Using Your Adapt9S12D Microcontroller Module

| Print |

Introduction

The flagship microcontroller of the Freescale S12 family is the 9S12D. It is a full-featured microcontroller, with Flash, EEPROM, and RAM memories, SCI, SPI, CAN, and IIC communications subsystems, and multi-channel analog-to-digital conversion capability. Adapt9S12D is a modular implementation that brings these features within easy reach of engineers, educators, and hobbyists. The flexible design of the entire "Adapt" series of microcontroller products supports all aspects of training, evaluation, development, prototyping, and even volume production.

This manual provides basic setup and operation instructions for the Technological Arts Adapt9S12D Microcontroller Module. Included are both the hardware and software information needed to get the board working with a variety of programming languages and development tools. Details of each of the module variations are covered.

Overview of features

Adapt9S12D modules come equipped with a variety of interfaces and on-chip resources. These include, but are not limited to, the following:
- A 16 MHz crystal is standard, providing an 8MHz nominal bus speed. This can be mutliplied by 3, using the on-chip PLL, resulting in a 24 MHz bus speed.
- Memory resources, depending on the variant purchased:512K/256K/128K Flash, 4K/2K EEPROM, 32K/20K/8K RAM
- Operating voltage: either direct 5 VDC at 60 mA, or 6-12 VDC at 100 mA
- Communications interfaces: choice of two RS232C, two CAN, and one RS485, depending on variant (M0, M1, or M2)
- User LED: red LED on Port P bit 7 available for application use.
- On-chip development support: standard Freescale D-Bug12 Monitor or Freescale AN2548 Serial Monitor or StickOS BASIC, depending on variant purchased.
- BDM connector: this 10 pin standard header connector allows the use of a separate BDM pod for debugging programs in real-time, and for programming all Flash and EEPROM memories. Supports 6-pin and 10-pin BDM pods.
- Expansion connectors: these two 50-pin standard connectors enable additional external hardware to be connected to the board, and optional Expanded Mode operation which supports external memory.

Variants Offered

Various memory sizes and memory-resident development tools are offered, as follows:

Adapt9S12DP256 with D-Bug12. D-Bug12 uses any ASCII terminal program as its user interface, and is not directly supported by CodeWarrior.
Adapt9S12DP512 with D-Bug12. D-Bug12 uses any ASCII terminal program as its user interface, and is not directly supported by CodeWarrior.

Adapt9S12DG128**S** with Serial Monitor. The Serial Monitor is supported by CodeWarrior for loading and source-level debugging.
Adapt9S12DP512**S** with Serial Monitor. The Serial Monitor is supported by CodeWarrior for loading and source-level debugging.

Adapt9S12DP512**B** with StickOS BASIC. StickOS BASIC uses any ASCII terminal program as its user interface and development environment.

In addition to the memory-resident tools mentioned above, modules may be purchased with various communications transceiver hardware configurations. The three options are denoted with suffixes M0, M1, and M2 in the order code, and have the following configurations:

M0: only the RS232 transceiver circuit is populated, supporting two RS232 ports (one on SCI0 and one on SCI1)
M1: the RS232 transceiver circuit and the CAN transceiver circuits are populated, providing two RS232 ports (via SCI0 and SCI1) and two CAN ports (via Port M)
M2: all transceivers are populated. One RS232 port via SCI0; SCI1 can be jumpered to provide RS232 or RS485; two CAN ports (via Port M).

Order Codes for Standard Module Configurations:
AD9S12DG128SM0-_-_
AD9S12DP256M0, AD9S12DP256M1, AD9S12DP256M2
AD9S12DP512M0, AD9S12DP512M1, AD9S12DP512M2
AD9S12DP512BM0
AD9S12DP512SM0, AD9S12DP512SM1, AD9S12DP512SM2

Order Code Format:
AD9S12DxxxxMx- _ - _ where _ represents a connector style option code for each of the two 50-pin I/O headers (H1 and H2). Valid option codes are shown here:
http://www.technologicalarts.ca/shop/connectoroptions.html

Order Codes for Standard Bundles:
AD9S12DP256EVP (M0 configuration; D-Bug12 in flash; prototyping cards for H1 and H2)
AD9S12DP256SEP (M0 configuration; Serial Monitor in flash; solderless prototyping cards for H1 and H2)
AD9S12DP512SEVP (M2 configuration, Serial Monitor in flash; prototyping cards for H1 and H2)
AD9S12DP512SSEP (M2 configuraion; Serial Monitor in flash; solderless prototyping cards for H1 and H2)
AD9S12DP512SSEP-UR (M2 configuration; Serial Monitor in flash; USB232 retrofit; solderless prototyping cards for H1 and H2)
AD9S12DP512BSEP-UR (M2 configuration; StickOS BASIC in flash; USB232 retrofit; solderless prototyping cards for H1 and H2)

Getting Started

The first step in preparing to use the module is to read through this manual, before attempting to apply power to the device. Next, verify that the power cable is present. It plugs into a matching two-pin connector on the board. There should also be a schematic of the board, and a pinout guide for the two 50 pin expansion headers on the board.

***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.*

Setup and Verification of Hardware

Power and serial connections, switch positions, default jumper positions

If you have purchased a bundle (i.e. an Evaluation Package or Solderless Experimentor Package), a suitable power supply is included. Otherwise, the power cable assembly #PCJ1-8, supplied with the board, can be used to connect to your external power source. The tinned ends of the cable assembly can be connected to a bench power supply, spliced onto the cut-off end of a power supply, connected to a battery pack, or used in some other arrangement which you find suitable for your application. The board requires a DC power source capable of delivering between 6 and 12 volts at a minimum of 100 mA. Most standard "wall wart" power supplies should be capable of meeting this requirement. If you need to purchase one, Technological Arts offers a few suitable choices:

1. #DC9V, a 9V 300mA North American style wall adapter, which has a 2-pin Molex connector, ready to attach to the board's power connector
2. #DC12VRS, a 12V North American style switchmode wall adapter, with 2.1mm center-positive barrel plug
3. #DC9VSI-INT DC, a 9V international Switching Power Supply with a set of interchangeable plugs to accommodate all regional power outlets.

Additional power cables are also available if needed. Item #BJ2PCJ1 includes a barrel jack to connect to items 2 and 3, listed above. Once the power cable has been attached to your power supply, use a voltmeter to verify that the red wire is in the +6 to +12 DC volts range in relation to the black (ground) wire. Only then should the cable be plugged into the matching two-pin power connector on the board. (Refer to the diagram below.) When power is first applied, the green LED on the board should light up. This provides verification that the module is receiving power. Unplug power before proceeding.

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

Jumper	Function	Default Setting
JB1	ModA select	0
JB1	ModB select	0

These should generally be left at their default settings, unless there is an application need to change it. The ModA and ModB settings shown select Single Chip mode. For Expanded Mode settings, refer to the Freescale S12D Device Guide.

The 9-pin D-sub connector on the board is a standard RS232 serial port implemented on SCI0 on the S12 chip. SCI1 is also implemented as an RS232 port, next to the 9-pin D-sub, as a four pin header. The board includes an RS232 level translator for both of these, so the output can attach directly to a serial port on your computer. Technological Arts provides a cable (Item #SCPC9) for SCI1 that plugs into the header and provides a standard 9-pin D-sub connector, just like SCI0. The pinouts for these two serial connectors are as follows:

SCI 0	Description	SCI 1	Description
Pin 2	Transmitter Output	Pin 1	Receiver Input
Pin 3	Receiver Input	Pin 2	Ground
Pin 5	Ground	Pin 3	Transmitter Output

Pin 4	No Connection

You will need a serial cable for the 9-pin SCI0 connector if you plan to use the pre-installed monitor to program the board and troubleshoot your applications, or if you are using the StickOS BASIC version of the board. If you prefer to use your host PC's USB port, a Technological Arts #USB232 retrofit module may be used instead, plugging on to the 9-pin D-sub connector. This module can optionally provide 5VDC to power the Adapt9S12D module from the host USB port, plugging into the J1 power connector via flying leads provided. If this configuration is used, an external power supply isn't necessary.

Using the Factory-Installed Demo Program

The **S** versions of Adapt9S12D modules (i.e. those with the Serial Monitor installed) come pre-programmed with a Demo program in Flash. You can run this with the **Load**/**Run** switch in the **Run** position when applying power or pressing reset. When the program starts, it will blink the red LED on Port P7 three times, to show that it is running. To use the program, though, you will need to connect a serial cable from the SCI0 serial connector to your computer. 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. The terminal program will need to be set up 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 Demo program starts, it prints out a menu somewhat like this:

```
Adapt9S12DP512 DEMO PROGRAM COMMAND MENU V1.00

A => SHOW PORT A STATUS
B => SHOW PORT B STATUS
C => CLEAR PORT P OUTPUTS
E => SHOW PORT E STATUS
F => FLASH LED (CONNECT TO PP7)
H => SHOW PORT H STATUS
J => SHOW PORT J STATUS
K => SHOW PORT K STATUS
M => SHOW PORT M STATUS
P => SHOW PORT P STATUS
S => SHOW PORT S STATUS
T => SHOW PORT T STATUS
R => SHOW 10 BIT REAL-TIME ANALOG VALUES
0 TO 7 => TOGGLE THE SELECTED P PORT LINE
?
```

Typing each of the available letter commands in sequence from A through R will generate output similar to the following:

```
PORTA=255
PORTB=000
PORT P CLEARED
PORTE=159
>>> LED FLASH! <<<
PTH=000
PTJ=247
PORTK=255
PTM=015
PTP=000
PTS=255
PTT=000
AD0=0384 AD1=0390 AD2=0391 AD3=0385 AD4=0384 AD5=0384 AD6=0384 AD7=0387
```

The analog output updates the same line continuously on the screen. You can terminate the update with the Enter key, which will reprint the menu. The number keys 0 through 7 will toggle the logic level of the corresponding pin on Port P. You can see this by typing 7, and watching the red LED on the board change state.

Connecting to D-Bug12

The default versions of Adapt9S12D modules come pre-programmed with the Freescale D-Bug12 Monitor program in Flash. D-Bug12 has four operating modes, which are set by jumper options on PAD00 and PAD01. These pins are read by the startup code in D-Bug12 whenever a reset event occurs. The operating modes and their settings are covered in the Freescale D-Bug12 Reference Guide.

To use D-Bug12, you will need to connect a serial cable from the SCI0 serial connector to your computer. D-Bug12 requires use of an ASCII terminal program, such as HyperTerminal on Windows systems, to provide a means for you to interact with the program. The terminal program will need to be set up with the following properties:

- Emulate ANSI terminal type
- 9600 baud
- 8 bits
- One stop bit
- No parity
- Software flow control (i.e. XON/XOFF)
- Do not append line feeds to incoming line ends
- No local echoing of typed characters

Refer to the D-Bug12 Reference Guide for all further information related to D-Bug12.

Application Programming

There are several language options available for writing application programs for the Adapt9S12D module. One can use Assembler, C, or BASIC, or a combination of these. This section will list descriptions and hardware specific setup instructions for the various systems 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.

For default configurations, D-Bug12 will be used to load the user program into memory. For the Serial Monitor configuration, the HiWave debugger supplied with CodeWarrior can be used to program the generated S-records into the S12 Flash memory. If CodeWarrior is not being used, a free Windows utility from Technological Arts, called uBug12, can be used for erasing and programming flash, and is also useful for simple debugging when a BDM pod is not available. You can find instructions on how to download and use uBug12 at this URL:

http://support.technologicalarts.ca/docs/uBug12/

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 the Adapt9S12D:

	Language	**Assembler**	**IDE**	**Terminal Program**	**BDM Com**	**Other**
CodeWarrior Special	C	S12X	x		x	32K limit for C unlimited assembler
CodeWarrior Pro	C	S12X	x		x	commercial product
HSW12	Assembler	S12X	x		x	Linux
AsmIDE	Assembler	S12X	x	x		
MiniIDE	Assembler	asm12	x	x		
SBASIC	BASIC	as12				DOS-based
GCC-Syncode	C	as12				
GCC-Eclipse	C	as12				
Imagecraft C	C	as12				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 what 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 Special suite, 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&fsrch=1`

You will find a link on this page to download the Special suite of the software. The link will download an executable to run that does the actual installation. Please be aware that the file size is 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

If you plan on using CodeWarrior to do interactive debugging with your hardware, you will need a BDM pod that is compatible with CodeWarrior if you don't have a Serial Monitor version of Adapt9S12D. It 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 low-cost #USBDMLT, which is an implementation of the open-source TBDML project works very well, and has the advantage of providing power to your target system 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 to follow.) The main difference with this development platform from 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://hotwolf.github.io/HSW12/hsw12asm.html
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 version, D-Bug12 Serial Bootloader for the standard version), or any compatible BDM pod. Note that when using D-Bug12, S-record files must be in a specific format. The D-Bug12 distributable includes a utility called SRECCVT for re-formatting the records. See the D-Bug12 Reference Guide for help with this.

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 an S1 S-record that can be loaded into the Adapt9S12D module.

MiniIDE

This is another IDE that comes with the asm12 assembler as its default. While it is designed to communicate with the target board via a serial port, it expects the target to be running DBUG12, not the serial monitor. 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 command line run compiler. 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 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 (if using the Serial Monitor version of the module).

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 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. The URL is at:

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 at:

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

StickOS BASIC

StickOS was originally written for a ColdFire 32-bit microcontroller, and later ported by its author to the 9S12DP512. It is an interactive development environment which enables you to write and save BASIC programs directly on the module. This capability does not exist with any of the other platforms mentioned. (They all require the program to be burned into Flash before you can test it on the module.) Because of this, the Flash will not be stressed nearly as much with erase-and-burn cycles as code is developed. The final result can be burned into Flash/EEPROM as an application that runs upon power up of the board. StickOS comes pre-programmed into flash on the B version of Adapt9S12DP512. If you have a BDM pod, you can erase the flash memory on your module and load StickOS into it.

Detailed info can be found at: http://www.soBASICsoEasy.com

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 Monitor (D-Bug12 or Serial Monitor, depending on the variant). 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 serial monitor preprogrammed into Adapt9S12DP512**S** and Adapt9S12DG128**S** variants 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. This can be uBug12, CodeWarrior, or noICE12

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 can not 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 at: http://www.freescale.com/files/microcontrollers/doc/app_note/AN2548.pdf?fsrch=1

uBug12

Technological Arts provides a Windows based 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, as well as provide 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

MicroBDM12LX5:
This pod is based on a 9S12DG128 MCU running Freescale's D-Bug12 Monitor in Pod Mode. Refer to the D-Bug12 Reference Guide for details on what its capabilities and limitations are, and how to use it. D-Bug12 is not supported as a debug interface in CodeWarrior. It can be used as a standalone debug interface with any terminal program. It is also supported by the NoICE12 debugger (see below).

USBDMLT:
This pod is based on the open-source OSBDM 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 Serial Monitor, D-Bug12-based pods, and the 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 MC9S12D 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 MC9S12D 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 the various memory resources that have been included on the S12D chip. The Flash consists of 128K, 256K, or 512K of memory (in DG128, DP256, and DP512 versions, respectively). Physically, a 512K Flash memory would have its memory cells addressed from $00000 to $3FFFF. 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 512K 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 512K 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 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 512K Flash, the 16K pages are numbered starting at $E0 and going to $FF. One would think that numbering them should start at $00 and go to $1F, 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. (With external addressing, a designer could add their own additional memory in this space externally, and then use the PPAGE register to access it.)

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; Actually 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, 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 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 device 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 SCICR1 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 SCICR2 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 MC9S12D 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.

The default settings of the clock registers are such that when your application first starts up, it is running at a speed of 8 MHz, which corresponds to the module oscillator frequency (16 MHz) divided by two. Most likely you will want to change this speed. The most common value used is 24 MHz, which is the maximum the MC9S12D will support.

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. Adapt9S12D uses a 16 MHz crystal, so this will be the oscillator value. The registers SYNR and REFDV are located in the register space at $34 and $35 respectively. The program will need to update these with the values needed to get the system clock at the desired value.

Let's take an example of a target of 24 MHz, and an oscillator frequency of 16 MHz. To get from 16 to 24, we can divide 16 by 2 (REFDV + 1) to get 8, and multiply that by 3 (SYNR + 1) to get 24. Since the SYNR and REFDV are one less than our scaling factors, we get SYNR equal to 2, and REFDV equal to 1. This makes our equation:

BusClock = (16 MHz) * (2 + 1) / (1 + 1) = 16 * 3 / 2 = 24

However one does not just write to these registers with a couple of write statements in the code. The phase lock loop (PLL) system has to first be disengaged from the system, turned on, updated, and checked for stability. Afterwards, the final step would be to engage the PLL to the rest of the MCU system.

So the programming steps become:

1. Disengage PLL from System: Clear bit 7 (MSB) in register CLKSEL ($39).
2. Turn on PLL: Set bit 6 of PLLON ($3A) register.
3. Write SYNR value into register at $34.
4. Write REFDV value into register at $35.
5. Wait at least two bus cycles before starting to check for stability. This can be done with a couple of NOP instructions in assembly.
6. Check to see that the PLL is stable: Check bit 3 of CRGFLG ($37) until it reads it is set.
7. Enable PLL to become BusClock: Set bit 7 of CLKSEL ($39).

Once this is done, the MCU bus will be running at the programmed rate as set above.

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 determine 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.

Setting the RTI or COP interval timers is similar. However these use the oscillator frequency rather than the PLL frequency. There are tables in the Freescale chip documentation listing what the divide values are. The oscillator frequency gets divided by the selected divide value to give the final RTI or COP frequency. For example, one can set the RTICTL register at location $3B to a value of $F7. This selects a decimal divider of 1.6 x 10^6. Since the oscillator clock is 16 MHz, this results in a 10 Hz RTI interrupt rate, or one interrupt every 100 milliseconds.

Hardware Details

This section of the manual lists various details of the Adapt9S12D module, such as jumper settings, connector pinouts, etc. Since the pinouts for SCI0 and SCI1 have already been provided, they will not be repeated here.

Jumper Settings

The following table lists all the jumpers on the board, their function, and their default setting.

Jumper	Function	Default Setting
W1	CAN1 termination resistor	Jumper On
W2	CAN0 termination resistor	Jumper On
W3	PS2 controls R pin on RS485	Wire In
W4	PS3 controls D pin on RS485	Wire In
W5	TxD0 of S12 to RS232 (U4)	Wire In
W6	RxD0 of S12 from RS232 (U4)	Wire In
W7	Connect RS232 (J4) pin 4 to 6,1	Open
W8	Connect RS232 (J4) pin 8 to 7	Open
W9	Connect RS232 (J4) pin 6 to 1,4	Open
W10	Connect RS232 (J4) pin 1 to 6,4	Open
W11	RS485 termination resistor	Jumper On
W12	Power to RS232 level translator	PCB Trace
W13	Power from regulator	Wire In
W14	Ground is VRL	Wire In
W15	VCC is source of VRH	Wire In
W16	RxCAN0 to RxD (U6)	PCB Trace
W17	TxCAN0 to TxD (U6)	PCB Trace
W18	RxCAN1 to RxD (U7)	PCB Trace
W19	TxCAN1 TxD (U7)	PCB Trace
W20	RxD1 of S12 from RS232 (U4)	Wire In
W21	TxD1 of S12 to RS232 (U4)	Wire In
W22	not implemented	
W23	not implemented	
W24	not implemented	
W25	not implemented	
JB1	ModA select	Jumper On 0
	ModB select	Jumper On 0
	00=Single Chip mode	
JB2	XCLK to Ground	Open
JB3	Dbug12 mode select; PAD0/1	see D-Bug12 docs
JB4	XIRQ to Ground	Jumper Off
SW1	Reset	
SW2	Load/Run to PA6	RUN = open
D1	LED to PP7	remove R11 to disable

Input/Output Connectors

The two I/O connectors each have 50 pins. Note that the pin-numbering sequence is sequential (i.e. not alternating, like a ribbon cable connector). The following table lists the signals for each pin. Several pins can have more than one function, depending on how the hardware registers are configured. Furthermore, some interfaces can be moved to different ports. Refer to the Freescale manual for the MC9S12D for details.

H1 Pin	Signal Name	H1 Pin	Signal Name
1	PS4/MISO0	50	GROUND
2	PS5/MOSI0	49	GROUND
3	PS6/SCK0	48	PS0/RXD0
4	PS7/SS0*	47	VCC
5	PS1/TXD0	46	PE1/IRQ*
6	PT7	45	PE0/XIRQ*
7	PT6	44	RESET*
8	PT5	43	PE7/XCLK*
9	PT4	42	PH0/KWH0
10	PT3	41	PH1/KWH1
11	PT2	40	PH2/KWH2
12	PT1	39	PH3/KWH3
13	PT0	38	PH4/KWH4
14	PP7/PWM7/SCK2	37	PH5/KWH5
15	PP6/PWM6/SS2*	36	PH6/KWH6
16	PP5/PWM5/MOSI2	35	PH7/KWH7
17	PP4/PWM4/MISO2	34	PS2/RXD1
18	PP3/PWM3/SS1*	33	PE4/ECLK
19	PP2/PWM2/SCK1	32	PS3/TXD1
20	PP1/PWM1/MOSI1	31	VRL
21	PP0/PWM0/MISO1	30	VRH
22	PAD00/AN00	29	PAD04/AN04
23	PAD01/AN01	28	PAD05/AN05
24	PAD02/AN02	27	PAD06/AN06
25	PAD03/AN03	26	PAD07/AN07

H2 Pin	Signal Name	H2 Pin	Signal Name
1	PA7	50	VCC
2	PA6	49	GROUND
3	PA5	48	PE7/XCLK*
4	PA4	47	PK7/ECS*
5	PA3	46	PK5
6	PA2	45	PK4
7	PA1	44	PK3
8	PA0	43	PK2
9	PB7	42	PK1
10	PB6	41	PK0
11	PB5	40	PJ0/KWJ0
12	PB4	39	PJ7/SCL
13	PB3	38	PJ6/SDA
14	PB2	37	PM7/TxCAN3
15	PB1	36	PM6/RxCAN3
16	PB0	35	PM5/TxCAN2
17	PE2/RW*	34	PM4/RxCAN2
18	PE4/ECLK	33	PM3/TxCAN1
19	PE3/LSTRB*	32	PM2/RxCAN1
20	PE1/IRQ*	31	PM1/TxCAN0
21	PJ1/KWJ1	30	PM0/RxCAN0
22	PAD08/AN08	29	PAD12/AN12
23	PAD09/AN09	28	PAD13/AN13
24	PAD10/AN10	27	PAD14/AN14
25	PAD11/AN11	26	PAD15/AN15

Voltage Regulator Configuration

The Adapt9S12D module has an on board low-dropout voltage regulator (LM2937ET-5) to provide stable power to the electronics. It can be seen mounted on the underside of the card. This takes the filtered DC voltage (6-12 VDC) applied to the J1 two pin connector on the board, and provides a smooth regulated 5 Volts DC to the components. The advantage of the regulator chosen is that it does not need a driving voltage much above the target 5 volts to run. If the board is run as a stand alone unit, the regulator will not require a heat sink to operate either.

However, if you plan to use the on board regulator to power your own circuits in addtion to the module, please be aware that the regulator is designed to only supply a maximum of 500 mA of current, even with a heat sink added to it. 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.

If your DC power supply connected to J1 is using a long cord to connect to the module, or is not well regulated DC, or is operating in a low temperature environment, then it is recommended that an additional capacitor be added on the board. You will need an electrolytic capacitor of 10uF at 25 VDC rating, with radial leads about 0.1 inch apart. This will be mounted on the board as C19, which is located directly behind the J1 power connector. Be sure to take standard electrostatic protection when soldering in the part. (i.e. Grounded soldering iron, etc.) The capacitor will have to be oriented on the C19 mounting area so that the positive (+) lead is towards the center of the board. Inserting an electrolytic capacitor backwards is guaranteed to do bad things, so don't do it.

The final power option available is to disable the regulator completely. This is recommended if you need more than 500 mA for your circuits, and will therefore be supplying your own regulated 5 volt DC power to the system. To disable the regulator, cut the wire jumper W13 on the board. W13 is located next to the solder pads that connect to the regulator. This is a permanent change, and breaking the W13 connection will prevent the 5V power you apply via the H1 or H2 headers from flowing into the regulator's output pin should something be connected to J1. Your power supply can feed 5 Volts in at the pins on the H1 or H2 header connectors on either side of the card.

Analog Voltage Reference

The module provides the option of adding a precision voltage reference for measuring analog voltages. You will need to obtain an LM336Z or similar reference device supplied in a TO-92 package. This will be mounted onto the card as U8, which is located next to pin 22 of the H1 header socket. The flat part of the TO-92 package will face towards the center of the board when it is installed. Standard electrostatic precautions are required during installation.

Once this is done, you will be able to program the S12 MCU to measure analog voltages applied to the AD port pins at 10 bit resolution.

Optional Oscillator

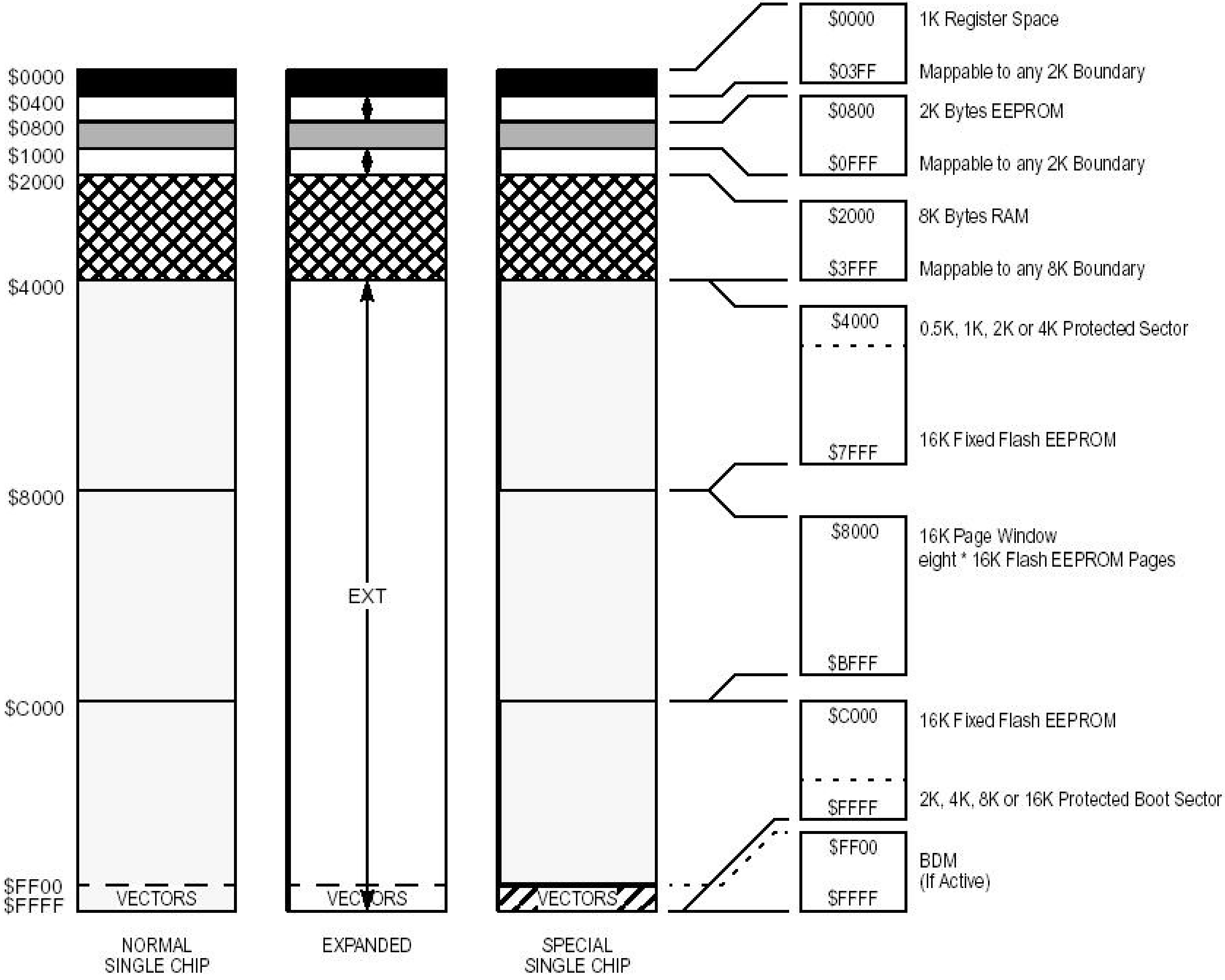
The circuit board was designed to accommodate an optional half-size oscillator circuit, which can be used in place of the supplied crystal osciallator circuit. Here is how to implement this option:
Use an Epson SG531P or similar 5-Volt oscillator. The input voltage level for the EXTAL pin of the MCU is 0 - 2.5V, so a simple resistive voltage divider is implemented via R17 and R18 on the board to scale the oscillator's 5V output waveform by 50%. The recommended value for both R17 and R18 is 1K, and the footprint is designed for standard 0805 surface mount parts. Be sure to use ESD-protection guidelines when handling and soldering the circuit board. First, carefully remove the existing crystal (Y1) and associated capacitors (C1 and C2). Next, mount the 1K surface mount resistors. Finally, install the oscillator over top of the resistors. Pin 1 of the oscillator should be inserted in the pin 1 pad on the circuit board (indicated by a square pad surrounding the hole).   Solder the pins on the back of the circuit board, using a solder containing a no-clean flux core. Before operating the board, place a shorting plug on JB2 so that the XCLK* pin is pulled low. The MCU will now be clocked by the waveform coming from the external oscillator, via the EXTAL pin.

Appendix

Electrical specifications

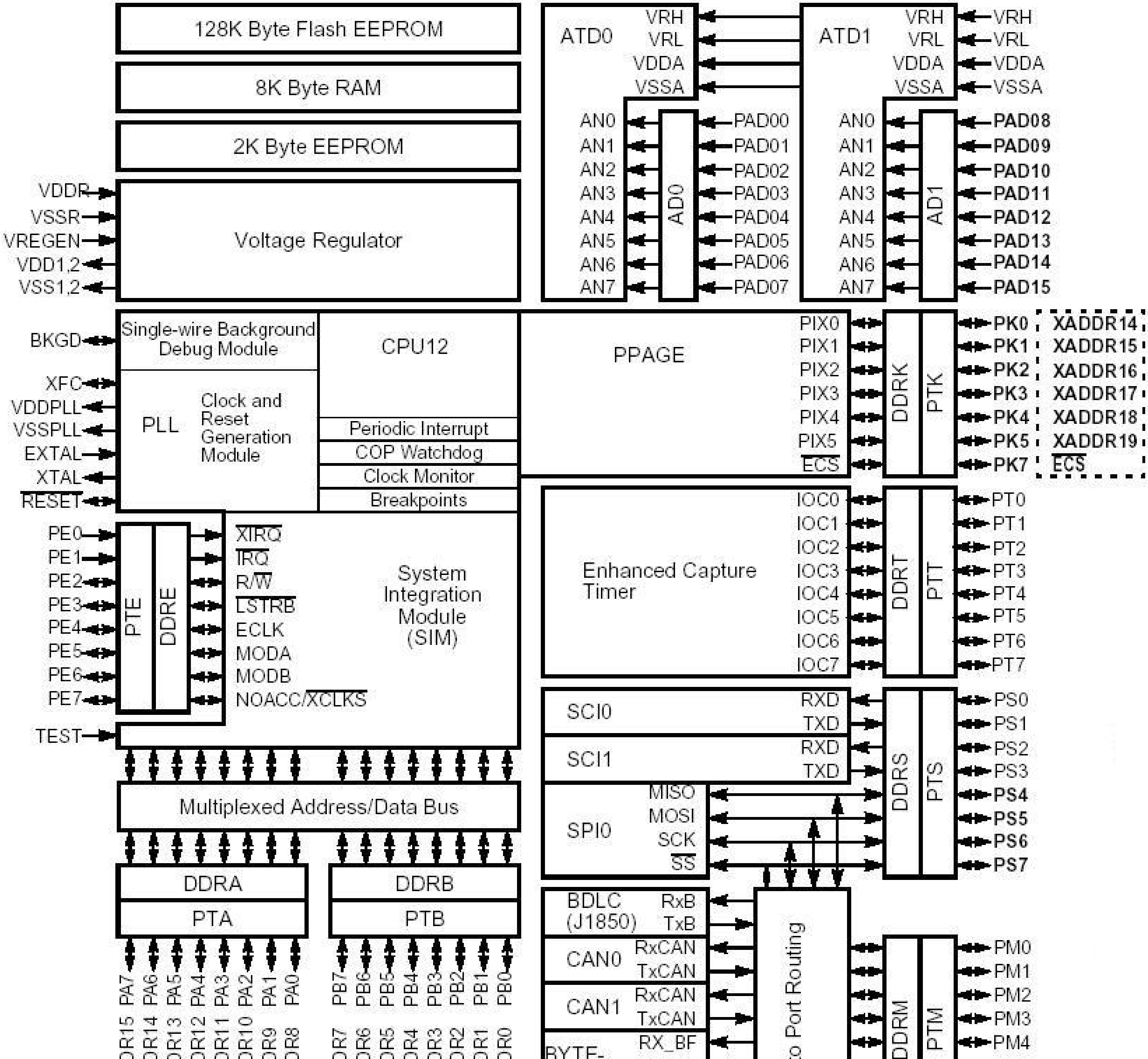
Input Voltage: J1 will accept a positive voltage on Pin 1 from 5-12 Volts DC. The input voltage is passed through up until about 5.7V or so, above which the regulator kicks in. Refer to the voltage regulator data sheet for details.
Operating Current: 60 mA nominal. The on-board regulator can provide up to 500 mA, so external circuitry connected to the expansion connectors can safely be powered from the regulator. Depending on the total current drawn, and the input voltage supplied to J1, heatsinking of the regulator may be necessary.
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 and extended-temperature range crystal or oscillator, or an external clock source.)
Board Dimensions: 3.25"(82.5mm) x 2.3"(58.5mm) x 0.75"(19mm), excluding connectors
Available Communications Interfaces (depending on configuration option):
- 2 x RS232C
- 1 x RS485
- 2 x CAN

Example Memory Map for a Dx128 MCU



The address does not show the map after reset, but a useful map. After reset the map is:
$0000 – $03FF: Register Space
$0000 – $1FFF: 8K RAM
$0000 – $07FF: 2K EEPROM (not visible)

MCU Block Diagram (Dx128 family shown)


128K Byte Flash EEPROM
8K Byte RAM
2K Byte EEPROM
Voltage Regulator
VDDR
VSSR
VREGEN
VDD1,2
VSS1,2
ATD0
VRH
VRL
VDDA
VSSA
AN0
AN1
AN2
AN3
AN4
AN5
AN6
AN7
AD0
PAD00
PAD01
PAD02
PAD03
PAD04
PAD05
PAD06
PAD07
ATD1
VRH
VRL
VDDA
VSSA
AN0
AN1
AN2
AN3
AN4
AN5
AN6
AN7
AD1
VRH
VRL
VDDA
VSSA
PAD08
PAD09
PAD10
PAD11
PAD12
PAD13
PAD14
PAD15
BKGD
Single-wire Background Debug Module
CPU12
XFC
VDDPLL
VSSPLL
EXTAL
XTAL
RESET
PLL
Clock and Reset Generation Module
Periodic Interrupt
COP Watchdog
Clock Monitor
Breakpoints
PPAGE
PIX0
PIX1
PIX2
PIX3
PIX4
PIX5
ECS
DDRK
PTK
PK0
PK1
PK2
PK3
PK4
PK5
PK7
XADDR14
XADDR15
XADDR16
XADDR17
XADDR18
XADDR19
ECS
PE0
PE1
PE2
PE3
PE4
PE5
PE6
PE7
PTE
DDRE
XIRQ
IRQ
R/W
LSTRB
ECLK
MODA
MODB
NOACC/XCLKS
System Integration Module (SIM)
TEST
Enhanced Capture Timer
IOC0
IOC1
IOC2
IOC3
IOC4
IOC5
IOC6
IOC7
DDRT
PTT
PT0
PT1
PT2
PT3
PT4
PT5
PT6
PT7
SCI0
RXD
TXD
SCI1
RXD
TXD
SPI0
MISO
MOSI
SCK
SS
DDRS
PTS
PS0
PS1
PS2
PS3
PS4
PS5
PS6
PS7
Multiplexed Address/Data Bus
DDRA
PTA
DDRB
PTB
PA7
PA6
PA5
PA4
PA3
PA2
PA1
PA0
PB7
PB6
PB5
PB4
PB3
PB2
PB1
PB0
BDLC (J1850)
RxB
TxB
CAN0
RxCAN
TxCAN
CAN1
RxCAN
TxCAN
BYTE-
RX_BF
to Port Routing
DDRM
PTM
PM0
PM1
PM2
PM3
PM4

Notes on Serial Monitor

The "S" version of Adapt9S12D comes pre-programmed with a serial monitor in the topmost 2K of Flash memory. This monitor is from Freescale's Application Note AN2548. It is designed to function using the SCI0 interface at 115200 baud with a 16 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.), open 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/Adapt9S12D/Code/

Last Updated (Wednesday, 17 April 2019 11:53)