A Software Modularity Strategy for Digital Control Systems

Dave Figoli

ABSTRACT

The benefits of structured modulator software is well known. This is especially true for large complex systems with many sub-blocks contributed by many individuals. There must be a standard or a set of rules and guidelines for all contributors to follow. Industry has learned the benefits of these methodologies and has realized the impact it can make on smoother system integration, reduced debug and troubleshooting time, a higher degree of visibility inside the software, and quicker system reconfiguration.

Within the digital control systems (DCS) space, where the TMS320C2000™ (C2000™) ISA (instruction set architecture) has been deployed, the TMS320™ DSP Algorithm Standard addresses the specific needs of application areas, such as digital motor control (DMC), industrial automation (IA), uninterruptible power supplies (UPS), plus a host of other control related areas. The familiar signal flow block diagram representation of digital control systems, as seen in control texts and university publications, lend themselves well to effective action implementation by software-interconnected module blocks (i.e., functions) with well-defined input and output ports. TMS320 DSP Algorithm Standard rules and guidelines work well here, and as a result, have allowed the DCS group to implement a DMC-specific module library (first in a series of libraries), plus numerous DMC systems based upon these building blocks.

Contents

1 Introduction .................................................................................................................................................. 3
2 Scope ......................................................................................................................................................... 4
3 Background ............................................................................................................................................... 5
4 The C2000 Platform .................................................................................................................................. 8
  4.1 Marketing Overview ............................................................................................................................. 8
  4.2 Device Selection Table ........................................................................................................................ 9
5 Modules – The System Building Blocks .................................................................................................... 10
  5.1 One-to-One Mapping Between Software and Signal Flow Block Diagram ........................................ 10
  5.2 Reusability, Compatibility, Predictability, and Expandability ............................................................ 11
  5.3 Module Categorization According to Peripheral and Target Dependence ......................................... 15
    5.3.1 Target Independent Modules ....................................................................................................... 15
    5.3.2 Driver Modules (Target Dependent/Application Configurable) .................................................... 17
    5.3.3 Utility/Debug Modules ................................................................................................................ 18
  5.4 Quick Module Evaluation and Testing ............................................................................................... 18

TMS320, TMS320C2000, and C2000 are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.
6 Addressing the Needs of ASM and C Customers

6.1 ASM Considerations and Specifics

6.1.1 Module Declaration

6.1.2 Variable Declaration and Data Space Allocation

6.1.3 Connecting Modules Together

6.2 C Considerations and Specifics

6.2.1 Module Declaration, Instantiation and Usage

6.2.2 Connecting Modules Together

7 Incremental System Builds Reduce Development Time

7.1 Stepping Through a Typical Incremental Build Process

7.1.1 Check System Vital Signs – Build Level 1

7.1.2 Check PWM Generation at the Target Hardware – Build Level 2

7.1.3 Check Power Inverter Hardware and Open Loop Motor Operation – Build Level 3

7.1.4 Closed Loop Motor Operation Under Voltage Control – Build Level 4

7.1.5 Closed Loop Current Control, Torque Mode Operation – Build Level 5

8 Software Standardization – TMS320 DSP Algorithm Standard Is the Ideal Vehicle

8.1 eXpressDSP Compliancy

8.2 Example of a DMC Algorithm With an IALG Abstract Interface

8.3 Scope of TMS320 DSP Standard Algorithms Within DMC

9 Packaged Solution – Code Composer and Documents Complete the Package

9.1 DCS Module Library Package

9.2 System Package

List of Figures

Figure 1. ACI Sensored FOC – System Block Diagram

Figure 2. ACI Sensored FOC – System Block Diagram Showing Clear One-to-One Mapping to Software

Figure 3. Target and Application Independent Modules (TI/AI) – Examples

Figure 4. Target Independent/Application Configurable Modules (TI/AC) – Examples

Figure 5. Driver Modules (Target Dependent/Application Configurable) (DRV)

Figure 6. Utility/Debug Modules

Figure 7. Testing and Evaluating Modules

Figure 8. Space Vector Generator Module With Magnitude and Frequency Control

Figure 9. Connecting Modules Together Within a System Framework

Figure 10. C-callable Assembly (CcA) Module Example With Multiple Input/Multiple Outputs/History

Figure 11. CcA Module Example With Multiple Input/Multiple Outputs/History

Figure 12. Connecting Modules Together

Figure 13. Final System Build of the Sensored FOC PMSM (Corresponds to PMSM3-1 in Table 1)

Figure 14. Example of Build Level 1 – Checking Vital Signs

Figure 15. Example of Build Level 2 – Checking PWM Driver Operation

Figure 16. Example of Build Level 3 – Checking Power Hardware and Motor Operation

Figure 17. Example of Build Level 4 – Closed Loop Voltage Control Operation

Figure 18. Example of Build Level 5 – Closed-loop Current Control, Torque-mode Operation

Figure 19. System ACI3–1 Implemented With an IALG Layer

Figure 20. Abstract Interfaces and Object Encapsulation

Figure 21. Example of System PMSM3-2 With Possible IALG Interface Implementation
1 Introduction

The benefits of structured modulator software are well known. This is especially true for large complex systems with many sub-blocks contributed by many individuals (e.g., defense systems, telecommunication networks, industrial automation, etc.). In all these cases there is a strong common set of needs. Efforts by contributors should only be used one time, and are reusable for future projects. The contributed software blocks (modules) must be predictable in their behavior and compatible with other blocks, both functionally and numerically. There must be a standard or a set of rules and guidelines for all contributors to follow.

Industry has learned the benefits of these methodologies, and has realized the impact it can make on smoother system integration, reducing debug and troubleshooting time, providing a higher degree of visibility inside the software, and quicker system reconfiguration. These attributes have impact on getting quality products out to market in a timely manner.

Texas Instruments (TI) recognizes that such methodologies need to be applied to high performance DSP systems. To address this, TI has introduced the TMS320 DSP Algorithm Standard, which is part of TI’s eXpressDSP™ technology initiative.

The algorithm standard is the backbone of this effort, and is being deployed across the three key TI platforms (TMS320C2000, TMS320C5000™, and TMS320C6000™). The algorithm standard focuses on a set of general rules and guidelines that can be applied to all algorithm intellectual property (IP). In addition, all algorithms are to comply with a generic resource management API, called IALG. Algorithms that comply with the standard are tested and awarded an “eXpressDSP-compliant” mark upon successful completion of the test.

In addition to providing solid software integration methodologies within a single company environment, the algorithm standard has a more wide-ranging scope for interoperability and compatibility across multiple algorithm vendors and various companies. A benefit of compliance with the standard is the resulting large third-party network of algorithm and IP contributors.

eXpressDSP, TMS320C5000, and TMS320C6000 are trademarks of Texas Instruments.
Within the digital control systems (DCS) space, where the C2000 ISA (instruction set architecture) has been deployed, the algorithm standard has been used to address the specific needs of application areas such as digital motor control (DMC), industrial automation (IA), uninterruptible power supplies (UPS), plus a host of other control related areas. The familiar signal flow block diagram representation of digital control systems, as seen in control texts and university publications, lend themselves well to effective action implementation by software-interconnected module blocks (i.e., functions) with well-defined input and output ports. Algorithm standard rules and guidelines work well here, and as a result, have allowed the DCS group to implement a DMC-specific module library (first in a series of libraries), plus numerous DMC systems based upon these building blocks. In keeping with the algorithm standard philosophy of software interoperability and compatibility across multiple algorithm vendors, this library is a catalyst in promoting a continued expansion of software module offerings fueled by the TI third-party network.

Once a rich set of library modules are available, with a well defined and proven interconnect methodology, systems can be realised fairly quickly. The logical extension is that various what if scenarios of system topologies and/or configurations can be explored relatively easily, and with little overhead. As a more basic or fundamental need, this same ease of module connectability allows a system designer to have a number of incremental build levels, ranging from a rudimentary system (3-4 modules) used to check-out key life signs, to a fully featured final system with a large number of connected modules. This is analogous to the commissioning steps or process which is typical in large engineering projects. This approach to system software commissioning is invaluable during the system integration phase and can greatly reduce the time and effort required.

The discussion so far has assumed text-based software compilation (i.e., the code is written with a text editor and module interconnection is represented as function calls with appropriate variable passing between modules). However, there is no real reason for this limitation, and a natural progression to the module interconnect problem is a graphical one, i.e., modules can be represented as graphical blocks with input and output connectors. A system then becomes a graphical representation of numerous modules all interconnected by signal flow lines analogous to the classical signal flow block diagram found in University texts and publications. This type of approach is not new and numerous companies already offer products like this.

2 Scope

In a continuing effort to provide our customers valuable applications material, the Digital Control Systems Group (DCS) at Texas Instruments has announced algorithm standard support for the C2000 platform. Through this company wide software initiative, TI is providing a large base of applications software collateral, aligned with the modular software strategy known as the TMS320 DSP Algorithm Standard. This collateral will be made available through TI's DCS website under the banner of TI Foundation Software.

The purpose of this application report is to consolidate the many ideas and features associated with numerous components of TI's Foundation software offering and to explain the overall philosophy in one centralized document. Although this document is not intended to replace the documentation supplied with the Web downloadable software, it has sufficient details to give system designers good guidance in the selection and usage of the material offered. This will allow customers to quickly select and customize the most appropriate reference software which will meet their requirements.
3 Background

It is widely agreed that a modular approach to software engineering is the methodology of choice. In the following section, some background thoughts and ideas will be explored which are the driving force behind the DCS modular software effort. Some of the ideas and philosophies are obvious, while others are more subtle. Areas touched upon in this background set the context of more detailed material, which will be covered later throughout this document.

Software documentation greatly benefits from clearly delineated modules and systems

Texas Instruments, as a leading supplier of DSPs, understands that a DSP solution is more than just silicon. Therefore, TI seeks to offer a rich collateral of software examples and working systems which are clearly documented and execute correctly on a customer’s DSP of choice. One area which has received much attention and improvement, is the description or documentation of application software. Through the use of software modularity, it is possible to delineate and give an unambiguous one-to-one mapping between a system block (signal flow) diagram and the actual software code in question. This, together with module documentation for each software block, shows the customer clearly where each piece of software is, what it is doing, and more importantly, the parameters or variables being passed and executed upon. All TI applications which are part of the foundation software offering now consist of module documents and system documents. System documents always include the system block or signal flow diagram, which among control engineers, remains one of the most useful communication tools.

Distinction between peripheral dependent and independent modules makes software porting easy

Clear delineation between peripheral independent modules and those which have dependency (also called peripheral drivers) allows a smooth transition across the various TMS320C24x™ (C24x™) DSP generation members. In addition, it helps in porting software from an evaluation system (EVMs) to a customer’s final system. During the software porting process, knowledge of module dependency enables the customer to focus quickly and efficiently on only those modules which are known to be peripheral drivers. Since the majority of modules are dependent on the C24x or TMS320C20x™ (C20x™) CPU (which are software-compatible), large portions of a system’s software will remain unchanged throughout the process, minimizing debug time and risk.

An example of this is when the PWM modulating signal is separated from the actual PWM generating peripheral. In this case, the module which generates the modulation function is totally independent of PWM carrier frequency, asymmetric or symmetric modes, active high/low convention, dead-band, pre-scaling and other target specific attributes. Hence, the modulation function can be purely mathematical and one of many choices within a library of modulation functions.

Incremental system builds cuts software development time

It is well known that, regardless of how much planning has gone into a system engineering project, chances are the final system won’t work the first time, usually because some subtle target dependency or requirement has been overlooked. This is normal engineering, and if a contingency plan or debug methodology has been comprehended, it’s not a problem. However, if not anticipated ahead of time, this can become frustrating and time-consuming.

TMS320C24x, C24x, TMS320C20x, and C20x are trademarks of Texas Instruments.
To prevent this, the system comes pre-packaged within a framework incorporating various system build levels or debug steps as part of the normal system software flow. A customer can move from a current build level to any other build level quickly, and without risk of losing configuration or run time information related to previous build levels. This allows a customer to commission system software in a step-by-step manner, and at each step validate key vital signs before moving on. The number of build levels may vary depending on final system complexity, but an important point to note, is that regardless of the number of build levels, this framework becomes a common point of reference should a customer need added assistance from hot-line support or through direct interaction with TI Applications personnel. In either case, it will greatly improve the chances of system software success.

As will be seen later, incremental system build (ISB) levels are supported by frameworks with various configurations of interconnected modules. Frameworks also provide invaluable skeletons for customers to modify or customize as required to suit their own target system requirements.

Well defined module input/output variables provide clear probe/debug points within the software

Knowing that the system/module framework has one-to-one mapping with the system signal flow diagram, makes it easy to probe signals of interest and to visualize them in real time. TI provides useful utility modules which can display time-varying signals (i.e., software variables) via a hardware DAC using an oscilloscope, or on screen with Code Composer™ using the data-logging module. The DAC_VIEW and DATA_LOG modules can be readily connected by way of pointers during run time to any module terminal variable and internal module variables for greater software visibility.

Peripheral driver modules enforce read-modify-write methodology

Peripherals are shared resources. Often a peripheral (e.g., Event Manager) can have many functions which may be used collectively with dependent interaction, or simply as independent stand-alone functions. Peripheral drivers can be regarded as objects with functional groupings rather than logical or physical silicon groupings. For this reason, it is important not to disturb any prior configuration which may have been initialized by another object. Peripheral functions may sometimes share the same control registers, resulting in enforcement of a read-modify-write methodology. This the only way to support various objects sharing the same peripheral.

Software modules make excellent sample code, especially for peripheral drivers

By definition, all modules have a well defined and documented input/output relationship. This cause and effect relationship between input and output variables provides easy-to-follow code samples which are narrow in context and help a customer’s understanding and focus. This is true especially for peripheral driver modules which have specific peripheral actions. Knowing these actions and how to cause them significantly accelerates a customer’s understanding of how to effectively deploy the many peripherals and to be productive quickly.

Known working library modules allow a “trust for now - investigate later” approach to development

Modules can provide a valuable starting point, even if they are not exactly the final requirement. Known good modules can be used initially on a trust for now basis as a system is built up and valuable experience is gained. Later on, these standard modules can be investigated on an as-needed basis, to learn more, or to modify or customize to suit. A modular approach is ideal for this, allowing standard modules to be swapped out for customized ones later on without risk or delays.

Code Composer is a trademark of Texas Instruments.
Software standardization – TMS320 algorithm standard is the ideal vehicle

As software reaches higher levels of complexity, there is a need for code reusability and standardization. C is the natural choice, together with the TMS320 DSP Algorithm Standard as the backbone. These algorithm standards are wrapped in an API layer known as IALG, which provides the packaging necessary for IP exchange amongst TI, customers and third parties. IALG lends itself well as a wrapper for larger and more complex algorithms and hence is suitable for encapsulation of entire control systems.

The TMS320C24x generation of DSPs is classified as DSP controllers focused on the digital control space. From an algorithm standpoint, the control space is characterized by systems built up from many smaller and reusable software blocks or modules, such as: PID controllers, coordinate transformations, trigonometric transformations, signal generators, etc. In addition, the C24x™ DSP controllers are offered in numerous memory configurations, with lower-cost devices having 4KW of program memory. This may impose some restrictions on how much overhead can be wrapped on each of these smaller modules when creating its IALG interface.

To better address these sensitivities within the control space, and in keeping true to the algorithm standard philosophy, it has been decided to encapsulate algorithms at two levels:

- **eXpressDSP-compliant algorithms**, which include IALG and are fully compliant to the standard; and

- **DCS modules**, which adhere to most of the rules and all of the good programming practices, but do not have an IALG layer implemented. DCS modules can be regarded as adhering to the algorithm standard. They provide the additional benefit of allowing software designers to quickly and efficiently build up eXpressDSP compliant algorithms based on a collection of useful DCS modules.

Although the algorithm standard calls for software to be written in C at the IALG level, DCS modules need not be. They can be made C-callable assembly as long as compliance is maintained (e.g., re-entrant, preemptive, relocatable, etc.). This allows algorithms and modules to be sensitive to the small memory spaces and critical execution often required in the control space, allowing greater efficiency and performance where needed.

**Addressing the needs of ASM customers**

Many software designers want the flexibility of having control at the bit level. Assembly language allows customers to get close to the TMS320C2x™ (C2x™) architecture and peripherals. TI recognizes that, within the memory limited and cost sensitive world of low-end digital control systems, certain customers prefer to program in the assembly-only domain. In response to this need, a set modules and systems (frameworks) is offered, which are written fully in assembly. They are fully relocatable, and borrow much from the algorithm standard philosophy and good programming practices outlined in the rules and guidelines. Moreover, nothing stops a customer from integrating any of the C-callable software modules into an assembly only framework and making a function call at the assembly level.

TMS320C2x and C2x are trademarks of Texas Instruments.
Packaged solution – Code Composer and documentation completes the package

The solutions collateral outlined so far for the digital control systems foundation software is comprehensive with many component parts. To bring it all together and to allow customers to test drive the many solution quickly, TI offers Code Composer for the C2000 platform as the Integrated Development Environment (IDE). Ready-to-run Code Composer projects and workspaces ensure customers are jump-started into productivity quickly. To complement the IDE, clear module and system documentation tie the solutions together and help customers understand how theory, implementation, and debug all come together to provide a working DSP solution.

4 The C2000 Platform

4.1 Marketing Overview

The C2000 platform currently consists of three generations of C2x CPU core-based DSP controllers. The first generation F240 devices, released in 1996, were the industry’s first Flash-based DSP controllers, and were optimized specifically for the digital motor control (DMC) market. Second generation F243/F242/F241 devices are more cost-optimized, and have been enhanced with the addition of a CAN controller and a faster 850nS ADC converter. Additionally, versions without an external memory interface are offered, making smaller packaging options available. More recently, the third generation LF2402, LF2406, and LF2407 devices have been announced and released. These DSP controllers are 3.3V chips with performance increased from 20 to 30 MIPS (40 MIPS in 4Q00). The 10-bit ADC has been totally overhauled to provide a sub-500ns conversion time with a sophisticated 16-buffer auto-sequencer. Flash size has also been increased to 32KW, supporting larger more complex systems. Because more program space requires more data space for variables, on-chip RAM has been increased to 2.5KW. This RAM can also function as program space and can be loaded at boot-up time via an on-chip ROM-based bootloader, which is standard on all F240x devices. Table 1 shows device selections for the 24x products.

To provide true multi-axis support, two Event Managers are offered on the x2404, x2406, and x2407 devices, providing four timers and 12 complementary PWMs with dead-band control. This allows control of two 3-phase motors configured to run either sensored or sensorless. Additionally, four auxiliary PWMs can support other functions like PFC, solenoid control, DAC functions, etc. With 30 MIPS and later 40 MIPS performance, the C24x devices are geared up to handle all the industry-accepted sensorless control schemes, which can be math-intensive, especially in the case of two concurrent motor systems, as is the case in two axis control.

To ensure continuity to higher levels of performance and system integration, TI has announced a fully code-compatible roadmap with the TMS320C28x™ (C28x™) generation. This generation will offer a highly C-optimized DSP core, setting the industry standard for C performance in the embedded market place.

TMS320C28x and C28x are trademarks of Texas Instruments.
### 4.2 Device Selection Table

#### Table 1. Device Selection Table for the 24x Product Offerings

<table>
<thead>
<tr>
<th></th>
<th>LF2407</th>
<th>LF2406</th>
<th>LF2402</th>
<th>LC2406</th>
<th>LC2404</th>
<th>LC2402</th>
<th>F240</th>
<th>C240</th>
<th>F241</th>
<th>C242</th>
<th>F243</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS</td>
<td>30/40</td>
<td>30/40</td>
<td>30/40</td>
<td>30/40</td>
<td>30/40</td>
<td></td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>RAM</td>
<td>2.5Kw</td>
<td>2.5Kw</td>
<td>544w</td>
<td>2.5Kw</td>
<td>1.5Kw</td>
<td>544w</td>
<td>544w</td>
<td>544w</td>
<td>544w</td>
<td>544w</td>
<td>544w</td>
</tr>
<tr>
<td>Flash</td>
<td>32Kw</td>
<td>32Kw</td>
<td>8Kw</td>
<td></td>
<td></td>
<td></td>
<td>16Kw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROM</td>
<td></td>
<td></td>
<td></td>
<td>32Kw</td>
<td>16Kw</td>
<td>4Kw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16Kw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boot ROM</td>
<td>256w</td>
<td>256w</td>
<td>256w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ext. Memory I/F</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event Manager</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GP timers</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Watchdog timer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>10-bit ADC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Channels</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Conv. time (min)</td>
<td>500ns</td>
<td>500ns</td>
<td>500ns</td>
<td>500ns</td>
<td>500ns</td>
<td>500ns</td>
<td>6.6us</td>
<td>6.6us</td>
<td>850ns</td>
<td>850ns</td>
<td>850ns</td>
</tr>
<tr>
<td>SPI</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>SCI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CAN</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Digital I/O pins</td>
<td>41</td>
<td>41</td>
<td>21</td>
<td>41</td>
<td>41</td>
<td>21</td>
<td>28</td>
<td>28</td>
<td>26</td>
<td>26</td>
<td>32</td>
</tr>
<tr>
<td>Voltage range</td>
<td>3.3 V</td>
<td>3.3 V</td>
<td>3.3 V</td>
<td>3.3 V</td>
<td>3.3 V</td>
<td>3.3 V</td>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
</tr>
<tr>
<td>Packaging</td>
<td>144</td>
<td>100</td>
<td>64</td>
<td>100</td>
<td>100</td>
<td>64</td>
<td>132</td>
<td>132</td>
<td>68 PLCC</td>
<td>68 PLCC</td>
<td>144</td>
</tr>
<tr>
<td>SPI</td>
<td>TQFP</td>
<td>TQFP</td>
<td>TQFP</td>
<td>TQFP</td>
<td>TQFP</td>
<td>TQFP</td>
<td>PQFP</td>
<td>PQFP</td>
<td>64 PQFP</td>
<td>PQFP</td>
<td>TQFP</td>
</tr>
<tr>
<td>SCI</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
</tr>
<tr>
<td>CAN</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
<td>4Q00</td>
<td>3Q00</td>
<td>3Q00</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
<td>Now</td>
</tr>
<tr>
<td>Samples</td>
<td>$10.45</td>
<td>$9.95</td>
<td>$8.75</td>
<td>$5.95</td>
<td>$5.45</td>
<td>$2.95</td>
<td>$14.28</td>
<td>$9.69</td>
<td>$11.37</td>
<td>$3.96</td>
<td>$12.85</td>
</tr>
<tr>
<td>Pricing (10Ku)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5 Modules – The System Building Blocks

5.1 One-to-One Mapping Between Software and Signal Flow Block Diagram

It is well known that systems and control engineers like to visualize systems or control strategies in the form of signal flow block diagrams. It makes good sense to separate main functions and to show how each is interrelated by explicit connection to other functions. Each function should be self-contained, and its boundaries or interface clearly delineated. This modular thinking and block diagram approach was taught to us at University, and is commonly used in many university texts and papers to explain complex control systems. A Digital motor control (DMC) system is a good example of this. Figure 1 shows a typical signal flow block diagram of an AC Induction motor controlled by a field-oriented control strategy. This is a useful representation, and it is classically found in many texts on DMC. However, several limitations are evident when trying to relate this diagram to an actual software implementation of the same.

By adopting a modular strategy and enforcing some clear definitions, the classic block diagram shown in Figure 1 can be rearranged and redrawn to reveal a great amount of information about the actual software which is used to implement such a system. This new style of system representation can be seen in Figure 2. The advantages and features of such a representation will be expanded upon in later sections, but a summary of key points are given here:

- The system block diagram has a clear one-to-one mapping to the modular system software.
- Each module graphic or block represents a self-contained software function or object of the same name.

Figure 1. ACI Sensored FOC – System Block Diagram

It is not usually clear how software variables are related to the signal parameters on the diagram, nor where in the software these parameters can be found and accessed. Moreover, the boundary between software and hardware is blurred, (i.e., where the software control the on-chip peripheral and where the peripheral output pins control the external hardware, such as the PWM-to-inverter interface.)

By adopting a modular strategy and enforcing some clear definitions, the classic block diagram shown in Figure 1 can be rearranged and redrawn to reveal a great amount of information about the actual software which is used to implement such a system. This new style of system representation can be seen in Figure 2. The advantages and features of such a representation will be expanded upon in later sections, but a summary of key points are given here:

- The system block diagram has a clear one-to-one mapping to the modular system software.
- Each module graphic or block represents a self-contained software function or object of the same name.
- Input and output terminals of each module correspond exactly to global variables within the software function.
- Modules are categorized (color coded) to clearly designate dependencies on peripherals and target hardware.
- Connections between modules show data flow via corresponding input/output variables.
- Each module is reusable and has its own documentation explaining usage and instantiation.

Figure 2. ACI Sensored FOC – System Block Diagram Showing Clear One-to-One Mapping to Software

5.2 Reusability, Compatibility, Predictability, and Expandability

Engineering reuse in today’s competitive environment is critical. Software modularity is the perfect vehicle to achieve this, especially in DMC systems. If we examine various motor control systems, it becomes clear that a large degree of commonality exists between them. Table 2 illustrates the degree of reuse possible among nine typical systems. The PID regulator, for example, is useful across all systems. Therefore, if each module is realized only once but implemented according to well defined guidelines, then compatibility and predictability can be assured across all modules. Since this approach allows efficient reusability, efforts which may typically be used to “reinvent the wheel” can be re-deployed on expanding the module library base for greater functionality and features.
Table 2. Module vs. DMC System Matrix

<table>
<thead>
<tr>
<th>MODULES</th>
<th>DMC SYSTEM</th>
<th>AC11</th>
<th>PMSM1</th>
<th>PMSM2</th>
<th>BLDC1</th>
<th>BLDC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHZ_PROF</td>
<td>SVPWM</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SVGEN_MF</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_PWM_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PID_REG</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAP_EVT_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEED_PRD</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLARK</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLARK_I</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARK_I</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARK</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QEP_THETA_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVGEN_DQ</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILEM2_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILEG2_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEED_FREQ</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUX_ANGLE</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC04_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINECOS_PH</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPEN</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLDC_PWM_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMTN_TRIG</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALL3_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRAS_SPEED</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMSM_SMO</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHASE_V3</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILEG2_VBUS_DRV</td>
<td>DRV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAND_GEN</td>
<td>Ti/AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLOG_VIEW</td>
<td>Util</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DACInView_DRV</td>
<td>Util</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- √O Optional
- √x N # of same module needed in system
- DRV Peripheral Driver (i.e., Target Dependent and Application configurable)
In Table 2, the modules column represents a subset of useful DMC functions which are all standalone and part of an expanding library. Check marks in each of the DMC system columns (e.g., ACI3–1, PMSM3–1, etc.) designate which of the library modules are required to implement that corresponding system. Some modules are used multiple times. For those that are optional or just useful for debug, refer to the table notes. As already mentioned, modules are also categorized according to type. This will be covered in detail in later text. For a description of each module’s function and type, refer to Table 4.

Nine typical DMC systems are described in more detail in Table 3. Here each system is described briefly together with its corresponding motor type. The systems outlined cover various control strategies, such as scalar and vector control. In each case, both a sensed and sensorless version is listed. Examination of Table 2 reveals that, in most cases, the difference between a sensed and sensorless system is only one or two modules, e.g., a position or speed estimator. The remaining modules are common. Therefore, in keeping with the reuse philosophy, design efforts can be focused on expanding the library with more robust estimators which meet various system requirements, rather than recreating entire sensed system infrastructures.

<table>
<thead>
<tr>
<th>System</th>
<th>Motor Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI1–1</td>
<td>1 ph AC induction</td>
<td>Sensored – Tacho i/p VHz/SinePWM/Closed loop (CL) speed PID</td>
</tr>
<tr>
<td>ACI3–1</td>
<td>3 ph AC induction</td>
<td>Sensored – Tacho i/p VHz/SVPWM/Closed loop speed PID</td>
</tr>
<tr>
<td>ACI3–2</td>
<td>3 ph AC induction</td>
<td>Sensorless – MRAS (speed estimator) VHz/SVPWM/Closed loop speed PID</td>
</tr>
<tr>
<td>ACI3–3</td>
<td>3 ph AC induction</td>
<td>Sensored – Tacho i/p FOC/SVPWM/Closed loop current PID for D,Q/CL speed PID</td>
</tr>
<tr>
<td>ACI3–4</td>
<td>3 ph AC induction</td>
<td>Sensorless – Direct Flux estimator + speed estimator FOC/SVPWM/Closed loop current PID for D,Q/CL speed PID</td>
</tr>
<tr>
<td>ACI3–5</td>
<td>3 ph AC induction</td>
<td>Sensorless – Kalman (speed estimator) FOC/SVPWM/Closed loop current PID for D,Q/CL speed PID</td>
</tr>
<tr>
<td>PMSM3–1</td>
<td>3 Ph Permanent Magnet Synch</td>
<td>Sensored – QEP FOC/SVPWM/Closed loop current PID for D,Q/CL speed PID</td>
</tr>
<tr>
<td>PMSM3–2</td>
<td>3 Ph Permanent Magnet Synch</td>
<td>Sensorless – SMO (Sliding mode observer) position estimator FOC/SVPWM/Closed loop current PID for D,Q/CL speed PID</td>
</tr>
<tr>
<td>BLDC3–1</td>
<td>3 ph Trapezoidal Brushless DC</td>
<td>Sensored – 3 Hall effect i/p Trapezoidal/Closed loop current PID/CL Speed PID</td>
</tr>
<tr>
<td>BLDC3–2</td>
<td>3 ph Trapezoidal Brushless DC</td>
<td>Sensorless – BEMF/Zero crossing detection Trapezoidal/Closed loop current PID/CL Speed PID</td>
</tr>
</tbody>
</table>
### Table 4. Sample of DMC Module Descriptions and Type Category

<table>
<thead>
<tr>
<th>#</th>
<th>Module</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FC_PWM_DRV</td>
<td>Full Compare PWM driver (configurable for active high/low, dead-band, or asm/sym)</td>
<td>Driver</td>
</tr>
<tr>
<td>2</td>
<td>CAP_EVT_DRV</td>
<td>Capture input event driver with 16 time stamps and pre-scaler selection</td>
<td>Driver</td>
</tr>
<tr>
<td>3</td>
<td>QEP_THETA_DRV</td>
<td>Quadrature Encoder Pulse interface driver with position (theta) as output</td>
<td>Driver</td>
</tr>
<tr>
<td>4</td>
<td>ILEM2_DRV</td>
<td>LEM based 3-phase current measurement driver (2 meas method)</td>
<td>Driver</td>
</tr>
<tr>
<td>5</td>
<td>ILEG2_DRV</td>
<td>Leg shunt resistor based on 3-phase current measurement driver (2 meas method)</td>
<td>Driver</td>
</tr>
<tr>
<td>6</td>
<td>ADC04_DRV</td>
<td>General purpose 4-conversion ADC driver with Gain/offset and channel selection</td>
<td>Driver</td>
</tr>
<tr>
<td>7</td>
<td>BLDC_PWM_DRV</td>
<td>BLDC PWM driver – uses high-side chopping and fixed on/off for low side</td>
<td>Driver</td>
</tr>
<tr>
<td>8</td>
<td>HALL3_DRV</td>
<td>Hall effect interface driver for sensored 3-phase BLDC trapezoidal control</td>
<td>Driver</td>
</tr>
<tr>
<td>9</td>
<td>LEG_VBUS_DRV</td>
<td>Leg shunt resistor and DC bus voltage meas driver used for MRAS ACI</td>
<td>Driver</td>
</tr>
<tr>
<td>10</td>
<td>DAC_VIEW_DRV</td>
<td>4-channel DAC driver (for EVM) useful for displaying real-time variables on scope</td>
<td>Util</td>
</tr>
<tr>
<td>11</td>
<td>VHz_PROF</td>
<td>Volts/hertz profile for ACI (voltage vs. frequency)</td>
<td>TI/AC</td>
</tr>
<tr>
<td>12</td>
<td>PID_REG</td>
<td>Proportional/integral/derivative controller with 32-bit integration</td>
<td>TI/AC</td>
</tr>
<tr>
<td>13</td>
<td>SPEED_PRD</td>
<td>Speed calculator based on period measurement between events (speed = 1/period)</td>
<td>TI/AC</td>
</tr>
<tr>
<td>14</td>
<td>SPEED_FRQ</td>
<td>Speed calculator based on frequency measurement – tacho style method</td>
<td>TI/AC</td>
</tr>
<tr>
<td>15</td>
<td>FLUX_ANGLE</td>
<td>Flux angle model for 3-phase ACI vector control</td>
<td>TI/AC</td>
</tr>
<tr>
<td>16</td>
<td>COMPEN</td>
<td>DC ripple compensator for single phase ACI variable speed drives</td>
<td>TI/AC</td>
</tr>
<tr>
<td>17</td>
<td>COMTN_TRIG</td>
<td>Commutation trigger generator for BLDC sensorless trapezoidal/BEMF/ZC tech.</td>
<td>TI/AC</td>
</tr>
<tr>
<td>18</td>
<td>PMSM_SMO</td>
<td>PMSM sliding mode observer for position estimation in sensorless 3-phase vector drives</td>
<td>TI/AC</td>
</tr>
<tr>
<td>19</td>
<td>ACI_M Ras</td>
<td>Model Reference Adaptive System speed estimator used for 3-phase ACI drives</td>
<td>TI/AC</td>
</tr>
<tr>
<td>20</td>
<td>PHASE_V3</td>
<td>3-phase voltage reconstruction function based on PWM duty cycle inputs</td>
<td>TI/AC</td>
</tr>
<tr>
<td>21</td>
<td>PFC_2V</td>
<td>Power factor correction based on 2-voltage meas method</td>
<td>TI/AC</td>
</tr>
<tr>
<td>22</td>
<td>SVGEN_MF</td>
<td>Space vector generator function with magnitude and frequency control (scalar drives)</td>
<td>TI/AC</td>
</tr>
<tr>
<td>23</td>
<td>CLARK</td>
<td>Clark transform – 3-phase to 2-phase (quadrature) conversion</td>
<td>TI/AC</td>
</tr>
<tr>
<td>24</td>
<td>CLARK_I</td>
<td>Inverse Clark transform – 2-phase (quadrature) to 3-phase conversion</td>
<td>TI/AC</td>
</tr>
<tr>
<td>25</td>
<td>PARK</td>
<td>Park transform – Stationary to rotating reference frame conversion</td>
<td>TI/AC</td>
</tr>
<tr>
<td>26</td>
<td>PARK_I</td>
<td>Inverse park transform – Rotating to stationary reference frame conversion</td>
<td>TI/AC</td>
</tr>
<tr>
<td>27</td>
<td>SVGEN_DQ</td>
<td>Space vector generator function with quadrature control (vector drives)</td>
<td>TI/AC</td>
</tr>
<tr>
<td>28</td>
<td>SINCOS_PH</td>
<td>2-phase sine generator function with variable phase control</td>
<td>TI/AC</td>
</tr>
<tr>
<td>29</td>
<td>RAND_GEN</td>
<td>Random generator function – useful in randomizing PWM modulation</td>
<td>TI/AC</td>
</tr>
<tr>
<td>30</td>
<td>DLOG_VIEW</td>
<td>Data logging utility – useful for variable graphing in Code Composer</td>
<td>Util</td>
</tr>
</tbody>
</table>
5.3 Module Categorization According to Peripheral and Target Dependence

Understanding the exact dependencies of a software module is very important. This knowledge is invaluable during debugging, software porting from one target to another, and on planning a system commissioning strategy. The modules which are part of the DCS foundation software library (examples of which are shown in Table 4) are categorized into two main types:

- Target “DSP” Independent (TI); and
- Drivers (i.e., target dependent and application configurable).

For convenience, the module graphics throughout the documentation are color coded to help customers quickly identify a module’s dependency within a system. Color designation is as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target independent/application independent</td>
<td>TI/AI</td>
</tr>
<tr>
<td>Target independent/application configurable</td>
<td>TI/AC</td>
</tr>
<tr>
<td>Drivers (Target dependent/application configurable)</td>
<td>Drv</td>
</tr>
<tr>
<td>Utility/Debug</td>
<td>Util</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Color Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>TI/AI</td>
</tr>
<tr>
<td>Lite Red</td>
<td>TI/AC</td>
</tr>
<tr>
<td>Blue</td>
<td>Drv</td>
</tr>
<tr>
<td>Green</td>
<td>Util</td>
</tr>
</tbody>
</table>

5.3.1 Target Independent Modules

Target independent modules do not directly access or modify control or status registers of any on-chip or off-chip peripherals, i.e., target DSP-specific resources. These modules are dependent only on the C2x CPU core. Target independent modules are totally portable across the entire C24x and C20x generations. Target independent modules can be further subdivided into two groups:

- Application configurable (AC); and
- Application independent (AI).

Examples of both of these module types are shown in Figure 3 and Figure 4, respectively.

TI/AI modules are typically characterized by mathematical type functions (e.g., trigonometric relationships, transforms, matrix operations, waveform generators, etc.). These functions are fixed and fairly standardized, and do not require configuration, knowledge of peripherals, or the end application itself.
TI/AC modules also do not need knowledge of device peripherals, but do need to be configured depending on the end application. Examples of these modules are PID controllers, speed estimators, voltage, current models, etc. In most cases, physical system or application parameters need to be known ahead of time, so these modules can be configured to operate appropriately and most optimally. Since these modules are dependent on physical quantities (e.g., current, voltage, duty-cycle, etc.), they rely heavily on driver modules that interface to peripherals such as ADC, PWM generator, event capture, etc. In all cases, TI/AC modules do not access these peripherals directly.

Figure 3. Target and Application Independent Modules (TI/AI) – Examples

Figure 4. Target Independent/Application Configurable Modules (TI/AC) – Examples
5.3.2 **Driver Modules (Target Dependent/Application Configurable)**

Driver modules are the interface between software and DSP device-specific peripheral hardware. Examples of such modules are shown in Figure 5. These modules have direct access to peripheral control and status registers and are dependent on one of the three device groups within the C24x generation.

### Device Group Designation

- LC/LF240 x240
- LC/LF24x x243
- LC/LF240x x2407

For ASM assembly-based drivers, a single file exists for each driver. This driver file can be re-targeted to any one of the device groups by modifying the included header file (x24xx_app.h, for example). The header configuration below has selected the target device to be an x243 derivative:

```
; Select the target device by setting 1

x240 .set 0 ; C/F240
x243 .set 1 ; C/F243/242/241
x2407 .set 0 ; C/F2407/2406/2404/2402
```

For the C-callback (CcA) assembly-based drivers, a separate driver file exists for each device group. System re-targeting is accomplished by linking in the appropriate driver file.

To ensure robust driver modules, a Read-Modify-Write control register access methodology has been adopted in all driver implementation. Peripherals are shared resources. Often a peripheral (e.g., Event Manager) can have many functions which may be used collectively with dependent interaction or simply as independent stand-alone functions. Driver modules can be regarded as objects with functional groupings rather than logical or physical silicon groupings. For this reason, it is important not to disturb any prior configuration which may have been initialized by another object. Peripheral functions may sometimes share the same control registers. Therefore, enforcement of a Read-Modify-Write methodology is the only way to support various objects sharing the same peripheral.

![Figure 5. Driver Modules (Target Dependent/Application Configurable) (DRV)](image-url)
5.3.3 Utility/Debug Modules

Utility and/or debug modules are mainly used during the software development and debug process. Typically they are removed at time of software completion, however they can also be left in the code for system diagnosis if required during field tests or evaluation. Two examples of these modules are shown in Figure 6. Both of these modules allow the software designer to probe any of the system variables which are in Q15 format, and display them in real time via a scope or a graphical output within Texas Instrument’s Code Composer Emulation/Debug environment.

Module DAC_VIEW uses the four-channel 12-bit DAC found on all C24x EVMs to output waveforms to a scope, while DLOG_VIEW provides a dual memory buffer (i.e., two channels) with trigger feature that allows Code Composer to display two graphical waveforms in real time. Graphical waveforms are continuously updated via the JTAG debug link while the customer’s application software continues to run. Both JTAG and the real-time data flow are non-intrusive to application software running on any C24x generation devices.

![Figure 6. Utility/Debug Modules](image)

5.4 Quick Module Evaluation and Testing

Apart from the more obvious benefits of software modularity previously described, some of the same ideas can be used to facilitate quick module testing or evaluation, i.e., checking how applicable or how a module performs. Additionally, it becomes easy to test several “what if” scenarios by simply reconnecting modules and evaluating several alternatives.

Figure 7 shows a “module under test” setup where a known input stimulus is applied (using a wavegen module, e.g., SINCOS_PH to SVGEN_DQ). The input stimulus and output response of this module under test can be conveniently monitored in real time on a scope via the DAC_VIEW utility module.

Apart from evaluating or confirming operation of a known good software module, this technique is useful when developing new modules. During debug, input stimulus signals can be swept over the entire input range to look for boundary condition problems or discontinuities in output response. It should be noted that this process is valid, provided known good stimulus modules are used as input signals. This technique allows a software designer to check validity of variable ranges, and to ensure waveform integrity over a required frequency range and system sampling rates.
6 Addressing the Needs of ASM and C Customers

Software is experiencing a transitional period within the embedded DSP marketplace. Assembly code is making way to C and C++ is also emerging in many DSP applications as the high-level language (HLL) of choice. Even so, assembly customers still make up a large portion of the digital control systems community.

Assembly language allows customers to get close to the C2x architecture and peripherals. TI recognizes that within the memory-limited and cost-sensitive world of low-end digital control systems, certain customers are reluctant to let go of their assembly-only domain. To address this, as part of TI’s C2000 foundation software, a set of modules and systems (frameworks) is offered which is written fully in assembly. The modules have been made fully relocatable, and borrow much from the Algorithm Standard philosophy and good programming practices outlined in the rules and guidelines.

As software reaches higher levels of complexity, there is a need for code reusability and standardization. C is the natural choice here. Moving forward, C-based modules and systems will form a more important part of TI’s C2000 foundation software offering. Moreover, many embedded DSP systems are sensitive to small memory spaces and critical execution times, which are often a requirement in the competitive control space. To best address this, a combination of full C and C callable assembly (CcA) systems and modules is made available. At the system level or framework, full C ensures maximum flexibility and maintainability. At the module level, CcA functions (or objects) allow memory and cycle efficient code to be developed, with all the desired attributes one expects from a C function (i.e., to be re-entrant, preemptive, relocatable, instanced multiple times, etc.).
With the announcement of the new more powerful and code compatible C28x generation, a C++
compiler will be the standard offering in the Code Composer tool set.

6.1 ASM Considerations and Specifics

This section provides an overview of various aspects a customer will find useful when using an
assembly-based module from the DCS C2000 Foundation library. As an example, the scalar
version of the space vector generator module (SVGEN_MF) is presented. The graphic symbol
for this module is shown in Figure 8.

![Graphic Symbol](image)

Figure 8. Space Vector Generator Module With Magnitude and Frequency Control

6.1.1 Module Declaration

The declaration for this module is shown below and is included in the system framework. Note
most modules have both a main function call (SVGEN_MF) and a one time initialization call
(SVGEN_MF_INIT). The graphic input/output terminals have corresponding software variables
(sv_gain, sv_offset, sv_freq). In addition, even though not shown as an input terminal
on the graphic, a configuration or initialization variable is also made global. In this case, used to
set an upper frequency limit.

```
; Reference/Prototype
;--------------------------------------------------------------
.ref SVGEN_MF, SVGEN_MF_INIT ;function call
.ref sv_gain, sv_offset, sv_freq ;Inputs
.ref Ta, Tb, Tc ;Outputs
.ref sv_freq_max ;Config / Init
```

6.1.2 Variable Declaration and Data Space Allocation

Each module has its own data RAM requirements declared for all of its local and global
variables.

```
; Variable Instantiation (declaration)
;--------------------------------------------------------------
ALPHA_SV .usect "svgen_mf",1
STEP_ANGLE_SV .usect "svgen_mf",1
ENTRY_NEW .usect "svgen_mf",1
ENTRY_OLD .usect "svgen_mf",1
SR_ADDR .usect "svgen_mf",1
SECTOR_PTR .usect "svgen_mf",1
dx .usect "svgen_mf",1
dy .usect "svgen_mf",1
T .usect "svgen_mf",1
Ta .usect "svgen_mf",1
Tb .usect "svgen_mf",1
```

A Software Modularity Strategy for Digital Control Systems
By using uninitialised section variables (.usect), the module variables are made fully relocatable and can be placed anywhere in the on-chip or off-chip data space.

6.1.3 Connecting Modules Together

Since all modules have a very clear interface and are self contained with regards to relocation, local and global variables, it is then a simple procedure to connect modules together (i.e., have them interact with each other to form a system). Systems typically consist of a top level framework with numerous function calls to various modules. Figure 9 shows an example of three modules connected together in graphical form and the corresponding asm code required to achieve this. The one-time initialization calls (SVGEN_DQ_INIT, RAND_GEN_INIT, etc.) are implied but have been left out for clarity. The \texttt{ldp#mfunc\_c1} instruction before the group of \texttt{bldd} instructions is needed to align the data page pointer to the appropriate data page in which the module variables reside. Any of the input terminal variable names can be chosen in the destination module, In this case the destination module (i.e., data flow is from source to destination) is FC\_PWM\_DRV, and for convenience terminal variable \texttt{mfunc\_c1} was chosen.

![Figure 9. Connecting Modules Together Within a System Framework](image-url)
6.2 C Considerations and Specifics

As previously explained, the C module library that is offered as part of the TI DCS C2000 Foundation software is based on the CcA approach. In addition to providing a useful set of C functions, these modules provide the additional benefit of allowing software designers to quickly and efficiently build up eXpressDSP™-compliant algorithms. (Refer to section 8 for more details on the algorithm standard). Although not eXpressDSP-compliant on their own, these modules adhere to most of the rules and all of the good programming practices within the algorithm standard, but do not have the IALG layer implemented. These CcA modules are regarded as adhering to the algorithm standard, meaning that these modules support, at a minimum, re-entrancy and multiple instantiation. To achieve this, a stack-based pointer passing strategy is used. Therefore, instead of passing numerous variables, wasteful on CPU cycles, a pointer to an array or structure is used as the function argument instead. This approach is more efficient and easily supports functions which need to maintain data history like filters, estimators, etc., and allows functions to be instantiated multiple times each time with different data sets.

On closer examination, CcA Modules need to cover the various operating modes or situations, shown in Table 5. All of these situations can be handled easily and efficiently by passing pointers to structures via the stack.

Table 5. Module Operating Situations Supported by CcA Library

<table>
<thead>
<tr>
<th>Module Configuration (or situation)</th>
<th>Config Data or History Maintained?</th>
<th>Function Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single input/single output</td>
<td>No</td>
<td>Trigonometric, sqrt, etc.</td>
</tr>
<tr>
<td>Single input/single output</td>
<td>Yes</td>
<td>Filters</td>
</tr>
<tr>
<td>Single input/multiple outputs</td>
<td>Yes</td>
<td>FFTs</td>
</tr>
<tr>
<td>Multiple input/single output</td>
<td>No</td>
<td>Transforms</td>
</tr>
<tr>
<td>Multiple input/single outputs</td>
<td>Yes</td>
<td>Wavegen</td>
</tr>
<tr>
<td>Multiple input/multiple outputs</td>
<td>No</td>
<td>Matrix operations, transforms</td>
</tr>
<tr>
<td>Multiple input/multiple outputs</td>
<td>Yes</td>
<td>State space filters/estimators</td>
</tr>
</tbody>
</table>

6.2.1 Module Declaration, Instantiation and Usage

Figure 10 shows an example of the CLARK transform module, this has multiple inputs and multiple outputs with no history. The function arguments are 2 pointers to structures, one for inputs and one for outputs.

Figure 10. C-callable Assembly (CcA) Module Example With Multiple Input/Multiple Outputs/History

eXpressDSP is a trademark of Texas Instruments.
Figure 11 shows an example of the SVGEN_MF module. This module has multiple inputs and outputs, and needs to have data history maintained. It operates on the structure type pointed to by &sv1. Here the variables alpha and sector maintain history, while freq_max is a one-time init or config parameter. The function call example shows two instantiations of the same module operating on different and independent data structures.

![SVGEN_MF Module Example](image)

**Figure 11. CcA Module Example With Multiple Input/Multiple Outputs/History**

### 6.2.2 Connecting Modules Together

As per the assembly case, the same philosophy applies when connecting CcA modules together. However, in this case, a fully C framework provides all the added flexibility and benefits expected from a C environment. As will be seen later in section 8, once a full system is realized using this approach, it can be readily encapsulated to form an eXpressDSP-compliant algorithm with minimal extra effort.

![Connecting Modules Together](image)

**Figure 12. Connecting Modules Together**
7 Incremental System Builds Reduce Development Time

Texas Instruments understands that, irrespective of how much planning has gone into a system engineering project, chances are the final (or complete) system will not work the first time, usually some subtle (or not) target dependency or requirement has been overlooked. This is normal engineering, especially in the case of software development, and if not anticipated ahead of time, can become a frustrating and time consuming disaster.

To prevent this, TI’s DCS foundation system software comes prepackaged within a framework (ASM or C), incorporating various system build levels or debug steps as part of the normal system software flow. A customer can move from a current build level to any other build level very quickly and without risk of losing configuration or run-time information related to previous build levels. This allows a customer to commission system software in a step-by-step manner, and at each step validate key vital signs before moving on. The number of build levels may vary depending on final system complexity and the type of control strategy used. It is important to note that regardless of the number of build levels used in the reference system, this framework is provided by TI as a guidance and becomes a common point of reference in cases where customer trouble shooting necessitates interaction with TI customer support engineers or hotline.

Incremental system build (ISB) levels are supported by frameworks with various configurations of interconnected modules. Frameworks also provide invaluable skeletons for customers to modify or customize as required to suit their own target system requirements, forming an excellent starting point.

7.1 Stepping Through a Typical Incremental Build Process

To better understand the philosophy and details of the incremental build approach, a typical motor control case, a permanent magnet synchronous motor is used to step through the various commissioning phases. The system shown in Figure 13 is the final build of PMSM3–1, as referred to earlier in Table 2. The subsequent block diagrams (Figure 14 to Figure 18) show examples of typical steps used to get to a solidly working final system.
7.1.1 Check System Vital Signs – Build Level 1

Figure 14 shows the module configuration of build level 1. Although the first is the simplest build, it is perhaps the most important one since many system fundamentals can be validated here. Build 1 uses only target independent modules whereby removing all external (peripherals, power hardware, motor, feedback circuitry, etc.) influences and allows a customer to focus on:

- Code compiling/assembling/linking using a Code Composer (CC) project
- Invoking and downloading to the CC debug environment
- Setting up or importing a CC workspace
- Running code in real-time mode
- Ensuring the C24x EVM (or customer’s own target) is functioning correctly

Assuming the above check list is ok, some key system vital signs can be validated. PMSM3–1, like most other DMC systems, is an interrupt driven, time sampled system, meaning that the modules shown in Figure 14 are executed on every interrupt tick. Validating both the stimulus waveforms (output of RAMP_GEN) and output of SVGEN via the DAC_VIEW utility confirms system interrupts are being generated and the main ISR is executing correctly. At this stage, the RAMP_GEN function can be manipulated via a CC watch window to change its frequency and to see the corresponding changes on the waveforms.
As was explained in section 5.3.3, the DAC_VIEW utility module can be used to probe any global Q15 variable. Inputs to DAC_VIEW are pointers, and consequently can be pointed to any variable in real time, using the CC watch window facility while running in real-time mode. Pointing (or probing) to a variable is simply a matter of typing the symbolic name of the variable in question in the watch window which is opened and displaying the variables DAC_iptr0,1,2 and 3. For example, to check if the PARKI (inverse Park) transform is correctly generating quadrature sinewaves at its outputs, change the contents of memory locations DAC_iptr0 and DAC_iptr1, to the address values of Ipark_d and Ipark_q respectively. The symbolic name will be converted to an address by CC.

Systems supplied as part of the DCS Foundation software offering are configured to operate in build level 1 as default. To change build levels is simple, and can be done at the system framework in either an ASM- or C-based system. The following software excerpt is an example of how the build level is changed in an ASM based system:

```
;***********************************************
; Select Incremental Build of Main Control Loop
;***********************************************
I_build1 .set 0 ; Code framework and vital signs
I_build2 .set 0 ; forward control path and PWM generation
I_build3 .set 0 ; Encoder i/f driver, calibration and speed meas. validation
I_build4 .set 0 ; Current sensing and feedback path, Voltage mode FOC operation
I_build5 .set 1 ; Current mode FOC and current regulators.
F_build1 .set 0 ; Speed calculator FOC and speed loop - Final system build.
```
7.1.2 Check PWM Generation at the Target Hardware – Build Level 2

Build 1 has confirmed vital signs are ok and the space vector generator waveforms at the outputs of Ta, Tb, and Tc show the characteristic three phase space vector (SV) shape. Also frequency can be adjusted and can be set to an appropriate value in preparation for connection of power hardware and motor. Build 2, shown in Figure 15, now introduces the PWM driver. At this point, it is important to verify that the space vector signals (Ta, Tb, and Tc) are correctly modulating the PWM outputs. This is easily checked by filtering or suppressing the high frequency carrier (typically 10–20KHz) at the PWM output pins by a simple RC filter with approximately 1kHz cutoff. The filtered waveforms should then resemble the unmodulated ones seen at variables Ta, Tb, and Tc using the DAC-VIEW module.

We now have a working forward path for the PMSM controller running in open loop, and are ready to connect the power inverter hardware and motor.

![Watch window](image)

Figure 15. Example of Build Level 2 – Checking PWM Driver Operation

7.1.3 Check Power Inverter Hardware and Open Loop Motor Operation – Build Level 3

Build level 3, shown in Figure 16, now allows us to run the PMSM open loop. Build 3 also implements the angular position sensor feedback path, based on the QEP_THETA_DRV driver module which outputs both the mechanical and electrical angles. The optical position sensor and QEP interface driver are operating correctly when the motor is spinning, variable theta_elec can be viewed on the scope. Its output waveform should correspond in shape to the stimulus theta angle generated by the RAMP_GEN module. This indicates that position information is being measured correctly.

The speed measurement module, SPEED_FRQ, is in the feedback path, and can also be validated at this stage. The speed measurement is dependent on motor shaft rotation, and the SPEED_FRQ module is driven from the theta_mech output of QEP_THETA_DRV. Since its output value should be constant, i.e., the motor is running at constant synchronous speed, the calculated speed, speed_frq, can be viewed via a watch window instead of on the scope. The RAMP_GEN frequency can be increased slowly up or down to check whether the speed measurement varies appropriately.
7.1.4 Closed Loop Motor Operation Under Voltage Control – Build Level 4

It was confirmed in build level 3 that position information (both electrical and mechanical) was correctly appearing at the outputs of the QEP_THETA_DRV module. In build level 4, the simulated angle stimulus provided by RAMP_GEN is no longer needed. This can now be replaced by the actual measured electrical angle, \( \theta_{elec} \), which is used as angular position feedback for the inverse Park transform, PARKI. Although not yet utilizing closed loop current feedback, build level 4 allows the system to run in a "quasi FOC" (field oriented control) voltage controlled mode. The motor speed can be controlled by changing input, \( Ipark_D \) of the PARKI module using a watch window. Unlike build 3, speed was varied by modifying the synchronous frequency applied to the stator. Here we are changing the voltage magnitude applied to the stator and the resulting speed depends on the balance between shaft load and the applied voltage magnitude.

With the motor running in a stable state, it is possible to validate the current measurement feedback path. The peripheral driver ILEG2_DRV measures 2 inverter leg currents using ADC inputs and reconstructs the motor phase currents. The 120° phase currents are then transformed to 90° quadrature currents by the CLARK transform module. Using DAC_VIEW, the shape, phase and quality of the current waveforms can be inspected. The scope waveforms in Figure 17 show both the expected phase and quadrature currents. If appropriate, gain and offset adjustments can be made to the current measurements using the ILEG2_DRV module. This can be performed easily in real time via a watch window. The module document for ILEG2_DRV gives details on which configuration variables are required. Once the current feedback path has been tuned, the temporary changes made via the watch window can be made permanent in the configuration or initialization section of the system framework code.
7.1.5 **Closed Loop Current Control, Torque Mode Operation – Build Level 5**

Build level 4 allowed the open loop current feedback path to be tuned. Build level 5 (shown in Figure 18) introduces PID regulators and closes both the direct and quadrature (i.e., D and Q) current loops. The topology of build level 5 allows true FOC operation, however the PID regulators may need tuning to ensure the system is running optimally. The proportional, integral and derivative constants were initially calculated theoretically, but can now be tuned on-line while the motor is spinning under load. A watch window showing $K_p$, $K_i$ and $K_d$ can be very useful for this purpose. In addition, the tuning process the 120° phase currents and the rotating reference frame currents ($park_D$, $park_Q$) can be viewed in real time, allowing response to changes to be monitored immediately and visually.

Build level 5 operates in a torque control mode, since torque is proportional to current, maintaining a constant value for $park_D$ using the PID regulator forces the system to keep a constant motor torque. The torque setting can be changed by setting the PID controller reference input, $ref$, via a watch window. If the requirement of the system is to operate only in a torque control mode, then build level 5 can be the final system build. If closed loop speed control is desired, motor torque is continuously adjusted by the PID controller. To keep a constant speed under changing load conditions, the final system build becomes that shown in Figure 13.
Figure 18. Example of Build Level 5 – Closed-loop Current Control, Torque-mode Operation

8 Software Standardization – TMS320 DSP Algorithm Standard Is the Ideal Vehicle

The proceeding sections of this chapter will further elaborate on the Algorithm Standard within the context of how it applies to the C2000 (namely x24x and x24xx) platform. However, the reader is encouraged to reference the official TMS320 DSP Algorithm Standard documents for full coverage on this topic. The main documents of interest are:

- TMS320 DSP Algorithm Standard Rules and Guidelines (SPRU352)
- TMS320 DSP Algorithm Standard API Reference (SPRU360)
- The TMS320 DSP Algorithm Standard (SPRA581)
8.1 eXpressDSP Compliancy

In order for an algorithm to be eXpressDSP-compliant, the algorithm must implement the IALG interface specified by the standard. The IALG interface is an abstract interface or a Service Provider Interface (SPI) and is defined in the ialg.h header file. In addition to implementing the IALG interface, the algorithm must obey a number of rules. For example, external identifiers need to follow the naming conventions, algorithms must never directly access any peripheral device, and the algorithm code must be fully relocatable. Table 6 shows a summary of the Rules, and Table 7 shows the guidelines. These are shown for convenience only. For more details, please see the official TMS320 DSP Algorithm Standard documents referred to earlier.

The algorithm must implement an algorithm-specific interface, defined by an algorithm interface creator, that extends the IALG interface in order to run the algorithm. The example included within this application report is based on the simple VHz control algorithm for a 3-phase ACI motor. This control scheme is referred to in Table 2 as system ACI3–1. The example outlines how to take a set of algorithm standardized modules from the C-module library and wrap them with the IALG interface layer, thereby generating an eXpressDSP-compliant algorithm. Figure 19 shows the original ACI3–1 system and which modules are wrapped by the IALG layer.

The algorithm’s client can manage an instance of the algorithm by calling into a table of function pointers (v-table). Every algorithm must create the v-table to be eXpressDSP-compliant. Through the v-table, the application can manage the algorithm instance (e.g., create and delete an instance object of the algorithm as well as run the algorithm).

Table 6. TMS320 DSP Algorithm Standard Summary of Rules

<table>
<thead>
<tr>
<th>I. Rules for Good Programming Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Modules (algorithms) must follow the run-time conventions imposed by TI’s C compiler.</td>
</tr>
<tr>
<td>• Modules must be re-entrant within a preemptive environment (and invocable multiple times).</td>
</tr>
<tr>
<td>• All module data references must be fully relocatable (no hard-coded data locations).</td>
</tr>
<tr>
<td>• All module code must be fully relocatable (no hard-coded program locations).</td>
</tr>
<tr>
<td>• Modules must NEVER directly access any peripheral devices, i.e. I/O, EV, SPI, SCI, ADC, etc.</td>
</tr>
<tr>
<td>• Modules must supply an initialization and finalization method.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>II. eXpressDSP-compliant Specific Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Modules must implement the IALG interface.</td>
</tr>
<tr>
<td>• Modules must follow the naming conventions of the DSP/BIOS.</td>
</tr>
<tr>
<td>• All undefined references must refer to either the C run-time library or DSP/BIOS operations.</td>
</tr>
<tr>
<td>• Each module must be packaged in an archive having a name that follows a uniform naming convention.</td>
</tr>
<tr>
<td>• Each module header must follow a uniform naming convention.</td>
</tr>
<tr>
<td>• Different versions of a module from the same vendor must follow a uniform naming convention.</td>
</tr>
<tr>
<td>• If a module’s header includes definitions specific to a debug variant, it must use the symbol __DEBUG</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>III. Rules for Performance Characterization/Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Modules must characterize their worst-case heap data memory requirement.</td>
</tr>
<tr>
<td>• Modules must characterize their worst-case stack space memory requirement.</td>
</tr>
<tr>
<td>• Modules must characterize their static data memory requirement.</td>
</tr>
<tr>
<td>• Modules must characterize their program memory requirement.</td>
</tr>
<tr>
<td>• Modules must characterize their worst-case Interrupt latency impact.</td>
</tr>
<tr>
<td>• Modules must characterize their worst-case and typical execution time.</td>
</tr>
</tbody>
</table>
Table 7. TMS320 DSP Algorithm Standard Summary of Guidelines

**Recommended Guidelines/Good Programming Practices**

- Modules should keep stack size requirements to a minimum.
- Modules should minimize their static memory requirements.
- Modules should never have any scratch static memory.
- Interrupt latency should be kept to a minimum (< 10uS).
- Modules should avoid the use of global registers.
- Modules should minimize their persistent data memory requirements in favor of scratch memory.
- Each initialization and finalization function should be defined in a separate object module.
- Modules should implement the trace interface ITRC.

8.2 Example of a DMC Algorithm With an IALG Abstract Interface

![Diagram of a DMC Algorithm](image)

**Figure 19. System ACI3–1 Implemented With an IALG Layer**
An algorithm vendor usually implements a module as a set of data objects and code functions operating on that collection of data objects. In most cases, it is useful for the algorithm to implement functions to access the data objects in a well-defined manner. This could be with data abstraction in mind, or just to prevent undesired interactions with other system elements. The idea of an abstract interface can be graphically represented as shown in Figure 20.

![Figure 20. Abstract Interfaces and Object Encapsulation](image)

Figure 19 shows an example of a system encapsulation within an IALG interface. Figure 20 shows the access protocol for any client application accessing the object implementation through the abstract interface. An IALG is an abstract interface implementing basic interface functions. The intent is for module interfaces to derive from (i.e., extend) this by adding functionality relevant to the module. The basic IALG functions are a source for performing system integration tasks, e.g., getting memory allocation information (algAlloc), activating the object (algActivate), and initializing the object (algInit). They are required functions and any IALG interface must implement these. The other functions are optional, and the decision whether to implement the functionality defined for these functions is left to the algorithm implementer. The adoption of the IALG interface and the Algorithm Standard standardizes the interface and enhances interoperability.

For any object to be eXpressDSP-compliant, the object must implement the IALG interface as described below:

```c
typedef struct IALG_Obj {
    struct IALG_Fxns *fxns;
} IALG_Obj;

typedef struct IALG_Fxns {
    Void *implementationId;
    Void (*algActivate)(IALG_Handle);
    Int (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);
    Int (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
    Void (*algDeactivate)(IALG_Handle);
    Int (*algFree)(IALG_Handle, IALG_MemRec *);
    Int (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
    Void (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
    Int (*algNumAlloc)(Void);
} IALG_Fxns;
```
The TMS320 DSP standard requires that IALG_Obj be the first member of the module object implementing an eXpressDSP-compliant interface. The *fxns in the IALG_Obj is initialized by the creation functions to the instance of IALG functions, or the v-table containing the IALG functions and any extensions. Note that the IALG interface is generic (i.e., it only implements system integration functions). This is a base object. Any object that extends the IALG must add functionality relevant to this object. This can be done as indicated below:

/* First create an object that implements IALG_Object as the first member of the structure */

typedef struct VHZ_TI_Obj {
  IALG_Obj alg;         /* MUST be first field of all IVHZ objs */
  int speed_setpt;
  int speed_value;
  int direction;
  int closed_loop_flag;
  VHZPROFILE vhzprof;   /* xDAIS-ready Volts/Hertz Profile module */
  SVGENMF svgen;        /* xDAIS ready Space Vector Waveform Generation Module */
  PID pid;              /* xDAIS ready Space Vector Waveform Generation Module */
} VHZ_TI_Obj;

/* Second, build a V-table that implements the IALG functions and the extensions (or additions) */

typedef struct IVHZ_Fxns {
  IALG_Fxns ialg;    /* IVHZ extends IALG */
  void (*setSpeed)(IVHZ_Handle handle, int speed);
  int (*getSpeed)(IVHZ_Handle handle);
  void (*run)(IVHZ_Handle handle );
  void (*shutdown)(IVHZ_Handle handle );
  void (*reportSpeed)(IVHZ_Handle handle, int speed);
  void (*getMotorCmds)(IVHZ_Handle handle, void *);
} IVHZ_Fxns;

In the creation of the object, the algorithm creation code binds the v-table to the object instance.

    VHZ_Handle hVhz;
    ...
    hVhz=VHZ_create(VHZ_TI_IVHZ, &VHZ_PARAMS);

VHZ_create performs among other things the following operation:

    /* Since the ALG_Create uses an internal handle which has the type
       ALG_Handle, the pointer is accessed as hCreatedObj->fxns */
    hCreatedObj->fxns=(struct IVHZ_Fxns *)fxns;

The symbol VHZ_TI_IVHZ is a valid, correctly initialized v-table made available by the algorithm implementation. This symbol, which follows the naming convention imposed by the Algorithm Standard, bears the following information. VHZ supplies an algorithm name, from the TI vendor, and IVHZ signifies that this is an interface being implemented by the algorithm. Another interface which the module can implement would be for example VHZ_TI_XXX, which could implement functions in a custom interface.
The module definition in VHZ_TI_Obj may not be made available to the client application. The only requirement is that the algorithm implement enough methods in the interface(s) to perform all the functions necessary for the component. By distributing only object code, the algorithm provides the client application implementation a means to instance algorithm objects and performs all algorithm functions, without revealing the internal details of the algorithms.

8.3 Scope of TMS320 DSP Standard Algorithms Within DMC

The IALG interface layer can be implemented at almost any level. At the small end of the scale, a math function like a Park or Clarke transform can be wrapped. On the larger scale, an entire system, minus peripheral drivers, can also be wrapped. There are no hard rules in this regard; however, two significant factors may influence this decision:

- Does the algorithm have significant intellectual property (IP) or complexity to be valuably traded or exchanged?
- Is the overhead that IALG introduces a significant part of the unwrapped algorithm?

Let’s consider each factor in more detail:

**Algorithm IP Value or Complexity**

If an algorithm is very complex and/or has valuable IP associated with it, then it becomes a good candidate for an IALG interface, especially if it is widely distributed (inter-company) or sold as in the case of a third-party consultant or software house. In both cases, a common or standard interface takes out the guesswork and allows for quick system integration. In the case of valuable IP, it is possible to trade the standard algorithm at the Object file level, while keeping the source code secure. The standard interface also applies here, but users of this algorithm need not know about the detailed internals of the source code. The key is in the interface, and the algorithm can be used like a black box.

For DMC, both cases may apply. Figure 21 shows a typical sensorless control PMSM system. The PMSM speed controller layer may be an appropriate partitioning point in this system due to its complexity and reusability. Note that by definition, the driver modules must be excluded from algorithm encapsulation. Alternatively, a single module like the position estimator may also warrant a separate IALG interface.
IALG Overhead

Although not a huge amount of overhead, IALG does impose some overhead, and this needs to be balanced against the algorithm itself. Generally, for a typical DMC system algorithm, IALG introduces of the order of 100-200 words of program and 20-50 words of data as overhead. For small algorithms like park, clark, svgen_dq, etc., which have approximately 100-150 program words and 10-20 data words, this becomes unnecessary overhead. It becomes obvious that the algorithm standard makes sense only at the complete system level such as ACI3–2, PMSM3–2, etc., where the code has at least 2,000 to 3,000 program words and 30-100 data words. In conclusion, for a moderately-sized DMC system, the standard IALG interface introduces approximately 5-10 percent overhead. In terms of execution overhead, IALG does not impose significantly more overhead as compared to a well structured algorithm.

9 Packaged Solution – Code Composer and Documents Complete the Package

Although a standardized library of software modules is very useful in helping to build a custom system, it is often more valuable to have a typical or reference system which works and utilizes these modules. This can accelerate a customer’s understanding of the modules and how they go together to form a system. It can significantly reduce a customer’s design cycle time. Systems can be quite complex, so it is important to have a comprehensive and easy-to-use environment. As part of the DCS Foundation software offering this environment is delivered through the Code Composer Studio™ integrated development environment (IDE) in conjunction with good System and Module documentation. The Web downloadable DCS foundation software consists of two types of bundles or packages:

Code Composer Studio is a trademark of Texas Instruments.
9.1 DCS Module Library Package

The DCS module library is a collection of software functions which are considered useful in the implementation of Digital control systems. Table 2 shows a sample set of such modules. Logistically a library module consists of multiple components, at a minimum, a module document, and source code given in assembly (ASM). Additionally, to support the TMS320 DSP Algorithm Standard effort, the bulk of modules are also offered in CcA and are designated algorithm-standard ready. The module library is invaluable for customers wanting out-of-the-box functions, while concentrating on their own customized system solution.

9.2 System Package

A system package consists of everything a customer needs to implement a working system. It is a self-contained package with all relevant documents and software, both system and module. The system package also contains the relevant Code Composer Project and workspace files required to easily compile, load and run the application “out of the box” on any of the F24x or LF24xx EVM boards. Figure 23 shows a typical Code Composer workspace, used to debug a DMC system. Note that all relevant watch windows are configured with variables of interest, and graphs are set up to show both the sinusoidal phase-to-phase voltage and space vector phase-to-neutral voltage. For clarity, Table 8 describes a typical check list of items which are included in a downloaded system package.
Table 8. Checklist for a Typical System Bundle

<table>
<thead>
<tr>
<th>No.</th>
<th>Item</th>
<th>Description</th>
<th>Offering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System software or framework with ISB. (asm based)</td>
<td>System or infrastructure software, which provides the required framework for all modules to work within. It is responsible for system related resources like Interrupts, DSP initialization, etc. Equally important it provides a ready-made incremental build infrastructure, which helps in system commissioning, (see section 7).</td>
<td>Standard</td>
</tr>
<tr>
<td>2</td>
<td>System software or framework with ISB. (C-based)</td>
<td>Same as no. 1, and if available at the time, offered as a C system</td>
<td>Optional</td>
</tr>
<tr>
<td>3</td>
<td>eXpressDSP-compliant IALG wrapper</td>
<td>eXpressDSP-compliant algorithm wrapper or software layer which provides a standardized API.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
| 4   | System document                                                     | • Document explaining system-level related items, such as:  
  • Compiling, loading, and running the application.  
  • Specifics for ASM case and C-system case.  
  • Modules used and how they are connected together  
  • Stepping through the various build or commissioning stages  
  • What to expect (i.e., waveforms, values, etc.) for each build  
  • Setting up the recommended power invertor hardware | Standard |
| 5   | Excel spreadsheet                                                    | Some systems are quite complex requiring initialization and/or configuration of many system variables based on target (e.g., motor) parameters. These calculations typically are based on real-world values, with results needing to be in the appropriate hex Q-format. To help customers through this process, an Excel spreadsheet is provided. | Optional |
| 6   | Code Composer project                                               | The entire system software (system framework plus modules) can be conveniently managed by a Code Composer project. The Code Composer project offered with each system provides the “make” function and keeps track of compile and link dependencies between main files and library objects. | Standard |
| 7   | Code Composer workspace                                             | A CC workspace is a convenient way to customize the Code Composer environment for a specific application or debug session. The workspace provided with each system sets up the Code Composer graphic user interface (GUI) to allow the customer easy interaction with the specific control system at hand. Watch windows with key variables, graph windows, CPU registers, and relevant memory blocks are all auto-arranged after a workspace is loaded. | Standard |
| 8   | Module software (asm-based)                                         | All appropriate software modules making up a particular system are provided. These modules are in full C2xx assembly source.                                                                                   | Standard |
| 9   | Module software (C-based)                                           | Same as no. 8, and if available at the time, offered as CcA modules (i.e., C-callable assembly source). These modules are eXpressDSP-compliant and ready. (see section 8).                                      | Optional |
### Table 8. Checklist for a Typical System Bundle (Continued)

<table>
<thead>
<tr>
<th>No.</th>
<th>Item</th>
<th>Description</th>
<th>Offering</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Module documents</td>
<td>Documents explaining Module related items, such as:</td>
<td>Standard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• How to instantiate and use a module within a system framework</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Theory and math related to the module’s operation or function</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specifics for ASM case and C case</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Readme file</td>
<td>A very useful text file reflecting any recent changes or upgrades a customer should know about when installing and running a system reference. Information such as:</td>
<td>Standard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Drivers for XDS510pp, XDS510pp-plus emulators</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Naming conventions for files and directories</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Code composer upgrades or changes</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 23. A Typical Code Composer Workspace Supplied as Part of the System Package**

Although the downloadable bundle consists of source software and documentation, all reference systems have been configured to run on a standard hardware platform. The default platform recommendation is:

**DSP target:** TMS320F243 or TMS320LF2407 EVMs

**Power inverter target:** DMC1500 or DMC1000 (from Spectrum Digital)
Each system document gives a summary of the hardware configuration (jumper settings) required to successfully run the system. Standardizing on a given platform combination provides customers with a working benchmark or reference. For a modest investment in the recommended platform, all the guesswork is avoided, and customers have a common point of reference during the various system commissioning stages (build levels). If problems or issues still persist, this common point of reference allows for a more intelligent or structured dialog between customers and TI during hotline or direct interaction sessions. Figure 24 shows a typical bench setup with an EVM and a DMC1000 Power inverter. This setup is quite flexible and can be debugged in real-time mode using the XDS510pp emulator. For convenience and safety, a variac has been used to gradually raise the DC bus voltage during initial system checkout. DMC1x00 boards can support a fully variable DC bus (down to zero volts) by using an optional 18v input power supply, ensuring that the IGBT driver chips are active, even with zero DC bus voltage.

Although a standardized platform is recommended for initial prototyping or evaluation, customers are not discouraged in developing and using their own custom hardware directly. In fact, the entire basis for the modular approach is to provide a smooth methodology for customers to quickly port the reference software to customized hardware. TI’s modular philosophy, which clearly separates modules into CPU and peripheral-dependent (drivers) categories, greatly simplifies the porting process. A customer can confidently port all peripheral-independent (i.e., target-independent) modules almost seamlessly, while focusing efforts on the few drivers which need custom configuration.

Figure 24. Typical Hardware Setup for Running a Reference System Package
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2002, Texas Instruments Incorporated