EsduinoXtreme User Guide

Introduction

The latest Arduino format board from Technological Arts is implemented with a full-featured Freescale S12 microcontroller having Flash, EEPROM, and SRAM memories, SCI, SPI, and CAN communications subsystems, dual 8-bit DACs, sophisticated 16-bit timer channels, PWM, and multi-channel 12-bit analog-to-digital conversion capability. EsduinoXtreme brings these features within easy reach of engineers, educators, and hobbyists. Unlike other "hobby" boards which may use sub-standard parts and undergo minimal testing, EsduinoXtreme is manufactured to industrial specs and is suitable for OEM use.

This guide provides basic setup and operation instructions for EsduinoXtreme. Included are both the hardware and software information needed to get the board working with a variety of programming languages and development tools. The first step in preparing to use EsduinoXtreme is to read through this manual, before attempting to apply power to the device.

***WARNING**: The board contains electrostatic sensitive components, and it is recommended that standard electrostatic precautions be taken when handling the module. There does not have to be a visible spark for a dangerous voltage to affect the electronics. Just walking accross a carpet on a dry day is enough to build up a potentially damaging amount of static charge. Recommended precautions include using a wrist grounding strap and/or a grounded workstation. The module can also be installed in a protective housing to keep it isolated from undesired external voltage sources.*

Overview of features

EsduinoXtreme comes equipped with a variety of interfaces and on-board resources. These include, but are not limited to, the following:

- 8 MHz crystal, providing a 4 MHz nominal bus speed, when enabled. This can be mutliplied by six, using the on-chip PLL, resulting in a 24 MHz bus speed
- 1 MHz internal oscillator is the default clock source following powerup, before the oscillator is enabled. This minimizes current draw and EMI.
- Memory resources: 240K Flash, 4K EEPROM, 11K RAM
- Operating voltage: 5V or 3.3V, jumper-selectable
- Power sources: via USB port by default; can be powered externally via optional input jack through the on-board 5V regulator using an external 6-12 VDC power source
- USB communication interface to host computer: via FTDI chip, utilizing microB USB connector
- User LED: red LED on DIG13 (Port J bit 0) available for application use
- On-chip development support: adaptation of Freescale AN2548 Serial Monitor
- BDM connector: 6-pin standard header enables the use of a separate BDM pod for for programming all Flash and EEPROM memories, and debugging programs in real-time
- Input/output headers: emulate the Arduino Leonardo pinout, with four of the signals user-assignable via jumper blocks
- Footrpint for optional user-installed SPI memory-device (EEPROM or Flash in SO-8 package)
- Optional header footprint for adding wireless communications via the Digi XBee module, using the Technological Arts ADXB adapter

Setup and Verification of Hardware

Power and serial connections, default jumper positions

There are two basic ways to use EsduinoXtreme in development, which are described in detail below:

1. On-chip Serial Monitor with uBug12:
   You will need a microB USB cable in order to establish communications between EsduinoXtreme and your host computer. The cable can also provide power for the board, to a maximum of 500 mA, supplied by your computer's USB port. (Note: if you plug EsduinoXtreme into an upowered USB hub, the available current will be less than 500 mA, depending on what else is plugged into the hub.)
2. External Background Debug Module (BDM) pod:
   Use a BDM pod, such as our USBDMLT, to program and power the board via your computer's USB port. There is a jumper block on USBDMLT which you can use to select 3V or 5V operation.

If you want EsduinoXtreme to run your program without a computer attached, you will need to supply power through one of several methods, detailed in the next section. Refer to the board's schematic diagram and feature sheet for details.

There are several jumpers placed on the board as well as pads on which to place jumpers. When the board is first used, there are only a few jumpers that need to be checked. These are listed below, along with their default settings. (A full jumper list is provided later.)

JB1 selects the voltage source for running the board. The factory default is USB, meaning that 5V from your computer's USB port will supply power to everything on the board. The VIN position would only be used if you were to supply power externally via optional 2-pin Molex header J7, or optional barrel jack J6, or via power pins on header J4.

JB3 through JB6 provide user-selectable signal assignments to some of the J1 and J3 header pins, and can be moved to the desired position for specific applications. Their configuration does not affect the basic operation of the board.

JB7 selects the operating voltage of the MCU. A shorting plug must be in place in one of the two positions in order for the MCU to operate. If it is missing, the MCU will not run (unless a BDM pod is connected).

JB8 selects the source of the Voltage Reference input pin (VRH) on the MCU, which is related to the analog-to-digital converter subsystem. The default position is VDD, meaning that analog voltage conversions will be referenced to the MCU operating voltage (set by JB7, described above). Leaving this jumper off will result in undefined operation of the analog-to-digital converter subsystem.

JB9 is a three-way selector block for communications between the host computer's USB port, the SCI0 pins of the MCU, and the optional XBee RF module interface. The default positions are US, meaning that the USB interface chip's RX and TX lines are connected to the MCU's RX0 and TX0 pins (i.e. SCI0). Refer to the schematic diagram and feature sheet for details on other possible configurations.

USB Interface

EsduinoXtreme can communicate with a PC using one of the MCU's on-chip Serial Communication Interfaces (SCIs) through a USB-to-UART interface chip made by FTDI. If you plan to use this interface, you'll need a microB to USB-A cable. The first time you plug the board into your computer, it will need to install USB drivers for the USB interface chip. If you are using Win7 or later, the operating system should find the necessary drivers and install them automatically. However, if you're using WinXP or some other OS, you'll need to download the drivers from www.ftdichip.com/vcp and install them yourself. After successful driver installation, you can determine which virtual com port Windows has assigned to your board by using Windows Device Manager (via MyComputer/Properties/Hardware) and looking at the comport list. It will

show you the USB com port assignment number (e.g. COM6). You'll need to know that in order to access it through a terminal program or via uBug12 with the built-in monitor program.

Using the Factory-Installed Demo Program

EsduinoXtreme ships from the factory pre-programmed with a simple SCI demo program in Flash. You can run this by placing the **Load/Run** switch (SW2) in the **Run** position when applying power or pressing reset. The demo program requires use of an ASCII terminal program (such as HyperTerminal, on Windows systems) to provide a means for you to interact with the program. Besides setting the correct ComPort (refer to http://support.technologicalarts.ca/docs/USB%20Boards/ConfiguringVirtualCOMports.pdf for help with this), you'll need to configure the terminal program with the following properties:

- Emulate ANSI terminal type
- 9600 baud
- 8 bits
- One stop bit
- No parity
- No flow control
- Do not append line feeds to incoming line ends
- No local echoing of typed characters

When the SCI demo program starts, it activates the red LED on the board and sends the following text out the serial port, which consequently appears in your terminal window:

```
Technological Arts - EsduinoXtreme SCI demo
Input String:
```

Go ahead and type a character string of up to 20 characters and press <enter>. Your string will then be echoed back in the terminal window, and you'll be prompted to enter a decimal number. Do so, followed by enter, and then enter a hex number when prompted. Each time, the value you entered will be echoed back in the terminal window (the size of the numbers allowed max out at 16-bit). Here's a typical session:

```
Technological Arts - EsduinoXtreme SCI demo
Input String: good morning!  Output String= good morning!
Input Decimal: 32768  Output Decimal= 32768
Input Hex: f07e  Output Hex= F07E
Input String:
```

Each time through the session, the program toggles the LED. If you'd like to examine the project, you'll find it (EX_SCI0) and other example CodeWarrior projects in our Support Library, at http://support.technologicalarts.ca/docs/Esduino/EsduinoXtreme/Code/CodeWarrior/C/

The demo program serves as a reasonable confirmation that your board is working properly. Now it's time to learn how to load other programs. There are two basic ways to do this, and we recommend using the first one unless you already know that you want to use a BDM pod.

1) Using uBug12

EsduinoXtreme comes pre-programmed with the Technological Arts version of Freescale's Serial Monitor program in Flash. After reset, the setting of switch SW1 is examined by the code and the following actions result:

1. If the switch is closed (i.e. LOAD position) control is passed to the Serial Monitor, whcih initializes various parameters of the MCU and waits for commands to be received on the serial port, via the USB connection. Since the SerialMonitor uses a binary interface (i.e. non-ASCII), it requires a companion application to be running on the host computer to provide an interface to the user. Technological Arts has created a freely-downloadable Java program called uBug12 for this purpose, which can run on Windows, Mac, and Linux hosts.
2. If the switch is open (i.e. RUN position), control is passed to the user program in Flash, via the user program Reset vector. If the user reset vector has not been programmed, control will revert to the monitor, as described above.

To use uBug12, you will first need to download and install it. Refer to the uBug12 manual for instructions and a link to the downloadable file.

When you are ready to proceed, place switch SW1 in the Load position and connect a microB USB cable between your computer and EsduinoXtreme.

You'll find detailed instructions on how to use uBug12 at this URL:

http://www.technologicalarts.ca/shop/documentation/63-debugging-tools/171-ubug12je-user-manual.html

2) Using a USBDMLT pod

1. Ensure that you have downloaded and installed the latest USBDM project from SourceForge
2. Use the firmware updater, if necessary, to ensure that you have the latest version of the USBDM firmware in your pod (at least 4.10.7) so that S12GA240 is included in the devices list.
3. Plug your USBDMLT pod into an available USB port on your PC
4. Launch the application called HCS12 Programmer that is included with the USBDM software
5. The program GUI has three tabs. On the Interface tab, you'll see the Select BDM field, which should now be displaying USBDM-JS16-0001, indicating that it has detected the pod. If not, click the Detect button to detect it. If it can't detect your pod, youo'll need ot visit the Help page of the USBDM project to resolve the issue
6. Next click the Target tab. Plug the 6-pin ribbon cable into your EsduinoXtreme BDM port. Note that this is the right-angle male header directly adjacent to the J1 header. Do not use the 6-pin vertical header adjacent to the slide switch-- that one is the SPI port. Ensure that the cable is plugged in the correct way: red stripe closest to the J1 header (i.e. the ribbon cable is exiting from the bottom of the connector). You may wish to snap the optional plastic strain relief onto the ribbon cable connector to make it easier to grasp for plugin and removal. Ensure that the jumper is in place on the USBDMLT pod, selecting either 3V or 5V operation. That way the pod will power your EsduinoXtreme via the USB port of your PC. Should you decide to use an external power source later (e.g. battery), you can remove the shorting plug entirely from the USBDMLT.
7. On the Target tab of the programmer GUI, click Detect Chip. The Device Selection box should update automatically to display GA-MC9S12GA240. If not, you'll need to do some troubleshooting to figure out why not.
8. At this point, you're ready to load a program into flash, so click on the Load Hex Files button to open a Windows file explorer from which you'll select the file you wish to load. (Just leave the default settings. You can read up on the options later in the USBDM project documentation if you want to.)
9. If you haven't already done so, downlaod and unZip one or more of the example projects for EsduinoXtreme from our Support Library at http://support.technologicalarts.ca/docs/Esduino/EsduinoXtreme/Code/CodeWarrior/C/
10. Using the file explorer opened previously via the HCS12 Programmer GUI, navigate to a project folder and the subfolder called **bin**. There you'll find the .s19 file to load (usually it's called Project.abs.s19). A suggested project to start with is Example_RTI. Load the Project.abs.s19 file found inside that project's bin folder.
11. Leave unchanged the defaults for Bus frequency and Security, and ensure EraseMass is selected under Erase Options in Device Operations. Click the Load and Go button and a second or two later you will see a Programming Completed message popping up. You should also notice that the red LED on your EsduinoXtreme is blinking rapidly. If you connect up another LED, in series with a current-limiting resistor to ground, with its anode connected to the DIG2 pin on the J1 header, you'll see it blink at a 0.5Hz rate.

Note that, once you have loaded a program with USBDMLT, the serial monitor that was pre-programmed into flash at the factory will no longer be present. If, for some reason, you wish to use the serial monitor instead of USBDMLT, you'll need to load it back in. Follow the steps from 4 to 11 shown above, and load the serial monitor s19 file found in the Support libary at http://support.technologicalarts.ca/docs/Esduino/EsduinoXtreme/Code/SerialMonitor/

Application Programming

There are several language options available for writing application programs for EsduinoXtreme. One can use Assembler, C, or BASIC, or a combination of these. This section lists descriptions and hardware specific setup instructions for the various options available. The choice of what to use is left to the user. Generally, though, it will be based on application requirements, budget, and what the programmer is familiar with or willing to learn.

Keep in mind that there is a difference between a programming language and an Integrated Development Environment (IDE). An IDE combines several functions of the development process into one program. It can be designed to work with a specific language, like C or Forth. However the language does not define the IDE functionality. An IDE can include an editor, assembler, compiler, simulator, debugger, serial monitor interface or BDM pod interface, or some subset of these capabilities. It will depend upon the IDE. So review what each can do before choosing one to work with. An IDE can be useful because it can combine several development steps into one, or allow blending of the functions. (e.g. source-level debugging, or using both assembly and the featured language.)

To assist you in getting a better idea of what is available, here is a table of the major development platforms that are available, and that can be used to develop software for EsduinoXtreme:

	Language	Assembler	IDE	Terminal Program	BDM Com	Other
CodeWarrior Special	C	S12	x		x	free 32K limit for C unlimited assembler
CodeWarrior Pro	C	S12	x		x	commercial product
HSW12	Assembler	S12	x		x	Linux
AsmIDE	Assembler	S12	x	x		Windows
MiniIDE	Assembler	asm12	x	x		Windows
SBASIC	BASIC	as12				DOS-based
GCC-Syncode	C	as12				Windows
GCC-Eclipse	C	as12				Windows
Imagecraft C	C	as12	x	x	some	commercial product
Cosmic C	C	S12X	x		x	commercial product

All of these will work with an assembler, and allow assembly code to be embedded within the language. The main difference will be in whch assembler is used. There are several S12 assemblers available that can be used to write software: Code Warrior from Freescale, Cosmic Software's cas12x (part of their C programming environment), and Dirk Heisswolf's HSW12. Each of these come with their own IDE programs as well.

What follows are brief descriptions, and instructions where needed, for each of these development tools.

CodeWarrior

CodeWarrior is considered the industry standard, and is available in different suites from Freescale. Each suite has different capabilities and prices. The least expensive of these is the Evaluation version, which is available for free. Its main limitation is that it only allows C programs to compile to a maximum size of 32 KB. The assembler is not bound by this limitation though. On the other end of the suites is the Professional version, which has no limitations. CodeWarrior includes an IDE and a simulator. You can find out more by visiting the Freescale site for CodeWarrior at:

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW-HCS12X&nodeId=0152102726E4C7E4CB&tab=Buy_Parametric_Tab

Here is the link to download the Special Edition of the software. The link will download an executable to run that does the actual installation. You will also need to download and install the relevant Service Pack to support the S12GA240 MCU used on EsduinoXtreme, and the ReadMeFirst document. Please be aware that the file sizes are over 300 MB, so it is best to use the fastest network connection possible. Alternatively, you may order a CD from Freescale to obtain the software.

Here are some other links for documentation to help get CodeWarrior up and running. This is a comprehensive, and therefore complex, package. Plan to spend some time learning how to use it.

CodeWarrior Documentation List
Quick Start Instructions
Compiler Documentation
CodeWarrior Development Tools Learning Center

While CodeWarrior can be used to generate s19 files, it does not support direct interface through the on-chip serial monitor. uBug12 can perform that function by enabling you load the generated files into your board. However, if you'd like to use CodeWarrior for seamless loading and interactive debugging, you will need a BDM pod that is compatible with CodeWarrior. CodeWarrior is not guaranteed to work with all BDM pods, so be sure to see the list of compatible BDM pods in the Freescale documentation. The Technological Arts USBDMLT, which is a low-cost implementation of the open-source USBDM project, works very well and has the advantage of providing power to your board via the host USB port.

Dirk Heisswolf's HSW12

This S12X/XGATE assembler is part of Dirk Heisswolf's HSW12 IDE. This is a freeware IDE and assembler available on the web. (The site is listed in the instructions that follow.) The main difference between this development platform and others is that it is designed to be run on Linux systems. The IDE is also designed to work with a BDM pod when loading and debugging programs. However the assembler itself can easily be made to run on Windows computers, with the additional installation of a Perl interpreter, which is also freely available.

Here are the instructions to install Dirk Heisswolf's assembler on a Windows computer:

1. Get the free ActiveState Perl executive installed. To do this, first go to the URL: `http://www.activestate.com`
2. Under Community Tools, click on ActivePerl.
3. On the next page that appears, click on the ActivePerl Download Now button.
4. Choose to Save the file.
5. After it is saved, then double click on the file to run it.
6. You have to agree to the license to proceed.
7. Standard settings work fine. Just click on Next each time it shows.
8. The Perl interpreter is now installed on your computer.
9. Get Dirk's freeware S12X assembler installed. To do this, go to his web site at: http://home.arcor.de/hotwolf/

10. Click on the Download link.
11. Click on Save when prompted, and choose where to save the file.
12. Once this file is saved, you will have to uncompress the archive. However, if you used IE to get the file, the file you get is also an archive! (If you use Firefox, you don't get this extra complication.)
13. To uncompress this file, first rename it so it has a '.zip' extension.
14. Now decompress the zip archive.
15. Take the folder that is created, called 'hsw12', and move it to the root directory of the C: drive.

To run the assembler, a DOS command window will have to be used. This can be done by going on your Windows system to the Start button, and clicking on Run under that. The command to run is 'cmd'. This will bring up a DOS window to type commands in. Here is an example of the command to run to use the assembler:

```
c:\Perl\bin\Perl.exe /hsw12/perl/hsw12asm.pl /MyProgram/Fconfig.asm -s19 -L /hsw12/perl
```

Note that there are no spaces used in the pathnames or filenames. If that was the case, that component would have to be enclosed in quotes. Complete pathnames have to be specified too for it to work. Therefore the command format used is cumbersome to type. To help make this process easier, an IDE like AsmIDE can be configured to do the command for us.

When using the HSW12 assembler, you will need to keep the following points in mind:

1. The assembler likes code fields separated by the Tab character rather than by a space. This seems to be more an effect of using the ActiveState Perl interpreter than Dirk's assembler. Under certain circumstances, the assembler will flag a line as an error when the first character is a space rather than a tab, or when a space separate fields like label and instruction. I haven't spent time noting the exact circumstances of the effect. I just use tabs regardless now, and have not had a problem.
2. Arithmetic expressions may have different rules of precedence than other assemblers. You need to specify parentheses for everything to make sure expressions get evaluated correctly. Evaluation seems to be right to left, whereas other assemblers may be left to right. The assembler is made to have the following precendence order: -/~, &, |, ^, >>, <<, *, /, %, +, -
3. Comments after an instruction need to be prefixed by a semicolon, or they are considered part of the instruction. This also means that regardless of where a semicolon is in a line, it starts the comment section. Therefore the semicolon can't be part of of an instruction parameter like FCS ";CODE", where the programmer assumes the semicolon is supposed to be part of a string.
4. If you need to use Indexed-indirect Program Counter Relative addressing, you will have to use the format [TARGET] for the instruction target address. Dirk does do offsets relative to the PC register, but uses this format to implement the PCR option seen in some other processor assemblers. So to do an indirect jump to an address held in memory location TARGET, the code will be: JMP [TARGET] ;Instruction is generated with offset relative to the PC register
5. S2 records generated need to be checked carefully to make sure they are created to go into the correct place in memory. Using the ORG instruction can be tricky, and will produce bad results if not done correctly. Please read DIrk's comments on this thoroughly.
6. The BRA instruction will change automatically to an LBRA instruction if the target address gets too far away. This is generally a good thing, but it does add two extra bytes to the code. So be aware of it if you end up needing to count bytes used by a routine. You can force the assembler not to switch to LBRA by prefixing the target address with the left angle bracket character "<".
7. By default, the assembler assumes direct page addressing for addresses starting in $00xx. If you change the contents of the Direct Page register, you can let the assembler know about it with the SETDP directive. You can force direct addressing by prefixing the target address with the left angle bracket character "".

Once you have an S-record generated, you can program it into your module using the appropriate method for the hardware variant you are using (e.g. uBug12 for the Serial Monitor, or any compatible BDM pod).

AsmIDE

AsmIDE is not an assembler, but an IDE that comes with the default as12 assembler, and is capable of being adapted to using Dirk's HSW12 assembler, if desired. The install package for this IDE is available from several sites. The URLs are:

http://hcs12text.com/files/asmide340.zip
http://mamoru.tbreesama.googlepages.com/asmide340.zip
http://mamoru.tbreesama.googlepages.com/asmide-src.zip (This provides the source.)

To set this up to use the HSW12 assembler, you will first need to install the assembler and Perl interpreter as per the instructions previously provided in the HSW12 section. Next, download the AsmIDE zip file and extract the files into a directory created to contain them. You will find in your directory an executable called AsmIDE.exe.

Start the AsmIDE.exe program so that it can be configured. Once the application window appears:

- Go on the Menu bar to View > Options. This will bring up a dialog box that sets the IDE options.
- Select the Assembler tab at the top.
- You will need to change one of the default CPU settings. Under 'Currently Selected Chip family' select 6808. The recommended tab to pick is '6808 Options', as this is hardly used now.
- For 'Full pathname of Assembler', enter: c:\Perl\bin\Perl.exe
- For 'Full pathname of helpfile..', enter: C:\hsw12\doc\hsw12.html
- The last text box on the dialog is for assembler switches. This should have: hsw12asm.pl % -L /hsw12/perl/ -s19
- You will need to have your source code located where the Perl assembler is, at: C:\hsw12\perl

Now you can use the IDE to create assembler source files. They can be assembled by going to Build > Assemble on the IDE. This will generate a file of S-records that can be loaded into EsduinoXtreme.

MiniIDE

This is another IDE that comes with the asm12 assembler as its default. It is located at URL:

http://www.mgtek.com/miniide/

SBASIC

The SBASIC language is a 'no line numbers' variation of BASIC, with a compiler that runs on Windows computers. It uses the as12 assembler to generate object code,. You will need to download as12 and install it to use SBASIC. The URL for obtaining both SBASIC and the as12 code is:

http://www.seanet.com/~karllunt/sbasic.htm

Or if you prefer, here at the direct links:

SBASIC ZIP archive
as12 ZIP archive

Once this is extracted to its own directory, you will find within it the SBASIC compiler, called sbasic.exe. SBASIC is a compiler that runs via a command line. Use Start/Accessories/CommandPrompt in WIndows to open a DOS Command Prompt window. Typing the command without any parameters generates the following message, providing a summary of it's usage:

```
SBasic compiler (version 2.7) for the 68HC11/68HC12
usage: sbasic  infile  [options]
where infile is the name of an SBasic source file.
Input files have a .bas extension by default.
Output will be written to stdout.
```

```
Options are:  /cxxxx   /vxxxx   /sxxxx   /b  /mxxxx  /i
where xxxx is an address given as four hex digits.
/cxxxx   sets first address of executable code
/vxxxx   sets start of variables
/sxxxx   sets top of return stack
/b       generates branch opcodes, not jump opcodes
/mxxxx   sets target MCU (6811 or 6812)
/i       does not generate interrupt vector table
```

There are several sample programs included in the archive with SBASIC, along with a manual that explains its use in greater detail. You can pipe the output to a file that can be used by the assembler:

```
sbasic  infile    >outfile
```

You must specify options to have the program generate code that will work on the S12, as the default values will not work. Read the manual for more information on the options for variable location, stack, etc. that must be configured.

Once SBASIC generates an assembly file, you will need to use as12 to parse that file. It in turn will generate an S-record file that you can program into your target board. To start the as12 assembler, enter the following command at the prompt:

```
as12 file.ext
```

where file.ext is the path, name, and extension of the file you want to assemble, in this case the output saved from the sbasic command. The as12 assembler will assemble the file, sending the listing output to the console and writing the S19 object output to a file named m.out. You will want to then rename the m.out file to something ending in .s19 to load into the target system with uBug12 or a BDM pod.

Please note that SBASIC and as12 by themselves do not do any hardware setup. It is up to you to write the code to set up the MCU hardware.

SynCode and Eclipse GCC

The URL for these is:

http://feaser.com/zip/

Imagecraft C

Imagecraft C for CPU12 is a commercial product that supports HC(S)12 microcontrollers. It includes support for the NoICE12 debugger. You can download and use it for free during a 45 day trial period. There are special-priced versions for students and non-commercial users. The URL is:

http://www.imagecraft.com/

Cosmic C Cross Development Environment

Cosmic C is a commercial product that includes the Cosmic S12 assembler and C compiler. They also have a full featured BDM source code debugger available, and full S12 simulator. Their IDE is called IDEA. It is available in versions for both Windows and Linux based computers. The URL is:

http://www.cosmic-software.com/s12x_des.php

Software Debugging

Writing software is only the first part of the development process. Next comes testing to verify the program functionality. This requires not just running the application on the target system, but having a means of tracing what is happening should something not function as expected. There are two methods of accessing the S12 processor to do debugging. The first is by using the flash-resident Serial Monitor. The second is the use of an external BDM pod. Each method has value and which is used will depend on the needs of the development process and the available budget.

Serial Monitor

The basis for the serial monitor pre-programmed into EsduinoXtreme is described in detail in Freescale document AN2548. It provides a means to assist in debugging S12 code, among other capabilities. However the serial monitor by itself can not do anything. It is designed to work with an external program that communicates with it: uBug12.

Because of this, the serial monitor has a requirement that an SCI port must be dedicated to its use. Therefore if you use the serial monitor as part of your troubleshooting process, that SCI port cannot be used by your application program. The serial monitor uses the SCI0 port on the board, so you are free to use SCI1 for your application, if needed.

The AN2548 document is available for reference at: http://www.freescale.com/files/microcontrollers/doc/app_note/AN2548.pdf?fsrch=1

uBug12

Technological Arts provides a multi-platform Java program called uBug12 as a means of interfacing with the serial monitor in Flash. The program enables the user to manage the use of Flash memory, and it provides standard troubleshooting capabilities such as examination and modification of both CPU registers and memory locations. A separate document on the Technological Arts website describes the installation, functionality, and operation of uBug12.

BDM Pods

USBDMLT:
This pod is based on the open-source USBDM project on the Freescale Community Projects board, and is supported as a Debug tool in CodeWarrior (select TBDML). It comes with a DLL for CodeWarrior that needs to be installed first, however. Check the Resources tab of the USBDMLT product web page for details and links to the files and documentation. A small standalone Windows application called HCS12_Flash is also provided, which enables loading s-record files into virtually any HC12, S12, and S12X target.

noICE12

This is a Windows-based source-level debugger for HC12, S12, and S12X targets, with support for BDM pods, including USBDMLT.

URL is: http://www.noicedebugger.com/

Software Considerations

Software development can proceed more easily if you have a good understanding of the basics of how the 9S12G hardware works. An understanding of memory addressing and interrupts is needed as a starting point. This section provides that introductory background. However for a complete definition of the 9S12G capabilities, read the Freescale reference.

Memory Map

PPAGE and banking are two terms you need to understand relating to addressing memory on the S12.

Let's start with the CPU (Central Processing Unit). This is the processor that handles the actual machine code instructions. It is designed with 16-bit registers, such as X, Y, and PC (Program Counter). As such, it can address some 65,536 bytes directly (64K), and not one byte more nor less. It does not care what is on the other end of an address, be it a register, RAM, Flash, or a penguin dancing on a keypad. An address is just a place to read or write an 8-bit byte. Every piece of hardware that the CPU communicates with has to map somewhere into this address space. We will refer to the CPU address space as the Logical Address. (It makes logical sense to the CPU to find stuff here!)

One consideration to keep in mind regarding memory accesses is that the CPU is designed as a 16-bit processor. As such, its accesses are optimized when it is doing fetches and writes to even addresses. (An even address has a zero in the Least Significant Bit position.) If the CPU has to do a word access with an odd address, then it actually has to make two accesses, and then manipulate the results to produce the expected results. This increases latency. So be sure to remember this when setting up tables that will be read frequently. (Instruction fetches have this issue mostly alleviated by the use of an instruction queue.)

There are various memory resources that have been included on the S12GA240 chip. The Flash consists of 240K of memory. Physically, a 240K Flash memory would have its memory cells addressed from $00000 to $3DFFF. This physical address for Flash is never used by the S12 directly, though. The only time you really need to be aware of it is if you are preparing an S-record file that will be used by Freescale to create a ROM version of the S12 for OEM use.

Notice that the address space for 240K Flash requires 20 bits to specify, not 16 bits. Obviously, there is an issue here, as the CPU can't directly access this much memory. Furthermore, Flash is not the only memory-mapped device we want the CPU to access. There are also: RAM, EEPROM, and various I/O (Input/Output) devices, such as that dancing penguin.

What Freescale did to solve this problem was to divide up the CPU logical space into sections, each dedicated to its specific device. We'll get back to this in a moment.

The next step was to devise a way to determine the value of the extra address lines each piece of hardware had. To do that there had to be a way of tying all this hardware (such as Flash and RAM), together physically so that the CPU could address them. The solution was to define something call the Global address space. Global addresses start at $000000 and go to $7FFFFF and are the physical address space used to reach real hardware, like Flash or RAM. It takes 23 bits to define this address space, providing room for 8 Megabytes of stuff. With this much addressing range, that 240K block of Flash can fit in just fine. Furthermore, room is reserved for larger Flash devices as well.

The internal devices of the S12 were then placed in the Global Address space at fixed locations. Notice that Flash is at the top of this address space, and here is where things like PPAGE fit in. To be able to specify the extra address bits which the CPU can't access, registers were defined to accommodate the values for the additional address lines. PPAGE is specifically for Flash. Code can only be run in the Logical Address space of the CPU. To allow Flash, for example, to be able to run code routines stored in it, it has to be mapped into the Logical Address space of the CPU directly. This is done by dividing up the Flash into pages of 16K each. Two of these pages are then fixed into the CPU Logical Address space. They are always there. (Okay there are lots of caveats and exceptions I could be going into here, but I'm not going that far for now.) This way code can be placed in these fixed pages so the CPU has something to tell it what to do when power is first applied. This is why the interrupt vector table is in Flash by default. Space is also allocated in the Logical Address space for a third 16K Flash page. The hardware Flash page appearing here, though, is determined by the value of the PPAGE register.

For a 240K Flash, the 16K pages are numbered starting at $F1 and going to $FF. One would think that numbering them should start at $00 and go to $0F, but such is not the case. This partially has to do with where the Flash is in the Global Address space, namely at the top. As the possibility exists for larger versions of Flash to become available over time, the space taken by Flash will grow downward. With 256 pages of 16k bytes each, the maximum possible Flash that can be used by the S12 is 4 megabytes. If that full amount were available, then the page numbering would go from $00 to $FF.

The default mapping of Flash to CPU Logical Address space on powerup is as follows:
Flash Bank $FD - CPU Address range is $4000 to $7FFF (Fixed)
Flash Bank $FE - CPU Address range is $8000 to $BFFF (Window; Actual Bank determined by PPAGE value.)
Flash Bank $FF - CPU Address range is $C000 to $FFFF (Fixed always)

The PPAGE register is in the CPU Address space at $0030. Changing the value written here determines the actual 16K Flash bank that is visible to the CPU in the $8000 to $BFFF address range. This is how the processor can execute code in any Flash bank. Furthermore, there is an instruction made just for this: CALL. With CALL, the programmer can set it up so that the executing code calls a subroutine in another Flash Bank.

Various blocks can be re-mapped via special registers following a reset event, if desired. The starting address for the RAM block is set via the INITRM register, the location of the register block is re-mapped via the INITRG register, and the location of the EEPROM block can be re-mapped via the INITEE register..

Interrupts

Embedded systems are called such because they interface directly with the real world. The interfaces used rarely are the video displays and keyboards a standard desktop computer has. More likely they are things like temperature sensors, motor drivers, and so forth. A common consequence of this is that the programs written for an embedded system have to be able to respond quickly to external events occurring asynchronously to the program execution state. The best way to process these events and have them work with your program is to set up interrupts.

An interrupt is a way to make an asynchronous event synchronous with your program. Basically, what happens is that some event external to the CPU signals the CPU to stop doing whatever it is doing, and to spend some short amount of time dealing with the external signal. After the CPU finishes what it needs to do, it can go back to whatever it was doing before the interrupt happened.

The typical way of doing interrupts would be to have a software routine, usually in assembly language, set up to be called at a specific address. (The address could be fixed or specified in a table.) An external pin on the processor would be pulled low by the device requiring handling. The only other fact to remember is that there is a flag set aside in the condition code register to mask the interrupt, or allow it to be processed. Usually the interrupt would only be masked when it was absolutely necessary in the main program. An example is when the program reads a variable that is itself changed by the interrupt routine. The interrupt must be masked while the read takes place and then enabled again after the read is completed.

The S12 can process an external interrupt this way. However, since the processor comes with several additional hardware interfaces built into it, it makes sense that these interfaces are also set up to be able to trigger their own interrupts. The result is that the system has now gone from one available interrupt to a large number of possible interrupts. Therefore some way has to be set up so that if two interrupts happen at the same time, one of the two can be specified as being at a higher priority and thus will be handled first. Also, not all the available hardware possibilities will be used in any one design. So the unused interrupts must not be enabled.

There is one more complication added when using the serial monitor, which appears at the top of flash memory. Instead of relying on one interrupt routine to determine what device requested an interrupt, each S12 interrupt source automatically causes the associated handling routine address to be fetched from a table, called the Interrupt Vector Table. For the S12, this table is by default at the very top of memory, right in that same 2K memory space that the serial monitor is in. Since that memory is protected, it can't be changed by the serial monitor. (Okay, a BDM pod can do it but that defeats the purpose of working with the serial monitor.) There are solutions to

these issues, and I'll cover them here.

As an example, I'll use SCI0 as the interface we want to set up to handle an interrupt from.

The steps we need to do are:
0) Set the stack pointer.
1) Initialize any pointers, buffers, etc, needed by the interrupt routine.
2) Initialize the S12 to handle interrupt input.
3) Initialize the hardware to enable its interrupt output.
4) Enable CPU interrupt flag in Condition Code register.

Stack Pointer

Before any interrupts can be handled, the CPU stack pointer has to be initialized. This is done with a LDS instruction, and need only be done once in your setup code. Be sure you set it up to point to a valid location in RAM, and that you reserve plenty of space for it to grow into. (Stacks grow down towards lower memory addresses.) Oh, yeah-- not using a valid RAM location will crash your system.

Usually the stack pointer is used to store the return address when calling a subroutine, or to store temporary variables. However in the case of an interrupt, all the CPU register values are stored on the stack when the interrupt begins processing. That way when the interrupt processing routine finishes, the entire CPU state is restored. Whatever routine got interrupted should never know it.

Therefore when writing an interrupt processing routine, it has to end with an 'RTI', or Return from Interrupt instruction. RTI expects to find all the CPU registers stored on the stack in a certain order, so it can restore the CPU state as required when it is executed. If your interrupt routine does not leave the stack exactly as it found it, you will crash the system. (Been there, done that!)

Interrupt Routine Initialization

Your setup code needs to make sure that before any interrupt handling is enabled, that any variables that the interrupt routine accesses are initialized correctly. This includes counters, buffer pointers, buffer contents, etc. This may seem obvious, but a lot of software problems come from not noticing or taking care of the obvious stuff.

S12 Interrupt Initialization

Now we start getting to the fun stuff! The first item to set up is where the table of interrupt vectors is in memory, and to do that we need to understand what exactly is an interrupt vector table.

In the typical processor discussed previously, there is only one interrupt address to deal with-- for the one interrupt pin on the processor chip. The S12 could have stayed with this, and left it to the interrupt routine to determine which hardware device triggered it. However, this approach ends up being wasteful of CPU cycles, and that can really hurt a system designed for high performance embedded use. The alternative is to have each integrated hardware device provide the CPU directly with the address of the routine to handle its interrupt needs. This is what the Interrupt Vector Table is for.

When an S12 device triggers an interrupt, that request is handled by the Interrupt module. The Vector table the module manages occupies a 256 byte block of memory. Since each routine address is specified by two bytes of memory, the table can hold up to 128 interrupt routine addresses. You will need to reference the S12 documentation to find out where each device's interrupt address is kept in the table.

The serial monitor implements a pseudo-table for interrupts, set aside at the $F7 page in memory. The reason for the pseudo-table is that the serial monitor has to be able to manage its interrupts to function correctly. The pseudo-table will also allow detection of an interrupt triggered that no routine is written for it, and let the serial monitor take over. This helps in the troubleshooting process. The pseudo-table exists from $F710 through $F7FF. Your 'reset' vector will need to be programmed at $F7FE, or your application will not start on powerup, even if the **Load/Run** switch is in the **Run** position.

You might ask yourself, "What happens if two devices simultaneously ask for an interrupt to be handled?" There are two parts to the answer to this question, and I'll handle the first part here. The position of an interrupt address in the Vector table gives it a priority in relation to all the other interrupts. So if two interrupts come in simultaneously, the one that has the vector address higher in the table will be handled first. That is why Reset, which is considered an Interrupt, has its vector at the top of the table at $FFFE. It trumps all other interrupts.

For our example, the SCI0 device is supposed to have it's interrupt handler address stored at the $D6 location in the table. The SCI0 interrupt handler's address should be in Flash at $FFD6 and $FFD7. (Relocated to $F7D6/7 with the serial monitor in use.) If two interrupts have the same priority, then table position will determine the higher priority one.

Hardware Interrupt Initialization

The next step is to set up the hardware interface so that it generates an interrupt when we want it to. By default, hardware generally will not send an interrupt to the CPU, and so we need to configure it to do this. Obviously this step is very dependent on the particular hardware we want to generate the interrupt. Therefore you will need to read closely the documentation for the hardware subsystem you are using to make sure that it is done correctly. Otherwise strange results will occur, if any results occur at all.

For SCI0, we need to set the baud rate first. This is done at $00C8 and $00C9. You will need to read the documentation to determine the values to write here, to get the baud speed you want based on the CPU bus clock. The default value for Control Register 1 is good, so we keep this by setting SCI0CR1 at $00CA to $00.

Finally, we enable the transmitter and receiver for SCI0, and enable their respective interrupts at the same time by writing a value of $AC to the SCI0CR2 control register at $00CB.

CPU Interrupt Enable

The final step is to enable the S12 CPU to handle interrupts. Following Reset, the Codition Code Register has the I (Interrupt) mask bit set. This bit must be cleared before the S12 will start handling interrupts. This can be done with the 'cli' assembler directive, which translates to 'andcc #$EF'. The bit is set whenever an interrupt routine is called by the Interrupt module.

It is possible for you to write your code to clear this flag on purpose while your interrupt routine is being processed. Doing so will allow interrupts of a higher priority to interrupt the current routine, while still preventing lower priority routines from stepping in. It will also keep high priority interrupts processed in a timely fashion.

As you can see, setting up interrupts is not for the faint of heart, and requires careful planning for it to be done correctly. Furthermore, debugging interrupt code can be a nightmare. (The IDE simulator with its breakpoints can help here, but it only goes so far.) For difficult problems this can require the use of a BDM pod with matching troubleshooting software, and even a logic analyzer. However don't let these difficulties keep you from using interrupts. You can still do a lot, even without a BDM pod. It just requires using that space between your ears more.

Other Interrupt Caveats

Some of the interrupts, like SWI and XIRQ, have slightly different rules for using them. However the basics outlined here still apply. Again, you will need to read up on the documentation to use these effectively.

S12 Clock

Since your application, most likely, will be designed to run upon power up of the module without the support of the serial monitor, it will need to configure the clock registers

on the 9S12G hardware. The reason for this is that if the **Load**/**Run** switch is in the **Run** position, the serial monitor detects this setting upon powerup or reset and jumps immediately to your application as pointed to by the address you put in Flash at $F7FE/F. (If you do not program this location, the serial monitor will take control regardless of the **Load**/**Run** switch position!)

When the serial monitor starts your application following reset, it does not configure any of the hardware registers first. It goes directly to your code.

After reset, MCU clock is in PEI mode (refer to 10.1.2 in S12G Manual), so Bus Clock (MCLK) is 6.25 MHz. This means the oscillator based upon the external crystal is not enabled, and the bus clock is generated from the internal 1MHz oscillator, boosted by the PLL to 25MHz and divided by four, resulting in an MCLK frequency of 6.25 MHz.

How is the clock set then to the speed you want? To do that, the hardware uses the following equations:

PLLClock = 2 * Oscillator * (SYNR + 1) / (REFDV + 1)
BusClock = PLLCock / 2

The BusClock frequency is the one we want. The Oscillator term is the frequency of the crystal oscillator attached to the chip, or the frequency being fed into the MCU on the EXTAL pin. EsduinoXtreme has an 8 MHz crystal, so this will be the oscillator value. The registers SYNR and REFDV will need to be initialized to get the system clock at the desired value.

Example taken from the Serial Monitor code (S12SerMon2r7):

```
SYNR: equ CRGV4+$00 ;CRG synthesizer register
REFDV: equ CRGV4+$01 ;CRG reference divider register
POSTDIV: equ CRGV4+$02 ;CRG post divider
CRGFLG: equ CRGV4+$03 ;CRG flags register
LOCK: equ %00001000 ;lock status bit
UPOSC: equ %00000001 ;Oscillator Status Bit
CLKSEL:    equ    CRGV4+$05    ;CRG clock select register
PLLSEL:    equ    %10000000    ;PLL select bit
PLLCTL:    equ    CRGV4+$06    ;CRG PLL control register
PLLON:    equ    %01000000    ;phase lock loop on bit
CPMUOSC:    equ    $02FA    ;S12CPMU Oscillator Register
OscFreq:    equ    8000    ;Osc speed (i.e. external crystal frequency)
initSYNR:    equ    $02    ; mult by synr + 1 = 3 (24MHz)
VCOFRQ:        equ    $40    ; vco gain
initREFDV:    equ    $00
REFFRQ:        equ    $80    ;pll stability
initPOSTDIV:        equ    $00    ;Post Divider Register (CPMUPOSTDIV)

    movb    #initSYNR+VCOFRQ,SYNR    ;set PLL multiplier
    movb    #initREFDV+REFFRQ,REFDV    ;set PLL divider
```

```
    movb    #initPOSTDIV,POSTDIV    ;set post-divider
    movb  #$C0,CPMUOSC  ;Enable external oscillator
    nop
    nop
    brclr    CRGFLG,LOCK,*+0    ;while (!(crg.crgflg.bit.lock==1))
    bset    CLKSEL,PLLSEL    ;engage PLL to system
```

Clock Usage

The oscillator and bus clocks are used not just to sequence through instructions and time memory and peripheral accesses. They are also used to dictate the timing of several other harware module capabilities, such as SCI baud rate, RTI/COP timing, and Flash/EEPROM programming.

For example, the baud rate for any of the SCI interfaces is set by this equation:

baudRegister = ((BusFreq / 16) * 10) / baudrate

Here the BusFreq is the system bus frequency as determined above, in KHz. This will usually be 24000 KHz. The baudrate value is the desired baud divided by 100. So 9600 becomes 96, etc. The resulting baudRegister value is then programmed into the SCIBDH and SCIBDL registers as the high and low bytes of the value, respectively.

Obviously, if the system bus clock is changed to a different value, then this will affect the baud rate programmed into the SCI module.

Example coming soon...

Hardware Details

Input/Output Connectors

The four main I/O connectors follow the Arduino Leonardo standard for pincount and location, and attempt to match up signals from the MCU to similar signal types used on Arduino Leonardo. Additionally, a 6-pin header bringing out the SPI pins is implemented in the same physical location of the board, to provide compatibility with shields that depend on this feature *(Note: due to a circuit board layout error, the pinout of this connector on Rev. 1 EsduinoXtreme is incompatible with Arduino shields; this will be corrected on the Rev. 2 board)*. Unused pins on this header have been assigned to VDD so as to make the header more versatile.

Since the S12GA240 MCU has a higher pincount than the Atmel MCU used on Arduino Leonardo, there are some extra choices that have been implemented with jumper blocks so that the user can decide assignments on an application-by-application basis.

I2C implementation: since the S12GA240 does not have a dedicated I2C subsystem, I2C functions must be bit-banged in software via the assigned pins connected to J3 pins 9 and 10 (or via any other convenient pins).

The EsduinoXtreme feature sheet details the signals for each header pin, using the pin names shown in Freescale documentation. Note that several pins can have more than one function, depending on how the hardware registers are configured. Furthermore, some hardware subsystems can be re-mapped to different ports. Refer to the Freescale manual for the MC9S12G for details.

Voltage Regulator Configuration

EsduinoXtreme has two on-board regulators: U2 supplies 5V and U3 supplies 3.3V. U2 takes the filtered DC voltage (6-12 VDC) applied to one of the optional VIN connectors (J6 and J7) or applied to the VIN pin on header J4 and provides a smooth regulated 5 Volts DC to the components. When jumper block JB1 is set to VIN, this 5V rail is active. Otherwise, 5V comes from the USB connector, and U2 is not used. In either case, the 5V rail is fed to 3-Volt regulator U3, which produces a 3.3V output for use when the MCU operating voltage is set via JB7 to 3V, and which also appears at the 3V3 pin on header J4, for use by application shields, if required. The 3V supply is also used to power the optional XBee module if it is attached via user-installed header J10, since it is a 3V device.

If you plan to use the on board regulators to power your own circuits in addtion to the module, please be aware that the regulators are rated to supply a maximum of 700 mA total current each, but the total current draw will be limited if the system is powered via the USB port instead of an external power supply. Another limiting factor is heat. At room temperature, the regulator by itself will only be able to dissipate about two watts of heat. The amount of dissipation needed will depend both on the input voltage applied, and the current drawn to feed the electronics. So a power supply at 6 volts will not cause the regulator to heat up as much as a 12 volt supply will. If you need to dissipate more heat than two watts though, you must add a heat sink to the regulator.

Appendix

Electrical specifications

Input Voltage: Optional connectors J6 or J7 will accept a positive voltage from 7-12 Volts DC to provide VIN when a USB connection or USBDMLT pod connection to a host computer is not used.
Operating Current: 50 mA nominal. The on-board regulators can provide up to 700 mA, so external circuitry connected to the headers can safely be powered from the regulator. The total current that may be drawn before the voltage regulator's thermal shutdown circuit engages depends on the input voltage and ambient temperature.
Operating Temperature Range: -20 C to +70 C (Note: the temperature range is limited mainly by the rating of the crystal. A broader range of -40 C to +85 C can be accommodated by using an extended-temperature range crystal or the internal 1 MHz oscillator.
Board Dimensions (nominal): 2.70"(68.7mm) x 2.1"(53.4mm) x 0.475"(12mm), excluding connectors

Notes on Serial Monitor

EsduinoXtreme comes pre-programmed with a serial monitor in the topmost 2K of Flash memory. This monitor is based on Freescale's Application Note AN2548, adapted for 9S12GA240 by Technological Arts. It is designed to function using the SCI0 interface at 115200 baud with an 8 MHz crystal, which it boosts to 24MHz using the PLL. Besides activating the PLL, it moves the RAM block and EEPROM block, via the INITRM and INITEE registers, to various addresses, depending on the MCU variant. To learn more about these details, or to customize options (such as which SCI to use, which port pin for the Load/Run selection, bus frequency, etc.), download and open the the serial monitor project in CodeWarrior, where you can view and modify the .def and .asm files, as required. After building the modified project, the new s-record file will have to be burned into flash using a BDM pod (e.g. USBDMLT from Technological Arts). You'll find the the serial monitor project in the Technological Arts Support Library, at:

http://support.technologicalarts.ca/docs/Esduino/EsduinoXtreme/Code/SerialMonitor/

Last Updated (Monday, 05 December 2016 10:57)