NXP Semiconductors
Application Notes

Document Number: AN13192
Rev. 0, 02/2021

S32V234 APEX SW Self-test Method and Example

Contents

1. Introduction

The S32V234 processor offers dual APEX accelerators that provides high-performance parallel processing capabilities. It is a parallel hybrid processor well suited for the processing of large amount of data. During the processor run-time, a random hardware failure may occur unpredictably. For example, physical damage particles (alpha, neutron) or EMI-radiation may lead to a permanent fault which cannot be recovered. Therefore, it is necessary to have a method to detect APEX hardware faults when APEX is used in safety related computation.

This application note describes how to detect permanent faults inside the hardware through software at the application level and provides an example to fulfill the published safety assumptions in S32V234 Safety Manual. It is assumed that the developer is familiar with the APEX software development environment and the code implementation is based on Vision SDK 1.5.1 and eIQ Auto 2.0.0.

This document contains the following sections:

- APEX SW self-test method: This section describes the approach towards APEX permanent faults detection based on safety manual assumption.
- APEX fault injection and reaction: This section describes how to inject faults into APEX and the corresponding reaction.
- APEX SW self-test implementation: This section describes a SW self-test example based on a specific pattern test on the entire APEX pipeline at the application level and an example for evaluating the self-test effectiveness.

The following abbreviations are used in the application note.

Table 1. Acronyms and abbreviations

Abbreviations	Description
APU	Array Processing Unit
ACF	APEX Core Framework
CU	Computational Unit
DMA	Direct Memory Access
FTTI	Fault Tolerant Time Interval
DMS	Driver monitoring system
NN	Neural network

2. APEX SW self-test method

The Figure 1 shows the APEX block diagram. In general, the fault detection for the APEX is the responsibility of the software since most hardware components of APEX are not designed with hardware safety mechanisms. For many applications, it is sufficient to check the data processing path for permanent faults once within FTTI.

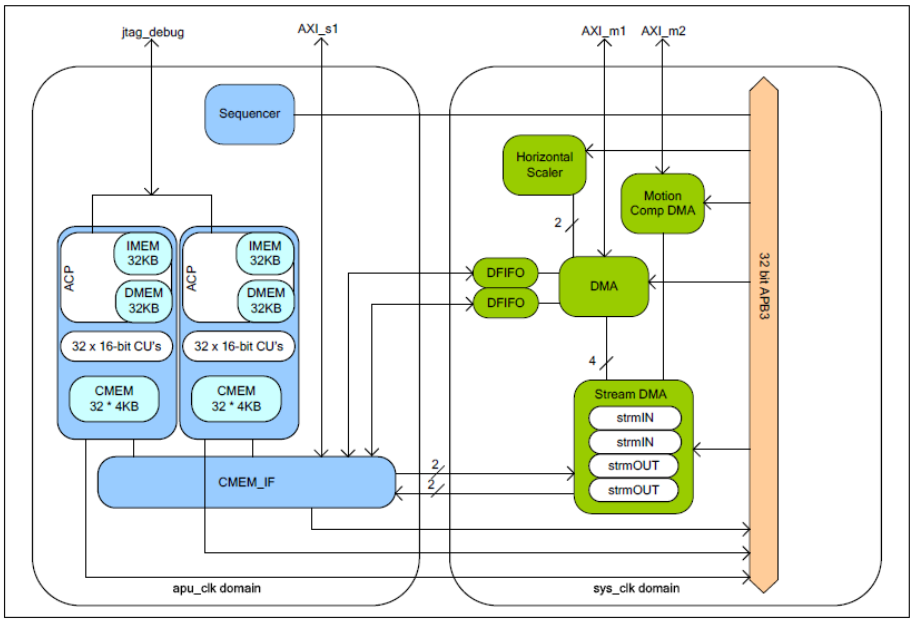

Figure 1. **APEX Block diagrams**

At a lower level, the APEX processing pipeline contains the following steps:

1. Transfer data from external/host memory to APU memory.
2. Process input data (residing in APU memory) with the APU processor to produce output data (also in APU memory).
3. Transfer output data from APU memory back to external/host memory.

Random hardware faults may occur at a random time in various hardware components (e.g. DMA, DFIFO). Software self-test aims to detect these faults during processing. There are several assumptions in the S32V234 Safety Manual, some of them are:

- Assumption:[SM_951] The APEX will be tested for permanent faults by SW. For example, known picture content will be processed and compared to a known reference.
- Assumption:[SM_952] The evaluation of the result will be done with a HW element independent from APEX (e.g. Cortex-A53®).
- Assumption:[SM_954] A timeout supervision will be used to detect a stalled APEX engine.

There are several approaches to detect permanent faults inside APEX based on the above assumptions.

1) Assumption:[SM_951]: Test-pattern method: A known test-pattern is processed and compared with an expected value, and this application note mainly introduces this approach.
2) Instruction level checks: It is possible to execute instruction-set level checks by developing a software test library that would test all the sub-components of the APEX (DMA, Computational Units or CU, etc.), but this is not covered in this document.
3) Assumption:[SM_952]: CPU Cross check: Cortex-A53 is used to compare APEX processing result with expected result. Cortex-A53 is used to calculate expected value based on dynamic test pattern input if necessary.

4) Assumption:[SM_954]: Timeout supervision: APEX timeout mechanism has been implemented in Vision SDK APEX driver. APEX timeout errors can be captured by the application level through the return value of the running process. The developer needs to deal with such errors as ACF_TIMEOUT_ERROR using appropriate reaction.

3. Fault injection and reaction

Some fault injection methods can be used to inject errors in APEX to trigger and verify the effectiveness of the safety mechanism related to APEX. During the system level development, the safety system developer is required to test the validity of the self-test approach in the specific safety-related system. Table 2 shows some ways to inject errors into APEX at the application level.

Table 2. Fault injection and reaction

Safety mechanism	Fault injection	Description	Reaction
APEX SW Self-test	Fault injection in test-pattern	Change the test-pattern's values to simulate data transfer errors.	Changes caused by error injection can be detected via self-test. The reaction to the fault is application dependent.
APEX C/SMEM EDC	Fault injection in APEX memories	S32V includes a feature to validate the APEX memory errors detection logic. The developer could inject faults into memories via programming. (refer to Reference Manual Rev. 5 chapter 71 for details)	The parity errors of APEX code memory and shared memory are directly reported to the FCCU for user-defined handling, e.g. interrupt or reset.
APEX Timeout Supervision	Fault injection at run time	Reset all APEX hardware blocks and APEX driver specific entities via programming.	A timeout error can be captured and handled by the application.

In most cases, the application probably take an aggressive recovery approach and call APEX_Reset(<apex id>) on the faulty APEX if possible.

4. APEX SW self-test implementation

This section demonstrates the test-pattern method implementation based on the Driver Monitoring System (DMS) demo. The test-pattern method is efficient to guarantee all the APEX resources used in your application can be tested and easy to implement. The APEX self-test workflow diagram is shown in the following figure.

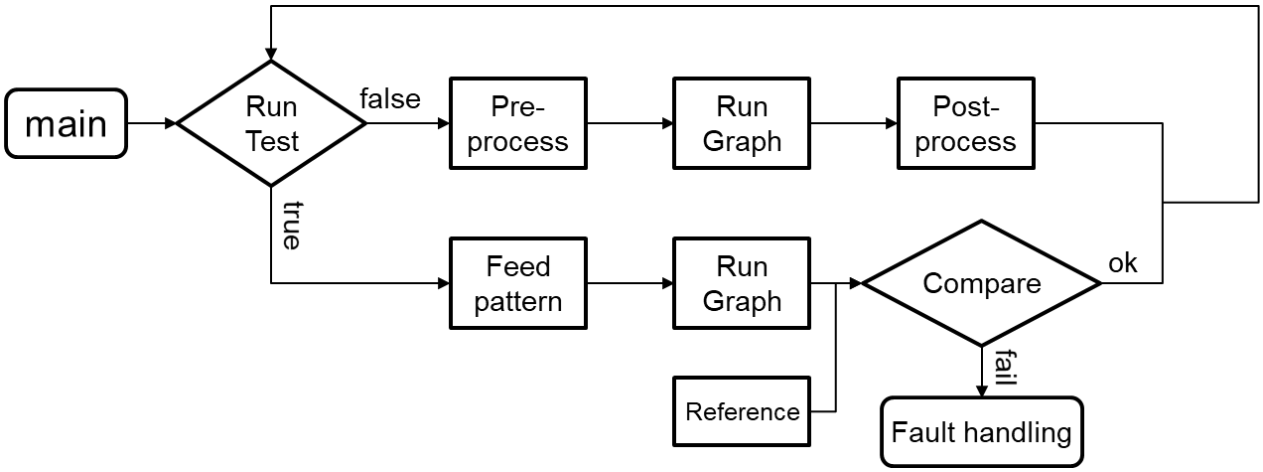

Figure 2. **Self-test work flow**

The APEX accelerator is typically used for a single functionality in a use-case, i.e. it is used for a single CNN process in eIQ Auto which is used for the entire run-time period. It means there is no graph change (graph is used to represent the execution of a NN) at run-time. Thus you can use this functionality to test for permanent faults inside the HW.

Example 1(eIQ Auto): Use the same NN with a different pre-defined input, and then check whether the output matches the expected value. This guarantees that all the APEX resources used in your application are tested.

Example 2(APEXCV): If your application is running algo X on APEX, e.g. HOG detector, it could use a mini invocation of it for the self-test with a small input, and then check whether the output matches the expected value.

4.1 Test-pattern example in eIQ Auto

The NXP eIQ Auto Deep Learning toolkit for S32V23x processors provides functionality for developers to design their own convolution neural network (CNN) based applications while taking advantage of NXP's massively parallel APEX architecture.

The workflow of eIQ Auto can be simplified as follows:

1) Fetch input data (with preprocessing if needed) and feed to the network input tensor.
2) Execute the network.
3) Fetch output data from network output tensor and continue postprocessing.

The design of the test-pattern affects fault detection coverage to a certain extent. The test-pattern can be customized by the developer to achieve the expected target of APEX SW self-test. There are two examples for reference shown in the following figure (Consider a 4x4 feature-map with three channels in NHWC format tensor).

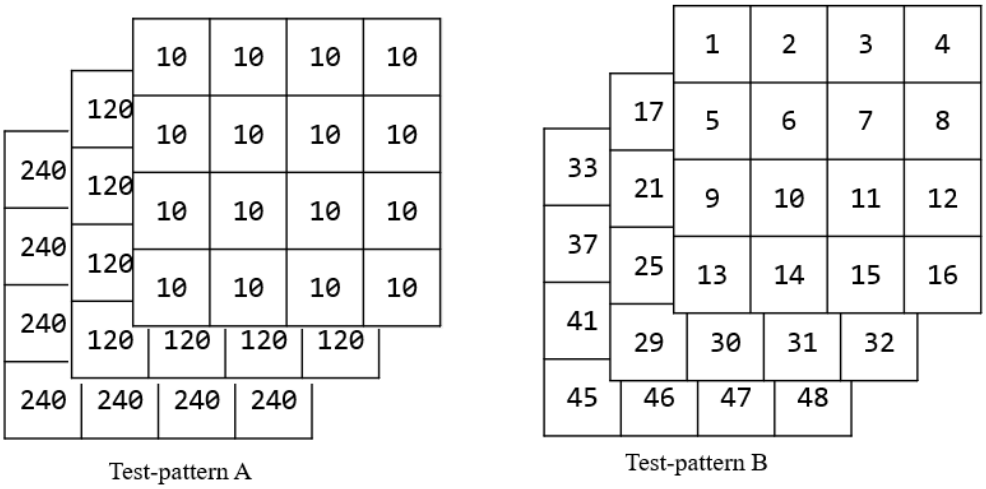

Figure 3. **Test-pattern data examples**

4.2 Implementation

In the aspect of safety, eIQ Auto does not verify the correctness of computations, it is the application's responsibility to detect errors in computations arising from hardware faults or other forms of data corruption. This section introduces the APEX SW self-test implementation based on the DMS demo.

The DMS provides a real-time evaluation of the presence and the state of the driver. The driver behavior classification NN was processed by eIQ Auto through APEX processor. The known test-pattern can be inserted into the processing stream periodically within the FTTI. The processing time of self-test depends on the complexity of the deployed NN in the application. If the self-test time takes too long, the developer should consider using small input size for self-test. The developer can also create a customized simple graph as a self-test graph to decrease the self-test time and increase the flexibility.

The following steps show running the same NN with a small input size for self-testing.

1) Define a small-size tensor as a test-pattern container, the tensor datatype should be int8 and the target name needs to be set to APEX.

```
// Clone the graph used in the application.
std::map<std::string, Tensor*> lInputTensors;
auto netSelfTest = originalGraph->Clone(workspace, lInputTensors);
// Configure input to self-test graph
lInput = netSelfTest->AddTensor(std::unique_ptr<Tensor>(Tensor::Create<>
("APEX_NET_INPUT_TENSOR", DataType_t::SIGNED_8BIT, TensorShape<TensorFormat_t::NHWC>{1, PATTERN_HEIGHT,
PATTERN_WIDTH, 3},TensorLayout<TensorFormat_t::NHWC>())));
lInput->SetQuantParams({QuantInfo(-1, 1)});
lInput->Allocate(Allocation_t::OAL);
```

```
// Set target to APEX
Status_t status;
status = netSelfTest->SetTargetHint(TargetType::APEX());
if (Status_t::SUCCESS != status)
{
  std::cout << "ERROR: Can not start APEX self test" << std::endl;
}
status = netSelfTest->Prepare();
if (Status_t::SUCCESS != status)
{
  std::cout << "ERROR: Can not start APEX self test" << std::endl;
}
```

2) Fill data for test-pattern

```
// Create test-pattern
cv::Mat lPattern(PATTERN_HEIGHT, PATTERN_WIDTH, CV_8UC3, cv::Scalar(10, 120, 240));
// Load pattern data to Tensor
TensorRange<int8_t> lTensorRange(*mInputTensor);
auto ptr = lTensorRange.begin();
for (int32_t row = 0; row < lPattern.rows; ++row)
{
  for (int32_t col = 0; col < lPattern.cols; ++col)
  {
    auto &Pixel = lPattern.at<cv::Vec3b>(row, col);
    *ptr++ = Pixel.val[0];
    *ptr++ = Pixel.val[1];
    *ptr++ = Pixel.val[2];
  }
}
```

3) Run self-test and compare output result with known results (Note that the developer should prepare the known reference output tensor by running the NN beforehand).

```
status = netSelfTest->Run();
if (Status_t::SUCCESS != status)
{
  std::cout << "APEX net verification failed" << std::endl;
  return false;
}
if(TensorEqual())
{
  std::cout << "* [APEX - AIRunner self test] SUCCESS *\n";
}
else
{
  std::cout << "* [APEX - AIRunner self test] ERROR *\n";
}
```

4.3 Example for evaluation of test-pattern method

Assumption on certain preconditions: For the APEX SW self-test approach mentioned in this document, it is assumed that APEX permanent faults will lead to incorrect final calculation results. However, it should be noted that some APEX internal faults may not be reflected in the calculation results. The diagnostic coverage is related to the algorithms that the developers use in self-test, e.g. if you use the APEX gaussian filter as a self-test algorithm, some of the test-pattern data will be smoothed or removed by the filter, which means it will lose some of the detection effectiveness during processing.

Example for test-pattern fault injection: Fault can be injected into the test-pattern by modifying one pixel's single-channel value to simulate APEX fault scenario (e.g. Input pattern is a BGR Mat with size of [32,32,3] accessing the elements of the matrix and modifying one pixel's R channel data), then count detected faults and total fault injection times.

The fault detection coverage is based on the following formula:

$$fault\ detection\ coverage\ = \frac{Detected\ faults}{Total\ injected\ faults}$$

Test Result:

Table 3. Fault detection coverage

QuantInfo	Detected faults	Total injected faults	Fault detection coverage	Self-test time
(-1, 0)	781477	786432	0.993699	2.2 ms
(0, 128)	775911	786432	0.986622	
(0, 255)	773843	786432	0.983992	

- The QuantInfo represents the input Tensor's Min/Max value that is for quantization. More details see eIQ_Auto_UserGuide chapter 13.32.
- Total injected faults: The test-pattern is a Mat with [32, 32, 3] shape, injecting fault by modifying one pixel's single-channel value from 0-255, so the number of total injected faults is 32*32*3*256 = 786432.
- The test results were based on test-pattern A in *Figure 3* and the same NN used in the DMS demo as the self-test algorithm.

5. Summary

APEX SW self-test can detect permanent faults inside the hardware and enhance the system's robustness, while the self-test approach is highly related to application scenarios. The developer needs to implement corresponding system level safety measures. The verification of self-test method also needs to be designed by the developer. The sample code in this note is for reference only. For more information, please refer to the device Reference Manual and the Safety Manual.

6. Reference

1. Safety Manual for S32V234, S32V234SM, Rev. 3, 10/2017
2. ACF User Guide, UG-10267-03-17 – included in VSDK release package
3. S32V234 Reference Manual, S32V234RM, Rev. 5, 11/2019
4. eIQ Auto UserGuide – included in eIQ Auto release package

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN13192
Rev. 0
02/2021