Debugging Setup for the NXT looks like the setup in Figure B.17.

\[\text{It should be noted that the GDB Stub was developed from scratch for the NXT platform, and the initial develop of the GDB stub (called Armdge) necessitated the use of the hardware based JTAG debugging tools in order to debug the GDB stub code itself. Fortunately once the GDB stub is done, it is no longer necessary to use the JTAG debugging tools for normal application debugging.}\]
B.3.5.3 Eclipse CDT Debugger with Armdebug

The Debug Environment Layout is configured by selecting the Debug Perspective from the top Toolbar. This brings up the Debug Control Window, Variable and Register Display Window, as well as Source and Disassembly Windows (Figure B.18).

After enabling the Debug Perspective, the next step is to connect the USB cable to the NXT, download the NxOS application using NeXT Tool, and run the NxOS Application on the NXT via NeXT Tool or the NXT LCD Menu. The NeXT Tool must be exited before the Debugging process begins since the Debugger needs access to the USB connection.

The External Tool Launcher is then activated to invoke the External Debugger Launcher, which runs the GDB Server process to connect to the NxOS application on the NXT. Finally, the Remote Debugging Launcher is configured for the correct application, and launched to start the actual Debugging session.

Buttons to Continue Execution, Step, and Quit GDB Remote debugging are provided in the Debug Control Window Sub-toolbar. In addition, the Routine Call Hierarchy is also indicated in the Debug Control Window. Variables and Register Values can be examined and modified via the Variable and Register Display Window. Finally, Breakpoints can be Set and Cleared using the Source Window.
B.3.5.4 Text-based GDB Commands

The following are commonly used GDB commands when debugging using the command line GDB client:

- `help <command>`: provide some basic information about the given command in GDB
- `target remote`: specify the GDB server process to connect to
- `list`: show the contents of the source file where the current execution is halted
- `disassemble`: show the assembly language instructions by reading contents of memory and interpreting them as instructions
- `info registers`: show the contents of registers for the processor
- `x`: examine contents of memory
- `backtrace`: show routine calling hierarchy (requires programs to follow C-style stack frame structure)
- `break`: set breakpoint
- `step`: execute one instruction from the current instruction address location, and return control to GDB
- `continue`: continue execution from the current instruction address location
- `detach`: disconnect from the remotely debugged application, which continues execution
- quit: exit GDB, abort execution of the target application if necessary

Detailed usage and all text-based commands are documented in [72].

### B.4 Cross-Development Toolchain Commands

Since ARM Assembly Language programming is done on a cross-development platform, the GNU Compiler, Assembler and Binutils tools are often named differently from the default names to avoid confusion. Most of the GNU cross-development toolchains will use the same names. The list of tools found in the most GNU Cross Compiler Toolchains are as follows:

<table>
<thead>
<tr>
<th>GNU Program</th>
<th>YAGARTO Name</th>
<th>Alternate Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert Addresses into Linenum/Filename</td>
<td>arm-none-eabi-add2line</td>
<td></td>
</tr>
<tr>
<td>Library Archiver</td>
<td>arm-none-eabi-ar</td>
<td></td>
</tr>
<tr>
<td>Assembler</td>
<td>arm-none-eabi-as</td>
<td></td>
</tr>
<tr>
<td>C++ Compiler</td>
<td>arm-none-eabi-g++</td>
<td>arm-none-eabi-c++</td>
</tr>
<tr>
<td>C++ Symbol Demangler</td>
<td>arm-none-eabi-c++filt</td>
<td></td>
</tr>
<tr>
<td>C Preprocessor</td>
<td>arm-none-eabi-cpp</td>
<td></td>
</tr>
<tr>
<td>C Compiler</td>
<td>arm-none-eabi-gcc</td>
<td></td>
</tr>
<tr>
<td>GCC Bug Reporter (?)</td>
<td>arm-none-eabi-gcov</td>
<td></td>
</tr>
<tr>
<td>Coverage Testing Tool</td>
<td>arm-none-eabi-gcov</td>
<td></td>
</tr>
<tr>
<td>Debugger</td>
<td>arm-none-eabi-gdb</td>
<td>arm-none-eabi-gdbtui</td>
</tr>
<tr>
<td>Profiler</td>
<td>arm-none-eabi-gprof</td>
<td></td>
</tr>
<tr>
<td>Linker</td>
<td>arm-none-eabi-1d</td>
<td></td>
</tr>
<tr>
<td>Symbol Table Lister</td>
<td>arm-none-eabi-nm</td>
<td></td>
</tr>
<tr>
<td>Binary Object Copier</td>
<td>arm-none-eabi-objcopy</td>
<td></td>
</tr>
<tr>
<td>Disassembler</td>
<td>arm-none-eabi-objdump</td>
<td></td>
</tr>
<tr>
<td>Update Library Archive Table of Contents</td>
<td>arm-none-eabi-ranlib</td>
<td></td>
</tr>
<tr>
<td>Display Contents of ELF Binary Files</td>
<td>arm-none-eabi-readelf</td>
<td></td>
</tr>
<tr>
<td>ARM Simulator (?)</td>
<td>arm-none-eabi-run</td>
<td></td>
</tr>
<tr>
<td>Print Size of Sections in Binary File</td>
<td>arm-none-eabi-size</td>
<td></td>
</tr>
<tr>
<td>Find Strings in Binary File</td>
<td>arm-none-eabi-strings</td>
<td></td>
</tr>
<tr>
<td>Remove Symbols</td>
<td>arm-none-eabi-strip</td>
<td></td>
</tr>
</tbody>
</table>

### B.5 Comparison of Assembler Directives

The original ARM Development Suite (ADS) Assembly Language syntax is defined by ARM [58]. Since the GNU Assembler (GAs) is a multi-platform assembler, it uses its own syntax and consequently understanding programs written using the ARM syntax and converting it to GAs syntax is essential. The definitive guide to GNU Assembler is [25]. However, since it was written to accommodate many different CPU architectures, it is difficult to determine which GAs directives maps to which specific ARM syntax directives. A good summary of the GNU Assembler Directives for the ARM Architecture is found in [73]. The following tables compares the common
directives used in GAs and IAR EWARM. Note: Both upper case and lower case names are equivalent, but not mixed case (i.e. sp or SP is allowed, but not Sp or sP). In the following tables, arguments are identified using angled brackets (‘<’ and ‘>’), while optional arguments are identified using curly braces (‘{’ and ‘}’). A variable number of arguments is indicated using ellipses (‘...’)

### B.5.1 Register Names

The use of Register Names in GAs follows a similar syntax to that used in the ARM Development Suite. Several aliases to registers with predefined usage are also specified to make it easy to recognize the role of the register as an instruction operand. This is summarized in Table B.3.

<table>
<thead>
<tr>
<th>Register Names</th>
<th>GNU Assembler</th>
<th>ADS Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 - R15</td>
<td>R0 - R15</td>
<td>R0 - R15</td>
<td>GAs accepts %r0 - %r15</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>Frame Pointer (R11), GAs accepts %fp</td>
</tr>
<tr>
<td>IP</td>
<td>IP</td>
<td>IP</td>
<td>Intra-Procedure call Scratch Register (R12), GAs accepts %ip</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>Stack Pointer (R13), GAs accepts %sp</td>
</tr>
<tr>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>Link Register (R14), GAs accepts %lr</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>Program Counter (R15), GAs accepts %pc</td>
</tr>
<tr>
<td>xPSR</td>
<td>xPSR, xPSR_all, xPSR_cxsf</td>
<td>xPSR</td>
<td>x = C/S; For CPSR, GAs accepts $psw xPSR_all deprecated</td>
</tr>
<tr>
<td>xPSR Control Fields</td>
<td>xPSR_c</td>
<td></td>
<td>Control field mask for xPSR, GAs accepts xPSR_ctl (deprecated)</td>
</tr>
<tr>
<td>xPSR Extension Fields</td>
<td>xPSR_x</td>
<td></td>
<td>Extension field mask for xPSR</td>
</tr>
<tr>
<td>xPSR Status Fields</td>
<td>xPSR_s</td>
<td></td>
<td>Status field mask for xPSR</td>
</tr>
<tr>
<td>xPSR Flags Fields</td>
<td>xPSR_f</td>
<td></td>
<td>Flag field mask for xPSR</td>
</tr>
<tr>
<td>xPSR Combination</td>
<td>xPSR_cx, xPSR_fs, etc.</td>
<td></td>
<td>Any combination of field masks can be used together</td>
</tr>
</tbody>
</table>

### B.5.2 General Assembler Directives

Assembler directives provide the Assembler with clues regarding what is found in the source file. The list given in Table B.4 is not meant as a comprehensive guide to all possible Assembler directives. More detailed information regarding GAs directives are found in [25].
<table>
<thead>
<tr>
<th>Purpose</th>
<th>GNU Assembler</th>
<th>ADS Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include Files</td>
<td>.include &quot;&lt;filename&gt;&quot;</td>
<td>INCLUDE &quot;&lt;filename&gt;&quot;</td>
<td>Include file given by filename</td>
</tr>
<tr>
<td>Enable UAL Syntax</td>
<td>.syntax &lt;unified</td>
<td>divided&gt;</td>
<td>unified: UAL Syntax; divided: Old Thumb syntax</td>
</tr>
<tr>
<td>Select Instruction Set</td>
<td>.code &lt;no_of_bits&gt;</td>
<td>CODE16, CODE32</td>
<td>no_of_bits = (16 (Thumb), 32 (ARM)); GAs accepts .arm and .thumb</td>
</tr>
<tr>
<td>Force THUMB Mode</td>
<td>.force_thumb</td>
<td>Force THUMB code generation even if not supported</td>
<td></td>
</tr>
<tr>
<td>Mark Entry Point as THUMB Code</td>
<td>.thumb_func</td>
<td></td>
<td>Force BX entry into code. Needed for public Thumb functions (whether called from Thumb or ARM)</td>
</tr>
<tr>
<td>Define Named Constant</td>
<td>.equ &lt;sym_name&gt;, &lt;value&gt;</td>
<td>EQU</td>
<td>Define named constants</td>
</tr>
<tr>
<td>Define Unique Named Constant</td>
<td>.equiv &lt;sym_name&gt;, &lt;value&gt;</td>
<td></td>
<td>Define unique named constants. If symbol name exists, it is an error.</td>
</tr>
<tr>
<td>Assign Value to Variable</td>
<td>.set &lt;var_name&gt;, &lt;var_value&gt;</td>
<td>SETA</td>
<td>Set variable to given value</td>
</tr>
<tr>
<td>Define Named Registers</td>
<td>&lt;reg_name&gt; .req &lt;reg_name&gt;</td>
<td></td>
<td>Define alternate names for registers (e.g., sum .req R4)</td>
</tr>
<tr>
<td>Declare String</td>
<td>.ascii &quot;&lt;strings&gt;&quot;</td>
<td>DCB</td>
<td>Declare ASCII strings (non-terminated)</td>
</tr>
<tr>
<td>Declare C-String</td>
<td>.asciz &quot;&lt;strings&gt;&quot;</td>
<td>-</td>
<td>Declare C-Strings (NULL-terminated)</td>
</tr>
<tr>
<td>Declare Byte</td>
<td>.byte &lt;byte1&gt; {,&lt;byte2&gt;} ...</td>
<td>DCB</td>
<td>Declare Byte (8-bit) sized variables</td>
</tr>
<tr>
<td>Declare Halfword</td>
<td>.hword &lt;short1&gt; {,&lt;short2&gt;} ...</td>
<td>DCW</td>
<td>Declare Halfword (16-bit) sized variables</td>
</tr>
<tr>
<td>Declare Word</td>
<td>.word &lt;word1&gt; {,&lt;word2&gt;}</td>
<td>DCD</td>
<td>Declare Word (32-bit) sized variables</td>
</tr>
<tr>
<td>Reserve Storage Space</td>
<td>.space &lt;no_of_bytes&gt; {,&lt;fill_byte&gt;}</td>
<td>SPACE</td>
<td>Reserve space in Memory, fill with fill_byte value (default 0)</td>
</tr>
<tr>
<td>Reserve Literal Pool Space</td>
<td>.ltorg</td>
<td>LTORG</td>
<td>Reserve space for Literal Pool (32-bit constants for LDR); GAs accepts .pool</td>
</tr>
<tr>
<td>End of Program</td>
<td>.end</td>
<td>END</td>
<td>Marks the end of the assembler code in a file (subsequent lines are ignored)</td>
</tr>
<tr>
<td>Halt Assembly</td>
<td>.err</td>
<td></td>
<td>Halt Assembly with error message</td>
</tr>
<tr>
<td>Current Address</td>
<td>.</td>
<td></td>
<td>A space (‘ ’) must follow the period (‘.’)</td>
</tr>
<tr>
<td>Label (unique)</td>
<td>&lt;label_name&gt;:</td>
<td>&lt;label_name&gt;</td>
<td>GAs requires colon (‘:’) after label name</td>
</tr>
<tr>
<td>Local Label (non-unique)</td>
<td>&lt;n&gt;:</td>
<td>(routine) ROUT &lt;n&gt;:[routine]</td>
<td>GAs requires colon (‘:’) after local label. n = (0..9) for GAs, ref. as n(f3) routine: ADS ROUT scope name n = (0..99) for ADS, ref. as %n(F,B,A,T) f: forward, b: backward a: all macro levels, t: this macro level only</td>
</tr>
<tr>
<td>Comments</td>
<td>@ or # or /* */</td>
<td>;</td>
<td>GAs uses semicolon (‘;’) as a statement separator, it WILL process the rest of the line as a normal statement</td>
</tr>
</tbody>
</table>
B.5.3 Linker Related Assembler Directives

Linker directives are used to specify symbols, code, and data sections in order to link various object files together into a single executable. A list of commonly used Linker directives are given in Table [B.5]

Table B.5: Comparison of Linker Directives

<table>
<thead>
<tr>
<th>Purpose</th>
<th>GNU Assembler</th>
<th>ADS Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare Code/Data Sections</td>
<td>.section &lt;section_name&gt; {, &quot;&lt;flags&gt;&quot;)</td>
<td>AREA</td>
<td>section_name = [.text, .data, .bss] flags = (a,w,x)</td>
</tr>
<tr>
<td>Insert Padding in Data Section</td>
<td>.align &lt;n&gt;</td>
<td>ALIGN</td>
<td>&lt;n&gt;: number of low order bits for alignment. If &lt;n&gt; omitted, default is 2 (word alignment). This directive is compatible with other ARM Assemblers</td>
</tr>
<tr>
<td>Address Alignment</td>
<td>.balign &lt;power_of_2&gt; [{, &lt;fill_value&gt; [{, &lt;max_padding&gt; }]}</td>
<td>ALIGN</td>
<td>Set address alignment for generated code and data (GAs specific directive)</td>
</tr>
<tr>
<td>Public Symbols</td>
<td>.global &lt;symbol&gt;</td>
<td>EXPORT / GLOBAL</td>
<td>Export variables to other modules</td>
</tr>
<tr>
<td>Referenced Symbols</td>
<td>.extern &lt;symbol&gt;</td>
<td>IMPORT / EXTERN</td>
<td>GAs ignores .extern as any unrecognized symbol is treated as external symbols</td>
</tr>
<tr>
<td>Declare Symbol as Function Entry Point</td>
<td>.type &lt;symbol&gt;, %function</td>
<td>-</td>
<td>Needed for interworking, to generate proper veneer to switch between ARM and Thumb modes</td>
</tr>
</tbody>
</table>

B.5.4 Macros and Conditional Assembly Directives

Macros are very effective for instruction synthesis and defining commonly used sequence of code which can be easily mistyped or implemented incorrectly. The directives used to define macros are given in Table [B.6]
### Table B.6: Macros and Conditional Assembly Directives

<table>
<thead>
<tr>
<th>Purpose</th>
<th>GNU Assembler</th>
<th>ADS Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Macro</td>
<td>.macro &lt;name&gt;</td>
<td>MACRO</td>
<td>Start Macro Definition Block. Substitute \langle_&lt;arg_1&gt;<em>\rangle \langle</em>&lt;arg_2&gt;<em>\rangle \ldots \langle</em>&lt;arg_N&gt;_\rangle in code with argument list when invoked in program. Use .endm to terminate block.</td>
</tr>
<tr>
<td>End Macro</td>
<td>.endm</td>
<td>MEND</td>
<td>End Macro Definition Block</td>
</tr>
<tr>
<td>Exit Macro</td>
<td>.exitm</td>
<td>MEXIT</td>
<td>Exit Macro Block before .endm</td>
</tr>
<tr>
<td>Conditional If</td>
<td>.if &lt;logical_expression&gt;</td>
<td>IF</td>
<td>Start Conditional Assembly if expression is TRUE. Stop Assembly when .endif or .else encountered</td>
</tr>
<tr>
<td>Symbolic If</td>
<td>.ifdef &lt;symbol&gt;</td>
<td></td>
<td>Start Conditional Assembly if symbol is defined</td>
</tr>
<tr>
<td>Negated Symbolic If</td>
<td>.ifndef &lt;symbol&gt;</td>
<td></td>
<td>Start Conditional Assembly if symbol is NOT defined</td>
</tr>
<tr>
<td>Conditional Else</td>
<td>.else</td>
<td>ELSE</td>
<td>Start Conditional Assembly if the logical expression for the .if Block is FALSE.</td>
</tr>
<tr>
<td>Conditional End</td>
<td>.endif</td>
<td>ENDF</td>
<td>End Conditional Assembly Block</td>
</tr>
<tr>
<td>Repeat Code Block Start</td>
<td>.rept &lt;number_of_times&gt;</td>
<td>WHILE</td>
<td>Repeat code block generation for number_of_times until .endr encountered</td>
</tr>
<tr>
<td>Repeat Code Block with Parameter Substitution</td>
<td>.irp &lt;param&gt; [,&lt;val_1&gt;],&lt;val_2&gt; ...</td>
<td></td>
<td>Repeat code block generation for each item in parameter value list, substituting \langle&lt;param&gt;_\rangle in code block with value in the parameter list, until .endr encountered</td>
</tr>
<tr>
<td>Repeat Code Block End</td>
<td>.endr</td>
<td>WEND</td>
<td>End repeated code generation block definition</td>
</tr>
</tbody>
</table>

### B.5.5 Common GAs Command Line Options

Additional control over the outputs generated by the GNU Assembler is controlled via command line options passed to GAs. The list of commonly used options is given in Table B.7.

#### Table B.7: Common Command Line Options for GAs

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Listing File</td>
<td>-ahls</td>
<td>Default options</td>
</tr>
<tr>
<td>Examine Macro Substitutions</td>
<td>-ahlsm</td>
<td>Expand Macros in listing</td>
</tr>
<tr>
<td>Specify Listing Filename</td>
<td>-a=&lt;listing_file&gt;</td>
<td>Must be last specified -a option</td>
</tr>
<tr>
<td>Generate Debug Information</td>
<td>-g</td>
<td></td>
</tr>
<tr>
<td>Specify Object File Name</td>
<td>-o &lt;object_file&gt;</td>
<td>If not specified, default is a.out</td>
</tr>
</tbody>
</table>
B.5.6 Assembling ARM Source Files with the C Preprocessor

Normally we would use the platform build tool such as scons to build NxOS. However, it is useful to know the commands used to build the individual source files so that we know what is happening behind the scene.

Since the LEGO NXT uses the Atmel AT91SAM7S256 microcontroller, we need to include the definitions of the various peripheral register addresses and register definitions in order to access them from our Assembly Language source files. Atmel provides the necessary header files for this purpose [74]. However, the header files are written mainly for C Language development using C Preprocessor syntax, with support for assembly language modules. Consequently, in order to make use of the AT91SAM7S256.h header file in assembly, we need to assemble the .S files using GCC instead. GCC can process assembly language input files, which is sent to GAs for actual processing. The syntax for building the assembly language source files which uses the AT91SAM7S256.h header file is as follows [75]:

```
arm-none-eabi-gcc -D__ASSEMBLY__ -x assembler-with-cpp -g -mcpu=arm7tdmi -c -Os -Wall -I./include <file>.s -o <file>.o
```

The __ASSEMBLY__ Define statement is needed to select the appropriate definitions in the AT91SAM7S256.h header file, while the ‘-x’ option informs gcc to process the file as an Assembly Language input file (gcc will try to automatically determine the file type via the file extension ‘.s’).

B.6 Executable Program Linking

The GNU Linker (GLd), is an essential part of the cross-assembly toolchain, since it converts object code into proper executable programs located and referenced at appropriate addresses in memory. In addition, it allows for instructions to be packed within the executable program file at a different address from the actual execution address, in order to enable the Runtime Loader to relocate the instructions to the correct address when required. A good introduction to the use of the GNU Linker is found in [76]. The official reference to GLd maintained by the Free Software Foundation is [77].

GLd defines Sections which typically contain either blocks data or code. 

Loadable Sections are loaded into memory at specified addresses when the executable output file is run. Allocated sections cause the Loader to reserve space in memory but not load any data there, other than initialize it to a default value (if specified). Other sections are used to store debugging information and are not Loadable nor Allocatable. Each Loadable or Allocated Section is mapped to a Virtual Memory Address (VMA) as well as a Load Memory Address (LMA). The VMA specifies the addresses that will be used for the Section when the program is executed, whereas the LMA specifies the address of the Section when the program is first loaded into memory. By default, LMA is the same as VMA except when a program is stored in Flash or ROM and then copied to RAM for execution or initialization of global variables. The ROM address in this case is the LMA, whereas the RAM address will be the VMA.
B.6.1 Interaction of Linker and the NXT Firmware

Since the approach taken in this book is to write programs that execute from RAM, we will need to deal with LMA and VMA configuration in the Linker Script. In addition, a Runtime Loader stub routine must be included to copy the program from Flash to the proper RAM locations before actual execution begins. This Runtime Loader stub is linked into each executable because the Enhanced NXT Firmware looks for the native program identifier string “NXTBINARY” in the RXE (NXT runtime executable) file header and will pass control to a fixed offset location within the RXE file without any further environment setup [13]. The RXE file header contents showing the “NXTBINARY” string is shown in the file dump below:

```
$ od -c tribot.rxe |head
0000000  N X T B I N A R Y \0 \0 \0 020 \0 \0 \0 020 000 001
0000020  001 \0 351 \0 \0 017 341 300 \0 200 343 \0 360 \0 001
0000040  001 \0 275 350 001 200 340 , 020 237 345 001 \0 002
```

Consequently it is the responsibility of the Runtime Loader stub routine to do the necessary loader setup tasks, which is to copy the executable code (minus the header and loader stub) to RAM, then pass control to the initialization routine for the executable. For the Atmel SAM7S256 microcontroller, the RAM start address is initially mapped to 0x0020 0000. Once the code has been copied from Flash to RAM, execution continues by forcing a software RESET by jumping to the RESET vector in RAM at 0x0020 0000. The RESET vector routine will then remap the RAM to start from 0x0000 0000 which is where the ARM processor expects the Exception Vectors to be stored [13]. After the address remap, other system initialization will be performed. Finally the nx__kernel_main() routine will be called to execute the actual program code.

![Figure B.19: Runtime Loader Execution](image_url)

B.6.2 GLd Directives

The GNU Linker has a long history, supporting many different CPU architectures and assembler generations. Consequently, its syntax and options

---

The GNU Linker has a long history, supporting many different CPU architectures and assembler generations. Consequently, its syntax and options

---

7RAM address locations can be accessed either at their original physical addresses starting from 0x0020 0000 or at the remapped addresses starting from 0x0000 0000.
might not be so intuitive since it does not cater only to the ARM architecture requirements. The most important thing to understand in terms of using GLd is the typical format of the Linker Script, which controls the executable generation process, and the directives that are used along with it. Table B.8 lists the most commonly used directives that affect the generation of the executable file.

### Table B.8: GLd Directives

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Directive</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTIONS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMORY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENTRY(&lt;symbol&gt;)</td>
<td></td>
<td>Specify the Entry Point (routine name)</td>
</tr>
<tr>
<td>File Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INCLUDE &lt;file&gt;</td>
<td></td>
<td>Include contents of file</td>
</tr>
<tr>
<td>INPUT (&lt;file&gt; [, &lt;file&gt;] ...)</td>
<td>Link listed object file(s), useful for startup stub code.</td>
<td></td>
</tr>
<tr>
<td>GROUP (&lt;file&gt; [, &lt;file&gt;] ...)</td>
<td>Link listed archive(s) into the executable.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Comma (’,’) as separator is optional.</td>
</tr>
<tr>
<td>AS_NEEDED(&lt;file&gt; [, &lt;file&gt;] ...)</td>
<td>Must be placed inside the INPUT or GROUP file list.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object code in file is added only if actually referenced (called).</td>
</tr>
<tr>
<td>Object File</td>
<td>OUTPUT_FORMAT(&lt;default&gt;, &lt;big&gt;, &lt;little&gt;)</td>
<td>Specify the output format (endianness) for big (-EB), little (-EL), and default (no Linker switch specified)</td>
</tr>
<tr>
<td>Formats</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUT_ARCH(&lt;arch&gt;)</td>
<td>Specify the machine architecture arch for the output file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In our case, use arm.</td>
</tr>
<tr>
<td></td>
<td>ABSOLUTE(&lt;expr&gt;)</td>
<td>Returns the absolute (non-relative) value of the expression</td>
</tr>
<tr>
<td></td>
<td>ADDR(&lt;section&gt;)</td>
<td>Returns the absolute address (VMA) of the section</td>
</tr>
</tbody>
</table>

**B.6.3 Simple GLd Script**

Linker Scripts are rather esoteric definition files, and must be correctly defined in order for the Runtime Loader to recognize the executable file created by the Linker. Consequently it is probably easier to use existing scripts as a template for modification rather than write one from scratch.

Nonetheless, a simple GLd script example is given in Listing B.3 to illustrate how the directives are used. Further information regarding linker scripts are found in [77]. Actual linker scripts used by NxOS is found in the nxos/systems subdirectory and should be used as the basis for creating your own linker scripts for use with the NXT.
Listing B.3: Contents of a simple GLd script

SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}

B.6.4 Common GLd Command Line Options

The output of the Linker can be tailored using various command line options. The common options are given in Table B.9.

Table B.9: Common Command Line Options for GLd

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Listing File</td>
<td>-T &lt;linker_script_file&gt;</td>
<td>Specify a Linker Script File</td>
</tr>
<tr>
<td>Specify Library Path</td>
<td>-L &lt;path&gt;</td>
<td>Specify location of library (*.a)</td>
</tr>
<tr>
<td>Specify Library File Name</td>
<td>-l &lt;library_file&gt;</td>
<td>Library name (lib&lt;library_file&gt;)</td>
</tr>
<tr>
<td>Specify Object File Name</td>
<td>-o &lt;executable_file&gt;</td>
<td>If not specified, default is a.out</td>
</tr>
</tbody>
</table>
Appendix C

NxOS Organization

‘The wonderful thing about standards is that there are so many of them to choose from.’
attributed to Grace Murray Hopper, 1906-1992

NxOS is an open source Robotic Operating System (ROS) for the LEGO MINDSTORMS NXT [28]. While there are several alternative ROS available for the NXT, the focus of NxOS is to provide a minimal platform for writing, developing, and testing native software running on the NXT. NxOS is hosted on github.com. This is an open source repository used for various projects, including NxOS. The version of NxOS used for this book is based on the nxos-armdebug project. This version of NxOS includes GDB debugging support to enable remote debugging of ARM code on the NXT.

C.1 Obtaining NxOS-Armdebug

nxos-armdebug can be retrieved in two ways, either via POSIX command line using git, or directly via the Eclipse IDE.

C.1.1 Retrieving NxOS-Armdebug via command line

On Mac OS X or Linux, the Terminal / Console provide the standard POSIX command line access to git, which is used to access the github.com source repository.

    git clone git://github.com/tcwan/nxos-armdebug.git

This will download the nxos-armdebug code into the current working directory on your system.
C.1.2 Importing NxOS-Armdebug Into Eclipse

C.1.2.1 Importing Existing NxOS-Armdebug project

If nxos-armdebug has been downloaded to your system via other means, such as the method described in Section C.1.1, then File->Import can be used, then selecting “Existing Project into Workspace” (Figure C.1).

Git support from within Eclipse may not be enabled if Import Existing Project were used. If the nxos-armdebug project does not indicate the project status via Git specific icons within the Eclipse project tree window, it could be added via the Project Context Menu (right click on the project folder), Team->Share Project... dialog (Figure C.2). Alternatively, the project repository can be updated via the command line using “git pull” from within the nxos-armdebug directory, which will retrieve any changes to the github project repository to the local system.

C.1.2.2 Importing NxOS Directly from GitHub

However, if NxOS does not exist on your system, then we can use the Git plugin within Eclipse to perform the import directly using File->Import and selecting “Projects from Git” (Figure C.3).

The URI option (Figure C.4) is required to access github.com to retrieve the nxos-armdebug repository (Figure C.5), located at:

git://github.com/tcwan/nxos-armdebug.git
Figure C.2: Team Share Project Dialog

Figure C.3: Import Projects from Git
After the repository has been downloaded and expanded, Eclipse prompts to confirm the Import (Figure C.6). After the completion of this step, the nxos-armdebug project will be accessible via Eclipse.

C.2 Configuring Eclipse for NxOS

C.2.1 Enabling scons support

In order to build the project from within Eclipse via Project->Build Project, it is necessary to specify a non-default project builder, since the build tool used by NxOS is scons and not the default make. There are two ways of accessing scons from within Eclipse. We can either define a custom Build Tool, or else, use SConsolidator [64] which provides much finer grain control over scons, including the ability to define multiple build targets interactively. Here we will look at how to setup the custom build tool first, then describe the setup for SConsolidator. In order to setup SConsolidator, the plugin must first be installed into Eclipse following the instructions in Section B.3.1.

C.2.1.1 Configuring the Custom Project Build Tool

A Custom Project Build Tool can be created via Project->Properties, and selecting the “Builders” option (Figure C.7). A new configuration is created (named scons), with the properties as shown in Figures C.8, C.9 and C.10. We need to provide the arguments ‘-C nxos’ to scons since the SConstruct file used to control the build process is not found in the project root level directory. The actual path of scons (Figure C.8) as well as the executable search path for the cross compiler toolchain (arm-none-eabi-*)
Figure C.5: Git Source Repository URI

Figure C.6: Git Confirm Import Project
Figure C.7: Custom Project Builder Configuration: Main

should be adjusted accordingly. In Figure C.10, the cross compiler toolchain used is from MacPorts and hence is found in /opt/local/bin.

These dialogs generate the launcher description file which is found in the specified project’s top level directory, under:

```bash
${projpath}/.externalToolBuilders/
```

An example based on the NxOS scons build tool is given in Listing C.1.

Listing C.1: Configuration for Scons Launcher

```xml
<launchConfiguration type="org.eclipse.ui.externaltools.ProgramBuilderLaunchConfigurationType">
    <stringAttribute key="org.eclipse.debug.core.ATTR_REFRESH_SCOPE" value="${project}"/>
    <mapAttribute key="org.eclipse.debug.core.environmentVariables">
        <mapEntry key="PATH" value="/opt/local/bin:${env_var:PATH}"/>
    </mapAttribute>
    <booleanAttribute key="org.eclipse.debug.ui.ATTR_LAUNCH_IN_BACKGROUND" value="false"/>
    <stringAttribute key="org.eclipse.ui.externaltools.ATTR_LOCATION" value="/opt/local/bin/scons"/>
    <stringAttribute key="org.eclipse.ui.externaltools.ATTR_RUN_BUILD_KINDS" value="full,incremental,full,incremental"/>
    <stringAttribute key="org.eclipse.ui.externaltools.ATTR_TOOL_ARGUMENTS" value="-C nxos"/>
    <booleanAttribute key="org.eclipse.ui.externaltools.ATTR_TRIGGERS_CONFIGURED" value="true"/>
    <stringAttribute key="org.eclipse.ui.externaltools.ATTR_WORKING_DIRECTORY" value="${workspace_loc:/nxos-armdebug}"/>
</launchConfiguration>
```

One disadvantage of using a custom build tool is that the project build targets (binary applications to build) can be changed easily. Instead, it needs to be specified as part of the argument list in Figure C.8. However, if you’re comfortable with invoking scons via the command line, then this would not be such a hassle. By default, if a build target was not explicitly specified, it will build the last specified build target. The last specified build target is cached by scons in:

```bash
${projpath}/nxos/scons.options
```
Figure C.8: Custom Project Builder Configuration: Main

Figure C.9: Custom Project Builder Configuration: Environment
C.2.1.2 Configuring SConsolidator

After importing the project, configure the per-project scons build parameters via the Project->Properties dialog. You would normally need to select the NxOS project on the left panel in order to access the correct Project Properties. First, check that SConsolidator has been selected as the Builder for the project (Figure C.11).

Select SCons in the dialog box, and fill in the following information (Figure C.12). The Starting Directory may need to be specified explicitly, in that case, this is the path to the root of the NxOS project tree. We need to specify ‘-C nxos’ as the SCons Options since the SConstruct file is not in the root level of the project tree. In addition, we need to override the default environment PATH variable to specify the location of the cross-compilation toolchain, via ‘PATH=<toolchain_paths>:PATH’. Multiple paths are separate by the colon ‘:’ character. In this example, the YAGARTO toolchain is used, which is found in /usr/local/yagarto/bin in my setup and may differ depending on the toolchain version and where it is installed. In addition, I need to include the path to the scons project build tool and the doxygen project documentation tool, which came from MacPorts and is found in /opt/local/bin.

Details regarding how to build NxOS and execute the code on the NXT is given in Section B.3.

After the SConsolidator settings have been saved, the newly imported project needs to be refreshed. This is an important step because otherwise, SConsolidator would not be able to store its configuration correctly for the newly imported project. To refresh the project, the Project Context Menu Scons->Refresh C/C++ project(s) from SCons can be used, other-

\[^{1}\text{SConsolidator 0.4.0 stores its configuration in the .cproject file in the project root directory. This file is not retrieved from the public NxOS repository and must be recreated locally.}\]
Figure C.11: SConsolidator Configuration: Main

Figure C.12: SConsolidator Configuration: Per-Project Settings
wise the Switzerland flag (white cross with red background) icon on the toolbar will also perform the same function.

C.3 Getting Around NxOS

NxOS is structured as a series of modules. Various packages (`nxt_python`, `pynxt`, `usb_console`, `remote_gdb_reference`), found at the top level of the NxOS project hierarchy, are included for historical reasons and as a reference, though these packages are not relevant to us. The `build_instr` subdirectory contains a LEGO Digital Designer file for constructing the Tri-bot using the 9797 Educational NXT Set. For Mac OS X, MacPorts user port files for `pyfantom` and `nxt-python` are provided inside the `macports` subdirectory. Various scripts in the `scripts` subdirectory will be used during the course of ARM assembly language program development. The actual NxOS code resides in the `nxos` subdirectory.

The organization of the modules in the `nxos` subdirectory is given in Table C.1. The `speedtest` program is an implementation of Steve Hassenplug’s Software Platform Benchmark Comparison Algorithm.

---

2 `nxt-python` is a newer package than `nxt_python` (obsoleted).
3 [http://www.teamhassenplug.org/NXT/NXTSoftware.html](http://www.teamhassenplug.org/NXT/NXTSoftware.html)
<table>
<thead>
<tr>
<th>Module</th>
<th>Submodule</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>armdebug</td>
<td>Debugger</td>
<td>Debugger Module</td>
</tr>
<tr>
<td></td>
<td>Host</td>
<td>PC Debug Support</td>
</tr>
<tr>
<td>base</td>
<td></td>
<td>Core NxOS Routines</td>
</tr>
<tr>
<td></td>
<td>*.h, *.c,</td>
<td>NXT HW Vectors and Init Routines</td>
</tr>
<tr>
<td></td>
<td>*.S</td>
<td></td>
</tr>
<tr>
<td>drivers</td>
<td></td>
<td>Low-level hardware drivers</td>
</tr>
<tr>
<td>lib</td>
<td></td>
<td>Mid-level support libraries</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>doc</td>
<td></td>
<td>API Documentation</td>
</tr>
<tr>
<td>scons_tools</td>
<td></td>
<td>Documentation Generation Scripts</td>
</tr>
<tr>
<td>scripts</td>
<td></td>
<td>Miscellaneous Support Scripts</td>
</tr>
<tr>
<td>systems</td>
<td></td>
<td>User Routines and Programs</td>
</tr>
<tr>
<td></td>
<td>*.ld</td>
<td>Linker Scripts</td>
</tr>
<tr>
<td></td>
<td>armskel</td>
<td>ARM Skeleton Program</td>
</tr>
<tr>
<td></td>
<td>bulldozer</td>
<td>Example using Bulldozer kit</td>
</tr>
<tr>
<td></td>
<td>examples</td>
<td>Low Level NxOS Test Programs</td>
</tr>
<tr>
<td></td>
<td>explore</td>
<td>Explore Bot</td>
</tr>
<tr>
<td></td>
<td>guittest</td>
<td>GUI Menu Test Program</td>
</tr>
<tr>
<td></td>
<td>marvin</td>
<td>Simple Task Scheduler Test Program</td>
</tr>
<tr>
<td></td>
<td>rs485_rx</td>
<td>High Speed Serial Receiver Test</td>
</tr>
<tr>
<td></td>
<td>rs485_tx</td>
<td>High Speed Serial Transmitter Test</td>
</tr>
<tr>
<td></td>
<td>speedtest</td>
<td>Steve Hassenplug’s Speed Test Benchmark</td>
</tr>
<tr>
<td></td>
<td>tag-route</td>
<td>Record and Replay a Tag Route</td>
</tr>
<tr>
<td></td>
<td>tests</td>
<td>Tests and Diagnostics</td>
</tr>
<tr>
<td></td>
<td>tribot</td>
<td>Examples using Tribot</td>
</tr>
<tr>
<td></td>
<td>tvm</td>
<td>Transterpreter kernel module (external)</td>
</tr>
</tbody>
</table>
Appendix D

LEGO MINDSTORMS Resources

“If I have seen further it is by standing on the shoulders of giants.’
attributed to Sir Issac Newton, 1642-1726

The resources listed in Table D.1 are all related to LEGO MINDSTORMS. While it is necessarily incomplete, it provides a starting point for further exploration of related topics as well as resources such as alternative sensors, robot building tips, and various discussion fora where you can connect with other enthusiasts.

Table D.1: Web Resources

<table>
<thead>
<tr>
<th>Site</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEGO Engineering</td>
<td><a href="http://legoengineering.com/">http://legoengineering.com/</a></td>
</tr>
<tr>
<td>Mindboards</td>
<td><a href="http://www.mindboards.net/">http://www.mindboards.net/</a></td>
</tr>
<tr>
<td>The NXT STEP</td>
<td><a href="http://thenxtstep.blogspot.com/">http://thenxtstep.blogspot.com/</a></td>
</tr>
<tr>
<td>Mindsensors</td>
<td><a href="http://www.mindsensors.com/">http://www.mindsensors.com/</a></td>
</tr>
<tr>
<td>HiTechnic</td>
<td><a href="http://www.hitechnic.com/">http://www.hitechnic.com/</a></td>
</tr>
<tr>
<td>Dexter Industries</td>
<td><a href="http://www.dexterindustries.com/">http://www.dexterindustries.com/</a></td>
</tr>
<tr>
<td>NxOS</td>
<td><a href="http://github.com/tcwan/nxos-armdebug">http://github.com/tcwan/nxos-armdebug</a></td>
</tr>
<tr>
<td>Java for LEGO Mindstorms</td>
<td><a href="http://lejos.sourceforge.net/">http://lejos.sourceforge.net/</a></td>
</tr>
<tr>
<td>nxtOSEK</td>
<td><a href="http://lejos-osek.sourceforge.net/">http://lejos-osek.sourceforge.net/</a></td>
</tr>
</tbody>
</table>
Bibliography


303


## Index

### A

<table>
<thead>
<tr>
<th>AAPCS</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Modes</td>
<td></td>
</tr>
<tr>
<td>Direct</td>
<td>43</td>
</tr>
<tr>
<td>Immediate</td>
<td>43</td>
</tr>
<tr>
<td>Indirect</td>
<td>44</td>
</tr>
<tr>
<td>Inherent</td>
<td>43</td>
</tr>
<tr>
<td>Register</td>
<td>43</td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>1D Arrays</td>
<td>150</td>
</tr>
<tr>
<td>Hysteresis</td>
<td>178</td>
</tr>
<tr>
<td>If...Then...Else</td>
<td>134</td>
</tr>
<tr>
<td>Leaky Accumulator</td>
<td>178</td>
</tr>
<tr>
<td>Memory Copying</td>
<td>157</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>157</td>
</tr>
<tr>
<td>Min-Max Search</td>
<td>97</td>
</tr>
<tr>
<td>Movement Control</td>
<td>157</td>
</tr>
<tr>
<td>Multi-byte Structures</td>
<td>151</td>
</tr>
<tr>
<td>Multi-dimensional Arrays</td>
<td>155</td>
</tr>
<tr>
<td>Optimizing If...Then...Else</td>
<td>134</td>
</tr>
<tr>
<td>Pattern Matching</td>
<td>160</td>
</tr>
<tr>
<td>PID Controller</td>
<td>200</td>
</tr>
<tr>
<td>Switch Statements</td>
<td>157</td>
</tr>
<tr>
<td>Application Binary Interface</td>
<td>262</td>
</tr>
</tbody>
</table>

### Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>39</td>
</tr>
<tr>
<td>Instruction Pipelining</td>
<td>26</td>
</tr>
<tr>
<td>Memory Architectures</td>
<td>24</td>
</tr>
<tr>
<td>Register-Memory</td>
<td>40</td>
</tr>
<tr>
<td>Register-Register</td>
<td>41</td>
</tr>
<tr>
<td>Stack-based</td>
<td>42</td>
</tr>
</tbody>
</table>

### ARM

<table>
<thead>
<tr>
<th>ARM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode 1</td>
<td>58</td>
</tr>
<tr>
<td>Addressing Mode 2</td>
<td>58</td>
</tr>
<tr>
<td>Addressing Mode 3</td>
<td>62</td>
</tr>
<tr>
<td>Addressing Mode 4</td>
<td>62</td>
</tr>
<tr>
<td>Addressing Mode 5</td>
<td>63</td>
</tr>
</tbody>
</table>

| Atmel AT91SAM7S | 220 |

### Bit Manipulation Instructions

| 61, 87 |

### Comparison Instructions

| 61, 90 |

### Condition Flags

| 91 |

### Conditional Execution Instructions

| 90 |

### Coprocessor Instructions

| 62, 96 |

### Data Movement Instructions

| 58, 59, 79 |

### Exception Instructions

| 61 |

### Flow Control Instructions

| 61, 92 |

### Instruction Encoding

| 66 |

### Instruction Suffix

| 63 |

### Instruction Summary

| 57 |

### Integer Arithmetic Instructions

| 60, 84 |

### Interworking

| 190 |

### Logic Instructions

| 61, 87 |

### Procedure Call Standard

| 46 |

### Processor Architecture

| 31 |

### Register File

| 44 |

### Thumb State

| 184 |

### Assembly Language

| ARM & Thumb Calls | 190 |

| C & Assembly Interfacing | 128 |

### Calling C from Assembly

| 130 |

### Direct ARM to Thumb Calls

| 193 |

### Direct Thumb to ARM Calls

| 192 |

### First Debugging Example

| 73 |

### First Programming Example

| 71 |

### Indirect ARM to Thumb Calls

| 195 |

### Indirect Thumb to ARM Calls

| 193 |
PUSH & POP Sequence, 119

Thumb
- Bit Manipulation Instructions, 186
- Code Size Reduction, 211
- Comparison Instructions, 189
- Data Movement Instructions, 186
- Direct ARM to Thumb Calls, 193
- Direct Thumb to ARM Calls, 192
- Flow Control Instructions, 189
- Indirect ARM to Thumb Calls, 195
- Indirect Thumb to ARM Calls, 194
- Indirect Thumb to Thumb Interworking Calls, 195
- Instruction Encoding, 190
- Integer Arithmetic Instructions, 186
- Interworking, 190
- Interworking Macros, 195
- Logic Instructions, 186
- Shift & Rotate Instructions, 186
- Thumb State, 184

Tribot
- Input Pre-Processing, 208
- Obstacle Avoidance, 208
- Output Clamping, 209
- PID Controller, 209
About the Author

Tat-Chee Wan received his BSEE and MSEE/CE from University of Miami, Florida, USA, and his PhD in Computer Science from Universiti Sains Malaysia, Penang, Malaysia. He is an Associate Professor with the School of Computer Sciences, Universiti Sains Malaysia.

He was formerly with Motorola Malaysia Sdn. Bhd. as a Senior R&D Engineer in Software Development for two-way radios. His current research interests include QoS mechanisms for Wireless Networks, Satellite-based Internet, and Real Time Embedded Systems.