

## **XGameStation™ Micro Edition User Guide**

Copyright © 2004-2005 Nurve Networks LLC

### **Author**

Alex Varanese

### **Editor/Technical Reviewer**

André LaMothe

### **Version**

1.2

### **ISBN**

Pending

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the user of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

### **Trademarks**

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Nurve Networks LLC cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

### **Warning and Disclaimer**

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “**as is**” basis. The authors and the publisher shall have neither liability nor any responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

### **eBook License**

This eBook may be printed for personal use and (1) copy may be made for archival purposes, but may not be distributed by any means whatsoever, sold, resold, in any form, in whole, or in parts. Additionally, the contents of the CD this eBook came on relating to the design, development, imagery, or any and all related subject matter pertaining to the XGameStation Micro Edition are copyrighted as well and may not be distributed in any way whatsoever in whole or in part. Individual programs are copyrighted by their respective owners and may require separate licensing.

# Licensing, Terms & Conditions

NURVE NETWORKS LLC, INC. END-USER LICENSE AGREEMENT FOR XGAMESTATION MICRO EDITION HARDWARE, SOFTWARE AND EBOOKS

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PRODUCT. IT CONTAINS SOFTWARE, THE USE OF WHICH IS LICENSED BY NURVE NETWORKS LLC, INC., TO ITS CUSTOMERS FOR THEIR USE ONLY AS SET FORTH BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE OR HARDWARE. USING ANY PART OF THE SOFTWARE OR HARDWARE INDICATES THAT YOU ACCEPT THESE TERMS.

**GRANT OF LICENSE:** NURVE NETWORKS LLC (the "Licensor") grants to you this personal, limited, non-exclusive, non-transferable, non-assignable license solely to use in a single copy of the Licensed Works on a single computer for use by a single concurrent user only, and solely provided that you adhere to all of the terms and conditions of this Agreement. The foregoing is an express limited use license and not an assignment, sale, or other transfer of the Licensed Works or any Intellectual Property Rights of Licensor.

**ASSENT:** By opening the files and or packaging containing this software and or hardware, you agree that this Agreement is a legally binding and valid contract, agree to abide by the intellectual property laws and all of the terms and conditions of this Agreement, and further agree to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under your control or in your service.

**OWNERSHIP OF SOFTWARE AND HARDWARE:** The Licensor and/or its affiliates or subsidiaries own certain rights that may exist from time to time in this or any other jurisdiction, whether foreign or domestic, under patent law, copyright law, publicity rights law, moral rights law, trade secret law, trademark law, unfair competition law or other similar protections, regardless of whether or not such rights or protections are registered or perfected (the "Intellectual Property Rights"), in the computer software and hardware, together with any related documentation (including design, systems and user) and other materials for use in connection with such computer software and hardware in this package (collectively, the "Licensed Works"). ALL INTELLECTUAL PROPERTY RIGHTS IN AND TO THE LICENSED WORKS ARE AND SHALL REMAIN IN LICENSOR.

## RESTRICTIONS:

- (a) You are expressly prohibited from copying, modifying, merging, selling, leasing, redistributing, assigning, or transferring in any manner, Licensed Works or any portion thereof.
- (b) You may make a single copy of software materials within the package or otherwise related to Licensed Works only as required for backup purposes.
- (c) You are also expressly prohibited from reverse engineering, decompiling, translating, disassembling, deciphering, decrypting, or otherwise attempting to discover the source code of the Licensed Works as the Licensed Works contain proprietary material of Licensor. You may not otherwise modify, alter, adapt, port, or merge the Licensed Works.
- (d) You may not remove, alter, deface, overprint or otherwise obscure Licensor patent, trademark, service mark or copyright notices.
- (e) You agree that the Licensed Works will not be shipped, transferred or exported into any other country, or used in any manner prohibited by any government agency or any export laws, restrictions or regulations.
- (f) You may not publish or distribute in any form of electronic or printed communication the materials within or otherwise related to Licensed Works, including but not limited to the object code, documentation, help files, examples, and benchmarks.

**TERM:** This Agreement is effective until terminated. You may terminate this Agreement at any time by uninstalling the Licensed Works and destroying all copies of the Licensed Works both HARDWARE and SOFTWARE. Upon any termination, you agree to uninstall the Licensed Works and return or destroy all copies of the Licensed Works, any accompanying documentation, and all other associated materials.

**WARRANTIES AND DISCLAIMER:** EXCEPT AS EXPRESSLY PROVIDED OTHERWISE IN A WRITTEN AGREEMENT BETWEEN LICENSOR AND YOU, THE LICENSED WORKS ARE NOW PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THE WARRANTY OF NON-INFRINGEMENT. WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO WARRANTY THAT (i) THE LICENSED WORKS WILL MEET YOUR REQUIREMENTS, (ii) THE USE OF THE LICENSED WORKS WILL BE UNINTERRUPTED, TIMELY, SECURE, OR ERROR-FREE, (iii) THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE LICENSED WORKS WILL BE ACCURATE OR RELIABLE, (iv) THE QUALITY OF THE LICENSED WORKS WILL MEET YOUR EXPECTATIONS, (v) ANY ERRORS IN THE LICENSED WORKS WILL BE CORRECTED, AND/OR (vi) YOU MAY USE, PRACTICE, EXECUTE, OR ACCESS THE LICENSED WORKS WITHOUT VIOLATING THE INTELLECTUAL PROPERTY RIGHTS OF OTHERS. SOME STATES OR JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IF CALIFORNIA LAW IS NOT HELD TO APPLY TO THIS AGREEMENT FOR ANY REASON, THEN IN JURISDICTIONS WHERE WARRANTIES, GUARANTEES, REPRESENTATIONS, AND/OR CONDITIONS OF ANY TYPE MAY NOT BE DISCLAIMED, ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION AND/OR WARRANTY IS: (1) HEREBY LIMITED TO THE PERIOD OF EITHER (A) FIVE (5) DAYS FROM THE DATE OF OPENING THE PACKAGE CONTAINING THE LICENSED WORKS OR (B) THE SHORTEST PERIOD ALLOWED BY LAW IN THE APPLICABLE JURISDICTION IF A FIVE (5) DAY LIMITATION WOULD BE UNENFORCEABLE; AND (2) LICENSOR'S SOLE LIABILITY FOR ANY BREACH OF ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION, AND/OR CONDITION SHALL BE TO PROVIDE YOU WITH A NEW COPY OF THE LICENSED WORKS. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT LICENSOR HAD BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE LICENSED WORKS. SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

**SEVERABILITY:** In the event any provision of this License Agreement is found to be invalid, illegal or unenforceable, the validity, legality and enforceability of any of the remaining provisions shall not in any way be affected or impaired and a valid, legal and enforceable provision of similar intent and economic impact shall be substituted therefore.

**ENTIRE AGREEMENT:** This License Agreement sets forth the entire understanding and agreement between you and NURVE NETWORKS LLC, supersedes all prior agreements, whether written or oral, with respect to the Software, and may be amended only in a writing signed by both parties.

## **Version & Support/Web Site**

This document is valid with the following hardware, software and firmware versions:

- XGS Micro Edition 1.5 or greater.
- XGS Micro Studio IDE version 1.0.
- XGS ME Programmer Unit Firmware version 1.0.

The information herein will usually apply to newer versions but may not apply to older versions. Please contact Nurve Networks LLC for any questions you may have.

---

Visit **[www.xgamerate.com](http://www.xgamerate.com)** for downloads, support, access to the XGS ME user community, and more!

For technical support, sales, or to ask questions or share feedback, please contact Nurve Networks LLC at:

**NURVE NETWORKS LLC**

402 Camino Arroyo West  
Danville, CA 94506  
USA

**[support@nurve.net](mailto:support@nurve.net)**

# Welcome!

Thank you for purchasing the XGameStation Micro Edition! We have worked hard to provide a unique, high-quality, and educational product that will both engage and entertain. The XGameStation Micro Edition is the world's first do-it-yourself video game system and an empowering tool that will bring you an unprecedented level of knowledge and understanding, whether you're a hobbyist, student, or both.

# Table of Contents

<b>CHAPTER 1: WELCOME!</b> .....	<b>1</b>
<b>1.1 - Package Contents .....</b>	<b>1</b>
1.1.1 - How Your Package May Differ .....	2
<b>1.2 - Important Safety Information!.....</b>	<b>2</b>
1.2.1 - Place the XGS ME in Safe Locations Only .....	2
1.2.2 - Be Careful of XGS ME Cords and Cables .....	2
1.2.3 - Connecting Cords and Cables to the XGS ME Safely .....	2
1.2.4 - Carpet and Other Sources of Electro-Static Discharge (ESD).....	3
1.2.5 - Do Not Expose the XGS ME to Liquids or Moisture .....	3
1.2.6 - When to Unplug the XGS ME .....	3
1.2.7 - Overloading Extension Cords, Wall outlets and Receptacles.....	3
1.2.8 - Only Use XGS ME-compatible Power Supplies.....	3
1.2.9 - Important Heat Sink Information .....	4
1.2.10 - Heat Sinks can be Very Hot .....	4
1.2.11 - If The Processor Heat Sink Detaches .....	5
1.2.12 - Warranty Information.....	5
<b>1.3 - What is the XGameStation Micro Edition?.....</b>	<b>6</b>
1.3.1 - System Overview .....	6
<b>CHAPTER 2: QUICK START GUIDES .....</b>	<b>10</b>
<b>2.1 - Guide 1 - How to Run the Pre-Loaded Demo .....</b>	<b>10</b>
2.1.1.1 - Step 1 – Power up the System .....	10
2.1.1.2 - Step 2 – Connect the System to a Television .....	10
2.1.1.3 - Step 3 – Connect the Controller .....	11
2.1.1.4 - Step 4 – Turn it On! .....	12
2.1.1.5 - Common Problems and Issues .....	13
<b>2.2 - Guide 2 – How to Load an Example Program onto the XGS ME.....</b>	<b>14</b>
2.2.1 - Using the Built-in Programmer and XGS Micro Studio IDE .....	14
2.2.1.1 - Step 1 – Set up the XGameStation Console .....	14
2.2.1.2 - Step 2 – Connect the XGS ME to the Development PC .....	14
2.2.1.3 - Step 3 – Install the XGS Micro Studio IDE and Example Programs on the PC .....	15
2.2.1.4 - Step 4 – Launch the XGS Micro Studio IDE.....	16
2.2.1.5 - Step 5 – Configure the XGS Micro Studio IDE.....	16
2.2.1.6 - Step 6 – Load and Program a Game Demo.....	17
2.2.1.7 - Common Problems and Issues .....	18
2.2.2 - Using the SX-Key .....	18
2.2.2.1 - Step 1 – Set up the XGameStation Console .....	18

2.2.2.2 - Step 2 – Connect the XGS ME to the Development PC .....	18
2.2.2.3 - Step 3 – Install the SX-Key IDE and Example Programs on the PC.....	19
2.2.2.4 - Step 4 – Launch the SX-Key IDE .....	20
2.2.2.5 - Step 5 – Configure the SX-Key IDE .....	20
2.2.2.6 - Step 6 – Load and Program a Game Demo.....	21
2.2.2.7 - Common Problems and Issues .....	22
<b>2.3 - Guide 3 – How to Write a Program for the XGS ME .....</b>	<b>22</b>
2.3.1 - Using the Built-in Programmer and XGS Micro Studio IDE .....	22
2.3.1.1 - Step 1 – Set up the XGameStation Console for Programming .....	23
2.3.1.2 - Step 2 – Create a New Document.....	23
2.3.1.3 - Step 3 – Enter a Demo Program .....	23
2.3.1.4 - Syntax Errors .....	25
2.3.1.5 - Common Problems and Issues .....	25
2.3.2 - Using the SX-Key .....	25
2.3.2.1 - Step 1 – Set up the XGameStation Console for Programming .....	25
2.3.2.2 - Step 2 – Enter a Demo Program .....	25
2.3.2.3 - Syntax Errors .....	27
2.3.2.4 - Common Problems and Issues .....	27
<b>CHAPTER 3: USING XGS MICRO STUDIO .....</b>	<b>28</b>
<b>3.1 - Installing the XGS Micro Studio Software.....</b>	<b>28</b>
<b>3.2 - Elements of the Interface .....</b>	<b>28</b>
3.2.1 - The Toolbar .....	29
3.2.2 - The Document Selector.....	31
3.2.3 - The Document Area .....	32
3.2.4 - The Output Window.....	32
3.2.5 - The Status Bar .....	33
<b>3.3 - Editing, Loading and Running Programs.....</b>	<b>33</b>
3.3.1 - Loading & Editing Source Code .....	33
3.3.1.1 - Line Numbers .....	33
3.3.1.2 - Document Editing Colors .....	34
3.3.2 - Running Programs .....	34
<b>3.4 - Configuring the Tools.....</b>	<b>34</b>
3.4.1.1 - The Assembler Input Tab .....	35
3.4.1.2 - The Assembler Output Tab .....	37
3.4.1.3 - The Hardware Tab.....	39
3.4.1.4 - Programmer Repetition .....	40
3.4.1.5 - Write Check Frequency .....	40
3.4.1.6 - Parallel Port .....	41
3.4.1.7 - Restore Hardware Defaults .....	41

3.4.1.8 - The Editor Colors & Styles Tab .....	41
3.4.1.9 - Font.....	41
3.4.1.10 - Size.....	42
3.4.1.11 - Text Color .....	42
3.4.1.12 - Background Color.....	42
3.4.1.13 - Tab Size.....	42
3.4.1.14 - Tabs as Spaces .....	42
3.4.1.15 - Show Line Numbers .....	42
<b>3.5 - The Real-Time SX Interface .....</b>	<b>42</b>
3.5.1 - The Main Interface Tab .....	43
3.5.1.1 - Programming the XGS Micro With a Hex Program .....	44
3.5.1.2 - Saving the XGS Micro to a Hex Program .....	45
3.5.2 - The Registers Tab.....	45
3.5.2.1 - Reading Registers .....	46
3.5.2.2 - Programming Registers .....	46
<b>3.6 - The Instruction Browser.....</b>	<b>47</b>
3.6.1 - Instruction Lookups .....	47
<b>CHAPTER 4: USING SX-KEY.....</b>	<b>49</b>
<b>4.1 - Installing the SX-Key Software.....</b>	<b>49</b>
<b>4.2 - Elements of the Interface .....</b>	<b>49</b>
4.2.1 - The Toolbar & Menu Bar .....	50
4.2.1.1 - The File Menu.....	51
4.2.1.2 - The Edit Menu .....	51
4.2.1.3 - The Run Menu .....	51
4.2.1.4 - The Help Menu .....	51
4.2.2 - The Document Selector.....	51
4.2.3 - The Document Area .....	51
4.2.4 - The Output Window.....	51
4.2.5 - The Status Bar .....	51
<b>4.3 - Editing, Loading and Running Programs.....</b>	<b>51</b>
4.3.1 - Loading & Editing Source Code .....	51
4.3.2 - Programming, Running and Debugging Programs .....	51
<b>4.4 - Configuring SX-Key .....</b>	<b>51</b>
<b>4.5 - The Device Window .....</b>	<b>51</b>
<b>4.6 - The Clock Window .....</b>	<b>51</b>

<b>4.7 - The Debugger .....</b>	<b>51</b>
4.7.1 - Debugger Overview.....	51
4.7.2 - The Debug Palette .....	51
4.7.3 - The Registers Window .....	51
4.7.3.1 - Real-Time Manipulation of Registers .....	51
4.7.4 - The Watch Window .....	51
4.7.5 - The Code Window.....	51
4.7.5.1 - Setting the Breakpoint .....	51
4.7.5.2 - The Code Window Toolbar.....	51
<b>CHAPTER 5: TROUBLESHOOTING .....</b>	<b>51</b>
<b>5.1 - XGameStation Micro Edition Console .....</b>	<b>51</b>
5.1.1 - Power Supply Plug Doesn't Fit Outlet.....	51
5.1.2 - System Does Not Turn On / No Power .....	51
5.1.3 - No Video Output.....	51
5.1.4 - No Audio Output.....	51
5.1.5 - Video is Black and White.....	51
5.1.6 - Video Signal is Fuzzy/Blurry/Noisy .....	51
<b>5.2 - XGS Micro Studio IDE.....</b>	<b>51</b>
5.2.1 - Cannot Initialize WinIO Library.....	51
5.2.2 - Erratic Behavior in IDE/Crashes.....	51
5.2.3 - Can't Communicate with Device/Corrupted Programming .....	51
5.2.4 - Program Won't Assemble/Assembler Crashes .....	51
5.2.5 - Instruction Browser Is Empty/Garbage .....	51
5.2.6 - "Please save this file before..." Error Message .....	51
<b>5.3 - Parallax SX-Key IDE.....</b>	<b>51</b>
5.3.1 - "SX-Key not found on COMx" Error Message.....	51
5.3.2 - Program Won't Assemble/Assembler Crashes .....	51
5.3.3 - "Current file is READ ONLY and must be saved..." Error Message .....	51
<b>5.4 - Sanity Checks .....</b>	<b>51</b>
5.4.1 - Complete Connections & Settings Reference.....	51
5.4.2 - Colored Dots on the Oscillator Chips .....	51
<b>CHAPTER 6: GUIDE TO INCLUDED DEMO PROGRAMS .....</b>	<b>51</b>
<b>6.1 - NTSC Demos .....</b>	<b>51</b>
6.1.1 - Fire Cube.....	51
6.1.2 - Flags.....	51
6.1.3 - Floormapper .....	51
6.1.4 - Pac Man .....	51

6.1.5 - Plasma.....	51
6.1.6 - Pong .....	51
6.1.7 - Raycaster .....	51
6.1.8 - Rem Colors .....	51
6.1.9 - Rotozoomer.....	51
6.1.10 - Sprites .....	51
6.1.11 - Starfield .....	51
6.1.12 - Tetris.....	51
6.1.13 - Racer .....	51
<b>6.2 - PAL Demos .....</b>	<b>51</b>
6.2.1 - Flags.....	51
6.2.2 - Floormapper .....	51
6.2.3 - Plasma.....	51
6.2.4 - Rotozoomer.....	51
<b>CHAPTER 7: HACKING THE DEMO PROGRAMS .....</b>	<b>51</b>
<b>7.1 - Fun For All Ages .....</b>	<b>51</b>
<b>7.2 - NTSC-Compatible Hacks.....</b>	<b>51</b>
7.2.1 - Recoloring the Raycaster .....	51
7.2.1.1 - Recommended Hacks .....	51
7.2.2 - Hacking the Plasma .....	51
7.2.2.1 - Recommended Hacks .....	51
7.2.3 - Altering the Rotozoomer Bitmap .....	51
7.2.3.1 - A Color Change .....	51
7.2.3.2 - Reassembling .....	51
7.2.3.3 - Troubleshooting .....	51
7.2.3.4 - Recommended Hack .....	51
7.2.4 - Changing the Sprite Demo Bitmaps .....	51
7.2.4.1 - The Sprite Format.....	51
7.2.4.2 - Hacking the Sprites .....	51
7.2.4.3 - Troubleshooting .....	51
7.2.4.4 - Recommended Hack .....	51
7.2.5 - Reshaping the Racer Mountain Range .....	51
7.2.5.1 - Square Wave Mountains .....	51
7.2.5.2 - The Racer City Hack.....	51
7.2.5.3 - Recommended Hacks .....	51
7.2.6 - Reshaping the Tetris Blocks .....	51
7.2.6.1 - Hacking the Block Tile .....	51
7.2.6.2 - Hacking the Tetris Pieces .....	51
7.2.6.3 - Recommended Hacks .....	51
7.2.7 - Hacking Pac-Man .....	51
7.2.7.1 - Redrawing Pac Man as Ms. Pac Man .....	51

7.2.7.2 - Changing the Pac Man and Ghost Colors .....	51
7.2.7.3 - Changing the Level Border Color .....	51
<b>7.3 - PAL-Compatible Hacks .....</b>	<b>51</b>
7.3.1 - Hacking the Plasma Text .....	51
7.3.1.1 - Decoding the String .....	51
7.3.1.2 - Hacking the Text.....	51
7.3.1.3 - Recommended Hack .....	51
7.3.2 - Hacking the Flag Bitmap .....	51
7.3.3 - Altering the RotoZoomer Bitmap.....	51
7.3.3.1 - Changing the Bitmap .....	51
7.3.4 - Hacking the Floormapper Demo .....	51
<b>7.4 - Moving On.....</b>	<b>51</b>
<b>CHAPTER 8: CASE STUDY: THE STARFIELD DEMO .....</b>	<b>51</b>
<b>8.1 - Organization of the Demo .....</b>	<b>51</b>
8.1.1 - Data Structures .....	51
8.1.1.1 - The Star List .....	51
8.1.1.2 - The Rest of the Program .....	51
8.1.2 - Algorithms and Logic.....	51
8.1.2.1 - Initialization .....	51
8.1.2.2 - Anatomy of the Video Kernel.....	51
<b>8.2 - Conclusion .....</b>	<b>51</b>
<b>CHAPTER 9: CASE STUDY: RACING ENGINE DEMO.....</b>	<b>51</b>
<b>9.1 - Data Structures .....</b>	<b>51</b>
<b>9.2 - The Sky Background and Mountain Range.....</b>	<b>51</b>
9.2.1 - The Mountain Height Table.....	51
9.2.2 - Another Approach to Lookup Tables.....	51
9.2.3 - Drawing the Background .....	51
<b>9.3 - Drawing the Race Track .....</b>	<b>51</b>
9.3.1 - A Procedural Race Track .....	51
9.3.2 - Adding Perspective to the Track .....	51
9.3.3 - Making Turns.....	51
9.3.3.1 - Deforming the Racetrack.....	51
9.3.3.2 - Generating the Curve Data.....	51
9.3.4 - Summary .....	51

9.4 - Player Input .....	51
9.5 - The BCD Speedometer .....	51
9.6 - Expanding the Demo .....	51
9.7 - Summary.....	51
<b>CHAPTER 10: THE SX PROGRAMMING API.....</b>	<b>51</b>
<b>10.1 - Programming Architecture .....</b>	<b>51</b>
10.1.1 - The User-Level Interface (Highest Level) .....	51
10.1.2 - The SX Programming API (Middle Level) .....	51
10.1.3 - The Programmer Unit Firmware (Lowest Level) .....	51
10.1.4 - The Full Communication Process .....	51
10.1.4.1 - Writing Data .....	51
10.1.4.2 - Reading Data.....	51
<b>10.2 - Using the Library .....</b>	<b>51</b>
<b>10.3 - Library Organization.....</b>	<b>51</b>
10.3.1 - Loading Programs into the API .....	51
10.3.2 - Writing Programs to the Physical SX52 .....	51
10.3.3 - Reading Programs from the Physical SX52.....	51
<b>10.4 - Library Reference .....</b>	<b>51</b>
10.4.1 - Globals .....	51
10.4.2 - Data Types .....	51
10.4.2.1 - The Programmer Unit Response Packet.....	51
10.4.3 - The SX_DEVICE Structure .....	51
10.4.4 - Constants .....	51
10.4.5 - Functions .....	51
10.4.6 - Macros.....	51
<b>10.5 - Complete Library Demos .....</b>	<b>51</b>
10.5.1 - Programming the SX52 .....	51
10.5.2 - Reading the SX52 .....	51
<b>10.6 - The Programmer Unit Firmware Source Code.....</b>	<b>51</b>
<b>CHAPTER 11: REPROGRAMMING THE PROGRAMMER UNIT FIRMWARE.....</b>	<b>51</b>
<b>11.1 - Reprogramming the SX20 .....</b>	<b>51</b>

<b>11.2 - An Example .....</b>	<b>51</b>
11.2.1 - Reprogramming the Firmware .....	51
11.2.1.1 - Step 1 – Connecting the SX-Key to the SX20 Programming Port .....	51
11.2.1.2 - Step 2 – Load the New SX20 Firmware Program .....	51
11.2.1.3 - Step 3 – Program the SX20.....	51
11.2.2 - Restoring the Programmer Unit Firmware .....	51
11.2.2.1 - Step 1 – Load the Programmer Unit Firmware.....	51
11.2.2.2 - Step 2 – Test the Programmer Unit.....	51
<b>CHAPTER 12: ADVANCED GRAPHICS: TILE GRAPHICS ENGINE .....</b>	<b>51</b>
<b>12.1 - Brief Overview of XGS ME Programming.....</b>	<b>51</b>
<b>12.2 - Different Approaches to Game Graphics .....</b>	<b>51</b>
12.2.1 - Vector Graphics.....	51
12.2.2 - Framebuffer/Pixel Graphics .....	51
12.2.2.1 - Framebuffer Memory Issues.....	51
12.2.2.2 - Framebuffer Speed Issues .....	51
12.2.3 - Tile/Character-Based Graphics .....	51
12.2.4 - Summary .....	51
<b>12.3 - Tile Graphics on the XGS ME .....</b>	<b>51</b>
<b>12.4 - Using the XGS ME Tile Graphics Engine.....</b>	<b>51</b>
12.4.1 - Source Code Structure & Organization .....	51
12.4.2 - Writing Client Programs .....	51
12.4.2.1 - Initializing the Client Program .....	51
12.4.2.2 - Implementing Client Program Game Logic .....	51
12.4.3 - Framebuffers .....	51
12.4.3.1 - Multiple Framebuffers.....	51
12.4.4 - Tiles & Bitmaps .....	51
12.4.5 - Maps & Tile Attribute Tables .....	51
12.4.5.1 - Tile Attribute Tables.....	51
12.4.5.2 - Maps .....	51
12.4.6 - Designing a Game Loop.....	51
12.4.6.1 - Initializing the Game .....	51
12.4.6.2 - Updating the Game .....	51
12.4.6.3 - Drawing Complex Graphics over Multiple Frames .....	51
12.4.6.4 - Vertical Scrolling and Page Flipping.....	51
<b>12.5 - Case Study: The Shooter Demo .....</b>	<b>51</b>
12.5.1 - The Graphics .....	51
12.5.2 - Data Structures .....	51
12.5.3 - Initialization.....	51
12.5.4 - Updating the Game .....	51

12.5.4.1 - Erasing the Game Objects .....	51
12.5.4.2 - Handling Player Input .....	51
12.5.5 - Conclusion.....	51
<b>12.6 - Case Study: XGS Bros.</b> .....	<b>51</b>
12.6.1 - Data Structures .....	51
12.6.2 - Initializing the Game .....	51
12.6.3 - Updating the Game .....	51
12.6.3.1 - Collision Detection .....	51
12.6.3.2 - Gravity-Based Jumping .....	51
12.6.3.3 - The Enemy .....	51
12.6.4 - Conclusion.....	51
<b>12.7 - Case Study: Venture.....</b>	<b>51</b>
12.7.1 - Overview .....	51
12.7.1.1 - The Game Environment .....	51
12.7.1.2 - The Object of the Game .....	51
12.7.2 - Data Structures .....	51
12.7.3 - Initializing the Game .....	51
12.7.4 - The World Map.....	51
12.7.5 - Enemy Movement .....	51
12.7.6 - Color Effects.....	51
12.7.7 - Conclusion.....	51
<b>12.8 - Case Study – The Text Console .</b> .....	<b>51</b>
12.8.1 - The Commands.....	51
12.8.2 - Graphics and Visuals .....	51
12.8.3 - Keyboard Input.....	51
12.8.4 - Text Input and Scrolling .....	51
12.8.5 - Parsing and Executing Commands.....	51
12.8.5.1 - The Do_Cmd Subroutine .....	51
12.8.5.2 - Reading the Command String .....	51
12.8.5.3 - Parsing the Command.....	51
12.8.6 - Conclusion.....	51
<b>12.9 - Conclusion .....</b>	<b>51</b>
<b>CHAPTER 13: THE SX/B BASIC COMPILER .....</b>	<b>51</b>
<b>13.1 - Using the SX/B BASIC Compiler .....</b>	<b>51</b>
<b>13.2 - A Brief Overview of the SX/B Language.</b> .....	<b>51</b>
13.2.1 - Structure of a BASIC Program .....	51
13.2.2 - Declarations .....	51
13.2.2.1 - Variables.....	51

13.2.2.2 - Arrays .....	51
13.2.2.3 - Bits .....	51
13.2.2.4 - Constants.....	51
13.2.3 - Fundamental Language Commands and Nuances .....	51
13.2.3.1 - Arithmetic Operations .....	51
13.2.3.2 - Logic and Program Flow.....	51
13.2.3.3 - Subroutines.....	51
13.2.4 - Inline Assembly .....	51
<b>13.3 - Integrating the Tile Graphics Engine with SX/B .....</b>	<b>51</b>
13.3.1 - SX/B Case Study: Pong .....	51
13.3.1.1 - Integration with the Tile Engine .....	51
13.3.1.2 - The Game Logic .....	51
13.3.1.3 - Initializing the Game .....	51
13.3.1.4 - Game States.....	51
13.3.1.5 - Updating the Game .....	51
13.3.1.6 - Displaying the Score.....	51
13.3.1.7 - Conclusion.....	51
13.3.2 - SX/B Case Study: Breakout .....	51
13.3.2.1 - The Block Array .....	51
13.3.2.2 - Conclusion .....	51
13.3.3 - The Beginnings of an SX/B Game – Frogger .....	51
<b>13.4 - Conclusion .....</b>	<b>51</b>
<b>APPENDIX A: SX52 INSTRUCTION SET REFERENCE.....</b>	<b>267</b>
<b>APPENDIX B: XGS ME SCHEMATIC REFERENCE.....</b>	<b>273</b>

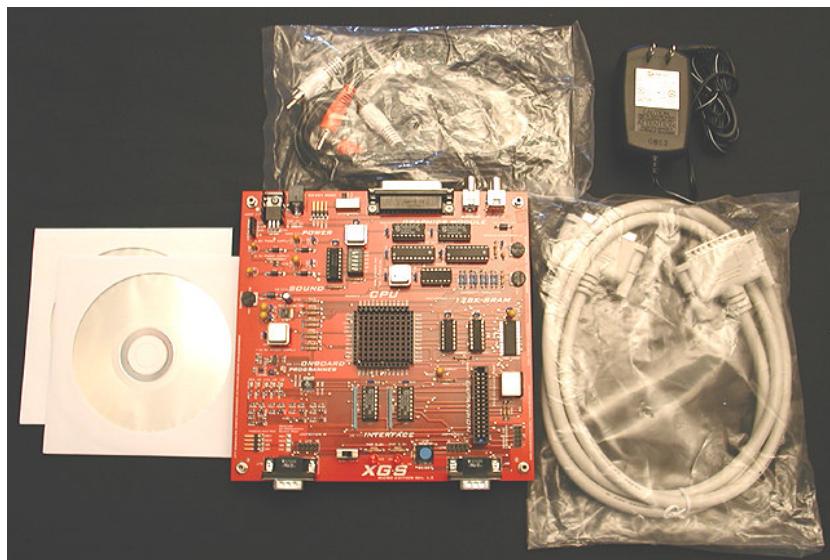
## Chapter 1: Welcome!

This chapter will quickly bring you up to speed with the XGameStation Micro Edition and contains the most important information to review after opening the package. Please carefully read the following two sections, **Package Contents** and **Important Safety Information**.

### 1.1 - Package Contents

The XGameStation Micro Edition is a complete, ready-to-use package that comes with everything you need to get started (as seen in Figure 1.1).

**Figure 1.1 – The XGameStation Micro Edition Package.**



The following items should be included in-box:

- (1) XGameStation Micro Edition console
- (1) Power Supply (United States or International)
- (1) Two-Lead RCA Cable for A/V connection to any standard television
- (1) DB25 Male-to-Male Parallel Cable for connection to any PC for programming
- (1) Atari-Compatible Joystick or Gamepad
- (1) CD-ROM containing eBooks, utilities and tools

- (1) QuickStart Guide illustrating the basic operation of the XGS ME

### 1.1.1 - How Your Package May Differ

The XGameStation Micro Edition package can be customized with a number of options that may result in your package differing slightly from the list above, such as the inclusion of one or more of the following:

- (1) Parallax, Inc. SX-Key programmer
- (1) Extra Atari-Compatible Joystick or Gamepad
- (1) *Programming the SX Microcontroller: A Complete Guide* book by Günther Daubach

## 1.2 - Important Safety Information!

This section contains important safety information and precautions for setting up and working with the XGameStation Micro Edition.

### 1.2.1 - Place the XGS ME in Safe Locations Only

Only place the XGS ME on sturdy, stable surfaces. Do not place the XGS ME close to edges and be especially mindful of cords, wires and cables connected to the console. These can easily snag on nearby objects and cause damage to the system.

### 1.2.2 - Be Careful of XGS ME Cords and Cables

Always be aware of how cords and cables are laid out and where. Avoid taut, tense connections and never stretch a cord or cable to its maximum length. Keep small children and pets away from the XGS ME and its cords and cables at all times. Always discontinue use of damaged cords immediately to prevent damage to connected devices.

### 1.2.3 - Connecting Cords and Cables to the XGS ME Safely

When connecting any cable or controller cord to any of the ports on the XGS ME, plug into the port gently but firmly, and always support the port with your other hand to ensure it is not pushed too hard in one direction. Failing to do so may lead to ports breaking or partially losing their connection to the XGS ME board. Remember, the XGS ME is a hobbyist unit and ports and connectors are not strongly supported.

## 1.2.4 - Carpet and Other Sources of Electro-Static Discharge (ESD)

Electro-Static Discharge (ESD) can potentially damage electrical devices such as the XGS ME, and the exposed design of the XGS ME board makes it more susceptible than standard consumer products. Please keep the following anti-static guidelines in mind at all times to prevent your XGS ME console from ESD damage:

- Do not place the XGS ME directly on carpet. Despite the standoffs in its four corners, it is not meant to be stored or used for extended periods on carpet.
- Do not drag your feet on carpet while handling the XGS ME. This creates static electricity which can discharge into the XGS ME unit.
- If possible, before touching the XGS ME, try touching a piece of metal to discharge any static electricity accumulated on your body or clothing.

## 1.2.5 - Do Not Expose the XGS ME to Liquids or Moisture

Even though the XGS ME is a digital device powered by low voltage, any source of liquid or moisture can be damaging to the electronics and cause internal shorts. Therefore, keep the XGS ME unit away from all sources of water, moisture, or any liquid containers that may spill.

Do not attempt to clean the XGS ME with liquid cleaners, solvents or aerosols.

## 1.2.6 - When to Unplug the XGS ME

Unplug the XGS ME during lightning storms or when not in use for long periods. Do not leave the XGS ME powered-up and unattended for long periods.

## 1.2.7 - Overloading Extension Cords, Wall outlets and Receptacles

The XGS ME consumes a meager 2-5 watts. Nevertheless, do not exceed the stated ratings of cords or receptacles.

## 1.2.8 - Only Use XGS ME-compatible Power Supplies

The XGS ME is designed only for use with the power supplies it comes with (US or International), as well as any power supply meeting the following specifications:

- 9V DC (does not need to be regulated).
- 500 mA (or greater).

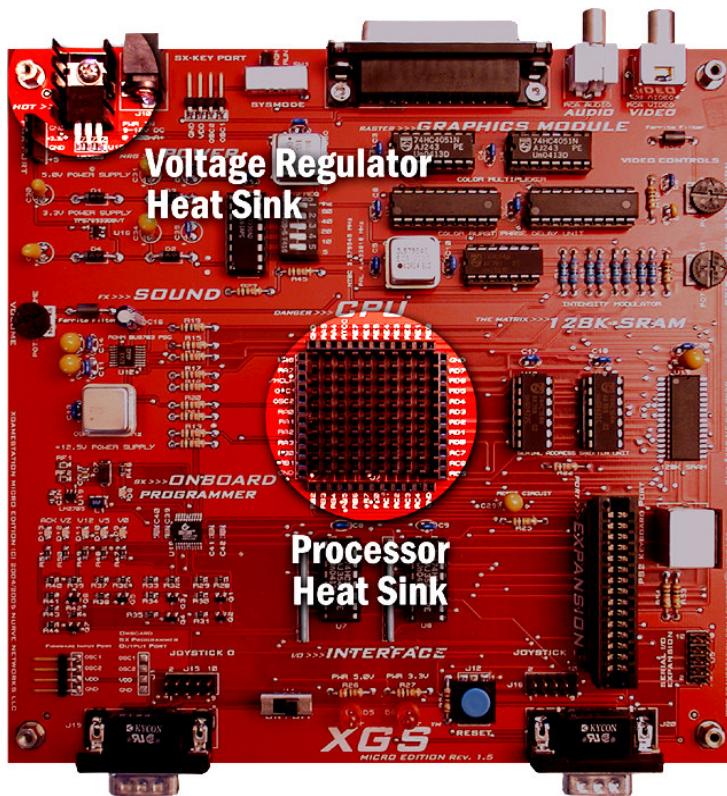
- 2.1mm male power jack with ring ground, tip positive.

Failure to use the proper power supply may cause the machine to overheat, burn out or be otherwise damaged or destroyed.

## 1.2.9 - Important Heat Sink Information

Two heat sinks are attached to the XGS ME; one underneath the 5V regulator, located in the upper-left corner of the board, and the other on top of the SX52 processor located in the center of the board. See Figure 1.2 for a picture. Both of these heat sinks are required for safe operation of the XGS ME console.

Figure 1.2 – The two XGS ME heat sinks.



## 1.2.10 - Heat Sinks can be Very Hot

Heat sinks help cool sensitive components by increasing the surface area for heat to dissipate through into the surrounding air. Naturally, they are hot to the touch, and flammable or combustible items should

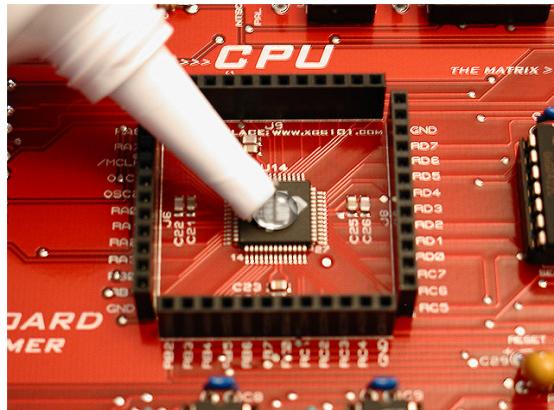
be not be left in contact with them. While the XGS ME's heat sinks should never reach temperature high enough to burn skin, they may be uncomfortably hot after extended use and contact should be avoided.

### 1.2.11 - If The Processor Heat Sink Detaches

If the XGS ME arrives without the main processor's heat sink fully attached, or the heat sink is somehow detached from the processor at any time, turn the console off immediately (if it is not already) and perform the following steps:

1. Apply a dab of RTV 108 silicon adhesive (or another silicon glue with similar properties) to the top of the processor as shown in Figure 1.3.
2. Gently but firmly press the heat sink onto the processor, taking care to keep the heat sink centered on the processor and within the edge connectors surrounding the processor. Hold the heat sink in place for 30-60 seconds.
3. Leave the console in a safe place for 24 hours to allow the glue to dry and cure properly.

**Figure 1.3 – Applying silicon glue to the processor to reattach the heat sink.**



#### WARNING!

The XGS ME will function properly without the heat sink, but the lifespan of the system may be considerably reduced.

### 1.2.12 - Warranty Information

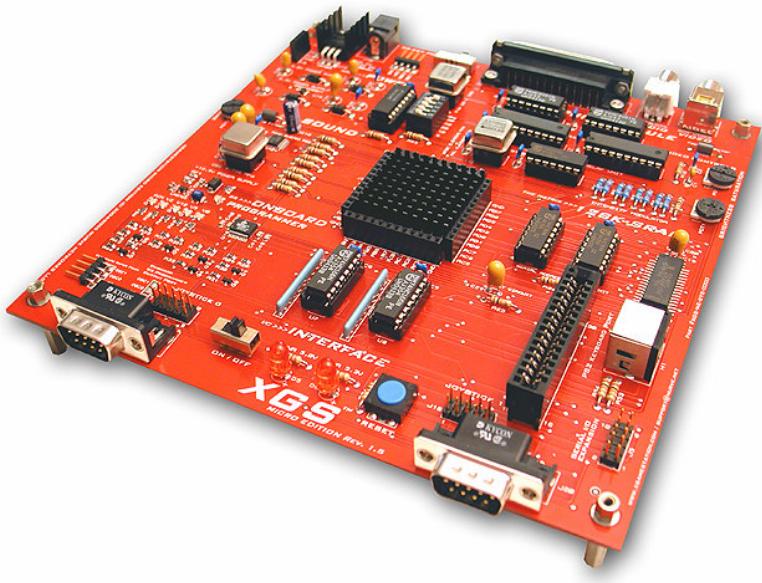
The XGS ME does not have a warranty of any kind due to its hands-on, hobbyist-oriented nature. Unlike consumer electronics, the XGS ME is designed for hands-on experimentation, and it is **always** possible

to damage or destroy the system through experiment-related accidents. XGS ME users are always responsible for their own actions and the actions of anyone allowed to use the device.

## 1.3 - What is the XGameStation Micro Edition?

The XGameStation Micro Edition (XGS ME) is a retro-inspired educational video game console designed specifically for both hardware and software hackers. The system is powered by an 80 MIP RISC processor, has direct raster controlled graphics, 3-channel sound, a built-in programmer, and is capable of outputting both NTSC and PAL composite video. Additionally, to round out the retro-roots of the XGS, it's directly compatible with vintage Atari 2600 joysticks as well as custom-designed game pads.

**Figure 1.4 – The XGameStation Micro Edition.**



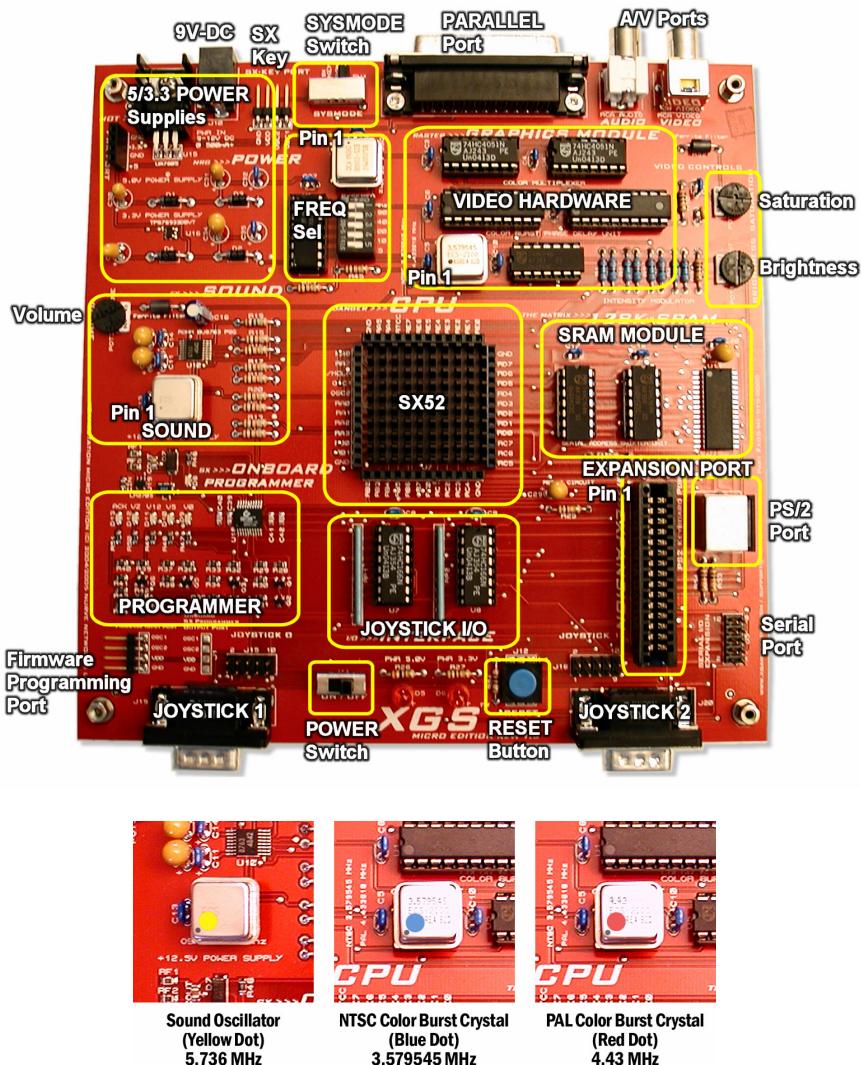
The purpose of the XGameStation Micro Edition (see Figure 1.4) is to teach the principals of digital electrical engineering, computer architecture, assembly-language programming and hardware device interfacing in a new way. Everything is taught consistently from the perspective of video game console development, keeping you engaged and entertained as you learn, and making the real-world application and relevance of each lesson crystal clear.

### 1.3.1 - System Overview

The XGS ME console is an exposed printed circuit board (PCB) with numerous subsystems visible on its surface. Figure 1.5 depicts the XGS ME with each of its primary subsystems highlighted.

Also note in Figure 1.5 that the sound and video oscillator chips **may** come with a colored dot to indicate their speeds. The sound oscillator may have a yellow dot indicating it runs at 5.736 MHz, or “5.736” may actually be printed on it. The NTSC color burst crystal may have a blue dot or “3.579545” on it. The PAL color burst crystal may have a red dot or “4.43” on it. In all cases the colored dot is placed over the alignment dot, which allows you to insert the chip into its socket with the correct orientation. For more information on socketing oscillators with the correct orientation, see **Chapter 5 – Troubleshooting**.

**Figure 1.5 – The XGameStation Micro Edition’s Primary Subsystems, along with other annotations.**



The following is a brief description of each subsystem of the XGameStation. Don’t worry if some of this doesn’t make sense at first; each system is explained in complete detail in the accompanying eBook, *Design Your Own Video Game Console*:

- Power**  
Responsible for regulating the console’s power supply and producing the different voltages needed by various components.

- **Video**  
An array of chips designed to assist programs with the generation of standard television signals to display onscreen graphics.
- **Sound**  
An audio system based around the ROHM sound chip to deliver asynchronous sound output.
- **CPU (SX52)**  
The processing heart of the console, an SX52 microcontroller running at 80 MIPS (millions of instructions per second).
- **SRAM**  
128K of off-chip RAM to store data and other program assets.
- **Onboard Programmer**  
A slave SX20 microcontroller that communicates with the main SX52 to read and write programs to and from its program memory, modify and read the configuration registers, and more. The programmer communicates with a development PC via the parallel port at the rear of the board.
- **I/O Interface**  
A handful of chips and two DB9 connectors that provide an interface for up to two Atari-compatible controllers (joysticks, gamepads, etc.)
- **Expansion Port**  
A 30-pin expansion bus that accepts XGameStation Expansion Cards. Allows the connection of external devices and hardware to interface with the XGS ME console.

## Chapter 2: Quick Start Guides

The following guides explain the main functionality of the XGameStation Micro Edition in a simple, tutorial-style format and are designed to get you up and running as quickly and easily as possible. If you have any trouble with any of these guides, refer to the **Common Problems and Issues** section, as well as **Chapter 5 - Troubleshooting**.

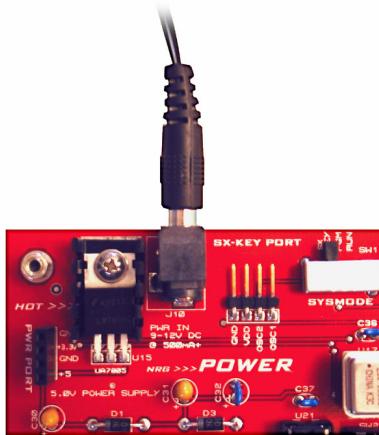
### 2.1 - Guide 1 - How to Run the Pre-Loaded Demo

When your XGS ME arrives, it's already pre-programmed with a demo so you can verify the system works, as well as get an immediate sense of what the XGS ME can do as soon as you boot the machine. This guide will help you set up and connect the hardware for the maiden boot-up.

#### 2.1.1.1 - Step 1 – Power up the System

First, make sure the power switch (near the controller ports on the front of the board) is set to **Off** so we can be sure the system won't get power until we're ready. Next, insert the jack-end of the power supply in the XGameStation's power port (near to the parallel port on the back of the board) and plug the power supply into a wall outlet. See Figure 2.1.

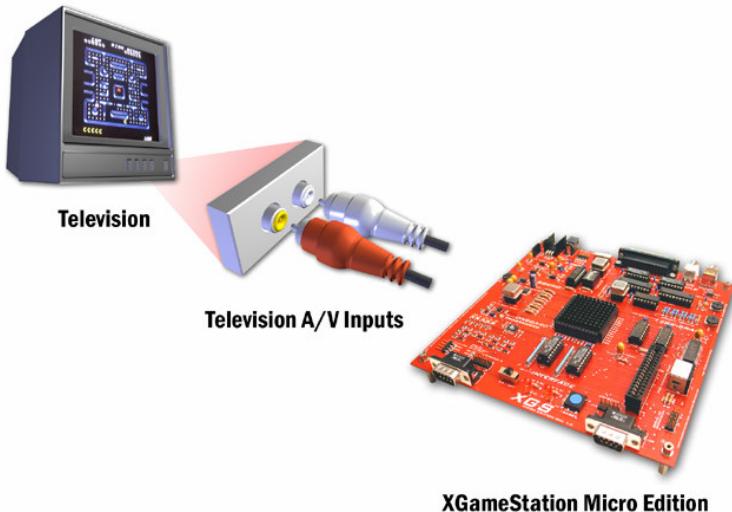
**Figure 2.1 – Powering up the XGS ME.**



#### 2.1.1.2 - Step 2 – Connect the System to a Television

Find the two-leaded RCA cable (depicted in Figure 2.2). The cable's ends may be red and white, yellow and white, or some similar pair of colors. *These colors do not matter*. Both wires are internally identical, so all that matters is that they *correspond* on the XGS and TV ends.

**Figure 2.2 – Connecting the XGS ME to a Television.**



Connect one color wire (it doesn't matter which) to the Audio port on the XGameStation. Connect the other end of the cable (using the same color end) to the Audio input port on your television (choose either the white or yellow port—it doesn't matter which).

Connect the other color wire to the Video port on the XGameStation. Connect the other end of this wire to the Video input port on your television (the red port).

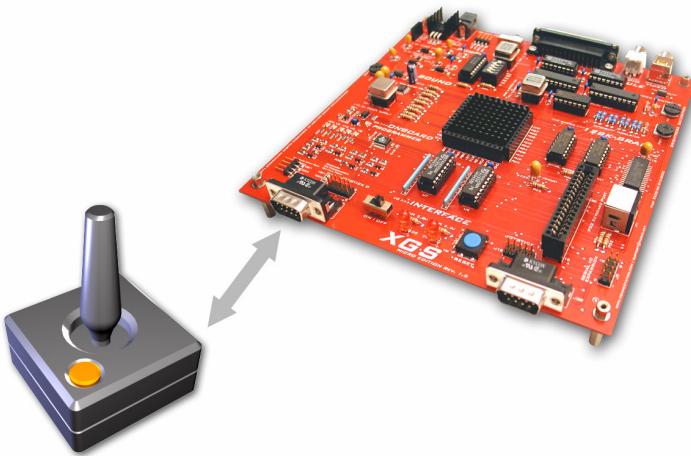
For example, if your wire tips are red and white, connect the red line to the video ports on the XGS ME and TV, and connect the white line to the audio ports.

Turn on your television. If your TV has multiple A/V inputs, make sure to select the one you used.

### 2.1.1.3 - Step 3 – Connect the Controller

Connect your controller to port 1 on the XGS ME console (the port on the left-hand side). See Figure 2.3.

Figure 2.3 – Connecting a Controller to the XGS ME.



Take care to press the controller in firmly but gently, and make sure to hold the port itself with the other hand to stabilize.

#### 2.1.1.4 - Step 4 – Turn it On!

Flip the power switch to the **On** position and press the Reset button (depicted in Figure 2.4). While pressing Reset is not completely necessary in all cases, it is a good practice to get into as it ensures that your XGS ME console is always running off a fresh reset.

Figure 2.4 – The XGS ME Reset Button.



Your XGameStation Micro Edition comes pre-programmed, which means you should see something cool as soon as you start it up!

### 2.1.1.5 - Common Problems and Issues

The following problems and issues are commonly experienced when starting up an XGS ME console for the first time.

- **No Video**

If you aren't seeing anything on your television, first verify that both leads on the A/V cable are connected snugly on both the XGameStation and your TV. Also make sure that the leads are in the right ports, as described above. Lastly, make sure that your television is both turned on and set to the proper A/V input source.

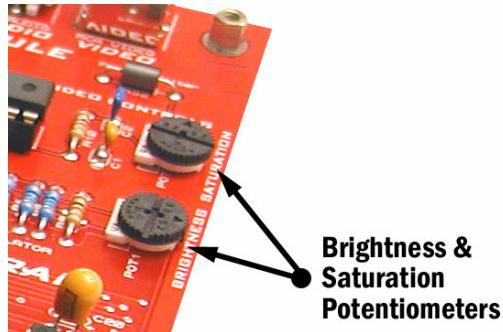
- **No Audio**

If you can't hear anything, the problem could be either the XGS or your TV. First make sure the volume is turned up on your XGameStation by adjusting the potentiometer marked **Volume**. Next, make sure the volume is turned up to an audible level on your TV.

- **Fuzzy/Blurry/Noisy Video Output**

Depending on your TV, the brightness and saturation settings on your XGameStation may produce a distorted or noisy image. Adjust the potentiometers marked **Brightness** and **Saturation** (see Figure 2.5) until you reach the optimal settings for your television. Remember, all TVs are different, and settings that may look great for one person may look awful for another. Especially bad settings may even cause the image to distort and shear, so don't panic if you see this!

**Figure 2.5 – The Brightness and Saturation potentiometers.**



If you're still having trouble, refer to the Troubleshooting chapter later in this user guide.

## 2.2 - Guide 2 – How to Load an Example Program onto the XGS ME

**NOTE**

Before attempting this tutorial, make sure you have set up your XGameStation as outlined in **Guide 1 – How to Run the Pre-Loaded Demo**.

This guide is split up into two parts. One illustrates how a program is loaded onto the XGS ME using the standard, built-in programmer and the XGS Micro Studio IDE. The other covers the same process using the external SX-Key programmer and the SX-Key IDE. If your XGameStation package does not include the SX-Key programmer, you can ignore this tutorial. Otherwise, you may use whichever programmer you like.

### 2.2.1 - Using the Built-in Programmer and XGS Micro Studio IDE

This version of the guide uses the built-in programmer and the XGS Micro Studio IDE. All XGameStation Micro Edition packages can be used with this tutorial.

#### 2.2.1.1 - Step 1 – Set up the XGameStation Console

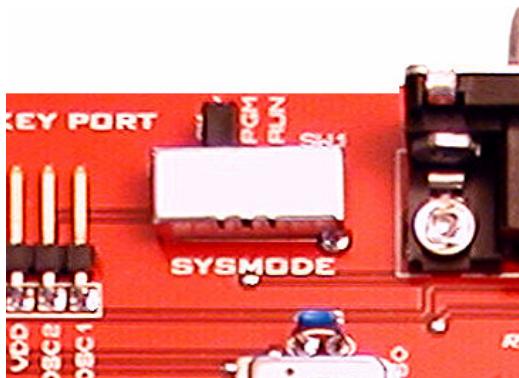
Again, make sure to follow the steps outlined in **Guide 1 – How to Run the Pre-Loaded Demo** before proceeding. This will ensure that your XGS ME console is ready to program and play.

#### 2.2.1.2 - Step 2 – Connect the XGS ME to the Development PC

The XGS ME and the development PC communicate via the parallel port. Take the following steps to properly connect the XGS ME to the PC:

1. Connect one end of the included male-to-male DB25 parallel cable to the XGameStation, and connect the other end to the parallel port on the PC.
2. If your PC has multiple parallel ports, take note of the port you used, as it will be important later on (you will be using LPT1 in most cases).
3. After connecting the parallel port, locate the switch marked **SYSMODE** near the rear of the board next to the parallel port (depicted in Figure 2.6). Set the switch to **PGM** mode.
4. Turn on the XGS ME console if it is not on already.

Figure 2.6 – The SYSMODE Switch.



**NOTE**

Realistically, you may not have your TV and development PC close enough for both to be connected at the same time. If this is the case, you will need to disconnect the RCA A/V cables from the XGameStation (leave them in the TV) and unplug the power supply from the wall outlet so it may be moved close enough to your development PC. Refer to Guide 1 again if necessary when reconnecting the XGS ME to the TV at the end of this guide.

### 2.2.1.3 - Step 3 – Install the XGS Micro Studio IDE and Example Programs on the PC

Take following steps to install the XGS Micro Studio IDE on your PC:

5. Insert the **XGS ME Software CD** in your PC's CD-ROM drive.
6. Double-click **My Computer**.
7. Double-click the icon corresponding to the CD-ROM drive into which you inserted the disc.
8. In the root directory, double-click **XGS\_Micro\_Studio\_Setup.exe**.
9. Follow the on-screen instructions. Use the default settings provided by the installer unless you have specific needs to do otherwise.
10. Drag the **Demos** folder to somewhere on your hard drive. We strongly recommend a directory as close to the root as possible, as it makes things easier on the IDE and minimizes the chances of error.

**TIP**

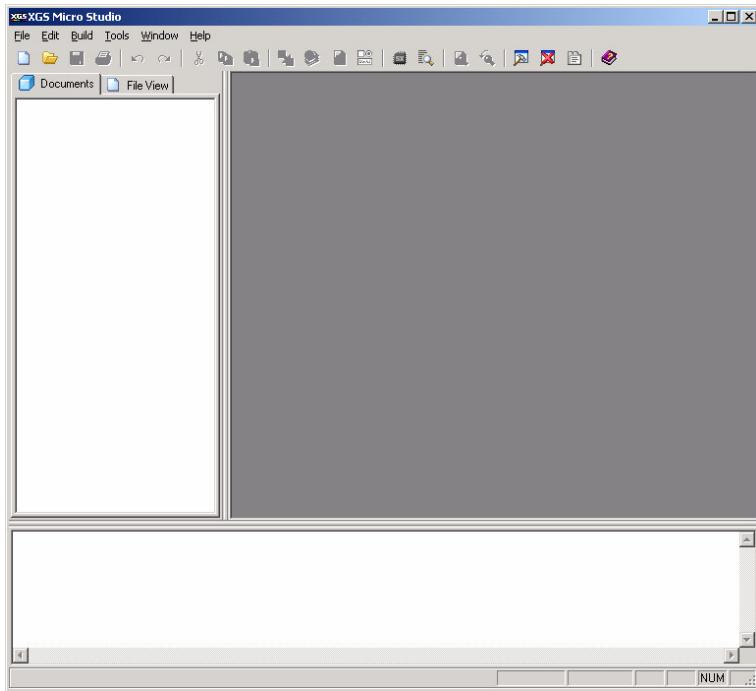
If you prefer, you may avoid the XGS Micro Studio installer and instead drag the **XGS\_Micro\_Studio** folder to your hard drive. The program will then run out of this directory.

#### 2.2.1.4 - Step 4 – Launch the XGS Micro Studio IDE

If the installer did not cause the XGS Micro Studio IDE program to launch, do so yourself:

1. Click the **Start** menu.
2. Click **Programs** (or **All Programs**, depending on your version of Windows) and find **XGS Micro Studio**. Run the program. You should see a screen resembling the screenshot in Figure 2.7.

**Figure 2.7 – The XGS Micro Studio IDE.**

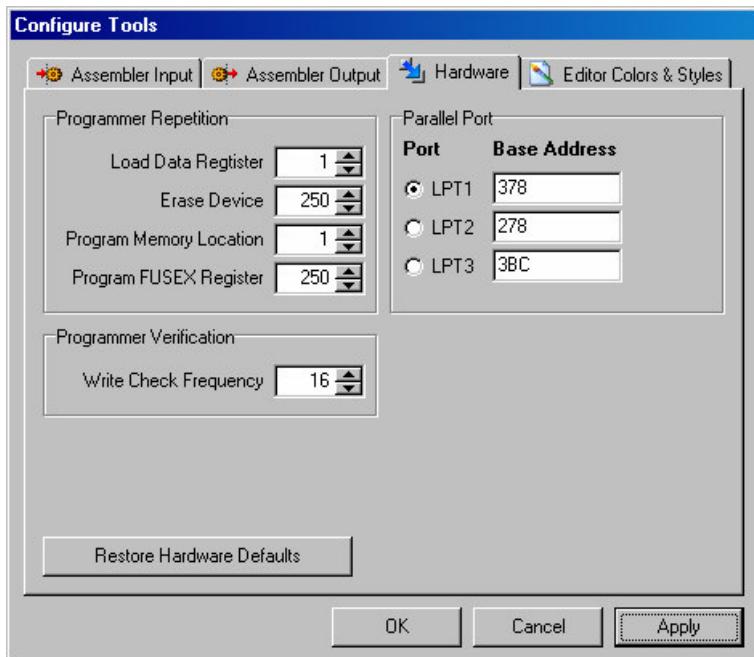


#### 2.2.1.5 – Step 5 – Configure the XGS Micro Studio IDE

Before you can use the IDE to program the XGS ME, you must ensure that it is configured to recognize the XGS ME via the parallel port:

- Under the **Build** menu, click **Tool Settings...** or press **Ctrl+T**. The **Configure Tools** window should appear.
- Click the **Hardware** tab (as depicted in Figure 2.8).
- In the **Parallel Port** box, select the LPT port (LPT1-LPT3) that corresponds to the physical port to which you connected the parallel cable.

**Figure 2.8 – The Hardware Tab of the Configure Tools Window.**



### 2.2.1.6 - Step 6 – Load and Program a Game Demo

Next, you'll need to load the assembly language source code of an XGS ME program. This will be assembled and programmed onto the machine in a moment.

- Under the **File** menu, select **Open...** or click **Ctrl+O**.
- In the **Open** window, find the **Demos** directory you dragged onto your hard drive in the previous step. Navigate to the **NTSC\Racer** subdirectory and open **Racer.src**. The source code to the racing engine demo should appear in a new window. If you are using the PAL version of the XGameStation, open any demo from the **PAL** subdirectory in place of the racer.
- Under the **Build** menu, click **Program and Run** or press **F5**. A window should appear indicating the progress of the program as it is written to the XGS ME console. When the programming is complete, click **OK** on the dialog box that appears.

- Set the **SYSMODE** switch on the XGS ME board to **RUN** mode. Press the reset button to ensure the system is running off a fresh reset.

If everything was done correctly, you should see the racing demo onscreen, or your chosen PAL demo if you are using the PAL version.

**NOTE**

If you disconnected the XGS ME from the television set to connect it to the PC, make sure to re-connect it to the TV at this time.

### 2.2.1.7 - Common Problems and Issues

The following are common problems and issues that arise when programming the XGS ME:

- Errors When Attempting to Program**

If errors occur when attempting to program the XGS ME (after selecting **Program and Run** from the Build menu), first make sure that the parallel cable is snugly connected to both the XGS ME and the PC. Next, ensure the proper parallel port is selected by XGS Micro Studio as specified above. Lastly, make sure the XGS ME console is set to **PGM** mode by the **SYSMODE** switch.

- Game or Demo will not Run after Programming**

In addition to making sure the XGameStation is properly connected to the TV as outlined in Guide 1, make sure the **SYSMODE** switch is set to **RUN** mode *after* the programming process is complete.

If you're still having trouble, refer to **Chapter 5 - Troubleshooting**.

## 2.2.2 - Using the SX-Key

This version of the guide uses the alternative SX-Key programmer and IDE from Parallax, Inc. The SX-Key is not part of the standard XGS ME package and can be ignored if you do not have one.

### 2.2.2.1 - Step 1 – Set up the XGameStation Console

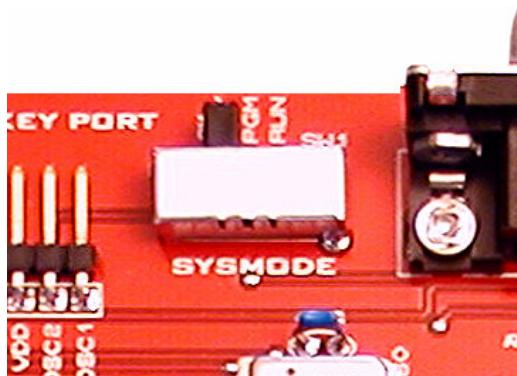
Again, make sure to follow the steps outlined in **Guide 1 – How to Run the Pre-Loaded Demo** before proceeding. This will ensure that your XGS ME console is ready to program and play.

### 2.2.2.2 - Step 2 – Connect the XGS ME to the Development PC

The XGS ME and the development PC communicate via the serial port when using the SX-Key. Take the following steps to properly connect the XGS ME to the PC:

1. Attach the SX-Key to the male end of a DB9 Male-to-Female Straight-Through Serial cable (not included). Attach the female end to the PC.
2. Locate the switch marked **SYSMODE** near the rear of the board next to the parallel port (depicted in Figure 2.9). Set the switch to **KEY** mode.
3. Turn on the XGS ME console if it is not on already.

Figure 2.9 – The SYSMODE Switch.



**NOTE**

Realistically, you may not have your TV and development PC close enough for both to be connected at the same time. If this is the case, you will need to disconnect the RCA A/V cables from the XGameStation (leave them in the TV) and unplug the power supply from the wall outlet so it may be moved close enough to your development PC. Refer to Guide 1 again if necessary when reconnecting the XGS ME to the TV at the end of this guide.

### 2.2.2.3 - Step 3 – Install the SX-Key IDE and Example Programs on the PC

Take following steps to install the XGS Micro Studio IDE on your PC:

1. Insert the **XGS ME Software CD** in your PC's CD-ROM drive.
2. Double-click **My Computer**.
3. Double-click the icon corresponding to the CD-ROM drive into which you inserted the disc.
4. Open the **SX\_Key\_IDE** directory and double-click **SX\_Key\_IDE\_Setup.exe**.
5. Follow the on-screen instructions. Use the default settings provided by the installer unless you have specific needs to do otherwise.

6. Drag the **Demos** folder to somewhere on your hard drive. We strongly recommend a directory as close to the root as possible, as it makes things easier on the IDE and minimizes the chances of error.

#### 2.2.2.4 - Step 4 – Launch the SX-Key IDE

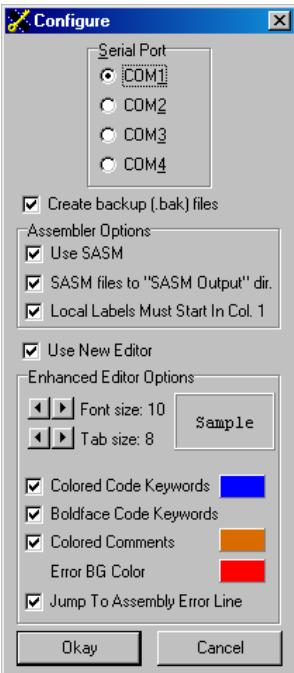
If the installer did not cause the XGS Micro Studio IDE program to launch, do so yourself:

1. Click the **Start** menu.
2. Click **Programs** (or **All Programs**, depending on your version of Windows) and find **Parallax, Inc.**. Run the program.

#### 2.2.2.5 - Step 5 – Configure the SX-Key IDE

Before you can use the IDE to program the XGS ME, you must ensure that it is configured to recognize the XGS ME via the serial port:

1. Under the **Run** menu, click **Configure...** or press **Ctrl+U**. The **Configure** window should appear (see Figure 2.10).
2. In the **Serial Port** box, select the COM port (COM1-COM4) that corresponds to the physical port to which you connected the SX-Key cable.

**Figure 2.10 – The SX-Key Configure Window**

### 2.2.2.6 - Step 6 – Load and Program a Game Demo

Next, you'll need to load the assembly language source code of an XGS ME program. This will be assembled and programmed onto the machine in a moment.

1. Under the **File** menu, select **Open...** or click **Ctrl+O**.
2. In the **Open** window, find the **Demos** directory you dragged onto your hard drive in the previous step. Navigate to the **NTSC\Racer** subdirectory and open **Racer.src**. The source code to the racing engine demo should appear in a new window. If you are using the PAL version of the XGameStation, open any demo from the **PAL** subdirectory in place of the racer.
3. Under the **Run** menu, select **Run** or press **Ctrl+R**. A window should appear indicating the progress of the program as it is written to the XGS ME console. When the programming is complete, click **OK** on the dialog box that appears.
4. Set the **SYSMODE** switch on the XGS ME board to **RUN** mode. Press the reset button to ensure the system is running off a fresh reset.

If everything was done correctly, you should see the racing demo onscreen, or your chosen PAL demo if you are using the PAL version.

**NOTE**

If you disconnected the XGS ME from the television set to connect it to the PC, make sure to re-connect it to the TV at this time.

### 2.2.2.7 - Common Problems and Issues

The following are common problems and issues that arise when programming the XGS ME:

- **Errors When Attempting to Program**

If errors occur when attempting to program the XGS ME (after selecting **Program** from the **Run** menu), first make sure that the SX-Key cable is snugly connected to both the XGS ME and the PC, and that the SX-Key itself is connected snugly to cable. Next, ensure the proper serial port is selected by XGS Micro Studio as specified above. Lastly, make sure the XGS ME console is set to **KEY** mode by the **SYSMODE** switch.

- **Game or Demo will not Run after Programming**

In addition to making sure the XGameStation is properly connected to the TV as outlined in **Guide 1 – How to Play a Game or Run a Demo**, make sure the **SYSMODE** switch is set to **RUN** mode *after* the programming process is complete.

If you're still having trouble, refer to the Troubleshooting chapter later in this user guide.

## 2.3 - Guide 3 – How to Write a Program for the XGS ME

This guide is split up into two parts. One illustrates how a program is written and programmed onto the XGS ME using the standard, built-in programmer and the XGS Micro Studio IDE. The other covers the same process using the external SX-Key programmer and the SX-Key IDE. If your XGameStation package does not include the SX-Key programmer, you can ignore this second tutorial. Otherwise, you may use whichever programmer you like.

**NOTE**

Before attempting this tutorial, make sure you have set up your XGameStation as outlined in **Guide 1 – How to Play a Game or Run a Demo**. Then make sure you have followed at least one version of **Guide 2 - How to Load an Example Program onto the XGS ME** and have successfully programmed and run the game demo.

### 2.3.1 - Using the Built-in Programmer and XGS Micro Studio IDE

This version of the guide uses the built-in programmer and the XGS Micro Studio IDE. All XGameStation Micro Edition packages can be used with this tutorial.

### 2.3.1.1 - Step 1 – Set up the XGameStation Console for Programming

Again, make sure to read both Guides 1 and 2 before attempting to complete this guide. Once you have connected the XGS ME to the PC, and installed and configured the IDE, you are ready to continue.

### 2.3.1.2 - Step 2 – Create a New Document

Launch XGS Micro Studio IDE and select **New** from the **File** menu or press **Ctrl+N**. A blank document window should appear.

### 2.3.1.3 - Step 3 – Enter a Demo Program

Enter the following code into the document window:

```
; ***** Directives
DEVICE      SX52, OSCHS3, IFBD, XTLBUFD
RESET      main
FREQ       80_000_000

; ***** Variables

ORG        $30
luma       DS      1

; ***** Main program

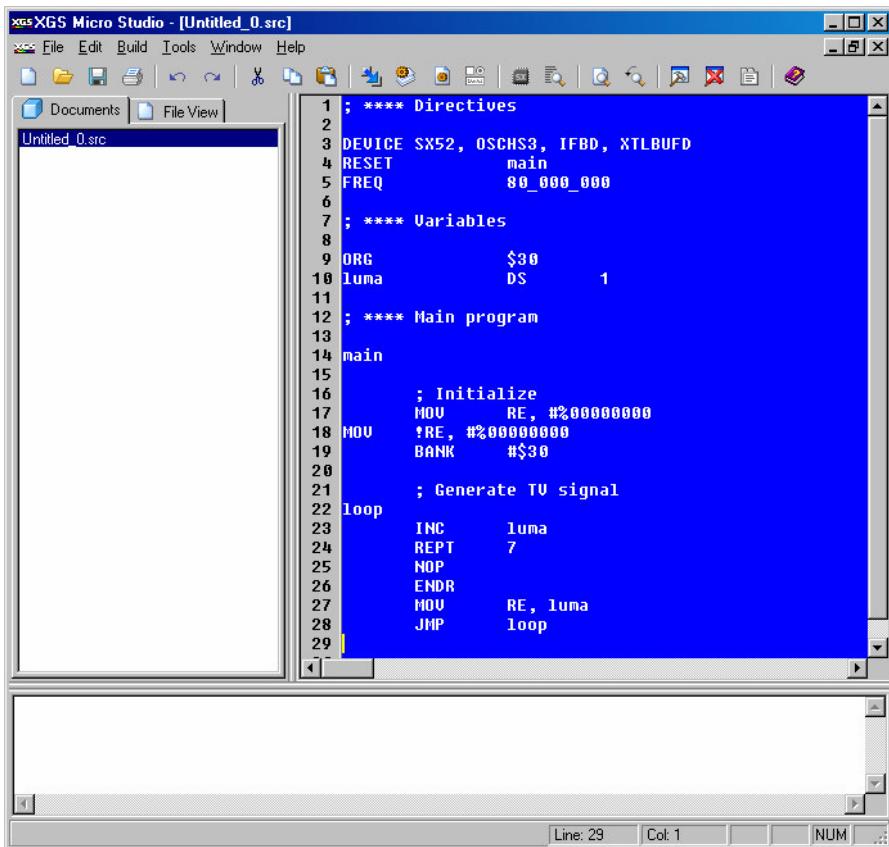
main

; Initialize
MOV        RE, #00000000
MOV        !RE, #00000000
BANK      #$30

; Generate TV signal
loop
INC        luma
REPT      7
NOP
ENDR
MOV        RE, luma
JMP        loop
```

Don't worry if it doesn't make sense yet. All that matters for now is that you can enter the source code as you would any other editor (as seen in Figure 2.11).

Figure 2.11 – Entering the example code in XGS Micro Studio.



The screenshot shows the XGS Micro Studio interface with a single open window titled "Untitled\_0.src". The window contains the following assembly code:

```

1 ; **** Directives
2
3 DEVICE SX52, OSCHS3, IFBD, XTLBUFD
4 RESET      main
5 FREQ       80_000_000
6
7 ; **** Variables
8
9 ORG        $30
10 luma      DS      1
11
12 ; **** Main program
13
14 main
15
16     ; Initialize
17     MOV    RE, #20000000
18 MOV    !RE, #20000000
19 BANK   #$30
20
21     ; Generate TV signal
22 loop
23     INC    luma
24     REPT   7
25     NOP
26     ENDR
27     MOV    RE, luma
28     JHP    loop
29

```

Once the code is entered, select **Program and Run (F5)** from the **Build** menu as you did in Guide 2. If the code was entered correctly, the progress window should appear as it did before and the assembled program should be written to the XGS ME. Once finished, press **OK** in the dialog that appears.

### NOTE

If you disconnected the XGS ME from the television set in order to connect it to the PC, make sure to re-connect it to the TV at this time.

Your results will vary, but essentially this program generates a very unstable television signal that may look like anything from crisp color bars to erratic patterns of color and noise.

### 2.3.1.4 - Syntax Errors

You may find that XGS Micro Studio cannot assemble the code you entered. If this is the case, the code must have been entered incorrectly. Try to find the error and reenter the problematic code.

### 2.3.1.5 - Common Problems and Issues

The following are common problems and issues that arise when writing programs for the XGS ME:

- **Syntax Errors**

Syntax errors occur when code is entered that is not a valid part of SX52 assembly language. These usually are the result of typos, and as said above, can be fixed by checking and reentering the invalid code.

- **No Video**

If you aren't getting any output, it could either be due to a bad connection between the XGS ME and the television (outlined in detail in Guide 1), or an error with the code that is a valid part of SX52 assembly language, but simply does not produce a video signal as it should. Once again, check the code listing above if anything does not seem to be working. Lastly, remember also that all TVs are different and this particular program is not stable from one TV to the next by nature.

If you're still having trouble, refer to the Troubleshooting chapter later in this user guide.

## 2.3.2 - Using the SX-Key

This version of the guide uses the SX-Key programmer and the SX-Key IDE.

### 2.3.2.1 - Step 1 – Set up the XGameStation Console for Programming

Again, make sure to read both Guides 1 and 2 before attempting to complete this guide. Once you have connected the XGS ME to the PC, and installed and configured the IDE, you are ready to continue.

### 2.3.2.2 - Step 2 – Enter a Demo Program

Launch the SX-Key IDE. A new document window will already be open for you. Enter the following code into the document window:

```
; **** Directives
DEVICE  SX52, OSCHS3, IFBD, XTLBUFD
RESET      main
FREQ       80_000_000

; **** Variables
```

```

ORG      $30
luma    DS     1

; ***** Main program

main

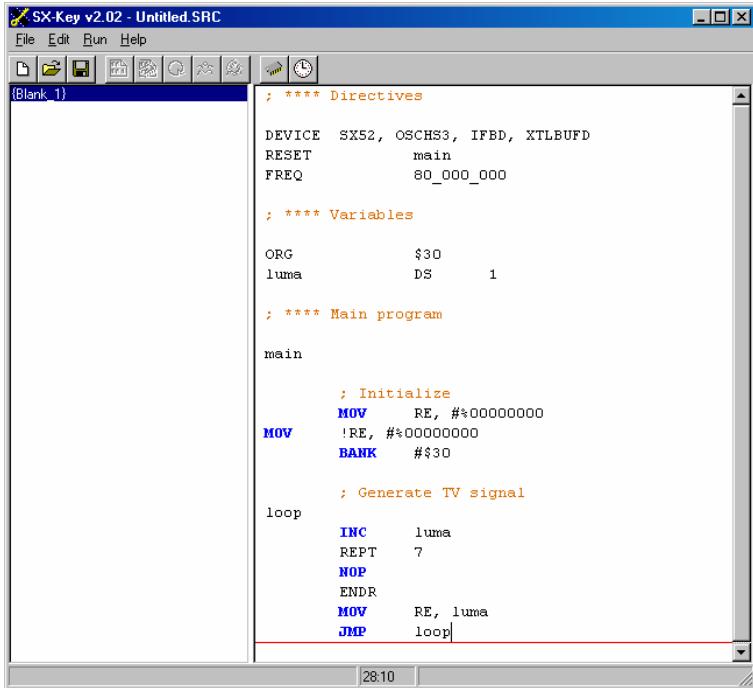
        ; Initialize
MOV      RE, #00000000
!RE, #00000000
BANK   #$30

        ; Generate TV signal
loop
INC      luma
REPT    7
NOP
ENDR
MOV      RE, luma
JMP      loop

```

Don't worry if it doesn't make sense yet. All that matters for now is that you can enter the source code as you would any other editor (as seen in Figure 2.12).

**Figure 2.12 – Entering the example code in SX-Key.**



Once the code is entered, select **Run (Ctrl+R)** from the **Run** menu as you did in Guide 2. If the code was entered correctly, the progress window should appear as it did before and the assembled program should be written to the XGS ME. Once finished, press **OK** in the dialog that appears.

**NOTE**

If you disconnected the XGS ME from the television set to connect it to the PC, make sure to re-connect it to the TV at this time.

Your results will vary, but essentially this program generates a very unstable television signal that may look like anything from crisp color bars to erratic patterns of color and noise.

### **2.3.2.3 - Syntax Errors**

You may find that XGS Micro Studio cannot assemble the code you entered. If this is the case, the code must have been entered incorrectly. Try to find the error and reenter the problematic code.

### **2.3.2.4 - Common Problems and Issues**

The following are common problems and issues that arise when writing programs for the XGS ME:

- **Syntax Errors**

Syntax errors occur when code is entered that is not a valid part of SX52 assembly language. These usually are the result of typos, and as said above, can be fixed by checking and reentering the invalid code.

- **No Video**

If you aren't getting any output, it could either be due to a bad connection between the XGS ME and the television (outlined in detail in Guide 1), or an error with the code that is a valid part of SX52 assembly language, but simply does not produce a video signal as it should. Once again, check the code listing above if anything does not seem to be working. Lastly, remember also that all TVs are different and this particular program is not stable from one TV to the next by nature.

If you're still having trouble, refer to the Troubleshooting chapter later in this user guide.

# Chapter 3: Using XGS Micro Studio

XGS Micro Studio is an IDE developed exclusively for assembly language development on the XGameStation Micro Edition. It supports standard tools for editing and assembling source code, as well as programming the results onto the XGS ME console via the parallel port. This chapter is a complete overview of how XGS Micro Studio is used.

## 3.1 - Installing the XGS Micro Studio Software

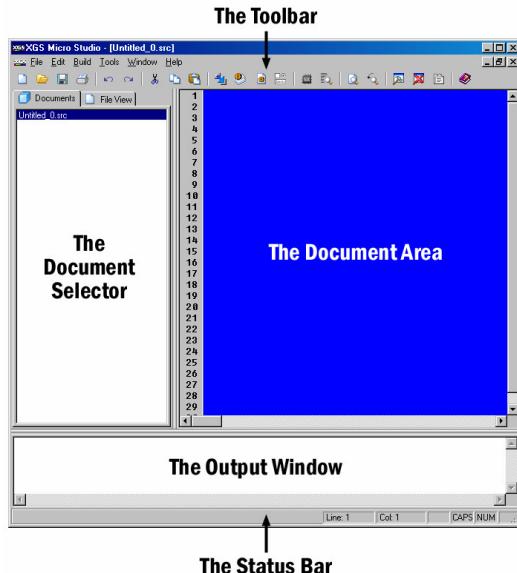
If you haven't already done so, you'll of course need to install XGS Micro Studio before you can use it.

See ***Guide 2 – How to Load an Example Program onto the XGS ME*** in Chapter 2 for step-by-step instructions for installing and configuring XGS Micro Studio.

## 3.2 - Elements of the Interface

XGS Micro Studio is designed to be familiar to users of existing IDEs. Figure 3.1 depicts the layout of the IDE as it usually appears.

Figure 3.1 – The XGS Micro Studio Interface.



The program is split up into five main areas:

- **The Toolbar**
- **The Document Selector**
- **The Document Area**
- **The Output Window**
- **The Status Bar**

The following sections explain these areas in full detail.

### 3.2.1 - The Toolbar

The toolbar (seen in Figure 3.2) encapsulates most of XGS Micro Studio's main functionality in easy-to-use buttons.

**Figure 3.2 – The XGS Micro Studio Toolbar.**



Use the toolbar to create, open and save documents, invoke built-in tools like the Real-Time SX Interface and the Instruction Browser, and assemble, write and view listings of programs.

The following is a brief explanation of each toolbar button:

-  **New (Ctrl+N)**  
Creates a new document.
-  **Open (Ctrl+O)**  
Opens a document for editing.
-  **Save (Ctrl+S)**  
Saves the currently selected document.
-  **Print (Ctrl+P)**  
Displays the print dialog box for the currently selected document.



- **Undo (Ctrl+Z)**

Undos the last action in the selected document. Undo is good for multiple levels, allowing entire sequences of modifications to be undone.



- **Redo (Ctrl+Y)**

Redos the last action undone in the selected document. Redo is good for multiple levels, allowing entire sequences of modifications to be redone.



- **Cut (Ctrl+X)**

Cuts the current selection from the selected document to the clipboard.



- **Copy (Ctrl+C)**

Copies the current selection in the selected document to the clipboard.



- **Paste (Ctrl+V)**

Pastes text content in the clipboard to the currently selected document.



- **Program and Run (F5)**

Assembles the currently selected document and program the XGS Micro with it. The results of the assembly process appear in the output window.



- **Assemble (F7)**

Assembles the currently selected document. The results are printed in the output window.



- **Generate Listing (Ctrl+L)**

Generates an annotated program listing of the currently selected document in the document's directory and open it.



- **Configure Tools (Ctrl+T)**

Displays the tool configuration window.



- **Real-Time SX Interface (Ctrl+I)**

Displays the Real-Time SX Interface tool.



- **Instruction Browser (Ctrl+B)**

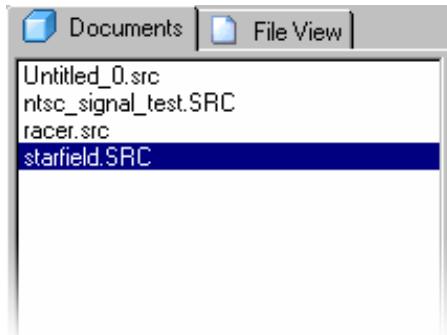
Displays the Instruction Browser tool.

-  **Find (Ctrl+F)**  
Displays the Find dialog, allowing the currently selected document to be searched.
-  **Replace (Ctrl+H)**  
Displays the Replace dialog, allowing the currently selected document to be searched for text strings to replace with a replacement string.
-  **Output Window**  
Toggles the output window.
-  **Clear Document Selector**  
Clears the document selector's contents.
-  **Document Selector**  
Toggles the document selector.
-  **Help (F1)**  
Displays the XGS Micro Studio help system.

### 3.2.2 - The Document Selector

The **document selector**, as seen in Figure 3.3, lists the currently open documents.

**Figure 3.3 – The XGS Micro Studio Document Selector.**



Note the **File View** tab (Figure 3.4), which provides a built-in view to the underlying file system. Double click any file in the File View tab to instantly open it in XGS Micro Studio.

**Figure 3.4 – The Document Selector's File View Tab.**

Click a file in the document selector to bring it to the foreground. As documents are opened and closed, the document selector automatically updates. When XGS Micro Studio closes, the document selector remembers the last files opened. When XGS Micro Studio is re-opened, the last files in the selector are automatically re-opened so you can resume your work.

### 3.2.3 - The Document Area

The **document area** is where all open documents are displayed. Documents can be maximized to fill the entire area, minimized to the corner, or left at their normal size for viewing multiple documents. The size of the document area can be increased or decreased as desired by resizing or toggling the **Document Selector** and **Output Window** panes.

### 3.2.4 - The Output Window

The **output window** contains the output of assembler operations, as seen in Figure 3.5.

**Figure 3.5 – The XGS Micro Studio Output Window.**

```
SASM Cross-Assembler for Scenix SX-based microcontrollers      Version 1.50.97
Copyright (C) 2000 Scenix Inc. All Rights Reserved.
Version as integer: 15097
CWD = D:\Business\XGameStation\Work\Weeknd_10_11_2004
D:\Business\XGameStation\Work\Weeknd_10_11_2004\ntsc_signal_test.SRC(3) Line 3, Warning 54, Pass 1: Overriding earlier target device declaration
D:\Business\XGameStation\Work\Weeknd_10_11_2004\ntsc_signal_test.SRC(1) Line 1, Warning 65, Pass 1: No IRC_CAL directive. Default IRC_SLOW being used

Expression Evaluation Stack Stats - Min: 0, Max: 1
Expression Evaluation Stack Stats - Min: 0, Max: 1

28 lines compiled in 0.00 seconds
8 symbols
2 warnings
0 severe errors
```

The output window will accumulate output from successive assembly operations. To clear the output window, select **Clear Output Window** under the **Window** menu.

### 3.2.5 - The Status Bar

The **status bar** (seen in Figure 3.6) runs along the bottom of the screen at all times and tracks basic information like the status of special keys and the location of the cursor within the current document, among other things.

Figure 3.6 – The XGS Micro Studio Status Bar.



## 3.3 - Editing, Loading and Running Programs

The two primary purposes of the IDE are editing source code and programming the assembled programs onto the XGS ME console.

### IMPORTANT!

When using XGS Micro Studio to program the XGS ME hardware, make sure the **SYSMODE** switch (located at the back of the board next to the parallel port) is set to **PGM** mode. Once the programming is complete, set the switch to **RUN** mode. When you're ready to program the system again, switch back to **PGM** mode. In short, a program cannot be written to the processor without **PGM** mode set, and the program cannot run without **RUN** mode set.

### 3.3.1 - Loading & Editing Source Code

Under the **File** menu, Select **Open...** (**Ctrl+O**) to open an existing source file, or **New** (**Ctrl+N**) to create a new source file. Once open, any source file can be edited just like any other standard windows program. Multiple levels of undo and redo are supported, as are the clipboard, find and replace, and so on.

#### 3.3.1.1 - Line Numbers

Unless disabled, all document windows display line numbers next to each line of source code (as seen in Figure 3.7). To turn this option on or off, see the next section, **Configuring the Tools**.

**Figure 3.7 – Document Line Numbers.**

```

1 ; **** Directives
2
3 DEVICE SX52, OSCHS3, IFBD, XTLBUFD
4 RESET main
5 FREQ 80_000_000
6
7 ; **** Variables
8
9 ORG $30
10 luma DS 1
11
12 ; **** Main program
13
14 main
15
16 ; Initialize
17 MOV RE, #%00000000
18 MOV !RE, #%00000000
19 BANK #$30

```

### 3.3.1.2 - Document Editing Colors

Currently, XGS Micro Studio allows the user to select foreground and background colors for the document windows, as well as the text font and size. To manipulate these options, see the next section, **Configuring the Tools**.

### 3.3.2 - Running Programs

XGS Micro Studio makes it easy to run your program once it's been written. Simply click **Program and Run (F5)** (making sure the **SYSMODE** switch is set to **PGM** mode). This will program the assembled code onto the hardware. Once the programming is complete, be sure to switch **SYSMODE** into **RUN** mode, as the program will not be able to execute otherwise. Don't forget to switch back to **PGM** when you're ready to program it again.

## 3.4 - Configuring the Tools

Under the **Build** menu, select **Tool Settings... (Ctrl+T)** to display the **Configure Tools** window. This window allows the underlying tools (the editor, assembler, and hardware driver) to be configured and customized.

This window is broken up into the following four tabs:

- Assembler Input

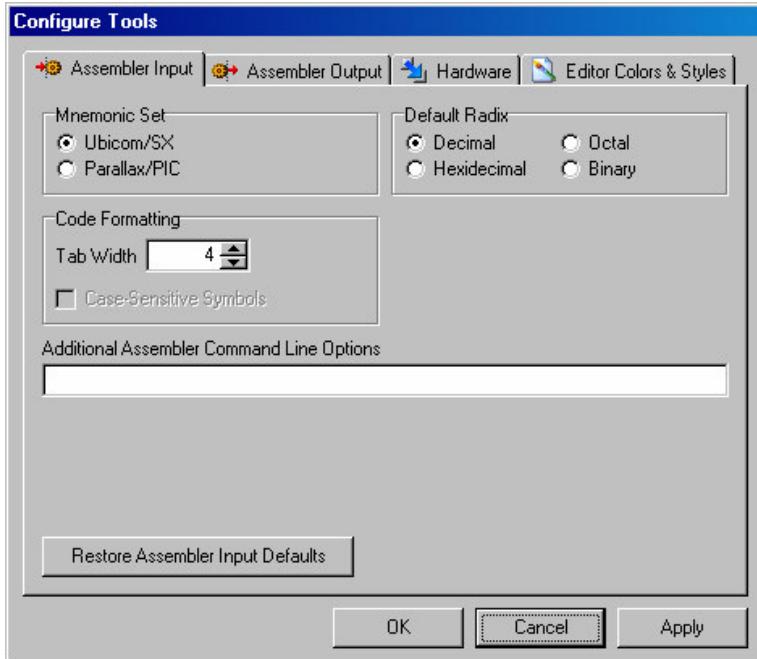
- Assembler Output
- Hardware
- Editor Colors & Styles

The following sections fully detail each of these tabs.

### 3.4.1.1 - The Assembler Input Tab

This tab, shown in Figure 3.8, configures of the data sent into the assembler from XGS Micro Studio.

**Figure 3.8 – The Assembler Input Tab.**



#### 3.4.1.1.1 - Mnemonic Set

The assembler accepts two different sets of mnemonics (which describe what the various assembly language instructions "look like"). The default is the standard Ubicom/SX mnemonic set, used in most SX documentation and literature, and supported by the XGS Micro Studio syntax highlighter. The less common Parallax/PIC dialect is also available (used primarily on PIC microcontrollers), making it easy to deal with source code written for other assemblers when necessary.

### 3.4.1.1.2 - Default Radix

Determines the base in which numeric values are expected when a prefix is not included. For example, the hex digit C9 must normally be written \$C9 (the \$ prefix denotes hexadecimal). If the default radix is hexadecimal, however, the value C9 (without the \$ prefix) will be considered equivalent. Of course, if you change the default radix, decimal values like 117 and 48 will be considered hexadecimal as well unless preceded by the decimal prefix.

XGS Micro Studio supports radix specification using the following prefixes. These prefixes are necessary when using a base that is not currently specified as the default. See Table 3.1 for a complete listing.

**Table 3.1 – Number base suffixes.**

Prefix	Description
D'	Decimal
\$	Hexidecimal
%	Binary
O'	Octal

### 3.4.1.1.3 - Tab Width

Sets the assumed width of a tab character. The maximum value is 20.

### 3.4.1.1.4 - Case-Sensitive Symbols

This option is currently disabled as it is not supported fully by the assembler.

### 3.4.1.1.5 - Additional Assembler Command-Line Options

While the Configure Tools window provides complete control over the assembler, it is sometimes convenient or necessary to specify custom command-line strings.

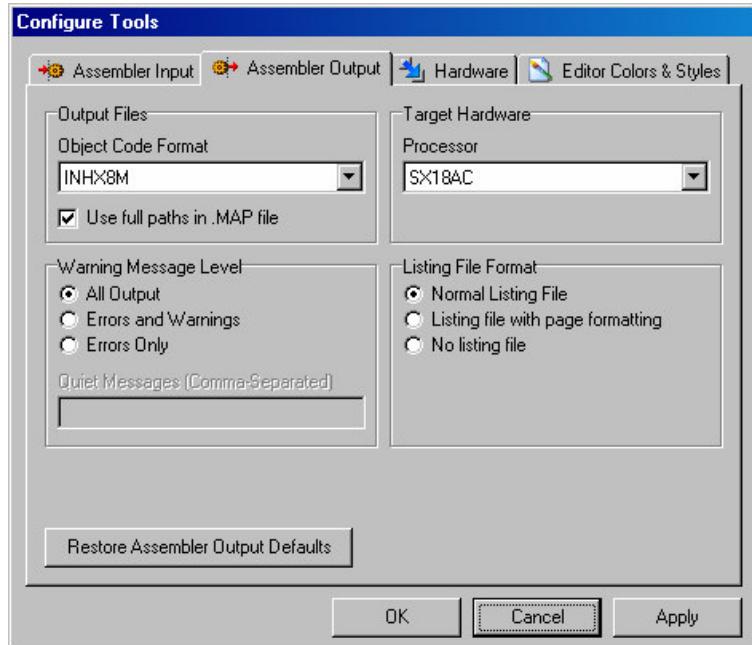
### 3.4.1.1.6 - Restore Assembler Input Defaults

Restores the default values for this tab.

### 3.4.1.2 - The Assembler Output Tab

This tab, shown in Figure 3.9, configures the output produced by the assembler.

Figure 3.9 – The Assembler Output Tab.



#### 3.4.1.2.1 - Object Code Format

XGS Micro can output assembled programs in a number of formats, as listed in Table 3.2.

**IMPORTANT!** Only INHX8M (.HEX) output files are compatible with XGS Micro Studio!

**Table 3.2 – Supported output formats.**

Option	Extension	Description
INHX8M	HEX	Intel 8-Bit merged hex
INHX8S	HXH/HXL	Low- and high-order bytes split among two files.
INHX16	HEX	16-bit version of INHX8M.
INHX32	HEX	32-bit version of INHX8M.
IEEE695	SXE	IEEE-695 compliant object code and debug info.
BIN16	OBJ	Binary object format.

### 3.4.1.2.2 - Use Full Paths in .MAP File

XGS Micro Studio does not currently use the .MAP file, but this option specifies whether or not it should contain full paths of referenced files. This option can be useful when exchanging assembler output between XGS Micro Studio and other development tools.

### 3.4.1.2.3 - Processor

The processor targeted by the assembler when generating object code. For XGS Micro compatibility, always leave this set to SX52.

### 3.4.1.2.4 - Warning Message Level

Controls the type and quantity of messages generated by the assembler. Table 3.3 lists the available options.

**Table 3.3 – Warning message level options for the SASM assembler.**

Level	Description
All Output	All messages will be output.
Errors and Warnings	Only severe errors and warnings will be output.
Errors Only	Only severe errors will be output. All non-critical messages are suppressed.

For most development purposes, **All Output** is the recommended setting.

#### 3.4.1.2.5 - Listing File Format

The assembler can produce a program's listing file in one of three ways:

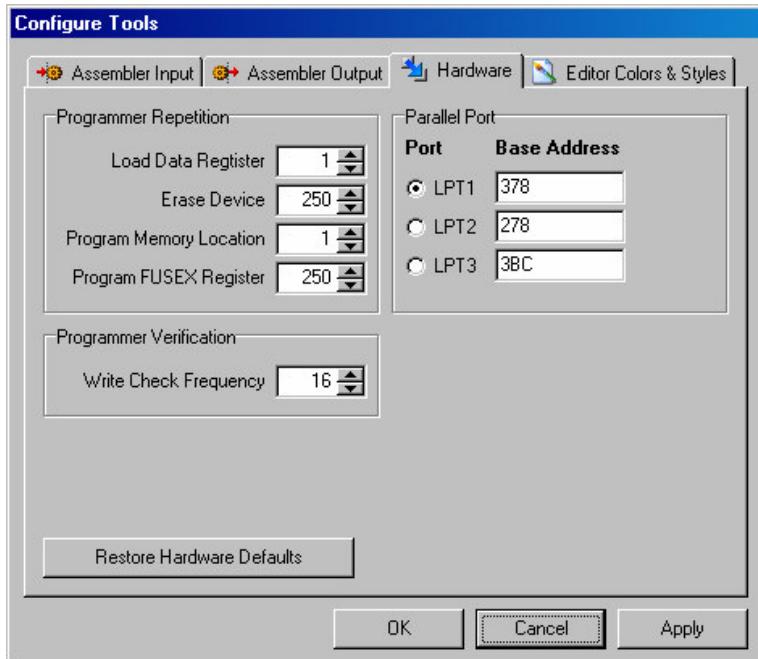
- Normal listing file
- Listing file with page formatting (makes printing listing files easier)
- No listing file

#### 3.4.1.2.6 - Restore Assembler Output Defaults

Restores the default values for this tab.

#### 3.4.1.3 - The Hardware Tab

This tab, shown in Figure 3.10, configures the interface (driver) between XGS Micro Studio and the physical XGS Micro hardware attached to the development PC.

**Figure 3.10 – The Hardware Tab.**

#### **3.4.1.4 - Programmer Repetition**

These values can be used to alter the number of times a command sent to the XGS Micro hardware is repeated while programming it. Higher numbers may, in some cases, increase the command's reliability (usually at a negligible performance cost).

Generally speaking these settings need not be changed.

#### **3.4.1.5 - Write Check Frequency**

As XGS Micro Studio programs the XGS Micro hardware, it will periodically verify data as it is written to help ensure correct programming. This value alters the frequency at which this verification occurs. The higher the number, the more reliable the programming and the longer it takes.

Generally speaking this setting need not be changed.

### 3.4.1.6 - Parallel Port

These settings allow LPT1 through LPT3 to be selected as the active parallel port. Furthermore, the location of the ports within memory can be altered in the case of unusual system configurations.

**NOTE**

Parallel port addresses are both entered and displayed in hexadecimal.

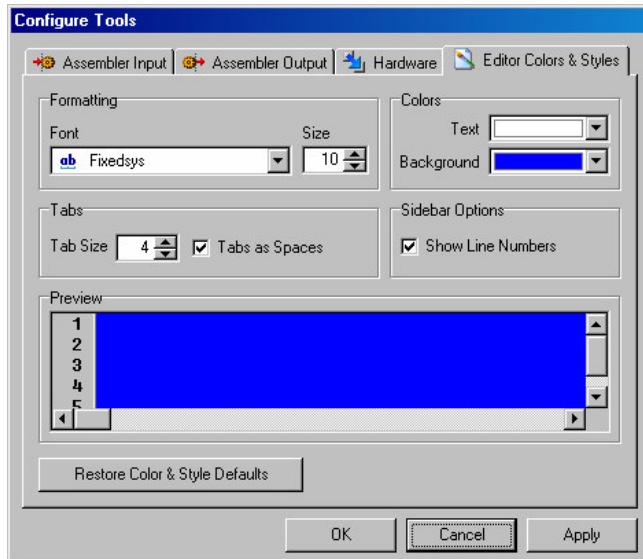
### 3.4.1.7 - Restore Hardware Defaults

Restores the default values for this tab.

### 3.4.1.8 - The Editor Colors & Styles Tab

This tab, shown in Figure 3.11, configures the editor's color and font settings. As the look and feel changes, the preview text box automatically updates to reflect it. Note that you can also edit the contents of the preview box to test various aspects of the look and feel before committing them.

**Figure 3.11 – The Editor Colors & Styles Tab.**



### 3.4.1.9 - Font

The font in which all document text is displayed. Also the font used for line numbers.

*XGameStation™ Micro Edition User Guide*

### 3.4.1.10 - Size

The size of the font in which all document text is displayed. Also the font size used for line numbers.

### 3.4.1.11 - Text Color

Foreground color used for text characters.

### 3.4.1.12 - Background Color

Document background color.

### 3.4.1.13 - Tab Size

Size (in characters) of document tabs.

### 3.4.1.14 - Tabs as Spaces

If checked, the editor generates a string of spaces instead of a tab character. Changing this option will not affect previously entered tabs.

### 3.4.1.15 - Show Line Numbers

If checked, a vertical gutter with line numbers is displayed in each document. Line numbers start at 1 (one).

## 3.5 - The Real-Time SX Interface

The **Real-Time SX Interface** is a powerful built-in tool that allows direct access to the XGS ME's SX52 processor. From this interface, you can perform the following tasks easily:

- Configure basic options like use of the carry flag, I/O sync, code protection and more.
- Configure more advanced functionality, like the reset and brownout timer durations and the oscillator/crystal.
- Read the current status of the device, including the device word, FUSE and FUSEX registers, and the program memory.
- Program the contents of a hex file to the processor's program memory.
- Limited control over the FUSE and FUSEX registers.

To display the Real-Time SX Interface, select **Real-Time SX Interface (Ctrl+I)** under the **Tools** menu.

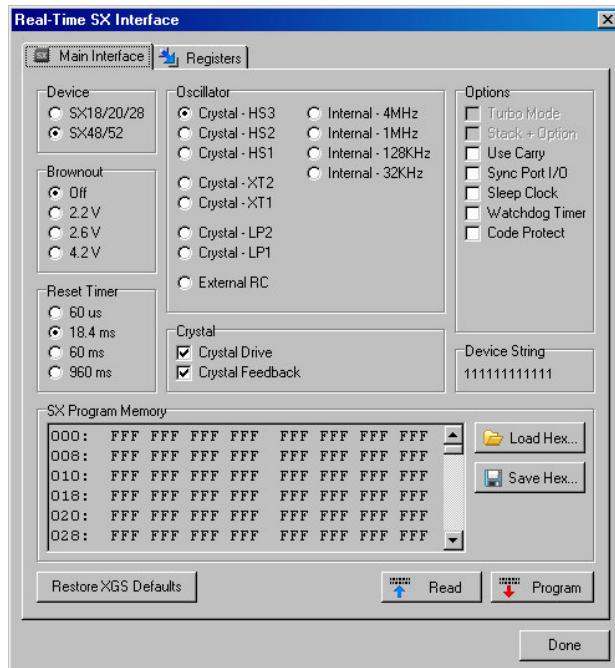
The SX interface consists of two tabs:

- The Main Interface
- Registers

### 3.5.1 - The Main Interface Tab

The main interface tab represents a high-level view of the SX52's configuration settings, and is displayed in Figure 3.12.

**Figure 3.12 – The Main Interface Tab of the Real-Time SX Interface.**



To program the XGS Micro using the current settings and program memory buffer, click **Program**. Click **Read** to read the status and program memory of the XGS Micro. Click **Restore XGS Defaults** to restore the factory settings of the XGS Micro (then click **Program** to write the changes to the physical hardware).

**NOTE**

The changes you make to the settings and program memory within the SX interface do **not** take effect until you click **Program**!

The options available within the Main Interface tab are as follows:

- **Device**  
Switches between the SX48/52 and the SX18/20/28 families of devices. For development on the XGS Micro, always leave this set for SX52. Changing this setting will result in incorrect programming of the device or other errors.
- **Brownout**  
Sets the duration of the brownout timer.
- **Reset Timer**  
Sets the duration of the reset timer, which dictates how long the system idles after reset before execution begins.
- **Oscillator**  
Configures the type of oscillator used to drive the SX. For most applications, the default settings need not be changed.
- **Crystal**  
Advanced control over the crystal used to drive the XGS Micro.
- **Options**  
Miscellaneous options, such as use of the carry flag, I/O port sync, sleep clock activation, watchdog timer activation, and code protection. The default settings here will usually do as well.
- **Device String**  
Reflects the device ID string of the last XGS Micro either read from or written to. This is updated each time you press the Program or Read buttons.

Aside from configuration options, an important task of the SX Interface is dealing with hex programs.

### 3.5.1.1 - Programming the XGS Micro With a Hex Program

Take the following steps to program the XGS Micro with a hex program. If you do not already have a hex program handy, assemble an existing source file using XGS Micro Studio. Make sure the output format is set for **INHX8M**.

- Click **Load Hex** and select the file you would like to program the device with.
- When the file is loaded, its program's options will override the current settings of the SX Interface. Confirm that these are the intended options, and make changes if necessary.
- Click **Program**.

### 3.5.1.2 - Saving the XGS Micro to a Hex Program

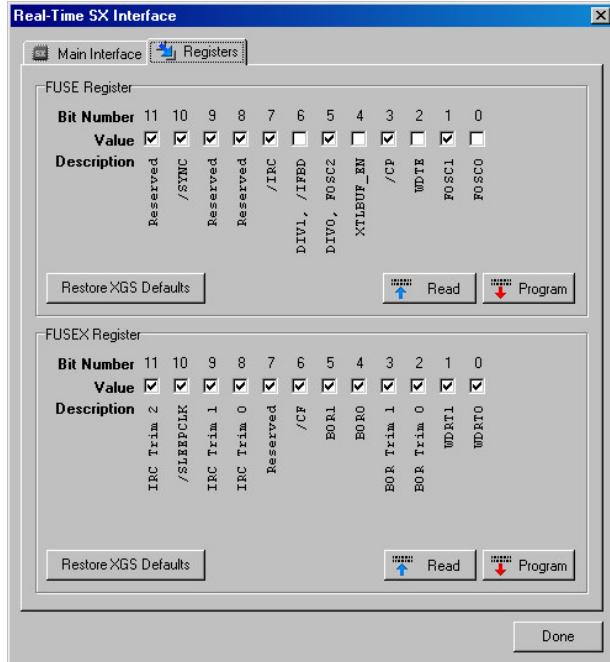
Take the following steps to save the current status and program memory of the XGS Micro to a hex file. Note that the results of this process can be used with the process above to program the subsequently device.

- Click **Read**. The device will be read into the program memory buffer and settings of the SX Interface.
- Click **Save Hex** and select the appropriate destination file.

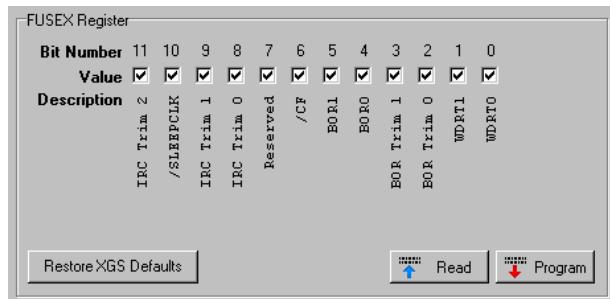
### 3.5.2 - The Registers Tab

The Registers Tab is the low-level counterpart to the Main Interface Tab, displaying the SX52's configuration settings directly, rather than translating and grouping their functionality. It is displayed in Figure 3.13.

**Figure 3.13 – The Registers Tab of the Real-Time SX Interface.**



The Registers Tab allows you to directly modify the SX's FUSE and FUSEX registers without reprogramming the entire device. For example, the clock rate of a running program can be changed quickly without reprogramming it entirely. Figure 3.14 displays the FUSEX register displayed in the form of checkboxes.

**Figure 3.14 – The SX52 FUSEX Register in Checkbox Form.****NOTE**

Due to the nature of the SX's program memory (where FUSE and FUSEX are stored), zero bits cannot be written to the device after it has been programmed! Only bits set to 1 may be changed. This is why the Registers tab only provides "limited" control over the device.

**NOTE**

In the Registers tab, register bits are edited and viewed as checkboxes. A checked box represents a one bit, while an unchecked box represents zero.

**3.5.2.1 - Reading Registers**

To read a register from the XGS Micro, click Read in either the FUSE or FUSEX pane (depending on which register you want to read). The register value will immediately update the bit check boxes.

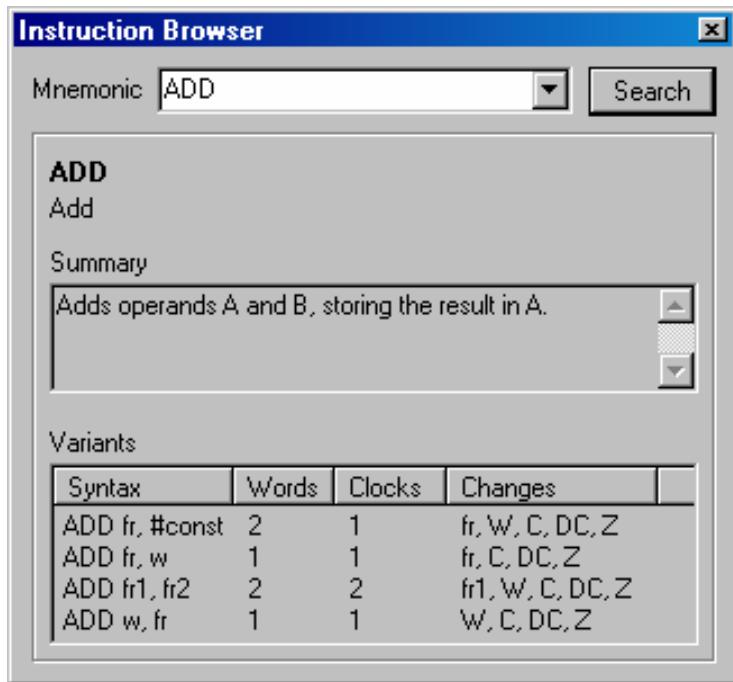
**3.5.2.2 - Programming Registers**

To program a register on the XGS Micro, click Program in either the FUSE or FUSEX pane (depending on which register you want to program). The values of the register's bit check boxes are used to program the register. Remember, only one bits may be programmed. Zero bits will have no effect.

## 3.6 - The Instruction Browser

The Instruction Browser (see Figure 3.15) allows you to quickly look up information on SX instructions using their mnemonic (**MOV**, **ADD**, **SUB**, etc.). Using the instruction browser allows you to easily calculate clock-accurate timings or object code size. It can also be used to clarify instructions you don't currently recognize or recall.

**Figure 3.15 – The Instruction Browser Displaying the ADD Instruction.**



Display the Instruction Browser by selecting **Instruction Browser (Ctrl+B)** under the **Tools** menu.

### 3.6.1 - Instruction Lookups

To look up an instruction, enter it in the text field marked **Mnemonic** and click **Search**.

If the instruction is found, it will appear in the area below. From top to bottom, the following pieces of information are displayed:

- **Instruction Mnemonic**

The instruction's mnemonic as understood by the assembler. This is the same mnemonic you entered in the **Mnemonic** box above.

- **Instruction Name**

A more descriptive name that briefly explains the instruction.

- **Summary**

A full explanation of the instruction, its functionality, operands and side effects.

- **Variants**

For each variant of the instruction, this table lists its syntax (operands), words of program memory required, clock cycles consumed during execution, and important registers changed after the instruction executes.

**TIP**

The instruction browser will always remain above your documents, allowing you to type while reading it simultaneously. Close the window to free up the screen space.

# Chapter 4: Using SX-Key

The SX-Key is a hardware/software package from Parallax, Inc. for programming SX microcontrollers, like the SX52 used in the XGameStation Micro Edition. The SX-Key IDE is the development software used to write programs for SX chips and program them via the SX-Key hardware. This chapter is a complete overview of how SX-Key is used.

## 4.1 - Installing the SX-Key Software

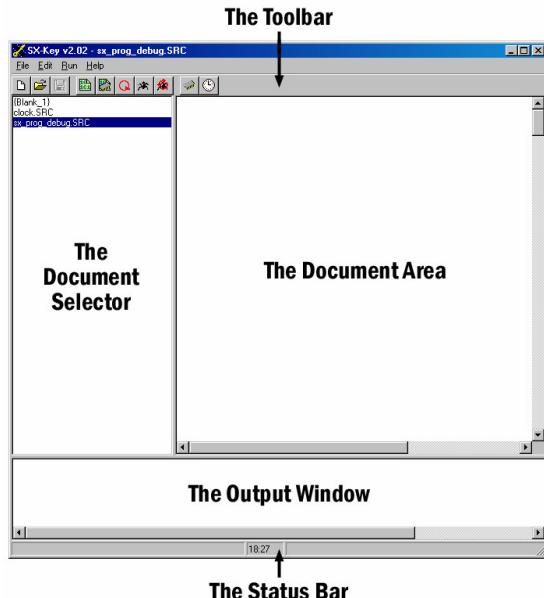
If you haven't already done so, you'll of course need to install SX-Key before you can use it.

See **Guide 2 – How to Load an Example Program onto the XGS ME** for step-by-step instructions for installing and configuring SX-Key.

## 4.2 - Elements of the Interface

The SX-Key IDE is designed to be familiar to users of other IDEs. Figure 4.1 depicts the layout of the IDE as it usually appears.

**Figure 4.1 – The SX-Key IDE Interface.**



The program is split up into five main areas:

- The Toolbar & Menu Bar
- The Document Selector
- The Document Area
- The Output Window
- The Status Bar

The following sections explain these areas in full detail.

#### 4.2.1 - The Toolbar & Menu Bar

The toolbar (seen in Figure 4.2) encapsulates most of the SX-Key IDE's in easy-to-use buttons.

**Figure 4.2 – The SX-Key IDE Toolbar.**



Use the toolbar to create, open and save documents, invoke built-in tools like the **Device** and **Clock** windows, and assemble, write and view listings of programs.

The following is a brief explanation of each toolbar button:

- **New**  
Creates a new document.
- **Open (Ctrl+O)**  
Opens a document for editing.
- **Save (Ctrl+S)**  
Saves the currently active document.
- **Assemble (Ctrl+A)**  
Assembles the currently selected document. The results are printed in the output window.

-  **Program (Ctrl+P)**  
Assembles the currently selected document and program it to the XGS ME. The results are printed in the output window.
-  **Run (Ctrl+R)**  
Assembles the currently selected document, programs it onto the XGS ME, and generates a clock signal to run the program. The results are printed in the output window.
-  **Debug (Ctrl+D)**  
Assembles, programs and runs the currently selected document. Also activates debugger, allowing the execution of the program to be controlled and its memory to be monitored.
-  **Debug (Reenter) (Ctrl+Alt+D)**  
Same as **Debug**, but does not reprogram the XGS ME. Allows the last debugging session to be resumed without waiting for the programming process. The current breakpoint and data watches can be changed, but the program source code cannot.
-  **Device (Ctrl+I)**  
Opens the **Device** window, allowing direct control over the SX52's configuration registers and program memory.
-  **Clock (Ctrl+K)**  
Opens the **Clock** window to allow control over the frequency and activity of the SX-Key oscillator as a program runs.

In addition to the items on the toolbar, the following options are available under the various menus (only items not covered in the toolbar section are listed):

#### 4.2.1.1 - The File Menu

- **Print...**  
Displays the print dialog box for the currently selected document.

#### 4.2.1.2 - The Edit Menu

- **Undo (Ctrl+Z)**  
Undoes the last action in the selected document. Undo is good for multiple levels, allowing entire sequences of modifications to be undone.

- **Redo (Ctrl+Y)**  
Redoes the last action undone in the selected document. Redo is good for multiple levels, allowing entire sequences of modifications to be redone.
- **Cut (Ctrl+X)**  
Cuts the current selection from the selected document to the clipboard.
- **Copy (Ctrl+C)**  
Copies the current selection in the selected document to the clipboard.
- **Paste (Ctrl+V)**  
Pastes text content in the clipboard to the currently selected document.
- **Find... (Ctrl+F)**  
Displays the Find dialog, allowing the currently selected document to be searched.
- **Find Next (F3)**  
Finds the next occurrence of the last search string in the selected document.
- **Find/Replace... (Ctrl+H)**  
Displays the Replace dialog, allowing the currently selected document to be searched for text strings to replace with a replacement string.
- **Go to Line Number... (Ctrl+G)**  
Displays a dialog asking for a line number. The currently selected document is then scrolled to the specified line.
- **Clear Errors**  
Clears and hide the Output window. The window will reappear upon the next assembly.

#### 4.2.1.3 - The Run Menu

- **View List (Ctrl+L)**  
Generates an annotated program listing of the currently selected document in the document's directory and opens it.

#### 4.2.1.4 - The Help Menu

- **Contents**  
Displays a brief explanation of the **WATCH** and **BREAK** directives, used with the debugger.

### 4.2.2 - The Document Selector

The document selector, as seen in Figure 4.3, lists the currently open documents.

**Figure 4.3 – The SX-Key IDE Document Selector.**

Click a file in the document selector to bring it to the foreground. As documents are opened and closed, the document selector automatically updates.

### 4.2.3 - The Document Area

The document area is where all open documents are displayed. Documents cannot be resized or tiled; they always remain maximized to fill the document area, and as such, only one document is visible at a time.

### 4.2.4 - The Output Window

The output window contains the output of assembler operations, as seen in Figure 4.4

**Figure 4.4 – The SX-Key IDE Output Window.**

```
D:\Business\XGameStation\Work\Weekend_9_24_2004\x86_prog_9_24_2004\x86_prog_debug.SRC[32] Line 32: Warning 51, Pass 1: Obsolete keyword: <option> for this device
D:\Business\XGameStation\Work\Weekend_9_24_2004\x86_prog_9_24_2004\x86_prog_debug.SRC[1] Line 1: Warning 65, Pass 1: No IIRC, CAL directive. Default IIRC_SLOW being used
D:\Business\XGameStation\Work\Weekend_9_24_2004\x86_prog_9_24_2004\x86_prog_debug.SRC[1] Line 1: Warning 66, Pass 1: No FREQ directive. Default 50 MHz being used
```

The output window will accumulate output from successive assembly operations. To clear and hide the output window until the next assembly, select **Clear Errors** under the **Edit** menu.

## 4.2.5 - The Status Bar

The status bar (seen in Figure 4.5) runs along the bottom of the screen at all times and tracks basic information like the status of special keys and the location of the cursor within the current document, among other things.

**Figure 4.5 – The SX-Key IDE Status Bar.**



## 4.3 - Editing, Loading and Running Programs

The two primary purposes of the IDE are editing source code and programming the assembled programs onto the XGS ME console.

### 4.3.1 - Loading & Editing Source Code

Under the **File** menu, Select **Open...** (**Ctrl+O**) to open an existing source file, or **New** to create a new source file. Once open, any source file can be edited just like any other standard windows program. Multiple levels of undo and redo are supported, as is the clipboard, find and replace, and so on.

### 4.3.2 - Programming, Running and Debugging Programs

#### NOTE

The XGS ME's **SYSMODE** switch (located at the back of the board near the parallel port) must be set to **KEY** mode in order for SX-Key to program the processor with your code. To run a program without SX-Key, the **SYSMODE** switch must then be set to **RUN** mode. Remember, however, that no programming, reading or debugging of the processor can be done if **SYSMODE** is not set to **KEY** mode.

Once a program is ready to execute, it can be programmed onto the XGS ME by the options found under the **Run** menu.

To verify that your program is free of syntax errors, or to generate an assembled object code file, select **Assemble** (**Ctrl+A**). Depending on how you have configured SX-Key, (see the next section, **Configuring SX-Key**) the assembled object file will appear in either the directory of your source file, or SX-Key's own internal directory for storing files output by the assembler.

Once your program is verified by the assembler, you can program it onto the XGS ME with the **Program** (**Ctrl+P**). This option will assemble and program your code to the hardware, but will not cause SX-Key to generate a clock signal of its own. You can, however, run this program using the XGS ME's built-in

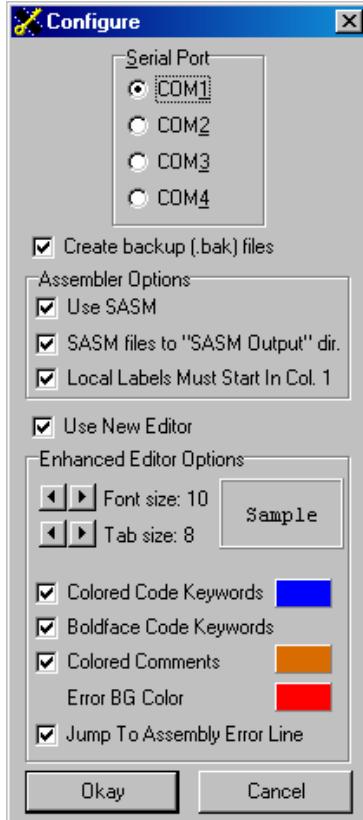
oscillator by switching the **SYSMODE** switch to **RUN** mode. Just remember to set it back to **KEY** mode before attempting to program it again.

If you would like to use the SX-Key's built-in oscillator to clock your program (allowing you to run at numerous different speeds with the **Clock** window or use the speed requested in your source code via the **FREQ** directive), select **Run (Ctrl+R)**. This will assemble your code and program it onto the hardware, and will cause SX-Key to generate its own clock signal, effectively allowing your program to execute.

## 4.4 - Configuring SX-Key

Select **Configure... (Ctrl+U)** under the **Run** menu to display the **Configure** window (depicted in Figure 4.6).

**Figure 4.6 – The SX-Key Configure Window.**



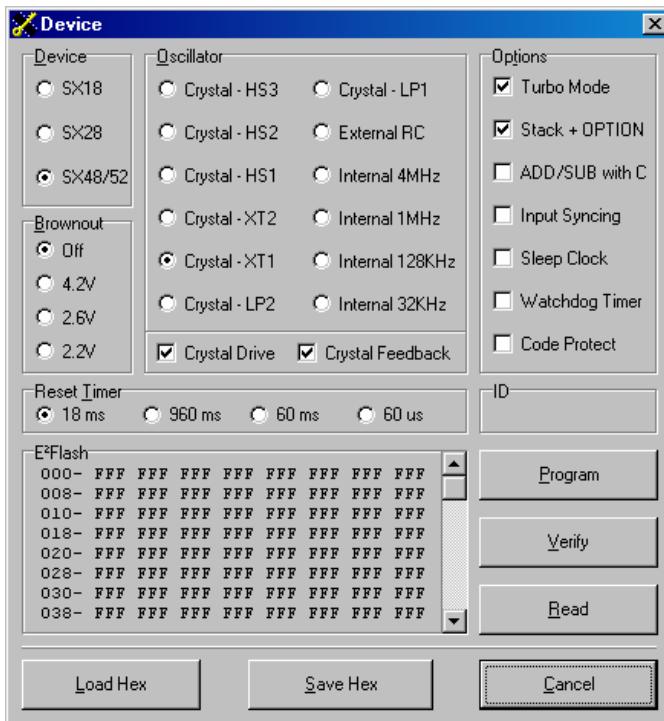
This window allows you to configure the SX-Key IDE. The following sections explain each feature of the window in detail:

- **Serial Port**  
Allows you to specify which COM port the SX-Key cable is connected to.
- **Create backup (.bak) files**  
If checked, SX-Key will create a backup of the previous version of a modified document when it is saved.
- **Use SASM**  
If unchecked, an older version of the assembler will be used to assemble source files, which may be useful for legacy code but will most likely cause errors with new code. It is unlikely that this option need be changed (unchecked) for XGS ME development.
- **SASM files to “SASM Output” dir**  
If unchecked, SASM-generated files will appear in the directory of the selected source file. If checked, these special files will appear in the “SASM Output” directory, located in SX-Key’s installation directory.
- **Local Labels Must Start in Col. 1**  
If checked, local labels must start in the first column of the source file. When unchecked, these labels may be indented.
- **Use New Editor**  
Activates or deactivates the latest version of the SX-Key editor, as well as the configuration options that appear below this checkbox. Leave this option checked.
- **Font Size**  
The size of the source code font.
- **Tab Size**  
The size (in spaces) of a tab indentation. Limited to 2, 4, 6 or 8.
- **Colored Code Keywords**  
If checked, assembler keywords (instructions) will be highlighted with the specified color.
- **Boldface Code Keywords**  
If checked, assembler keywords (instructions) will be displayed in bold.
- **Colored Comments**  
If checked, comments will be highlighted with the specified color.
- **Error BG Color**  
Color used to highlight erroneous lines flagged by the assembler.
- **Jump to Assembly Error Line**  
If checked, the cursor will jump to the line of the first error after assembly.

## 4.5 - The Device Window

Selecting **Device...** (**Ctrl+I**) under the **Run** menu opens the **Device** window (depicted in Figure 4.7), which allows direct control over the XGS ME's SX52 processor.

**Figure 4.7 – The SX-Key Device Window.**



The following sections explain each option in the window.

- **Device**  
Specify the type of device is being programmed and debugged. Always leave this set for SX48/52 when developing for the XGameStation.
- **Oscillator**  
Set the oscillator used to drive the SX52 at runtime. The Table 4.1 explains the available options.
- **Brownout**  
Specify the threshold voltage for a brownout reset (or disable brownout entirely). For XGS ME development this option should be left in its default state.

- **Reset Timer**

Specify how long the processor should delay after a reset before the code execution begins. For XGS ME development this option should be left in its default state.

- **Options**

Provides a number of very advanced options that are of little value to XGS ME development, some of which aren't even applicable to the SX52 processor. These should be left to their default values.

- **ID**

When the SX52 is read with the **Read** button, its device identification string is read as well, and displayed here. Since all XGS ME consoles use the SX52 model, the device string is of little value.

- **E<sup>2</sup>Flash**

After reading the SX52 with the **Read** button or assembling a source program, this window contains a complete hex dump of the program memory.

- **Program**

Programs the current hex dump displayed in the **E<sup>2</sup>Flash** window onto the SX52.

- **Read**

Reads the SX52's program memory into the **E<sup>2</sup>Flash** window.

- **Verify**

Determines if the program memory of the SX52 matches the hex dump in the **E<sup>2</sup>Flash** window.

- **Save Hex**

Saves the **E<sup>2</sup>Flash** hex dump to a file.

- **Load Hex**

Loads a hex file into the **E<sup>2</sup>Flash** window.

**Table 4.1 – SX52 oscillator settings.**

Setting	Description
HS1...3 XT1...2 LP1...2	Specifies the oscillator drive capacity for high speed, medium speed crystal/resonator, and low power crystal/resonator clocks.
External RC	Specifies special drive for external resistor-capacitor clock circuits.
Internal 32 KHz...4 MHz	Specifies internal clock at indicated frequency.

## 4.6 - The Clock Window

Select **Clock...** (**Ctrl+K**) under the **Run** menu to open the **Clock** window (depicted in Figure 4.8).

**Figure 4.8 – The SX-Key Clock Window.**

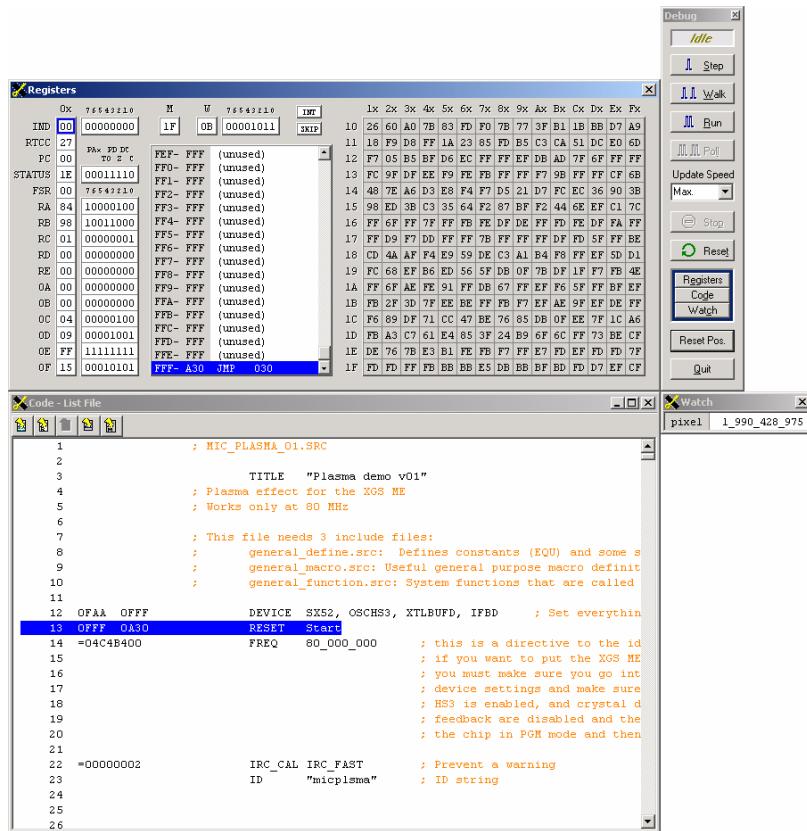
This window provides control over the activity and frequency of the SX-Key oscillator. The following sections cover each facet of the **Clock** window:

- **Freq (MHz)**  
Enter an exact clock rate, in MHz, for the SX-Key oscillator.
- **Frequency Slider**  
Vertical slider that allows the clock rate to be adjusted by hand in real-time, from 400 KHz (bottom) to 110 MHz (top).
- **On**  
When checked, the SX-Key oscillator is running. Unchecking this turns the oscillator off.
- **Reset**  
Reset the SX-Key oscillator.
- **Okay**  
Save the oscillator setting and close the Clock window.

## 4.7 - The Debugger

One of the most useful aspects of SX-Key is its integrated debugger, which is invoked by selecting **Debug (Ctrl+D)** from the **Run** menu. The **Debug (reenter) (Ctrl+Alt+D)** option will bring up the debugger as well, without reprogramming the XGS ME. The debugger, which spans multiple windows, is shown in Figure 4.9.

Figure 4.9 – The SX-Key Debugger.



The debugger is perhaps the most complex part of the SX-Key IDE. The following sections explain the debugger from both an overview and detail-oriented perspective.

### 4.7.1 - Debugger Overview

The debugger is composed of four main windows (some of which may not be visible at all times):

- The Registers Window
- The Code Window
- The Watch Window
- The Debug Palette

The **Debug** palette provides the most overall control of the debugging process, allowing you to stop, start and step through the program's execution, as well as hide or show the other three windows.

**TIP**

If at any time you find that a debugger window is not visible, press its corresponding button on the **Debug** palette to bring it back into the foreground.

The **Registers** and **Watch** windows allow you to view the exact state of the processor and its memory in real-time as it executes, or between runs. The **Code** window allows you to track the processor as it moves through your source code. You can also move the breakpoint using this window. Together, these windows give you total control over the processor and the ability to monitor and even alter every memory register.

The following sections describe each window in detail.

## 4.7.2 - The Debug Palette

The **Debug** palette, pictured in Figure 4.10, is the main control panel for the debugger.

**Figure 4.10 – The Debug Palette.**



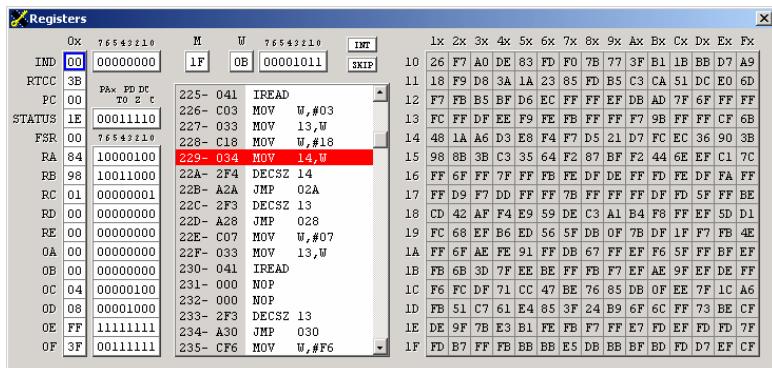
The palette consists of a vertical strip of buttons and a pull-down menu, described in the following sections:

- **Step (Alt+S)**  
Executes a single instruction and then immediately returns control to the debugger. Use this to incrementally step through your code and examine the effects of individual instructions.
- **Walk (Alt+W)**  
Executes code instruction-by-instruction, briefly pausing in between each to allow the user to watch the flow of execution and study its effects over time. Think of this mode as the automated equivalent of quickly and repeatedly using the **Step** button to move through the program.
- **Run (Alt+R)**  
Executes code until the breakpoint is reached or the user stops execution.
- **Poll (Alt+L)**  
If code is executing (via the **Run** button), **Poll** returns the state of the system to give the user an updated view while allowing execution to continue uninterrupted. If code is not executing, **Poll** runs until a breakpoint is hit, at which point it updates the display, then continues running again.
- **Update Speed**  
Leave this option at its default setting of **Max**.
- **Stop (Alt+P)**  
Stops execution and update the display, returning control to the debugger.
- **Reset (Alt+T)**  
Completely resets the processor to its initial state as if the board's reset button were pressed.
- **Registers (Alt+E)**  
Forces visibility of the **Registers** window. Useful if this window disappears or slips into the background.
- **Code (Alt+D)**  
Forces visibility of the **Code** window. Useful if this window disappears or slips into the background.
- **Watch (Alt+C)**  
Forces visibility of the **Watch** window. Useful if this window disappears or slips into the background.
- **Reset Pos.**  
Returns all windows to their default positions in the center of the screen.
- **Quit (Alt+Q)**  
Close the debugger and return to the editor.

### 4.7.3 - The Registers Window

This window, depicted in Figure 4.11, is the most informative part of the debugger interface, constantly updated to display all register values and the processor's location within the program (unless the program is in full-speed execution mode).

Figure 4.11 – The Debugger Registers Window.



Along the left-hand side of the screen, the global and system registers are displayed in both hexadecimal and binary format. Near the center, a scrolling window displays the disassembled program memory of the chip as well as the location of the next instruction to be executed. Above this window are the **M** (mode) and **W** (work) registers, as well as the **INT** (Interrupt) and **SKIP** flags. The flags turn blue when set, white when clear.

The last section, to the right, is a matrix of all registers across all banks of RAM, displayed in hexadecimal. The currently active bank is highlighted in white.

**TIP**

Whenever the **Registers** window is updated, any registers that have changed since the last update are highlighted in red. If a binary register display contains data that has changed, the changes are highlighted on a per-bit basis, allowing you to see exactly what changes have occurred.

#### 4.7.3.1 - Real-Time Manipulation of Registers

Perhaps the most powerful feature of the **Registers** window is that it can be manipulated by the user at runtime. Click on any register display, binary or hexadecimal, to select it. Once selected (visible by the inverted colors), enter a new value and press Enter. During modification, pressing Escape (before pressing enter) cancels the changes and reverts the register to its original value. Binary values are modified by clicking the desired bit, which toggles its value. The debugger will immediately update the processor, allowing you to see your changes take effect upon the next execution.

#### 4.7.4 - The Watch Window

This window, depicted in Figure 4.12, displays the all registers marked for watching by the source code. The **WATCH** directive is used in your source code to mark which registers should be displayed at debug

*XGameStation™ Micro Edition User Guide*

time in the **Watch** window, as well as their desired format and size. Like the **Registers** window, this window is constantly updated to allow real-time monitoring of watched registers.

**Figure 4.12 – The Debugger Watch Window. The four-byte value represented by pixel is being watched in decimal mode.**



Also like the **Registers** window, watched registers can be manipulated in real-time by simply clicking them and entering a new value. The watch window, depending on the value of the currently selected field, can accept values in binary, hexadecimal, decimal or string formats. To state the format of the value you're entering, use the standard prefix symbols used in the assembler-- % for binary, \$ for hexadecimal, and nothing for decimal. Strings are entered as-is.

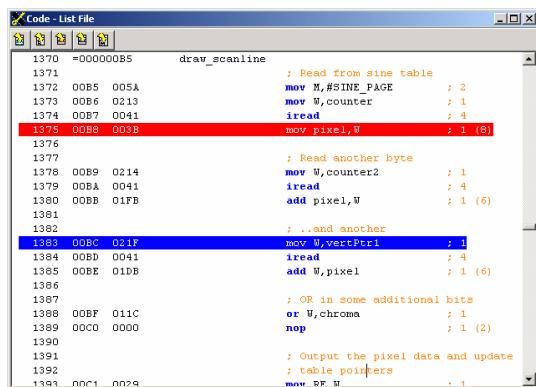
### TIP

Whenever the **Registers** window is updated, any registers that have changed since the last update are highlighted in red. If a binary register display contains data that has changed, the changes are highlighted on a per-bit basis, allowing you to see exactly what changes have occurred.

## 4.7.5 - The Code Window

Rounding out the debugger interface is the **Code** window, depicted in Figure 4.13.

**Figure 4.13 – The Debugger Code Window.**



The code window contains a complete listing of the original source file. The next instruction to be executed is highlighted with a blue line, and the currently active breakpoint is highlighted with a red line.

### 4.7.5.1 - Setting the Breakpoint

The initial breakpoint can be set by the source code itself, using the **BREAK** directive. At debug time, the **Code** window is used to move the breakpoint if necessary, simply by clicking the line that corresponds with the new instruction at the breakpoint's desired location. Assuming this is a valid line, it will turn red indicating that it is now the breakpoint location. If it does not turn red, the line is not a valid part of the code (it is either a blank line, or a non-functional part of the code annotation).

### 4.7.5.2 - The Code Window Toolbar

The **Code** window has a small toolbar of its own, shown in Figure 4.14.

**Figure 4.14 – The Code Window Toolbar.**



The action of each button is as follows:

- **Jump to Code**  
Scrolls the window to the source line corresponding to the start of the assembled code (not necessarily the start of execution).
- **Jump to Reset Line**  
Scrolls the window to the source line corresponding to the start of execution upon a reset.
- **Jump to Breakpoint**  
Scrolls the window to the location of the active breakpoint (if a breakpoint has been selected).
- **Jump to “Next Run” Line**  
Scrolls the window to the source line corresponding to the next instruction that will be executed.
- **Jump to Main**  
Scrolls the window to the source label **Main**, if defined.

## Chapter 5: Troubleshooting

This chapter deals exclusively with problems that may arise during the use of the XGameStation Micro Edition. The problems and their solutions are categorized by the operation they pertain to, making it easy to quickly refer to this section when trouble strikes.

### **WARNING!**

Always make sure to reset the system before deciding something isn't working. Often the system will behave erratically just after programming (among other times) if it isn't running off a fresh reset. Do not hesitate to reset the machine frequently, especially after programming, as this is often the problem. You can ensure that the system has been properly restarted by observing the brief LED animation next to the SX20 programmer on the lower left-hand side of the board.

## 5.1 - XGameStation Micro Edition Console

This section deals with hardware problems regarding the XGS ME console itself.

### **5.1.1 - Power Supply Plug Doesn't Fit Outlet**

If you are in the United States, only a US power supply will fit wall outlets. If the power supply that came with your XGS does not fit, your package came with the international power supply.

If you are outside of the United States, you will need the international power supply to connect to wall outlets. This power supply supports multiple blades, which allow it to fit into a variety of sockets. Be sure to try all included blades if you are having trouble connecting to the wall. If your power supply does not support multiple blades, your package came with the US power supply.

If your power supply is the incorrect model, has been lost, or needs to be replaced, you can use any power supply with the following specifications:

- 9V DC (does not need to be regulated).
- 500 mA (or greater).
- 2.1mm male power jack with ring ground, tip positive.

## 5.1.2 - System Does Not Turn On / No Power

This is a broad problem that may be the result of numerous conditions, but the most common are listed here. Remember that any such problem may not simply be the product of a single cause, but possibly multiple causes.

- Verify that the power supply is connected firmly to both the XGS ME power jack and the wall outlet.
- Verify that the power supply being used matches the specifications listed in the previous section.
- It is always possible that incorrect prior usage and/or experimentation has resulted in a burnout. If this is the case, the unit will need to be replaced.
- If the unit is brand new and has never been used, it is possible that you have received a defective product. If this is the case, contact Nurve Networks technical support. The defective unit will need to be returned and possibly analyzed.

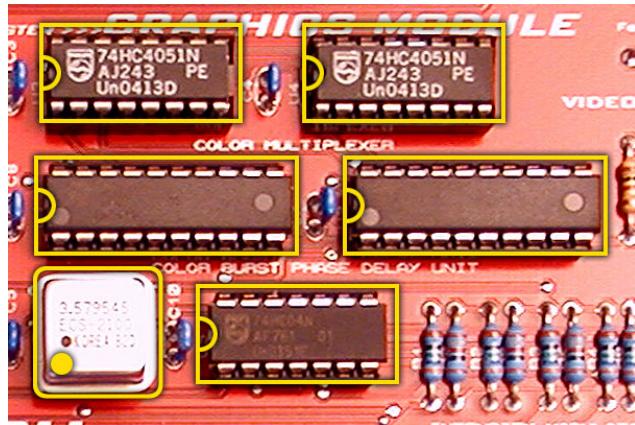
## 5.1.3 - No Video Output

If you have verified that the XGS ME console is powered and turned on (indicated by the power LEDs at the front of the board), and the system is not producing video, verify the following:

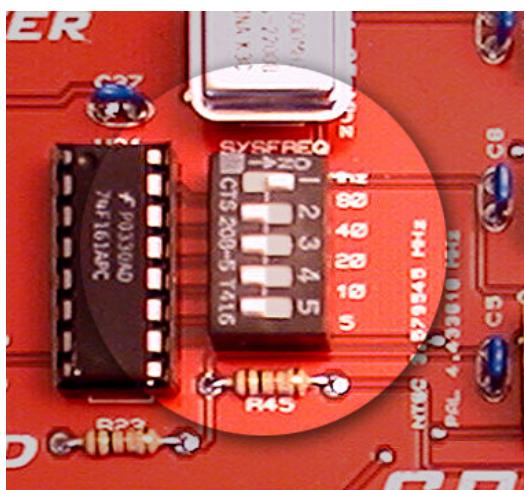
- The processor is programmed with a program that is reliably outputting video. Video output may not be seen if the processor is loaded with a buggy program or one that is not written to produce video. First reprogram the processor with one of the original game or graphics demos to verify video output with a reliable program.
- The color burst crystal and video subsystem chips are firmly connected to their sockets and are properly oriented. See Figure 5.1 for a visual reference of how these chips are properly connected with an overlaid outline guide.
- The **SYSMODE** switch is in **RUN** mode (or **KEY** mode if the SX-Key is attached and set to clock the system). If the switch is in **RUN** mode, also verify that the oscillator DIP switch is set to 80 MHz, the required speed for video signal generation, as shown in Figure 5.2.
- The XGS ME console is properly connected to a television set using the two-lead RCA cable included with the system (or equivalent). See **Guide 1 – How to Run the Pre-Loaded Demo** in Chapter 2 for a complete explanation of properly attaching the XGS ME to a television.
- The television is turned on and set to the proper A/V input source.
- The NTSC color burst crystal (**3.579545 MHz**) is connected if outputting to an NTSC television, and the PAL color burst crystal (**4.43 MHz**) is connected if outputting to a PAL television. Only NTSC and PAL standards are supported. Also make sure that the program running on the processor has been written for the correct standard.

- The **Brightness** and **Saturation** potentiometers are set to reasonable values. Bad values will usually only result in poor or shaky video signals, but certain TVs may not appear to output anything at all in extreme cases.
- The television's video configuration is set to reasonable settings, including its own control over brightness, contrast, saturation, etc.

**Figure 5.1 – The color burst crystal and video subsystem chips properly connected to the XGS ME. Notice the U-shaped alignment notch at the end of each chip, and the dot in the corner of the oscillator. Use the overlaid outlines as a guide.**



**Figure 5.2 – The System Oscillator DIP Switch Set to 80 MHz.**

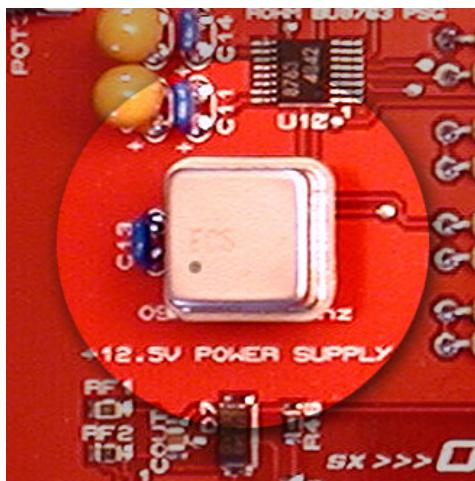


## 5.1.4 - No Audio Output

If you have verified that the XGS ME console is powered and turned on (indicated by the power LEDs at the front of the board), and the system is not producing audio, verify the following:

- The processor is programmed with a program that is reliably outputting audio. Audio output may not be heard if the processor is loaded with a buggy program or one that is not written to produce audio. First reprogram the processor with one of the original sound-producing demos to verify audio output with a reliable program. Remember that many demos and programs do not produce audio at all.
- The sound chip oscillator is firmly connected to its socket and is in the proper orientation, as shown in Figure 5.3.
- The **SYSMODE** switch is in **RUN** mode (or **KEY** mode if the SX-Key is attached and set to clock the system). If the switch is in **RUN** mode, also verify that the oscillator DIP switch is set to **80 MHz**, unless you know the currently loaded program is meant to run at a different speed.
- The XGS ME console is properly connected to a television set using the two-lead RCA cable included with the system (or equivalent). See **Guide 1 – How to Run the Pre-Loaded Demo** in Chapter 2 for a complete explanation of properly attaching the XGS ME to a television.
- The television is turned on and set to the proper A/V input source.
- The volume on both the XGS ME and the television is set to a reasonable level.

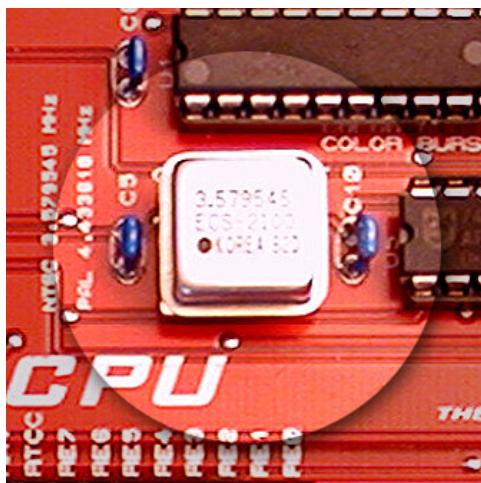
**Figure 5.3 – The Sound Chip Oscillator Properly Connected to the XGS ME.** The black corner dot is always on the bottom-left, allowing you to orient the chip properly within the socket.



### 5.1.5 - Video is Black and White

This is a simple and common problem arising from a poorly connected (or absent) color burst crystal. Make sure the color burst crystal, as seen in Figure 5.4, is firmly connected to its socket and in the proper orientation. Removal of the crystal will turn the video signal black and white, but is harmless.

**Figure 5.4 – The Color Burst Crystal Properly Connected to the XGS ME.**



## 5.1.6 - Video Signal is Fuzzy/Blurry/Noisy

Every TV is different, so the first consideration when trying to improve the video output of the XGS ME is the TV itself. Old and low-quality televisions may simply produce a blurry picture on their own. However, there are a couple sanity checks to perform if you would like to improve the quality of your video output:

- Make sure the XGS ME's **Saturation** and **Brightness** potentiometers are set to reasonable values. Tweak them until they seem optimal.
- Make sure the TV's own video configuration is set to reasonable settings, in terms of brightness, contrast, saturation, etc.
- Make sure the two-lead RCA A/V cable connecting the XGS ME and the television are connected firmly on both ends. If you are not using the cable included with the XGS ME, the cable itself may be of poor quality or poorly shielded, resulting in an excess of noise in the signal. Try upgrading to a higher-quality cable.

## 5.2 - XGS Micro Studio IDE

This section deals with software problems regarding XGS Micro Studio IDE and its use.

### NOTE

For any problems involving with the XGS ME hardware, make sure the system is properly powered as explained in the above sections!

### 5.2.1 - Cannot Initialize WinIO Library

On certain computers, XGS Micro Studio's first attempt at accessing the parallel port will fail. Simply run the program again and it should start up without a problem.

### 5.2.2 - Erratic Behavior in IDE/Crashes

In rare cases, the IDE will crash or behave erratically if an outdated or corrupt configuration file is present in its root directory.

- **Always** delete **main\_config.ini** when moving the XGS Micro Studio program directory from one location to another, especially if moving to another computer entirely.
- Delete **main\_config.ini** if you notice bizarre behavior that is not remedied by restarting the program.
- Hibernation/sleep mode in some computers and laptops can disrupt the port driver XGS Micro Studio uses to communicate with the XGS ME hardware. If you find the program crashes or behaves

erratically after returning from sleep mode, turn the system off completely, then boot up again. You may want to disable sleep mode in such cases if you plan to use XGS Micro Studio frequently.

### 5.2.3 - Can't Communicate with Device/Corrupted Programming

If XGS Micro Studio reports any trouble communicating with the device, such as errors or problems while reading and/or writing data during the programming process, check the following:

- The parallel cable is firmly connected to both the PC and the XGS ME.
- The **SYSMODE** switch is in **PGM** mode. The console cannot be programmed unless this mode is set.
- The proper parallel port is selected in XGS Micro Studio's **Configure Tools** window under the **Hardware** tab. **LPT1** is the most common port, but if your computer supports multiple ports, it may not be the one your cable is plugged in to. Also verify that **LPT1** is mapped to the proper port address. Remember that this address is both displayed and entered in hexadecimal (**without** the leading \$ symbol).
- If communication with the development PC fails continually, completely power down your machine and then boot it up again. In rare cases, the hardware itself, especially the parallel port itself, becomes permanently latched in the wrong state. In such cases, a simple restart **will not work**. A complete power-down of the system is required to restart all of the associated hardware.
- Make sure to close as many background tasks and applications as possible, especially virus checkers. The timings necessary to reliably communicate with the XGS ME's onboard programmer are delicate and can be disturbed by external processes in rare cases.

### 5.2.4 - Program Won't Assemble/Assembler Crashes

If you're having trouble getting your program to assemble, consider the following:

- Make sure the program is free of syntax errors.
- Make sure the assembler (**SASM.exe**) is present in the **Bin** directory and has not been renamed.
- There is a bug in the assembler that causes it to crash when include files are nested too deeply in directories and/or given filenames above a certain (long) length. To make sure this is the problem, temporarily move your program and all include files to the shallowest path you can, such as **C:\** or **C:\test**. If this solves the problem, your directory structure was triggering the bug. Work around this problem by moving your program into a directory closer to the root of the drive.
- Expanding on the item above, always use the shortest possible filenames and the shallowest possible directory structures you can.

## 5.2.5 - Instruction Browser Is Empty/Garbage

If you find the contents of the instruction browser either gone or suddenly garbage, the **Instr\_Ref.dat** file in the **Assets**/ directory is most likely missing or corrupt. Reinstall XGS Micro Studio to restore the file.

## 5.2.6 - “Please save this file before...” Error Message

Unsaved files cannot be assembled, programmed, or used to generate listing files. Please save your file first and try again.

# 5.3 - Parallax SX-Key IDE

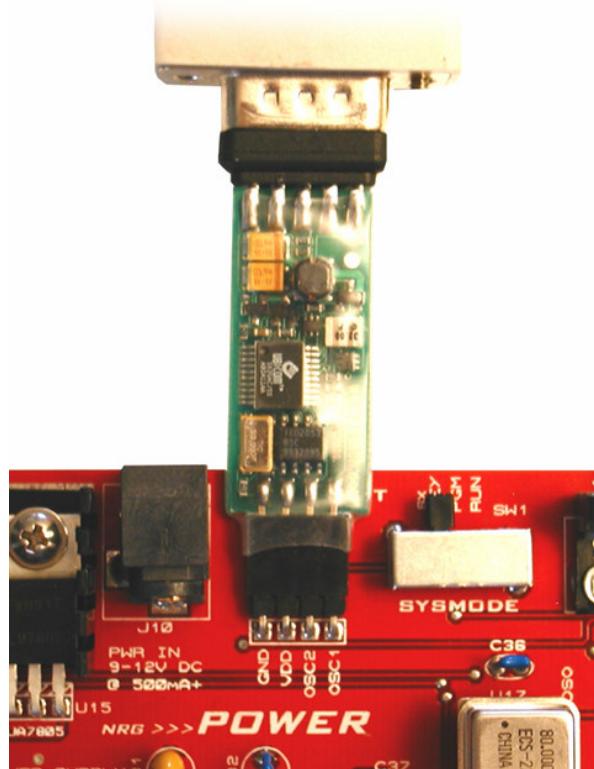
This section deals with software problems regarding Parallax SX-Key IDE and its use.

## 5.3.1 - “SX-Key not found on COMx” Error Message

This error message means that the SX-Key programmer unit cannot be detected by the SX-Key IDE. Check the following:

- The SX-Key cable is firmly connected to both the PC’s serial port and the SX-Key, and the SX-Key is firmly (but gently) connected to the SX-Key port (near the rear of the system by the parallel port). See Figure 5.5.
- The **SYSMODE** switch is in **KEY** mode. The console cannot be programmed via the SX-Key unless this mode is set.
- The proper serial port is selected in SX-Key’s **Configure** window. **COM1** is the most common port, but if your computer supports multiple ports, it may not be.
- In rare cases, prior usage/experimentation may have burned out the SX-Key. While this is unlikely, it is always a possibility with a hobbyist- and student-oriented product like the XGS ME.

Figure 5.5 – The SX-Key Properly Connected to the XGS ME.



### 5.3.2 - Program Won't Assemble/Assembler Crashes

If you're having trouble getting your program to assemble, consider the following:

- Make sure the program is free of syntax errors.
- Make sure the **SASM.dll** file is present in the root installation directory of SX-Key.
- There is a bug in the assembler that causes it to crash when include files are nested too deeply in directories and/or given filenames above a certain (long) length. To make sure this is the problem, temporarily move your program and all include files to the shallowest path you can, such as **C:\** or **C:\test**. If this solves the problem, you can conclude that your directory structure was triggering the bug. Work around this problem by moving your program into a directory closer to the root of the drive.
- Expanding on the item above, always use the shortest possible filenames and the shallowest directory structures you can.

### 5.3.3 - "Current file is READ ONLY and must be saved..." Error Message

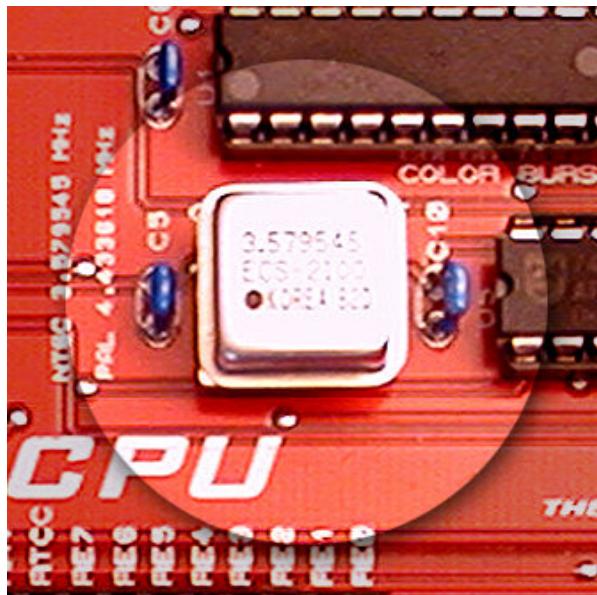
Unsaved files cannot be assembled, programmed, or used to generate listing files. Please save your file first and try again.

## 5.4 - Sanity Checks

Above all else, the following “sanity checks” are extremely important details to remember at all times when dealing with the XGS ME. More often than not, problems are the result of one or more of the following conditions not being met.

- Make sure the system is properly powered as explained in the above sections.
- If communication with the development PC fails continually, completely power down your machine and then boot it up again. In rare cases, the hardware itself, especially the parallel port itself, becomes permanently latched in the wrong state. In such cases, a simple restart **will not work**. A complete power-down of the system is required to restart all of the associated hardware.
- When interfacing between the PC and XGS ME, make sure to close as many background tasks and applications as possible, especially virus checkers. The timings necessary to reliably communicate with the XGS ME’s onboard programmer are delicate and can be disturbed by external processes in rare cases.
- Discrepancies between NTSC and PAL video signals. The XGS ME is designed to work with both PAL and NTSC TVs, but in order to do so, the proper color burst crystal must be connected (in the proper orientation) **and** the program running on the processor must be written for the correct standard as well. Figure 5.6 shows the correct orientation of the color crystal.
- Discrepancies between United States and international power supplies. The XGS ME can run in virtually any region provided it’s equipped with the proper power supply to connect to a wall outlet. For US residents, the US power supply comes standard and should work anywhere in the country. For non-US residents, the international power supply can be requested at the time of purchase, which comes with multiple blade attachments to fit the variety of non-US wall outlet sockets.
- As a general rule, allow 3-5 seconds to pass between the time you power the system down and power it back up again. While there are no specific dangers involved in switching the machine on and off faster than this, there have been rare cases of erratic behavior as a result of doing so.
- As a general rule, always use the shortest possible filenames and the shallowest possible directory structures you can.
- **Always** check the **Read Only** flag of any file or files you copy from a CD onto your hard drive. Attempting to work with read-only files can often lead to erratic behavior, crashes, freezes, and other such problems.

Figure 5.6 – Color crystal correctly connected and oriented. Note the location of the alignment dot.



#### 5.4.1 - Complete Connections & Settings Reference

This section is a list of every recommended or required connection and setting for the XGS ME console. When in doubt, simply compare your current connections and settings to those recommended by this list. Most common problems with the XGS ME are the result of one or more of these guidelines not being followed.

- **Connection:** Make sure the power supply is plugged firmly into the wall and firmly connected to the XGS ME power port on the rear of the board.
- **Connection:** When programming the XGS ME with the SX-Key, make sure the SX-Key is connected firmly but gently to the 4-pin right-angle SX-Key port next to the power socket, *not* the one near the joystick port on the front of the board! Also verify that the SX-Key is firmly connected to the SX-Key cable, and that the cable is securely connected to the PC's serial port.
- **Connection:** Make sure the components on the surface of the SX-Key are facing *up* in the same direction as the components on the XGS ME. Reversing this will result in the wrong pins being connected and could likely burn out both the chip and/or the console .
- **Setting:** When programming the XGS ME with the SX-Key *or* running a program with the SX-Key clock, set the **SYSMODE** switch to **KEY** mode. When programming the XGS ME with the parallel port, set it to **PGM** mode. When running *any* program without the SX-Key clock (for example, after programming via the parallel port), set it to **RUN** mode.

- **Connection:** When programming the XGS ME with the parallel cable, make sure the cable is firmly connected to the XGS ME parallel port, as well as the port on the PC.
- **Connection:** Make sure the two-lead RCA A/V cable is connected firmly between the XGS ME and television such that the one color connects the audio ports, and the other color connects the video ports. Remember that on the TV, the video port is yellow and the audio port is red or white (use either one). For the cleanest signal possible, ensure that the cable is laid out neatly, without kinks or bends, and is as far as possible from large sources of electro-magnetic interference (EMI).
- **Setting:** Verify that if the TV supports multiple A/V inputs, the correct one is selected.
- **Setting:** For most purposes, make sure the oscillator DIP switch is set for 80 MHz (as shown in Figure 5.7). Also remember that the 80 MHz setting is **required** for video generation.
- **Connection:** Make sure all socketed chips and oscillators are firmly connected and are properly oriented. See Figure 5.8 for an overview of the board showing the orientation of each socketed chip.
- **Setting:** Set the brightness and saturation potentiometers to whatever value looks cleanest on your TV. Also tune the TVs own video controls to match. There are no specific rules for this, as different TVs behave in different ways. Simply use trial and error to find your optional settings.
- **Setting:** Set the volume potentiometer to a reasonable volume. Also tune the TV's volume as appropriate.
- **Connection:** If using a keyboard, mouse or other PS/2 device, make sure the cable is firmly attached to the 6-Pin DIN PS/2 connector on the right-hand side of the board near the front.
- **Connection:** If using either joystick port, ensure that the joystick is firmly connected.
- **Connection:** Virtually all single-player games and demos use the left-hand joystick port. If you aren't sure which port to use, it's probably the left-hand port.
- **Connection:** Take note of the I/O extension headers next to the joystick ports. Make sure nothing is in physical contact with them unless specifically intended (such as other devices). Unintentional or bad connections with other devices may interfere with the ability to read the joystick ports themselves.
- **Connection:** Take note of the serial I/O expansion header on the lower-right hand side of the board. Make sure nothing is in physical contact with them unless specifically intended (such as other devices).
- **Setting:** Verify the power switch is set for **ON**.

### 5.4.2 - Colored Dots on the Oscillator Chips

As originally mentioned in Chapter 1, the sound and video oscillator chips *may* come with a colored dot to indicate their speeds. See Figure 5.8 for a visual reference. The sound oscillator may have a yellow dot indicating it runs at **5.736 MHz**, or “**5.736**” may actually be printed on it. The NTSC color burst crystal may have a blue dot or “**3.579545**” on it. The PAL color burst crystal may have a red dot or “**4.43**” on it. In all cases the colored dot is placed over the alignment dot, which allows you to insert the chip into its socket with the correct orientation.

#### WARNING!

Just a friendly reminder: Remember to *always* reset the XGS ME after programming, switch the **SYSMODE** switch, or any other major action or reconfiguration. Get in the habit of resetting before “expecting” something to work.

Figure 5.7 – Setting the oscillator DIP switch to 80 MHz.

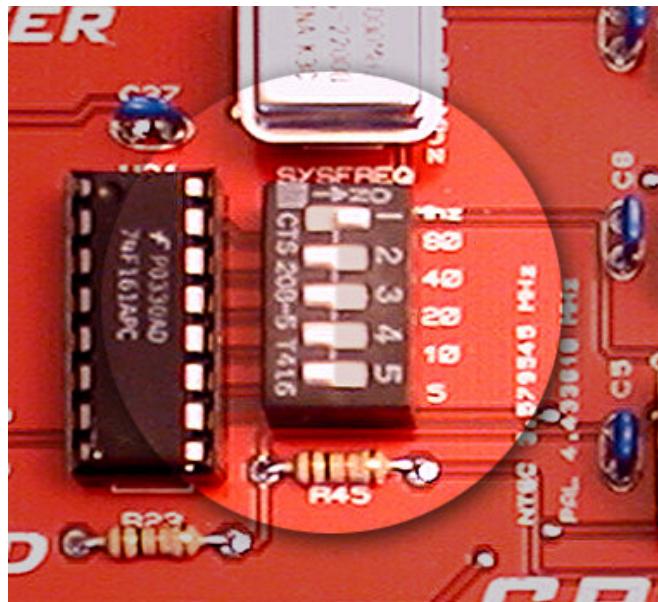
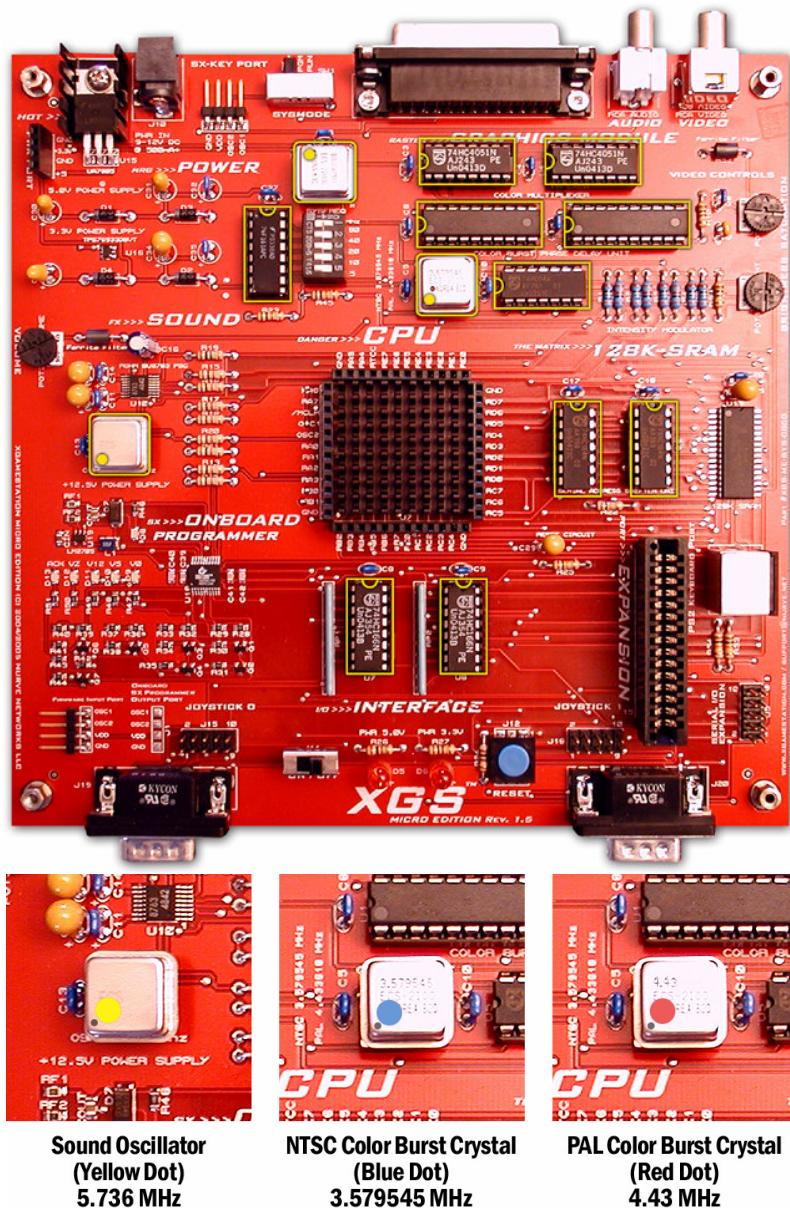


Figure 5.8 – An overview of the XGS ME board highlighting all socketed chip orientations.



## **Chapter 6: Guide to Included Demo Programs**

This chapter is a brief guide to the demo programs that are included with the XGS ME. Exploring and understanding these demos is a great first step towards developing your own programs. The next chapter, continuing on this path, discusses simple but effective ways in which the source code for these demos can be hacked to change their appearance and behavior.

The most important thing to do with this chapter is *actually load and run these programs!* Simply reading about them won't give you any hands-on experience with the software or the hardware. If you have not already done so, check out the quick start guides chapter 2, which provide easy-to-follow instructions for loading the XGS ME with a program.

Demos have been written for both NTSC and PAL televisions. This chapter is split into two sections, one for the NTSC demos, and the other for the PAL demos.

All demos are found on the **XGS ME Software CD**.

### **6.1 - NTSC Demos**

The following demos were written specifically for NTSC televisions.

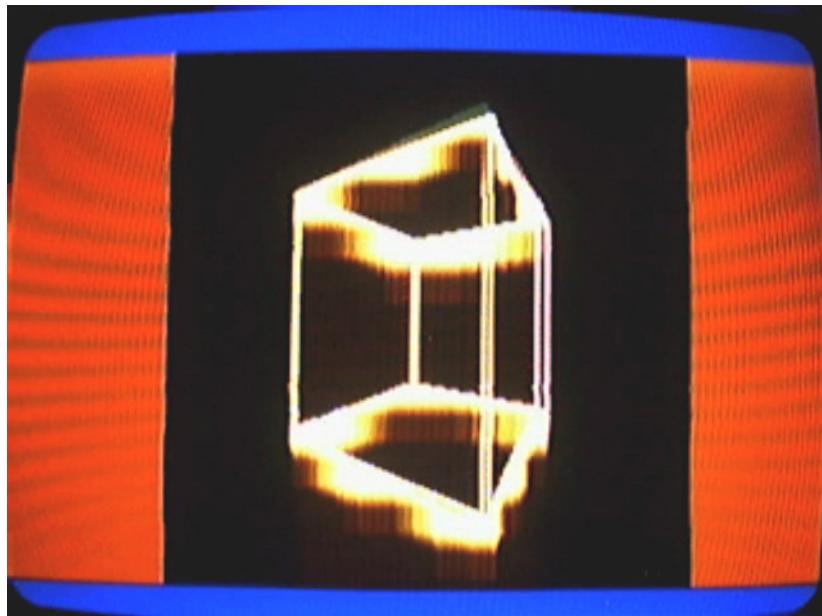
## 6.1.1 - Fire Cube

Author: **Kieren Johnstone**

Path: **Demos\NTSC\Fire\_Cube\kj\_3d\_01.src**

Renders a rotating, wireframe 3D cube and creates a downward, fire-like effect coming off each line. See Figure 6.1 for a screenshot.

**Figure 6.1 – The Fire Cube demo.**



## 6.1.2 - Flags

Author: **Michael Ollanketo**

Path: **Demos\NTSC\Flags\mic\_flags\_01.src**

Creates the illusion of a waving, three-dimensional flag by drawing a textured rectangle that is distorted along the Y-axis by a sine wave. To give the illusion of depth, another sine wave modulates the darkness of each pixel within the rectangle along the X-axis. A background pattern scrolls behind the flag, making transparent segments of the flag visible, thus allowing for non-rectangular flag designs. See Figure 6.2 for a screenshot.

**Figure 6.2 – The Flags demo.**



### 6.1.3 - Floormapper

Author: **Michael Ollanketo**

Path: **Demos\NTSC\Floormap\mic\_plane\_01.scr**

Demonstrates another common demo effect. This time, the effect is called ***floormapping*** and can be thought of as a rotozoomer in 3D. A texture or pattern is tiled infinitely over a plane viewed from a first-person perspective. In this particular demo, two such planes are mapped at once and joined in the center to form what appears to be the inside of a flattened cylinder. See Figure 6.3 for a screenshot.

**Figure 6.3 – The Floormapper demo.**



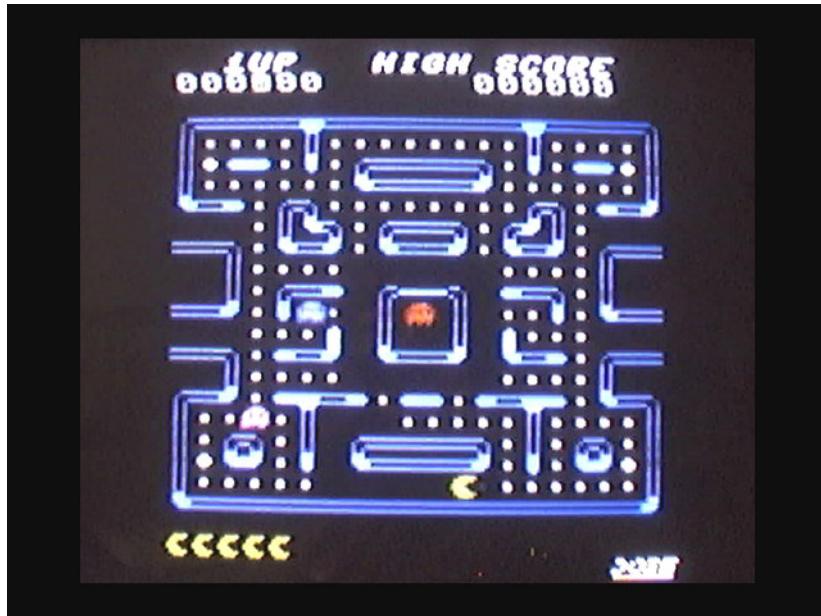
## 6.1.4 - Pac Man

Author: **Remi Veilleux**

Path: **Demos\NTSC\Pac\_Man\rem\_pac\_01.src**

A near-complete implementation of the classic arcade game. The demo is implemented with a tile system similar to the one seen in Tetris, but with the addition of four free-moving sprites overlaid on top for the characters. See Figure 6.4 for a screenshot.

**Figure 6.4 – The Pacman demo.**



## 6.1.5 - Plasma

Author: **Michael Ollanketo**

Path: **Demos\NTSC\Plasma\mic\_plasma\_01.src**

The plasma demo creates a warping, blobbing effect based on the intersection of multiple sine waves on perpendicular axes. A line of text is scrolled vertically on either side of the screen as well. See Figure 6.5 for a screenshot.

**Figure 6.5 – The Plasma demo.**



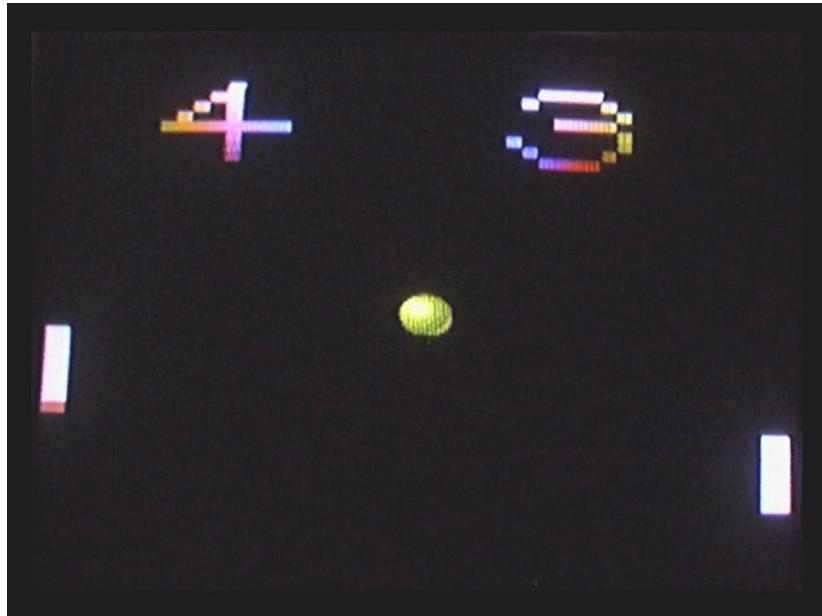
## 6.1.6 - Pong

Author: **Kieren Johnstone**

Path: **Demos\NTSC\Pong\kj\_pong\_01.src**

A simple take on the age-old classic. Bounce a virtual tennis ball towards your opponent and he'll bounce it back. See Figure 6.6 for a screenshot.

**Figure 6.6 – The Pong demo.**



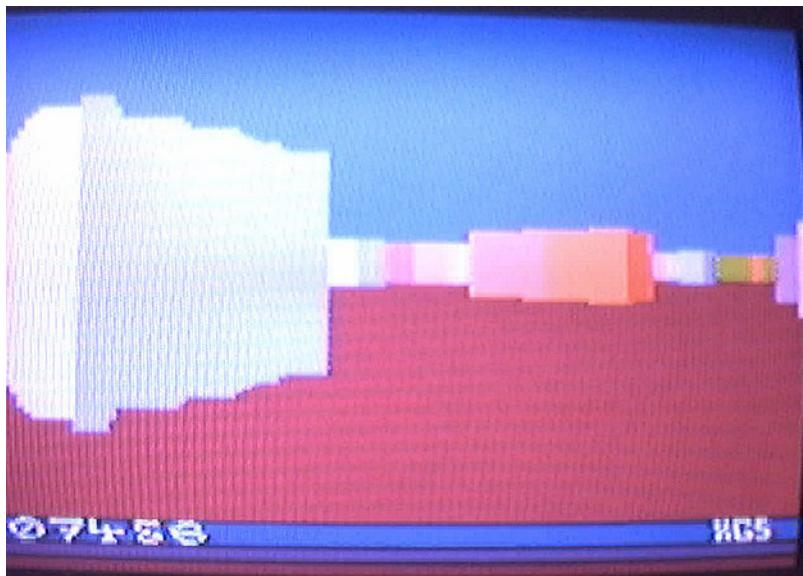
## 6.1.7 - Raycaster

Author: **Kieren Johnstone**

Path: **Demos\NTSC\Raycaster\kj\_ray\_01.src**

Demonstrates a technique for drawing 3D environments called ***raycasting***. Essentially, this technique allows a two-dimensional map to be “extruded” into a first-person perspective in which the user can navigate. See Figure 6.7 for a screenshot.

**Figure 6.7 – The Raycaster demo.**



## 6.1.8 - Rem Colors

Author: **Remi Veilleux**

Path: **Demos\NTSC\Rem\_Color\rem\_color\_01.src**

One of the first XGS ME demos. This demo warps a colorful plasma-like image using sine wave distortion patterns. See Figure 6.8 for a screenshot.

**Figure 6.8 – The Rem Colors demo.**



## 6.1.9 - Rotozoomer

Author: **Michael Ollanketo**

Path: **Demos\NTSC\Rotozoomer\mic\_rzoom\_01.src**

Demonstrates a classic demo effect in which a bitmap is tiled infinitely across the screen and rotated. See Figure 6.9 for a screenshot.

**Figure 6.9 – The Rotozoomer demo.**



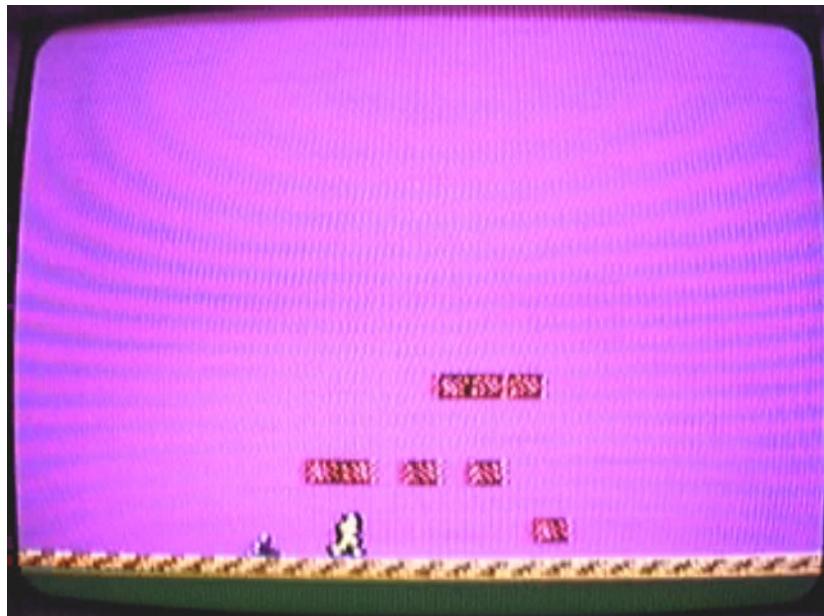
## 6.1.10 - Sprites

Author: **Kieren Johnstone**

Path: **Demos\NTSC\Sprites\kj\_sprites\_01.src**

Displays a user-controlled running character over a scrolling background in the style of classic side-scrolling console games. See Figure 6.10 for a screenshot.

**Figure 6.10 – The Sprites demo.**



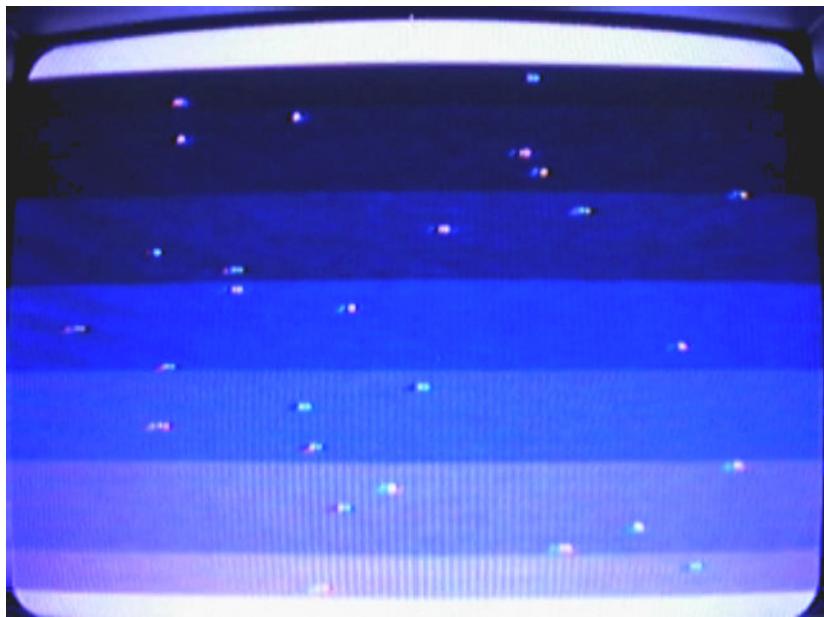
### 6.1.11 - Starfield

Author: **Alex Varanese**

Path: **Demos\NTSC\Starfield\starfield\_01.src**

Displays a field of stars at varying distances moving over a dark sky background. Comes with numerous variants, including different background colors and styles, as well as different foreground star styles. See Figure 6.11 for a screenshot.

**Figure 6.11 – The Starfield demo.**



### 6.1.12 - Tetris

Author: **Remi Veilleux**

Path: **Demos\NTSC\Tetris\rem\_tetris\_01.src**

A complete implementation of the classic block-drop puzzle game. The demo is implemented with a tile system in which the screen is drawn entirely based on a map of 8x8 pixel tile graphics through which blocks fall and the interface is rendered. See Figure 6.12 for a screenshot.

**Figure 6.12 – The Tetris demo.**



### 6.1.13 - Racer

Author: **Alex Varanese**

Path: **Demos\NTSC\Racer\racer\_01.scr**

A simple racing engine demo that allows the user to drive along a 3D perspective track and control when the direction turns left or right. The effect is completed with a background that scrolls depending on your turning speed and direction. See Figure 6.13 for a screenshot.

**Figure 6.13 – The Racer demo.**



## 6.2 - PAL Demos

The following demos were written specifically for PAL televisions.

### 6.2.1 - Flags

Author: **Michael Ollanketo**

Path: **Demos\PAL\Flags\mic\_flags\_pal\_01.src**

Creates the illusion of a waving, three-dimensional flag by drawing a textured rectangle that is distorted along the Y-axis by a sine wave. To give the illusion of depth, another sine wave modulates the darkness of each pixel within the rectangle along the X-axis. A background pattern scrolls behind the flag, making transparent segments of the flag visible, thus allowing for non-rectangular flag designs. See Figure 6.14 for a screenshot.

**Figure 6.14 – The Flags demo.**



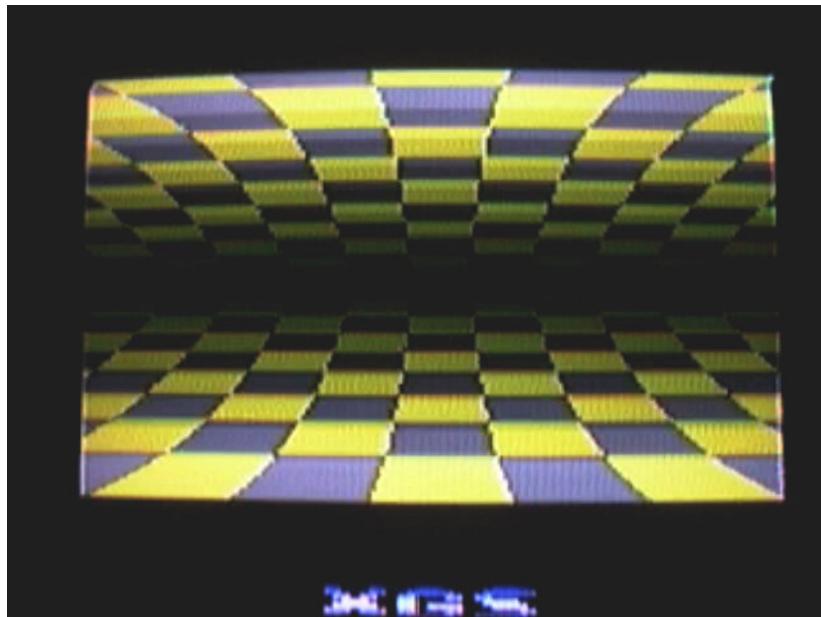
## 6.2.2 - Floormapper

Author: **Michael Ollanketo**

Path: **Demos\PAL\Floormap\mic\_plane\_pal\_01.src**

Demonstrates another common demo effect. This time, the effect is called ***floormapping*** and can be thought of as a rotozoomer in 3D. A texture or pattern is tiled infinitely over a plane viewed from a first-person perspective. In this particular demo, two such planes are mapped at once and joined in the center to form what appears to be the inside of a flattened cylinder. See Figure 6.15 for a screenshot.

**Figure 6.15 – The Floormapper demo.**



### 6.2.3 - Plasma

Author: **Michael Ollanketo**

Path: **Demos\PAL\Plasma\mic\_plasma\_pal\_01.src**

The plasma demo creates a warping, blobbing effect based on the intersection of multiple sine waves on perpendicular axes. A line of text is scrolled vertically on either side of the screen as well. See Figure 6.16 for a screenshot.

**Figure 6.16 – The Plasma demo.**



## 6.2.4 - Rotozoomer

Author: **Michael Ollanketo**

Path: **Demos\PAL\Rotozoomer\mic\_rzoom\_pal\_01.src**

Demonstrates a classic demo effect in which a bitmap is tiled infinitely across the screen and rotated. See Figure 6.17 for a screenshot.

**Figure 6.17 – The Rotozoomer demo.**



## **Chapter 7: Hacking the Demo Programs**

Before getting into the real nitty-gritties of developing XGameStation Micro Edition software, this chapter will provide an easy and fun introduction to XGS ME programming by walking you through a number of “hacks” that can be made to the demo programs.

All hacked demos can be found on the **XGS ME Software CD**.

### **7.1 - Fun For All Ages**

You do not need to know SX assembly language to follow these tutorials. The exact changes necessary to make are listed explicitly, so even users without any programming experience will be able to follow them. If you have not yet learned SX assembly, this chapter is a fun way to get a feel for what the language is like before you take the plunge.

In fact, the most important thing to remember throughout this chapter is that it *not* meant to teach you meaningful programming. The algorithms, data structures and techniques used in these demos will not be explained, aside from cursory rundowns where appropriate. Rather, the point of this chapter is to show what programmers and non-programmers alike can do by simply tweaking numeric values and data sets directly in the existing source code, then reassembling that code to see the results. By the end of this chapter, however, you will be more familiar with the look of SX52 assembly, and intermediate and advanced programmers will have probably picked up on how a lot of the language works simply by toying with it.

These hacks are broken up into two categories; NTSC-compatible and PAL-compatible.

Once you've read this chapter, you can move on to learning SX assembly language with the ***Beginning Assembly Language for the SX Microcontroller*** eBook, found in the **eBooks** directory on the **XGS ME Software CD**.

### **7.2 - NTSC-Compatible Hacks**

The following hacks can be applied to the NTSC-compatible demos.

#### **7.2.1 - Recoloring the Raycaster**

The raycaster demo, written by **Kieren Johnstone**, demonstrates a technique for drawing 3D environments called **raycasting**. Essentially, this technique allows a two-dimensional map to be “extruded” into a first-person perspective in which the user can navigate.

You can find the raycaster demo and its hacked variants here:

**Hacks\NTSC\Raycaster\raycaster\_01.src**.

The raycaster demo (depicted in Figure 7.1) is our first example in basic code hacking. Our first impulse might be doing something really cool like changing the world map to create new environments to explore. While this is definitely an interesting idea, the XGS ME raycaster demo stores its map data in a compressed format that is expanded into the SRAM at runtime. Since the very first hacking example would probably be a bit too advanced if it involved reverse-engineering the compression algorithm applied to an unknown dataset, we're going to set our sights a little lower.

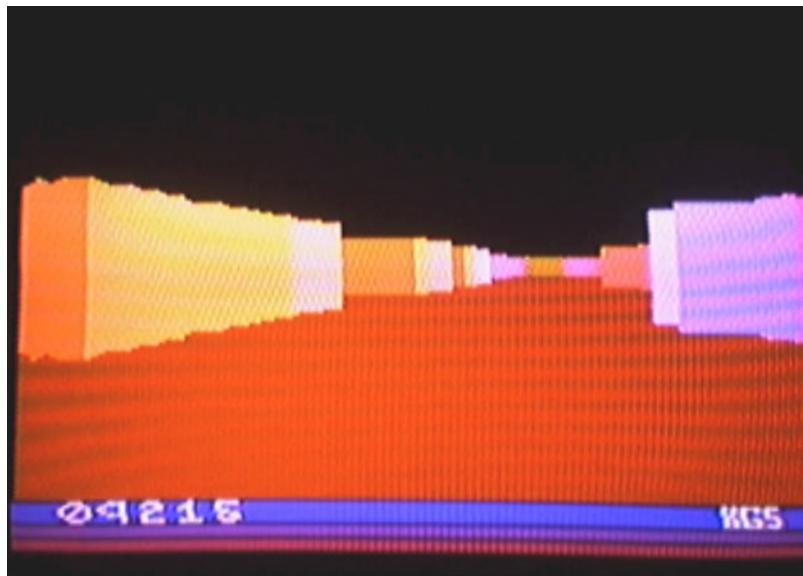
Figure 7.1 – The raycaster demo before hacking.



This hack will concern itself with nothing more than changing the floor and ceiling colors. Fortunately for us, this is all too easy as the floor and ceiling colors are stored in aptly named constants. Take the following steps to perform this simple hack:

- Save **raycaster\_01.src** as **raycaster\_hack/src**.
- Use the editor to perform a search for the text string “floor”, as if you were trying to find some mention of the floor to determine how and where its color is set.
- The first match should be on line 34, where the constant **FLOOR** is defined. Jackpot! This is the floor color. As an added bonus, the ceiling color is found on the next line, stored in the constant **CEILING**.
- Change the value of **FLOOR** to **(COLOR3 + 1)** (a deep red), and the value of **CEILING** to **(COLOR15 + 3)** (a dark brown).
- Reassemble the program and write it to the XGS ME. Notice the new colors? Figure 7.2 illustrates them.

Figure 7.2 – The new, hacked floor and ceiling colors in the raycaster.



#### 7.2.1.1 - Recommended Hacks

To hack this program further, try identifying which constants refer to other colors you see onscreen (such as the two bars at the bottom, or the color of the text). Can you change them as easily? If not, what effect did your changes to the code have?

#### 7.2.2 - Hacking the Plasma

The plasma demo, written by **Michael Ollanketo**, creates a warping, blobbing effect based on the intersection of multiple sine waves on perpendicular axes. Lines of text are scrolled vertically on either side of the screen as well. Figure 7.3 is a screenshot of the plasma demo.

Figure 7.3 – The plasma demo.



You can find the plasma demo and its hacked variants here: [Hacks\NTSC\Plasma\plasma\\_01.src](#).

This hack will actually involve modifying functional code, albeit in a very simple way. One quick and easy way to modify this effect is to alter the rate at which these sine waves are traversed, thus compressing or expanding the waves and blobs on the corresponding axis.

This hack will compress the plasma effect by a factor of three on the Y axis. Take the following steps to implement it:

- Save **plasma\_01.src** as **plasma\_hack.src**.
- Locate line 197 (**dec vertPtr1**).
- Copy and paste the line twice below it (*without* overwriting anything), turning the one **dec** instruction into three. Lines 197, 198 and 199 should now all be **dec vertPtr1**.
- Reassemble the program and write it to the XGS ME. Notice the compressed appearance of the plasma? Check it out in Figure 7.4.

Figure 7.4 – The plasma effect, compressed down to a third of its original height.



The effect works by speeding up the rate at which the vertical sine wave is traversed. This wave is calculated beforehand and stored in a lookup table. The index into this table is `vertPtr1`, which we're now moving through the table three times faster (with three consecutive decrement instructions instead of one). If this doesn't make sense to you, however, don't worry—this section is simply intended to show you what can be done with simple modifications to source code, not to teach the original effect itself.

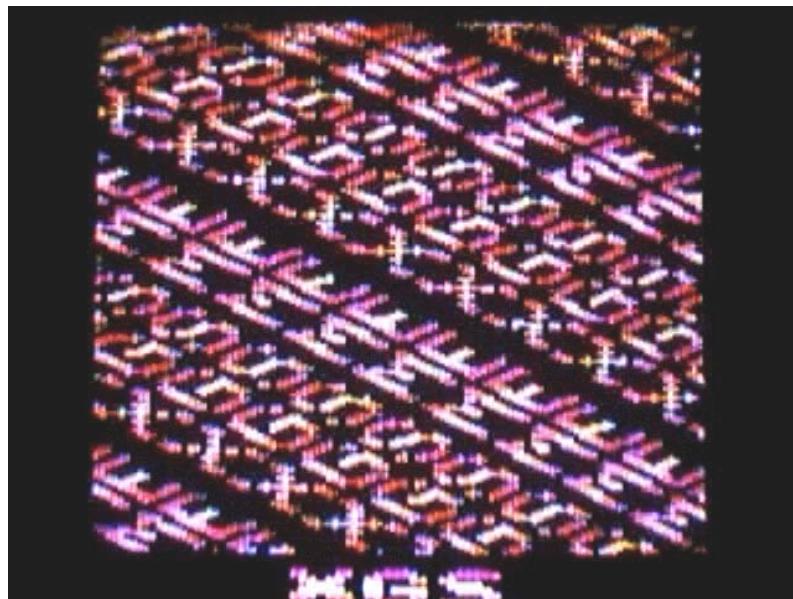
### 7.2.2.1 - Recommended Hacks

To expand the complexity of this hack, try applying this technique to one of the other sine wave table indices. The trick is first finding which variables they are! Once you know which variables are used as indices into these tables, you can try to find out where in the code they're being incremented or decremented, and modify that code to alter their value by a different amount.

### 7.2.3 - Altering the Rotozoomer Bitmap

The rotozoomer demo, written by **Michael Ollanketo**, demonstrates a classic demo effect in which a bitmap is tiled infinitely across the screen and rotated. Figure 7.5 is a screenshot of the rotozoomer demo.

Figure 7.5 – The rotzoomer demo.



You can find the rotzoomer demo and its hacked variants here:

[Hacks\NTSC\Rotozoomer\rotzoomer\\_01.src](#).

In this hack, we're going to alter the bitmap by changing its representation in the **xgsme.src** include file. This file is entirely dedicated to defining the rotzoomer's bitmap in program memory through a long series of **DW** directives. The directives, in order, define each pixel in the bitmap from left to right, top to bottom.

### 7.2.3.1 - A Color Change

While it would be difficult to change the actual shape of the bitmap due to the unintuitive layout of the source code, it's easy to change the colors, so let's start there. As is, the bitmap consists of the text "XGS ME" in gradient-filled letters over a black background. By making the following changes with your code editor's find and replace function, the new bitmap will be black text over a bright pink background:

- Change all occurrences of **BLACK** to **COLOR9+8**.
- Change all occurrences of **COLOR7+4** to **BLACK**.
- Change all occurrences of **COLOR7+5** to **BLACK**.
- Change all occurrences of **COLOR7+6** to **BLACK**.
- Change all occurrences of **COLOR7+7** to **BLACK**.

- Change all occurrences of **COLOR8+4** to **BLACK**.
- Change all occurrences of **COLOR8+5** to **BLACK**.
- Change all occurrences of **COLOR8+6** to **BLACK**.

### 7.2.3.2 - Reassembling

To see your changes in action, perform the following steps:

1. Save the changes to a new file called **xgsme\_hack.src**.
2. Open **rotozoomer\_01.src**.
3. On line 521, change the **include** directive filename to **xgsme\_hack.src**.
4. Reassemble and run the program.

If all went well, the rotating image should now be black on bright pink, like in Figure 7.6.

Figure 7.6 – Hacking the bitmap colors of the Rotozoomer demo.



### 7.2.3.3 - Troubleshooting

- The first and most obvious problem you may have is a syntax error that stems from incorrectly replacing the various color values.

- Make sure to perform the color changes in the order listed! If you change one or more of the colors to black, then change all occurrences of black to another color, you'll get a totally different (and possibly invisible) result!
- If you aren't seeing a difference, make sure you've opened all of the right files in the right directory; sometimes it's easy to open the main file from one directory and the include file from another directory and not realize they're two separate copies of the same program.

#### 7.2.3.4 - Recommended Hack

Knowing that the dimensions of the bitmap are 32x8, try reformatting the organization of the **xgsme\_hack.src** bitmap so it can be edited by hand more easily. Check out the PAL version of this hack in the next section to see an example of turning the rotzoomer bitmap into something more game-like. The techniques used there apply to this demo as well.

### 7.2.4 - Changing the Sprite Demo Bitmaps

The sprite demo, written by **Kieren Johnstone**, displays a user-controlled running character over a scrolling background in the style of classic side-scrolling console games. Figure 7.7 is a screenshot of the sprite demo.

Figure 7.7 – The sprite demo.



You can find the sprite demo and its hacked variants here: **Hacks\NTSC\Sprites\sprites\_01.src**.

The bitmaps for these sprites, luckily for us, are stored in an easily-hackable format that we can take advantage of to write our own graphics into the demo. For this hack, we're going to try to create new sprite designs. First, save **sprites\_01.src** as **sprites\_hack.src**. Next, locate the sprite tables found near the end of the file. The sprites we'll be hacking start on line 1236.

### 7.2.4.1 - The Sprite Format

This demo utilizes 2-color (1 bit per pixel) sprites; one color represents a solid pixel, the other “color” is transparent. Within the sprite table, each bit corresponds to one pixel. Since these sprites are 8x8 pixels, 8 bytes of 8 bits each will store the complete sprite. Furthermore, using binary numbers lets us visually see each pixel as a zero or one, similar to the way it will appear at runtime.

### 7.2.4.2 - Hacking the Sprites

Now that we understand how the sprites are stored, we can change them. The first and most dramatic change we'll make will be done to the girder sprite, which is currently a bit too sparse for my tastes. The replacement will turn the girder into a more solid brick tile, suitable for any Super Mario Bros. ripoff game.

On line 1268, the **Girder** sprite is defined. Replace that definition with the following:

```
Girder
dw      %11111011
dw      %11111011
dw      %00000000
dw      %10111111
dw      %10111111
dw      %00000000
dw      %11110111
dw      %11110111
```

If you were to reassemble and run the program now, you'd see an immediate change.

Next, we're going to change the power-ups from a heart and a ring-shaped coin to a “1up” and a sword.

On line 1246, the **Coin** sprite is defined. Replace that definition with the following to get a “1up” symbol:

```
Coin           ; POWERUP_COIN
dw      $B2F          ; Colour = bright yellow
dw      %01011011      ; Bitmap data
dw      %11011011
dw      %01011010
dw      %01011010
dw      %01000000
dw      %01000000
dw      %01000000
dw      %11100000
```

Next, go to line 1257 and replace the definition for **Health** with the following to get a cool sword:

```
Health        ; POWERUP_HEART
```

```

dw    $B9D          ; Colour = red
dw    %11000000      ; Bitmap data
dw    %10100000
dw    %01010000
dw    %00101010
dw    %00010100
dw    %00001010
dw    %00001110
dw    %00010111
dw    %00000011

```

Lastly, the spike, defined on line 1236, is going to be replaced with something a bit more interesting. This particular sprite is a bit hard to figure out, so it can be anything from a distant house or hut, to a shrine-type something-or-other. Maybe it's Boba Fett's helmet. In any case it looks kinda cool:

```

Spike
dw    %00111100
dw    %01111110
dw    %11111111
dw    %00000000
dw    %10100101
dw    %11100111
dw    %11100111
dw    %10100101

```

Now, with all of the sprites updated to new graphics and colors, reassemble and run the demo. Pretty cool huh? A significant change in appearance has resulted from only simple changes to some easy-to-read sprite tables. Check out Figure 7.8 to see it.

**Figure 7.8 – The sprite demo updated with hacked graphics.**



### 7.2.4.3 - Troubleshooting

Numerous definitions of memory blocks were changed here, and if you're having trouble assembling your hacked version, it's probable that somewhere in the definitions a declaration got screwed up. Make sure none of the original symbols are missing, that each `DW` is on its own line, and so on.

### 7.2.4.4 - Recommended Hack

The player sprite is a cool rotoscoped animation of a running man. Try redrawing each frame of animation to create something different, like a rolling tank, spinning wheel, or something else.

## 7.2.5 - Reshaping the Racer Mountain Range

The racer demo, written by **Alex Varanese**, produces an effect seen in many early racing games in which the track is composed of a vertical "stack" of horizontal black lines that are compressed and distorted with sine waves to create the illusion of perspective and curves. Figure 7.9 is a screenshot of the racer demo.

Figure 7.9 – The racer demo.



You can find the racer demo and its hacked variants here: [Hacks\NTSC\Racer\racer\\_01.src](#).

As is explained in more detail in the **Case Study: Racer Engine Demo** chapter, the racer engine displays its mountain range background using a height map. In other words, a map of height values that

determine, for each pixel along the mountain range's X-axis, how tall the mountain is at that point. When these heights are drawn next to one another, they come together to form a smooth, mountain-like solid shape.

### 7.2.5.1 - Square Wave Mountains

The height map defined starting on line 1413. To illustrate exactly how the hacking of the mountain range's shape works, save **racer\_01.src** as **racer\_hack.src** and replace lines 1413 to 1421 with the following:

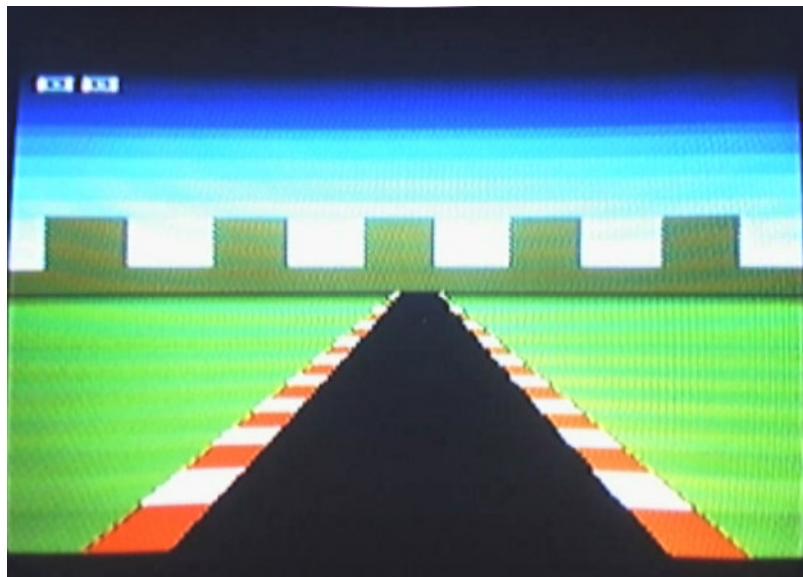
```
mount_line
    DW      10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10
    DW      30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30
    DW      10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10
    DW      30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30,30
```

Of course, in reality you probably aren't masochistic enough to type that all in manually, so it's easier to check out this hack by opening the included copy of **racer\_hack.src**. The key, however, is that instead of the gradually rising and falling values of the original height map, the new values alternate between 10 or 30, each being repeated 16 times. This will change the shape of the mountain range to a perfect square wave. The range will be exactly 10 pixels tall for 16 pixels, then exactly 30 pixels tall for the next 16 pixels, and so on. Reassemble and run the program to see the effect, or check out Figure 7.10.

#### NOTE

All height map code listings in this section have been condensed horizontally to fit the page. In the actual source code, the height map definition is formatted twice as wide and half as tall, but the data itself is identical.

Figure 7.10 – The racing mountain ranged hacked into an unnatural square wave.



With a better understanding of how the values in the height map correspond to the onscreen graphics, try putting in new values and noting their results. Find out how high you can make the mountains reach before something goes wrong, for example. Be creative! Hacking source code like this is all about letting your imagination run wild and figuring out ways to use data structures and code to do new things it wasn't necessarily intended for.

#### 7.2.5.2 - The Racer City Hack

Messing with the mountain range alone is fun, and I'm sure there are a lot of cool tricks you can pull, but when you throw in some color changes as well, you can create an entirely new type of background. This hack performs a few changes to the program colors, as well as cleverly reshaping the mountain range into a more rigid pattern that resembles a city skyline.

The following is the new city skyline height map, which is to be directly pasted over the original mountain range as in the previous section:

```

mount_line
DW    10,10,10,10,20,30,20,20,15,15,35,50,35,35,40,40
DW    20,20,20,40,40,39,39,20,20,50,50,50,40,40,50,40
DW    10,10,30,30,40,40,40,30,30,10,35,35,35,45,35,35
DW    50,50,50,49,48,47,46,30,30,40,40,55,40,20,20,30

```

Even by reading this code dump it should be easy to see the difference in the two shapes. The formerly smooth and flowing mountain range is now held at the same height exactly for varying lengths (creating rooftops), after which it sharply juts up or down to a new level (creating sides, or walls). The occasional single-pixel height jump is also included to create the illusion of thin antennae reaching from the buildings.

But as is, this will simply create a brown, boxy shape over the same blue sky we've already seen. It's not a very convincing city skyline due to the weird color choices. To really complete the city image, the skyline should be a dark grey and the sky gradient should become something more dramatic, like perhaps a deep red evening color. The following steps allow these changes to be made as well:

- On line 70, change (`COLOR11 - 1`) to (`COLOR8 - 1`). This changes the hues available to the sky and mountain background area, allowing it to display more reds, oranges and yellows.
- On line 72, change (`COLOR0`) to (`COLOR14`). This will set the sky color to a striking red.
- On line 73, change (`COLOR13 + 4`) to (`BLACK + 4`). This will set the former mountain range to a dark grey, much more appropriate for its new city skyline shape.

That's it! Open up the included **racer\_city.sfc** to check out this hack, and take a look at Figure 7.11 for a screenshot.

Figure 7.11 – The racing demo’s “city hack”.



#### 7.2.5.3 - Recommended Hacks

Pretty incredible, huh? With only a few color changes and the simple remapping of the mountain range height map, an array of dramatically differing backgrounds and effects can be achieved. Try creating new backgrounds this way. For example, having already seen brown mountains on a blue sky background, and the city skyline at sunset, perhaps a snowy arctic setting is in order. Create a new set of mountains, perhaps taller and more jagged than the originals, and make them white. Then set the sky to a slightly purplish-blue to set it apart from the others.

#### 7.2.6 - Reshaping the Tetris Blocks

The Tetris demo, written by **Remi Veilleux**, is a complete implementation of the classic block-drop puzzle game. The demo is implemented with a tile system in which the screen is drawn entirely based on a map of 8x8 pixel tile graphics in which blocks fall and the interface is rendered. Figure 7.12 is a screenshot of the Tetris demo.

Figure 7.12 – The Tetris demo.



You can find the Tetris demo and its hacked variants here: [Hacks\NTSC\Tetris\tetris\\_01.src](#).

The Tetris demo is just begging to be hacked. Not only is the tile graphic used to draw the blocks open to our meddling, but so are the blocks themselves! By changing the block shapes and their rotated versions, you can literally change the gameplay of Tetris in dramatic ways. Not bad considering you won't have to write a single line of functional code.

### 7.2.6.1 - Hacking the Block Tile

Begin by resaving `tetris_01.src` as `tetris_hack.src`.

The block tile graphic is the graphic used to draw the individual segments of each Tetris piece. Changing this to something other than the “square inside a square” design it's initially set to can create some pretty striking effects.

The Tetris game is designed around a tile system that can display 8x8 tile graphics on a screen-sized tile map. The trick to changing the block tile is finding its definition in the array of tile graphics starting on line 1382. On line 1432 you'll find the following tile definition:

```
dw 255,129,189,189,189,129,255 ; tile 50 (address: 400)
```

Notice there are 8 decimal values here. Since the tiles are 8x8 1-bit pixels, each of these 8-bit values represent one of the 8 pixel rows of the tile. The only problem is, since they're written in decimal instead of binary and listed horizontally instead of vertically, they aren't very easy to edit.

Fortunately for us, all we need to do is rewrite the definition ourselves using a more readable format. Remember—8 consecutive **DW** directives is the same thing as one **DW** directive with 8 values. So, by rewriting these values using binary and breaking up the 8 values into separate lines, we can visualize the tile and draw any pattern we want with ease. Replace the single-line declaration currently at line 1432 with the following multi-line declaration. Take care not to overwrite any of the lines above or below the one we're replacing:

```
DW %00111100      ; Happy Face
DW %01111110
DW %11011011
DW %11111111
DW %10111101
DW %11000011
DW %01111110
DW %00111100
```

Reassemble and run. Pretty bizarre, huh? The formerly mild-mannered puzzle game is now a sadistic ritual in which grinning, disembodied heads are stacked to form blasphemous structures and patterns. Check out Figure 7.13.

Figure 7.13 – Replacing the standard Tetris block tile with a happy face.



With our easy-to-edit binary representation of the tile, you can hack any design you'd like into the block tile. Also in the included version of **tetris\_hack.src** are two other shapes to try out. The first is a brick pattern that gives the game a much more heavy-duty look:

```
DW %10111111 ; Brick
DW %10111111
DW %00000000
DW %11110111
DW %11110111
DW %00000000
DW %10111111
DW %10111111
```

Last is a pattern I call "hologram", since it creates a very hollow, 80's art-deco look:

```
DW %11111111 ; Hologram
DW %10000000
```

Just remember not to include more than one of these tile definitions in the code at the same time, as it will disrupt the structure of the rest of the tile table. If you have more than one, make sure all but one are commented out.

### 7.2.6.2 - Hacking the Tetris Pieces

#### WARNING!

The hack described in the last section changed the line numbering within the file and as such, all line number references from here on will be wrong unless your copy of **tetris.src** has been hacked in the manner described above.

Changing the block tile is fun, but hacking the pieces themselves can really turn Tetris upside down. Not only do such hacks bring about visual changes, but they can drastically affect the gameplay mechanics, for better or worse.

The shape of each piece is stored in a format that isn't much more readable than the tiles. Fortunately, in the comments above each definition is an ASCII-art rendition of what the piece looks like within its 4x4 grid. Also of note is that the pieces are not actually rotated by the Tetris code itself; all four rotations of each piece are declared manually. As we'll see later, this opens up some interesting possibilities.

The second Tetris piece definition starts on line 1470 and looks like this:

```
; ....
; .xx.
; xx..
; ....
```

```

dw %0000_0000_0110, %0000_1100_0000
; X...
; XX...
; .X...
; ...
dw %0000_1000_1100, %0000_0100_0000
; ...
; .XX...
; XX...
; ...
dw %0000_0000_0110, %0000_1100_0000
; X...
; XX...
; .X...
; ...
dw %0000_1000_1100, %0000_0100_0000

```

The comments make it easy to see which variant of which piece is being defined. Each piece is defined by four 4-bit nibbles, creating a 1-bit, 4x4 grid. Since each program word on the SX52 is 12-bits wide, the lower 8-bits of two consecutive program words are used.

Read the following very carefully to understand how each piece is encoded: In each **DW** directive, the first nibble of the first word is unused, the second nibble of the first word is the first grid row, and the third nibble of the first word is the second grid row. The first nibble of the second word is unused, the second nibble of the second word is the third grid row, and the third nibble of the second word is the fourth grid row. Phew! It's also fairly easy to just figure out the format visually, but in either case, actually encoding the piece by hand is a bit of a pain.

Let's hack the first piece, the square, the definition of which starts on line 1477. For starters, we're going to hack in a very imposing, lumbering piece I like to call the "Disruptor", which is like the standard vertical line piece if it had a second, perpendicular line attached to one of its ends:

```

; XXXX
; X...
; X...
; X...
dw %0000_1111_1000, %0000_1000_1000
; X...
; X...
; X...
; X...
; XXXX
dw %0000_1000_1000, %0000_1000_1111
; ...X
; ...X
; ...X
; ...X
; XXXX
dw %0000_0001_0001, %0000_0001_1111
; XXXX
; ...X
; ...X
; ...X
dw %0000_1111_0001, %0000_0001_0001

```

Remember, this hack only works if you directly replace the old piece with this one. Once pasted in, take a moment to study how the ASCII-art pattern of each piece was transcribed into the actual binary data used by the game. Check out Figure 7.14 for a screenshot of this new piece in action.

Figure 7.14 – The large “Disruptor” piece making a bold entrance in the hacked Tetris.



The included version of `tetris_hack.src` hacks the next two pieces as well, creating an interesting piece that is actually two separate parts, as well as a single-block piece that can come in handy in some of Tetris's tighter moments. As you hack pieces yourself and play with them, take note of how different the game can feel, and how the usual Tetris-playing strategies can be either enhanced or rendered completely useless.

#### 7.2.6.3 - Recommended Hacks

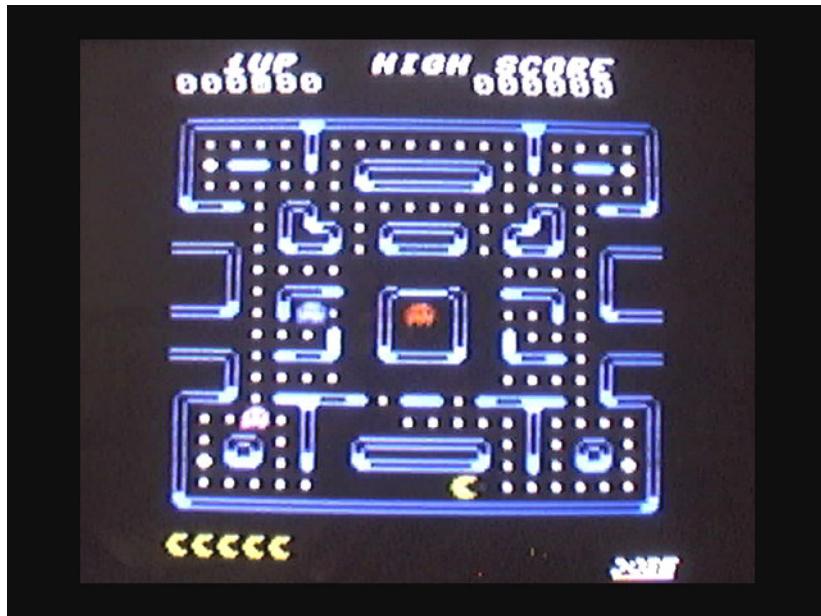
The fact that each rotated variant of each piece is encoded as separate data affords us some interesting opportunities. Because the “rotation” of the pieces is really just the result of the way each variant is designed, you can actually change the rules of Tetris entirely by “designing” behavior into the variants other than rotation. For example, imagine a piece that expands and contracts instead of rotating, allowing it to squeeze through small holes near the top of the pile, then expand to fill out open spaces near the bottom.

Another interesting (and weird) idea would be to change every variant of every piece into totally random, disjointed data. Try and see how far you can get when the falling pieces follow no rhyme or reason no matter what you do!

### 7.2.7 - Hacking Pac-Man

The Pac Man demo, written by **Remi Veilleux**, is a near-complete implementation of the classic arcade game. The demo is implemented with a tile system similar to the one seen in Tetris, but with the addition of four free-moving sprites overlaid on top for the characters. Figure 7.15 is a screenshot of the Pac Man demo.

Figure 7.15 – The Pac Man demo.



You can find the Pac Man demo and its hacked variants here: [Hacks\NTSC\Pac\\_Man\rem\\_pac\\_01.src](#).

This section will present the most complex hack yet; turning Pac Man into a reasonable approximation of Ms. Pac Man (graphically) using nothing more than color and sprite table changes. Hacking Pac Man into Ms Pac Man actually isn't that complicated, but it involves multiple steps:

- Redrawing the Pac Man sprites to include a crude “bow”, like Ms. Pac Man
- Changing the colors of Pac Man and the ghosts to lighter, pastel versions
- Changing the color of the level borders from Pac Man's blue to Ms Pac Man's pink.

If you haven't already, please check out the previous section on hacking Tetris. These two demos were written by the same author (Remi Veilleux), and as such the code presented here will seem much more familiar to you.

Begin this hack by saving **rem\_pac\_01.src** as **ms\_pac\_man.src**. All changes will be made to this file.

### 7.2.7.1 - Redrawing Pac Man as Ms. Pac Man

In the real arcade game, Ms. Pac Man looks like the original yellow Pac Man, except pink lipstick, a pink bow, black eyelashes and a beauty mark have been added. The XGS ME implementation of Pac Man uses 1-bit sprites, which means multiple colors within the same sprite are not possible—unfortunately precluding the pink lips and bow on a yellow body we would like. Aside from lightening up the yellow color to a more pastel shade in the next section, we'll have to rely on drawn-in detail to bring Ms. Pac Man to life.

The tiles and sprites used in this demo are stored in the same format used in the Tetris demo, which means they're difficult to edit by hand. Fortunately, the hard work is already done. The sprite definitions for Pac Man begin on line 1670. Replace lines 1670-1674 with the following block of code:

```
; dw 060,126,126,255,255,126,126,060 ; tile 52 (address: 416)
DW      %00111100          ; Neutral
DW      %01110110
DW      %11100111
DW      %11111001
DW      %11111011
DW      %11111111
DW      %01111110
DW      %00111100

; dw 062,124,112,224,240,120,126,060 ; tile 53 (address: 424)
DW      %00111100          ; Right, Mouth Open / Icon
DW      %01110110
DW      %11100100
DW      %11111000
DW      %11111000
DW      %11111100
DW      %01111110
DW      %00111100

; dw 060,126,126,255,247,231,195,066 ; tile 54 (address: 432)
DW      %00111100          ; Down, Mouth Open
DW      %01110110
DW      %11100111
DW      %11111001
DW      %11111011
DW      %11100111
DW      %01000010
DW      %00000000

; dw 060,126,030,015,007,014,062,124 ; tile 55 (address: 440)
DW      %00111100          ; Left, Mouth Open
DW      %01110110
DW      %00100111
DW      %00011001
DW      %00011011
DW      %00111111
DW      %01111110
DW      %00111100
```

```
; dw 066,195,231,247,255,126,126,060 ; tile 56 (address: 448)
DW      $00000000      ; Up, Mouth Open
DW      $01000010
DW      $11100111
DW      $11111111
DW      $11111111
DW      $11111111
DW      $01111110
DW      $00111100
```

The original definitions have been left in the form of comments to make it easier to cross reference these changes with the original code. Just make sure they stay comments, as the order of the data will be corrupted otherwise and the program will not even assemble.

These new sprites redraw the Pac Man character slightly to make room for a bow, which is drawn about as well as a 1-bit, 8x8 sprite could hope for. ☺

### 7.2.7.2 - Changing the Pac Man and Ghost Colors

Ms. Pac Man and the ghosts will be altered next, to lighter shades of their normal colors. Fortunately, this is an easy step—the colors used for the characters are only repeated in one or two places each, making them easy to change.

Ms. Pac Man is always drawn using the same color, which means this color value can be hard coded and does not need to be stored in a variable. The only other place the Ms Pac Man color appears is when drawing the icons that represent how many lives are left. The ghosts, on the other hand, all turn blue when a power pill is eaten. Because of this, the current ghost colors are stored in variables. The colors are set once and restored later, which means the changes must be made in two places.

The following lists each step that must be taken to change the character colors to lighter, pastel versions:

- On line 869, change **#9** to **#12**. This lightens up the Ms. Pac Man character.
- On line 1741 (or 1790 in **ms\_pac\_man.src** after applying the previous changes), change **#9** to **#12**. This changes the Ms Pac Man icons used to represent the number of lives left.
- On line 530, change **#87** to **#91**. On line 534, change **#168** to **#172**. On line 538, change **#231** to **#235**. This makes each ghost 4 shades brighter, thus making their appearance more pastel.
- After the ghosts return from their blue-colored panic state, they must resume their normal colors. This step ensures they return to the new pastels. On line 726, change **#87** to **#91**. On line 727, change **#168** to **#172**. On line 728, change **#231** to **#235**.

### 7.2.7.3 - Changing the Level Border Color

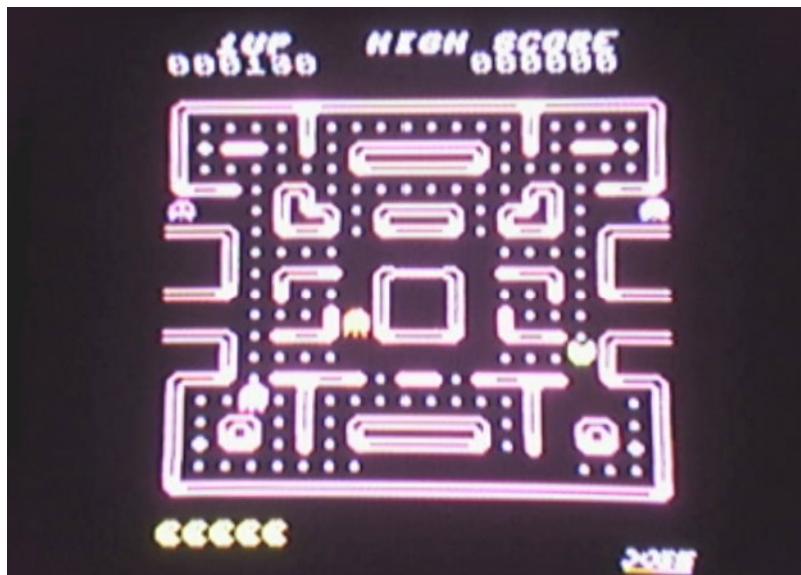
The characters are now lighter, which means the last step is to change the color of the level borders themselves. In Pac Man, these borders are an almost neon blue shade. In Ms Pac Man, they turn pink.

One caveat is that the XGS ME version of Pac Man changes the color of these borders each time a new level is reached. Since this is only a superficial hack, we aren't going to worry about these color changes and instead will only focus on changing the color of the first level.

The first level color is set in three places. On lines 192, 505 and 736, change `#231` to `#172`.

That's it! Reassemble and run the program and the result should be a fairly convincing rendition of Ms. Pac Man, which isn't bad considering that once again, no functional code was changed or added. If your version of the code doesn't look right, or you're simply too lazy to make these changes yourself, check out the included version of `ms_pac_man.src`, and take a look at the screenshot in Figure 7.16.

Figure 7.16 – Pac Man hacked into a rendition of Ms Pac Man.



## 7.3 - PAL-Compatible Hacks

The following hacks can be applied to the PAL-compatible demos.

### 7.3.1 - Hacking the Plasma Text

The plasma demo, written by **Michael Ollanketo**, creates a warping, blobbing effect based on the intersection of multiple sine waves on perpendicular axes. A line of text is scrolled vertically on either side of the screen as well. Figure 7.17 is a screenshot of the plasma demo.

Figure 7.17 – The plasma demo.



You can find the PAL plasma demo and its hacked variants here: [Hacks\PAL\Plasma\plasma\\_01.src](#).

The plasma effect itself may be the most important part of the plasma demo, but another interesting aspect is the scrolling text found on either side of the screen. This text is drawn using a 5x7 pixel font, and read from a 32-character text string.

### 7.3.1.1 - Decoding the String

This text string is defined at the end of the **plasma\_01.src** source file, on line 488:

```
dw      $10
dw      $01
dw      $0C
dw      $00
dw      $10
dw      $0C
dw      $01
dw      $13
dw      $0D
dw      $01
dw      $00
dw      $06
dw      $0F
dw      $12
dw      $00
dw      $14
dw      $08
dw      $05
```

```

dw    $00
dw    $18
dw    $07
dw    $13
dw    $1B
dw    $0D
dw    $05
dw    $00
dw    $00
dw    $00
dw    $1C
dw    $00
dw    $00
dw    $00

```

Since it's not likely that an XGS ME demo adheres to the ASCII format, we shouldn't assume that this string is encoded in any way we would recognize. Since we can't be sure how the font characters map to the numeric values seen in this string, we'll have to figure it out by looking for a pattern since we can figure out what the string says by running the demo.

Fortunately for us, we don't even have to go this far. Line 477 contains a comment explaining that character 0 is a space, and that characters 1-26 are the capital letters. What isn't mentioned in this comment, but can be determined using the method described above, is that character 27 is a hyphen and character 28 is a bullet point.

As a first step, resave **plasma\_01.src** as **plasma\_hack.src**.

### 7.3.1.2 - Hacking the Text

So we've got capital letters, a space, a hyphen and a bullet point to work with. We've also got 32 total characters to fill with our hacked message. The next step is to write the message we want and figure out what each corresponding character code will be. Since A is 1, B is 2, C is 3 and so on, this isn't going to be particularly difficult. I've encoded the message "**THIS TEXT HAS BEEN HACKED**", followed by three bullet points, in the following replacement for the existing string:

```

dw    20      ; T
dw    8       ; H
dw    9       ; I
dw    19      ; S
dw    0       ;
dw    20      ; T
dw    5       ; E
dw    24      ; X
dw    20      ; T
dw    0       ;
dw    8       ; H
dw    1       ; A
dw    19      ; S
dw    0       ;
dw    2       ; B
dw    5       ; E
dw    5       ; E
dw    14      ; N
dw    0       ;
dw    8       ; H
dw    1       ; A

```

```
dw    3      ; C
dw    11     ; K
dw    5      ; E
dw    4      ; D
dw    0      ;
dw    0      ;
dw    28     ; *
dw    28     ; *
dw    28     ; *
dw    0      ;
dw    0      ;
```

Simply replace lines 488 through 519 with this and reassemble the demo. When reassembled and run, the new message should scroll across each side of the screen in place of the old one. Check out Figure 7.18 for a screenshot.

Figure 7.18 – The hacked plasma demo scroll message.



### 7.3.1.3 - Recommended Hack

I took the path of least resistance by hacking such a tame message into the demo. However, in the privacy of your own home, school, or religiously-oriented youth group, it is your sacred duty as a hacker to code something profoundly offensive, obscene, and/or personally incriminating into it.

### 7.3.2 - Hacking the Flag Bitmap

The flag demo, written by **Michael Ollanketo**, creates the illusion of a waving, three-dimensional flag by drawing a textured rectangle that is distorted along the Y-axis by a sine wave. To give the illusion of depth and lighting, another sine wave modulates the brightness of each pixel within the rectangle along the X-axis. A background pattern scrolls behind the flag, making transparent segments of the flag visible, thus allowing for non-rectangular flag designs. Figure 7.19 is a screenshot of the flag demo.

Figure 7.19 – The flag demo.

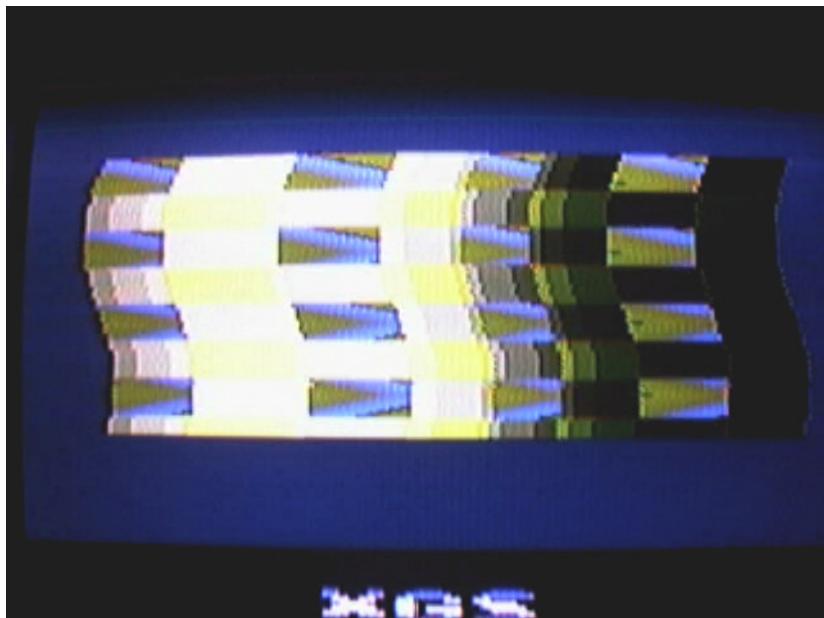


You can find the flag demo and its hacked variants here: [Hacks\PAL\Flags\flags\\_01.src](#).

The bitmap used to texture the flag is stored in a file called **flags1.src**. This bitmap data is stored in a fairly simple format that is easy to hand-edit. Each **DW** directive is a single pixel, allowing for the full 8 bits of color data to be used. The only problem is that the pixel values are listed vertically, rather than organized two dimensionally like the bitmap they describe. The flag's dimensions are 16x16, so I've taken the liberty of presenting a replacement bitmap that's formatted in a more readable fashion.

There's no real point in listing the source code here, since it's simply another design written just as the original was (aside from the improved formatting). Instead, check out **flags1\_hack.src**, which contains two flag bitmaps you can select in and out using conditional compilation. One is blank, allowing you to easily draw in your own designs, and the other is a multi-colored checker pattern. You'll have to experiment to get the color values the way you want. Figure 7.20 is a screenshot of my checker pattern flag:

Figure 7.20 – The flag demo with a hacked checker pattern.



### 7.3.3 - Altering the RotoZoomer Bitmap

The rotzoomer demo, written by **Michael Ollanketo**, demonstrates a classic demo effect in which a bitmap is tiled infinitely across the screen and rotated. Figure 7.21 is a screenshot of the rotzoomer demo.

Figure 7.21 – The rotzoomer demo.



You can find the PAL rotzoomer demo and its hacked variants here:

[Hacks\PAL\Rotzoomer\rotzoomer\\_01.src](#).

In this hack, we're going to alter the bitmap by changing its representation in the **xgsme.src** include file. This file is entirely dedicated to defining the rotzoomer's bitmap in program memory through a long series of **DW** directives. The directives, in order, define each pixel in the bitmap from left to right, top to bottom.

### 7.3.3.1 - Changing the Bitmap

If you've read the NTSC version of this hack earlier in the chapter, I opted to avoid changing the bitmap itself due to the unfriendly format of the bitmap definition. This time, however, I decided to go ahead with a reformat and have put together a human-editable version of the bitmap definition along with a cool game-like texture to display in the rotzoomer. Check out Figure 7.22 for a screenshot.

Here's the reformatted code for the new bitmap. Once again, in order to fit the page the code had to be rewritten to be twice as tall and thus half as wide, but aside from that this is the code itself. You can find a copy of this in the **xgsme\_hack.src** file in the same directory as this demo.

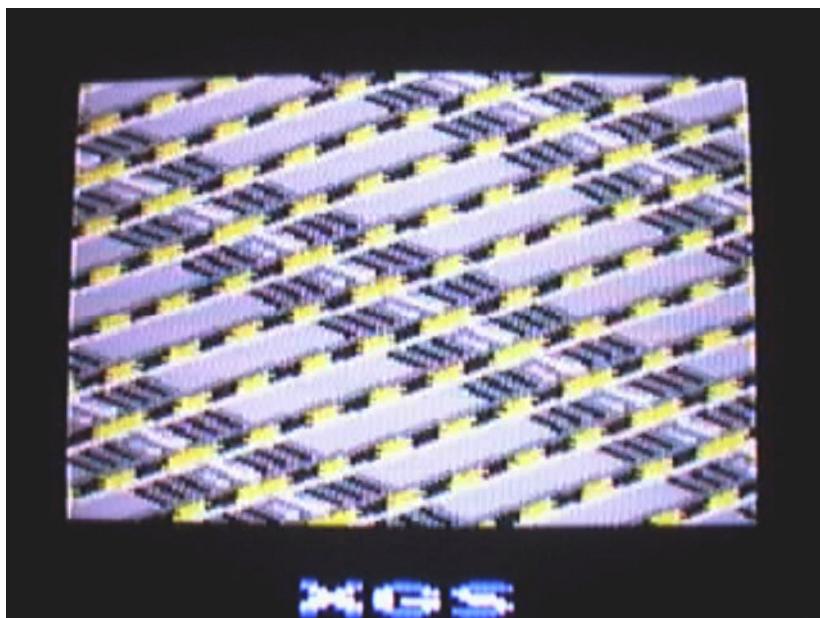
```

DW    255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,254
DW    255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,254
DW    255,253,250,253,250,253,250,253,254,254,254,254,254,254,254,254,254,254,254,254,253,253
DW    254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,253,253
DW    255,254,250,253,250,253,250,253,254,254,254,254,254,254,254,254,254,254,254,254,254,253,253
DW    254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,253
DW    255,254,250,253,250,253,250,253,254,254,254,254,254,254,254,254,254,254,254,254,254,254,253
DW    254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,253
DW    255,255,250,253,250,253,250,253,254,254,254,254,254,254,254,254,254,254,254,254,254,253,253
DW    254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,253
DW    254,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253
DW    253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253
DW    060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060
DW    250,250,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060
DW    060,060,250,250,250,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060
DW    250,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060,060

```

The main source file for the rotozoomer demo, **rotozoomer\_01.src**, has already been edited to use the hacked version of the texture. Assemble and run the program to see it in action. If all went well, the rotating texture should now be the new game-like, futuristic panel, seen in Figure 7.22.

**Figure 7.22 – Hacking the bitmap of the RotoZoomer demo into something more game-like.**



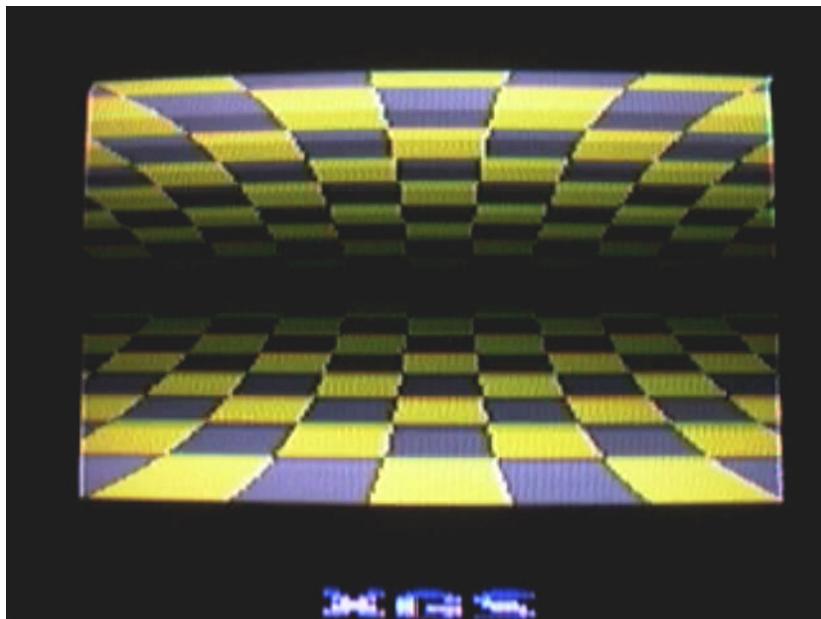
Now that you have an easily editable bitmap, try putting your own designs in!

### 7.3.4 - Hacking the Floormapper Demo

The floormapper demo, written by **Michael Ollanketo**, demonstrates another common demo effect. This time, the effect is called **floormapping** and can be thought of as a rotozoomer in 3D. A texture or pattern

is tiled infinitely over a plane viewed from a first-person perspective. In this particular demo, two such planes are mapped at once and joined in the center to form what appears to be the inside of a flattened cylinder. Figure 7.23 is a screenshot of the floormapper demo.

Figure 7.23 – The floormapper demo.



You can find the floormapper demo and its hacked variants here:

[Hacks\PAL\Floormap\floormap\\_01.src](#).

As you can see in the screenshot, the texture being mapped along the surface of the floors is actually just a simple checkerboard pattern. In fact, this pattern is so simple that there is no corresponding bitmap in the demo; it is simply drawn on the fly by reading bit 3 of a running counter that is updated after drawing each scanline.

This hack will modify the frequency of this checkerboard pattern by simply changing the starting state of one of these counters. The counters we will modify are called `du` and `dv`. Each counter is represented in the code with two variables, however; `du_w` and `du_f`, and `dv_w` and `dv_f`, respectively. The “w” and “f” refer to the **whole** and **fractional** components of each of these counters. These two variables come together to form a 16-bit, 8.8 fixed-point value, thus giving the program a greater range of precision than would be possible using standard 8-bit integers.

Of course, a real discussion of fixed-point math is beyond the scope of this guide and not something that is relevant to simple hacking anyway. It is important, however, to understand at least something about the code you’re tinkering with if you can. So, with the preamble out of the way, check out line 112:

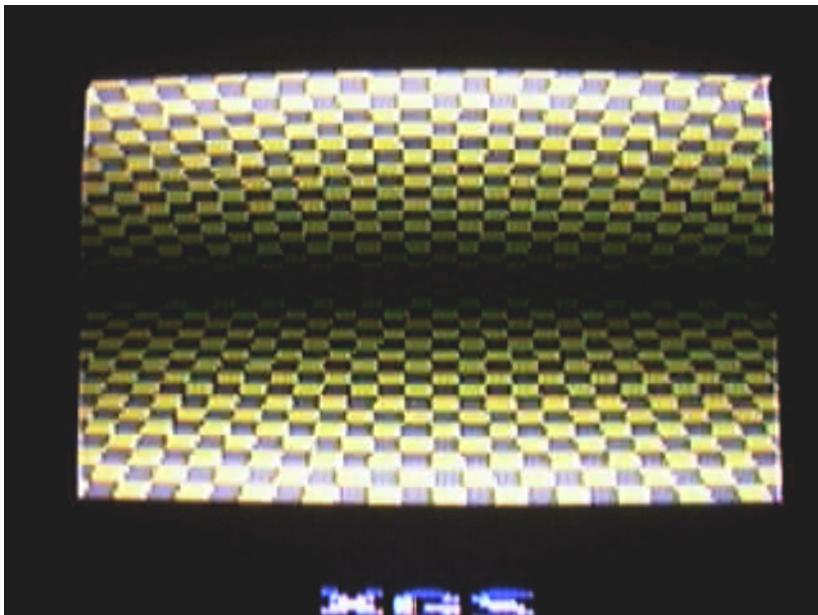
```
clr du_w
mov du_f,#64
clr dv_w
mov dv_f,#64
```

This code is initializing the U and V axis counters. It's clearing the whole part and setting each fractional part to 64. What's interesting is what happens when these initial values are changed. For example, change the `#64` on line 112 to `#32`, and the `#64` on line 114 to `#255`, so it looks like this:

```
clr du_w
mov du_f,#32
clr dv_w
mov dv_f,#255
```

Reassemble the program and check out the results (depicted in Figure 7.24). Cool, huh? The checkerboard has broadened considerably on the horizontal axis, and been condensed vertically. Try different values in both the whole and fractional components of the counters. What kind of results do you get?

Figure 7.24 – The floormapper with hacked checker dimensions.



## 7.4 - Moving On

This chapter has given you an introduction to XGS ME development through the simple hacking of many of the included demos. As a next step, read the included eBook, **Beginning Assembly Language on the SX Microcontroller**, found in the **eBooks** directory, to start real assembly language programming on the

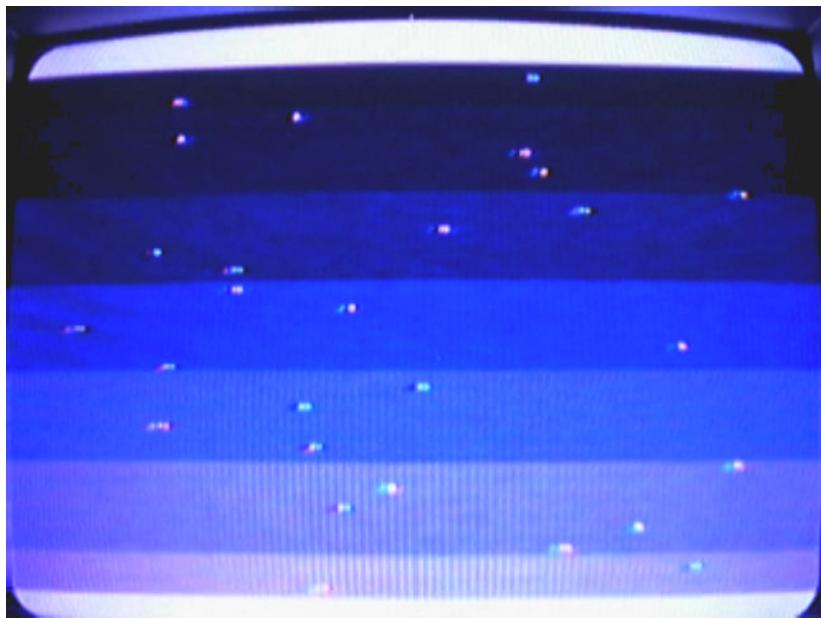
SX52. Of course, to really code anything, you'll need to understand the material presented in the real XGS eBook, ***Design Your Own Video Game Console***.

## Chapter 8: Case Study: The Starfield Demo

This chapter provides a brisk overview of developing a full, graphical program for the XGS ME. While full coverage of XGS ME development is relegated to the main eBook, **Design Your Own Video Game Console**, this chapter can be thought of as an informative crash course in the most important parts of any XGS ME program—the TV signal driver and the core program logic.

The demo presented in this chapter is a simple graphical effect called a **starfield**, in which pixel-sized “stars” scroll horizontally across the screen at different rates. These different rates give the illusion of depth, since the slower “stars” appear to be further away than the faster ones. When coupled with a background that resembles a sky and a brightness level for each star that is proportional to its speed, the end result is a convincing sky backdrop. See Figure 8.1 for an example of this:

**Figure 8.1 – The Starfield Demo.**



You can find the starfield demo in the **Demos\NTSC\Starfield** directory on the **XGS ME Software CD**.

### 8.1 - Organization of the Demo

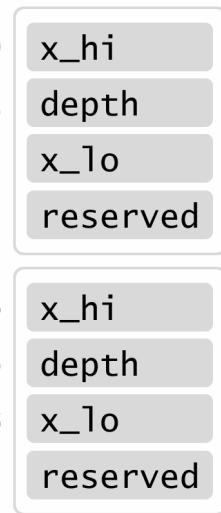
Before getting into the specifics of programming and implementation, let's discuss the general organization of the demo program in terms of its algorithms and data structures.

## 8.1.1 - Data Structures

### 8.1.1.1 - The Star List

The heart of the algorithm is maintaining a list of stars, wherein each star is a data structure containing both its current horizontal location as well as its speed. The star's X coordinate is stored as a 16-bit fixed-point number to allow for smoother motion, which will be explained in more detail shortly. In addition to these two bytes, one byte is used to represent the star's "depth", a value used to determine its speed and brightness. Lastly, the structure is padded to four bytes with a single unused byte. Whenever possible, align the size of your data structures with powers of two, or at least multiples of 8, 4 or 2. Figure 8.2 illustrates this data structure visually.

**Figure 8.2 – The starfield is represented as a list of simple Star data structures.**



One interesting note about development in SX assembly language (as well as others), is that data structures need not be specifically "allocated". Rather, the structure's space taken up in RAM is simply "set aside" and avoided by other parts of the program. In other words, a list of 32 stars that each require a 4-byte data structure require a total of 128 bytes. As long as the base address of a contiguous 128-byte region is known, and the rest of the program is written such that it does not use any of the 128 bytes following that address, the data structure can be safely built and used within that space. We'll come back to this in a moment.

### 8.1.1.2 - The Rest of the Program

Aside from the star list, the program needs a handful of globals and variables to manage the generation of the TV signal.

The globals are available no matter which bank is selected, which makes them flexible and, depending on the situation, faster than the rest of the memory. Because of this, I like to use as much space as I can in my globals for simple temporary registers, giving my program fast, easy access to “scratch pad” registers whenever it needs them. In the case of the starfield, I didn’t need any globals other than the temporaries, so all six globals were used:

```
ORG      $0A
t0        DS      1      ; Temporary registers
t1        DS      1
t2        DS      1
t3        DS      1
t4        DS      1
t5        DS      1
```

The remaining variables are used to generate the TV signal, and are given their own bank:

```
ORG      BANK_TV_SIGNAL
luma     DS      1      ; Temp for luma
chroma   DS      1      ; Temp for chroma
comp_video DS      1      ; Temp sum of luma and chroma
burst_phase DS      1      ; Temp for burst phase index

scanline  DS      1      ; Scanline counter
counter   DS      1      ; General counter
counter2  DS      1      ; General counter
```

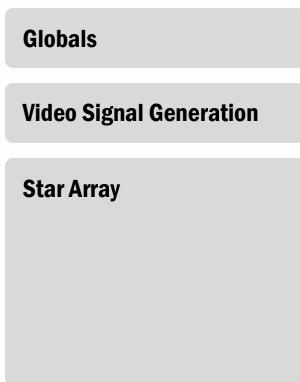
The globals and TV signal generation globals have been allocated, and the remainder of the memory space is free for use. Since only the first 7 bytes of the first bank of memory have been used, there is plenty of room left over for our 128-byte starfield array. And while this memory is free to use as-is, and no explicit declaration is necessary, it’s still nice to do the following:

```
ORG      BANK_STARFIELD
```

This leaves a visual cue in the source code that a memory region is in use starting at this address, making it more readable. Also, if in the future we decide to allocate specific registers within this region for whatever reason, the `ORG` is already there to provide a base for the declarations.

That takes care of the data structures—the global scratch registers, TV signal generation registers, and starfield array are all ready to use. The beauty of an architecture as simple as the SX52 is that once a program’s data has been organized, a system-wide memory map is very easy to visualize. Figure 8.3 is illustrates the final layout of the starfield demo’s memory map.

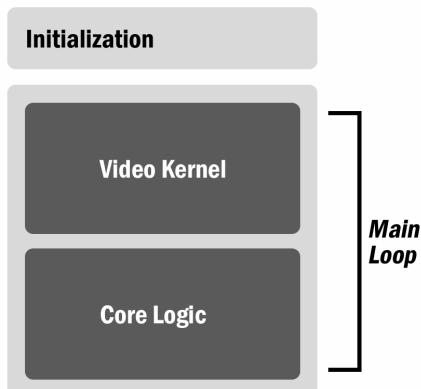
**Figure 8.3 – The Starfield Demo’s Conceptual Memory Map**



## 8.1.2 - Algorithms and Logic

The code for any graphical XGS ME program follows a relatively uniform structure. The heart of the program is the **video kernel**, which is the program’s main loop. The video kernel is responsible for maintaining a solid TV signal, which keeps the program’s graphics smoothly displayed and updated on the screen. Preceding the video kernel is the initialization code, where the SX52 is configured, and global data structures are initialized. Optionally following the video kernel code are subroutines, which may be called by the video kernel or initialization code. Since the XGS ME is a traditional video game console, it may be physically turned off at any time and without warning. As such there is no need for any particular “shut down” code. For example, there is no underlying operating system or parallel tasks and processes to which resources must be returned. This program organization is illustrated in Figure 8.4.

**Figure 8.4 – The Starfield Demo’s Program Organization.**



### 8.1.2.1 - Initialization

Initializing the program is relatively easy compared to the core logic that follows, as is usually the case. The first step in the initialization of any XGS ME program is configuring the SX52 chip itself. In this case, that really just means setting the RE port to output. The RE port is responsible for sending the video signal to the video subsystem, which is why each of its 8 pins must be set to output:

```
MOV     RE, #00000000 ; Set RE to output
MOV     !RE, #00000000
```

The next and final step in initializing the program is more complicated. The starfield array we've outlined in the last section must now be initialized; in other words, each of our 32 stars must be placed in their initial positions onscreen and given the depth values that determine their speed and brightness level. This is handled by a subroutine called `Init_Stars()`.

The subroutine works by iterating through each star in the starfield (the 4-byte star data structure described above), and giving them a random X location and a random depth value. The first problem here should be obvious—how do you generate a random number on the SX52?

This issue is more complex than you may think. If you're used to programming in high-level languages like C and C++, you take your standard library's random number functions for granted. In the world of low-level system programming, however, there usually isn't room to write an entire random number generator, especially on a system with only 4K of program memory like the SX52. And even if there were, you'd have to write it yourself, because there isn't a standard library to include in the first place. Instead, this problem is solved with lookup tables.

Random number lookup tables are nothing more than a string of numbers generated beforehand by a random number generator and saved for later use. Once this list is generated, it's formatted in a special way using the `RETW` instruction:

```
Rand_Num_0_255
JMP    PC + W
RETW  $29, $6B, $D6, $EB, $2C, $A9, $03, $21
RETW  $BB, $EF, $5F, $4C, $FC, $10, $EC
RETW  $BE, $D4, $ED, $51, $06, $45, $4D, $99
RETW  $25, $8E, $51, $65, $53, $05, $5C, $33
RETW  $EC, $3F, $54, $16, $A7, $22, $CD, $CC
RETW  $8F, $60, $D4, $F3, $4E, $4A, $60, $3D
RETW  $CB, $EE, $2F, $68, $16, $75, $93, $6D
RETW  $35, $33, $F4, $0D, $4C, $E6, $05, $39
```

The interesting thing here is that even though this is conceptually just a table of data, it is actually implemented as executable code. In order to read from this table, it is called as if it were a subroutine, like so:

```
CLR    W
CALL  Rand_Num_0_255      ; Set W to zero
                           ; Read the random number at index 0 and store it in W
```

Once this function is called, execution moves to the **JMP PC + W** instruction seen at the beginning of the table. This is a special instruction that, as it appears, jumps to the current instruction (the **JMP** instruction itself) **plus** the current value of **W**. So, if **W** is zero, execution will jump to the first instruction after the **JMP**, which is the first value on the first **RETW** line (**\$29**). If **W** is one, it will jump to the second value on the first **RETW** line (**\$6B**). If **W** is two, it will jump to the third value on the first **RETW** line (**\$D6**). In other words, if we think of each value in each **RETW** instruction as a separate item in the table, then the value of **W** upon entering this subroutine is really an index into this table.

The other half of the puzzle is **RETW** itself. This instruction is just like **RET** or **RETP**, except that it sets **W** to the specified value before jumping back to the function, which is similar to a function's return value in a high-level language like C or Java. Before calling the subroutine, **W** contains the index of the desired item to read. After returning, **W** contains the value found at that index. This is how **CALL**, **JMP PC + W**, and **RETW** work together to implement very fast lookup tables.

Lastly, in case you're confused, listing multiple comma-separated values in a **RETW** instruction is the equivalent of separate **RETW** instructions for each of those values. It is simply a notional convenience, but in the final object code, there is a separate **RETW** opcode emitted for each value listed. For example,

```
RETW    10, 20, 30
```

is the same as

```
RETW    10
RETW    20
RETW    30
```

The following is the source code to the **Init\_Stars()** subroutine:

```
Init_Stars

MOV      FSR, BANK_STARFIELD          ; Point to the base of the star array
MOV      t0, #STAR_COUNT             ; Number of stars to initialize
CLR      t2                           ; <t2> is the random number table index

:nex_star

; Get a random value between 0 and 159 by first getting a random value
; between 0 and 128 and centering it.
MOV      W, t2                         ; Get the next random table value
CALL    Rand_Num_0_255
AND     W, #127                        ; W %= 128
MOV      t3, W                         ; Save the X coordinate in <t3>
ADD     t3, #16                         ; Center the X within 160 pixels
MOV      IND, t3                       ; Set the star X coordinate
INC     FSR                           ; Move to the next star attribute

; Get a random depth value
MOV      W, t2                         ; Get the next random table value
CALL    Rand_Num_1_4
MOV      IND, W                         ; Set the star "depth"
INC     FSR                           ; Move to the next star attribute

; Clear the low byte of the X coordinate
MOV      IND, #0
```

```

INC      FSR
; Clear the reserved attribute
MOV      IND, #0
INC      FSR

INC      t2          ; Move to the next value in the random tables
DJNZ    t0, :next_star ; Move to next star

_BANK   ( BANK_TV_SIGNAL ) ; Return to the default bank

End_Init_Stars

```

Once you understand how random numbers are implemented, and how the stars are stored in memory (covered above), the subroutine itself is pretty self-explanatory. First, the **FSR** register is initialized to the base address of the star array. This points to the first byte of the first star record. Then, a counter is initialized in **t0** to the **STAR\_COUNT** constant (32) to loop through each star in the array.

At each iteration of the loop, a random number between 0 and 255 is read from the table and converted to an appropriate horizontal star location, and stored in that first byte. Remember that the star location is a 16-bit fixed point value; the first byte is the whole part of the number, and the second is the fractional part. To start each star off, all that must be set is the whole part. This means that the initial locations for the stars might be 122.0, 11.0, 76.0, 150.0, etc.

**FSR** is then incremented, pointing it at the second byte of the first record, which is where the depth value is stored. A second lookup table is used to generate a random value between 1 and 4 instead of 0 and 255. Since depth values range from 1 to 4 (meaning stars can move between 1 and 4 pixels at a time), this table allows the depth field of each star to be initialized very easily. The value is simply read from the table and stored directly.

**FSR** is again incremented, pointing it at byte three of the record, where the fractional part of the X-coordinate is stored. Again, there is no need to start stars off on fractional locations, so this is simply cleared to zero. **FSR** is incremented a third time to point it at the record's last byte, the reserved field. This field is cleared simply for the sake of being neat and tidy. Lastly, **FSR** is again incremented, now pointing at the first byte of the **next** record, and the loop iterates so that record may be initialized as well.

### 8.1.2.2 - Anatomy of the Video Kernel

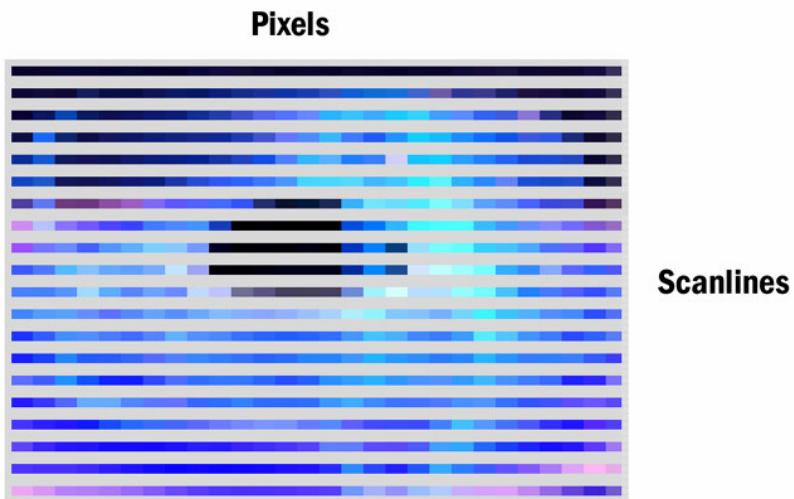
The heart of the program is, of course, the video kernel. This continuous loop implements the two most important parts of the program: generating the video signal and implementing the “core logic” of the program itself. Not surprisingly, this usually comprises at least 90% of any XGS ME’s source code and must be organized cleanly.

Although a complete treatise on the generation of video signals is provided in the main eBook, **Design Your Own Video Game Console**, the basic anatomy of a video kernel will be covered here as well.

Generating a video signal really means sending the television a series of **frames**. A frame is a single, full-screen image, composed of roughly 200 horizontal **scanlines**. Each scanline is a strip of color data

extending from one side of the screen to the other, broken up on regular intervals to form pixels. When you organize these scanlines vertically, the end result is a complete static image. Figure 8.5 illustrates the structure of a single TV frame.

Figure 8.5 – Conceptual format of a TV frame.



The entire video signal is sent over a single wire, which means all data is sent over time in a serial fashion—pixel after pixel, scanline after scanline, frame after frame. This data must be sent in adherence to an extremely timing-sensitive sequence. The TV can't stop to wait for you to “catch up” if you take too long sending something, nor can it keep pace if you send data too fast; it just keeps displaying whatever data, or lack of data, it has received. If your code loses synchronization with the TV at any time, the image displayed onscreen will immediately begin degrade and ultimately fall apart completely. Because of this, the code responsible for generating this signal must be timed extremely precisely. You will note that in most of the XGS ME demo programs, each instruction in the signal-generation code is commented with a number, such as (2) or (4). This is the number of clock cycles the instruction takes to execute, and when converted to a time in microseconds (based on the 80 MHz clock speed of the SX52), the timing of the signal generation code can be made to sync up exactly with the timing expected by the TV.

It is beyond the scope of this guide to cover the exact details of TV signal generation, but for the sake of understanding the starfield demo, we will now take a brief look at the major parts of the signal generation routines. Remember that the real info is found in the main eBook, **Design Your Own Video Game Console**.

### 8.1.2.2.1 - Overview

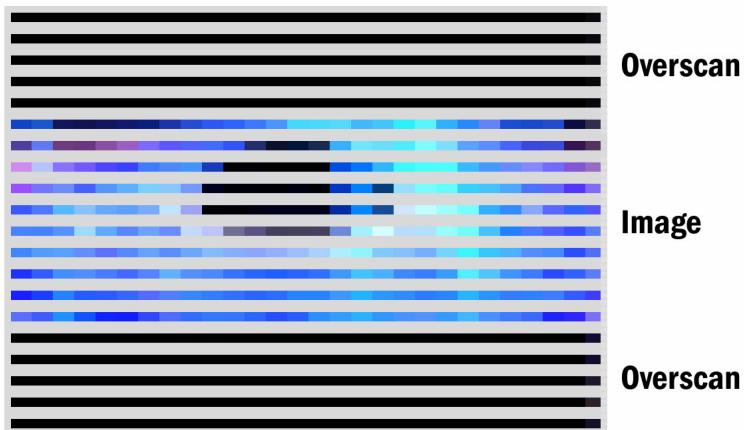
The SX52 writes all TV signal data to the RE port. This port inputs into the XGS ME's video subsystem, which completes the signal and sends it to the TV. As for the signal itself, the following steps are taken to generate a frame of TV data according to the standard TV signal specification:

1. Start the frame by generating the screen's top overscan. This is a region of approximately 20-40 blank scanlines that precedes the visible part of the frame.
2. Generate each scanline of visible screen data. Each scanline begins with a brief period of horizontal sync, after which follows the visible scanline itself. At regular intervals along the scanline, proportional to the width of a pixel on the physical screen, the color is changed to the next pixel's value.
3. The frame is completed by drawing the bottom overscan, another region of blank scanlines roughly the size as the top overscan, located below the visible region of the frame.
4. A relatively long period of time is required for the TV's electron guns to reset to the top of the screen again, having worked their way down to the bottom while drawing the frame. This period is called the vertical sync, and, fortunately for us, gives us plenty of time to perform the core logic of the program in preparation for the next frame.

If that didn't make sense to you, don't worry—the purpose of this tutorial is not to teach you the ins and outs of video signals. For the complete, ground-up coverage of this subject you'll need in order to write your own XGS ME games and demos, refer to the main eBook, ***Design Your Own Video Game Console***.

For now, simply understand that the overscan regions and vertical sync are necessary in order for the TV to properly understand and translate the image data sent in between, and that the code shown in this section is capable of generating these signals. Figure 8.6 illustrates this process visually.

**Figure 8.6 – Basic anatomy of a TV frame.**



### 8.1.2.2.2 - Drawing the Frame

During each scanline, aside from formatting the TV signal properly, our job is of course to send out the right pixel data to draw the screen as we wish it to be seen. As a given frame is being drawn, the raster is always moving left-to-right, top-to-bottom. This is how we must approach the process of translating our star data into the image displayed on the screen.

The starfield draws 192 scanlines of vertical resolution in between the top and bottom scanline regions. Within reason, this number can be increased or decreased by adjusting the size of these scanline regions to suit the vertical resolution to your particular needs.

The current scanline is stored in a counter called **scanline** which counts from 192 down to 1, since the SX52 instruction happens to make it easier to count from a given number down to zero, rather than from zero up to a given number. Since most graphics displays think in terms of the upper-left corner being defined as coordinate 0, 0, this somewhat inverted approach should be kept in mind.

The first step in drawing the scanline is determining the background color to draw behind the stars. The starfield background is a purple gradient, which is easy to implement by simply scaling the current scanline down to a range that can be used as a luminance value. Bit rotation is used to scale the scanline value down by a factor of 32:

```
; Get the next color in the sky gradient, based on the scanline counter

MOV    t4, #192           ; (2) Get the current scanline, inverted
SUB    t4, scanline        ; (2)
CLC    t4                 ; (1) Divide it by 32
RR     t4                 ; (1)
CLC    t4                 ; (1)
RR     t4                 ; (1)
MOV    t5, #COLOR14        ; (2) Set the base color of the sky
ADD    t5, t4              ; (2) Modulate the intensity to the current level
```

Note that the clock cycles required by each instruction are noted in the comments. This is because the logic that draws the scanline will most likely not take up the exact amount of time for which the TV expects the scanline signal to last. Since it will probably take somewhat less time, an empty delay must follow this logic for the remainder of the time to keep the program in sync with the TV. **t5** now contains the background color value for the current scanline.

Since there are 192 scanlines and only 32 stars, it is obvious that not every scanline will contain a star. In fact, most scanlines won't. Because of this, the scanline logic must now branch off into one of two possibilities—scanlines that are simply pure background color, and scanlines that contain a star. The logic for determining this is simple—since there are 32 stars and 192 scanlines, drawing a star on every 6<sup>th</sup> scanline will space them out evenly down the screen. Since 6 is not an easy divisor when dividing with bit shifting, a counter stored in **t3** is used:

```

; Determine if a star lies on this scanline and branch to the proper
; scanline rendering code

INC    t3          ; (1) Increment the scanline counter
CJB    t3, #192 / STAR_COUNT, no_star ; (4/6) Reset after each star is
; drawn
;
CLR    t3          ; (1)
NOP
JMP    star_scanline ; (3) Draw a scanline containing
; a star
no_star JMP    blank_scanline ; (3) Draw an empty/blank scanline

```

Don't worry if it sounds like this even spacing will make the starfield look unnatural—due to each star moving at a random speed and being drawn with a random brightness, it does not create a noticeable pattern as more than two stars rarely line up.

Scanlines that do not contain a star are decidedly easy to rasterize. The appropriate background color (now stored in **t5**) is simply sent out to the video hardware, and a long delay is used to hold that color for the duration of the scanline:

```

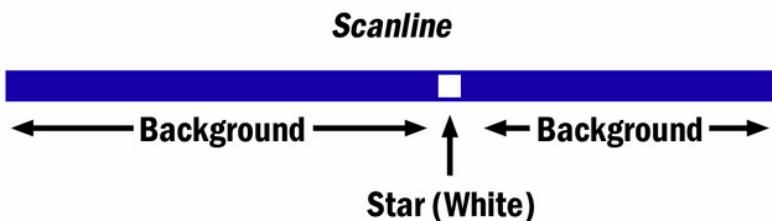
blank_scanline

MOV    RE, t5          ; (2) Set the base color of the sky
DELAY  ( 4208 - 2 - 28 ) ; Delay for the duration of the scanline
DJNZ   scanline, raster_scanline ; (2/4) Next scanline

```

In the case of non-blank scanlines, the logic is more complex. Since we have no control over when each pixel is drawn, we must simply keep up with the TV's expectation of the scanline being sent left-to-right. On a more traditional computer display in which we can actively draw any pixel at any time, drawing the star would be as easy as setting the corresponding value in video memory. By contrast, we must passively *wait* for the point at which the TV is drawing the pixel containing the star before we can draw it, and draw the background color the rest of the time. Figure 8.7 visualizes this concept.

**Figure 8.7 – A timeline view of when a star is drawn within the scanline signal.**



The following code is responsible for drawing the star at the right time by setting the background color, waiting for the TV to reach the star, setting the star color (with a brightness based on its depth value), and then restoring the background color and delaying until the end of the scanline is reached:

```
star_scanline
```

```

; **** Get the attributes of the current star

MOV    FSR, BANK_STARFIELD      ; (2) Set the base offset of the star array
MOV    t0, t2                   ; (2) Get the address of the star in <t0>
CLC
RL    t0                      ; (1) Multiply index by 4 for address
RL    t0                      ; (1)
ADD    FSR, t0                 ; (2) Add the star offset to the base
MOV    t1, IND                 ; (2) Save the star's X coordinate in <t1>
INC    FSR                     ; (1) Move to the next attribute
MOV    t0, IND                 ; (2) Save the star's depth in <t0>
CLC
RR    t0                      ; (1)
RR    t0                      ; (1)
RR    t0                      ; (1)
RR    t0                      ; (1)
MOV    t4, #COLOR14           ; (2) Set base color of the star
ADD    t4, t0                 ; (2) Set the star's intensity
_BANK  ( BANK_TV_SIGNAL )    ; (2)
INC    t2                      ; (1) Move to the next star

; **** Draw the scanline

MOV    t0, #160                ; (2) Rasterize 160 pixels across
                                ; the scanline

pixels_loop

    CJNE  t0, t1, empty_pixel  ; (4/6) Is this the star pixel?
    NOP
    NOP
star_pixel
    MOV    RE, t4               ; (2) Set the star's color
    JMP    pixel_set            ; (3) The pixel color is set
empty_pixel
    MOV    RE, t5               ; (2) Set the base color of the sky
    NOP
    NOP
    NOP
pixel_set
    DELAY ( 10 )              ; Pad the remainder of the pixel
    DJNZ  t0, pixels_loop     ; (4) Next pixel

; **** Complete the scanline

DELAY ( 48 - 28 - 26 )        ; Pad the remainder of the scanline

DJNZ  scanline, raster_scanline ; (2/4) Next scanline
JMP   scanlines_done          ; (3) Skip past the scanline
                                ; rasterizer below

```

The actual drawing routine is fairly simple if you understood how the initialization routine worked. In this case, a counter stored in **t2** tracks which star to draw next. This counter must be multiplied by 4 to convert it into an actual pointer to the star record's location in memory. This pointer is stored in **t0**, which is used as a base reference to the star record's field.

The star is drawn using only the whole part of its 16-bit fixed point X coordinate; the fractional part is only used when moving the star, so we need only read the first byte to determine where onscreen the star should appear. This value is stored in **t1**. After that, the brightness is read, added to **COLOR14** (the white base color of the stars), and stored in **t4**.

With the X-coordinate and brightness read from the array, the actual drawing can take place. This is done with two loops. The first loop, after the background color is set, delays from the start of the scanline until just before the star must be drawn. This effectively rasterizes the blank sky on the left side of the star. The star is then drawn. The background color is quickly set once again (to ensure the star only occupies a tiny fraction of the screen), and the second loop stalls until the end of the scanline is reached, filling in the blank sky on the star's right side.

### 8.1.2.2.3 - Updating the Demo

At each vertical sync, as the TV is re-synchronizing itself in order to draw the next frame, we'll have plenty of time to update the program. This updating is ultimately done by a subordinate called `Update_Stars()`, listed below:

```
Update_Stars

    MOV     FSR, BANK_STARFIELD      ; (2) Point to the base of the star array
    MOV     t0, #STAR_COUNT         ; (2) Number of stars to initialize

:nex_star

    ; Get the star's depth
    INC     FSR                  ; (1) Skip to the depth attribute
    MOV     t1, IND                ; (2) Save the depth attribute

    ; Perform a 16-bit "fixed point" addition on the 8.8 X value
    INC     FSR                  ; (1) Skip to the low byte of X
    CLC
    ADD     IND, t1                ; (2) Add the star's speed to it
    DEC     FSR                  ; (1) Move back to the high byte of X
    DEC     FSR                  ; (1)
    ADDB   IND, C                 ; (2) Add the carry from the low byte to the high byte

    ; Bounds check the star's X coordinate
    CJB     IND, #160, :in_screen  ; (4/6)
    NOP
    NOP
    CLR     IND                  ; (1) Clear X.hi
    INC     FSR
    INC     FSR
    CLR     IND                  ; (1) Clear X.lo
    JMP     :bounds_checked       ; (3)

:in_screen
    INC     FSR                  ; (1) Skip to the last attribute
    INC     FSR
    NOP
    NOP
    JMP     $ + 1                 ; (3)

:bounds_checked

    INC     FSR                  ; (1) Skip to the reserved attribute

    ; Handle the reserved attribute
    MOV     IND, #0                ; (2)
    INC     FSR                  ; (1) Skip to the next star

    ; Move to next star
    DJNZ   t0, :next_star        ; (2/4)

    _BANK   ( BANK_TV_SIGNAL )    ; (2) Return to the default bank
```

```

RETP          ; (3)
; ***** Total: ( 32 * STAR_COUNT ) + 2 + 3 + 2 + 2

End_Update_Stars

```

This subroutine is very similar to `Init_Stars()`. It loops through each star in the array, but this time, instead of initializing the values, it simply moves the star to its next location by performing a simple 16-bit fixed point addition. The depth value is read and stored in `t1`. `t1` is then added to the fractional byte of the X-coordinate. The carry flag, either one or zero, is then added to the whole byte. If the fractional byte wraps-around from 255 to 0 when the depth value is added to it, the carry flag will be set and the whole byte will be incremented. This is how the multi-byte addition allows the stars to move at a smooth, controlled rate; by altering the depth value, which is the value by which the fractional part increments, we can control how fast the whole part increments as well. Since the whole part is all we use to actually draw the star within the scanline, this has a direct impact on how long the star appears to take to move across the screen.

A bounds check is then performed to see if the star has moved beyond the edge of the screen. If so, it is reset to the other edge, allowing each star to loop at its own pace. Since the stars all move at different speeds, it is not easy to tell that the stars are looping, and it appears as if unique stars are constantly scrolling into view.

## 8.2 - Conclusion

That's it! You've seen how this graphical demo is initialized, and how it is both displayed and updated at each frame. During the initialization, drawing and updating you can do anything you want, from displaying a starfield to implementing an entire game. In other words, what we have covered is a flexible skeleton upon which any graphical XGS ME program can be built.

Of course, not every important detail has been covered here, as this is intended primarily as a superficial guide to give you the idea of what's involved in writing graphical XGS ME programs without drowning you in every relevant detail of the exact implementation. For this, please refer to the demo's full source code, as well as the main eBook, ***Design Your Own Video Game Console***, of course.

## Chapter 9: Case Study: Racing Engine Demo

The last chapter, a case study on the graphical starfield demo, covered numerous aspects of writing graphical XGS ME programs, most of which pertained to the all-important TV signal generation. Since this case study covers a considerably more complex program, it will not concern itself with listing this information again. Please read the last chapter's case study before attempting to read this one, as much of this chapter is dependant on information presented in the last.

The racing demo is a reasonably complex example of detailed, full-screen graphics with a pseudo-3D perspective. It creates a smoothly animating race track with a distant mountain range as its backdrop that scrolls appropriately as the driver turns. Special details are included to enhance the visuals, such as a blue gradient effect in the sky and the classic red-and-white track border that scrolls with the road. See Figure 9.1 for a screenshot of the racing engine demo in action.

The racing engine demo can be found in the **Demos\NTSC\Racer** directory.

**Figure 9.1 – The racing engine demo.**



There are a handful of major elements required to make this demo work:

- The two-digit speed printed at the top
- The blue-gradient sky
- The mountain range that scrolls with the turns of the track

- The track and its pseudo-3D perspective, as well as its ability to warp into the shape of a left or right turn
- The red and white track markers that scroll along with the player's movement
- Joystick input from the player

Once again, since the last chapter covered the aspects of writing an XGS ME program that are not specific to this demo, they will not be covered again here since they are more or less the same. As a result, this chapter will focus entirely on the specifics of how each element was implemented and each effect achieved.

## 9.1 - Data Structures

Unlike the starfield demo and its star array, the racing engine demo does not have a large, central data structure upon which everything else is based. Because of this, it would make more sense to individually cover each aspect of the demo and its related data, rather than cover all of the data first and all of the code afterwards.

## 9.2 - The Sky Background and Mountain Range

The first major element of the racing engine demo to discuss is the background, consisting of a blue gradient sky and a scrollable mountain range. The mountain range in particular is an illuminating look at how complex graphics can be implemented on the XGS ME using simple tricks to overcome its incredibly small amount of on-chip memory.

The mountain range is a detailed, high-resolution graphic that covers the entire screen. It is a total of 256 pixels across, which is even wider than the screen's horizontal resolution. The mountain range is a total of 44 scanlines tall, which means that a bitmap capable of containing this entire mountain range would be  $44 \times 256 = 11\text{K}$ ! With only 262 bytes of RAM and 4K of program space, this is definitely impossible. So where does the mountain range come from?

### 9.2.1 - The Mountain Height Table

The answer is a lookup table similar to the one we discussed in the last chapter for producing pseudo-random numbers. Since a complete 11K bitmap is not possible on the SX52 (without using the slower, off-chip SRAM), the mountain range had to be represented in a form that only stored the absolute minimum amount of data. It also had to be fast, however, because each pixel drawn in the mountain range only has about 20 clocks to perform its logic. This means that any sufficiently sophisticated compression algorithm would probably be too complex to implement on a per-pixel basis.

The solution was a **height map**. A height map, in this case, allows us to think of the mountain range as a simple one-dimensional entity, rather than a two-dimensional bitmap. Why is the mountain range only

1D? Because it only changes along the X-axis. If you were to look at any 1-pixel vertical strip of the mountain range, you would find that above the mountain range, the strip is solid sky, and below the mountain range, it is solid mountain. Figure 9.2 illustrates this.

**Figure 9.2 – Analyzing 1-pixel vertical strips of the mountain range graphic.**



Because of this, when drawing any pixel in the background, we do not need to perform a 2D bitmap lookup to determine the color; all we need to know is whether or not we are above the level of the mountain range at this particular point on the X-axis. If so, we draw using the sky background color. If not, we draw using the mountain color. Think of the mountain range table as any mathematical function (such as trig functions), except it's drawn not as a line, but as one solid color above, and another solid color below.

This effectively removes the Y-axis from the definition of the mountain range and thus reduces the memory needed from an impossible  $256 \times 44$  bytes to a more-than-possible 256 bytes. With a 256-byte single lookup table, an entire mountain range can be stored:

```
mount_line
DW 15,14,14,13,11,10,09,08,06,05,06,06,07,08,09,11
DW 12,14,16,18,19,20,22,24,24,24,25,26,26,24,23,23
DW 22,21,20,19,19,18,18,18,17,17,17,17,17,18,18,18
DW 19,20,21,22,23,23,24,24,25,25,26,26,27,27,28,28
DW 27,27,26,25,25,24,23,23,23,23,24,24,25,25,26,27
DW 28,28,28,28,28,29,29,29,29,30,30,31,31,32,33
DW 34,35,36,36,36,36,35,34,33,32,31,30,27,24,23,22
DW 23,23,24,24,25,26,27,27,28,28,29,32,34,34,35,35
DW 36,37,36,35,34,33,32,31,30,30,29,28,27,26,25,25
DW 24,24,24,23,22,21,20,20,19,19,18,17,17,16,15,15
DW 16,17,17,18,19,19,19,18,17,16,15,14,14,13,13,12
DW 12,12,12,11,11,11,12,12,13,14,15,16,17,19,21,22
DW 23,24,24,25,27,28,29,31,30,30,29,28,27,26,25,25
DW 24,24,24,23,22,21,20,20,19,19,18,17,17,16,15,15
DW 16,16,17,17,18,19,20,22,24,25,26,27,27,26,25,25
DW 24,24,24,23,22,21,20,20,19,19,18,17,17,16,15,15
```

To avoid confusion, note that this table has been condensed to fit the page; it appears twice as wide and half as tall in the actual source code.

**NOTE**

While the mountain range height table data could be generated in numerous ways, it was literally created by hand-editing each value in the table seen above, recompiling, and making changes until it looked right. While it would be overkill for a program such as this, any program that needs a lot of high-quality lookup-table-based graphics would benefit from a graphical utility that translates mouse-drawn curves and shapes into a table as seen above.

### 9.2.2 - Another Approach to Lookup Tables

Something worth noting is that this lookup table is not implemented exactly like the random number tables from the last chapter were. Note that the **JMP PC + w** and **RETW** instructions are nowhere to be found. Instead, this particular approach to lookup tables uses the **DW** directive, which directly writes 12-bit word values to the program memory.

To read from a table such as this, a new instruction called **IREAD** is used. **IREAD** accepts a 12-bit program memory address stored in **M:w** and returns the 12-bit program word found at that address, also in **M:w**. In both cases, the high-nibble is stored in **M**, and the low-byte is stored in **w**.

The advantage to using **IREAD** is that it can be used to read from much larger tables; since **w** alone is used as the table index when reading with the **CALL** instruction, only 256 total bytes can be read. Using **M:w** and **IREAD** allows for a 12-bit address space and thus tables up to 4096 bytes (although in reality such a table would not be practical).

The disadvantage is that **IREAD** is somewhat slower; the **IREAD** instruction alone takes 4 clock cycles to execute, whereas **CALL** only takes 3. Furthermore, loading both **M** and **w** with a table index of course takes more time than simply loading **w**, making the **IREAD** approach more costly as far as clocks go.

### 9.2.3 - Drawing the Background

Now that we understand that data structure behind the mountain range, let's talk about drawing it. To draw each pixel of the background, the following steps are taken:

- Determine the current color in the background gradient using the scanline counter.
- Determine whether or not the pixel is above or below the mountain range at that location along the X-axis.
- If the pixel is above the mountain, draw it using the sky background color.
- If the pixel is below the mountain, draw it using the mountain color.

In the actual implementation, each scanline is drawn in a single loop. The sky color for that scanline is determined outside of that loop, saving each pixel from having to re-calculate the color itself. Here is the code:

```

; 44 scanlines of lower sky are left after above loop
draw_sky_loop_1

; ***** SET UP SCANLINE *****
CALL    @Start_Scanline

; ***** DRAW SCANLINE *****
; ***** Set up the scanline

; Put the color of the current sky scanline in <t1>
MOV    t0, #64          ; (2) Base the gradient on the
                        ; scanline counter
SUB    t0, scanline      ; (2)
CLC
RR     t0               ; (1)
CLC
RR     t0               ; (1)
CLC
RR     t0               ; (1)
ADD    t0, #2            ; (2) Increase the brightness
MOV    t1, #COLOR_SKY_BASE ; (2) Set the sky's base color
ADD    t1, t0            ; (2) Modulate the sky intensity

; ***** Render each pixel of the scanline

MOV    t0, #182           ; (2) <t0> is the pixel counter
MOV    t3, #mount_line     ; (2) Start at the base of the
                        ; table base address
ADD    t3, bg_scroll      ; (2) Add background scroll
                        ; offset

sky_pixel_loop_1

; Put the next mountain Y pixel location in <t2>
MOV    M, #mount_line >> 8 ; (1) Point M:W at the table
MOV    W, t3               ; (1) Complete the address
IREAD
MOV    t2, W               ; (4) Read the table value
INC    t3                 ; (1) Move to the next pixel

; If the current pixel is above the mountain line, draw sky; otherwise,
; draw the mountain color
CJA    scanline, t2, sky_pixel_1 ; (4/6)
NOP
NOP
MOV    RE, #COLOR_MOUNT   ; (2) Draw a mountain color pixel
JMP    pixel_done_1        ; (3)

sky_pixel_1
MOV    RE, t1               ; (2) Render the sky
JMP    $ + 1                ; (3) Pad the jump above

pixel_done_1

DJNZ   t0, sky_pixel_loop_1 ; (2/4) Next pixel
DJNZ   scanline, draw_sky_loop_1 ; (2/4) Next scanline

```

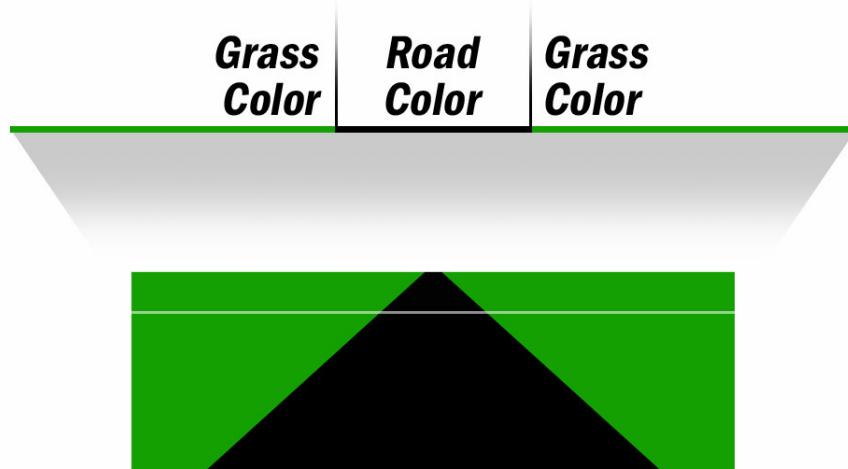
## 9.3 - Drawing the Race Track

The second major visual element of the racing engine demo is of course the race track itself. Like the mountain range, the track consists of many large, solid elements that would take massive amounts of memory if they were somehow based on raw bitmaps.

### 9.3.1 - A Procedural Race Track

A solution similar to the height map used in the last section once again comes to the rescue. If you look at the track, you can see that like the mountain range, it can be defined rather easily using only a couple of data points, rather than a complete 2D bitmap to cover the entire region of the screen. This time, break the track up into horizontal strips and analyze them as in Figure 9.3.

**Figure 9.3 – Analyzing horizontal strips of the racetrack.**



Notice that only two color changes are necessary to draw a black racetrack on a green grass background. The beginning of the scanline is always green, which is held until the left edge of the racetrack strip is reached, at which point the color switches to black. Black is then held until the right edge of the track strip is reached, at which point the color switches back to green until the end of the scanline. It would not be difficult to expand this process slightly to include two extra data points so the red and white stripes along the track can be drawn as well. The technique is the same either way.

The most important lesson to learn here is the concept of using **procedural** graphics instead of bitmaps. Since only the smallest bitmaps can fit economically within the limited address space of the XGS ME (without using the off-chip 128K SRAM, of course), large graphics can usually only be done with clever alternatives that use code to **generate** an image based on a set of rules, rather than data to **store** it. In

addition to the huge savings on memory, procedural graphics also often boast fluid movement and flexibility that would be difficult or impossible with standard bitmaps.

### 9.3.2 - Adding Perspective to the Track

It is now clear that a convincing track can be drawn using just a few data points that cause color changes at the proper times along each scanline. The question now, is, where do these data points come from? The track must appear narrow near the top of the screen and wide near the bottom to give the illusion of a three-dimensional perspective. This means our data points must be close together near the top of the screen and move apart from each other as they approach the bottom.

We could perhaps use a lookup table as we did with the mountain range, containing the slope of a diagonal line that matches the desired angle for the race track. This would work, but unlike the mountain range, the data needed to trace the slope of a diagonal is not unpredictable and arbitrary like the shape of mountains. It is in fact so predictable that a table is not needed at all; all that is needed is a tracking variable that follows a line's slope as the scanlines are drawn. Figure 9.4 demonstrates this idea.

**Figure 9.4 – Using tracking variables to follow the slope of a line as the track is drawn down the screen.**



The algorithm is actually quite simple:

- Initialize a 16-bit fixed-point tracking variable near the center of the screen, representing the distant end of the track.
- At each scanline, draw green from the left side of the screen to the left edge of the track. Use black to draw from the left edge of the track to the right edge. Use green again to draw from the right edge of the track to the right side of the screen.
- After drawing a scanline, perform a 16-bit fixed-point subtraction on the tracking variable to slowly move it away from the center of the screen, allowing the black (road) portion of each scanline to broaden as if it were coming closer to the viewer. Using fixed-point math instead of a simple 8-bit value allows the any slope to be used easily rather than forcing the track to follow a 45 degree angle.

Two separate tracking variables, of course, are used in the actual program so both the track, as well as the red and white border, can be drawn with reasonably correct perspective.

### 9.3.3 - Making Turns

Equally important to the perspective of the track is giving it the ability to bend as if its path was curving. This technique is actually rather simple, and simply requires the use of yet another lookup table to apply a predefined curve to the track scanlines as they're drawn from top to bottom.

#### 9.3.3.1 - Deforming the Racetrack

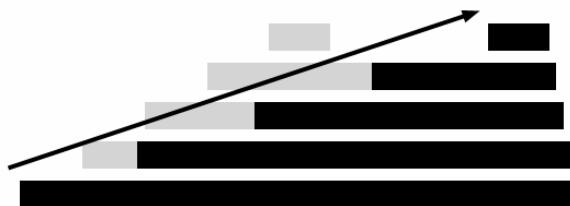
To understand this technique, imagine if a specific value were added to every datapoint on each scanline as they were drawn. If the value was positive, it would shift the entire track towards the right side of the screen. If it were negative, the track would shift to the left. This effect is illustrated in Figure 9.5.

**Figure 9.5 – Shifting the track left or right by adding an unchanging value to each scanline's data points.**



Now, imagine that another tracking variable was maintained and added to the location of every data point as each scanline was drawn, instead of the unchanging value in the last example. If this tracking variable increased after each scanline, the road would appear to veer off at an angle, as shown in Figure 9.6.

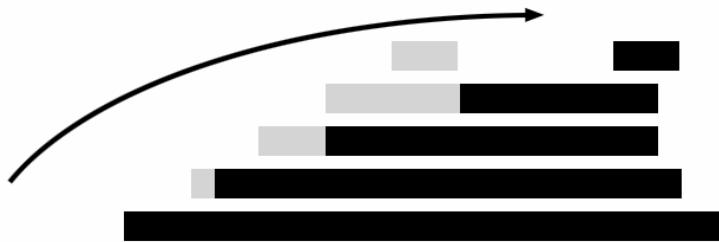
**Figure 9.6 – Bending the track along a diagonal by adding an increasing value to each scanline's data points**



Finally, imagine that instead of increasing the new tracking variable by an unchanging value at each scanline, thus bending the track linearly, the tracking variable instead increased in some non-linear

fashion, perhaps exponentially. This would cause the track to curve, because the rate of increase would be different at the top of the track than at the bottom. Figure 9.7 illustrates this.

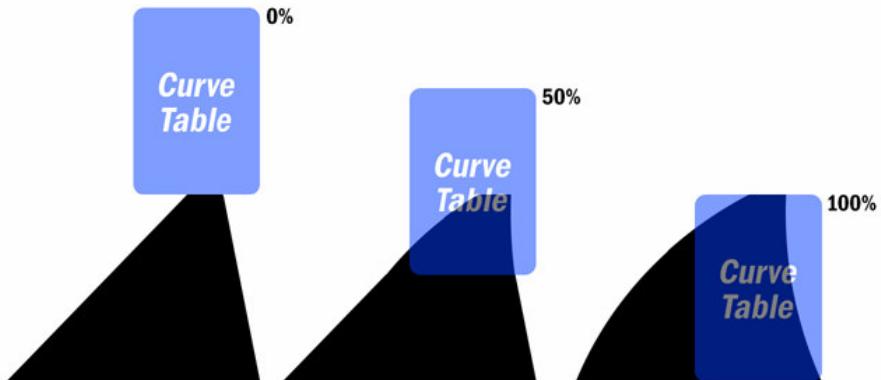
**Figure 9.7 – Curving the track by adding a non-linear tracking variable to each scanline’s data points.**



The trick is to step through a curve and apply it to the scanlines as they’re drawn in order to shift the track towards the desired shape. The only question remaining is how to gradually transition from a straight track to a track that moves in one direction or the other. Since the track can’t suddenly switch from perfectly straight to completely curved, there needs to be a way for the curve to be applied gradually.

Figure 9.8 presents a solution. Imagine that the curve data is stored in a lookup table such that each index in the table corresponds to one scanline on the track. If the entire table were applied such that each scanline were affected, the entire track would warp into a full curve shape. Now, if only the first half of the table were applied to the track, thus affecting only the top half of the road scanlines, the curve would be half as apparent and appear to be halfway down the road.

**Figure 9.8 – “Raising” and “lowering” a curve table into and out of the scanlines to control the perceived distance of the curve.**



In other words, by “lowering” and “raising” the table into and out of the track scanlines, the curve shape can appear to be moving towards or away from the player, just like a real turn on an actual road.

### 9.3.3.2 - Generating the Curve Data

As for the curve data itself, the easiest approach is simply to sample the appropriate segment of a sine wave. A C program was used to generate and format the following table based on the standard library's `sin()` function. The result was then simply copied and pasted into the racer demo source code.

```
curve_table

; Left turn curve
DW    $E3, $E4, $E5, $E6, $E6, $E7, $E8, $E9
DW    $EA, $EA, $EB, $EC, $ED, $EE, $EF
DW    $F0, $F0, $F1, $F2, $F3, $F3, $F4, $F5
DW    $F5, $F6, $F6, $F7, $F8, $F8, $F9, $F9
DW    $FA, $FA, $FB, $FB, $FC, $FC, $FD, $FD
DW    $FD, $FE, $FE, $FE, $FE, $FE, $FE, $FE
DW    $FE, $FE, $FE, $FE, $FF, $FF, $FF, $FF
DW    $FF, $FF, $FF, $FF, $FF, $FF, $00, $00

; Right turn curve
DW    $1D, $1C, $1B, $1A, $1A, $19, $18, $17
DW    $16, $16, $15, $14, $13, $12, $12, $11
DW    $10, $10, $0F, $0E, $0D, $0D, $0C, $0B
DW    $0B, $0A, $0A, $09, $08, $08, $07, $07
DW    $06, $06, $05, $05, $04, $04, $03, $03
DW    $03, $02, $02, $02, $02, $02, $02, $02
DW    $02, $02, $02, $02, $01, $01, $01
DW    $01, $01, $01, $01, $01, $01, $00, $00
```

Note that the table is split into two blocks. The first block is the curve used to form left turns, while the second is for right turns.

### 9.3.4 - Summary

All told, there were a number of main steps in implementing the track. The first was approaching the track in a manner similar to the mountain range—by recognizing that the track is really just defined by lines that mark the left and right sides of the track and the striped track border. Next, understanding how these lines can be made diagonal to produce a perspective effect for the track. Lastly, a curve table was applied to the track to varying degrees to create a flexible and easily controlled turn effect that works in both directions.

Coupled with the mountain range described in the previous section, the racing engine demo creates colorful, detailed, full screen graphics without the huge memory overhead usually associated with such results. This is the trick to writing graphical programs on the XGS ME—figuring out ways to produce complex and meaningful graphic effects, images and animations without resorting to bitmaps or other large data structures.

## 9.4 - Player Input

Player input is handled in a subroutine called `Handle_Input()`:

```

Handle_Input

; ***** READ THE JOYSTICK *****
CALL    @Read_Joystick

; ***** HANDLE STEERING *****
; Don't allow turning if the player isn't moving
BANK    BANK_GAME
MOV     t0, speed
BANK    BANK_MAIN
CLC
CSA    t0, #0
JMP    :skip_steering

; Turn left
SNB    data8.2
JMP    :skip_steer_left
BANK    BANK_GAME
CLC
CSA    wheel_angle, #1
JMP    :skip_steer_left
STC
SUB    wheel_angle, #2
:skip_steer_left
BANK    BANK_MAIN

; Turn right
SNB    data8.3
JMP    :skip_steer_right
BANK    BANK_GAME
STC
CSB    wheel_angle, #254
JMP    :skip_steer_right
CLC
ADD    wheel_angle, #2
:skip_steer_right
BANK    BANK_MAIN

:skip_steering

; ***** HANDLE GAS PEDAL *****
; Speed up
SNB    data8.0
JMP    :skip_speed_up
BANK    BANK_GAME
INC_BCD speed_bcd
BANK    BANK_MAIN
:skip_speed_up

; Slow down
SNB    data8.1
JMP    :skip_speed_down
BANK    BANK_GAME
DEC_BCD speed_bcd
BANK    BANK_MAIN
:skip_speed_down

RETP

End_Handle_Input

```

This fairly simple subroutine starts by calling `Read_Joystick()` to put the joystick status in the `data8` global. `Read_Joystick()` uses the fundamental joystick reading techniques discussed in the main eBook, ***Design Your Own Video Game Console***. The `data8` bits corresponding to the left/right and up/down joystick directions are then read to update the steering wheel and gas pedal, respectively.

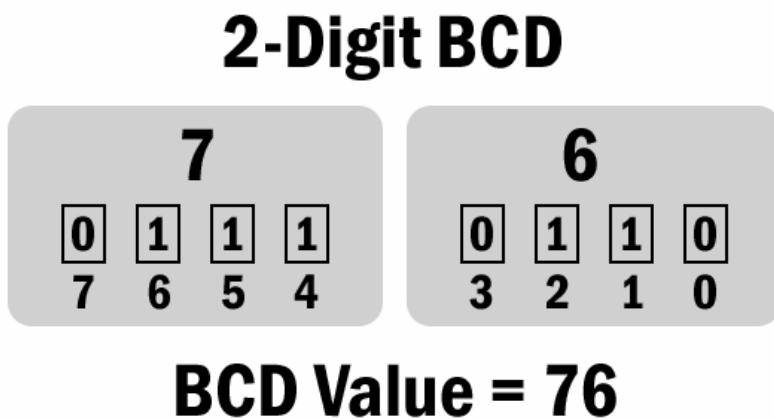
While the subroutine is very straightforward, there are a couple notes worth mentioning. First, turns cannot be made unless the user is moving. Before checking the status of the “steering” bits (left and right joystick directions), the speed is checked. If it is zero, the steering bits are ignored.

Second, when the user changes the speed with the up and down joystick directions, the `speed_bcd` variable is incremented and decremented, not the `speed` variable itself. `speed_bcd` is a special BCD (binary coded decimal) variable that stores the speed in a format that can be easily read and displayed on the screen using two digits. `speed` is a separate variable that determines how fast the track itself scrolls. At each frame, the speed value in `speed_bcd` is converted from BCD to straight binary and stored in `speed`.

## 9.5 - The BCD Speedometer

As stated above, input from the player first affects the `speed_bcd` variable, which is then converted to binary for storage in the real `speed` variable, simply because it's easier to increment an existing BCD value and convert to binary on the SX than it is to convert binary to BCD. Check out the `INC_BCD` and `DEC_BCD` macros in the source code for more information on how the increment and decrement works if you aren't already familiar with BCD. Check out Figure 9.9.

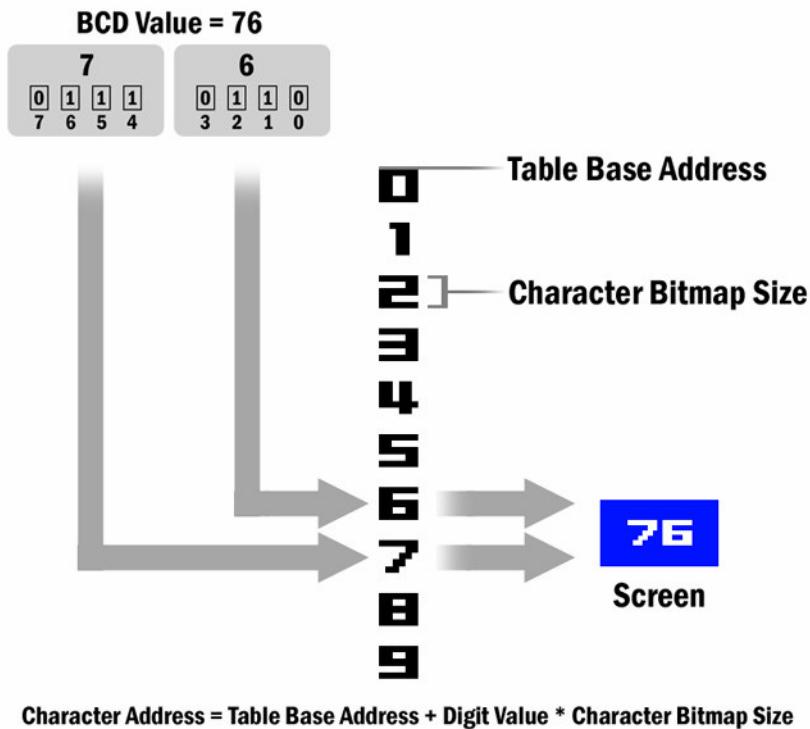
Figure 9.9 – Two BCD digits stored in a single byte.



A BCD speed variable is needed so a two-digit speedometer can be easily displayed in the upper-left corner of the screen. A BCD digit (0-9) is stored in 4 bits, which means the 8-bit `speed_bcd` variable can

store both digits needed for the speedometer output. The code responsible for drawing the speedometer then needs only to isolate each 4-bit digit and use it as an index into a table of digit characters. Figure 9.10 illustrates this process.

**Figure 9.10 – Converting a BCD counter into a 2-digit display with a 10-digit font table.**



This table is stored in program memory using the **DW** directive much like the lookup tables we've seen so far. The following is the declaration of this table. Only the first three characters are shown to save space:

```
num_font
num_0
    DW      %11111111
    DW      %11000011
    DW      %11011011
    DW      %11000011
    DW      %11111111
    DW      0, 0, 0

num_1
    DW      %00111000
    DW      %00011000
    DW      %00011000
    DW      %00011000
    DW      %11111111
    DW      0, 0, 0
```

```
num_2
DW      %11111111
DW      %00000011
DW      %11111111
DW      %11000000
DW      %11111111
DW      0, 0, 0
```

If you look closely, you can actually make out the bitmap image of each digit in the ones and zeroes.

## 9.6 - Expanding the Demo

While the demo produces a nice effect, it's not particularly entertaining. A considerable amount of work would have to be done in order to create a true game, but here's an outline of the three most pressing omissions as of now:

- **Cars**  
Needless to say, the most blatant missing elements are actual cars to drive. While you might be able to get away with pretending the player is in a first-person perspective from which his own car is not visible, there's still no excuse for not having competitors driving around. The biggest hurdle here is simply drawing a sprite of any reasonable complexity over the road, considering how dense the logic already is for drawing the road itself. Remember that in order to introduce new elements, especially those that overlap with existing ones, a considerable amount of logic must be added.
- **A Real Track**  
Currently, you "design" the track as you drive along it by controlling when it turns and bends. A real track map would not be difficult to implement; as long as your location within the track (which must be thought of as one-dimensional) is known, you can determine how close you are from the next turn and, depending on the distance and the direction of the turn, "lower" the proper curve table into the track scanlines.
- **A Proper Game Interface**  
Last but most certainly not least, a true game interface will complete the program. Think title screens, maybe even option screens, and perhaps screens for selecting cars or tracks.

## 9.7 - Summary

This case study has been a more conceptual, algorithmic discussion rather than an annotated source code dump. The complete racer engine demo is a fairly complex program and while it's certainly not beyond the grasp of most intermediate programmers, it's a lot of detail to stuff into one small chapter. So, instead of bogging you down with every conceivable detail, the purpose has been to give you an informative overview of what you can expect when writing more sophisticated programs, especially those that approach the level of a full game.

There are still numerous aspects of the demo that were not covered, such as the red and white stripes that move along the road, the way the mountains scroll, and some other details, but this case study was

not meant to be exhaustive program-wide documentation. Aside from the main eBook, ***Design Your Own Video Game Console***, of course, the next step in expanding your XGS ME skills would be to take a look at the racer engine demo source code itself, where all of the remaining details holding these concepts together come into view.

## Chapter 10: The SX Programming API

Included with the XGameStation Micro Edition's collection of software, tools and utilities, is the source code to a library of functions for communicating with the SX20 programmer unit on the board. Using this library of functions, you can write your own tools for managing the programming, reading, and configuring processes of the XGS ME.

This section is a brief guide to using this library. While the source code is highly commented and written to be understandable, the purpose of this section is not to explain how the library was written, but rather how it can be used in client programs.

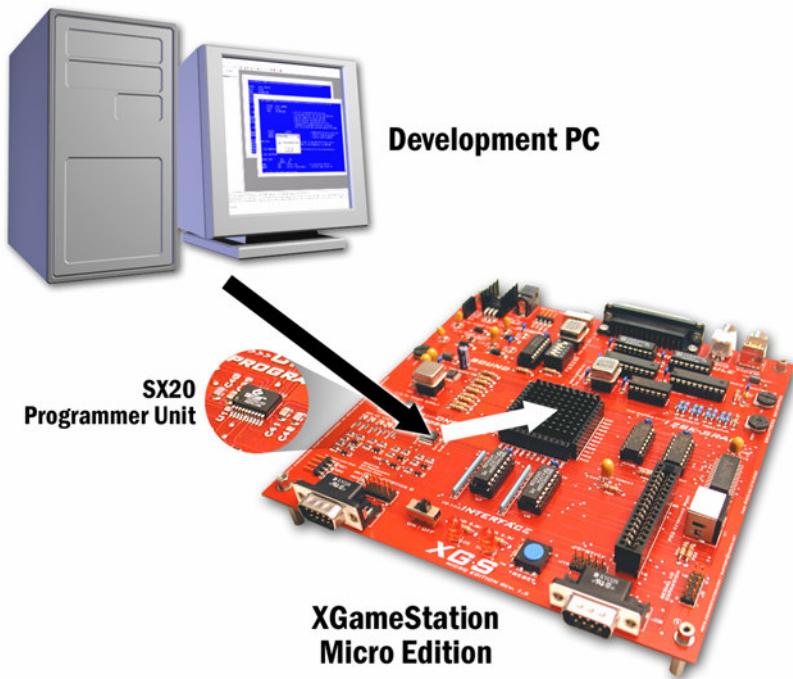
**NOTE**

This chapter assumes at least a basic familiarity with the various aspects of the XGameStation Micro Edition's processor, the SX52. If you find any references to the processor confusing, please refer to the complete SX52 coverage found in the main eBook, *Design Your Own Video Game Console*.

### 10.1 - Programming Architecture

The PC communicates with the XGS ME's SX52 processor via a secondary **slave** processor programmer unit rather than talking to it directly (see Figure 10.1). Like the main processor, the programmer unit is an SX chip; specifically, a scaled-down model known as the SX20. The programmer unit is programmed with firmware that implements a basic communication protocol and all of the necessary commands to write to, read from, and configure the SX52 during the programming process. The source code to this firmware, written in SX assembly language, is included and discussed briefly in a later section.

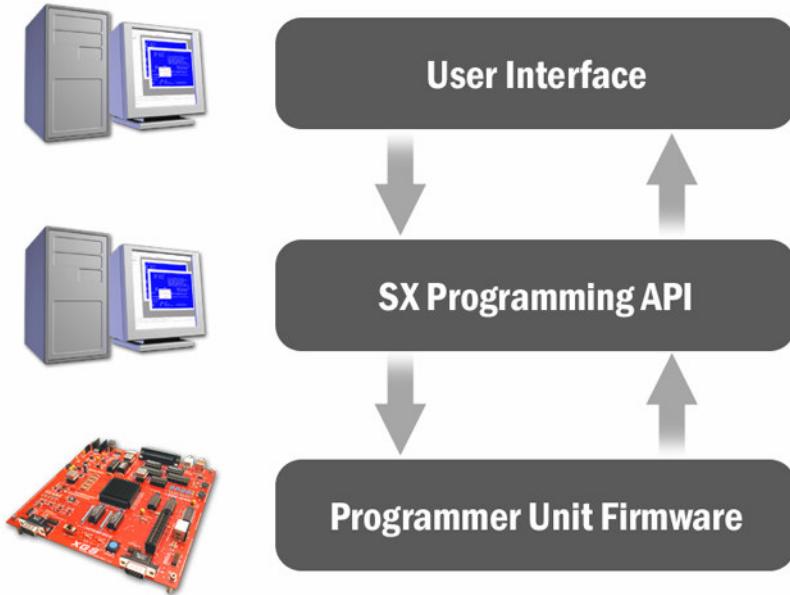
Figure 10.1 – The PC communicates with the XGS ME via the programmer unit.



From a software perspective, the process of programming the XGS ME consists of three levels, described below and illustrated in Figure 10.2:

- The User-Level Interface (on the PC)
- The SX Programming API (on the PC)
- The Programmer Unit Firmware (on the SX20)

**Figure 10.2 – The three-level software approach to programming the XGS ME.**



### 10.1.1 - The User-Level Interface (Highest Level)

The highest level is the user-level interface, which is how the user interacts with the programmer ultimately. The XGS Micro Studio IDE (discussed earlier in this guide) provides this interface in the form of a graphical Windows application that allows users to edit source code and manage the programming process.

### 10.1.2 - The SX Programming API (Middle Level)

XGS Micro Studio does not directly communicate with the programmer unit, however, as its primary goal is simply providing the user-level interface. The SX Programming API is a small set of functions that provide direct communication with the programmer unit. Separating this level into its own, self-contained API allows these functions to be easily used in the development of other, alternative tools and interfaces.

### 10.1.3 - The Programmer Unit Firmware (Lowest Level)

The last level of software involved in programming the SX52 is implemented as firmware running on the SX20, which communicates with the SX52 over a hardwired connection. Since the SX20 is a fast, self-contained microprocessor, it can talk to the SX52 in the fastest, most efficient and most reliable way

possible. This is why an intermediate programmer unit separates the SX52 and the PC; the PC *could* talk directly to the SX52, but due to the delicate timing issues required to program an SX chip, and the relatively inaccurate timing facilities available to PC programs, it's considerably less reliable.

The SX Programming API and the programmer unit communicate with one another via the PC's parallel port.

## 10.1.4 - The Full Communication Process

With the three levels of communication in mind, it should be clear how the XGS ME is ultimately programmed. The highest level, the user-level interface (such as XGS Micro Studio) allows the user to send commands to the middle level, the SX Programming API, which in turn relays the command to the lowest level, the SX20 programmer unit via the parallel port. The command is then implemented by the SX20 by communicating directly with the SX52.

### 10.1.4.1 - Writing Data

When writing data to the SX52 from the PC, it moves from the user-level interface, to the SX Programming API, then from the SX Programming API to the programmer unit via the parallel port, and finally from the programmer unit to the SX52 via their hardwired connection on the board.

### 10.1.4.2 - Reading Data

When reading data from the SX52 to the PC, it moves from the SX52 to the programming unit via their hardwired connection on the board, from the programmer unit to the SX Programming API via the parallel port, and finally from the SX Programming API to the user interface.

The remainder of this chapter will focus on the SX Programming API, and how it can be used to develop new user-level interfaces and tools for programming the XGS ME.

## 10.2 - Using the Library

The library is implemented in two files: **sx\_prog\_api\_01.cpp** and **sx\_prog\_api\_01.h**. Both files can be found on the **XGS ME Software CD** with the XGS Micro Studio IDE:

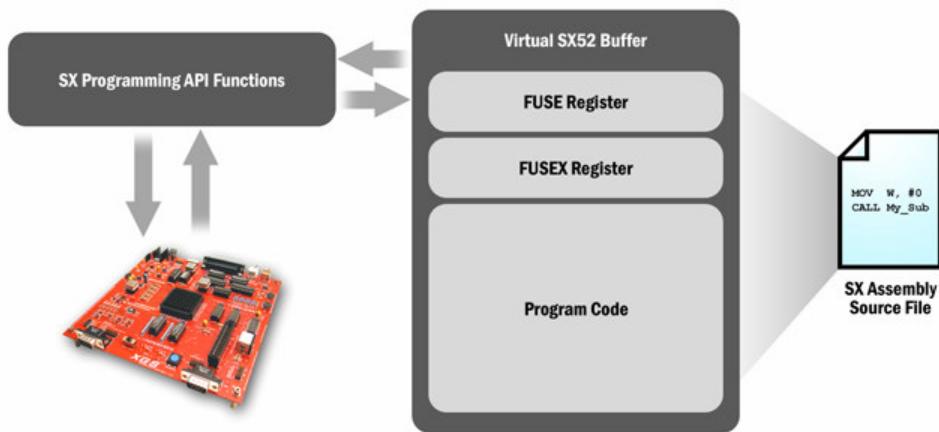
**XGS\_Micro\_Studio\_IDE\SX\_Prog\_API\**

Include **sx\_prog\_api\_01.h** and link **sx\_prog\_api\_01.cpp** with your project as you would any other library.

## 10.3 - Library Organization

The overall organization of the library and its functionality, which the following section will explain in full, is illustrated in Figure 10.3. The two main components of the library are the functions themselves, and the global `g_sx_device`, a structure designed to reflect the organization of data on the physical SX52 processor.

**Figure 10.3 – The overall organization of the SX Programming API library and its relationship with the programmer unit and SX52 processor.**

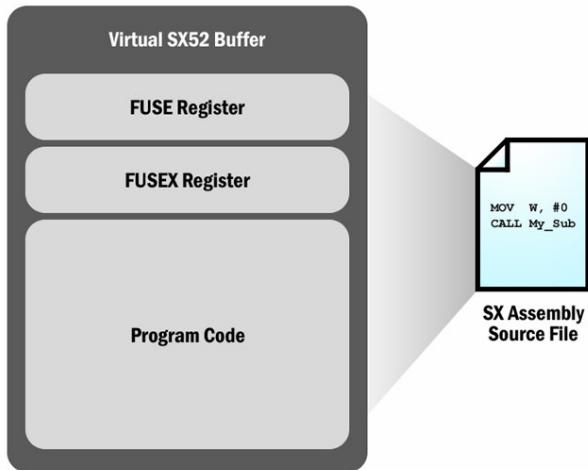


Since the SX52 consists primarily of its program memory and configuration registers, so too does `g_sx_device`.

### 10.3.1 - Loading Programs into the API

Whenever the library loads an assembled object program from a file, it is loaded into `g_sx_device`. The object code is inserted directly into the program memory, and the FUSE and FUSEX register settings are written to the structure's corresponding FUSE and FUSEX register fields. See Figure 10.4 for a visual reference.

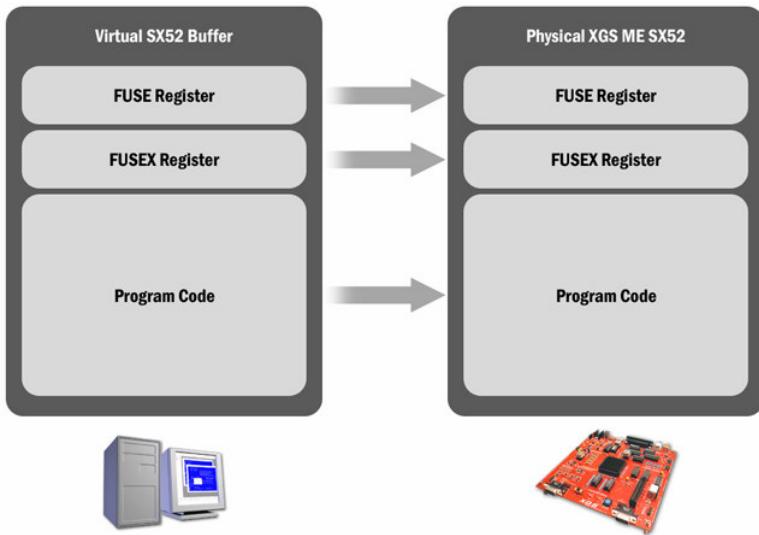
**Figure 10.4 – Loading source programs into the SX Programming API library.**



### 10.3.2 - Writing Programs to the Physical SX52

When it comes time to write the program to the SX52 itself, the fields once again correspond directly. The program memory is written to the real SX52's program memory, and the FUSE and FUSEX registers are copied as well. See Figure 10.5 for a visual reference.

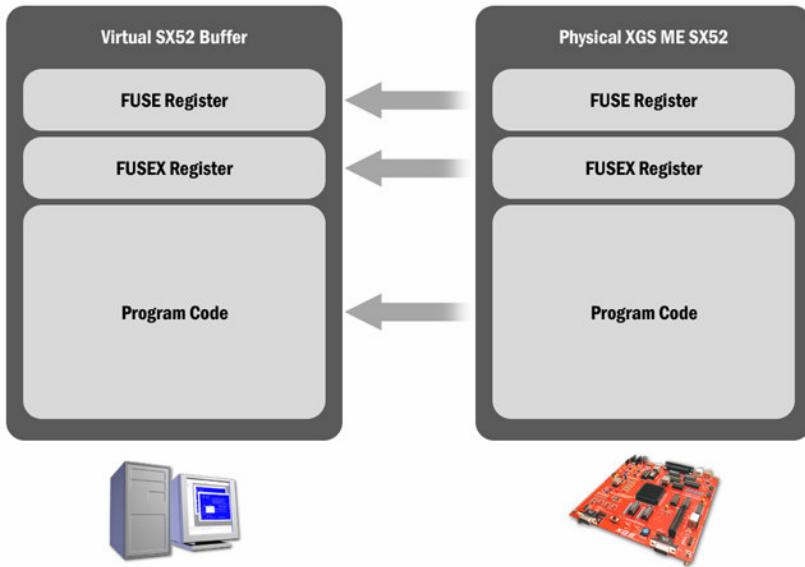
**Figure 10.5 – Writing to the SX52 with the SX Programming API library.**



### 10.3.3 - Reading Programs from the Physical SX52

Reading from the SX52 is treated like the reverse of the writing process. Again, the program memory is read from the SX52 and directly copied into `g_sx_device`'s program memory. The FUSE and FUSEX registers are copied from the SX52 into the appropriate register fields as well. Lastly, when reading, the SX52's read-only DEVICE register is copied into the DEVICE register field. See Figure 10.6 for a visual reference.

Figure 10.6 – Reading from the SX52 with the SX Programming API library.



When all data is written to and read from the same virtualized structure of the processor, it becomes easy to visualize where data is, and where that data is going, at all times.

## 10.4 - Library Reference

The following is a reference for the entire SX Programming API library from the perspective of the client program using it. This reference is broken up into categories like globals, functions, and so on.

### 10.4.1 - Globals

`g_sx_device`

A local image of the SX52 device. This structure stores data to be written to the SX52, or data read from the SX52. After reading almost any aspect of the SX52, for example, the appropriate members of `g_sx_device` will contain the information read, organized in the same way that information was found on the physical SX52.

## 10.4.2 - Data Types

### `SX_WORD`

Represents a 12-bit program word on the SX52. Used when passing program word values to and from the API functions.

### `SX_CMD`

Represents a 32-bit command or response packet passed between the programmer unit and the API functions. Used for returning response packets to the client program.

### 10.4.2.1 - The Programmer Unit Response Packet

Most functions that talk to the programmer unit return an `SX_CMD` value. This is because the response returned by the programmer unit is a 32-bit packet containing a success/failure flag, an error/status code, as well as 16 bits of data. Client programs can extract the relevant fields from this returned packet using the `SX_GET_RESPONSE_DATA`, `SX_GET_RESPONSE_STATUS` and `SX_CMD_FAILED` macros, explained below.

## 10.4.3 - The `SX_DEVICE` Structure

The `SX_DEVICE` structure is a representation of a physical SX chip's data. The relevant fields of this structure are as follows:

### `SX_WORD fuse, fusex, device;`

These fields of course correspond to the SX52's FUSE, FUSEX and DEVICE registers. After calling `SX_Read_FUSE()`, `SX_Read_FUSEX()`, or `SX_Read_DEVICE()`, these fields will contain the register values read. `fuse` and `fusex` are also initialized after loading an SXH object file.

### `SX_WORD program [ SX_PROGRAM_SIZE ];`

This array corresponds to the 4096-word program memory found on the SX52. After reading the SX52 with `SX_Read_Program()`, this array will contain the image of the device's program memory.

The remainder of this structure is for internal use and should not be manipulated by the client program.

#### 10.4.4 - Constants

```
SX_DEVICE_SX18  
SX_DEVICE_SX20  
SX_DEVICE_SX28  
SX_DEVICE_SX48  
SX_DEVICE_SX52
```

Represent the various SX processors supported by the SX Programming API. Only **sx\_device\_sx52** is needed for XGS ME development.

##### **SX\_PROG\_BUFFER\_SIZE**

Size, in 12-bit program words, of the programmer unit's onboard data buffer.

##### **SX\_PROG\_BUFFER\_BYTE\_SIZE**

Size, in bytes, of the programmer unit's onboard data buffer. Since two consecutive bytes are used to store each 12-bit program word, this is always twice the size of **SX\_PROG\_BUFFER\_SIZE**.

```
STATUS_SUCCESS  
STATUS_ERROR  
STATUS_TIMEOUT
```

Status codes that report the result of a given command. **STATUS\_SUCCESS** means the command was successful, **STATUS\_ERROR** means that the command failed for some reason, and **STATUS\_TIMEOUT** means the API lost contact with the programmer unit.

#### 10.4.5 - Functions

```
bool SX_Init ( int device, char *config_file );
```

Called at the start of a program to initialize the SX Programming API. Use the constant **SX\_DEVICE\_SX52** for the **device** parameter, and pass an empty string (*not* a null string) as the **config\_file** parameter. Call this function before calling **any** other API function. A false return value most likely means that the library could not establish a connection with the I/O ports, which is usually the result of another program being open at the same time that uses the same port abstraction layer. Close all programs that access the parallel port and try again.

```
bool SX_Shut_Down();
```

Called at the end of a program to shut down the SX Programming API.

```
void SX_Set_Device( int device );
```

Changes the device the API is interfacing with to `device`. Since XGS ME development will always be based around the SX52, this function does not need to be explicitly called by the client program. See the Constants section above for a list of device constants for use with this function.

```
SX_CMD SX_Init_ISP();
```

Initializes the processor's ISP (in-system programming) mode, allowing it to be programmed, read and configured by the programmer unit. This function *must* be called successfully before any commands are sent to the programmer unit.

```
SX_CMD SX_Shut_Down_ISP();
```

Shuts down the ISP mode set by `SX_Init_ISP()`. Call this before the end of a program (and before calling `SX_Shut_Down()`).

```
void SX_Erase_Buffer();
```

Clears the API's local program memory buffer. This function does not have any affect on the programmer unit or the SX52 processor. This function does not generally need to be called by the client program. Not to be confused with `SX_Erase_Device()`.

```
void SX_Map_Code_Runs();
```

Creates a map of the currently loaded programs code for use when writing the program to the SX52. This function is only necessary when not loading program code with the `SX_Load_SXH_File()` function (which calls it automatically).

```
bool SX_Load_SXH_File( char *filename );
```

Loads the SXH file (assembled SX52 object code) specified by `filename` into the program memory buffer. Also loads the program's desired FUSE and FUSEX register settings.

```
bool SX_Save_SXH_File( char *filename );
```

Saves the current program memory buffer to the SXH file specified by `filename`, for use with this API or any other SXH-compatible program (such as XGS Micro Studio and SX-Key). Also saves the current FUSE and FUSEX settings.

```
bool SX_Read_Program( int *progress, SX_CMD *result );
```

Reads a segment of the SX52's program memory into the local buffer. Returns the status by reference in the **result** parameter, and the amount of code read so far (as a percentage) in **progress**. The Boolean return value is true when the program has been fully read. This function is designed to be called iteratively to allow a real-time user interface to update a progress display and to allow the process to be cancelled before completion.

```
bool SX_Write_Program ( int *progress, SX_CMD *result );
```

Writes a segment of the local program memory buffer to the SX52's program memory. Returns the result status by reference in the **result** parameter, and the amount of code written so far (as a percentage) in **progress**. The Boolean return value is true when the program has been fully written. This function is designed to be called iteratively to allow a real-time user interface to update a progress display and to allow the process to be cancelled before completion.

```
SX_CMD SX_Erase_Device ();
```

Erases the SX52's program memory. Does not affect the local program memory buffer. Not to be confused with **SX\_Erase\_Buffer()**.

```
SX_CMD SX_Inc_Mem_Ptr ( int count );
```

Increments the SX52's internal memory pointer by **count**. Not needed by client programs for most purposes.

```
SX_CMD SX_Read ();
```

Reads a single program word from the SX52 at the location of its internal memory pointer, then increments that pointer. Only necessary for specialized purposes; to read the entire program memory, use **SX\_Read\_Program()**.

```
SX_CMD SX_Write ( SX_WORD x, bool verify );
```

Writes a single program word (**x**) to the SX52 at the location of its internal memory pointer, then increments that pointer. Set **verify** to true to verify that the value was written correctly (generally not necessary). Only necessary for specialized purposes; to write an entire program to the SX52, use **SX\_Write\_Program()**.

```
SX_CMD SX_Write_FUSE ();
```

Writes the contents of the API's local FUSE register buffer to the SX52 at the location of its internal memory pointer, then increments that pointer. Since the FUSE register can only be written after this pointer is reset, call this function before any other write function.

```
SX_CMD SX_Read_FUSEX ();
```

Reads the FUSEX register from the SX52 into the API's local FUSEX register buffer.

```
SX_CMD SX_Write_FUSEX ( bool verify );
```

Writes the FUSEX register to the SX52 from the API's local FUSEX register buffer. Set **verify** to true to verify that the register was written correctly (generally not necessary).

```
SX_CMD SX_Read_DEVICE ();
```

Reads the DEVICE word from the SX52 into the API's local DEVICE register buffer.

```
SX_CMD SX_Write_Buffer ( SX_WORD data, int addr );
```

Writes a program word (**data**) to the programmer unit's intermediate buffer at **addr**, allowing chunks of program memory to be built up before being written to the SX52's flash memory. Not generally needed by client programs.

```
SX_CMD SX_Read_Buffer ( int addr );
```

Reads the program word from the programmer unit's intermediate buffer at **addr**. This function can be used to quickly read the SX52's program memory by first buffering it in the intermediate buffer by calling **SX\_Stream\_Buffer\_In()**. Not generally needed by client programs.

```
SX_CMD SX_Stream_Buffer_Out ( int size );
```

Streams the contents of the programmer unit's intermediate buffer to the SX52's flash memory. By first filling this buffer with **SX\_Write\_Buffer()**, chunks of program memory can be written to the SX52 faster than they could be word-by-word using **SX\_Write()**. **size** is the number of 12-bit program words to stream out of the buffer; use **SX\_PROG\_BUFFER\_SIZE** to stream the whole buffer. Generally not needed by client programs.

```
SX_CMD SX_Stream_Buffer_In ( int size );
```

Streams the contents of the SX52's flash memory to the programmer unit's intermediate buffer. This buffer can then be read quickly with **SX\_Read\_Buffer()**, allowing a chunk of the SX52's program memory to be read faster than it could be word-by-word using **SX\_Read()**. **size** is the number of 12-bit program words to stream into the buffer; use **SX\_PROG\_BUFFER\_SIZE** to stream the whole buffer. Generally not needed by client programs.

## 10.4.6 - Macros

```
SX_GET_RESPONSE_DATA ( r )
```

Returns the data field of response packet **r**.

**SX\_GET\_RESPONSE\_STATUS ( r )**

Returns the status code field of response packet **r**.

**SX\_CMD\_FAILED ( r )**

Returns true if response packet **r** reports a failed command, false otherwise.

## 10.5 - Complete Library Demos

The following source code examples use the SX Programming API to program, configure, and read the XGS ME's SX52 processor. This illustrates the basis for writing alternative IDE's, programming utilities, and the like.

The source to this demo is available on the **XGS ME Software CD** along with the rest of the SX Programming API:

**XGS\_Micro\_Studio\SX\_Prog\_API\sx\_prog\_write\_demo\_01.cpp**

### 10.5.1 - Programming the SX52

The following demo programs the SX52 with the racing demo. In addition to writing the program memory itself, the FUSE and FUSEX registers must be written in order to fully write a functional program onto the system.

```
/*
 * SX Programming API Demo
 *
 * by Alex Varanese
 * 10.7.2003
 */
#include "sx_prog_api_01.h"

main ()
{
    // ***** INITIALIZE *****
    // First, initialize the library itself
    SX_Init ( SX_DEVICE_SX52, "" );

    // Now, attempt to initialize the programming mode
    if ( SX_CMD_FAILED ( SX_Init_ISP () ) )
    {
        printf ( "\nCould not initialize ISP mode.\n" );
        return ( 0 );
    }
    else
    {
        printf ( "\nISP mode initialized." );
    } // End if
}
```

```

// ***** PROGRAM/READ/CONFIGURE ****
***** PROGRAM/READ/CONFIGURE *****

// Load the racing demo as an example
if ( SX_Load_SXH_File ( "racer.sxh" ) != true )
{
    printf ( "\nCould not load source file.\n" );
    return ( 0 );
}
else
{
    printf ( "\nSource file loaded." );
} // End if

// Erase the device of any current data
if ( SX_CMD_FAILED ( SX_Erase_Device () ) == true )
{
    printf ( "\nCould not erase device.\n" );
    return ( 0 );
}
else
{
    printf ( "\nDevice erased." );
} // End if

// Write the FUSE register
if ( SX_CMD_FAILED ( SX_Write_FUSE () ) == true )
{
    printf ( "\nCould not write FUSE register.\n" );
    return ( 0 );
}
else
{
    printf ( "\nFUSE register written." );
} // End if

// Write the program

printf ( "\nWriting program" );

bool is_done = false;
while ( is_done == false )
{
    // Progress and result
    int P;
    SX_CMD r;

    // Write the next chunk of the program
    is_done = SX_Write_Program ( true, &p, &r );

    // Write the next chunk of the program
    if ( SX_CMD_FAILED ( r ) )
    {
        printf ( "\nCould not write to program memory.\n" );
        return ( 0 );
    } // End if

    printf ( ".", p );
} // End for

printf ( "\nDevice programmed." );

// Write the FUSEX register
if ( SX_CMD_FAILED ( SX_Write_FUSEX () ) == true )
{
    printf ( "\nCould not write FUSEX register.\n" );
    return ( 0 );
}

```

```

else
{
    printf ( "\nFUSEX register written." );
} // End if

// ***** SHUT DOWN *****
***** SHUT DOWN *****

// Shut down the ISP mode
if ( SX_CMD_FAILED ( SX_Shut_Down_ISP () ) )
{
    printf ( "\nCould not shut down ISP mode.\n" );
    return ( 0 );
}
else
{
    printf ( "\nISP mode shut down." );
} // End if

// Finally, shut the library down
SX_Shut_Down ();

return ( 0 );
} // End main ()

```

When the program is run, provided the source file is present and the XGS ME is turned on and properly connected, the following output should appear:

```

SX PROGRAMMER DEMO

ISP mode initialized.
Object code file loaded.
Device erased.
FUSE register written.
Writing program.....
.
.
.
Device programmed.
FUSEX register written.
ISP mode shut down.

```

As demonstrated here, the order in which a program is written to the SX52 is as follows:

- Erase device
- Write FUSE register
- Write program memory
- Write FUSEX register

After reading the function reference, the rest of the program should be self-explanatory. Each function is called to perform its action in sequence (initialization, loading source, programming, etc.) and branches off into either a success or failure block to track errors along the way. Throughout, the **SX\_CMD\_FAILED** macro is used to quickly check for errors.

## 10.5.2 - Reading the SX52

This demo performs the opposite function and reads the SX52. Along with the program memory, the FUSE and FUSEX registers are written as well. Lastly, and unlike in the previous example, the read-only DEVICE register is read.

To save space, the initialization and shutdown code has been omitted from the source listing.

The full source to this demo is available on the **XGS ME Software CD**:

**XGS\_Micro\_Studio\SX\_Prog\_API\sx\_prog\_read\_demo\_01.cpp**

```

printf ( "\n" );
printf ( "\nREADING CHIP..." );
printf ( "\n" );

// Read the DEVICE register
if ( SX_CMD_FAILED ( SX_Read_DEVICE () ) == true )
{
    printf ( "\nCould not read DEVICE register.\n" );
    return ( 0 );
}
else
{
    printf ( "\nDEVICE register read." );
} // End if

// Read the FUSE register
if ( SX_CMD_FAILED ( SX_Read_FUSE () ) == true )
{
    printf ( "\nCould not read FUSE register.\n" );
    return ( 0 );
}
else
{
    printf ( "\nFUSE register read." );
} // End if

printf ( "\nReading program" );

bool is_done = false;
while ( is_done == false )
{
    // Progress and result
    int p;
    SX_CMD r;

    // Write the next chunk of the program
    is_done = SX_Read_Program ( &p, &r );

    // Write the next chunk of the program
    if ( SX_CMD_FAILED ( r ) )
    {
        printf ( "\nCould not read from program memory.\n" );
        return ( 0 );
    } // End if

    printf ( ".", p );
} // End for

printf ( "\nDevice read." );

```

```

// Read the FUSEX register
if ( SX_CMD_FAILED ( SX_Read_FUSEX () ) == true )
{
    printf ( "\nCould not read FUSEX register.\n" );
    return ( 0 );
}
else
{
    printf ( "\nFUSEX register read." );
} // End if

// Print out registers
printf ( "\n" );
printf ( "\nFUSE:    $%03X", g_sx_device.fuse );
printf ( "\nFUSEX:   $%03X", g_sx_device.fusex );
printf ( "\nDEVICE:  $%03X", g_sx_device.device );
printf ( "\n" );

// Save the read file
if ( SX_Save_SXH_File ( "program.sxh" ) != true )
{
    printf ( "\nCould not save object code file.\n" );
    return ( 0 );
}
else
{
    printf ( "\nObject code file saved." );
} // End if

```

When the program is run, provided the XGS ME is turned on and properly connected, output similar to the following should appear:

```

SX PROGRAMMER DEMO

ISP mode initialized.

READING CHIP...

DEVICE register read.
FUSE register read.
Reading program.....
.....
.....
Device read.
FUSEX register read.

FUSE:    $FAA
FUSEX:   $7BF
DEVICE:  $002

Object code file saved.
ISP mode shut down.

```

The FUSE and FUSEX register printouts will of course change depending on the exact values set on your particular SX52. The DEVICE register will remain the same across all XGS ME systems, however.

## 10.6 - The Programmer Unit Firmware Source Code

The source to the programmer unit firmware is written in SX assembly language and is well-commented. While it is beyond the scope of this user guide to explain how this firmware was written, the code is pretty much self-explanatory and a programmer sufficiently comfortable with SX assembly should be able to figure it out without much trouble.

The firmware source code can be found on the **XGS ME Software CD** along with the XGS Micro Studio IDE:

**XGS\_Micro\_Studio\SX\_Prog\_Firmware\sx\_prog\_firmware\_01.SRC**

# Chapter 11: Reprogramming the Programmer Unit Firmware

In Chapter 10, we saw how the SX20 programmer unit is used to program and read the SX52 processor. Perhaps the most important part of the XGS ME's programming functions is the programmer unit's firmware, which implements the communication protocol as well as all of the commands themselves for manipulating the SX52's program memory and configuration registers. In order to program the XGS ME with the built-in programmer, this firmware must be present.

However, there may be times when updating or changing this firmware is useful. For example, if an updated or alternate version of the programmer firmware should become available, you would need some way to get it onto your SX20.

Even more interesting is the possibility of using the SX20 not as a programming unit, but as a slave processor that a program running on the SX52 can utilize to perform tasks in parallel with its own execution. There is a two-wire interface between the SX52 and SX20 available specifically for this possibility, as shown in Figure 11.1. The SX20's RB1 (pin 8) connects to the SX52's RA7 (pin 2). The SX20's RB2 (pin 9) also connects to the SX52's RB7 (pin 19). If you are using the SX-Key programmer instead of the built-in XGS ME programmer, the SX20 is not needed in its role as the onboard programmer unit and can be reprogrammed for your own purposes.

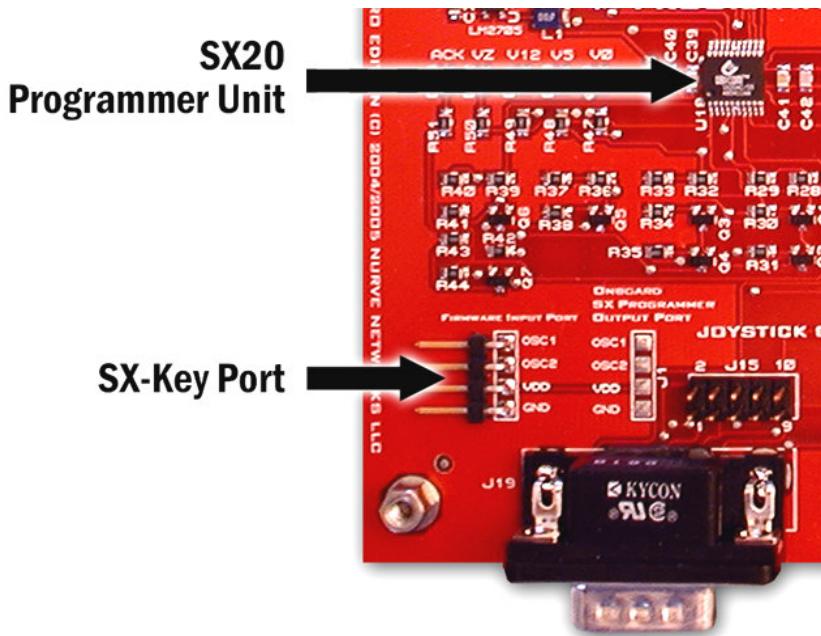
Figure 11.1 – The communication lines between the SX52 and SX20.



## 11.1 - Reprogramming the SX20

The SX20 is located in the lower-left corner of the board, as seen in Figure 11.2. You will notice that near the corner of the board, just above the joystick port, a second SX-Key port is hidden. This port works just like the SX-Key port at the back of the board, except it programs the SX20 instead of the SX52. This of course means that you'll need an SX-Key in order to reprogram it.

Figure 11.2 – The SX20 and its hidden SX-Key port.



One thing that may cause confusion is the **SYSMODE** switch used to control the SX52. This switch has *absolutely no effect* on your ability to program the SX20 via its own SX-Key port. The SX52 needs a three-way mode switch because it can be programmed by both the built-in programmer unit, as well as the SX-Key, and has a separate mode in which it runs using its own on-board oscillator. However, the SX20's programming interface is totally unrelated and does not need any special settings. In any mode, at any time, the SX20 can be reprogrammed.

**NOTE**

Like anything else, always reset the system after reprogramming the SX20 to ensure that everything is running off the same fresh reset.

Another important note is that the source code to the programmer unit firmware is included, and can be used to restore the programmer unit's functionality should it be intentionally changed or accidentally corrupted at any time. You can find the SX20 programmer unit source code here, along with the XGS Micro Studio IDE:

[XGS\\_Micro\\_Studio\SX\\_Prog\\_Firmware\sx\\_prog\\_firmware\\_01.SRC](#)

## 11.2 - An Example

The following is a step-by-step example of reprogramming and restoring the SX20 programmer unit.

### NOTE

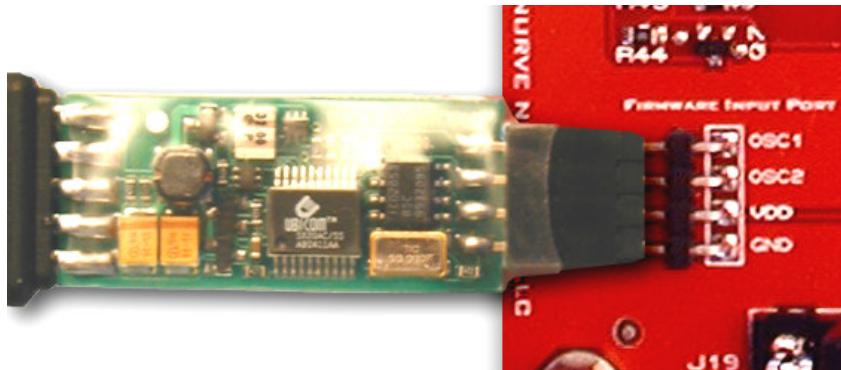
Before attempting either of the following tutorials, please take all of the usual steps to ensure that the system is properly powered-up and turned on. If you haven't already, make sure to read the Quick Start guides in Chapter 2 and the troubleshooting information in Chapter 5.

### 11.2.1 - Reprogramming the Firmware

#### 11.2.1.1 - Step 1 – Connecting the SX-Key to the SX20 Programming Port

Connect the SX-Key to the SX-Key port found near the SX20 as shown in Figure 11.3. Remember, the **SYSMODE** switch has no bearing on the SX20 and can be on any setting.

Figure 11.3 – Connecting the SX-Key to the SX20 via its programming port.



#### 11.2.1.2 - Step 2 – Load the New SX20 Firmware Program

Launch the SX-Key IDE and load the source code to the new firmware you'd like to program the SX20 with.

For the purpose of this example, load the program **sx20\_led.src** from the following directory:

XGS\_Micro\_Studio\SX\_Prog\_Firmware\

### 11.2.1.3 - Step 3 – Program the SX20

From the **Run** menu, select **Run** or press **Ctrl+R**. The progress window should appear and the code should be written to the SX20. Once the programming is complete, make sure to reset the system.

You should now see an animated LED pattern. This is your new SX20 program! Don't panic, however, because the next tutorial will explain how to restore the original programmer unit firmware.

### 11.2.2 - Restoring the Programmer Unit Firmware

#### 11.2.2.1 - Step 1 – Load the Programmer Unit Firmware

With SX-Key still open, load the file **sx\_prog\_firmware\_01.src** from the following location:

XGS\_Micro\_Studio\SX\_Prog\_Firmware\

From the **Run** menu, select **Run** or press **Ctrl+R**. The progress window should appear and the code should be written to the SX20. Once the programming is complete, make sure to reset the system.

**TIP**

Upon reset, you should notice the same animated LED pattern that originally ran before reprogramming the SX20. This is part of the programmer unit firmware and is included to let the user of the system know that the SX20's program memory is intact.

#### 11.2.2.2 - Step 2 – Test the Programmer Unit

Launch XGS Micro Studio and make sure the programmer unit is once again functioning normally by running any of the included graphical demos. Remember that when programming with the built-in programmer, the **SYSMODE** switch must be set to **PGM** mode while programming, and then set to **RUN** mode after programming so the program can execute. And of course, don't forget to reset the system after switching back to **RUN** mode.

The program you selected should now be running, meaning you have successfully restored the SX20 programmer unit.

# Chapter 12: Advanced Graphics: Tile Graphics Engine

## 12.1 - Brief Overview of XGS ME Programming

Graphics on the XGameStation Micro Edition is a complex and fascinating subject. It's rare that a game or graphics programmer these days has any need to generate or even understand a TV signal, but on the XGS ME, it's the only way to get anything on the screen. Furthermore, the system's intriguing mix of very small but fast on-chip memory, very large but slow off-chip memory, and ultra fast processing power provide an interesting set of constraints. On the one hand, a bloated, bitmap heavy display is going to be a formidable challenge, while procedurally generated, on-the-fly graphics techniques feel right at home.

## 12.2 - Different Approaches to Game Graphics

Throughout the years of game programming evolution, developments in both hardware and software have produced a multitude of different ways to put something on the screen. As is always the case, the problem is a matter of balancing graphical detail and color depth with processing speed and memory consumption. The following are a few commonly-used approaches to game graphics, both new and old.

### 12.2.1 - Vector Graphics

Since the earliest video games predated the common use of the raster displays we associate with computers today, game developers didn't have concepts like "pixels" and "sprites" in their vocabulary. Instead, their graphics were displayed using streaks of illuminated phosphor traced out by an electron gun controlled by their code. Rather than design graphics with colors, shapes and textures, imagery was composed entirely of a monochrome network of interconnected lines, or **vectors**. Perhaps the most famous example of a true vector game is **Spacewar!**, written in 1962 at MIT on the PDP-1. Check out Figure 12.1 for an example of another famous vector game, Atari's **Battlezone**.

**Figure 12.1 – A vector-based game consists of bright lines on a dark background, rather than colored pixels.**



What vector graphics lack in detail and color, they make up for in flexibility. Since an object rendered with vectors can be thought of as nothing more than a set of coordinates, it can be rotated, scaled, distorted and warped easily and will always appear equally crisp. Contrast this with pixel-based graphics, which can often degrade heavily as such transformations are applied.

Vector graphics are a possibility on the XGameStation Micro Edition, but pose a reasonable challenge. Firstly, the limited on-chip RAM of the system limits the complexity of your game objects, since only a modest number of multi-byte vectors can be stored economically. Second, transforming and manipulating vectors on a system with no native support for multiplication, division or 16-bit arithmetic (much less floating point arithmetic) means numerous math subroutines must be written. Lastly, and perhaps most importantly, a TV-driving video kernel that can determine if any given pixel lies within any vector's path within its allotted amount of clocks is non-trivial.

Regardless, vector displays have been successfully written for the XGS ME. Especially when it comes to nostalgia value, vectors are still an interesting consideration for game graphics.

## 12.2.2 - Framebuffer/Pixel Graphics

Framebuffer/pixel-based graphics are most commonly seen throughout the history of game development on the PC. PCs, usually far more expensive than video game systems, had larger, simpler memory

architectures that easily supported the storage of at least one full-screen buffer of pixel data. These buffers, called **framebuffers**, stored exactly what their names suggest—one complete frame of video data ready to be displayed on the output device (monitor). In other words, it was a memory based copy of what the user saw on the screen that was easily modifiable.

The beauty of pixel-based graphics is that *anything* is possible. Virtually everything you see around you can be represented on a 2D pixel display provided reasonable color depth and resolution is available. This means that game objects of all kinds as well as complex effects and multi-layered animation can be done. Think of a framebuffer as a blank canvas onto which anything can be painted. Once access to a framebuffer is provided, all that's really needed is enough memory to store bitmapped images of the game's graphics, as well as sufficient processing speed to copy and possibly transform those graphics to the framebuffer for display.

While a pixel-based framebuffer is ultimately the most powerful and flexible way to display game graphics, it suffers from two major drawbacks—memory and speed requirements.

### 12.2.2.1 - Framebuffer Memory Issues

Framebuffers are quite large by nature. To calculate the amount of memory needed to store a given screen, use the following formula:

$$\text{X\_Resolution} * \text{Y\_Resolution} * (\text{Color\_Bit\_Depth} / 8)$$

In other words, a display of 320x240 pixels wherein each pixel color is represented by one 8-bit byte, would require  $320 * 240 = 76,800$  bytes! On a system with 256 bytes of on-chip RAM (not counting the global registers), that's asking a lot. Even at an ultra-chunky resolution of 80x60 pixels (which would barely be usable anyway), that's still 4,800 bytes—4,544 too many.

Of course, the off-chip SRAM can store such a buffer. But then speed becomes the problem. How exactly does one disperse the loading of 76,800 bytes of SRAM over the video kernel's generation of the TV signal? This brings us to the next problem associated with framebuffer graphics.

### 12.2.2.2 - Framebuffer Speed Issues

Even if you have the memory for a framebuffer, you still have to get your game's graphics *into* that memory somehow. For many games, this may even entail drawing an entire, full-screen background *before* drawing the foreground objects, which basically means each individual pixel will be written to once, and possibly even multiple times. If a system's memory access is anything less than lightning fast, this may render framebuffer graphics too slow to be feasible (at least full-screen).

While the XGS ME is a very fast system with ultra-fast access to its on-chip RAM, the external 128K SRAM is only accessible via a limited I/O pin interface that brings with it significant overhead. Writing to a random location in SRAM requires shifting out a 12-bit page address that can mean around 100-200

clocks *per byte read or written*. Naturally, this is not sufficient in the event that 76,800 pixels need to be copied.

### 12.2.3 - Tile/Character-Based Graphics

Since the video game consoles of the 80's and early 90's couldn't affordably incorporate enough memory and bandwidth into their designs to provide programmers with usable access to a full-screen framebuffers, a different approach was taken. Instead of giving programs direct access to an array of pixels, they would describe graphics in terms of higher-level constructions like backgrounds, sprites and text. Graphics were drawn in small, square chunks called **tiles** or **characters**, then organized like puzzle pieces on the screen to create a final image. Programmers simply focused on the placement and contents of these tiny square image chunks, while specialized, high-speed hardware was responsible for translating this collage of tiles into a final full-screen image.

If, for example, a system's tiles were 16x16 pixels, and the resolution of the screen itself was 256x192 pixels, it would take only  $( 256 / 16 ) * ( 192 / 16 ) = 16 * 12 = \mathbf{192\ bytes}$  to cover the entire screen in tile-based graphics. The only real constraint placed on developers is that the total number of unique tile bitmaps available dictates the ultimate level of detail possible on the screen. For example, if the system only supported 64 unique tiles, certain tiles would have to be in more than one place on-screen in order to fill all 192 tile locations. Fortunately, most games feature a lot of "graphical redundancy", such as constant patterns and colors used in backgrounds, as well as a single character design being used for multiple on-screen characters.

For an example of how good tile graphics can look, despite being confined to a grid, check out the screenshot in Figure 12.2 from **Seiken Densetsu 3 (Secret of Mana 3)**, a Japanese-only action RPG for the Super Famicom (Super NES).

Figure 12.2 – A screenshot of a well-illustrated tile-based game.



#### 12.2.4 - Summary

All methods of displaying graphics are in some way a trade-off between quality and feasibility. Naturally, the choice for a given game's approach to graphics has a lot less to do with personal preference and a lot more to do with the limitations of the hardware.

### 12.3 - Tile Graphics on the XGS ME

All things considered, tile graphics offer the best combination of quality and feasibility on the XGameStation Micro Edition. The XGS Tile Graphics Engine balances the use of on-chip RAM, external SRAM and program memory to minimize overhead for the game that uses it, and allows the game logic to focus entirely on the tile buffer (which determines which tiles appear where onscreen) rather than the video signal itself. In fact, it's even written in such a way that game logic can be written without any regard for timing issues and the rest of the V-sync. We'll learn more about this in a moment.

**NOTE**

This chapter refers to **version 1.3.1** of the Tile Graphics Engine. Future versions will remain as compatible as possible, but differences between your version of the software and version 1.3.1 may exist. Be sure to check out the included **ReadMe.txt** file for more information.

The Tile Graphics Engine is not perfect, however, and suffers from a few limitations:

- No more than 64 unique tiles can exist in a given game. This limitation is usually not a problem, but worth keeping in mind.
- Objects must move along tile boundaries. Since tiles are 8x8 pixels, this can result in jerky movement for very slow or small movement. In most cases the effect is acceptable, however.
- Tiles cannot overlap. All tiles are drawn on tile boundaries, which means there's no way to shift a given object less than 8 pixels over another object, since 8 pixels is the minimum increment of movement allowed by the engine.
- Unless modified, the tile engine can only display its screen-sized tile map. In other words, unless you write your own changes into the tile engine's video kernel, only the tile map will be seen on the screen. This means if you wrote some sort of graphical effect of your own, it won't be able to share the screen with tile graphics.

Of course, the Tile Graphics Engine is ultimately a useful tool for developing games, for reasons such as the following:

- Complex, colorful graphics with a console-style look can be implemented easily.
- The engine includes a complete video kernel, allowing you to focus entirely on game logic and graphics while the engine handles the actual generation of a TV signal for you.
- A level of full-screen graphical detail can be attained at a fraction of the memory and processing time it might take using more "brute-force" methods.

## 12.4 - Using the XGS ME Tile Graphics Engine

The XGS ME Tile Graphics Engine is provided in the form of a "skeleton" source file into which you write your game. Games written into the source code are referred to as **client programs**. This section explains how the engine is used by a game.

The tile graphics engine can be found in the **Tile\_Engine** folder, and the skeleton source file can be found here:

**Tile\_Engine\xgs\_me\_tile\_gfx\_engine\_1\_2.SRC**

To understand and use the tile engine, you must understand the following concepts:

- Source Code Structure & Organization
- Framebuffers
- Tiles & Bitmaps
- Maps & Tile Attribute Tables
- Designing a Game Loop

### 12.4.1 - Source Code Structure & Organization

Because the Tile Graphics Engine is provided in the form of source code, you must understand the layout of this code in order to write your game into it.

The Tile Graphics Engine is implemented entirely within a single source file. The relevant sections of the source file are as follows:

#### 1. Constants

Constants used by the engine, as well as a few of general use that you may find handy (such as **TRUE** and **FALSE**, cardinal directions, etc.).

#### 2. Variables

Allocates a few banks of registers for internal use by the tile engine. Also defines numerous temporary registers that can be used by client programs.

#### 3. Macros

Defines both macros for internal use as well as “wrapper” macros that make it easier to call engine functions.

#### 4. Main

The video kernel and main execution loop for Tile Graphics Engine programs. This section of the source code is responsible for generating a video signal based on the tile map and

#### 5. Subroutines

Numerous subroutines for controlling the Tile Graphics Engine, as well as two very important ones—**Game\_Init()** and **Game\_Update()**, which we will discuss shortly.

### 12.4.2 - Writing Client Programs

Writing client programs that use the Tile Graphics Engine is very straightforward. Each aspect of a complete program is already outlined within the source code as needed for the engine itself, leaving you to simply “fill in” these areas with your own code. For example, as you define constants for your game, you can add them under the **Constants** heading already present in the file. The same goes for your own banks, variables and macros.

Defining your own constants, variables and macros is as easy as adding them under the appropriate heading within the source file. But where does the game code itself go? In the case of most of your game-specific subroutines, the answer, of course, is that they go anywhere you want. However, in the case of two very important subroutines, a place has already been reserved.

### 12.4.2.1 - Initializing the Client Program

Upon reset, both the client program and Tile Graphics Engine need to be initialized. The Tile Graphics Engine already runs its own initialization code, so all that's left is for your game to initialize itself as well. The `Game_Init()` subroutine is provided specifically for this task.

Inside the subroutine you may find some demo code already there, spelling out “**XGS TILE GFX**” on a colored background. Go ahead and delete this, making sure *not* to delete the `RETP` instruction at the end of the subroutine.

This subroutine is automatically called when the Tile Graphics Engine initializes, so it's the place to put all of your startup code for initializing variables, setting initial game states, and so on.

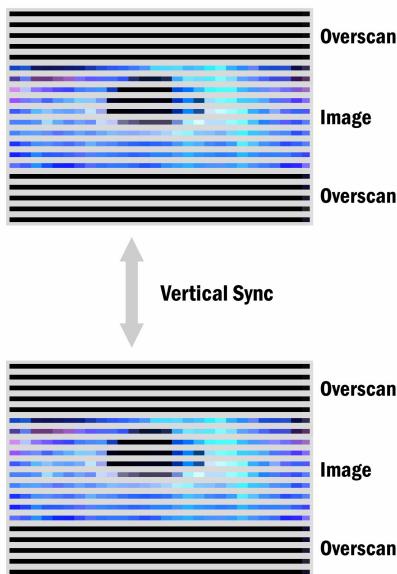
### 12.4.2.2 - Implementing Client Program Game Logic

Due to the nature of the XGS video kernel, XGS games are written around a single main loop that executes throughout the lifespan of the program, responsible for both generating the video signal and updating game logic on a per-frame basis.

Since the Tile Graphics Engine is already handling the video signal, all your game needs to do is take care of its own logic. This is accomplished by filling in the `Game_Init()` subroutine with whatever your game needs to do in a *per-frame basis* to keep itself updated.

`Game_Update()` is called during the **V-sync**, which is a period of approximately 60,000 clock cycles during which the video kernel is letting the TV catch up so it will be ready for the next frame. This period gives client programs ample time to perform their own game logic and keep the game in-sync with the screen. Figure 12.3 visualizes this.

**Figure 12.3 – Updating game logic in between each frame.**



Normally, code executing during the V-sync must keep track of how many clock cycles it takes to execute so that it can “pad” the rest of the sync with a delay that occupies the remainder of those approximately 60,000 clocks. This is because the V-sync period **must** take up that exact period of time, whether the code executing within it needs it all or not. So, if your game update code is finished within 10,000 clocks, you’ll need to sit in an empty loop for around 50,000 additional clocks to ensure the TV signal is not corrupted. Equally important, however, is that the game update code does not *exceed* the V-sync period either.

Fortunately for client programs, the Tile Graphics Engine keeps track of the execution of `Game_Update()` using the SX52’s RTCC (Real-Time Clock Counter). After the update function returns, the engine determines how much time is left and handles the delay automatically. This means that client programs don’t have to worry about timing at all while processing game logic, which is a huge relief! The only thing your game must still ensure, of course, is that it does not *exceed* the allotted period. As long as it remains under the limit, the Tile Graphics Engine will handle the rest.

### 12.4.3 - Framebuffers

At each iteration of the main loop, the Tile Graphics Engine redraws the screen using a **framebuffer** stored in the external SRAM. As described earlier, a framebuffer is a complete representation of the screen that is used directly to generate the video signal. In the case of the Tile Graphics Engine, there is no direct pixel access, so the framebuffer is composed of an array of tiles and their associated attributes.

The resolution of the Tile Graphics Engine screen is 128x192 pixels and the tiles themselves are 8x8 pixels. This means the screen is composed of a map of 16x24 tiles. Each tile is represented with three fields of data:

- **Bitmap Index**

Tells the engine which 8x8 graphic is to be drawn in this tile. Tile graphics are stored in program memory, and will be discussed in the next section.

- **Foreground Color**

The 8-bit foreground color of the tile.

- **Background Color**

The 8-bit background color of the tile.

The main framebuffer, referred to as the **visible buffer**, is automatically created at page 0 (the lowest address) of the SRAM. Since each tile requires 3 bytes, each row of tiles requires  $16 * 3 = 48$  bytes (3 SRAM pages). Each row is actually stored in 64 bytes, however, with an extra page used simply to pad the width of each row to a power of two for faster addressing. Multiply the row width by 24, the number of rows of tiles drawn down the screen, and you need  $64 * 24 = 1536$  bytes (96 pages) of SRAM to store one Tile Graphics Engine framebuffer.

#### 12.4.3.1 - Multiple Framebuffers

The default framebuffer is directly copied to the video signal at every frame, which means you've only got one copy of your game's background at any given time. This becomes a problem if you draw foreground objects over this background, since they will leave a trail of foreground tiles as they move around the screen.

To solve this problem, the Tile Graphics Engine allows multiple framebuffers to be created and copied to and from one another at any time. This way, the game background can be safely copied to a second framebuffer and recopied every frame to clear the previous foreground objects and allow those objects in their new positions to be redrawn without leaving a trail.

Use the **M\_COPY\_SCREEN** macro to copy a framebuffer from one 12-bit page address to another.

```
M_COPY_SCREEN source_hi, source_lo, dest_hi, dest_lo
```

The **source\_hi:source\_lo** arguments form a 12-bit page address of the source framebuffer, and **dest\_hi:dest\_lo** forms a 12-bit page address to the destination framebuffer. The framebuffer is then directly copied from the source to the destination. This operation has been optimized to be suitable for refreshing a background on a per-frame basis, but generally should not be called more than once inside **Game\_Update()**, as it begins to encroach too heavily on your remaining time.

## 12.4.4 - Tiles & Bitmaps

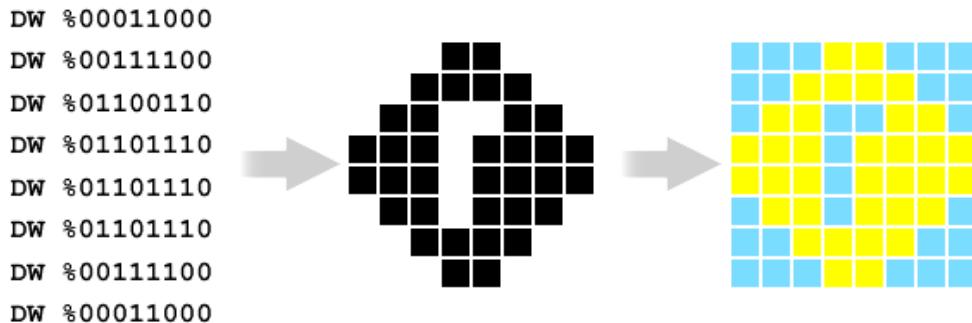
Tiles are of course the heart of the Tile Graphics Engine, and are the smallest graphical unit that can be manipulated. Each tile is represented by an 8x8 matrix of 1-bit pixels. A value of 1 represents the foreground color, while 0 represents the background color. Tiles are stored directly in program memory to ensure the fastest possible rasterization. To further speed up the decoding of the tile bitmaps, each row of 8 pixels is given its own program word, even though program words are 12 bits wide and can technically store 1.5 tile rows. Because of this, 8 program words are required to represent one tile.

As an example of entering and editing tile graphics, take a look at the following code:

```
DW  %00011000      ; Coin
DW  %00111100
DW  %01100110
DW  %01101110
DW  %01101110
DW  %01101110
DW  %00111100
DW  %00011000
```

This code fragment, taken from the XGS Bros. demo, represents a coin. Notice that the “foreground” area of the tile, the coin itself, is “drawn” as a solid circular formation of 1’s. The “background”, representing the empty space around the coin, is zeroed out. See Figure 12.4 for an example of how tile bitmaps relate to different foreground and background colors in the final image.

**Figure 12.4 - How tile bitmaps relate to different foreground and background colors in the final image.**



The Tile Graphics Engine supports up to 64 unique tile bitmaps starting at the **tiles** label (defined by default at \$F00, leaving room for exactly 64 tiles reaching the end of the SX52’s program memory).

**TIP**

Even though the engine supports up to 64 tile bitmaps, you don't need to use them all. Any space left unused after your last tile definition is free to use for your own purposes. If you're running low on program memory, don't hesitate to stuff some extra code in there!

Tiles can be directly drawn at any time to the visible buffer, after which they will immediately appear on the screen. There are numerous ways to add tiles to the visible buffer, using the following macros:

```
M_SET_TILE x, y, tile_index, fg_color, bg_color
```

Sets a tile with the specified bitmap, foreground color and background color at the specified X, Y location. Automatically clips to the framebuffer boundaries so you can safely draw tiles offscreen when necessary without overwriting other areas of memory.

```
M_SET_TILE_FG x, y, tile_index, fg_color
```

Works almost exactly like **M\_SET\_TILE**, but does not change the background color of the existing tile at the specified X, Y location. Use this macro to draw foreground objects that should not disturb the background. By leaving the background color intact, the appearance of sprite transparency is simulated to an acceptable degree in most cases. Also clips to the framebuffer automatically.

```
M_GET_TILE_INDEX x, y
```

This function does not draw a tile, but rather returns the bitmap index of the tile at the specified location. Useful for collision detection against the background map or detecting collisions with other game entities, like projectiles or power-ups.

```
M_FILL_ROW y, tile_index, fg_color, bg_color
```

Fills an entire row of tiles with the specified bitmap, foreground color and background color. Useful when setting up a background image, especially when creating vertical gradients or other vertical patterns.

```
M_FILL_SCREEN tile_index, fg_color, bg_color
```

Fills the entire screen with the specified bitmap, foreground and background color. Most useful for clearing the screen or laying down a full-screen background pattern.

**WARNING!**

Due to the relatively slow speed of the SRAM, try to minimize calls to tile drawing functions during **Game\_Update()**. There is enough time to draw a reasonable number of foreground objects, but it's not hard to exceed the time allocation and damage the TV signal. Also, be sure to use **M\_SET\_TILE\_FG** whenever possible as well, since it's roughly 1/3 faster than **M\_SET\_TILE**.

## 12.4.5 - Maps & Tile Attribute Tables

The macros listed in the previous section are good for setting up backgrounds and drawing foreground objects, but when it comes to plotting out intricate background shapes and patterns, especially those used to depict a detailed game environment, using macros to “hardcode” the desired map is not only labor intensive and difficult, but quickly eats up your program memory with bloated calls to subroutines.

Since most games need potentially detailed, full-screen backgrounds, another set of macros is available for storing full-screen maps in program memory and writing them to the visible buffer at runtime. This allows the entire map to be moved to SRAM in a single call. Furthermore, with the addition of **tile attribute tables**, the representation of even a full-screen map is very efficient and will not take a particularly large amount of program memory. Games can comfortably use three or even four full-screen maps while still having enough code space left over for full-featured game logic and tile bitmaps.

### 12.4.5.1 - Tile Attribute Tables

As discussed earlier, the tile screen is 16x24 and requires 3 bytes of data to represent a single tile. However,  $16 * 24 * 3 = 1152$  words is over a *fourth* of the available program memory on the SX52! One map alone would take a major chunk of program memory away from the game logic and tile bitmaps, and to make matters worse, many games need more than one map.

Clearly, maps must be stored in a much smaller format. One might next consider data compression as a solution, but this is hardly a viable option for three reasons:

- The compression ratio would vary wildly depending on the map data itself, meaning only some maps would benefit significantly. Others would compress poorly and in such cases, the problem would remain unsolved.
- The compressed data would not be human-readable or human-editable, meaning an entire separate utility would be required just to generate the map data declarations. This would add an extra step to every modification made to a map, and would make spontaneous tweaks and changes difficult.
- The code to decompress the map into SRAM could very well be complex enough to outweigh the size reduction of the map data in the first place.

Since the data cannot be compressed per se, it is simply represented in a simpler format. The depth of detail allowed in a given tile map is limited by confining each tile location in the map to a selection of 16 **pre-defined** tiles. This allows a single tile to be stored in 4 bits, instead of requiring 24 bits for an arbitrary bitmap index, foreground color, and background color. Since each program word on the SX52 is 12 bits wide, three tiles can be stored in a single word. This means that a 16x24 map can be represented with only  $(16 * 24) / 3 = 128$  words. However, since each 16-tile row now requires a total of 64 bits, and 12-bit program words do not divide evenly into 64, the row must be rounded up to 72 bits, or 6 program words. This brings the total for an entire map to  $6 * 24 = 144$  words, which is still a huge savings compared to the original requirement of 1152.

Maps are loaded from program memory into the visible buffer in SRAM with the **M\_LOAD\_MAP** macro:

```
M_LOAD_MAP map, tat
```

To call this macro, pass the label pointing to your map data like this:

```
M_LOAD_MAP my_map, tile_attr_table
```

What's **tile\_attr\_table**? This second label points to a **tile attribute table**, which is a 16-element table that **M\_LOAD\_MAP** uses to convert each 4-bit tile index found in the map to an actual 3-byte tile definition. Each entry in the table is a complete tile definition, consisting of a bitmap index, foreground color and background color.

Before designing a map, you create a corresponding tile attribute table that defines the tiles available to that map. Think of it as a "tile palette". You can create as many tile attribute tables as you want, allowing different maps to choose from different selections. While 16 unique tiles may not sound like much, it's enough to comfortably design most types of game environments, and since different maps can have their own attribute tables, the limitation is minimal. Besides, the ability to store full-screen maps with any level of detail in such an efficient and readable format is well worth it. And remember, this limitation does not effect which tiles you can draw to the screen *after* loading the map, such as your foreground objects.

### TIP

Even though a map loaded by **M\_LOAD\_MAP** is limited to 16 unique tiles, no one ever said you can't draw your own tiles to the screen on *top* of the map after it's loaded.

**tile\_attr\_table** points to the default tile attribute table, which looks like this:

```
tile_attr_table

DW    0, WHITE, BLACK, 0      ; 0
DW    0, WHITE, BLACK, 0      ; 1
DW    0, WHITE, BLACK, 0      ; 2
DW    0, WHITE, BLACK, 0      ; 3
DW    0, WHITE, BLACK, 0      ; 4
DW    0, WHITE, BLACK, 0      ; 5
DW    0, WHITE, BLACK, 0      ; 6
DW    0, WHITE, BLACK, 0      ; 7
DW    0, WHITE, BLACK, 0      ; 8
DW    0, WHITE, BLACK, 0      ; 9
DW    0, WHITE, BLACK, 0      ; A
DW    0, WHITE, BLACK, 0      ; B
DW    0, WHITE, BLACK, 0      ; C
DW    0, WHITE, BLACK, 0      ; D
DW    0, WHITE, BLACK, 0      ; E
DW    0, WHITE, BLACK, 0      ; F
```

By default, each entry is defined as tile bitmap zero with a white foreground and black background. As you design your map, you will modify the table to include other tiles, with the appropriate bitmap and colors. For example, here's the attribute table used to define the maps in the Venture demo. As you can see, it took all 16 available tile definitions:

```
tile_attr_table

DW      0,  COLOR_7 + 2, COLOR_7 + 3, 0 ; 0 - Grass
DW      2,  COLOR_14 + 3, COLOR_14 + 4, 0 ; 1 - Dirt Road
DW      2,  COLOR_14 + 0, COLOR_14 + 1, 0 ; 2 - Dirt Road Shadow (full)
DW      1,  COLOR_14 + 4, COLOR_14 + 1, 0 ; 3 - Dirt Road Shadow (diagonal)
DW      4,  COLOR_0 + 4, COLOR_0 + 2, 0 ; 4 - Pond
DW      6,  BLACK + 3, BLACK + 5, 0 ; 5 - Castle Brick (straight)
DW      7,  BLACK + 5, BLACK + 7, 0 ; 6 - Castle Brick (perspective left)
DW      8,  BLACK, BLACK + 3, 0 ; 7 - Castle Brick (perspective right)
DW      0,  COLOR_7 + 0, COLOR_7 + 1, 0 ; 8 - Grass Shadow (full)
DW      1,  COLOR_7 + 3, COLOR_7 + 1, 0 ; 9 - Grass Shadow (diagonal)
DW      9,  COLOR_7 + 3, BLACK + 7, 0 ; A - Castle + Grass Shadow (full)
DW      10, COLOR_7 + 3, BLACK + 3, 0 ; B - Castle + Grass Shadow (diagonal)
DW      11, BLACK, BLACK + 5, 0 ; C - Castle Window
DW      6,  BLACK + 1, BLACK + 3, 0 ; D - Castle Brick (shadow)
DW      12, BLACK, BLACK + 2, 0 ; E - Castle Gate
DW      3,  COLOR_0 + 4, COLOR_0 + 2, 0 ; F - Pond Rim
```

As general good practice, make sure to label your tiles as you define them with a comment so you don't forget which tile each entry refers to.

By default, the tile attribute table is allocated 64 bytes below the base of the tile bitmap data to ensure no space is wasted.

### 12.4.5.2 - Maps

Now that you've seen the definition of both tile bitmaps and tile attribute tables, let's take a look at the maps themselves. A default blank map is included in the engine source code at the **screen\_0** label:

```
screen_0

DW      $000,$000,$000,$000,$000,$000
```

Remember, even though 6 program words are needed to represent a full 16-tile row, the last two nibbles of the last word are not used. For example, to cover a row with tile index \$c, you would enter the following:

```
DW      $ccc, $ccc, $ccc, $ccc, $ccc, $c00
```

The organization of the map code directly maps to the tile arrangement on screen—left to right, top to bottom. This allows maps to be easily edited and read directly in the source code.

**TIP**

Use hex digits to fill in tile maps, as they never need a second digit and will easily keep the map data aligned visually within your code.

**NOTE**

Note that the last two nibbles remain \$0. While you could set them to \$c if you want, it would have no effect. It's best to leave these tiles zeroed to clearly define the edge of the map. If you think details you're drawing into the map are not being loaded properly, make sure you aren't accidentally drawing them into these unused last nibbles.

## 12.4.6 - Designing a Game Loop

At this point, you've seen how framebuffers, tiles, maps and tile attribute tables work together to create onscreen graphics. Now, using `Game_Init()` and `Game_Update()`, let's see how a complete game loop works.

### 12.4.6.1 - Initializing the Game

The first step, of course, is adding your initialization code to `Game_Init()`. In addition to your own game logic setup, an important step to take within this subroutine is copying your maps from program memory to the SRAM. Check out this example code:

```
M_LOAD_MAP      screen_0, tile_attr_table
M_COPY_SCREEN   #$00, #$00, #$00, #$60

M_LOAD_MAP      screen_1, tile_attr_table
M_COPY_SCREEN   #$00, #$00, #$00, #$C0

M_LOAD_MAP      screen_2, tile_attr_table
M_COPY_SCREEN   #$00, #$00, #$01, #$20
```

Each call to `M_LOAD_MAP` copies the program memory map into the visible buffer in SRAM. Each corresponding call to `M_COPY_SCREEN` copies the visible buffer to a non-visible, extra framebuffer

somewhere else in SRAM. Since this all occurs before the TV signal is generated, the user never sees these maps during their brief time in the visible buffer.

Note the source and destination addresses of the maps in each **M\_COPY\_SCREEN** call. In each case, the source address is **\$0000**. The destinations, **\$0060**, **\$00C0**, and **\$0120**, are all exactly one framebuffer's size (96 pages) apart.

## WARNING!

Remember, **M\_COPY\_SCREEN** expects the source and destination addresses as 12-bit *page* addresses, not 16-bit *byte* addresses. So if your framebuffer starts at **\$01D0**, pass **\$01D** to **M\_COPY\_SCREEN**.

The most important thing **Game\_Init()** can do to prepare the game for execution is loading the maps into SRAM. The rest of the function is your own game-specific initialization.

### 12.4.6.2 - Updating the Game

Once per frame, during the V-blank, the Tile Graphics Engine calls **Game\_Update()** and gives the game a chance to handle input, redraw the screen, update its own internal variables, and so on. The majority of this function is of course the implementation of your game logic, but there are a number of things it must do graphically that we will discuss here.

In the case of most games, a constant background is needed underneath the foreground objects. Since the drawing of a foreground object to the buffer is a permanent change (until that tile is again overwritten), moving objects will leave a “trail” of their previous locations unless they are somehow erased. Fortunately for us, we have all of our backgrounds already copied from program memory into secondary SRAM framebuffers, which means they're ready to be copied back into the visible buffer at any time.

Once again the **M\_COPY\_SCREEN** macro comes into play. This time, the source framebuffer is located at an arbitrary page address in SRAM, and the destination is the visible buffer located at address zero. The first order of business in **Game\_Update()** is usually repainting the background, like so:

```
M_COPY_SCREEN    #$01, #$20, #$00, #$00
```

The source background is located at **\$0120**, which was the third of three maps loaded in **Game\_Init()**. Of course, depending on the needs of your game, you may have multiple backgrounds and will most likely need to add some simple logic to determine which background should be drawn based on the game's state.

With the background repainted, the visible buffer is once again a clean slate upon which the foreground objects and interface can be drawn. In most cases, **M\_SET\_TILE\_FG** is the weapon of choice. This macro will draw onto the visible buffer but will not change the background color of the destination tile. By only changing the tile bitmap and foreground color, a convincing approximation of sprite transparency can

be achieved in most cases. Another advantage to `M_SET_TILE_FG` is that it's faster than `M_SET_TILE`, since it has one less attribute to modify, which means less SRAM access.

Despite the blazing speed of the SX52, you face two major bottlenecks inside `Game_Update()`. The first is the extremely slow speed of the external SRAM when compared to the SX52's internal RAM and program memory. This means that accessing the SRAM is an expensive operation that must be minimized as much as possible. Second, remember that `Game_Update()` is only allocated a timeslice of approximately 60,000 cycles. While this is definitely ample time for most games, it tends to go *very* quickly the more foreground objects you want to draw.

Copying the entire background map from SRAM is an expensive operation as well, but is relatively efficient; because the background map is drawn in a totally predictable, left-to-right, top-to-bottom fashion, numerous optimizations can be taken advantage of. While it should not be called more than once per frame, it runs about as efficiently as can be expected.

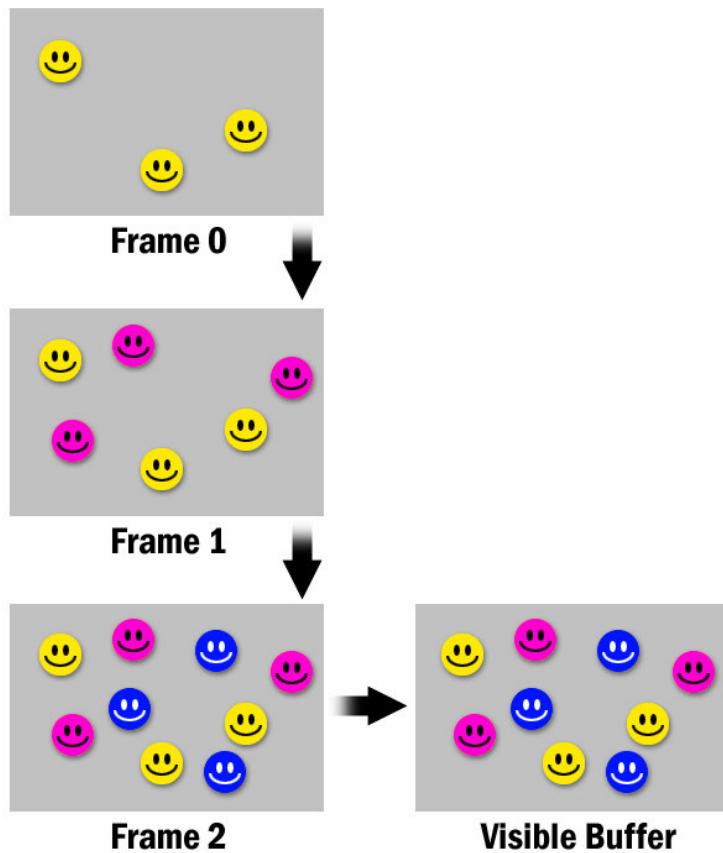
The random drawing of foreground objects, however, has no rhyme or reason and "thrashes" the SRAM as a result. Few optimizations can be made when the location of the next read or write is not known or even predictable, and as a result, foreground graphics are relatively inefficient. The general rule of thumb is to add foreground elements one by one and verify each time that the game still runs without exceeding its update timeslice. Once you see the signal begin to bend and degrade, you've hit the wall and must either remove foreground objects, or find a way to work around the limitation through some sort of clever re-organization.

#### 12.4.6.3 - Drawing Complex Graphics over Multiple Frames

One last advanced topic worth discussing is an approach to drawing complex graphics in which multiple real-time frames are used to build up one complete graphical frame. In other words, instead of drawing your entire game screen in a single call to `Game_Update()`, the subroutine could instead be written to work in phases that are tracked by a global counter.

Here's the general algorithm: In `Game_Init()`, a counter, called `frame_phase` for example, is initialized to zero. Each time `Game_Update()` is called, `frame_phase` is used to jump to a specific subroutine written to draw only a part of the final frame. The trick is that this drawing does *not* occur in the visible buffer, but rather another secondary buffer somewhere else in SRAM. After each call to `Game_Update()`, `frame_phase` is incremented. When `frame_phase` reaches its final value (say, phase 2 or 3), the completed frame is copied with `M_COPY_SCREEN` into the visible buffer and the phase counter is reset. Figure 12.6 illustrates this concept.

Figure 12.6 – Building a complex game frame over the course of multiple TV frames.



For example, phase 0 might start building the frame by redrawing the background as discussed earlier, then perhaps drawing a few of the foreground objects. Phase 1 would then layer on additional foreground objects, or just flesh out large objects that consist of multiple tiles. Lastly, phase 2 would overlay the interface, such as player health meters, inventory displays, or perhaps a map of the environment. As a final step, phase 2 would copy the completed frame to the visible buffer, reset the phase counter, and the process would begin again on the next frame.

Of course, like everything we've seen so far there's a trade-off to consider. First, all this extra graphical code may paint your game logic into a corner by consuming too much program memory. Second, the perceived frame rate of your game will be significantly reduced due to each game frame taking more TV frames to draw. This is because the Tile Graphics Engine redraws the screen 60 times per second. If, for example, your game requires two phases (two V-sync `Game_Update()` passes) to create one completed game frame, your game would now update at 30 frames per second. Of course, 30 FPS is more than adequate for any game, but at three or four phases, the frame rate drops to 20 and 15 FPS, respectively.

Around this level things can start to get choppy. Anything above four phases will result in a completely unacceptable frame rate for all but the slowest games.

#### 12.4.6.4 - Vertical Scrolling and Page Flipping

The beauty of the XGS ME's SRAM module is that it provides so much space for offscreen buffers. As of version 1.3 of the tile engine, a new way to use this memory is provided by a set of macros that allow the screen to be drawn starting from any page address, and allow all tile drawing functions (loading maps, drawing individual tiles, filling rows, etc.) to be performed relative to another page address.

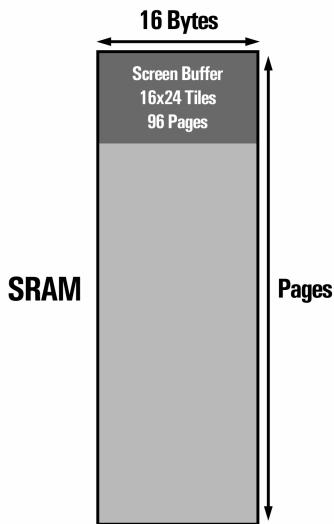
The page address at which the screen is drawn marks the start of a screen-sized block of memory called the **visible page**. By moving this address around, different parts of the SRAM can be displayed on the screen easily. The page address at which tile drawing functions are performed marks the start of another screen-sized block of memory called the **active page**. By moving this address around, different parts of the SRAM can be drawn to.

This new functionality enables a lot of cool things. Some ideas are presented in the following list:

- Vertical scrolling can be achieved easily by incrementing the visible page by one or two rows per frame. Of course, the area of memory being scrolled through must first be filled with the appropriate graphics. This is good for overhead racing games, or games centered around a vertical descent.
- One area of the memory can be displayed while another is drawn to by setting the active page and visible page to different addresses. This allows graphics to be prepared offscreen without affecting the visible buffer.
- Scrolling text consoles can be implemented easily, as seen in the text console case study later in this chapter.

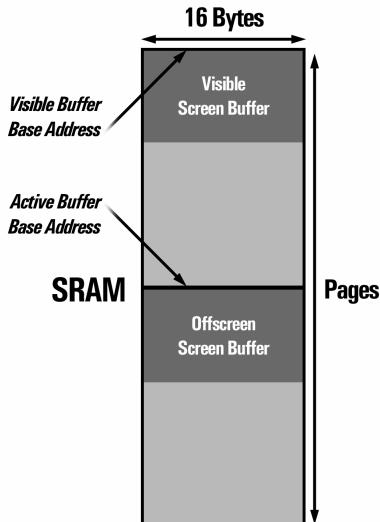
Understanding the visible and active pages is easier with a visual reference. Figure 12.7 presents the SRAM memory as it relates to the Tile Graphics Engine:

**Figure 12.7 – The SRAM memory layout as it relates to the Tile Graphics Engine.**



The visible page is an address representing the base of the screen-sized buffer that is directly copied to the screen. The active page points to a similar buffer, except this buffer represents the target of all tile drawing operations. Figure 12.8 illustrates these two concepts:

**Figure 12.8 – The visible page and active page offsets within the SRAM.**



Four variables in the video bank are allocated to track the base addresses of the visible page and active page. The base of the visible page is stored in `visible_page_hi:visible_page_lo`, and the base of the active page is stored in `active_page_hi:active_page_lo`. These variables are used on a per-frame basis, so changes to either have an immediate effect on the program. Changing the visible page causes an immediate change in the onscreen display, and changing the active page causes all subsequent calls to tile drawing subroutines to draw to a different part of the SRAM.

There does exist one gotcha to be mindful of at all times when working with the visible page and active page base addresses. The **visible page** base address is a **12-bit page address**, whereas the **active page** base address is a **16-bit byte address**. The reason for this has to do with the way the tile engine was written; the visible page can be more quickly accessed if its address is already expressed in a 12-bit page address form, whereas the active page is more efficiently used when it's already in a full 16-bit format. Mixing up the two will result in erroneous behavior, so keep the difference in mind.

Setting the value of either the visible or active page is easy with the help of the following two macros:

```
SET_VISIBLE_PAGE hi, lo
```

This macro sets the visible page to the **12-bit page address** specified by the `hi` and `lo` parameters.

```
SET_ACTIVE_PAGE hi, lo
```

This macro behaves exactly like `SET_VISIBLE_PAGE`, except it sets the active page. As an added convenience, this macro accepts the address as a 12-bit page address, just like `SET_VISIBLE_PAGE`.

Two more macros are provided as a simple convenience to increment either base address easily:

```
INCR_VISIBLE_PAGE
```

Increments the visible page by one row of tiles.

```
INCR_ACTIVE_PAGE
```

Increments the active page by one row of tiles.

## WARNING!

Remember, when directly manipulating the visible and active page base address variables, the **visible page** is a **12-bit page address**, and the **active page** is a **16-bit byte address**. However, as a convenience, when setting their values with the `SET_VISIBLE_PAGE` and `SET_ACTIVE_PAGE` macros, **both macros** accept the address as a **12-bit page address**.

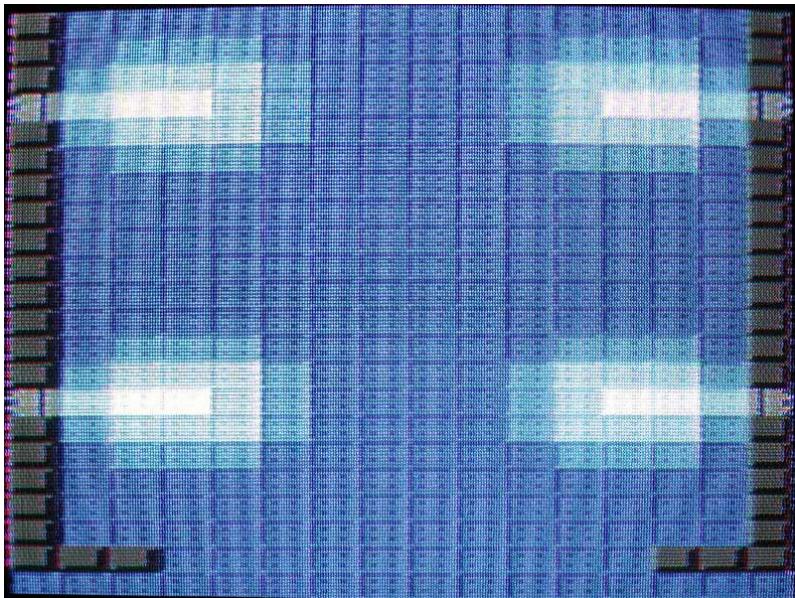
As an example of manipulating both the visible and active pages, check out the vertical scrolling demo, found here:

### Tile\_Engine\scroll\_test.srC

The demo first creates a long, vertical strip of connected screens in the SRAM. This is accomplished with a loop that loads a map into the active page, then increments the active page to the next screen down. When finished, this loop generates a long environment through which the user can scroll.

Next, during the game loop, the joystick can be used to scroll up and down throughout the environment created in the last step. When the joystick is pressed up, the vertical page base address is decremented. When the joystick is pressed down, the address is incremented. Check out Figure 12.9 for a screenshot.

**Figure 12.9 – The vertical scrolling demo.**



## 12.5 - Case Study: The Shooter Demo

The shooter demo was the first program written to demonstrate the Tile Graphics Engine, and is decidedly simple as a result. The objectives were as follows:

- A static background consisting of water, a mountainous terrain, and a gradient sky that fades upward to black.
- A non-functional example of a game display, consisting of a score readout and the number of ships left in reserve.
- A moving, player-controlled ship that is bound to the area between the left and right side of the screen, below the game display and above the mountain backdrop.

- One laser projectile that can be fired from the ship.

Check out figure 12.10 for a screenshot of the game.

Figure 12.10 – The shooter demo.



The source code to the shooter demo can be found here:

[Tile\\_Engine\shooter\\_1\\_0.src](#)

### 12.5.1 - The Graphics

Tile graphics form the foundation for any Tile Graphics Engine game. The shooter demo needs the following tiles:

- The player ship, facing two directions. Since an 8x8 ship would be a bit too small, we'll create a larger ship with multiple tiles. The ship used in the game is 24x8 pixels, or 3 tiles horizontally by 1 tile vertically.
- Numeric digits 0-9 and the word “**SCORE**” drawn in condensed letters over the span of 3 tiles. Condensing the word “**SCORE**” not only saves tile memory, but screen real estate as well. Lastly, a small 8x8 ship icon is used to represent the remaining ships on the right-hand side of the screen.

- A handful of terrain tiles for creating seamless water, a shoreline, a mountain range with varying mountain heights, and a fading sky background.

## 12.5.2 - Data Structures

Given the objectives listed at the beginning of this section, we can then outline the data structures necessary to keep track of everything. The following are the variable declarations used by the program:

```
; **** Game logic
ORG      BANK_GAME

ticks          DS     1           ; Tick count (increments once per frame)
p_x            DS     1           ; Player X, Y coordinates
p_y            DS     1
p_dir          DS     1           ; Player direction
l_x            DS     1           ; Laser X, Y coordinates
l_y            DS     1
l_dir          DS     1           ; Laser direction
l_active       DS     1           ; Is the laser currently active?
exhaust        DS     1           ; Ship exhaust color
```

**ticks** is updated every frame and is used for timing of various events. We'll see how this is used later. **p\_x**, **p\_y** and **p\_dir** track the location and the facing direction of the player. **l\_x**, **l\_y** and **l\_dir** track the location and direction of movement of the laser. **l\_active** is a flag that determines if the laser is visible onscreen, as well as whether or not another laser can be fired (since only one can be onscreen at once). Lastly, **exhaust** is an animation flag that toggles on and off, used to achieve the exhaust flicker effect.

## 12.5.3 - Initialization

The first game-specific code that will be automatically run by the Tile Graphics Engine is **Game\_Init()**. The game variables are initialized, setting the player to the center of the screen and facing to the right, turning off the laser (since nothing has been fired yet), and setting the exhaust flag to **\$FF** to make it visible (we'll see how this works shortly). The ticks counter is also reset, which will be important during **Game\_Update()**.

```
; ***** INITITALIZE GAME LOGIC *****
; BANK  BANK_GAME
CLR   ticks          ; Reset the tick counter
MOV   p_x, #6         ; Start at the center of the screen
MOV   p_y, #8
MOV   p_dir, #RIGHT   ; Start facing right
MOV   l_active, #FALSE ; The laser has not been fired
MOV   exhaust, #$FF    ; The ship exhaust is visible
```

Next, the background is set up. Normally, backgrounds are best implemented with maps, tile attribute tables, and the **M\_LOAD\_MAP** macro, but in this demo was written before these features were available. It remains as an example of how backgrounds can be manually assembled through code instead of data, should you ever decide to take this route. Here's the code for setting up the background:

```

_BANK   BANK_VIDEO

; Clear background
M_FILL_SCREEN #0, #WHITE, #BLACK

; Water/terrain/mountain/sky background
M_FILL_ROW    #23, #20, #COLOR_0 + 6, #COLOR_0 + 4
M_FILL_ROW    #22, #20, #COLOR_0 + 7, #COLOR_0 + 4
M_FILL_ROW    #21, #19, #WHITE, #COLOR_0 + 4
M_FILL_ROW    #20, #18, #WHITE, #COLOR_12 + 2
M_FILL_ROW    #19, #15, #COLOR_12 + 2, #COLOR_12 + 4
M_FILL_ROW    #18, #16, #COLOR_12 + 3, #COLOR_14 + 5
M_FILL_ROW    #17, #17, #COLOR_14 + 5, #COLOR_14 + 3
M_FILL_ROW    #16, #17, #COLOR_14 + 3, #COLOR_14 + 1
M_FILL_ROW    #15, #17, #COLOR_14 + 1, #BLACK

; Distribute randomly sized mountains over mountain range row
M_SET_TILE   #3, #18, #21, #COLOR_12 + 3, #COLOR_14 + 5
M_SET_TILE   #4, #18, #22, #COLOR_12 + 3, #COLOR_14 + 5
M_SET_TILE   #12, #18, #22, #COLOR_12 + 3, #COLOR_14 + 5
M_SET_TILE   #7, #18, #21, #COLOR_12 + 3, #COLOR_14 + 5

; **** Score display icon
M_SET_TILE   #0, #0, #1, #WHITE, #COLOR_2 + 2
M_SET_TILE   #1, #0, #2, #WHITE, #COLOR_2 + 2
M_SET_TILE   #2, #0, #3, #WHITE, #COLOR_2 + 2

; **** Score
M_SET_TILE   #4, #0, #4, #WHITE, #BLACK
M_SET_TILE   #5, #0, #7, #WHITE, #BLACK
M_SET_TILE   #6, #0, #5, #WHITE, #BLACK
M_SET_TILE   #7, #0, #8, #WHITE, #BLACK

; **** Life display
M_SET_TILE   #13, #0, #23, #COLOR_14 + 7, #BLACK
M_SET_TILE   #14, #0, #23, #COLOR_14 + 7, #BLACK
M_SET_TILE   #15, #0, #23, #COLOR_14 + 7, #BLACK

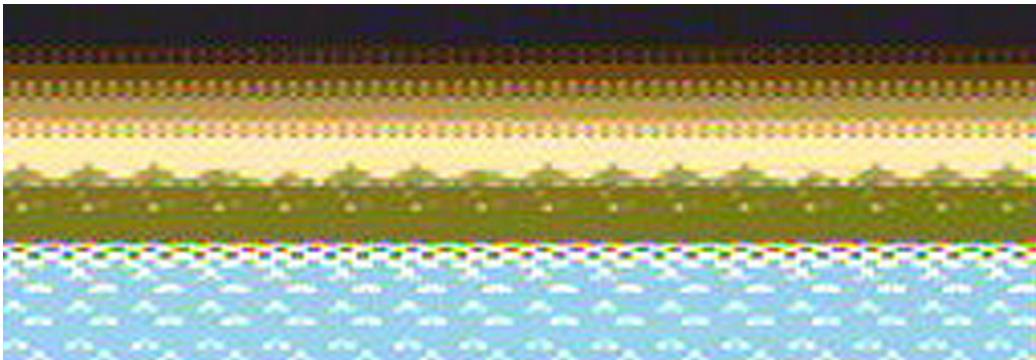
```

First and foremost, note that the first line in this code sets the bank to **BANK\_VIDEO**. This is important since *all* tile drawing macro rely on the video bank. Calling any tile drawing macro, such as **M\_SET\_TILE** and **M\_FILL\_ROW**, require the video bank be active. Once the calls are finished the bank is of course once again under your control.

Since the background is primarily horizontal, the **M\_FILL\_ROW** macro is used to draw the brunt of it. **M\_SET\_TILE** is used on the mountain tile row to insert some varying mountain heights that give it a more natural look. Lastly, **M\_SET\_TILE** is again used to manually construct the “SCORE” icon, followed by the score itself, and the life display on the other side of the screen.

Check out Figure 12.11 for a screenshot of the game background.

Figure 12.11 – The shooter demo background.



#### 12.5.4 - Updating the Game

At this point, the game has a background and a non-functional example of a typical game display. Now it's time to add some functionality. Each time `Game_Update()` is called, the game is updated by performing the following tasks:

- Update the tick counter for use in timing.
- Erase the on-screen game objects (but not the background) so they can be moved without leaving a trail. In the case of the shooter demo, only two game objects can exist on the screen at any one time—the ship, and if active, the laser.
- Input is read from the joystick, allowing the player to move and shoot. As the player moves, they are clipped to the screen region above the background and below the display. The left and right sides of the screen are boundaries as well. Lastly, the player can only fire if the last laser fired has already moved off-screen.
- The laser, if active, is moved in whichever direction the player was facing when it was fired.
- The game objects are drawn. In the case of the laser, it is only drawn if it is active. Lastly, the ship's exhaust is drawn if the `exhaust` flag is nonzero. By toggling this flag each frame, the exhaust will flicker on and off. Depending on the direction the player is facing, one of two sets of player ship tiles is used.
- Let's take a look at each of these tasks in more detail.

##### 12.5.4.1 - Erasing the Game Objects

In the case of the shooter demo, the background area the player over which the player can appear is solid black. This means that nothing needs to be redrawn each frame, so instead, the player (and laser, if

visible) are erased with black tiles using `p_x`, `p_y`, `l_x`, and `l_y`. Here's the code to erase the player and exhaust:

```
_BANK BANK_GAME
MOV t4, p_x
MOV t5, p_y
_BANK BANK_VIDEO
M_SET_TILE t4, t5, #0, #BLACK, #BLACK

_BANK BANK_GAME
MOV t4, p_x
MOV t5, p_y
_BANK BANK_VIDEO
INC t4
M_SET_TILE t4, t5, #0, #BLACK, #BLACK

_BANK BANK_GAME
MOV t4, p_x
MOV t5, p_y
_BANK BANK_VIDEO
ADD t4, #2
M_SET_TILE t4, t5, #0, #BLACK, #BLACK

_BANK BANK_GAME
MOV t4, p_x
MOV t5, p_y
_BANK BANK_VIDEO
ADD t4, #3
M_SET_TILE t4, t5, #0, #BLACK, #BLACK
```

### 12.5.4.2 - Handling Player Input

Next, the player's joystick is read to determine if the ship should move, or if a laser should fire. The first step is calling `Read_Joysticks()`. `Read_Joysticks()` is called once every four frames, instead of each frame, to prevent the player from moving too quickly.

```
; Check for input every 4 ticks
MOV t0, ticks
AND t0, #3
CJNE t0, #0, :skip_handle_input

; Read the joysticks
CALL @Call_Read_Joysticks
```

The joystick is read, and an 8-bit vector of each button's status is stored in `t0`. Check out table 12.1 for a listing of each bit's value.

**Table 12.1 – Bit meanings in the joystick status vector.**

Bit	Value
0	Up Direction
1	Down Direction
2	Left Direction
3	Right Direction
4	Button

**NOTE**

Notice that the `Read_Joysticks()` subroutine appears in the code as a call to `Call_Read_Joysticks()`. `Call_` is appended to all subroutines because they are called through a jump table rather than directly. `Read_Joysticks()` and `Call_Read_Joysticks()` refer to the same subroutine, but only `Call_Read_Joysticks()` should be used in code.

All joystick status bits are **active low**, meaning they're 0 when the corresponding button is down, and 1 when the corresponding button is up. So while you may expect the a status bit to be set when the button is being depressed, the opposite is true.

The following code handles the player's directional movement based on these principals:

```
; Save current location for bounds checking
MOV      t2, p_x
MOV      t3, p_y

; Check each direction
SB       t0.2          ; Move player left
DEC     p_x
SB       t0.3          ; Move player right
INC     p_x
SB       t0.0          ; Move player up
DEC     p_y
```

```

SB      t0.1                      ; Move player down
INC    p_y

; Restore old locations if bounds crossed
CJA    p_y, #0, :skip_clip_y_min ; Check minimum Y bound
MOV    p_y, t3

:skip_clip_y_min
CJB    p_y, #15, :skip_clip_y_max      ; Check maximum Y bound
MOV    p_y, t3

:skip_clip_y_max
CJB    p_x, #16 - 3, :skip_clip_x      ; Check X bounds
MOV    p_x, t2

:skip_clip_x

; Change the player's facing position if necessary

JB      t0.2, :skip_face_left
MOV    p_dir, #LEFT
:skip_face_left
JB      t0.3, :skip_face_right
MOV    p_dir, #RIGHT
:skip_face_right

```

Directional movement is handled in a series of discreet steps. First, the location *before* the movement is stored in **t2** and **t3**. This way, if the player ends up moving somewhere beyond the boundaries of the screen, his original location can be restored. In the case of vertical movement, the player's location is directly compared against the bottom of the display (where the score and lives appear) and the top of the background image. Horizontally, collision differs in one subtle way—because the X location is assumed to be unsigned, moving past tile 0 wraps around to 255, which means that in both cases of horizontal bounds crossing, the player's location will be greater than 15 (the highest possible tile location on the X axis).

The laser is next, and is fired using a simple algorithm. If a laser is not currently on the screen (based on the **l\_active** flag), the player is free to fire. In this case, the laser's position is set to just beyond the nose of the player's ship, and the laser's direction is set to the direction the player is facing. If a laser is already on the screen, nothing happens. Here's the code:

```

; Check for laser fire if one is not already active
CJE    l_active, #TRUE, :laser_fire_done

; Is a laser being fired?
JB      t0.4, :laser_fire_done

; Activate the laser
MOV    l_active, #TRUE

; Set the laser's direction and Y location
MOV    l_dir, p_dir
MOV    l_y, p_y

; Which direction is the player facing?
CJE    p_dir, #LEFT, :fire_laser_left

; Right, so place the laser two tiles to the right of <p_x> to clear
; the ship itself
MOV    l_x, p_x
ADD    l_x, #3
JMP    :laser_fire_done

:fire_laser_left

```

```

; Left, so place the laser at the player's X position
MOV    l_x, p_x

:laser_fire_done

```

Now that the player's input has been handled, the rest of the game objects need updating. As we just saw, the player can't fire the laser until the last laser has left the screen, so here's the code to facilitate the laser's movement. The flickering effect of the ship's exhaust is implemented here as well:

```

; Move the laser every 2 clocks
MOV    t0, ticks
AND    t0, #1
CJNE   t0, #0, :laser_update_done

; Which direction is the laser moving?
CJE    l_dir, #LEFT, :move_laser_left

; Right
INC    l_x
JMP    :laser_update_done

:move_laser_left

; Left
DEC    l_x

:laser_update_done

; Basic laser bounds checking
CJB    l_x, #16, :skip_kill_laser
MOV    l_active, #FALSE
:skip_kill_laser

; Toggle the exhaust visibility flag every 4 clocks
MOV    t0, ticks
AND    t0, #2
CJNE   t0, #0, :exhaust_toggle_done
NOT    exhaust
:exhaust_toggle_done

```

Everything here should be pretty self explanatory, but one noteworthy detail is the way the **exhaust** flag is toggled. Rather than use conditional logic to switch between 0 and 1, a single **NOT** instruction is used. Because it was originally initialized to **\$FF**, a **NOT** operation will toggle each bit to **\$00**. Another **NOT** will switch each bit back, setting the value back to **\$FF**. Presto! The last step is drawing the game objects in their new positions, which doesn't need much explanation.

## 12.5.5 - Conclusion

The shooter demo itself is very simple, but it illustrates many concepts that can be used to create complex games. With its simple formula of drawing a static background, manually erasing each sprite and redrawing them after accepting player input, many types of games can be created. However, there are more advanced ways to handle background and foreground graphics using the Tile Graphics Engine, which we'll see in the case studies for the following demos.

The shooter demo is a small program with lots of room for expansion. Try hacking some changes into the game such as allowing multiple projectiles on the screen at once, adding scrolling, or simply adding a game objective of some type. The layout and gameplay of the demo are conducive to games such as **Defender** and **Choplifter**, for example.

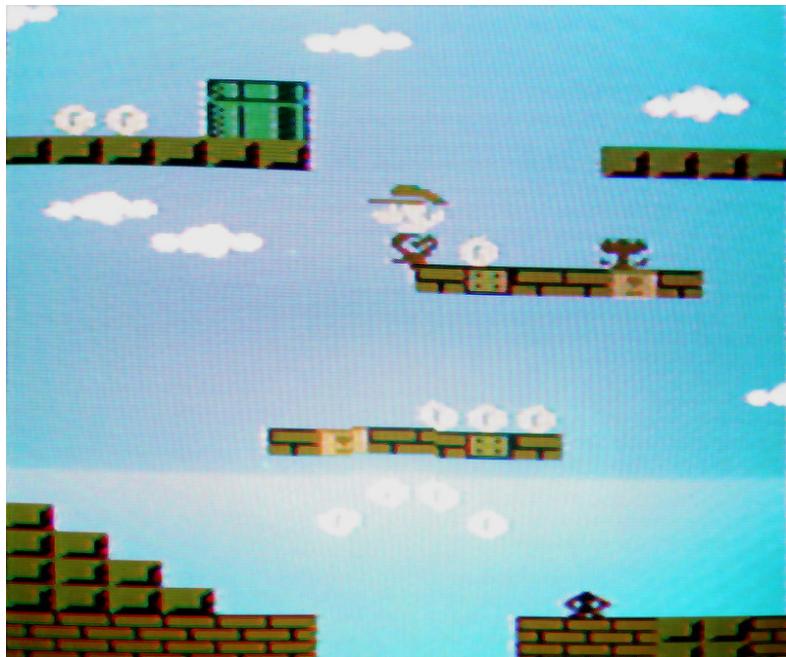
## 12.6 - Case Study: XGS Bros.

The shooter demo was a nice introduction to writing games with the Tile Graphics Engine, but it wasn't particularly functional and was extremely simple. XGS Bros. is a demo written to illustrate more complex games. Specifically, this one uses the style of Super Mario Bros. to illustrate how platform games can be created. The demo has the following objectives:

- Display a title screen
- Allow the player to run and jump while colliding properly with the environment, such as platforms and the ground.
- Provide three screens of environment to explore.
- Animate the player based on movements such as running and jumping.
- Place one animated enemy on each screen that moves between two endpoints.

A screenshot of XGS Bros. can be seen in Figure 12.12.

Figure 12.12 – The XGS Bros. demo.



The source code to the XGS Bros. demo can be found here:

### Tile\_Engine\xgs\_bros\_1\_0.SRC

If you haven't read the case study of the shooter demo in the last section, be sure to do so before reading this one. It covers many simple concepts that will not be repeated here for the sake of brevity.

## 12.6.1 - Data Structures

The data structures of XGS Bros. are along the lines of the simple variables used in the Shooter, but expanded to include additional parameters like a gravitational force applied to the player for realistic jumping, as well as other details.

Also, note that an extra set of variables are set aside from tracking one on-screen enemy, and that these variables are located in another bank. This is because 16 variables were needed to represent the player and other game-related variables, and the **BANK\_GAME** bank couldn't fit the enemy tracking variables as well.

## 12.6.2 - Initializing the Game

This time, let's start by looking at **Game\_Init()** as a whole:

```
; ***** INITITALIZE GAME LOGIC *****
; BANK      BANK_GAME
CLR      ticks           ; Reset the tick counter

; Initialize title screen
MOV      state,    #STATE_TITLE   ; Start on the title screen
MOV      bg_index, #WHITE; Start the blinking "PRESS START!" text on white
MOV      button_up, #FALSE; Not yet waiting for a button up event

; ***** LOAD SCREENS INTO FRAMEBUFFERS *****
M_LOAD_MAP      screen_0, tile_attr_table
M_COPY_SCREEN   #$00, #$00, #$00, #$60

M_LOAD_MAP      screen_1, tile_attr_table
M_COPY_SCREEN   #$00, #$00, #$00, #$C0

M_LOAD_MAP      screen_2, tile_attr_table
M_COPY_SCREEN   #$00, #$00, #$01, #$20

; ***** LOAD THE TITLE SCREEN *****
M_LOAD_MAP      title_screen, title_screen_tat
```

Surprisingly, initialization of the XGS Bros. demo is simpler than the Shooter demo, even though XGS Bros. is by far the more complex game. This is because the Shooter demo builds its background image manually with calls to **M\_FILL\_ROW** and **M\_SET\_TILE**. Since XGS Bros. has much more complex

backgrounds (and more of them), as well as a title screen. Instead, the maps are “drawn” into the source code as data and loaded from program memory into SRAM with `M_LOAD_MAP`.

Note that this time, the player is not initialized in `Game_Init()`. This is because the game is now split into two *states*—a title screen state, and a gameplay state. If the game is running in the title state, the title screen is displayed and an input is handled differently. If the game is running in the gameplay state, the game screen is displayed and the user controls the player. When the game initially transitions from the title screen state to the gameplay state, the player is initialized. Since the initialization is handled then, it can be ignored for now.

Once the state is initialized, each map is loaded into the visible buffer and then copied to its own framebuffer elsewhere in SRAM. Lastly, the title screen is loaded into the visible buffer and left there so the player will see it upon startup.

### 12.6.3 - Updating the Game

Needless to say, updating the XGS Bros. is more complex than the Shooter demo when it comes to updating. Not only is the player’s movement more complex, but there is now animation to consider, a separate enemy character that moves on its own, numerous backgrounds, and two entirely separate game states. In fact, the logic is so much more complex that it doesn’t fit within a single code page and had to be split up into numerous functions. When studying the XGS Bros. source code, you’ll find the game update logic broken up among the following functions:

```
Is_Tile_Solid()
Handle_Input()
Draw_Objects()
Init_Enemies()
```

We’ll talk more about these functions in the next section, but the XGS Bros. demo source code is too lengthy to cover in its entirety. Instead, only the key concepts will be covered. All things considered, the most important and unique functionality of the demo are:

- Collision detection
- Gravity-based jumping
- The enemy

These subjects will be covered in the following sections.

#### 12.6.3.1 - Collision Detection

In the shooter demo, collision detection was done entirely through a couple of hardcoded rules; the player simply couldn’t pass the boundaries of a rectangle that was established inside all of the non-passable screen regions, such as the bottom of the score display and the top of the background. In the XGS Bros.

demo, however, the character is in an arbitrarily designed, map-driven environment that does not fall into such simple rules.

To correctly restrict the player's movement at the right times, the game must be able to determine exactly which tile the player is about to pass into, and move the player back to his original position if the tile is not-passable, or "solid". Examples of solid tiles are bricks and pipes, while non-solid tiles include the blank sky tile, clouds, and the vine.

Whenever the player attempts to move, the **Is\_Tile\_Solid()** subroutine is called. This subroutine accepts an X, Y location in **t4**, **t5**, and uses them to read from the SRAM visible buffer. The subroutine returns **TRUE** or **FALSE** in **t0**, indicating whether or not the tile is solid and therefore non-passable. If **TRUE** is returned, the caller knows that the tile is solid and will prevent the player from moving. Here's the code:

```
Is_Tile_Solid

    ; Get the tile's bitmap index in <t1>
    M_GET_TILE_INDEX t4, t5
    MOV    t1, t0

    ; Assume the tile is solid
    MOV    t0, #TRUE

    ; Check for each passable tile
    CJE    t1, #0, :tile_passable ; Sky
    CJE    t1, #9, :tile_passable ; Cloud (Left)
    CJE    t1, #10, :tile_passable ; Cloud (Right)
    CJE    t1, #8, :tile_passable ; Vine
    CJE    t1, #12, :tile_passable ; Coin
    CJE    t1, #13, :tile_passable ; Fire Flower

    ; The tile is solid
    JMP    :tile_done

:tile_passable
    MOV    t0, #FALSE

:tile_done
    RETP

End_Is_Tile_Solid
```

The only real logic performed in this subroutine is the series of **CJE** instructions that compare the tile index returned by **M\_GET\_TILE\_INDEX**.

The player is capable of moving in all four directions, two of which are directly controlled by the joystick (left and right). As an example of how this subroutine is used for collision detection, let's take a look at the code that moves the player left in the event that the joystick left direction is down:

```
; Move to the left
DEC    p_x

; Skip collision tests if the player moved offscreen
CJA    p_x, #15 - 1, :move_right_done

; Check for collisions along the left three player tiles
; X, Y
```

```

MOV    t4, p_x
MOV    t5, p_y
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :left_collision

; X, Y + 1
MOV    t4, p_x
MOV    t5, p_y
INC    t5
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :left_collision

; X, Y + 2
MOV    t4, p_x
MOV    t5, p_y
ADD    t5, #2
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :left_collision

; No collisions
MOV    p_x_last, p_x           ; Update cached player position
MOV    p_y_last, p_y
JMP   :move_left_done

; The player collided moving left
:left_collision
    MOV   p_x, p_x_last

:move_left_done

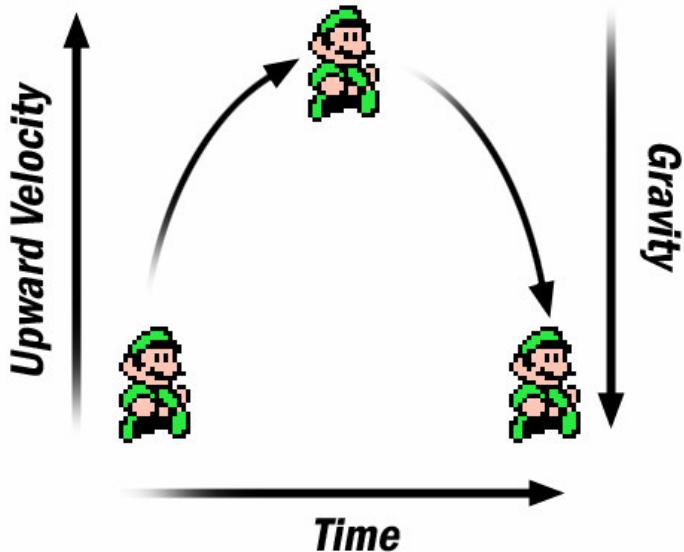
```

As in the shooter demo, the player is moved first with the `DEC` instruction. Once moved, the new location is checked for collisions along all three vertical tiles of the player (since the player is three tiles tall). If a collision occurred, the player's last position, already stored in `p_x_last`, `p_y_last`, is restored. If not, `p_x_last` and `p_y_last` are updated with the new position before handling the other movement directions.

### 12.6.3.2 - Gravity-Based Jumping

Jumping in a platform game has to be reasonably realistic insofar as the player must follow a parabolic arc from the start of the jump to the landing. The easiest way to achieve this effect is by constantly applying a downward gravity vector to the player's location, then giving the player a boost of upward velocity when the jump button is pressed. This velocity lifts the player into the air but is quickly damped and ultimately negated by the constant gravity vector, causing the player to rise quickly, hang in the air for just a moment, then fall back to the ground. Of course, whenever the player runs into a solid tile, the downward movement must stop. Check out figure 12.13 for a visual reference of this technique.

Figure 12.13 – Using vectors to simulate gravity and upward movement in jumps.



The key to smooth movement in these operations is the use of fractional values. However, since the SX52 only natively supports 8-bit integer arithmetic, we'll have to implement our own solution using fixed-point math. While many fixed-point formats could be used, I opted for a simple 8.8 approach to avoid having to mask and shift bit regions of a single integer value. It makes the code easier to read and a bit faster. Of course, memory can run out fast on the XGS ME, and in a tighter situation, using two full bytes for the whole and fractional values might not be possible.

First, the player's vertical location is stored in `p_y` and `p_y_f`. When the player is displayed, the value in `p_y` alone is used. However, when the player moves vertically, the velocity vector is first added to `p_y_f` and, only in the event of an overflow, is `p_y` incremented as well.

`p_yv` and `p_yv_f` store the player's vertical **velocity**, which is added to the player's vertical location at each frame to simulate gravity. This value is of course fixed-point as well to allow for fractional velocities.

To handle vertical movement, the following steps are taken at each frame:

1. Apply the downward gravity force to the player using a fixed-point, fractional value. Make sure the player's downward velocity does not exceed a certain ceiling.
2. Check for collisions along the player's two bottom tiles. If a collision occurred, clear the player's downward velocity. This prevents the player from falling "through" solid objects.

3. Check for collisions along the player's two upper tiles, in case the player is moving upward (as in a jump). If a collision occurred, dampen the player's upward velocity. This causes the player to immediately fall back down upon impact.

With this strategy in mind, the actual code is pretty easy to understand:

```

; Clip the gravity pull at 1.0 if the pull is not negative (upward)
CJA    p_yv, #127, :clip_yv_done
CJB    p_yv, #1,   :clip_yv_done
MOV    p_yv, #1
CLR    p_yv_f
:clip_yv_done

; 8.8 fixed-point addition of gravity to player Y-position
ADD    p_y_f, p_yv_f
ADDB   p_y,   C
ADD    p_y,   p_yv

; Add a fractional gravity coefficient to p_yv
ADD    p_yv_f, #16
ADDB   p_yv,   C

; Check for collisions along the bottom two player tiles

; X, Y + 2
MOV    t4, p_x
MOV    t5, p_y
ADD    t5, #2
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :down_collision

; X + 1, Y + 2
MOV    t4, p_x
MOV    t5, p_y
INC    t4
ADD    t5, #2
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :down_collision

; Check for collisions along the top two player tiles

; X, Y
MOV    t4, p_x
MOV    t5, p_y
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :up_collision

; X + 1, Y
MOV    t4, p_x
MOV    t5, p_y
INC    t4
CALL   @Call_Is_Tile_Solid
CJE   t0, #TRUE, :up_collision

; Check for the player crossing the vertical screen boundaries
MOV    t0, p_y          ; Translate <p_y> into a 1-24 range for easier bounds
checking
INC    t0
CJB   t0, #1,   :up_collision      ; Check collisions with the top of the screen
CJA   t0, #24 - 2, :down_collision ; Check collisions with the bottom of the screen

; No collisions
JMP   :move_y_done

```

```

; The player collided moving up
:up_collision
    MOV     p_y, p_y_last
    CLR     p_yv
    MOV     p_yv_f, #32
    JMP     :move_y_done

; The player collided moving down
:down_collision
    MOV     p_y, p_y_last
    CLR     p_yv
    CLR     p_yv_f
    MOV     p_can_jump, #TRUE
                    ; The player can jump again

:move_y_done

```

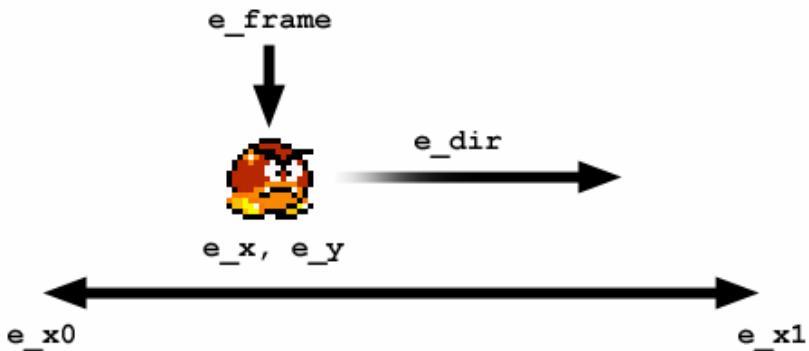
### 12.6.3.3 - The Enemy

Each background in the demo features an enemy moving in a unique location. While the enemy and player don't actually interact, the enemy is animated, and does adhere to the layout of the level. This is accomplished with the following variables, seen in the **BANK\_ENEMIES** bank listed above:

ORG	BANK_ENEMIES		
e_x	DS	1	; Enemy X, Y location
e_y	DS	1	
e_x0	DS	1	; Enemy path start and finish
e_x1	DS	1	
e_dir	DS	1	; Enemy direction of movement
e_frame	DS	1	; Enemy animation frame

**e\_x** and **e\_y** track the enemy's current on-screen location. Each time the enemy moves, it is moved in the direction stored in **e\_dir**. The enemy constantly moves back and forth between two path endpoints set by **e\_x0** and **e\_x1**, and along the way is animated between two frames of animation tracked by **e\_frame**. Figure 12.14 illustrates how these variables work.

Figure 12.14 – The enemy tracking variables.



In order to place the enemy in the correct position in each map, the `Init_Enemies()` subroutine is called each time a new map is loaded. This subroutine checks for one of three cases (map 0, 1 or 2) and hardcodes the enemy's location, endpoints and direction manually:

```
Init_Enemies

; Which background is active?
.BANK BANK_GAME
MOV t0, bg_index

; Clear the enemy animation frame
.BANK BANK_ENEMIES
CLR e_frame

; Map 0
CJNE t0, #0, :bg_index_1
MOV e_x, #9
MOV e_y, #12
MOV e_x0, #7
MOV e_x1, #13
MOV e_dir, #LEFT
RETP

; Map 1
:bg_index_1
CJNE t0, #1, :bg_index_2
MOV e_x, #9
MOV e_y, #4
MOV e_x0, #9
MOV e_x1, #15
MOV e_dir, #RIGHT
RETP

; Map 2
:bg_index_2
MOV e_x, #12
MOV e_y, #21
MOV e_x0, #10
MOV e_x1, #15
MOV e_dir, #RIGHT
RETP
```

**End\_Init\_Enemies**

It should be noted of course that in most cases, a lookup table with such information is preferable to a hardcoded subroutine such as this, but in the case of only three levels, I felt this code was somewhat more readable and explains the point a bit more directly.

When the screen is drawn, the enemy is drawn using **e\_x** and **e\_y**. The enemy has two frames of animation, which are controlled by the **e\_frame** variable. Here's the code for drawing the enemy:

```
; Draw the enemies
_BANK BANK_ENEMIES
MOV t4, e_x ; Enemy X, Y location
MOV t5, e_y
MOV t0, e_frame ; Enemy animation frame
ADD t0, #40
_BANK BANK_VIDEO
M_SET_TILE t4, t5, t0, #COLOR_13, #COLOR_0 + 4
```

Every 8 clocks, the enemy is moved using **e\_x0** and **e\_x1** as the endpoints of its path. The code here simply moves the enemy in its current direction, checks for collisions with either endpoint, and reverses the direction if necessary:

```
; Move enemies every 8 clocks
MOV t0, ticks
AND t0, #7
CJNE t0, #0, :move_enemies_done

_BANK BANK_ENEMIES

; Toggle the animation frame
XOR e_frame, #%00000001

; Move the enemy based on its direction
CJE e_dir, #LEFT, :move_enemy_left

; Right
INC e_x ; Move the enemy
CJBE e_x, e_x1, :e_x1_ok ; Check against X1 bound
MOV e_x, e_x1 ; Clip the position to the X1 bound
MOV e_dir, #LEFT ; Switch directions
:e_x1_ok
JMP :move_enemies_done

; Left
:move_enemy_left
INC e_x0 ; Translate into 1-16 range for easy bounds checking
CJAE e_x, e_x0, :e_x0_ok ; Check against X0 bound
MOV e_x, e_x0 ; Clip the position to the X0 bound
MOV e_dir, #RIGHT ; Switch directions
:e_x0_ok
DEC e_x0 ; Translate back into 0-15 range
DEC e_x

:move_enemies_done
```

These three simple blocks of code create an enemy that, aside from player interaction, is a convincing part of the game world. By extending this enemy tracking info into an array, multiple enemies would be

easily supported, and by adding player collision detection, would be able to actually hurt the player upon contact.

## 12.6.4 - Conclusion

The XGS Bros. demo represents a new dimension of possible games using the Tile Graphics Engine. Platform games are not trivial undertakings, but are not only rewarding to create but fun to play. With enough effort and some clever planning, a complete game could certainly be based on the humble beginnings presented in this demo.

XGS Bros. is written in a straightforward manner and should be easy to hack. Try modifying the game to include multiple enemies, new maps, or functional interaction between the player and the enemy characters. One unique challenge would be making it possible to collect the coins, which would require physically changing the SRAM framebuffer in which they reside so they do not appear again on the screen.

## 12.7 - Case Study: Venture

The last Tile Graphics Engine demo we'll be discussing is called **Venture**, and is loosely based on the classic Atari game **Adventure**. It features a large game world, large enemy sprites, color effects and a complete objective-based quest. Starting at the locked gate to a castle, the player must collect three keys, a glowing blue chalice, and approach the gate with the items in hand. Along the way, three dragons guard the keys and attack the player when he approaches. Check out Figure 12.15 for a screenshot.

Figure 12.15 – The *Venture* game.



The code for Venture is the most complex of the three Tile Graphics Engine demos, and as in the case of the XGS Bros. case study, we'll only be taking a look at the relevant and unique sections. Again, be sure to read both of the preceding case studies before attempting to read this, as it will make references to topics they have already covered without rehashing them.

The source code to Venture can be found here:

**Tile\_Engine\venture\_1\_0.SRC**

Many aspects of Venture rely on techniques already covered, such as:

- **Player Movement**

The player can move up, down, left and right. Because of the game's overhead perspective, there is no gravity to consider when moving vertically. These directions are actually represented better as

north, south, east and west, which is how you'll see them in the code. In this way, the movement of the player is actually simpler than in XGS Bros. and more like the Shooter demo.

- **Maps and Collision Detection**

Like XGS Bros., Venture uses map-driven backgrounds and tile-based collision detection. Collisions are handled with another `Is_Tile_Solid()` subroutine. It simply uses different tile values.

- **Animation**

Unlike XGS Bros., the Venture character is not animated. In fact, his facing position doesn't even change with his direction of movement. Because of this, drawing the player is very easy.

However, other aspects of the game are more complicated than the previous demos, and warrant further discussion:

- **The World Map**

Venture takes place in a large castle. Each screen of this castle must not only be aware of its own background map, but must link up with other background maps through the north, south, east and west doors.

- **Enemy Movement**

Unlike the simple back-and-forth movement of the XGS Bros. enemy, Venture features dragons that actually track the player and hurt him on contact.

- **Color Effects**

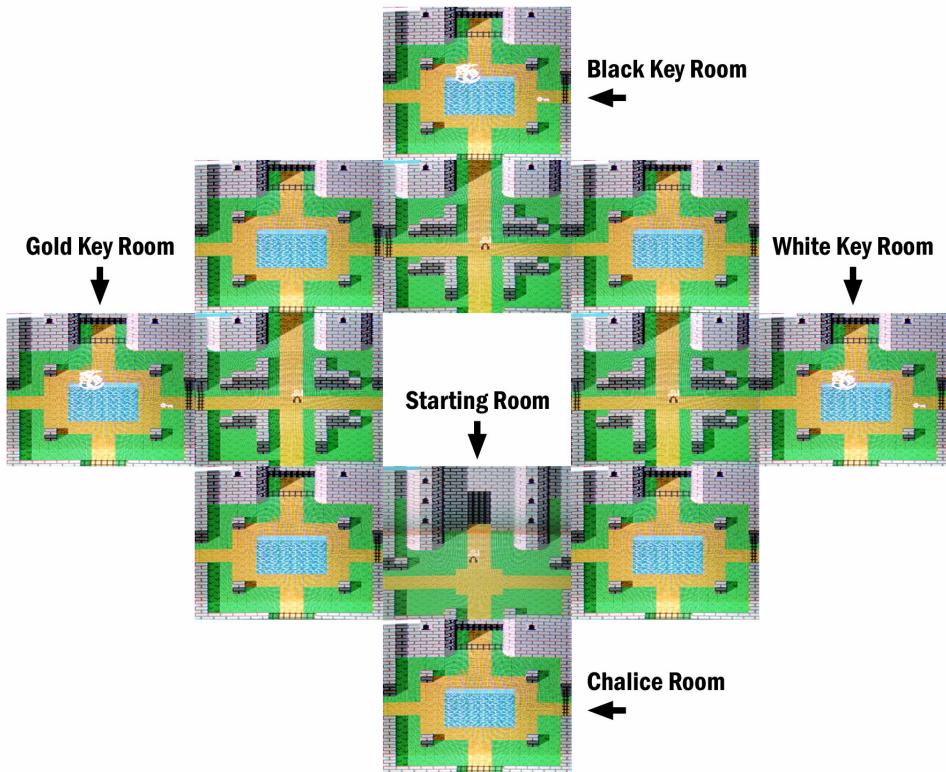
Flashing/pulsating color effects are used to represent special items and events.

## 12.7.1 - Overview

Before getting into the specifics, let's discuss a technical overview of how Venture is designed and implemented.

### 12.7.1.1 - The Game Environment

Venture takes place in a castle environment consisting of multiple, linked courtyards. Each courtyard has a doorway to the north, south, east and west, each of which can be either permanently blocked, or linked to another courtyard. A map of 12 courtyards was created to link these “rooms” together, providing a large game world to explore. The game only uses three unique maps, but these maps are interspersed throughout the 12 rooms to create a varying environment. Check out Figure 12.16 to see the map of the castle.

**Figure 12.16 – A map of Venture’s castle.**

In three of the 12 rooms, dragons guard special keys the player must collect. In another one of the rooms, a glowing blue chalice can be found (without a protecting dragon). A large gate leading into the castle is in the room in which the game starts, which is unlike any of the other rooms. In each room, various tiles are solid, preventing the player from passing through them to create a path that the player must follow. These solid tiles include stone walls, pillars, and pools of water.

### 12.7.1.2 - The Object of the Game

The purpose of the game is to traverse the rooms of the castle, picking up the three keys while avoiding the dragons. The player cannot pick up the blue chalice found in the southernmost room before all three keys have been collected. Once the keys are in hand, the player can take the chalice to the main castle gate (found in the starting room). When the player makes contact with the gate with the chalice, the player takes on the chalice's blue glow, indicating the game is over.

Along the way, the player will encounter the three dragons. The dragons need only make physical contact with the player to inflict damage, and if the player is killed, he will appear in a flashing red color. The

dragons don't feature any sort of actual AI or strategy-based movement; instead, they simply fly towards the player.

A blue "health meter" is found in the upper-left corner of the screen. As the player is hurt, the meter becomes increasingly white, until finally the player dies. The keys the player has collected appear on the upper-right corner.

## 12.7.2 - Data Structures

Venture has the most variables of all three demos to track the numerous game entities and states. Let's take a look at the declarations, bank by bank.

First of up is **BANK\_GAME**, which is used by the central game logic:

ORG	BANK_GAME	
ticks	DS	1
stick_state	DS	1
button_up	DS	1
state	DS	1
p_x	DS	1
p_y	DS	1
p_x_last	DS	1
p_y_last	DS	1
p_dir	DS	1
p_frame	DS	1
p_item	DS	1
p_life	DS	1
p_items	DS	1
game_state	DS	1
seq_timer	DS	1

; Tick count (increments once per frame)  
; State of joystick  
; Tracks a button up event  
; Current game state  
; Player X, Y location  
; X, Y location before last move  
; Player facing direction  
; Player animation frame  
; Is the player holding an item?  
; Player life energy  
; Items in inventory (bitvector: 0-2 = keys, 3 = chalice)  
; Game state (running, over, completed)  
; Timer for special sequences

Many of these are variables are simply used for menial tasks that pertain to any game or demo, such as state transitions, input, tick counting, and so on. In the case of the Venture demo, the relevant specifics include **p\_x**, **p\_y**, which track the player's location, **p\_item**, which tracks whether or not the player is holding an item, **p\_life**, which is the player's remaining health, and **p\_items**, a bit vector with a separate flag for each of the four items.

The next bank, **BANK\_ROOM**, is used to describe the room the player is currently in:

ORG	BANK_ROOM	
r_room	DS	1
r_map_index	DS	1
r_doors	DS	1
r_n_dest	DS	1
r_e_dest	DS	1
r_s_dest	DS	1
r_w_dest	DS	1

; Index of current room  
; Tile map  
; Bit vector of doors (open/closed)  
; North door destination map  
; East door destination map  
; South door destination map  
; West door destination map

**r\_room** is an index into the world map structure (covered below) that tracks the player's current room. **r\_map\_index** is the tile map used to draw the room on the screen. **r\_doors** is a 4-bit vector in which

each of the first four bits represents one of the doors in the room. A value of 1 indicates the bit's corresponding door is open. A value of 0 means it's closed. Lastly, **r\_n\_dest**, **r\_e\_dest**, **r\_s\_dest** and **r\_w\_dest** are indices into the world map as well, linking the map to its adjacent maps through each door.

The last bank, **BANK\_OBJECTS**, tracks game “objects” which is simply a blanket term for any entity within the room aside from the player. Specifically, this includes collectable items and dragons:

```
ORG      BANK_OBJECTS

; Dragon
d_x      DS     1          ; Dragon X, Y location
d_y      DS     1
d_dir    DS     1          ; Facing direction
d_color  DS     1          ; Color
d_state  DS     1          ; State

; Item
i_x      DS     1          ; Item X, Y location
i_y      DS     1
i_index  DS     1          ; Item bitmap index
i_id     DS     1          ; Which item exactly?
i_state  DS     1          ; State of item
i_color  DS     1          ; Item color
i_luma   DS     1          ; Luminance of item
i_luma_dir DS     1          ; Luminance direction of item (up or down)
i_p_items DS     1          ; Local buffer for player inventory
```

The dragon's location and facing direction is tracked with **d\_x**, **d\_y** and **d\_dir**. Since there are three dragons, each of a different color, **d\_color** stores the currently visible dragon's color. **d\_state** determines whether there is a dragon in the current room or not.

If an item is in the current room, **i\_x**, **i\_y** stores its location and **i\_index** determines which tile bitmap is displayed there (either a key or chalice). **i\_id** is a bit more specific, determining which exact item it is (white key, black key, gold key, or chalice). As with the dragon, **i\_state** determines if an item is in the current room or not. **i\_color** is the base color used to draw the item on screen. At each frame, **i\_luma** is added to the base color giving the item an adjustable luminance, which is controlled over time with **i\_luma\_dir**.

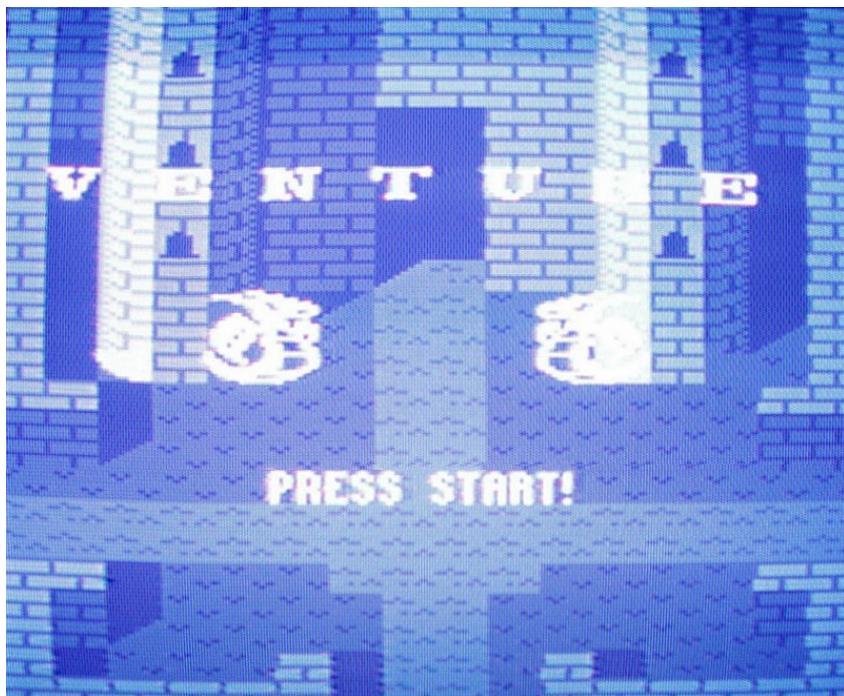
**i\_p\_items** is simply a location within the **BANK\_OBJECTS** room to store the player's inventory, which is normally in **BANK\_GAME**. In certain parts of the code this minimizes bank switching.

### 12.7.3 - Initializing the Game

The initialization of Venture is not unlike that of XGS Bros. The title screen state is set, the maps are loaded into SRAM framebuffers, then the title screen is drawn to the visible buffer for display once the main execution loop begins. In the case of Venture, program memory was saved by not giving the title screen its own unique map as in XGS Bros.; instead, the gate room map was used as a background for the title. The trick, however, was using a different tile attribute table when loading it—one in which all the original table's colors had been converted to monochrome blue. This gives the map a faded, stylistic look

that makes it suitable as a title screen background. Changing the tile attribute table without changing the map can often be a clever way to both save memory and achieve interesting effects. Check out the title screen in Figure 12.17.

**Figure 12.17 – The Venture title screen.**



#### 12.7.4 - The World Map

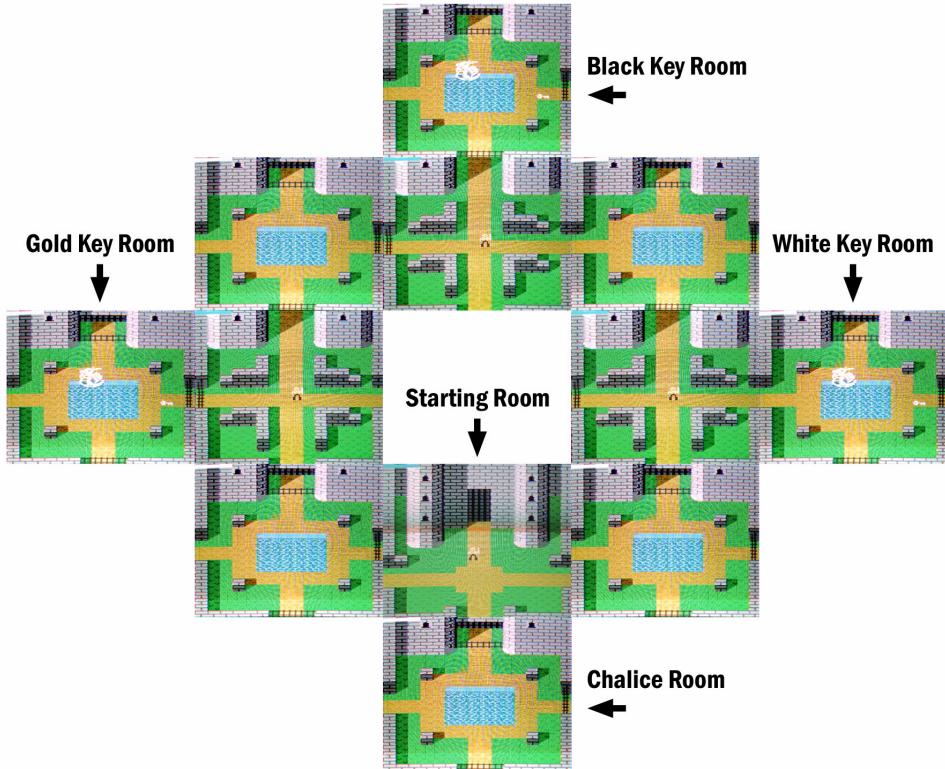
The first and most important difference between Venture and the other demos is that it creates a large game world by using the same tile maps multiple times. To describe this world, a more complex mapping system is needed than the technique used by XGS Bros. in which each available tile map links to the next, then wraps back around to the first.

The map in Venture is a diamond-shape castle, in which each room has four possible exits that lead to adjacent rooms. In order to describe each room, a few pieces of information are necessary:

- The tile map to draw as a background
- The status of each of the four doors (north, south, east and west). In other words, whether or not each door is open or closed.
- The room into which each door leads.

Check out figure 12.18 to see the Venture map.

**Figure 12.18 – The Venture world map.**



On a system with less memory restrictions, the map for a game world such as this might simply be a 2D array of structures in which each element is a room structure containing the above information. If the player passes through the north door, the element at  $x, y - 1$  is loaded as the next room. If the player passes through the east door, the element at  $x + 1, y$  is loaded, and so on. However, in the case of Venture, the game world is not rectangular, and wasting any elements in an array such as this would be an unacceptable use of valuable program memory.

To optimize memory usage as much as possible, only the existing rooms themselves are represented in a simple table. Each room entry in the table provides four pointers to other elements in the tables, corresponding to the north, south, east and west doors.

Each room needs a tile map to draw as its background. Four bits are allocated within the room structure for the tile map, which is more than enough. A total of 16 possible rooms can exist in the map, meaning only four bits are needed to represent a world map index. Furthermore, since each room has exactly four doors, a 4-bit vector is sufficient to represent the open or closed state of each door. So, if each room

needs a 4-bit tile map index, a 4-bit door vector, and four 4-bit adjacent room indices for its doors, we need a total of 24 bits to represent one room. This fits into exactly two 12-bit SX52 program words, making the structure very compact and efficient. Table 12.2 lists this structure nibble by nibble.

**Table 12.2 – Structure of a world map room.**

Nibble	Definition
0	Tile Map Index
1	Door Bit Vector, NESW order ( <i>1 = Open, 0 = Closed</i> )
2	North Door Map Destination
3	East Door Map Destination
4	South Door Map Destination
5	West Door Map Destination

The world map itself is stored directly in program memory. Since a 24-bit structure of packed nibbles is not particularly easy to read, each element in the array is heavily commented. Here's the complete world map declaration:

```

; LEGEND
; -----
; M = Main Entrance/Gate (0)
; G = Garden (1)
; H = Halls (2)

; MAP (12 ROOMS)
; -----
; . . G . . . 6 . .
; . G H G . . 5 A 7 .
; G H . H G   4 B . 9 8
; . G M G . . 3 0 1 .
; . . G . . . 2 . .

world_map

;          N   ESW           NESW
DW      $0FO, $123 ; 0 - Map = 0; Doors = 1111
DW      $199, $000 ; 1 - Map = 1; Doors = 1001

```

```

DW      $180, $000      ; 2 - Map = 1; Doors = 1000
DW      $1CB, $000      ; 3 - Map = 1; Doors = 1100

DW      $140, $B00      ; 4 - Map = 1; Doors = 0100
DW      $160, $AB0      ; 5 - Map = 1; Doors = 0110
DW      $120, $0A0      ; 6 - Map = 1; Doors = 0010
DW      $130, $09A      ; 7 - Map = 1; Doors = 0011
DW      $110, $009      ; 8 - Map = 1; Doors = 0001
DW      $2E7, $810      ; 9 - Map = 2; Doors = 1110
DW      $2D6, $705      ; A - Map = 2; Doors = 1101
DW      $2B5, $034      ; B - Map = 2; Doors = 1011

```

## 12.7.5 - Enemy Movement

In each of the three key rooms, a colored dragon attacks the player to guard the item. Upon contact, the player's health meter is decremented and the player will eventually die if he doesn't get out of the way.

The dragon does not move with any particular strategy other than to directly attack the player. The most direct approach to this pattern would be comparing the dragon's location to that of the player's at each frame, and if necessary, moving one tile closer along each axis in which they are not already touching. Over the course of a few frames, the dragon would move from its original location to wherever the player is.

This is a simple and space-effective way to move the dragon, but it is a bit unfair; the dragon will have little trouble not only catching the player, but staying on him no matter where he goes. To make the movement of the dragon appear slightly less contrived and give the player a better chance of evasion, one constraint was made to this approach—the dragon can only move along one axis at a time. At any given frame, the dragon can move horizontally closer to the player or vertically closer, but not both.

The dragon is updated during `Handle_Input()`, the same subroutine in which the player is moved. The following code is executed every 8 ticks, like the player's movement, to prevent the dragon from moving too fast:

```

; Move the dragon closer to the player (move only one direction at a time)
MOV    t0, p_x                      ; Put the player X, Y in <t0, t1>
MOV    t1, p_y
_BANK  BANK_OBJECTS

; Move up towards the player
CJAE  d_y, t1, :dragon_not_below
INC   d_y
JMP   :dragon_move_done

; Move down towards the player
:dragon_not_below
CJBE  d_y, t1, :dragon_not_above
DEC   d_y
JMP   :dragon_move_done

; Move right towards the player
:dragon_not_above
CJAE  d_x, t0, :dragon_not_right
INC   d_x
JMP   :dragon_move_done

; Move left towards the player

```

```

:dragon_not_right
    CJBE    d_x, t0, :dragon_move_done
    DEC     d_x
    JMP     :dragon_move_done

:dragon_move_done

    ; Make sure the dragon is always facing the player
    CJAE    d_x, t0, :face_left
    MOV     d_dir, #RIGHT
    JMP     :dragon_face_done
:face_left
    MOV     d_dir, #LEFT
:dragon_face_done

```

At any time, the dragon can move in one of four ways—left, right, up or down. The trick to implementing our constraint is causing each of these cases to immediately exit the routine after executing, preventing any further movements from occurring. This ensures the dragon cannot move along more than one axis at once.

Lastly, the dragon can be drawn in one of two facing directions, and depending on whether or not the player is to the dragon's right or left, one of these two directions is selected and stored in `d_dir`.

## 12.7.6 - Color Effects

The keys and chalice all emit a pulsating glow to indicate their status as special items and to attract the player. This effect is easy to implement and, when used in moderation, is a cool way for an object to stand out against the background and appear special or unique.

As mentioned in the data structures section, items in venture are drawn with the “base color” stored in `i_color`, and a separate luminance value stored in `i_luma` which is added to the base. The reason for this separation is that it allows an item to be drawn in any given shade of the same color depending on `i_luma`'s value. By changing this value over the course of multiple frames, the color's luminance can fade in and out, creating a smooth “pulsing” effect that resembles an ethereal glow or shimmer.

This cycling luminance value is implemented with two variables—`i_luma`, which of course stores the current luminance value at a given time, and `i_luma_dir`, which determines whether the luminance will increase or decrease on the next frame. After increasing or decreasing, `i_luma` is compared to the minimum and maximum luminance values, determined by the `ITEM_LUMA_MIN` and `ITEM_LUMA_MAX` constants. If either threshold is exceeded, the direction is reversed.

The following code is taken from `Game_Update()` and implements the luminance effect:

```

; Update effect every 4 ticks
_BANK   BANK_GAME
MOV     t0, ticks
AND     t0, #3
CJNE   t0, #0, :skip_flash
_BANK   BANK_OBJECTS

; If the item is currently at either endpoint, reverse the increment

```

```

CJNE    i_luma, #ITEM_LUMA_MIN, :luma_not_min
MOV     i_luma_dir, #1           ; Move up (1)
:luma_not_min
CJNE    i_luma, #ITEM_LUMA_MAX, :luma_not_max
MOV     i_luma_dir, #255        ; Move down (-1)
:luma_not_max

; Add the increment to the value
ADD     i_luma, i_luma_dir

:skip_flash

```

The code executes every four frames to prevent the effect from being too fast. Note that every time the luminance changes, `i_luma_dir` is added but never subtracted. This works because `i_luma_dir` is either equal to **1** or **255**, which is equivalent to the two's compliment **-1**. When the luminance value is being decreased, **-1** is being added, equivalent to a subtraction of 1. This simplifies the logic and reduces code. In environments like the XGS ME, where program space is always valuable, these kinds of little tricks that replace conditional logic with simple arithmetic are a great way to save both memory and speed.

## 12.7.7 - Conclusion

Venture is the most complex of the three Tile Graphics Engine demos and shows that with a little planning, a complete game with detailed graphics, real objectives and hostile enemy characters can be achieved. In addition, smaller details like the translucent, overlapping wall shadows and the tile attribute table swap that produces the blue-shaded title screen illustrate that tons of code and clock cycles are not always required to get interesting results. Often, simple tweaks to something that already exists is enough to create something significant.

If you would like to continue studying this program, consider hacking some modifications into it. Try modifying the map, altering the behavior of the dragons, adding new items or perhaps changing the objective, or maybe add some new uses of the luminance effect.

## 12.8 - Case Study – The Text Console

The text console demo was created to demonstrate keyboard input on the XGS ME, as well as lay the groundwork for XOGO, a custom language currently in development for the XGS ME that combines elements of BASIC and LOGO.

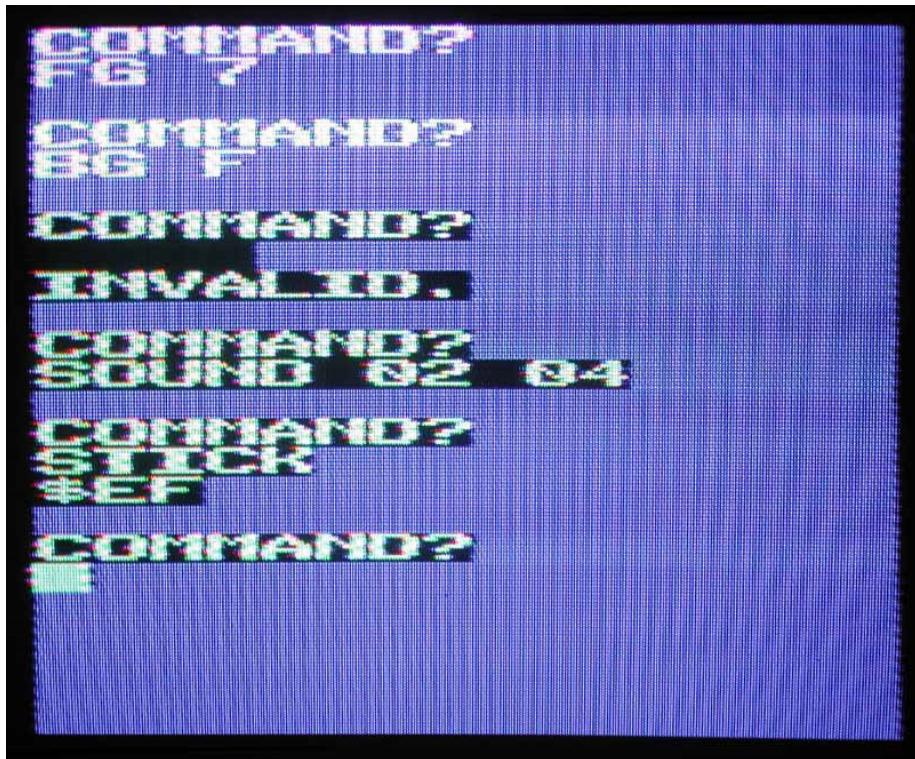
The source code to the Text Console can be found here:

[Tile\\_Engine\text\\_console\\_1\\_0.src](#)

The demo presents a blue screen and blinking cursor in the style of the command-line interfaces found in many older computers. A prompt asks the user to enter a command. Valid commands cause various things to happen, including results that are printed back to the console. Invalid commands cause an error

message to appear. As new lines of text are entered, either from user input or the prompts themselves, the console automatically scrolls upward. Check out Figure 12.19 for a screenshot:

Figure 12.19 – The text console demo.



### 12.8.1 - The Commands

The text console accepts a handful of commands to demonstrate its capabilities. To make the parsing of each command easier, commands **must** start at the first character on the line. Each parameter is separated by a single space, and all parameters are accepted in hex to make translation easier.

The commands are as follows:

**FG color**

Sets the foreground color to **color**, a single-digit hex value. The foreground color does not affect text already entered; it only applies to text entered after the command.

**BG color**

Works just like **FG**, but sets the background **color**. The color parameter is also a single-digit hex value.

**STICK**

Prints the current stick status as a two-digit hex value to the console.

**OUTP value**

Writes **value**, a 3-bit value expressed as a single hex digit, to **RB0–RB2**.

**INP**

Reads the 3-bit value at **RB0–RB2** and prints it to the console as a single-digit hex value.

**SOUND freq vol**

Plays a sound of the specified frequency and volume. Both parameters are 2-digit hex values.

## 12.8.2 - Graphics and Visuals

Graphically, the text console is perhaps the most simplistic demo made with the Tile Graphics Engine, as it consists entirely of a solid blue screen upon which text is typed. When the program is initialized, the “text document” (SRAM) is cleared to this blank blue background. This is accomplished by first clearing the visible buffer, then copying the buffer to the rest of the memory. Normally, this could simply be done by moving the active page address to each screen and calling **M\_FILL\_SCREEN**, but this code was written before control over the active page was allowed by the engine.

## 12.8.3 - Keyboard Input

Input from the keyboard is read each frame with the **Read\_KB** subroutine. This subroutine is not particularly complicated, but due to the relatively slow speed of the keyboard’s communication protocol, the very process of communication takes so much time that the time allocated to **Game\_Update** is literally exceeded and the video signal visibly jerks as a result. Fortunately, this is only a momentary disruption and unless you type particularly fast, is bearable for most purposes.

The following is the source to the **Read\_KB** subroutine:

```
Read_KB
    CLR      t2
    ; test if CLOCK and DATA are low signifying a START bit
    MOV      counter2, #7
    CLR      counter
:KBD_Wait_Clock_Low
    jnb      KBD_PORT.KBD_CLOCK, :kb_active
    DJNZ    counter, :KBD_Wait_Clock_Low ; wait for CLOCK=0
```

```

DJNZ    counter2, :KBD_Wait_Clock_Low
RETP
:k_b_active

; CLOCK=0, verify start bit, i.e. DATA=0
; delay into signal 5.0 us to get solid sample
;DELAY(CLK_SCALE * 50)
snb    KBD_PORT.KBD_DATA
ret    ; DATA is high return

; CLOCK=0 and DATA=0, therefore start bit detected, data is now being streamed
; at 10 - 16.7Khz, or 60-100us clock cycles

; now sync to high clock pulse
:KBD_Wait_Clock_High

sb    KBD_PORT.KBD_CLOCK
jmp  :KBD_Wait_Clock_High      ; wait for CLOCK=1

; CLOCK=1 and DATA=0, therefore start bit detected, data is now being streamed
; at 10 - 16.7Khz, or 60-100us clock cycles
:KBD_Init_Read_Loop

clr   kbddata           ; clear the data storage
clc   ; make sure carry is clear
mov   kbdcounter, #8    ; read 8-bits

:KBD_Next_Bit
; at entrance to this loop, we are in the high phase of the clock,
; wait for the low transition then sample...

; wait for CLOCK to go low

:KBD_Wait_Clock_Low2

snb    KBD_PORT.KBD_CLOCK
jmp  :KBD_Wait_Clock_Low2      ; wait for CLOCK=0

; the clock is now low
; center sampling point 5.0 us into DATA bit
;DELAY (CLK_SCALE * 50)

; sample data on DATA line and shift into position
movb  kbddata.7, KBD_PORT.KBD_DATA
rr    kbddata

; clock is still low, wait for high transition

:KBD_Wait_Clock_High2

sb    KBD_PORT.KBD_CLOCK
jmp  :KBD_Wait_Clock_High2      ; wait for CLOCK=1

; loop 8 times
djnz  kbdcounter, :KBD_Next_Bit

; we overshifted one bit, restore
rl    kbddata

; disregard parity and stop bit for now

MOV    t2, #1

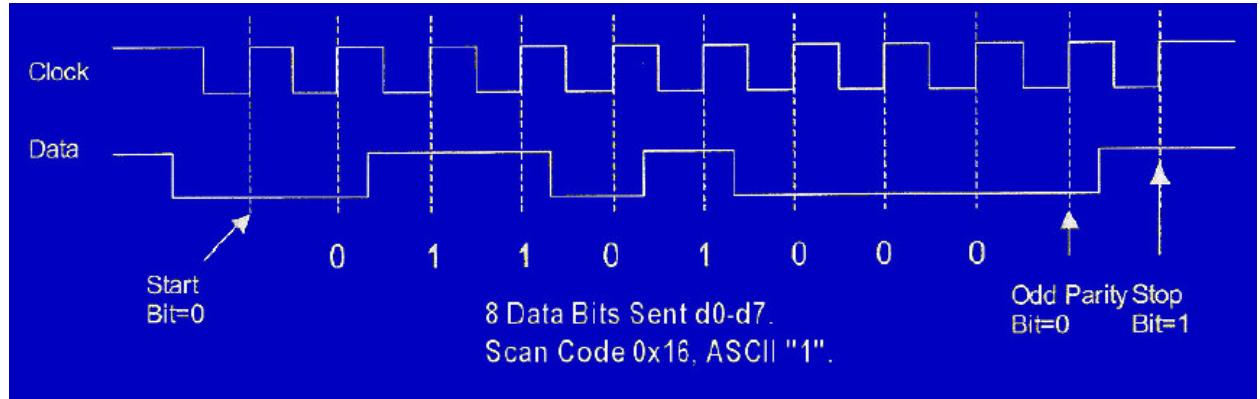
; kbddata holds keyboard scan code
retp

```

**End\_Read\_KB**

This subroutine communicates with the keyboard using the standard PS/2 protocol. While a complete description of PS/2 device communication can be found in the ***Design Your Own Video Game Console*** eBook, Figure 12.20 presents a graphical summary of the protocol.

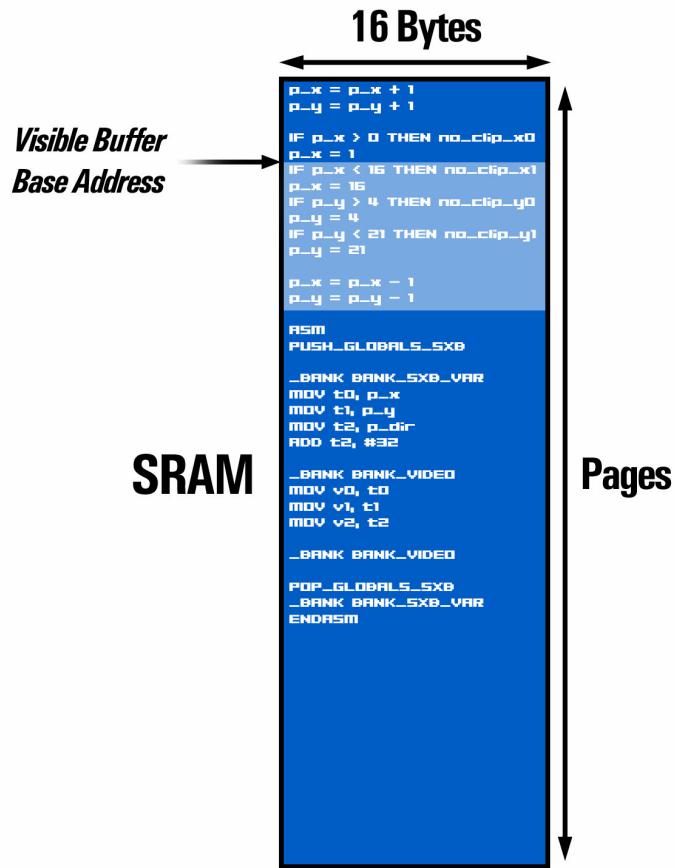
**Figure 12.20 – The PS/2 protocol.**



#### 12.8.4 - Text Input and Scrolling

The automatic scrolling of the text console is actually the easiest part of the demo. All text entry is sent directly to the SRAM at the current active page, and remains there until it is overwritten. A cursor X and Y location stored in the **cx** and **cy** variables determines where the next character read from the keyboard should be written within the currently visible screen. When the cursor reaches the bottom of the screen, the visible page and active page are incremented by one row. This causes an automatic upward shifting of all text currently onscreen and ensures that the cursor follows as well, so new characters will always be written to the next. Figure 12.21 is a visual representation of how scrolling works in the text console demo in terms of the SRAM and the visible page.

Figure 12.21 – Scrolling the visible page through the SRAM to create a text console.



Every time a text character, text string or newline is entered into the text console, the location of the cursor must be updated and the screen may need to be scrolled. Three macros were written for the demo that make inserting text into the console easy:

#### M\_PRINT\_CHAR c

Prints the character **c** to the console. The cursor is incremented by one, and if the character is written to the end of the line, it automatically appears as the first character in the next line. If the cursor is at the bottom of the screen when this occurs, the console automatically scrolls up by one line.

#### M\_PRINT\_STR str

Prints the string **str** to the console. **str** is a 12-bit program memory address pointing to a string of characters terminated by the **EOS** constant. After the string is printed, a newline is automatically inserted

causing the next printing to appear on the next line. The text console scrolls automatically if the string is printed on the last line of the current screen.

#### **M\_PRINT\_NEWLINE**

Moves the cursor to the next line and automatically scrolls the text console if the bottom of the screen is reached.

## 12.8.5 - Parsing and Executing Commands

When the enter key is pressed, the current line is evaluated to determine if a valid command has been entered. This is done in a simple, “brute force” fashion in which a separate block of code for each command is run that directly compares each character in the row of text to the characters that comprise a possible command.

The first step is getting the command itself from the document. At the time the enter key is pressed, we know that the cursor’s current line within the SRAM contains the command. As an added constraint to make things simpler, the command cannot be preceded by any whitespace on the line. Therefore, the first character on the line is the first in the command, the second character on the line is the second in the command, and so on.

### 12.8.5.1 - The Do\_Cmd Subroutine

Whenever the enter key is pressed, the **Do\_Cmd** subroutine is called. This subroutine parses the command by reading each character on the current line, determines which command is to be executed and what parameters have been provided, and finally, performs the action itself.

### 12.8.5.2 - Reading the Command String

The command to be executed is initially located in SRAM along with the rest of the text document. To efficiently parse the text, it’s preferable to have the command string itself located in the SX52’s on-chip RAM. Fortunately, the **M\_GET\_TILE\_INDEX** macro makes it easy to read the bitmap index of a tile on the screen. In our case, this bitmap index corresponds directly with the character.

The following code copies each character of the line into a 16-character command string buffer:

```
; ***** READ THE COMMAND FROM THE TEXT WINDOW
_BANK    BANK_VIDEO

; Copy the command into the command buffer
MOV      t3, #16
MOV      t4, #0
:read_char

; Get the next character and increment the pointer
_BANK    BANK_EDITOR
```

```

MOV      t5, cy
_BANK   BANK_VIDEO
M_GET_TILE_INDEX t4, t5

; Copy the character into the command buffer
CLR      FSR
_BANK   BANK_CMD_STRING
ADD      FSR, t4
MOV      IND, t0

; Increment the pointer
INC      t4

DJNZ    t3, :read_char

```

At this point, the command string buffer, located in RAM, contains the complete command string as well as any parameters that may be included as well.

16 variables are declared the alias each of the 16 memory locations within the command buffer, named **c0** through **c15**. These allow any given character within the buffer to be easily read without the need for any sort of indirect addressing scheme.

### 12.8.5.3 - Parsing the Command

With the command string in fast on-chip RAM, the string can be parsed to determine which command was entered, which parameters are present, and what their values are. Commands are identified using a simple brute-force approach in which each character is directly compared to its corresponding character in a given command name. If none of the comparisons fail, the command has been safely identified.

This approach is fine for a handful of commands, but quickly consumes program memory. In the case of programs that need to interpret a large number of commands, a string table should be set up containing each possible command, and a subroutine should be written that can compare a given string to each string in the table and return the index of the matching command (if a match is found).

For example, the following block of code determines if the **FG** (foreground color) command has been entered:

```

; ***** FG - Set foreground color *****
CJNE    c0, #16, :no_cmd_fg      ; F
CJNE    c1, #17, :no_cmd_fg      ; G
CJNE    c2, #0,  :no_cmd_fg       ; <Space>

; Set the foreground color based on the color argument
DEC     c3
CLC          ; Multiply by 16 to get the base color
RL     c3
RL     c3
RL     c3
RL     c3
ADD    c3, #BLACK_LEVEL
ADD    c3, #7                   ; Add the luminance
MOV     t0, c3
_BANK  BANK_EDITOR
MOV     pen_fg_color, t0 ; Set the color

```

```
JMP      :cmd_done
:no_cmd_fg
```

The first three characters *must* be **F**, **G** and space in order for the **FG** command to be successfully recognized. The space is required so the parser knows that additional characters don't follow **FG**, and to ensure that the parameter, a single hex digit specifying the color, follows the command with the proper spacing.

If **c0-c2** contain the command name and the trailing space, then **c3** must contain the single hex digit specifying the color to which the foreground is to be changed. This makes our job easy—all that remains is to translate the value of **c3**, which is currently a character index, into an actual foreground color value. Since the characters **0-F** are found in tile indices 1 through 15, we know that the character index *minus one* is directly equal to the actual values **0** through **F**. The first step is then to decrement **c3**. The base color value is then calculated by multiplying this value by 16 (four left shifts). Finally, the NTSC base black level and a luminance value of seven (to ensure a bright foreground color) are added. This value is now ready to use for all future character tiles written to the text document, and is stored in the **pen\_fg\_color** variable. As you'd probably guess, this variable tracks the foreground color of the text.

#### NOTE

If the process by which the foreground color was calculated was confusing to you, make sure to read the **Designing Your Own Video Game Console** eBook, which explains color TV signal generation in full detail.

### 12.8.6 - Conclusion

That pretty much wraps up the text console. If you understood how the **FG** command was handled, the rest of the commands won't be difficult to figure out. Make sure to check out the source code and get the details on how the rest of it was done. As stated in the introduction, this text console demo was created not only to demonstrate the concepts involved in creating a text console, but also as a test bed for the development of XOGO, an on-board language currently in development for the XGameStation Micro Edition. The XOGO language combines elements from both BASIC and LOGO, to create a simple and fun programming platform that anyone with an XGS ME, television set and PS/2 keyboard can use immediately. Check out the official XGameStation web site, [www.xgamestation.com](http://www.xgamestation.com), for updates and information as it becomes available.

### 12.9 - Conclusion

The Tile Graphics Engine is a powerful and easy-to-use solution for making graphical XGS ME programs. The included demos illustrate the majority of what can be done using the engine without bending over backwards to optimize and squeeze out every last drop of program space and performance. While the limitations of purely tile-based graphics can be a bit limiting, the resulting overall detail and simplicity are usually enough to make up for the trouble.

## Chapter 13: The SX/B BASIC Compiler

Parallax Corporation has released the SX/B BASIC compiler, developed by Terry Hitt. SX/B allows programs to be written for both the SX28 and the SX52 (the processor used in the XGS ME), in a custom dialect of BASIC. Rather than directly generate object code, SX/B converts BASIC source code into its equivalent assembly language counterpart, then automatically feeds this intermediate version of the program into the SX assembler to create a finished program.

SX/B is under continued development and is frequently being improved and expanded. At the time of this writing, only the SX28-compatible version of the compiler is publicly available. An advanced beta of the forthcoming SX52-compatible version was used to write the demos in this chapter, but the full version is slated for release shortly and is not expected to differ in any significant way.

### 13.1 - Using the SX/B BASIC Compiler

SX/B development has been seamlessly integrated into the SX-Key IDE starting with version 3.0. The latest version of the SX-Key IDE can always be downloaded for free at Parallax Corporation's website:

[www.parallax.com](http://www.parallax.com)

Inside the IDE, the **File** menu now has two **New** options; **New (Assembly)** and **New (SX/B)**. Once a file has been created using the **New (SX/B)** option, the **Assemble** item under the **Run** menu becomes the **Compile** option. If you have not done so already, please read **Chapter 4: Using SX-Key** for complete coverage on using the SX-Key IDE.

The only gotcha to remember is that the SX/B compiler produces an assembly source file that is assembled with the SX assembler. Because of this, certain errors don't emerge until the assembly stage, in which case the temporary assembly equivalent of your BASIC program is displayed in the error listing. Remember that this is **not** the program source file and should not be edited. It is merely a way to display your program's error.

Lastly, full documentation is provided for SX/B. The **SX/B Help** item under the **Help** menu opens extensive BASIC documentation including a complete language reference and multiple example programs.

#### NOTE

To use SX/B as it was meant to be used, the SX-Key is required, as SX/B is integrated directly with the SX-Key IDE starting with version 3.0. However, SX/B can still be used for XGS Micro Studio users by compiling BASIC programs down to their assembly equivalents, then loading the generated assembly source file into XGS Micro Studio for assembly and programming.

## 13.2 - A Brief Overview of the SX/B Language

SX/B is a high-level language that strikes a practical balance between ease of use and performance. On a processor like the SX52, where both RAM and program memory are at a premium, highly abstracted, feature-rich languages can easily produce prohibitively bloated object code that negates their usefulness. Pure assembly, on the other hand, provides the maximum efficiency at the cost of ease of use; programmers often spend as much time implementing a program's core logic as they do worrying about low-level system details.

SX/B provides a syntactic “upgrade” to the native SX52 assembly language with a distinctive BASIC look and feel, but at the same time remains close enough to the low-level world to ensure the compiled program stays within the reasonable limits of code size and performance. While it is primarily a non-optimizing compiler, a handful of simple optimizations are thankfully taken care of, such as substituting division and multiplication by powers of two with bit shifting.

This section is not meant to be an exhaustive tour of the SX/B language. Rather, it is a simple primer designed to quickly get you up to speed with the look and feel of the language, allowing you to better understand the demos in this chapter and more easily digest the full language reference found in the SX/B documentation (included with SX-Key 3.0 and above).

### 13.2.1 - Structure of a BASIC Program

An SX/B program is structured in a manner similar to a pure assembly program. Most of the directives supported by the SX assembler are directly supported by SX/B as well. The following is a simplified version of the standard programming template included with SX/B:

```
' -----
' Device Settings
' -----
DEVICE      SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ       4_000_000

' -----
' INTERRUPT
' -----
ISR_Start:
  ' ISR code here

ISR_Exit:
  RETURNINT  {cycles}

' =====
' PROGRAM Start
' =====

Pgm_ID:
  DATA  "SX/B Template", 0

Start:
  ' initialization code here
```

```

Main:
  ' main code here
  GOTO Main

' -----
' Page 1 Code
' -----


Page_1:
  ADDRESS $200

P1_Start:
  GOTO P1_Start                                ' error if Pg0 overruns Pg1

```

Without any actual BASIC code included, this template doesn't look much different than a straight assembly program would. A few minor differences are important to note, however; in true BASIC style, label declarations end in colons, and comments are delimited by the apostrophe (') instead of the semicolon (;).

## 13.2.2 - Declarations

Both variables and arrays are directly supported by SX/B. Furthermore, individual bits can be declared as well, and SX/B takes care of the dirty work of actually managing their location in memory. Lastly, SX/B transparently handles the trouble of RAM banks, allowing any variable or array to be accessed arbitrarily with little concern for their actual location in memory.

### 13.2.2.1 - Variables

Byte variables are declared with the **VAR** keyword:

```

x      VAR      BYTE
y      VAR      BYTE
my_var  VAR      BYTE

```

This code declares three byte variables; **x**, **y** and **my\_var**.

### 13.2.2.2 - Arrays

Arrays are declared using a modified version of the variable declaration seen above:

```

array_0  VAR      BYTE ( 4 )
array_1  VAR      BYTE ( 3 )

```

This code creates two arrays, **array\_0** and **array\_1**, consisting of four and three byte elements, respectively.

### 13.2.2.3 - Bits

While the SX52, like most processors, cannot directly access individual bits, SX/B simulates this ability by allowing the declaration of bit-sized variables. SX/B manages the actual location and access of these bits transparently, allowing you to think of Boolean values and flags in more appropriate terms:

```
my_flag VAR BIT
```

This code declares a single bit variable called `my_flag`.

### 13.2.2.4 - Constants

Lastly, constants can be declared in a manner familiar to assembly language programmers:

```
min CON 10
max CON 100
```

This code declares the constants `min` and `max`, respectively set to values of 10 and 100. The only difference here from SX assembly is that the `EQU` directive is replaced with the `CON` keyword.

## 13.2.3 - Fundamental Language Commands and Nuances

As stated above, this section is not about learning SX/B from start to finish. Rather, its purpose is to get you comfortable with the look and feel of the language, especially in places where the differences from pure assembly are particularly important.

### 13.2.3.1 - Arithmetic Operations

One of the biggest differences between SX/B and assembly is the way in which arithmetic operations are carried out. For example, if you were to add two variables, `v0` and `v1`, and store the sum in `v2` without disturbing `v0`, you'd write something like the following in SX assembly:

```
MOV    W, v0
ADD    W, v1
MOV    v2, W
```

In SX/B, the code required is much simpler and more natural:

```
v2 = v1 + v0
```

SX/B will produce the following code:

```
MOV W, v0           ;v2 = v0 + v1
ADD W, v1
MOV v2, W
```

Not bad, huh? In many cases, the code produced by SX/B is no different than the code you would write yourself. There are only so many ways to express a simple idea, after all. It's only when code becomes more complicated that the differences between SX/B and a human assembly programmer become apparent.

While the syntax provided by SX/B for arithmetic operations is indeed simpler and more abstract, one key difference between SX/B and other high-level languages is that multiple operations are not allowed in the same expression. For example, the following expression:

```
v0 = ( v1 + v2 ) / v3
```

Is not directly supported. Rather, it must be broken down such that each operation is given its own expression and then merged:

```
temp = v1 + v2
v0 = temp / 3
```

The upside to this approach is that you'll never have to worry too much about the code SX/B is producing for your expressions. With a maximum of one operation per line, you can keep a clear idea of the resulting assembly code in mind at all times.

### 13.2.3.2 - Logic and Program Flow

Replacing the comparison and jump instructions of assembly are the more BASIC-like **IF** and **GOTO** commands. To jump from one location in the program to another at any time, use **GOTO** just like you would use **JMP**:

```
GOTO my_label
    ' ... This code is skipped ...
my_label:
    ' ... This code is implemented ...
```

To jump to a certain line of code depending on a given condition, use the **IF** command:

```
IF x > y THEN my_label
```

This line of code jumps to **my\_label** if the value of **x** is greater than that of **y**.

Like the examples of assigning the result of an arithmetic expression show above, the **IF** statement does not allow more than one operation in its expression. This means that any extra operations required for the **IF** comparison must be carried out beforehand and stored in a temporary variable.

Lastly, **FOR** loops are provided as a powerful way to implement controlled loops:

```
FOR x = 0 TO 10
    my_array ( x ) = x + 2
NEXT
```

In this example, 10 elements of the array `my_array` are set to the value of `x + 1` at each iteration.

### 13.2.3.3 - Subroutines

The next major stop in this brief tour of SX/B is the subject of subroutines. As we've seen so far, subroutines are an important part of assembly language programming, and SX/B supports them in a very similar fashion.

Subroutines are declared with a label, just like they are in assembly:

```
My_Sub:
    ...
    ; ... Subroutine code
    ...
RETURN
```

Note the use of the `RETURN` command—this is SX/B's equivalent to SX assembly's `RETP` instruction.

Calling a subroutine is simple as well; the `GOSUB` command takes the place of the traditional `CALL` instruction:

```
GOSUB My_Sub
```

One important feature added by SX/B is the ability to pass parameters to subroutines. SX/B automatically declares four special variables, named `__PARAM1` through `__PARAM4`, used to store up to four parameters passed to a subroutine. To pass parameters to a subroutine, simply append them to the usual `GOSUB` command as a comma-delimited list:

```
GOSUB My_Sub, x, 10, 20
```

This passes three parameters to `My_Sub`; the variable `x` and the literal values `10` and `20`. There are two very important details to notice in this example. First, a comma must separate the name of the subroutine and the first parameter. Second, literal values in SX/B *do not* require the leading `#` symbol as they do in assembly.

Inside the code for your subroutine, `__PARAM1` through `__PARAM4` can be treated just like any other variable. However, because these special variables are used somewhat unpredictably by the SX/B internally, there is no guarantee that their values will remain intact for long. Because of this, the first step in any subroutine that accepts parameters is to save the parameters in variables you declared yourself:

```
My_Sub:
    p0 = __PARAM1
    p1 = __PARAM2
    p2 = __PARAM3
RETURN
```

This subroutine immediately copies the three parameters into the user-defined `p0` through `p2`.

### 13.2.4 - Inline Assembly

As a final point of interest before moving on to usage of SX/B in real projects, let's discuss inline assembly. **Inline assembly** is a term that refers to assembly language written within high-level language code. Inline assembly is useful when you'd like to write the majority of your program in a high-level language, but still need certain time- or size-critical blocks of code in raw assembly to ensure efficiency is maximized.

Fortunately, since SX/B generates an assembly source file directly compatible with the SX assembler, inline assembly is a natural addition to the language.

Inline assembly can be accomplished in two ways. First, single lines of assembly code can be inserted anywhere by simply beginning the line with a backslash (\) character. For example:

```
v0 = 10
v1 = 20
\ADD    v1, v0
```

This code initializes the variables **v1** and **v0** in BASIC, but adds them together in assembly.

Of course, most of the time, inline assembly is more than just inserting single assembly instructions. Usually, the inline assembler is used to write entire subroutines, which is why the more elegant **ASM..ENDASM** syntax is available:

```
ASM
MOV v0, #10
MOV v1, #20
ADD v1, v0
ENDASM
```

This code implements the entire example from above in assembly, requiring three instructions. Note that when using the inline assembler, *all* of the SX assembler's rules once again apply. This means literal values are preceded with the # symbol, for example.

As a last note, one important aspect of inline assembly is that of banking. SX/B manages the **FSR** register automatically and transparently to access whichever variable or array element the program may need. In the case of pure BASIC programs, this is of little concern to the programmer, but in the case of programs containing large blocks of assembly language that manipulate the **FSR** register either directly or indirectly, it is vital that the assembly code restore **FSR** to its expected state before returning. SX/B references variables and arrays directly using the **FSR** and **IND** registers. This means that you need only **clear FSR** at the end of any assembly code that uses the **BANK** instruction. Failing to do so will leave SX/B in an unexpected state and can immediately result in erratic behavior.

Besides simply clearing **FSR** before returning to BASIC, however, inline assembly must be extremely careful with the contents of the first four global registers. These registers correspond to the internal SX/B variables **\_PARAM1** through **\_PARAM4**, and must be in the same state when returning to BASIC as they were when entering the inline assembly. In the case of the case studies that follow, a new bank was

created to store the contents of these registers while inside inline assembly blocks. Two macros, `PUSH_GLOBALS_SXB` and `POP_GLOBALS_SXB`, were created to automate the process of copying the globals to this special bank and back. As long as this bank is never modified by the rest of the program, the globals can be safely saved and restored as necessary.

## 13.3 - Integrating the Tile Graphics Engine with SX/B

The Tile Graphics Engine integrates fairly smoothly with the SX/B compiler. Aside from a few details, the only real change is that `Game_Init` and `Game_Update` are written in BASIC, along with other game-specific subroutines. Think of the Tile Graphics Engine as a pure-assembly library of functions that can be added to a BASIC program to create graphical displays.

Integration of the Tile Graphics Engine with SX/B is best explained with actual examples. With that in mind, let's check out the following SX/B demo case studies.

### 13.3.1 - SX/B Case Study: Pong

The first demo written with the Tile Graphics Engine and SX/B was a Pong clone. To maximize the size of the playfield, the game is played vertically instead of the traditional horizontal layout, to take advantage of the higher vertical resolution of the Tile Graphics Engine's 16x24 screen.

The source code to Pong can be found here:

[Tile\\_Engine\sxb\pong\ sxb\\_pong\\_1\\_0.sxb](#)

**NOTE**

If you do not have have SX/B or the SX-Key programmer, a pre-assembled hex version of this program can be found in this directly as well, ready to be programmed to your XGameStation via XGS Micro Studio.

A screenshot of Pong can be seen in Figure 13.1.

**Figure 13.1 – SX/B Pong.**

### 13.3.1.1 - Integration with the Tile Engine

The first challenge in creating Pong was integrating the Tile Graphics Engine with SX/B. To make things simple, the project is kept in two separate files: ***pong\_gfx\_engine.sxc***, which is the original tile graphics engine, and ***sxb\_pong\_1\_0.sxb***, which contains the Pong game logic.

***pong\_gfx\_engine.sxc*** is a pure assembly file that is identical to the original tile engine, except for three sections, which have been moved into the ***sxb\_pong\_1\_0.sxb***. The constants were moved first, because they are needed by both the graphics engine and the SX/B program. The **Game\_Init** and **Game\_Update** subroutines were also moved, as they will now be implemented in BASIC, rather than assembly.

Lastly, the assembly file is included with the SX/B file using the **include** directive.

### 13.3.1.2 - The Game Logic

Within the game, the most important elements are the horizontal locations of each paddle, the horizontal and vertical location of the ball, the scores of each player, and the current game state. These are tracked with a small set of variables defined below:

game_state	VAR	BYTE	' Game state
bx	VAR	BYTE	' Ball X, Y location
by	VAR	BYTE	
bx_v	VAR	BYTE	' Ball X, Y velocity
by_v	VAR	BYTE	
p1_x	VAR	BYTE	' Player 1 location
p1_score	VAR	BYTE	' Player 1 score
p2_x	VAR	BYTE	' Player 2 location
p2_score	VAR	BYTE	' Player 2 score

Other variables are declared as well, for more specific purposes throughout the code. Check the source code for further details.

### 13.3.1.3 - Initializing the Game

Within the **Game\_Init** subroutine, all the main game elements are initialized. This includes the paddles, the score, the game state, and the ball's location and velocity:

```

' Reset both players' to the center of the screen
p1_x = 6
p2_x = 6

' Reset both players' scores
p1_score = 0
p2_score = 0

' Reset the ball to the center of the screen
bx = 7
by = 12

' Set the ball's velocity towards the player
bx_v = 1
by_v = 1

' Reset the game state
game_state = GAME_STATE_TITLE

```

Once the game elements are initialized, the background is drawn and copied to an offscreen buffer. As you'll notice, this block of code is still written entirely in assembly:

```

' Clear the screen
\M_FILL_SCREEN    #21, #COLOR_0, #COLOR_0 + 1

' Draw the interface
\M_FILL_ROW      #0,  #14,      #COLOR_0 + 2, #COLOR_0 + 4
\M_SET_TILE      #0,  #0,  #13,      #COLOR_0 + 2, #COLOR_0 + 4
\M_SET_TILE      #15, #0,  #15,      #COLOR_0 + 2, #COLOR_0 + 4

' Score 1
\M_SET_TILE      #2,  #0,  #22,      #COLOR_0 + 2, #COLOR_0 + 4
\M_SET_TILE      #5,  #0,  #13,      #COLOR_0 + 2, #COLOR_0 + 4

```

```

' Score 2
\M_SET_TILE      #10, #0, #23, #COLOR_0 + 2, #COLOR_0 + 4
\M_SET_TILE      #13, #0, #13, #COLOR_0 + 2, #COLOR_0 + 4

' PONG Centerpiece
\M_SET_TILE      #7, #0, #11, #COLOR_13 + 2, #COLOR_13 + 5
\M_SET_TILE      #8, #0, #12, #COLOR_13 + 2, #COLOR_13 + 5

' Draw the dotted line
\M_FILL_ROW      #12, #16, #COLOR_0 + 3, #COLOR_0 + 1

' Copy the background to an offscreen buffer
\M_COPY_SCREEN   #$00, #$00, #$00, #$60

```

As you'll notice throughout these case studies, graphics are still handled with assembly. These graphics calls could be made from BASIC, but in order to do so, each graphics engine subroutine would have to be wrapped in a BASIC subroutine, which would also have to change and restore the active memory bank. If you want to use the Tile Graphics Engine with an SX/B program that will be making many arbitrary graphics calls, this is a viable option. In the case of Pong, however, and the rest of these case studies, graphics are only handled during initialization and once during each frame.

### 13.3.1.4 - Game States

Pong is always running in one of a handful of possible states. It begins in **GAME\_STATE\_TITLE**, which displays a “**Press Start!**” string until the fire button is pressed. It then enters the **GAME\_STATE\_PLAY** state, which allows player input to control the paddles and moves the ball based on its velocity. When a point is scored on either side, the **GAME\_STATE\_SCORED** state is entered, which sets a timer that freezes the action for a brief period while the ball is reset and gives the players a chance to prepare for the next serve. Once either player reaches the maximum score of 15, the **GAME\_STATE\_OVER** state is entered, which displays a “**Game Over!**” message and once again waits for the player to press the fire button to restart the game.

### 13.3.1.5 - Updating the Game

At each frame, the game is updated based on the current state. For the gameplay state, this means reading in the joystick with a call to **Read\_Joysticks**, moving each player based on the position of the joysticks, and updating the ball’s position (**bx** and **by**), based on its velocity (**bx\_v** and **by\_v**). Collision detection is also performed by comparing the location of the ball to each wall and to the paddles. Upon collision, the ball’s velocity is reversed on the appropriate axis.

### 13.3.1.6 - Displaying the Score

Displaying the score in any game is a deceptively simple task, since the binary format of a score variable doesn’t easily map to the decimal format that a player would expect to see on the screen. There are many possible ways to solve this problem, ranging from the complex, direct approach of translating binary to decimal at each frame, to “hacked” solutions such as maintaining both a binary and decimal version of the

score; one is used for calculations within the game engine, while the other is used exclusively for display on the screen.

Fortunately, the score in Pong can never exceed 15, which gives us an easy way out. Using a lookup table of only 16 elements, each value from 0 to 15 can be converted from binary to BCD quickly and easily. The table looks like this:

score_trans:		
DATA	\$00	' 0
DATA	\$01	' 1
DATA	\$02	' 2
DATA	\$03	' 3
DATA	\$04	' 4
DATA	\$05	' 5
DATA	\$06	' 6
DATA	\$07	' 7
DATA	\$08	' 8
DATA	\$09	' 9
DATA	\$10	' 10
DATA	\$11	' 11
DATA	\$12	' 12
DATA	\$13	' 13
DATA	\$14	' 14
DATA	\$15	' 15

The score itself, stored in binary, works as-is as an index into the table. The result is its BCD equivalent, each nibble of which can be used almost directly as an index into the font table. This process is illustrated visually in figure 13.2.

**Figure 13.2 – Converting the binary version of the score to its BCD equivalent for display onscreen.**

Binary to BCD Conversion Table	
Score	→
<i>Used directly as offset</i>	
00	0
01	
02	
03	
04	
05	
06	6
07	
08	
09	
10	
11	
12	
13	
14	
15	

*Nibbles used as font table offsets*

### 13.3.1.7 - Conclusion

Pong is a simple but complete game that puts SX/B and the Tile Graphics Engine to good use. Try expanding the game by adding non-45 degree angles for the ball to follow depending on where on the paddle the ball hits. Other interesting ideas might be adding solid obstacles near the center of the screen capable of bouncing the ball back towards the player, or perhaps extra balls that appear randomly throughout the game, forcing the players to juggle.

### 13.3.2 - SX/B Case Study: Breakout

The next demo written with SX/B is a near-complete Breakout clone, which features a more ambitious design than Pong but is based on many of the same principals. A screenshot of Breakout can be seen in Figure 13.3.

The source code to Breakout can be found here:

[Tile\\_Engine\sb\breakout\ sxb\\_breakout\\_0\\_8.sxb](#)

**NOTE**

If you do not have have SX/B or the SX-Key programmer, a pre-assembled hex version of this program can be found in this directly as well, ready to be programmed to your XGameStation via XGS Micro Studio.

Figure 13.3 – SX/B Breakout



Breakout is a single-player game in which a ball is bounced off a paddle into an array of blocks. Each time the ball comes in contact with a block, the block is destroyed and the ball is bounced in a random direction. The player advances to the next level if the ball can be kept from falling into the pit at the bottom of the screen long enough for all of the blocks to be destroyed.

The key elements of Breakout are the player's score, which ranges from 0 to 9999, the location of the player's paddle, the ball's location and velocity, the array of blocks, and state data like the game's current state and the currently active level.

Having already seen the overall design of a game with SX/B and the integration of the Tile Graphics Engine in the last section, let's take a look at some of the major elements of Breakout, and a few of the more subtle complexities.

### 13.3.2.1 - The Block Array

Each level of Breakout presents the block array in a unique and colorful formation. For example, the first level is a simple design in which three rows of blocks are stacked vertically, while later levels feature more interesting shapes such as a happy face and the a sprite from *Space Invaders*.

In most Breakout games, an array is maintained in which each index corresponds to a block on the screen. Depending on the complexity of the game, each element in this array may store as much information as the block's status (destroyed or not destroyed), the block's color or texture, and its location on screen. This array is initialized to the correct pattern at the start of each level and is slowly changed over the course of the game as blocks are destroyed.

In the case of SX/B and the SX52, an array capable of storing all of this information in RAM would be impractical at best. The array could also be stored in SRAM, but new assembly routines would have to be written to read from this array in an efficient manner each frame.

In the case of the Tile Graphics Engine, however, the solution is built into the graphics themselves. Once the block array is drawn into the offscreen buffer, the tiles themselves can, for example, be for collision detection. As the ball moves through the screen, the tiles over which it's drawn can be read to find out if they're empty background tiles or block tiles. If a block tile is read, it's drawn over with a background tile and the ball is bounced in a random direction.

Unfortunately, in order to determine when all blocks are destroyed, this approach breaks down. The process of reading each tile on the screen to make sure no block tiles remain is prohibitively expensive in terms of clock cycles and would not be possible on a per-frame basis. Instead, a counter is set at the start of each level to the number of blocks in the level's block array formation. Each time a block is detected and destroyed, this counter is decremented. When the counter reaches zero, each tile can be safely assumed destroyed and the level is complete.

#### 13.3.2.1.1 - Storing the Level Data

Each block array is stored using the SX/B language's **DATA** statement, which can be used to store static program data as it can be in many BASIC dialects. In the case of SX/B, **DATA** is analogous to SX assembly's **DW** directive in that the data is stored in program memory. Data stored using this method can be read at any time using the **READ** command. Unlike most BASIC dialects, however, SX/B does not force this data to be read sequentially, but rather allows random access given a label pointing to the base of the data and an offset. The description for level 4 is shown below:

```
' **** LEVEL 4
' Green background
```

```

DATA COLOR_0

' Space Invader
DATA %000001000, %00010000
DATA %00000100, %00100000
DATA %000001111, %11110000
DATA %00011011, %11011000
DATA %001111111, %11111100
DATA %00101111, %11110100
DATA %00101000, %00010100
DATA %00000110, %01100000

```

The first element in any level's description is the background color. In the case of level 4, this is green. Following the background color are 16 words in which 8-bits of each are used to create a 16x8 bitmap.

### 13.3.2.1.2 - Reading and Displaying the Level Data

At the start of each level, this data is read and drawn to the offscreen buffer by the **Draw\_Level** subroutine. At the heart of this subroutine are two nested **FOR** loops that draw each row and column of the block array:

```

' Reset the block count
game_vars ( GAME_VAR_BLOCK_COUNT ) = 0

' Draw the block array
FOR temp_0 = 0 TO 7

    ' Get the row bitmap data
    temp_3 = temp_0 * 2
    temp_2 = temp_3 + 1
    temp_2 = temp_2 + game_vars ( GAME_VAR_LEVEL_OFFSET )
    READ level_data + temp_2, bx
    temp_2 = temp_3 + 2
    temp_2 = temp_2 + game_vars ( GAME_VAR_LEVEL_OFFSET )
    READ level_data + temp_2, by

    ' Draw the next row
    FOR temp_1 = 0 TO 15

        ' Get the row's block color
        READ row_base_colors + temp_0, temp_2
        temp_3 = temp_2

        ' Get the column's luminance
        bx_f = temp_1 AND %00000011
        IF temp_1.2 = 0 THEN skip_inv_luma
        bx_f = 3 - bx_f
        skip_inv_luma:

        ' Add the column's luminance
        temp_2 = temp_2 + bx_f
        temp_3 = temp_3 + bx_f
        temp_2 = temp_2 + 2
        temp_3 = temp_3 + 4

        ' Is the block solid?
        IF bx.7 = 0 THEN skip_draw_block

        ' Draw the block
        ASM
        PUSH_GLOBALS_SXB

```

```

    _BANK      BANK_SXB_VAR
    MOV        t0, temp_1
    MOV        t1, temp_0
    ADD        t1, #3
    MOV        t2, temp_2
    MOV        t3, temp_3

    _BANK      BANK_VIDEO
    M_SET_TILE t0, t1, #22, t2, t3

    POP_GLOBALS_SXB
    _BANK      BANK_SXB_VAR
    ENDASM

    ' Increment the block count
    game_vars ( GAME_VAR_BLOCK_COUNT ) = game_vars ( GAME_VAR_BLOCK_COUNT ) + 1

skip_draw_block:

    ' Shift the block row data
    \RL by
    \RL bx

    NEXT temp_1      ' Column

NEXT temp_0      ' Row

```

For each row in the pattern, the two halves of the bitmap data are loaded into **bx** and **by** (simply because those two variables happen to be free; don't confuse this with their use during gameplay as the ball's X and Y location). Once inside the row loop, the high bit of the 16-bit **bx:by** vector is read, and the vector is shifted by one bit to the left. If the bit is set, a block tile is drawn using a color that is determined by its location onscreen. If it is clear, nothing is drawn and the background shows through. Each time a block is drawn, the block counter is incremented by one. This is an easy way to determine how many blocks are present in the formation without explicitly including the data in the level description.

### NOTE

You will notice throughout the Breakout source code that an array called **game\_vars** is used occasionally where a single variable might be expected instead. This is due to a limitation in the beta version of SX/B used that prevents more than 18 variables from being declared. After 18, only arrays can be declared, which means all extra variables must be grouped into an array and accessed as individual elements where needed. This is scheduled to be corrected in time for the public release.

#### 13.3.2.1.3 - The Score

As in Pong, displaying the score is one of the minor complexities in Breakout. However, since the design of Breakout calls for a score that ranges from 0 to 9999, the lookup table solution presented in the last chapter would hardly be appropriate.

The Breakout score is implemented as four BCD digits stored in two bytes. Along with a third byte, called the **accumulator**, this data is stored in an array called **bcd\_score**. The accumulator, which will be

discussed shortly, is element zero of the array. Elements one and two are the low and high bytes of the score, respectively.

To increment a BCD value, the low nibble is incremented independently of the high nibble. The high nibble is then incremented in the event that the low nibble reaches a value greater than 9, and the low nibble is reset to zero. In the case of the Breakout score, which is 4 digits wide, and therefore 2 bytes wide, the high nibble of the low byte causes the low nibble of the high byte to increment when it exceeds 9.

The score is incremented by some small amount whenever the ball collides with a block. This point value could then be added to the 4-digit BCD value and the score would immediately update onscreen. However, to simplify both the logic of adding to the BCD score, as well as adding a cool “rolling” effect seen in many arcade and pinball games, a different approach is taken.

At each frame, if the accumulator (element zero of the `bcd_score` array) is greater than zero, the lowest nibble of the 4-digit BCD score is incremented. If this nibble exceeds 9, it is reset to zero, the next nibble is incremented, and the process continues. The accumulator is then decremented.

Using this method, points scored at any time are simply *added* to the accumulator (whatever its current value may be), and are gradually absorbed into the player’s score over the course of multiple frames. As stated above, this not only simplifies the process of incrementing the score, but creates an interesting effect.

### 13.3.2.2 - Conclusion

Breakout is a relatively complex game showing that a lot can be done with SX/B and the Tile Graphics Engine without an extreme amount of effort. Colorful graphics, engaging gameplay and multiple levels are no problem as long as a little bit of clever thinking is used. Tricks like using the screen buffer itself to manage the block array and the accumulator-based approach to the BCD score are effective in producing impressive results without the need for huge algorithms and data structures.

As an experiment, try expanding Breakout to include more complete game states, such as a title screen and a game over screen. For a more advanced challenge, try giving the ball a path that does not lie on a 45 degree angle. The ball already moves using an 8.8 fixed-point value and is already capable of such trajectories; you simply need to work it into the game logic.

### 13.3.3 - The Beginnings of an SX/B Game – Frogger

Pong and Breakout are examples of near-complete games wherein the graphics are handled by assembly, while the high-level game logic is written in BASIC with SX/B. The Frogger demo, however, also included with the SX/B demos, is an example of a game in which very little of the game logic has been written. Currently, the majority of the game is assembly, including the horizontal scrolling effect used to move the logs, turtles and cars, with only the basic beginnings of the game logic written in BASIC.

The source code to Frogger can be found here:

[Tile\\_Engine\sxb\frogger\ sxb\\_frogger\\_0\\_2.sxb](#)

**NOTE**

If you do not have have SX/B or the SX-Key programmer, a pre-assembled hex version of this program can be found in this directly as well, ready to be programmed to your XGameStation via XGS Micro Studio.

Check out Figure 13.4 for a screenshot of Frogger in action:

Figure 13.4 – SX/B Frogger.



At each iteration of the game loop, the logs, turtles and cars are scrolled by one tile using a special assembly subroutine. This subroutine quickly scrolls a given row of tiles by only moving the tile index itself; the foreground and background colors are left alone to save time. Because the background is always black, and every tile on the moving rows use the same foreground color, the scrolling routine can safely ignore them. See the `Scroll_Row_Left` and `Scroll_Row_Right` subroutines in the source code for details.

Currently, the only real task of the BASIC side of the source code is to handle the frog's movement based on the joystick. The following code reads the joystick, moves the frog, then performs collision detection to keep the frog from leaving the screen or disturbing the score and timer displays on the top and bottom.

```

' Get the joystick status
ASM
PUSH_GLOBALS_SXB
CALL    @Call_Read_Joysticks
_BANK   BANK_SXB_VAR
MOV     stick_status, t0
POP_GLOBALS_SXB
_BANK   BANK_SXB_VAR
ENDASM

' Move the frog and change its direction based on the joystick
IF stick_status.2 = 1 THEN no_move_left
p_dir = LEFT
p_x   = p_x - 1
no_move_left:
IF stick_status.3 = 1 THEN no_move_right
p_dir = RIGHT
p_x   = p_x + 1
no_move_right:
IF stick_status.0 = 1 THEN no_move_up
p_dir = UP
p_y   = p_y - 1
no_move_up:
IF stick_status.1 = 1 THEN no_move_down
p_dir = DOWN
p_y   = p_y + 1
no_move_down:

skip_input:
' ***** COLLISION DETECTION *****
' Translate the player coordinates to a 1-based coordinate system
p_x = p_x + 1
p_y = p_y + 1

' Perform bounds checking
IF p_x > 0 THEN no_clip_x0
p_x = 1
no_clip_x0:
IF p_x < 16 THEN no_clip_x1
p_x = 16
no_clip_x1:
IF p_y > 4 THEN no_clip_y0
p_y = 4
no_clip_y0:
IF p_y < 21 THEN no_clip_y1
p_y = 21
no_clip_y1:

' Translate the player coordinates to a zero-based coordinate system
p_x = p_x - 1
p_y = p_y - 1

```

Note that the coordinates of the frog are translated to and from a 1-based coordinate system. Normally, the frog's location is relative to zero (0,0 being the coordinates of the tile in the upper-left corner of the screen). This is fine for most purposes, but things get a bit tricky when the variables used to store the coordinates are 8-bit and unsigned. To make the comparison easier, the coordinates are temporarily

thought of as 1-16 and 1-24, instead of 0-15 and 0-23. This allows the frog's location to be compared to zero without worrying about wrap-around issues.

As you can see, Frogger is a decidedly simple example of SX/B, since the majority of the game is still the assembly framework. Fortunately, this also makes it the ideal candidate for expansion and modification. See how much of the actual Frogger game logic you can implement using the existing code as a starting point. The most logical places to start would be:

- Collision with the cars, logs and turtles
- A display for the player's remaining lives and a game over screen when all lives are lost
- The ability to "ride" the turtles and logs left and right to cross the pond
- Making the score and timer displays functional
- A title screen

With the graphical work already done, handling the game logic is a much simpler task, made even easier with the SX/B language.

## 13.4 - Conclusion

SX/B is a useful addition to the XGS ME programmer's toolkit. With its simplified syntax, it takes much of the headache out of assembly language programming and allows you to focus on more the high-level logic of your program and less of the low-level details.

Furthermore, by integrating SX/B with the Tile Graphics Engine, games with elaborate visuals can be created while still taking advantage of the simplicity of the BASIC language. We've seen how Pong and Breakout can be implemented more easily by mixing BASIC and assembly language.

## Appendix A: SX52 Instruction Set Reference

The following is a reference of the instruction set available on the SX52. Note that this information is also available in the XGS Micro Studio IDE with the **Instruction Browser** tool.

Instruction Set Reference Key	
Light Grey	These instructions cannot follow a <b>skip</b> instruction
Dark Grey	These instructions rely on the status of the carry flag if the <b>CARRYX</b> option is active. Make sure to set the carry flag to a known state before these instructions execute.
White	These instructions behave normally in all conditions.

SX52 Instruction Set				
Instruction	Words	Cycles	Affects	Operation
ADD fr, #Const	2	2	fr, w, C, DC, Z	fr = fr + Const (CLC)
ADD fr, w	1	1	fr, C, DC, Z	fr = fr + w (CLC)
ADD fr1, fr2	2	2	fr, w, C, DC, Z	fr1 = fr1 + fr2 (CLC)
ADD w, fr	1	1	w, C, DC, Z	w = w + fr (CLC)
ADDB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 + NOT fr2.Bit
ADDB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 + fr2.Bit
AND fr, #Const	2	2	fr, w, Z	fr = fr AND Const
AND fr, w	1	1	fr, Z	fr = fr AND w
AND fr1, fr2	2	2	fr1, w, Z	fr1 = fr1 AND fr2
AND w, #Const	1	1	w, Z	w = w AND Const
AND w, fr	1	1	w, Z	w = w AND fr
BANK fr	1	1	fsr	fr.(7-5) -> fsr.(7-5)

CALL Addr	1	3	pc	pc = Addr, Push (2)
CJA fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr > Const (CLC)
CJA fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 > fr2 (STC)
CJAE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr >= Const (STC)
CJAE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 >= fr2 (STC)
CJB fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr < Const (STC)
CJB fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 < fr2 (STC)
CJBE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr <= Const (CLC)
CJBE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 <= fr2 (STC)
CJE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr = Const (STC)
CJE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 = fr2 (STC)
CJNE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr <> Const (STC)
CJNE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 <> fr2 (STC)
CLC	1	1	C	C-Flag = 0
CLR !wdt	1	1	wdt, TO, PD	!wdt = 0, TO = 1, PD = 1 (1)
CLR fr	1	1	fr, Z	fr = 0
CLR w	1	1	w, Z	w = 0
CLRB fr.Bit	1	1	fr.Bit	fr.Bit = 0
CLZ	1	1	Z	Z = 0
CSA fr, #Const	3	3/4	w, C, DC, Z, (pc)	pc++, if fr > Const (CLC)
CSA fr1, fr2	3	3/4	w, C, DC, Z, (pc)	pc++, if fr1 > fr2 (STC)
CSAE fr, #Const	3	3/4	w, C, DC, Z, (pc)	pc++, if fr >= Const (STC)
CSAE fr1, fr2	3	3/4	w, C, DC, Z, (pc)	pc++, if fr1 >= fr2 (STC)
CSB fr, #Const	3	3/4	w, C, DC, Z, (pc)	pc++, if fr < Const (STC)

CSB fr1, fr2	3	3/4	w, C, DC, Z, (pc)	pc++, if fr1 < fr2 (STC)
CSBE fr, #Const	3	3/4	w, C, DC, Z, (pc)	pc++, if fr <= Const (CLC)
CSBE fr1, fr2	3	3/4	w, C, DC, Z, (pc)	pc++, if fr1 <= fr2 (STC)
CSE fr, #Const	3	3/4	w, C, DC, Z, (pc)	pc++, if fr = Const (STC)
CSE fr1, fr2	3	3/4	w, C, DC, Z, (pc)	pc++, if fr1 = fr2 (STC)
CSNE fr, #Const	3	3/4	w, C, DC, Z, (pc)	pc++, if fr >> Const (STC)
CSNE fr1, fr2	3	3/4	w, C, DC, Z, (pc)	pc++, if fr1 >> fr2 (STC)
DEC fr	1	1	fr, Z	fr = fr - 1
DECSZ fr	1	½	fr	fr = fr - 1, pc++, if fr = 0
DJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr - 1, pc = Addr, if fr >> 0
IJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr + 1, pc = Addr, if fr >> 0
INC fr	1	1	fr, Z	fr = fr + 1
INCSZ fr	1	1/2	fr, (pc)	fr = fr + 1, pc++, if fr = 0
IREAD	1	4	w, m	(m:w) -> m:w
JB fr.Bit, Addr	2	2/4	(pc)	pc = Addr, if fr.Bit = 1
JC Addr	2	2/4	(pc)	pc = Addr, if C = 1
JMP Addr	1	3	pc	pc = Addr
JMP pc+w	1	3	pc, C, DC, Z	pc = Addr+w (CLC)
JMP w	1	3	pc	pc = w
JNB fr.Bit, Addr	2	2/4	pc	pc = Addr, if fr.Bit = 0
JNC Addr	2	2/4	pc	pc = Addr, if C = 0
JNZ Addr	2	2/4	pc	pc = Addr, if Z = 0
JZ Addr	2	2/4	pc	pc = Addr, if Z = 1
MODE Const	1	1	m	m = Const

MOV !option, #Const	2	2	Option, w	Option = Const
MOV !option, fr	2	2	Option, w, Z	Option = fr
MOV !option, w	1	1	Option	Option = w
MOV !port, #Const	2	2	!port, w	Port-Config. = Const
MOV !port, fr	2	2	!port, w, Z	Port-Config. = fr
MOV !port, w	1	1	!port	Port-Config. = w
MOV fr, #Const	2	2	fr, w	fr = Const
MOV fr, w	1	1	fr	fr = w
MOV fr1, fr2	2	2	fr1, w, Z	fr1 = fr2
MOV m, #Const	1	1	m	m = Const
MOV m, fr	2	2	m, w, Z	m = fr
MOV m, w	1	1	m	m = w
MOV w, #Const	1	1	w	w = Const
MOV w, fr	1	1	w, Z	w = fr
MOV w, /fr	1	1	w, Z	w = NOT fr
MOV w, ++fr	1	1	w, Z	w = fr + 1
MOV w, <<fr	1	1	w, C	w = RL fr
MOV w, >>fr	1	1	w	w = SWAP fr
MOV w, >>fr	1	1	w, C	w = RR fr
MOV w, --fr	1	1	w, Z	w = fr - 1
MOV w, fr-w	1	1	w, C, DC, Z	w = fr - w (STC)
MOV w, m	1	1	w	w = m
MOVB fr1.Bit, /fr2.Bit	4	4	fr1.Bit	fr1.Bit = NOT fr2.Bit
MOVB fr1.Bit, fr2.Bit	4	4	fr1.Bit	fr1.Bit = fr2.Bit

MOVSZ w, ++fr	1	1	w	w = fr + 1, pc++, if w = 0
MOVSZ w, --fr	1	1	w	w = fr - 1, pc++, if w = 0
NOP	1	1	-	-
NOT fr	1	1	fr, Z	fr = fr XOR \$FF
NOT w	1	1	w, Z	w = w XOR \$FF
OR fr, #Const	2	2	fr, w, Z	fr = fr OR Const
OR fr, w	1	1	fr	fr = fr OR w
OR fr1, fr2	2	2	fr1, w, Z	fr1 = fr1 OR fr2
OR w, #Const	1	1	w, Z	w = w OR Const
OR w, fr	1	1	w, Z	w = w OR fr
PAGE Addr	1	1	PA2-0	PA2...PA0 = Addr.(11...9)
RET	1	3	pc	Pop (3)
RETI	1	3	pc, C, DC, Z	Pop, Restore (3) (4)
RETIW	1	3	pc, C, DC, Z	Pop, Restore, RTCC += w (3) (4)
RETP	1	3	pc	Pop, Restore Page (5)
RETW Const	1	3	pc	w = Const, Pop (3)
RL fr	1	1	fr, C	fr.(7..1) = fr.(6..0), fr.0 = C, C = fr.7
RR fr	1	1	fr, C	fr.(6..0) = fr.(7..1), fr.7 = C, C = fr.0
SB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 1
SC	1	1/2	(pc)	pc++, if C = 1
SETB fr.Bit	1	1	fr.Bit	fr.Bit = 1
SKIP	1	2	pc	pc++
SLEEP	1	1	TO, PD	TO = 1, PD = 0, Stop clock
SNB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 0

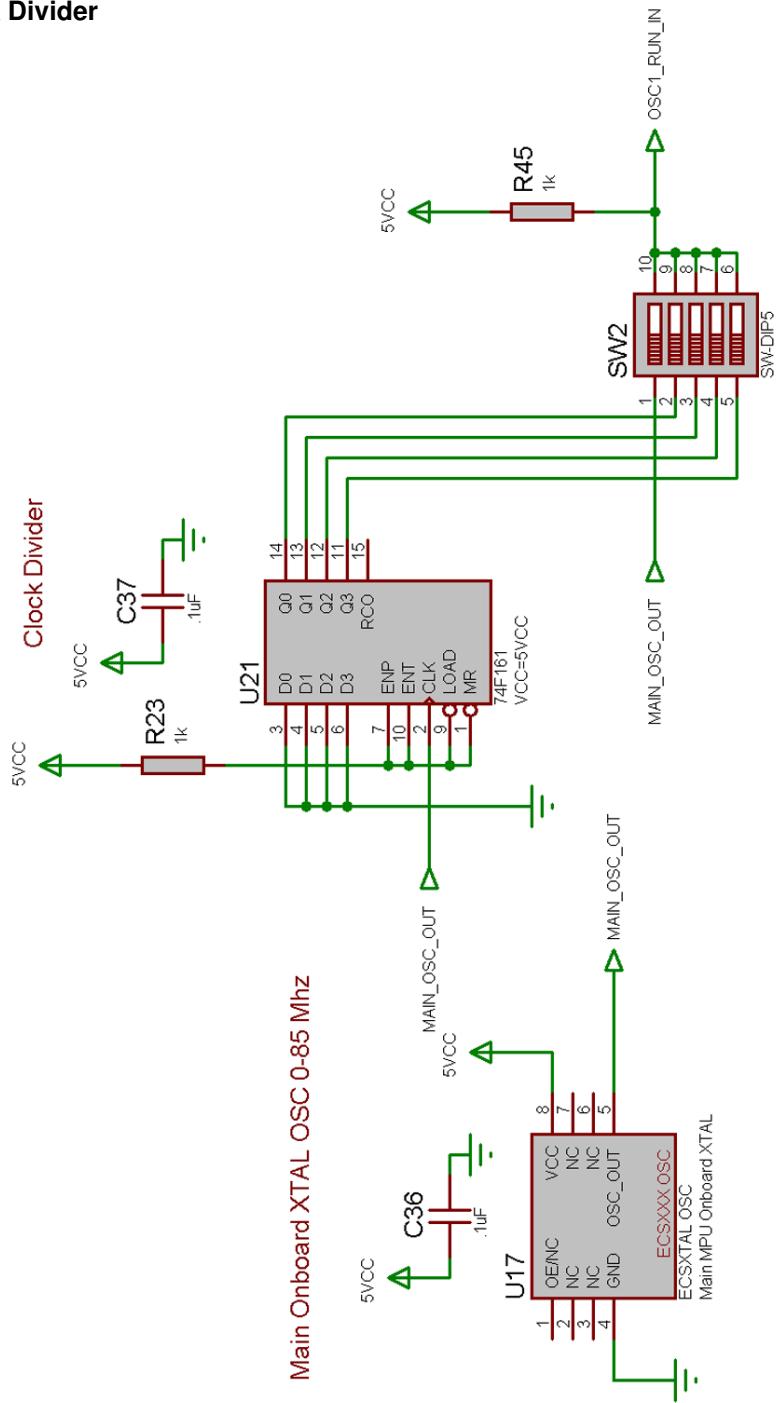
SNC	1	1/2	(pc)	pc++, if C = 0
SNZ	1	1/2	(pc)	pc++, if Z = 0
STC	1	1	C	C = 1
STZ	1	1	Z	Z = 1
SUB fr, #Const	2	2	fr, w, C, DC, Z	fr = fr - Const (STC)
SUB fr, w	1	1	fr, C, DC, Z	fr = fr - 1 (STC)
SUB fr1, fr2	2	2	fr1, w, C, DC, Z	fr1 = fr1 - fr2 (STC)
SUBB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 - NOT fr2.Bit (STC)
SUBB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 - fr2.Bit (STC)
SWAP fr	1	1	fr	fr.(7..4) = fr.(3..0), fr.(3..0) = fr(7..4)
SZ	1	1/2	(pc)	pc++, if Z = 1
TEST fr	1	1	Z	Z = 1, if fr = 0
TEST w	1	1	Z	Z = 1, if w = 0
XOR fr, #Const	2	2	fr, W, Z	fr = fr XOR Const
XOR fr, w	1	1	fr, Z	fr = fr XOR w
XOR fr1, fr2	2	2	fr, W, Z	fr1 = fr1 XOR fr2
XOR w, #Const	1	1	w, Z	w = w XOR Const
XOR w, fr	1	1	w, Z	w = w XOR fr

## Appendix B: XGS ME Schematic Reference

This chapter contains schematics that each describe one aspect of the XGS ME hardware. The following table lists each schematic and its page number.

<b>Clock Divider</b>	Figure B.1	Page 274
<b>Expansion Interface Map</b>	Figure B.2	Page 275
<b>Processor I/O Map</b>	Figure B.3	Page 276
<b>Joystick/Serial Interface</b>	Figure B.4	Page 277
<b>Keyboard Interface</b>	Figure B.5	Page 278
<b>3.3/5.0/12.5V Power Supplies</b>	Figure B.6	Page 279
<b>Onboard Programmer</b>	Figure B.7	Page 280
<b>Sound Subsystem</b>	Figure B.8	Page 281
<b>SRAM</b>	Figure B.9	Page 282
<b>Video Subsystem</b>	Figure B.10	Page 283

**Figure B.1 - Clock Divider**



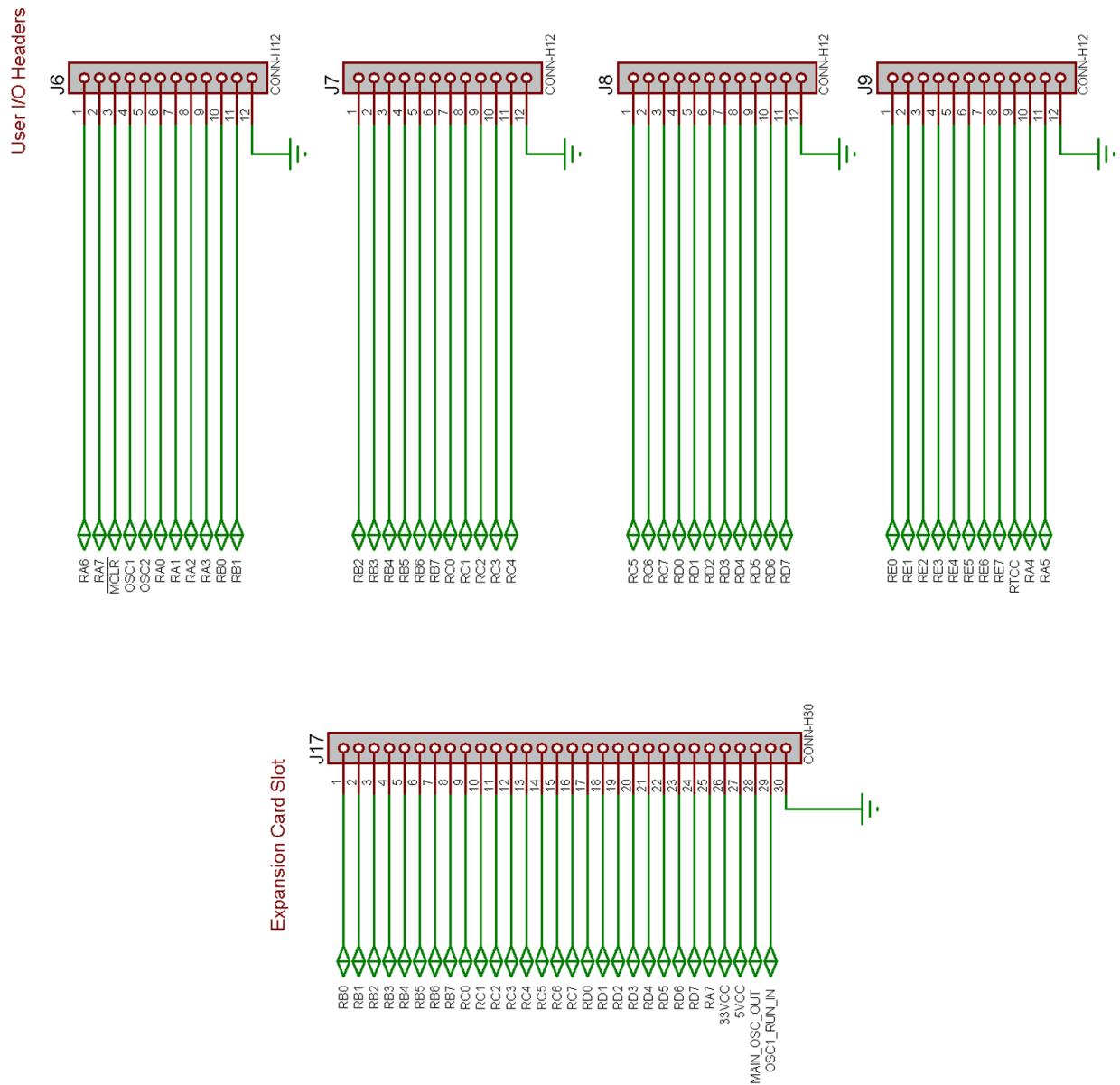
**Figure B.2 - Expansion Interface Map**

Figure B.3 - Processor I/O Map

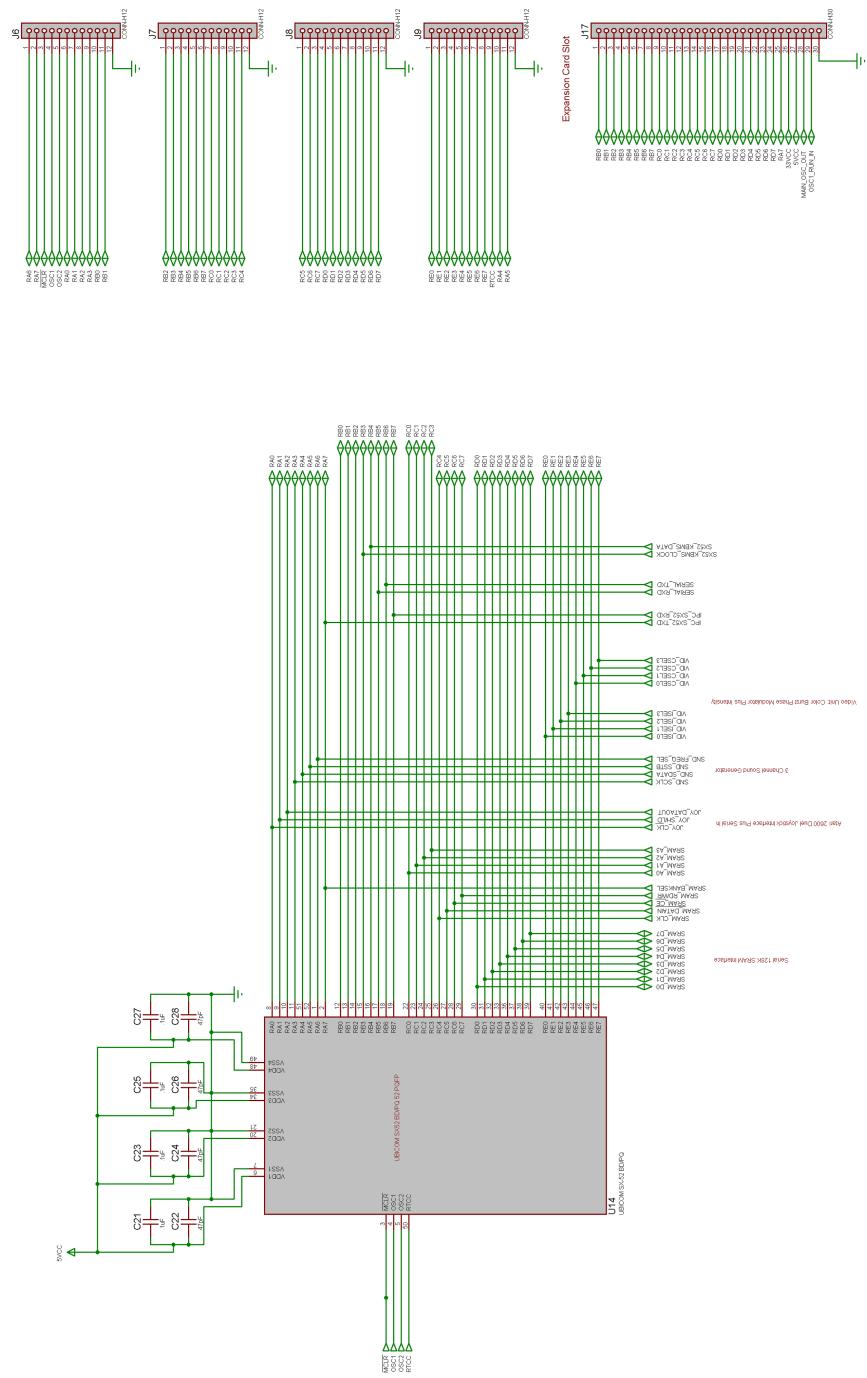


Figure B.4 - Joystick/Serial Interface

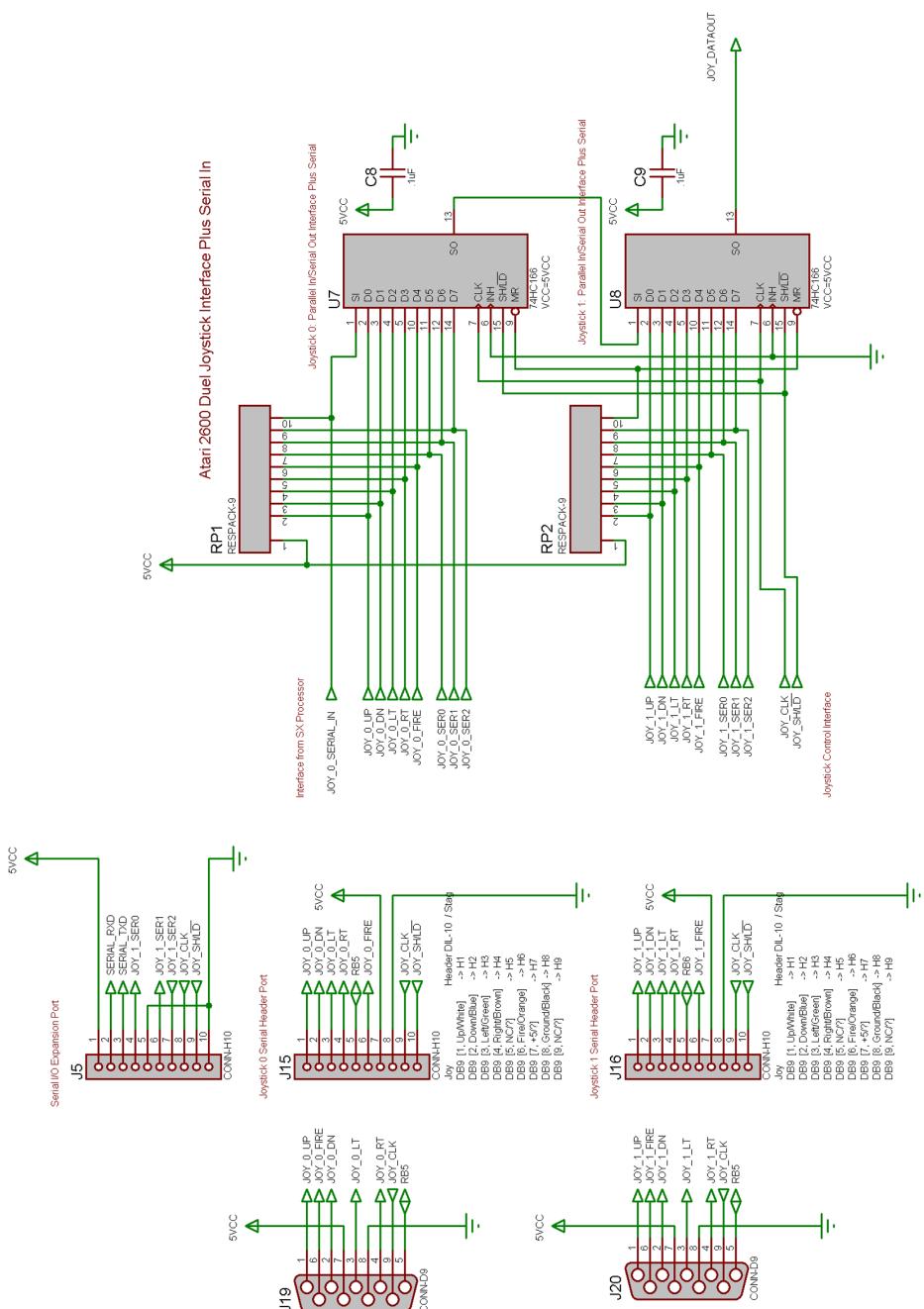
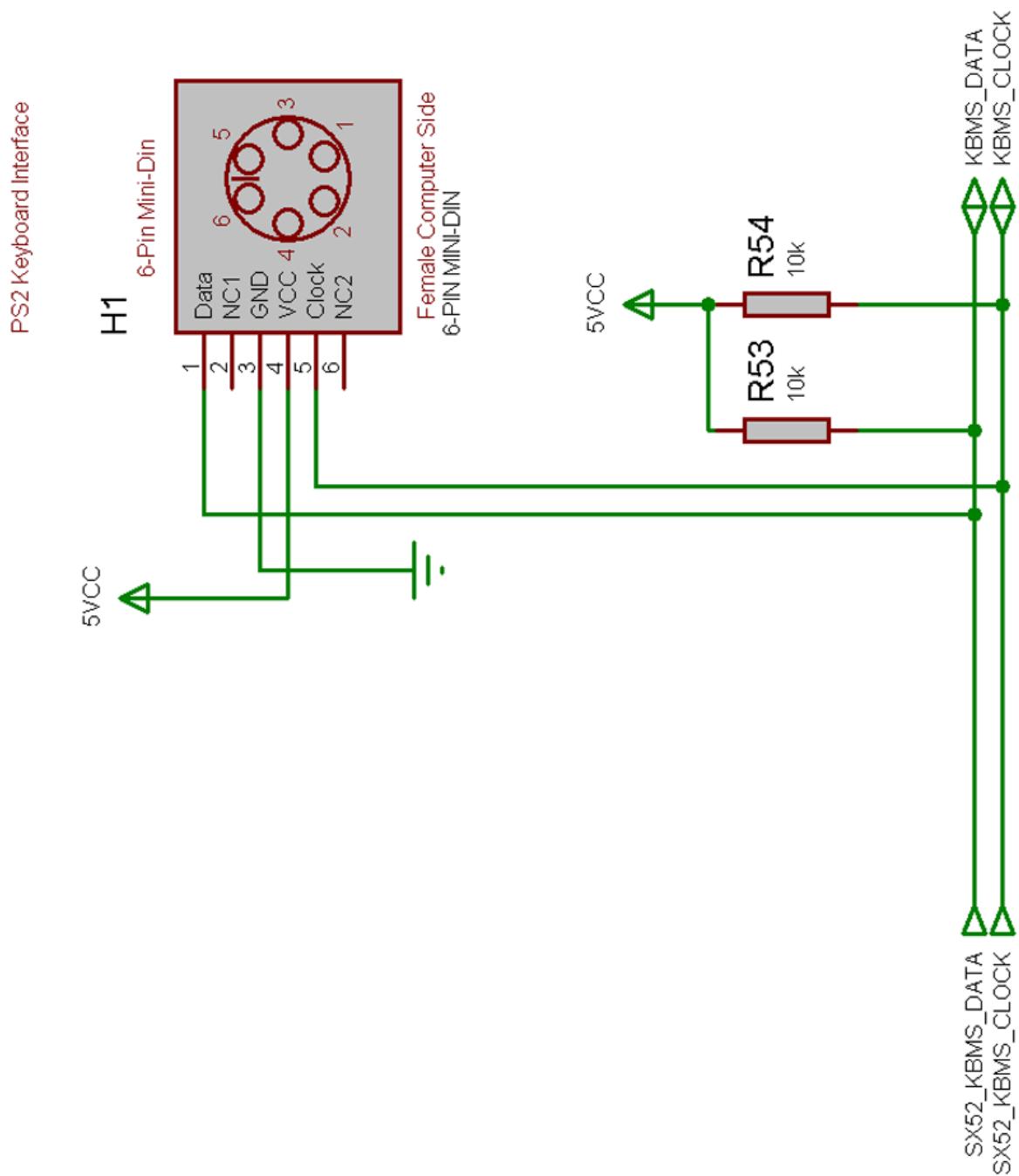
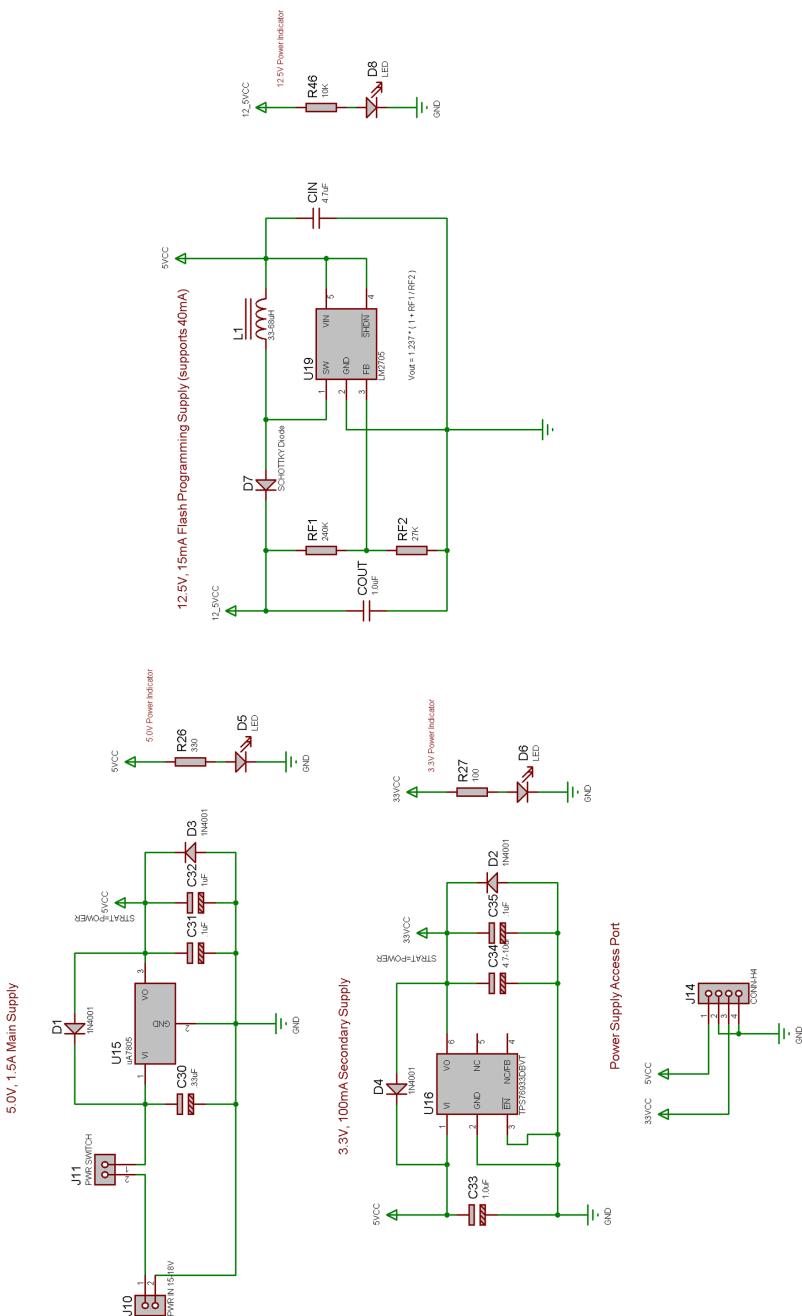


Figure B.5 - Keyboard Interface



**Figure B.6 - 3.3/5.0/12.5V Power Supplies**

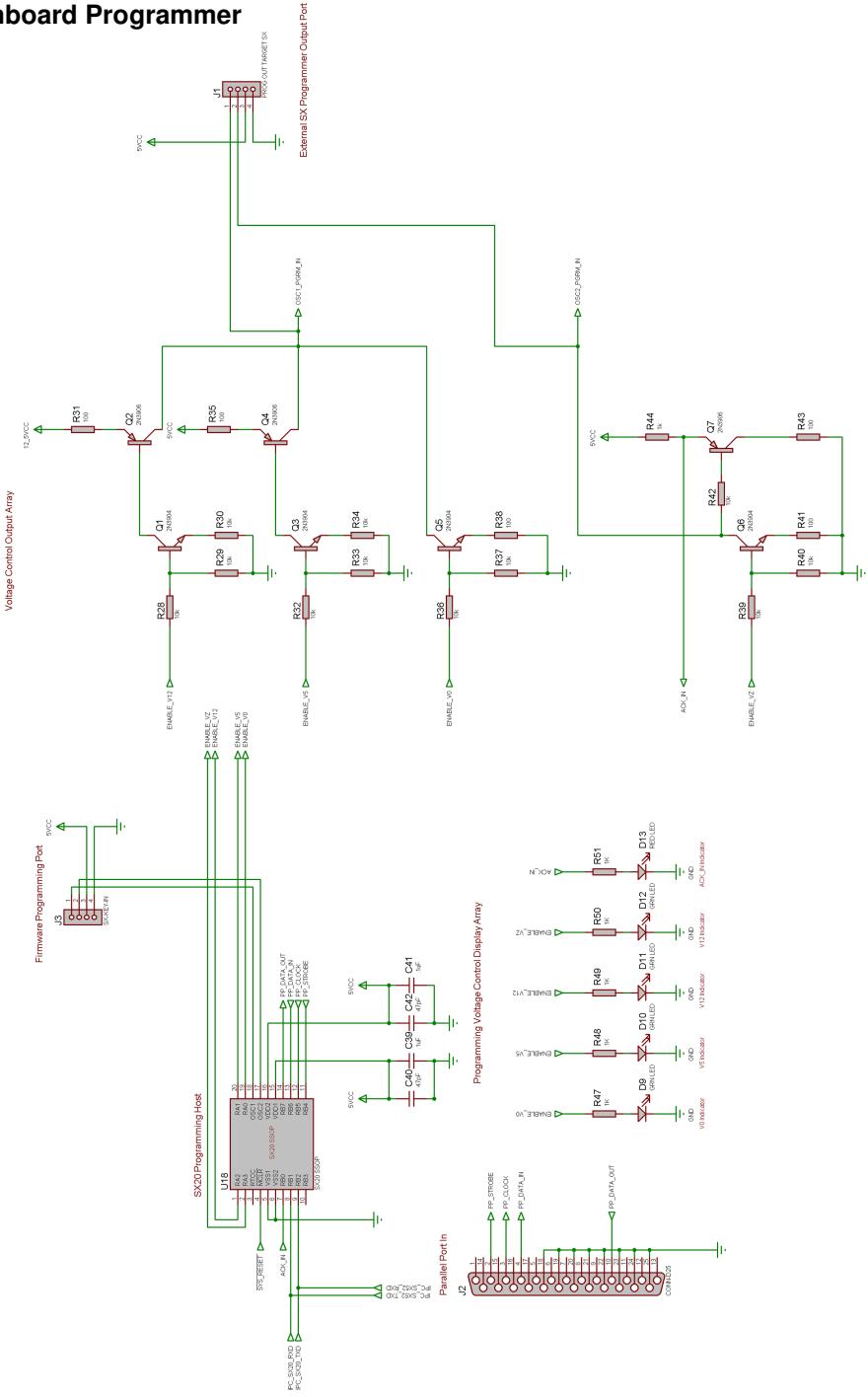
**Figure B.7 - Onboard Programmer**


Figure B.8 - Sound Subsystem

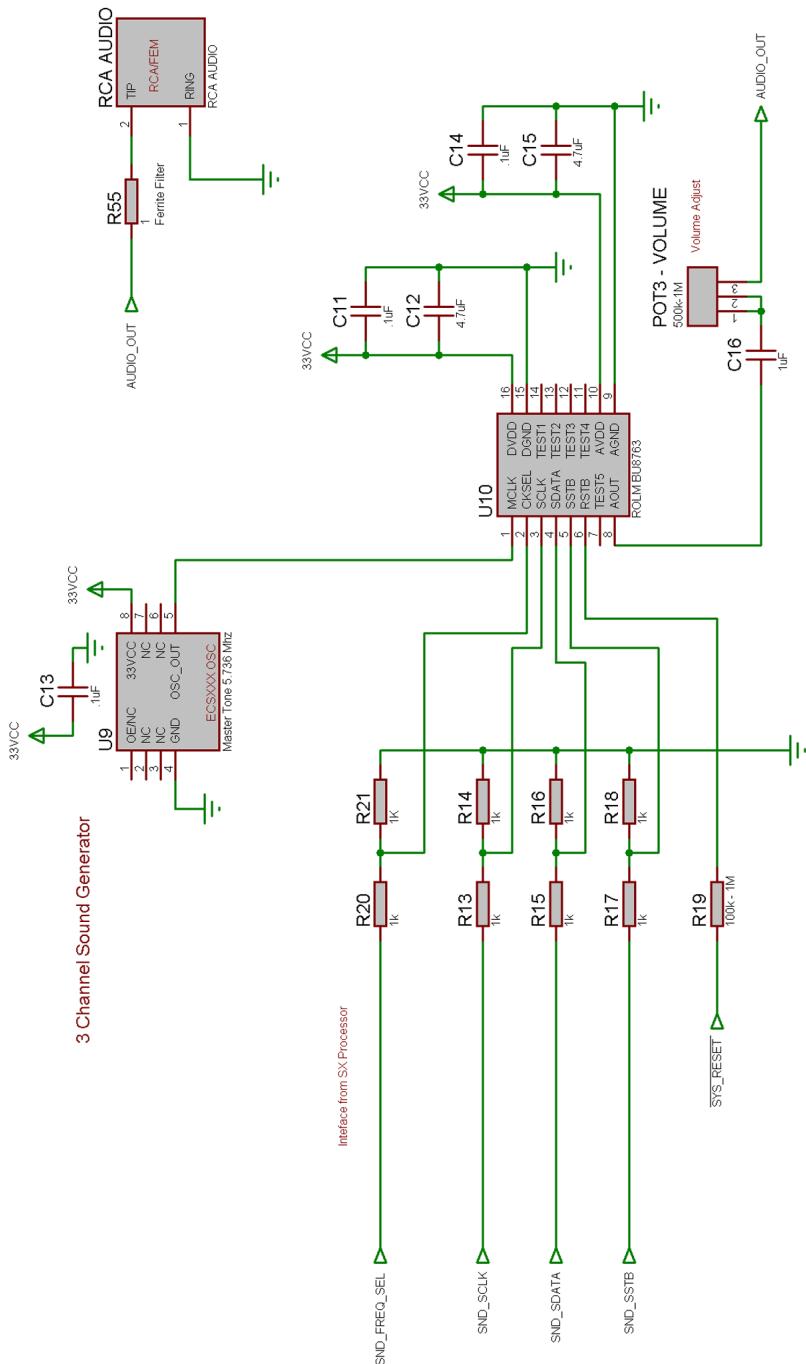


Figure B.9 - SRAM

Serial 128K SRAM Interface

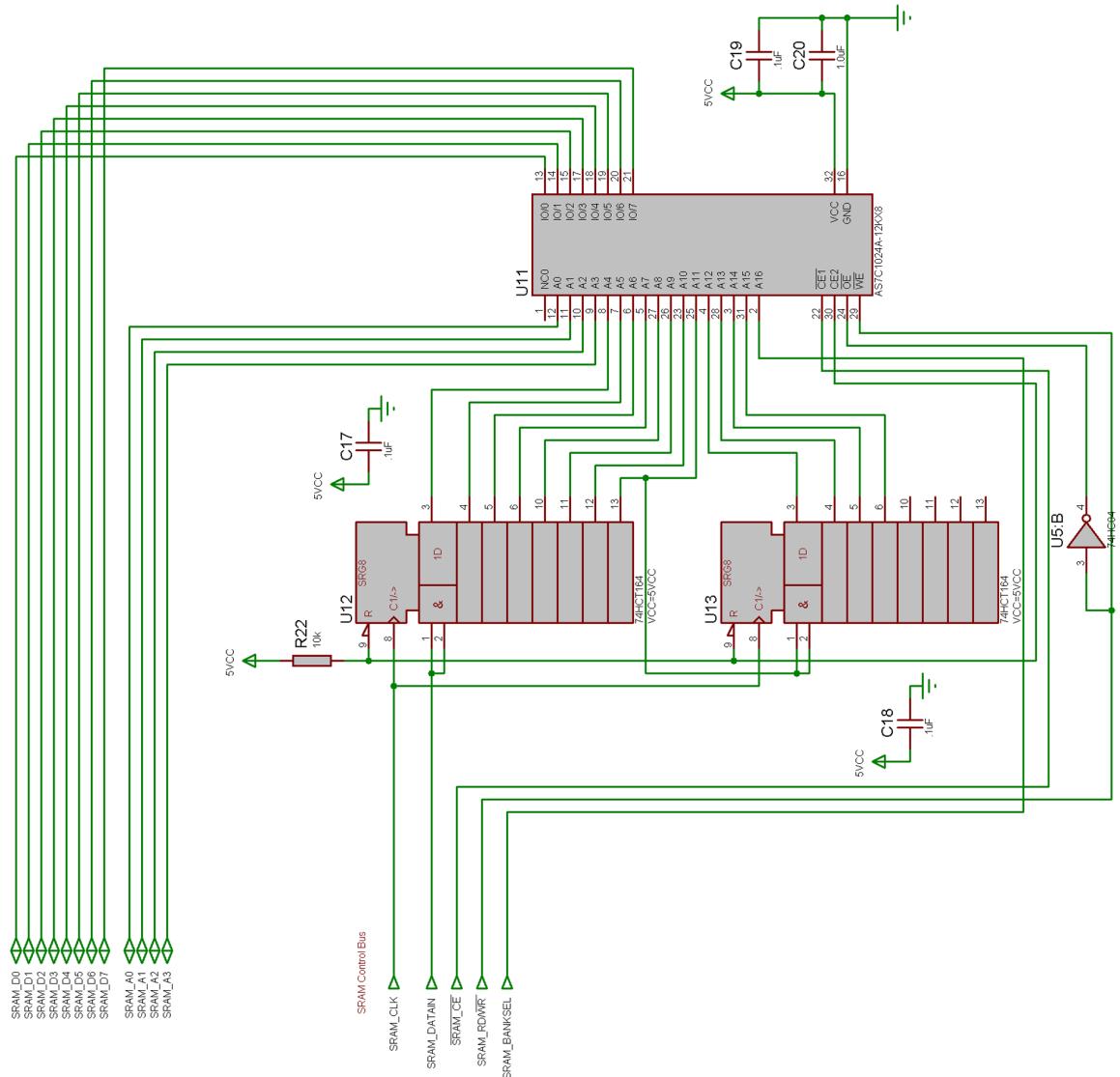
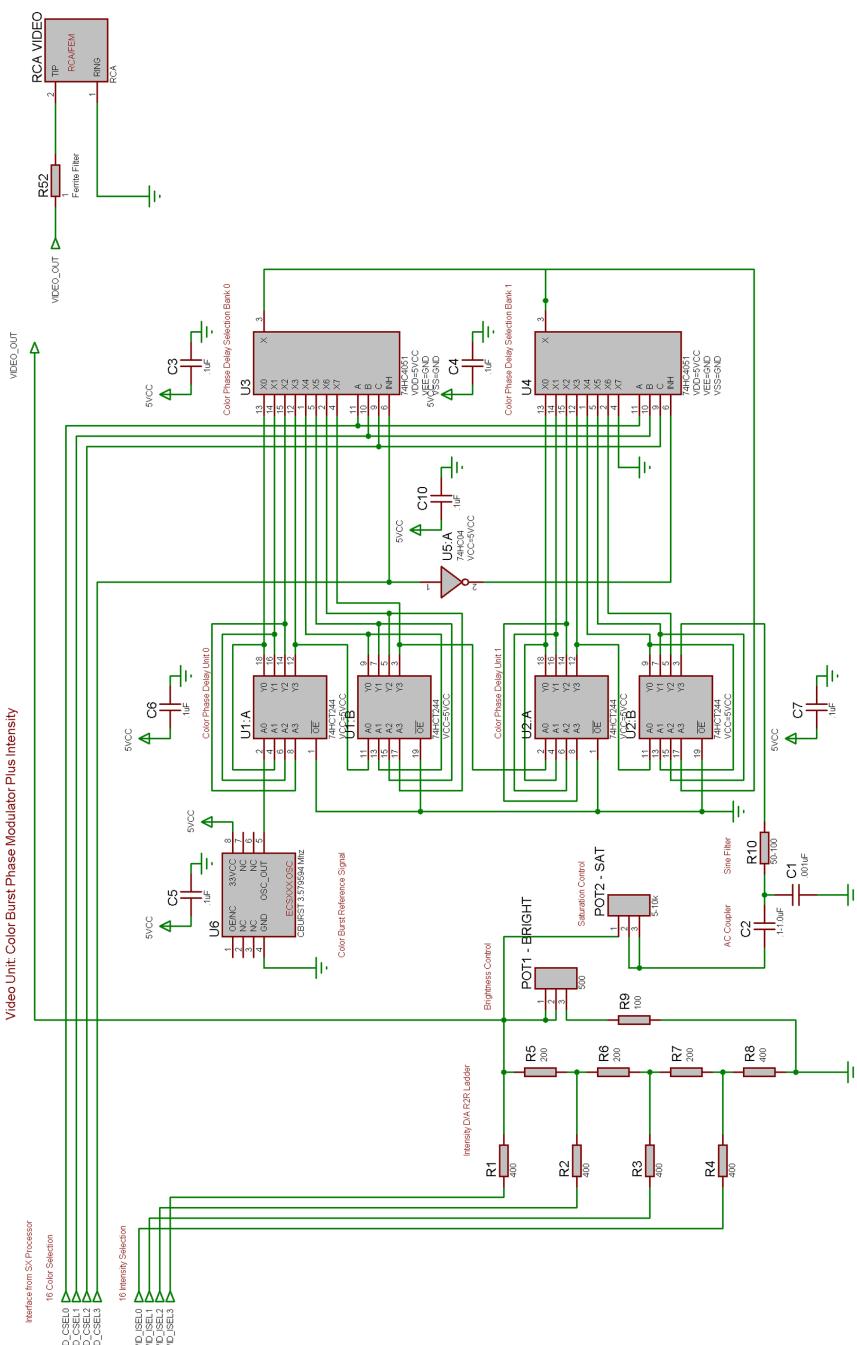


Figure B.10 - Video Subsystem



# Index

ADD instruction .....	267
ADDB instruction .....	267
AND instruction .....	267
Assembler Input .....	34
Assembler Output.....	35
assembly-language .....	6
Audio .....	9
BANK instruction .....	267
BCD.....	158
BIN16 .....	38
Bitmaps .....	106
breakpoint.....	65
brightness.....	13
Brownout .....	44
Built-in Programmer .....	14
CALL instruction .....	268
CJA instruction .....	268
CJAE instruction.....	268
CJB instruction .....	268
CJBE instruction.....	268
CJE instruction .....	268
CJNE instruction.....	268
CLC instruction .....	268
Clock .....	59
CLR instruction.....	268
CLRB instruction .....	268
CLZ instruction .....	268
COM .....	56
computer architecture.....	6
Crystal .....	44
CSA instruction.....	268
CSAE instruction .....	268
CSB instruction.....	268
CSBE instruction .....	269
CSE instruction.....	269
CSNE instruction .....	269
data structures.....	99, 111, 133, 134, 135, 136, 156
DB25 .....	1, 14
Debugger.....	60
DEC instruction .....	269
DECSZ instruction.....	269
Default Radix.....	36
Device String.....	44
digital electrical engineering .....	6
DJNZ instruction.....	269
Document Area .....	29
Document Selector.....	29
Editor Colors & Styles.....	35
Electro-Static Discharge .....	3
ESD .....	3
Expansion Bus .....	9
Extension Cords .....	3
Fire Cube.....	82
Firmware .....	1, 163, 164, 179, 180, 182, 183
Floormapper .....	84
FUSE.....	46
FUSEX .....	46
Fuzzy/Blurry/Noisy Video Output.....	13
Gamepad.....	1, 2
Hacking .....	99
Hardware .....	35
Heat Sinks .....	4
height map.....	109, 110, 111, 113, 148, 152
I/O .....	9
IEEE695 .....	38
IJNZ instruction .....	269
INC instruction.....	269
INCSZ instruction .....	269
INHX16 .....	38
INHX32.....	38
INHX8M .....	38
INHX8S .....	38
Initialization.....	137
Instruction Browser.....	47
IREAD .....	150
IREAD instruction .....	269
JB instruction.....	269
JC instruction.....	269
JMP instruction.....	269
JMP PC + W .....	138
JNB instruction .....	269
JNC instruction .....	269
JNZ instruction .....	269
Joystick.....	1, 2, 148, 157, 158
JZ instruction .....	269
line numbers.....	33
lookup tables .....	137, 138, 150, 159
LPT1 .....	14, 41
Mnemonic Set .....	35
MODE instruction .....	269
MOV instruction .....	270
MOVB instruction .....	270
MOVSZ instruction .....	271
Ms. Pac Man .....	120
NOP instruction .....	271
NOT instruction .....	271

NTSC.....	6, 68, 76, 81, 99, 128
Object Code Format.....	37
OR instruction.....	271
Oscillator .....	44
Output Window.....	29
PAGE instruction.....	271
PAL .....	6, 68, 76, 81, 95, 99, 106, 122, 123, 128
parallel port.....	14
PCB.....	6
Perspective.....	153
PGM mode .....	14
Plasma .....	86
Poll .....	63
Pong .....	87
potentiometers.....	13
power supply Detaches .....	4
Power Supply .....	1
printed circuit board.....	6
Programmer Unit .....	9
Raycaster .....	88
RCA cable .....	10
RCA Cable .....	1
Receptacles.....	3
Rem Colors .....	89
Reset Timer.....	44
RET instruction.....	271
RETI instruction.....	271
RETIW instruction .....	271
RETP instruction .....	271
RETW.....	137
RETW instruction .....	271
RISC.....	6
RL instruction .....	271
Rotozoomer.....	90
RR instruction.....	271
Run.....	63
SASM .....	56
saturation.....	13
SB instruction .....	271
SC instruction .....	271
SETB instruction.....	271
sine..	83, 86, 89, 95, 97, 101, 102, 103, 109, 122, 126, 156
SKIP instruction.....	271
SLEEP instruction .....	271
SNB instruction.....	271
SNC instruction .....	272
SNZ instruction .....	272
Square Wave.....	110
SRAM.....	9, 100, 148, 152
Starfield .....	92
Status Bar.....	29
STC instruction.....	272
Step .....	63
STZ instruction .....	272
SUB instruction.....	272
SUBB instruction .....	272
SWAP instruction .....	272
SX Programming API ....	162, 163, 164, 165, 166, 167, 168, 170, 171, 174
SX18.....	44
SX48.....	44
SX-Key .....	2, 14
SX-Key IDE .....	14
SYSMODE .....	14
SZ instruction .....	272
TEST instruction.....	272
Tetris .....	93, 94
text string.....	100, 123
Toolbar .....	29
Troubleshooting.....	67
Ubicom .....	35
User-Level Interface .....	163
Video .....	9
video kernel.....	136, 139
Walk .....	63
Wall outlets.....	3
Warranty.....	5
XGameStation Micro Edition .....	1, 6
XGS ME .....	6
XGS Micro Studio IDE.....	14
XOR instruction .....	272

---

## Notes

---

## Notes

---

## Notes