AROCOM Technical Note TN0019

AMX Controller der NX-Serie: Programmier-Empfehlungen

History

Revision	Date	Creator	Description
a	22.11.2022	CL/LT	First edition

Products

AMX NX Controller
AMX MCP ControlPads
AMX DVX/DGX with NX Controller

File Name

AROCOM_TN0019a_AMX_NX_Programmier_Empfehlungen_DE

Inhaltsverzeichnis

Referenzierte Dokumente

Dieses Dokument TN0019 bezieht sich unter anderem auf folgende Dokumente, welche AMX veröffentlicht hat:

- NetLinx.LanguageReferenceGuide.pdf
  - Kapitel «Understanding When DEFINE_PROGRAM Runs»
- NX-Series.WebConsole-ProgrammingGuide.pdf
  - Kapitel «Differences in DEFINE_PROGRAM Program Execution»
- NX_Controller_Best_Practices_for_Longevity.pdf

Einleitung

Beim Programmieren von AMX-Kontrollern der NX-Serie muss auf bestimmte Details geachtet werden, welche bei den Vorgänger-Typen, den NI-Kontrollern, weniger wichtig waren.

Dieses Dokument ist nicht nur als Empfehlung zu betrachten. Einige der enthaltenen Anweisungen müssen zwingend eingehalten werden, damit die bewährte Zuverlässigkeit der AMX-Kontroller gewährleistet ist.

Variablen

Globale Variablen

Globale Variablen, welche in der Section `DEFINE_VARIABLE` definiert werden, sind per Default `NON_VOLATILE`. Das bedeutet, dass sie nicht im schnellen RAM gespeichert werden, sondern im NVRAM.

Als «NVRAM» (Non-Volatile) wird der Speicher bezeichnet, welcher ohne Stromversorgung erhalten bleibt. Dieser ist jedoch etwas langsamer und auch deutlich kleiner.

In den meisten Anwendungen werden keine `NON_VOLATILE` Variablen benötigt. Daher wird empfohlen, Variablen in der Section `DEFINE_VARIABLE` immer als `VOLATILE zu` definiert. Die Definition `NON_VOLATILE` sollte wirklich nur zum Einsatz kommen, wenn diese unumgänglich ist.

Beispiel:

```
DEFINE_VARIABLE
VOLATILE     INTEGER nCounter
VOLATILE     CHAR    nLabels[3][50]
NON_VOLATILE INTEGER nStartupMode
VOLATILE     FLOAT   fGainLevel
VOLATILE     INTEGER nMicMute[8]
VOLATILE     ...
VOLATILE     ...
VOLATILE     ...
```

Hinweis:

Die folgenden zwei Varianten der Variablen-Definition bewirken das Gleiche, weil globale Variablen per Default `NON_VOLATILE` sind.

```
DEFINE_VARIABLE
NON_VOLATILE INTEGER nStartupMode
INTEGER nStartupMode
```

Konstanten

Konstanten werden in der Section `DEFINE_ CONSTANT` definiert und sind immer im RAM gespeichert.

Eine Definition als `VOLATILE` oder `NON_VOLATILE` ist deshalb nicht notwendig.

Lokale statische Variablen

Lokale statische Variablen, in Netlinx als `LOCAL_VAR` bezeichnet, werden verwendet in

- Events
- Functions
- anderen, nicht globalen Scopes wie zum Beispiel Codeblöcken in geschweiften Klammern {…} innerhalb von `DEFINE_PROGRAM` oder `DEFINE_START`.

Diese Variablen werden per Default im NVRAM gespeichert (NVRAM = langsamer und kleiner).

Deshalb wird empfohlen, auch `LOCAL_VAR` Variablen als `VOLATILE` zu definieren, wenn sie nicht zwingend erhalten bleiben müssen.

Beispiel:

```
DATA_EVENT[vdvPROJ]
  {
  ONLINE:
    {
    LOCAL_VAR VOLATILE INTEGER nForLoop        // RAM
    LOCAL_VAR INTEGER nCounter                 // NVRAM
    FOR(nForLoop=1; nForLoop<=12; nForLoop++)
      {
      ...
      }
    nCounter++
    }
  }
```

Das Beispiel zeigt, dass die Variable *nForLoop* volatile ist. Non-Volatile würde hier keinen Sinn ergeben, da der Variable *nForLoop* in der FOR-Anweisung jedes Mal der Wert 1 zugewiesen wird.

Beachte:

Lokale, Non-Volatile, Variablen behalten ihren Wert nach einem Startup. Im Beispiel zählt die Variable *nCounter* nach dem Startup jeden Online-Event von *vdvPROJ* und wird nicht auf 0 gesetzt. Die Zuweisung eines Initialisierungswertes ist bei lokalen Variablen nicht möglich:

```
LOCAL_VAR INTEGER nCounter = 5     // = Error
```

Es gibt also keine Möglichkeit, der Variable *nCounter* nach einem Startup einen definierten Wert zuzuweisen! Dies sollte beim Erstellen von Netlinx-Code immer berücksichtigt werden.

BUTTON_EVENT

Beim `BUTTON_EVENT` muss das `PUSH:` Statement und das `RELEASE:` Statement zwingend vorhanden sein. Das `HOLD:` Statement ist optional.

Wird ein `PUSH:` Statement oder ein `RELEASE:` Statement nicht benötigt, muss es dennoch mit leeren Klammern im Code aufgeführt werden.

Folgendes Beispiel zeigt einen `BUTTON_EVENT` bei welchem nur im `PUSH:` Statement Code ausgeführt wird und das `RELEASE:` Statement nicht benötigt wird:

```
BUTTON_EVENT[dvTP,0]
   {
   PUSH:
      {
      PULSE[dvRelay,1]
      }
   RELEASE:{}
}
```

CHANNEL_EVENT

Beim `CHANNEL_EVENT` muss das `ON:` Statement und das `OFF:` Statement zwingend vorhanden sein.

Wird ein `ON:` Statement oder ein `OFF:` Statement nicht benötigt, muss es dennoch mit leeren Klammern im Code aufgeführt werden.

Folgendes Beispiel zeigt einen `CHANNEL_EVENT`, bei welchem nur im `ON:` Statement Code ausgeführt wird und das `OFF:` Statement nicht benötigt wird:

```
CHANNEL_EVENT[dvIO,3]
   {
   ON:
      {
      PULSE[dvRelay,1]
      }
   OFF:{}
}
```

DEFINE_PROGRAM

VARIABLEN

Das Ändern von globalen Variablen in der Section `DEFINE_PROGRAM` wie in folgendem Beispiel muss **zwingend** vermieden werden:

```
DEFINE_PROGRAM
nStatuscounter = nMode+1                    // Don't do this !!!
```

Das Abfragen von globalen Variablen wie in folgendem Beispiel ist jedoch unkritisch:

```
DEFINE_PROGRAM
[dvTP,201] = (nCurrentInput = 1)
[dvTP,202] = (nCurrentInput = 2)
[dvTP,203] = (nCurrentInput = 3)
[dvTP,204] = (nCurrentInput = 4)
```

Dennoch wird auch bei diesem Beispiel empfohlen, den Code in einem WAIT auszuführen.

WAIT

Generell wird dringend empfohlen, dass Code in der Section `DEFINE_PROGRAM` immer in einem WAIT (zum Beispiel ein WAIT 1 = 100ms) ausgeführt wird. Folgendes Beispiel ist unkritisch, obschon die Variable *nINC* innerhalb von `DEFINE_PROGRAM` geändert wird:

```
DEFINE_PROGRAM
WAIT 1
  {
  STACK_VAR INTEGER nINC
  FOR(nINC=1;nINC<=8;nINC ++)
    {
    [dvTP,200+INC] = (nCurrentInput = nINC)
    }
  }
```

Dieses Beispiel ist deshalb unkritisch, weil…

- die Variable *nINC* nur innerhalb des Scopes von WAIT 1 gültig ist (die Variable ist nicht global).
- die Variable *nINC* als STACK_VAR definiert ist und somit im RAM gespeichert wird.
- dieser Code nur alle 100ms ausgeführt wird.

TIMELINE

Anstelle eines WAIT in der Section `DEFINE_PROGRAM` kann auch eine `TIMELINE` verwendet werden, wie folgendes Beispiel zeigt:

```
DEFINE_CONSTANT
TL2_ID_FEEDBACK = 2



DEFINE_VARIABLE
LONG TL2_INTERVAL_FEEDBACK[] = {100}      // 100ms



DEFINE_START
TIMELINE_CREATE(TL2_ID_FEEDBACK, TL2_INTERVAL_FEEDBACK, 1,
                              TIMELINE_RELATIVE, TIMELINE_REPEAT)



DEFINE_EVENT
TIMELINE_EVENT[TL2_ID_FEEDBACK]           // Timeline-2: 100ms
  {
  hier Code einfügen welcher alle 100ms ausgeführt werden soll,
  zum Beispiel Feedback von Buttons auf Touchpanel

  IF(TIMELINE.REPETITION % 5 = 0)
    {
    hier Code einfügen welcher alle 500ms ausgeführt werden soll
    }
  }
```

SEND_STRING 0

Eine beliebte Methode, um Debug-Informationen auszugeben, ist die Verwendung von `SEND_STRING 0`.

Wichtig zu wissen ist, dass `SEND_STRING 0` -Meldungen auch als AUDIT LOG gespeichert werden. Die AUDIT LOGs werden auf der SD-Karte gespeichert. Eine SD-Karte kann eine hohe Anzahl Schreibzyklen verarbeiten, jedoch nicht unendlich viele. Dies bedeutet, dass eine übermässige Verwendung von `SEND_STRING 0` langfristig die SD-Karte beschädigen kann.

AMX_LOG()

Folgendes Beispiel zeigt den alternativen Befehl `AMX_LOG()` anstelle des Befehls `SEND_STRING 0,"'Hello World'"`:

```
AMX_LOG(AMX_DEBUG,"'Hello World'")
```

Um die Meldung 'Hello World' in Diagnostics zu sehen, muss zuvor der LOG LEVEL auf DEBUG eingestellt werden. Dazu gibt es zwei Möglichkeiten:

1. In der Netlinx-Software z.B. in der Section `DEFINE_START` mit dem Befehl `SET_LOG_LEVEL(AMX_DEBUG)`
2. In Telnet oder SSH mit dem Befehl `netlinx log level` … und anschliessend Level 4 für Debug wählen

Virtuelles Device

Eine weitere Möglichkeit, Debug-Informationen auszugeben, ist die Verwendung eines beliebigen virtuellen Devices, von welchem jedoch kein DATA_EVENT implementiert sein darf.

Arocom verwendet dazu standardmässig folgendes virtuelles Device:

```
DEFINE_DEVICE
vdvDebOutp = 33011:01:0      // Ausgabe von Debug
                                Informationen als SEND_STRING
```

Module

Es darf nicht vergessen werden, dass all diese Massnahmen auch für Module und Include-Files gelten!

Sind Module nur in kompilierter Form vorhanden, muss zwingend auch der Source-Code dazu bekannt sein und auf Korrektheit überprüft werden.

AUDIT LOG

Neben dem bereits erwähnten Befehl `SEND_STRING 0` speichert das Netlinx Betriebssystem auch direkt verschiedene Ereignisse in den AUDIT LOGs. Je nach Funktionalität der Software kann dies recht häufig geschehen. Versucht beispielsweise ein Kontroller häufig eine IP-Verbindung zu einem Gerät zu öffnen, welches im Netzwerk nicht vorhanden ist, wird jeder dadurch entstandene IP-Error in den AUDIT LOGs gespeichert.

Ab der Master Firmware Version 1.6.205 ist das Speichern dieser Ereignisse ausschaltbar. Deshalb empfehlen wir diese oder eine höhere Firmware Version zu verwenden.

FTP Write

Das Schreiben von Dateien auf den FTP-Speicher mit den Befehlen `FILE_WRITE` und `FILE_WRITE_LINE` sollte minimiert werden. Auf keinen Fall sollten in einem schnellen Loop regelmässig Dateien auf den FTP-Speicher geschrieben werden.

Spezialfall MCP

In den Produkten der MCP-Serie ist kein NVRAM eingebaut. Dies sollte beim Erstellen von Netlinx-Code immer berücksichtigt werden.

Das bedeutet: In Netlinx-Code für MCPs macht es keinen Sinn, Variablen als `NON_VOLATILE` zu deklarieren, denn sie werden sich wie `VOLATILE` Variablen verhalten.

Dies ist auch beim Erstellen von Netlinx-Modulen zu berücksichtigen. Müssen in einem Modul zwingend `NON_VOLATILE` Variablen verwendet werden, ist das Modul nicht mit MCPs kompatibel.

SWITCH...CASE

Beim `SWITCH...CASE` Statement muss darauf geachtet werden, dass im SWITCH-Ausdruck nicht auf ein Array mit ungültigem Index zugegriffen wird. Ein solcher Zugriff wird zwar bei korrekter Programmierung nicht entstehen, kommt aber in der Praxis immer wieder vor.

Bei NX-Kontrollern führt dieser Fehler in bestimmten Fällen zu einem Reboot, was bei den Vorgängermodellen der NI-Serie nicht so war. Daher ist gerade bei der Anpassung einer bestehenden, älteren Lösung an einen NX-Kontroller Vorsicht geboten.

Folgendes Beispiel ist unkritisch, da *nMode* kein Array ist:

```
STACK_VAR INTEGER nMode
...
SWITCH(nMode)
  {
  CASE 0:  { ...}
  CASE 1:  { ...}
  DEFAULT: { ...}
  }
```

Bei folgendem Beispiel darf die Variable *nIndex* auf keinen Fall den Wert 0 oder einen Wert >6 erreichen, da das Array *nSetup[]* nur 6 gross ist. Wenn dieser Code in einer Funktion ausgeführt wird, verursacht dies einen Crash des Kontrollers:

```
DEFINE_FUNCTION Test()
  {
  STACK_VAR INTEGER nIndex
  STACK_VAR INTEGER nSetup[6]
  ...
  SWITCH(nSetup[nIndex])
    {
    CASE 0:  { ...}
    CASE 1:  { ...}
    DEFAULT: { ...}
    }
  }
```

Um dies zu verhindern, kann der Ausdruck im SWITCH vorher in eine Variable gespeichert werden. Im SWITCH wird danach diese Variable abgefragt:

```
DEFINE_FUNCTION Test()
  {
  STACK_VAR INTEGER nIndex
  STACK_VAR INTEGER nSetup[6]
  STACK_VAR INTEGER nTemp
  ...
  nTemp = nSetup[nIndex]
  SWITCH(nTemp)
    {
    CASE 0:  { ...}
    CASE 1:  { ...}
    DEFAULT: { ...}
    }
  }
```

DIAGNOSTICS

Es wird empfohlen, der DIAGNOSTICS-Ausgabe in NetLinxStudio genügend Beachtung zu schenken und sie regelmässig zu konsultieren.