SystemView
BY ELANIX

A Guide
To
AUTOMATIC
C-CODE GENERATION

EAGLEWARE
Electronic Design Automation Software
ELANIX
# TABLE OF CONTENTS

1.0 INTRODUCTION ............................................................................................................. 5  
   1.1 KEY FEATURES ........................................................................................................ 5  

2.0 INSTALLATION .............................................................................................................. 6  
   2.1 INSTALLATION ........................................................................................................ 6  
   2.2 SOFTWARE REQUIREMENTS .................................................................................. 6  
   2.3 HARDWARE REQUIREMENTS ................................................................................ 6  

3.0 C-CODE GENERATOR OVERVIEW ........................................................................... 7  
   3.1 CODE GENERATOR MODEL .................................................................................. 8  
   3.2 INVOKING C-CODE GENERATOR ....................................................................... 9  
   3.3 C-CODE GENERATOR PROCEDURE .................................................................. 9  
      3.3.1 STEP 1 – TARGET PROCESSOR SELECTION .............................................. 10  
      3.3.2 STEP 2 – SOURCE CODE SHELL ............................................................... 11  
      3.3.3 STEP 3 – INPUT BUFFER SIZE ................................................................. 12  
      3.3.4 STEP 4 – IN-LINE SOURCE SELECTION .................................................. 12  
      3.3.5 STEP 5 – DIRECTORY PATH ....................................................................... 13  
      3.4 CODE GENERATOR STATUS INFORMATION ............................................. 14  

4.0 AUTOMATIC C-CODE GENERATION EXAMPLE ................................................ 15  
   4.1 EXAMPLE SYSTEM ............................................................................................... 15  
   4.2 EXAMPLE C-CODE GENERATOR OPTIONS ..................................................... 15  
   4.3 C-CODE GENERATOR SOURCE CODE ................................................................ 17  

5.0 SYSTEMVIEW CUSTOM CODE EXAMPLE ........................................................... 19  
   5.1 CUSTOM CODE OVERVIEW ............................................................................... 19  
   5.2 CUSTOM CODE SOURCE GENERATION ........................................................... 19  
   5.3 CUSTOM CODE DLL CREATION ....................................................................... 19  
   5.4 CUSTOM CODE CO-SIMULATION ..................................................................... 20  
   5.5 REAL-TIME DSP ARCHITECT CO-SIMULATION ........................................... 21  

APPENDIX A – CONTROL PROGRAM SOURCE LISTING ........................................... 22  

Automatic C-Code Generation
1.0 Introduction

The SystemView Automatic C-Code Generator feature allows users to create ANSI C-code from SystemView designs for use in DSP code development, standard C-code applications, or for use within the SystemView environment. This document describes the key features and explains the use of the C-Code Generation tool.

SystemView provides options for the Code Generator tool in order to produce source code customized for a desired platform. The generated code is efficient, taking into account token parameter settings, and is highly readable with descriptive variable names and clear implementation.

The C-Code Generator can be used with the Real-Time DSP Architect (RTDA) to produce hardware-in-the-loop simulation with a Texas Instruments EVM or DSK board.

1.1 Key features

Key features of the C-Code generation software include:

- ANSI C-Code generation from specified SystemView tokens.

- User-specified platform description for generating target-specific source code.

- Optional SystemView interface for creating custom user code DLLs or for creating Real-time DSP architect tokens.

- Rapid transition from system design to DSP implementation.
2.0 Installation

2.1 Installation
The Automatic C-Code Generation software option is automatically installed with SystemView.

2.2 Software Requirements
The Automatic C-Code Generation software option is available with SystemView by Elanix, version 5.0. A Win32 C-compiler, such as Microsoft Visual C++, is required for creating SystemView custom code DLLs from the source code. Code Composer Studio version 1.19 or greater, available from Texas Instruments, is required for creating executable files for the SystemView Real-Time DSP Architect (RTDA) option. The RTDA feature requires the SystemView DSP library.

2.3 Hardware Requirements
To use the features of SystemView and the Automatic C-Code Generation software, your PC should be configured as a Pentium 400 MHz or faster, having a minimum of 128MB of RAM (256MB is recommended), with at least 100MB of hard disk space available.
3.0 C-Code Generator Overview

The fixed and floating point source code generation feature of SystemView allows a user to create ANSI C-code from a single token, a set of tokens, or a complete simulation system. The source code is fully functional, containing code for each of the SystemView tokens and can optionally contain a master control program that performs system scheduling, and handles data passing between tokens. The generated code is efficient, based on user parameter settings, and is highly readable with descriptive variable names and clear implementation.

The Code Generator can optionally create source code that serves as an interface for co-simulation, or for providing an interface for Texas Instruments eXpressDSP-compliant algorithm development. Co-simulation can be performed through a Windows DLL using a SystemView custom code token. Real-time hardware-in-the-loop simulation is available using SystemView’s Real-Time DSP Architect (RTDA) feature.

The Code Generator includes virtually all of the SystemView main library tokens, including System environment settings, and tokens from the Source, Operator and Function libraries. The tokens within the DSP and Communications libraries are also included. The RF and Logic libraries are not supported since they are designed to simulate hardware. The DVB, CDMA nor any of the third party libraries are currently supported. The DVB and CDMA are planned to be included with a future release. The third party library source code must be negotiated with the respective owners.
3.1 Code Generator Model

The SystemView simulation engine uses a dynamic, time-based model, which is described in the SystemView User Guide, see chapter 4 of this document. Each sample input or output from a token has an absolute time associated with it. The SystemView Code Generator uses the sample-based approach, where each functional unit, symbolized in SystemView as a token, operates only on a sample or group of samples received. The sample-based approach is well suited for implementation on a DSP or for generating reentrant functional blocks of code. The differences in the two approaches, however, may cause disparities between output from SystemView and output from generated source code.

The removal of absolute time in the generated code results in restrictions placed on token rates within a SystemView system when rate-sensitive tokens are generated. Rate-sensitive tokens are those that have a different output rate relative to input rate, such as samplers, encoders or hold tokens. When samplers are used within a system, the up or down sampling ratio must be an integer multiple of the input rate (1:N or N:1). Tokens that take more than one input, such as an adder, require that the rates of the inputs be the same when executing the source code generator. The Code Generator does not support tokens that can vary their own rate based on an input value, examples of which are the variable delay and extract tokens.

The SystemView C-Code Generator creates efficient code specifically designed for a particular target platform, chosen by the user. The platform description includes C data types supported by that target processor. When DSP tokens are part of a system sent to the code generator, the DSP data types must match that of the target or they will be automatically propagated to the data types directly supported by the target. For example, if a DSP token uses a 12-bit integer type, the type will be propagated to a 16-bit integer when generating code for a TI C54x or C6x DSP since these processors do not support 12-bit integer.
3.2 Invoking C-Code Generator

The SystemView C-Code Generator is invoked by first selecting a group of tokens then choosing “Code Generator” from the SystemView Tools menu. Alternatively, one may right-click on the selected block and choose “Code Generator” from the pop-up menu. Selecting Code Generator will bring up the C-Code Generator dialog shown below in Figure 3-1. The dialog takes the user through the five-step process for generating the desired source code: 1) Selecting the target processor; 2) Choosing the type of shell to envelope the source code; 3) Setting the processing buffer size; 4) Selecting which token functions should be generated inline; and 5) Choosing the directory in which to place the generated source code.

![C-Code Generator Dialog](image)

Figure 3.1 Code Generator Dialog

3.3 C-Code Generator Procedure

The next few sections will describe each of the five steps required for generating source, explaining the options available. An example system is shown in the next chapter along with samples of the output source code.
3.3.1 Step 1 – Target Processor Selection

The first step for C-code generation is to choose the target processor. The selection informs the code generator about the C language data types supported by the platform. There are pre-defined options for Texas Instruments C6x and C54x DSPs as well as for 32-bit Windows systems. A custom option is also provided, where a user can specify the appropriate register and exponent sizes for the particular platform being used. Using the custom option, source code can be produced for virtually any machine architecture desired.

The custom processor definition dialog, shown below, allows a user to specify the register and exponent sizes for each of the C language data types supported by the target. The register sizes will determine the minimum and maximum values represented by the target. The code generator uses these sizes to create source code compatible with the chosen machine architecture. Note the optional check box to indicate whether the particular machine treats the character type as always unsigned.

![Custom Processor Definition Dialog](image)

Figure 3.2 Custom Processor Definition Dialog
3.3.2 Step 2 – Source Code Shell

The second step in the code generation process is to select what type of shell, if any, will surround the source code. The Code Generator is capable of creating three layers of source code for the group of selected tokens. The bottom layer is the source code for the individual tokens. The next layer up is the control program that schedules the tokens and passes data between them. The shell is the top layer, providing an external interface to the control program. The user has several shell options including having no shell, or for producing only the individual routines with no control program. Individual routines are useful to examine the functionality of an individual token.

The first two shell options are only available when a Texas Instruments target platform is selected. The first option, TI CCS, produces the framework needed to run a co-simulation on a TI DSP board through SystemView with the optional RTDA token. The RTDA token is available through the DSP library when the Real-Time DSP Architect feature is purchased. The TI CCS option generates the source code for the selected tokens, the control program plus a DSP “main” routine for communicating with CCS through the TI RTDX interface. The second option, TI XDAIS, will produce the first two source code layers and automatically generate the files needed to support the IALG interface needed for TI eXpressDSP compliance. This is especially useful for designers that want to create compliant DSP algorithms for the Texas Instruments platform.

Moving down the “Select Shell Type” list in figure 3.1, the next option is for creating a selected SystemView Custom token. The SystemView shell will produce the source code for each of the individual tokens, the control program, and the interface needed to create a custom token for use within SystemView. This interface contains the type and rate information needed to interact with the SystemView simulation engine. The generated control program and token functions simply operate on a given block of data. The SystemView shell option is useful for examination and/or modification of the functionality of a SystemView token or group of tokens.
The “Stand-Alone Executable” and “Tokens with Connectivity” options from figure 3.1 produce only the control program, and the individual token functions. The options are used to incorporate the functionality of the selected tokens into their own application, on any platform. The former option produces global object memory, whereas the latter uses dynamically allocated memory and results in re-entrant code. The “Tokens Only” option produces only the functionality of the individual tokens selected.

### 3.3.3 Step 3 – Input Buffer Size

The third step in the code generation process is to select the minimum buffer length for processing data through the selected subsystem. The number shown below the edit box is the minimum number of input samples that can be processed at one time by the generated code. If there is more than one input, to the tokens selected for code generation, the number corresponds to the smallest of the input vector lengths. The remaining input vector lengths are determined proportionally from their respective rates. The user can scale up the minimum vector size by providing an integer scale factor in the edit box. The vector size is reflected in the generated source code.

### 3.3.4 Step 4 – In-Line Source Selection

The fourth step for code generation is to specify which, if any, of the selected tokens are placed in-line within the control program. Selecting the “Select Inline” button from the source code generator dialog activates the Inline Code Selection dialog shown in figure 3.3. By default, each distinct token generates a source code routine placed in its own file. Checking the box next to a particular token places the source for the token inside the control program and no separate source file for that token is created.
3.3.5 Step 5 – Directory Path

The fifth step for generating source code is to select the directory to place the generated source code. Select the directory using the standard Windows tree structure shown in the code generator dialog. Right clicking and choosing “new folder” can create a new directory. Once the directory has is selected simply press “OK” and the code generator will create the files necessary, and place them in that directory.
3.4 Code Generator Status Information

Status information from each step of the code generation process is written to a log file. The log file is created in the directory for source code placement and is named after the SystemView system with a “.log” extension. Any problems encountered during code generation can be investigated by noting at which token the failure occurred. External input and output buffer sizes expected by the generated code are also written to the file.

When the C-Code Generator has reached successful completion, a dialog will be displayed as shown below in figure 3.4. Errors encountered will present the user with an error message box and a descriptive message of the problem.

![SystemView Code Generator](image)

**Figure 3.4 C-Code Generator Successful Completion Dialog**
4.0 Automatic C-Code Generation Example

4.1 Example System
The example system shown in Figure 4.1 is for a Gaussian minimum shift-keying (GMSK) baseband modulator. The GMSK system is constructed using many of the tokens from SystemView’s optional DSP library. The DSP tokens operate in bit-true mode with user-defined register sizes, which are displayed in the text boxes next to the tokens. For the example, we will generate source code for tokens 2-9, by dragging a box over the tokens and selecting Code Generator from the Tool menu.

Figure 4.1 GMSK Modulator Example System

4.2 Example C-Code Generator Options
For the example shown in figure 4.1, we want to generate source code for each of the selected tokens (tokens 2-9) along with the control program...
containing connectivity and the execution sequencing. In step one of the Code Generator Dialog, shown in figure 3.1, select Win32 as the platform, with step two being “Tokens with Connectivity”. There are no rate changes in the system, thus the minimum sample size for processing through the selected tokens is one sample. Select one sample as the processing buffer size for step 3.

The tokens used in the example are all fairly simple operations, so for step 4 we will choose to produce tokens 3,4,5,7 and 8 in-line within the control program to save the overhead involved when using a function call. The last step is to choose the directory to place the files, after which the code generator will produce the source code files shown below in figure 4.2.

![Figure 4.2 GMSK Example Source Code Files](image)

The Code Generator creates a C language header file (<token>.h) for each of the tokens in the selected group, whether or not the token is selected for Automatic C-Code Generation.
inlining. The header files contains a list of function prototypes for the token functions plus definitions of structures needed by the token. The source code for the inline token is placed within the control program. Code Generator creates source files (<token>.c) containing the token functionality and, if needed, files for initialization (<token>_init.c) and pointer assignment (<token>_ptr.c) for non inline tokens.

The control program is divided into seven files. Each is prefixed with the name of the SystemView simulation (“GMSK” in the example). The “GMSK” header files contain the dynamic and read-only memory requirements for each of the tokens. The read-only tag is a suggestion to the user that the block of memory is only read from by the generated source. The “GMSK” source files are broken into separate initialization, pointer-assignment, execution and finalization files.

### 4.3 C-Code Generator Source Code

A source code listing for the GMSK master control program (GMSK.c) is shown in Appendix A. Key parts of the generated source code are described below, including the function argument list, the master while loop, token-ready toggles, token function code and external output assignment.

Tokens external to those selected for code generation send/receive data from the control program routine through arrays shown in the program argument list. Also in the list are the dynamic and read-only memory blocks used by the tokens in the generated system. A list of where the external inputs/outputs is shown for informational purposes.
The master while loop manages the execution of the control program by running the system until all external inputs are used and all external outputs are filled. The Code Generator is designed to work with complicated SystemView systems, therefore, the generated code accounts for multiple rates and varying data types. When systems containing rate-sensitive tokens are selected for code generation, the control program must account for the situation where a multiple-input token has one input ready while waiting for the other. The generated source code accounts for this situation by creating a data-ready toggle for each of the affected tokens. A wrapper is placed around the token function call, or function source for inline tokens, that checks whether the toggles are set to ready.

The control program precedes each token function block by a comment telling the user the token number and a token description. The control program will optionally check the data-ready toggles and then call the token function. If the token is connected externally, the output from that token is placed in an external output buffer.
5.0 SystemView Custom Code Example

5.1 Custom Code Overview
The SystemView User’s Guide contains detailed information about building a Custom Library in Chapter 5, Section 9. Essentially, the SystemView Custom Code option allows users to expand SystemView functionality by creating their own Win32 DLLs containing source for customized tokens. Using the GMSK example presented in Chapter 4, a Custom Code DLL is built using the source code generated from the Code Generator.

5.2 Custom Code Source Generation
Looking back at figure 3.1, step 2, choose the SystemView Custom Token option. For step three from the same figure, change the value to 32, so that we are processing a block of 32 samples at a time. The tokens used in the example from chapter 4 are chosen for inlining, tokens 3,4,5,7,8. After setting the directory to "C:\Program Files\SystemView\Examples\CCodeGenerator\GMSK\Custom", press the OK button and all source code needed to build a Custom Code DLL is created. Two additional files, relative to the example in chapter 4, are created for building a Custom Code DLL. The first is a SystemView interface file that communicates with SystemView and calls the master control program. The second additional file is a Win32 DLL export definition file (<system>.def) containing a list of externally callable routines from the DLL.

5.3 Custom Code DLL Creation
The following steps assume use of the Microsoft Visual C++ comiler, although any C compiler that can create a Win32 DLL should work. The project files used in this example are included with the SystemView distribution. The first step once the compiler is running is to create a new Win32 DLL, name this DLL, Custom and place it in the directory used for the Code Generator. The next step is to include all the files created by code generator from the destination directory. The third and last step is to build the DLL.
5.4 Custom Code Co-Simulation

The example GMSK Custom Code DLL can then be brought back into SystemView and run in co-simulation with the SystemView engine. Token 14, in the example system shown below in Figure 5.1, is a custom token from the DLL created using the source code generated in the example. When the system is run, the top path, tokens 2-13, run through the SystemView engine, while the bottom path is run through our custom DLL. The real-time sinks shows that the code is generating the identical result, therefore, providing the functionality of the selected tokens (2-9).

Figure 5.1 GMSK Custom Token Co-simulation
5.5 Real-Time DSP Architect Co-Simulation

The Real-Time DSP Architect (RTDA) feature supports hardware-in-the-loop simulation capability within SystemView, which is documented in a separate RTDA manual. The RTDA feature requires a Texas Instruments DSK or EVM board and their Integrated Development Environment Code Composer Studio. The code generation procedure is similar to that presented for creating a Custom Code DLL. Changing the selection in step 2 of figure 3.1 to “TI CCS” produces the source code shown in chapter 4, plus source files containing interface code to communicate with SystemView through an RTDA token.

A SystemView RTDA token communicates with Code Composer Studio through a Texas Instruments COM automation object. The on-board DSP program is driven using SystemView’s proprietary protocol over Texas Instruments’ RTDX channel.

During a SystemView simulation, data is passed from SystemView to an RTDA token, which then passes the data through Code Composer Studio to the DSK board, where it is processed and sent back through Code Composer Studio to SystemView. The RTDA option provides implementation on a real DSP.
Appendix A – Control Program Source Listing

/*********************************************************************
Copyright© 2005 by Eagleware Corporation

Certain rights reserved under the U.S. Copyright Act.

This source code is for the exclusive use of the authorized SystemView by ELANIX Licensee, and may not be distributed to any other persons, organizations or entities without the expressed written consent of Eagleware Corporation

**********************************************************************/

#include "GMSK_Defs.h"

Void GMSK(Char nExtIn0,
         Short *nExtOut0,
         Short *nExtOut1,
         struct GMSK_pRom *pRom,
         struct GMSK_pMem *pMem)
{
    /*
    nExtIn0 is Input 0 From t1
    nExtOut0 is Port 0 Out to t10
    nExtOut1 is Port 1 Out to t11
    */
    Int nExtOutInd0 = 0;
    Int nExtOutInd1 = 0;
    Int nTemp;

    while ((nExtOutInd0 < EXT_OUT_LEN_0)
            || (nExtOutInd1 < EXT_OUT_LEN_1))
    {
        /* Token 2 - Gaussian */
        if (!pMem->bOutToggle_t2 ^ pMem->bInToggle_t2t3)
        {
            DspFIR(nExtIn0,
                   &(pMem->nFIRout_t2),
                   3,
                   &(pMem->DspFIRObj_t2));
            pMem->bOutToggle_t2 = !pMem->bOutToggle_t2;
        }
        if (pMem->bOutToggle_t3 ^ pMem->bInToggle_t3t4)
        {
            /* Token 3 - Cnst Mltply */
            if (!pMem->bOutToggle_t3 ^ pMem->bInToggle_t3t4)
            {
                pMem->nGain_t3 = pMem->nFIRout_t2 * 0x02b6;
                pMem->bInToggle_t2t3 = pMem->bOutToggle_t2;
                pMem->bOutToggle_t3 = !pMem->bOutToggle_t3;
            }
            if (pMem->bOutToggle_t3 ^ pMem->bInToggle_t3t4)
            {  

    Automatic C-Code Generation
/* Token 4 - Integrator */
if (!(pMem->bOutToggle_t4 ^ pMem->bInToggle_t4t5))
{
    if (pMem->bReset_t4 > 0)
    {
        pMem->nIntg_t4 = 0;
    }
    pMem->nIntg_t4 += pMem->nGain_t3;
    pMem->bInToggle_t3t4 = pMem->bOutToggle_t3;
    pMem->bOutToggle_t4 = !pMem->bOutToggle_t4;
}

if (pMem->bOutToggle_t4 ^ pMem->bInToggle_t4t5)
{
    /* Token 5 - Xtract Bits */
    if (!(pMem->bOutToggle_t5 ^ pMem->bInToggle_t5t6)
        && !(pMem->bOutToggle_t5 ^ pMem->bInToggle_t5t9))
    {
        nTemp = pMem->nIntg_t4 << 9;
        pMem->nOut_t5 = (Uns)nTemp >> 23;
        pMem->bInToggle_t4t5 = pMem->bOutToggle_t4;
        pMem->bOutToggle_t5 = !pMem->bOutToggle_t5;
    }
}

/* Token 7 - Step Fct */
if (!(pMem->bOutToggle_t7 ^ pMem->bInToggle_t7t6))
{
    pMem->dSrcOut_t7 = pRom->dAmplitude_t7;
    pMem->bOutToggle_t7 = !pMem->bOutToggle_t7;
}

if ((pMem->bOutToggle_t5 ^ pMem->bInToggle_t5t6)
    && (pMem->bOutToggle_t7 ^ pMem->bInToggle_t7t6))
{
    /* Token 6 - Sin Cos */
    pMem->bIsSin_t6 = (Int) pMem->dSrcOut_t7;
    CosLUT_Select(pMem->nOut_t5,
                  pMem->bIsSin_t6,
                  &pMem->nSinLUTOut_t6,
                  512,
                  pRom->nSinLUT_t6);
    nExtOutInd0++;
    *nExtOut0 = pMem->nSinLUTOut_t6;
    pMem->bInToggle_t5t6 = pMem->bOutToggle_t5;
    pMem->bInToggle_t7t6 = pMem->bOutToggle_t7;
}

/* Token 8 - Step Fct */
if (!(pMem->bOutToggle_t8 ^ pMem->bInToggle_t8t9))
{

pMem->dSrcOut_t8 = pRom->dAmplitude_t8;
pMem->bOutToggle_t8 = !pMem->bOutToggle_t8;
}
if ((pMem->bOutToggle_t5 ^ pMem->bInToggle_t5t9) && (pMem->bOutToggle_t8 ^ pMem->bInToggle_t8t9))
{
    /* Token 9 - Sin Cos */
    pMem->bIsSin_t9 = (Int) pMem->dSrcOut_t8;
    CosLUT_Select(pMem->nOut_t5, pMem->bIsSin_t9,
                  &pMem->nSinLUTOut_t9, 512,
                  pRom->nSinLUT_t9);
    nExtOutInd1++;
    *nExtOut1 = pMem->nSinLUTOut_t9;
    pMem->bInToggle_t5t9 = pMem->bOutToggle_t5;
    pMem->bInToggle_t8t9 = pMem->bOutToggle_t8;
}
}