**먼저 LOOP JSUB GETC부터 보자. 이 명령이 실행되면 PC가 GETC로 이동한다. GETC에서 STX TEMP는 X 레지스터의 값(0)을 임시로 TEMP에 저장한다. 그후 LDX INCNT, LDA INCNT로 GETC루틴안에서 사용될 X 레지스터와 A레지스터를 INCNT(0)으로 초기화한다. 그 후 ADD #1, STA INCNT는 INCNT를 1증가시키는 과정이다.(A레지스터에 1을 저장하고 A레지스터의 값을 INCNT에 저장). LDCH INDEV,X는 INDEV에서 문자하나를 가져오라는 명령이다. INDEV C’0010030C1006’에서 맨 앞의 ‘0’을 가져와서 A레지스터에 저장한다. ASCII코드에서 ‘0’은 16진수 30에 대응하므로 A레지스터에는 30이 저장된다(메모리를 보면 ‘0’이 30으로 표현되어 저장되었음을 볼 수 있다). 그 후 A레지스터 값을 EOF(63)와 비교해서 같으면 EXADDR로 점프한다. 아직 같지 않으므로 점프하지 않는다. 그 후 COMP 48(아스키코드로 10진수 ‘0’) ,JLT GETC에서 A레지스터의 값과 48을 비교해서 A레지스터 값이 48보다 작으면 GETC로 점프한다. 현재는 A레지스터에 ‘0’(16진수 30)이 저장되어있으므로 작지 않다. 따라서 다음줄 SUB 48을 실행하면 ‘0’-‘0’으로 A레지스터에는 0이 저장되어야한다고 생각했었다. 하지만 주어진 툴로 A레지스터의 값을 보면 SUB 48을 실행해도 0이 되지 않는다. 그 이유를 생각해보니 SUB 48을 실행하면 주소 30(10진수 48 = 16진수 30)에 들어있는 값을 빼는 것이다. 하지만 주소 30에는 00이 들어있으므로 빼봣자 A레지스터 값은 그대로 인것이다. 목적코드가 1F0030인걸보면 주소 30에 들어있는 값을 이용해서 연산을 한다는걸 한번더 확인할 수 있다.(48을 #48로 바꾸어서 실행해본 결과 EXADDR에는 INDEV에 저장되어있던 ‘0010030C1006’,’000005FFFFF’가 저장됬다) 어쨋튼 다시 SUB 48이후로 돌아가자. COMP #10은 A레지스터의 값을 10진수 7를 비교한다. 그 결과가 만약 A레지스터값이 7보다 작다이면 JLT RETURN이 실행되어 RETURN으로 점프한다. 하지만 A레지스터는 30(16진수)이므로 JLT는 무시된다. 그 다음 SUB #7을 하면 A레지스터의 30(16진수)에서 7(10진수)를 뺀 결과(16진수 29)를 A레지스터에 저장한다. 그후 RETURN LDX TEMP로 TEMP에 저장되어있던 값(0)으로 X레지스터를 초기화한다. 그후 RSUB로 원래의 장소(JSUB가 실행됬던 곳)로 돌아온다. 이제 RMO A,S를 실행하면 A레지스터의 값(16진수 29)이 S레지스터에 저장된다. SHIFTL S,4로 S레지스터의 값이 왼쪽으로 SHIFT되어 290이된다.이제 JSUB GETC로 다시한번 GETC를 수행한다. GETC가 수행되면 A레지스터에는 29(16진수)가 저장된다. 그 후 ADDR S,A로 S+A(2B9)를 A레지스터에 저장한다. STCH EXADDR,X는 EXADDR의 0번째 인덱스에 A레지스터에서 1바이트만큼 저장된다. 따라서 EXADDR의 에는 앞의 2가 짤려서 B9가 저장된다. TIXR X,X는 X레지스터를 1증가시킨다.**

**이 과정을 반복하면 EXADDR에는 16진수로 B9 C9 BC CC C9 BF B9 B9 BE 2F 2F 2F E2 15 D8 D2 98 A8 BE DB B1 44 DD 60이 저장된다.**