TI Standard for Writing Algorithms

ELG6163 Report

Geoffrey Green
Carleton #100350275
geoffgreen@ieee.org
March 17, 2006
For: Dr. Miodrag Bolic

Abstract

As the complexity of DSP systems increases and the required time-to-market window shrinks, the task of systems integration is becoming more and more important. It is less common for all algorithms to be developed "in-house" as designers choose instead to use third party algorithms in their designs. In order to ensure that components from multiple vendors work together and concurrently in the overall system, a standard interface is required for DSP algorithms. Texas Instruments (TI) launched the eXpressDSP Algorithm Standard Interface (XDAIS) in 1999 to fill this need. This report provides the reader with an introduction to XDAIS – it explains the motivation for this specification, and provides a high-level description of the standard's rules and guidelines. It also describes the design tools provided by TI to assist in the development of compliant algorithms, and concludes with an overview of the compliance process.

1.0 Introduction and Motivation for DSP Algorithm Standards

In the product development or research environment in which DSP engineers find themselves today, they are faced with several challenges that must be overcome to ensure success in their endeavours. In commercial systems (especially consumer electronics) that have a limited product lifetime, reducing the time-to-market is crucial. Increased complexity in DSP systems is a fact of life which manifests itself in many ways. Among these include evolving communications standards, more sophisticated algorithms emerging from R&D labs, the need to support more interfaces, the requirement to increase throughput speed, the trend towards increased product "density" (*e.g.* cell phones that also act as cameras, organizers and MP3 players), and the presence of multi-processor systems.

In this environment, it has become virtually impossible for engineers to implement completely "home-grown" DSP systems while still maintaining market relevance. Increasingly, it is necessary for design teams to outsource various portions of an overall implementation, allowing the group to focus on the development of core functionalities that differentiate their products. As an example, consider a small startup company that develops thermal cameras based on a certain technology for which they hold a patent. The expertise in this company is best directed towards endeavours related to thermal imaging, for example, increasing temperature resolution. Though important for their success in the marketplace, writing JPEG encoders and USB interface code is probably not a wise use of this company's limited resources. These necessary components can be acquired externally (be it in the form of other vendors' IP products, or

open source offerings) and then incorporated into the final design. These "canned" routines are known as commercial off-the-shelf (COTS) software [1].

Clearly, the role of systems integration becomes very important to a DSP design team in this setting. A systems integrator is responsible for interfacing all of the various software components (from both in-house teams and third-party sources) and making sure that they function properly in the overall design. Without standards to govern interoperability between components, the systems integrator must spend significant time and effort writing "glueware" to ensure that various COTS components from different sources (each with a proprietary interface in possibly C or assembler) work together. From a company's perspective, that is likely no more desirable than having that person writing the outsourced algorithm's functionality in the first place.

The preceding discussion motivates the use of a *standard* interface for DSP algorithms – a common set of rules and conventions that these algorithms must follow in order to "plug in" to a larger system. Ensuring that the standards are adhered to is done with an accompanying compliance process. The benefits of this standards framework are obvious [2]:

- for the systems integrator, the task becomes the simpler one of incorporating an algorithm that is written to that standard into the overall design, confident in the knowledge that it will function appropriately.
- for the algorithm developer, incorporating a standardized interface into the delivered algorithm ensures that they can market their code to the maximum number of clients without having to support multiple interfaces
- decreased time-to-market, as resources are not tied up writing "glueware"
- algorithm quality improvement (as deployment increases, creators will receive feedback from users)
- free up skilled resources for algorithm development/creativity/innovation
- simplifies the evaluation of algorithms from multiple vendors with the ability to quickly plug out and plug in various algorithms to compare (size, speed, etc.) or upgrade components
- instils confidence in the product since algorithm has been verified by independent compliance process

2.0 Texas Instruments XDAIS Algorithm Standard

2.1 History and Related Components

Recognizing these advantages and seeking to promote a rich industry of COTS DSP algorithms, TI launched its eXpressDSP Algorithm Interface Standard (XDAIS) in 1999 and has maintained it since then. XDAIS is just one of several

components of TI's eXpressDSP program which is designed to help DSP engineers work more effectively. Other key components of eXpressDSP include:

- Code Composer Studio, an integrated development environment
- DSP/BIOS, a real-time operating system kernel, and
- Reference Frameworks, basic DSP software structures ready for refinement by a specific application

Figure 1 clarifies how these components fit together – the XDAIS standard provides the specification for the "sockets" and "plugs" that help to incorporate an algorithm into the overall system framework [1].

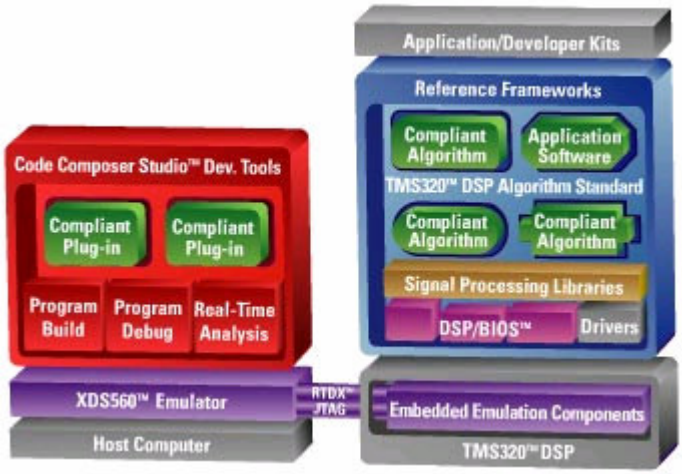

Figure 1 – Relationship between eXpressDSP components and XDAIS

An algorithm that meets the standard can be certified as eXpressDSP-compliant and are allowed to display the logo shown in Figure 2.

Figure 2 –eXpressDSP Compliant Logo

2.2 XDAIS Standard Details

2.2.1 Guiding Principles

The following were deemed to be the most important requirements during the development of the standard [2]:

- algorithms that originate from multiple vendors should be able to interoperate in the same system
- algorithms should be "framework-agnostic" – in other words, a given algorithm can be re-used in different applications
- algorithm use should be possible in both static and dynamic environments
- algorithms can be delivered in binary form, which safeguards the vendor's IP, and eliminates the user's need to re-compile
- integrating the algorithm into the system should not require recompilation of the client application (reconfiguration and relinking may be required)

The XDAIS standard rules and guidelines (described below) are a direct result of these requirements.

2.2.2 Omissions

The following are not included in the current version of the standard but are considered best practices and may be incorporated in the future [2]:

- version control for published algorithms
- licensing, encryption, and IP protection support (*e.g.* evaluation copies)
- installation and verification (*e.g.* digital signatures)
- documentation and online help (and installation into CCS)

2.2.3 Structure and Content

Note: It is impossible in a report of this length to provide details on each rule and guideline - the interested reader is referred to the standard itself for more information [2].

1. Rules and Guidelines (TI document SPRU352)

The XDAIS standard consists of a set of *rules* and a set of *guidelines*, each of which is defined at various algorithmic levels of abstraction (shown in Figure 3). A summary of these is provided in Appendix A. An algorithm must obey all of the rules in order to be eXpressDSP compliant. It is recommended that the algorithm also adhere to all guidelines but this is not required.

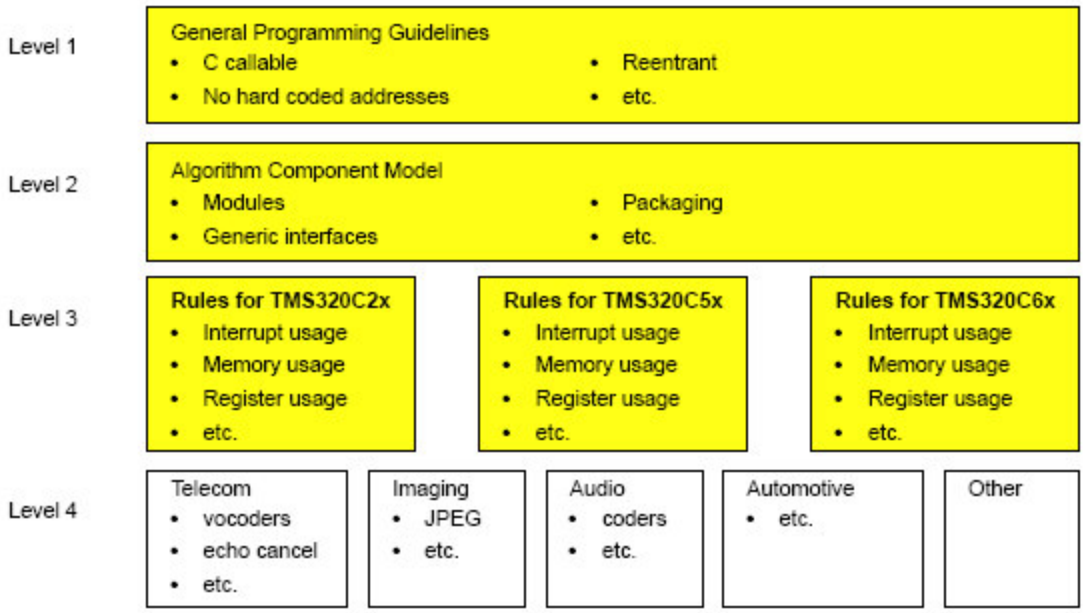

Figure 3 – Basis of XDAIS standard

The rules and guidelines at level 1 are very high level and cover all application areas. Level 2 specifies an abstraction layer between the algorithm itself and the processor's resources (such as memory, I/O, etc.). This ensures that multiple algorithms (or instances of the same algorithm) can co-exist on the DSP without "trampling" each other's resources. Level 3 deals with specific vendors' DSP families (e.g. TMS320C6000) - while consensus does not exist today, the guidelines in the standard at least provide a starting point. If a vendor deviates from this, it will be easy to draw attention to this in documentation. Level 4 is specific to application domains and is not covered in the XDAIS standard. A DSP algorithm that meets the rules defined in levels 1-3 is considered eXpressDSP-compliant.

2. Application Programming Interfaces (APIs) (TI document SPRU360)

Several rules and guidelines in the XDAIS standard specify the use of various APIs that should be used to wrap DSP algorithm code in an application, such as [3]:

- **IALG API** (algorithm instance interface) – responsible for creating an algorithm instance at run-time ensuring safe operation in any environment (static/dynamic or pre-emptive/non pre-emptive). The most critical role performed here is that of memory management. All memory references in an algorithm must be directed through this API – no explicit memory references should exist in the code. An eXpressDSP compliant algorithm is required to use this API.

- **IDMA2/ACPY2 APIs** (direct memory access (DMA) interface) – responsible for handling the DMA resource. Though not required, any eXpressDSP compliant algorithm that wants to use DMA must implement this interface, as well as the ACPY2 interface to request DMA services[1].
- **IRTC API** (real-time trace control interface)
- Supplementary APIs – these are user-supplied APIs that call the algorithm code (should be similar to TI demonstration applications) and are optional. They are intended to increase the usability of XDAIS algorithms in applications, and sit a layer above the required IALG API.

3.0 Tools for developing XDAIS compliant DSP code

TI provides several tools to assist in the implementation of eXpressDSP-compliant algorithms.

3.1 Documentation

Table 1 gives a list of the main TI documents related to the XDAIS standard.

TI document number	Title (*with comments*)
SPRU352	TMS320 DSP Algorithm Standard Rules and Guidelines
SPRU360	TMS320 DSP Algorithm Standard API Reference
SPRA581	White Paper – The TMS320 DSP Algorithm Standard (*general overview*)
SPRA810	A Consumer's Guide to Using eXpressDSP-Compliant Algorithms (*for algorithm users*)
SPRU424	TMS320 DSP Algorithm Development Guide (*for algorithm producers*)
SPRU361	TMS320 DSP Algorithm Standard Demo Application

Table 1 – XDAIS standard documentation

3.2 Developer's Kit

The XDAIS Developer's Kit is a software package written by TI that acts as a plug-in to Code Composer Studio IDE [4]. It is designed to facilitate the writing, integration and testing of XDAIS algorithms. It includes:

- Files required for XDAIS standard – these include header files, algorithm APIs and libraries, as well as example algorithms and applications.

---

[1] IDMA2 and ACPY2 APIs replace and deprecate the IDMA and ACPY interfaces that were defined in the earlier revisions of the T*MS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) and *TMS320 DSP Algorithm Standard API Reference* (SPRU360).

- eXpressDSP Component Wizard – helps the designer automatically generate generic code for an algorithm interface. The files are then modified for a specific application, a process in which the algorithm code itself is added. This is the TI recommended way of writing compliant code, since it allows designers focus on their code and not the XDAIS inner workings.
- QualiTI algorithm validation tool – enables DSP engineers to automate the testing of their algorithm.

3.3 Compliance Testing

The responsibility to ensure that the conditions for eXpressDSP compliance are met lies with the algorithm writer. The QualiTI tool (see above) is also used for XDAIS compliance self-testing. The writer, after successfully passing QualiTI testing and examining the detailed report that is generated, submits the test result to TI (along with other relevant information such as processor platform, application area, etc.). Upon review by TI, the algorithm is then certified as eXpressDSP compliant and qualifies to use the logo in Figure 2 [2].

It is interesting to note that at present, there exists a large choice of over 700 eXpressDSP compliant algorithms from roughly 100 third party participants [5].

4.0 Conclusion

This report presents an overview of the TI XDAIS DSP algorithm standard. The motivation and reasons for the standard's development is explained. While not presenting excessive detail on standard specifics, an overview of the key areas covered is presented, along with a discussion of the development tools provided by TI to assist in compliance.

5.0 References

[1] Texas Instruments, "White Paper – The TMS320 DSP Algorithm Standard", TI documentation number SPRA581, http://www.ti.com

[2] Texas Instruments, "TMS320 DSP Algorithm Standard Rules and Guidelines", TI documentation number SPRU352, http://www.ti.com

[3] Texas Instruments, "TMS320 DSP Algorithm Standard API Reference", TI documentation number SPRU360, http://www.ti.com

[4] https://www-a.ti.com/downloads/sds_support/targetcontent/XDAIS/index.html

[5] Texas Instruments, "Complete Listing of eXpressDSP-Compliant Third Party Algorithms", TI documentation number SPRM088, http://www.ti.com

[6] Texas Instruments, "A Consumer's Guide to Using eXpressDSP-Compliant Algorithms", TI documentation number SPRA810, http://www.ti.com

[7] Texas Instruments, "TMS320 DSP Algorithm Development Guide", TI documentation number SPRU424, http://www.ti.com

Appendix A

List of Rules and Guidelines in XDAIS Standard

General Rules

Rule 1 All algorithms must follow the run-time conventions imposed by TI's implementation of the C programming language.
Rule 2 All algorithms must be re-entrant within a pre-emptive environment (including time-sliced pre-emption).
Rule 3 All algorithm data references must be fully relocatable (subject to alignment requirements). That is, there must be no "hard coded" data memory locations.
Rule 4 All algorithm code must be fully relocatable. That is, there can be no hardcoded program memory locations.
Rule 5 Algorithms must characterize their ROM-ability; i.e., state whether they are ROM-able or not.
Rule 6 Algorithms must never directly access any peripheral device. This includes but is not limited to on-chip DMAs, timers, I/O devices, and cache control registers. Note, however, algorithms can utilize the DMA resource by implementing the IDMA2 interface.
Rule 7 All header files must support multiple inclusions within a single source file.
Rule 8 All external definitions must be either API identifiers or API and vendor prefixed.
Rule 9 All undefined references must refer either to the operations specified in Appendix B (a subset of C runtime support library functions and a subset of the DSP/BIOS HWI API functions) or TI's DSPLIB or IMGLIB functions, or other eXpressDSP-compliant modules.
Rule 10 All modules must follow the eXpressDSP-compliant naming conventions for those external declarations disclosed to the client.
Rule 11 All modules must supply an initialization and finalization method
Rule 12 All algorithms must implement the IALG interface.
Rule 13 Each of the IALG methods implemented by an algorithm must be independently relocatable.
Rule 14 All abstract algorithm interfaces must derive from the IALG interface.
Rule 15 Each eXpressDSP-compliant algorithm must be packaged in an archive which has a name that follows a uniform naming convention.
Rule 16 Each eXpressDSP-compliant algorithm header must follow a uniform naming convention.
Rule 17 Different versions of an eXpressDSP-compliant algorithm from the same vendor must follow a uniform naming convention.

Rule 18 If a module's header includes definitions specific to a "debug" variant, it must use the symbol _DEBUG to select the appropriate definitions; _DEBUG is defined for debug compilations and only for debug compilations.
Rule 25 All C6x algorithms must be supplied in little-endian format
Rule 26 All C6x algorithms must access all static and global data as far data.
Rule 27 C6x algorithms must never assume placement in on-chip program memory; i.e., they must properly operate with program memory operated in cache mode.
Rule 28 On processors that support large program model compilation, all function accesses to independently relocatable object modules must be far references. For example, intersection function references within algorithm and external function references to other eXpressDSP-compliant modules must be far on the C54x; i.e., the calling function must push both the XPC and the current PC.
Rule 29 On processors that support large program model compilation, all independently relocatable object module functions must be declared as far functions; for example, on the C54x, callers must push both the XPC and the current PC and the algorithm functions must perform a far return.
Rule 30 On processors that support an extended program address space (paged memory), the code size of any independently relocatable object module should never exceed the code space available on a page when overlays are enabled.
Rule 31 All C55x algorithms must document the content of the stack configuration register that they follow.
Rule 32 All C55x algorithms must access all static and global data as far data; also the algorithms should be instantiable in a large memory model.
Rule 33 C55x algorithms must never assume placement in on-chip program memory; i.e., they must properly operate with program memory operated in instruction cache mode.
Rule 34 All C55x algorithms that access data by B-bus must document: the instance number of the IALG_MemRec structure that is accessed by the B-bus (heapdata), and the data-section name that is accessed by the B-bus (static-data).
Rule 35 All TMX320C28x algorithms must access all static and global data as far data; also, the algorithm should be instantiable in a large memory model.

Performance Characterization Rules

Rule 19 All algorithms must characterize their worst-case heap data memory requirements (including alignment).
Rule 20 All algorithms must characterize their worst-case stack space memory requirements (including alignment).
Rule 21 Algorithms must characterize their static data memory requirements.
Rule 22 All algorithms must characterize their program memory requirements.

Rule 23 All algorithms must characterize their worst-case interrupt latency for every operation.
Rule 24 All algorithms must characterize the typical period and worst-case execution time for each operation.

DMA Rules

DMA Rule 1 All data transfer must be completed before return to caller.
DMA Rule 2 All algorithms using the DMA resource must implement the IDMA2 interface.
DMA Rule 3 Each of the IDMA2 methods implemented by an algorithm must be independently relocateable.
DMA Rule 4 All algorithms must state the maximum number of concurrent DMA transfers for each logical channel.
DMA Rule 5 All algorithms must characterize the average and maximum size of the data transfers per logical channel for each operation. Also, all algorithms must characterize the average and maximum frequency of data transfers per logical channel for each operation.
DMA Rule 6 C6000 algorithms must not issue any CPU read/writes to buffers in external memory that are involved in DMA transfers. This also applies to the input buffers passed to the algorithm through its algorithm interface.
DMA Rule 7 If a C6000 algorithm has implemented the IDMA2 interface, all input and output buffers residing in external memory and passed to this algorithm through its function calls, should be allocated on a cache line boundary and be a multiple of the cache line length in size. The application must also clean the cache entries for these buffers before passing them to the algorithm.
DMA Rule 8 For C6000 algorithms, all buffers residing in external memory involved in a DMA transfer should be allocated on a cache line boundary and be a multiple of the cache line length in size.
DMA Rule 9 C6000 Algorithms should not use stack allocated buffers as the source or destination of any DMA transfer.
DMA Rule 10 C55x algorithms must request all data buffers in external memory with 32-bit alignment and sizes in multiples of 4 (bytes).
DMA Rule 11 C55x algorithms must use the same data types, access modes and DMA transfer settings when reading from or writing to data stored in external memory, or in application-passed data buffers.

General Guidelines

Guideline 1 Algorithms should minimize their persistent data memory requirements in favour of scratch memory.
Guideline 2 Each initialization and finalization function should be defined in a separate object module; these modules must not contain any other code.

Guideline 3 All modules that support object creation should support design-time object creation.
Guideline 4 All modules that support object creation should support run-time object creation.
Guideline 5 Algorithms should keep stack size requirements to a minimum.
Guideline 6 Algorithms should minimize their static memory requirements.
Guideline 7 Algorithms should never have any scratch static memory.
Guideline 8 Algorithm code should be partitioned into distinct sections and each section should be characterized by the average number of instructions executed per input sample.
Guideline 9 Interrupt latency should never exceed 10 µs.
Guideline 10 Algorithms should avoid the use of global registers.
Guideline 11 Algorithms should avoid the use of the float data type.
Guideline 12 All C6x algorithms should be supplied in both little- and big-endian formats.
Guideline 13 On processors that support large program model compilations, a version of the algorithm should be supplied that accesses all core run-time support functions as near functions and all algorithms as far functions (mixed model).
Guideline 14 All C55x algorithms should not assume any specific stack configuration and should work under all the three stack modes.

DMA Guidelines

Guideline 1 The data transfer should complete before the CPU operations executing in parallel (DMA guideline).
Guideline 2 All algorithms should minimize channel (re)configuration overhead by requesting a dedicated logical DMA channel for each distinct type of DMA transfer it issues, and avoid calling ACPY2 configure and preferring the new fast configuration APIs where possible.
Guideline 3 To ensure correctness, All C6000 algorithms that implement IDMA2 need to be supplied with the internal memory they request from the client application using algAlloc().
Guideline 4 To facilitate high performance, C55x algorithms should request DMA transfers with source and destinations aligned on 32-bit byte addresses.
Guideline 5 C55x algorithms should minimize channel configuration overhead by requesting a separate logical channel for each different transfer type. They should also call ACPY2_configure when the source or destination addresses belong in a different type of memory (SARAM, DARAM, External) as compared with that of the most recent transfer.